Java™ magazine

By and for the Java community

WHAT'S NEW IN NASHORN 34 | INCUBATING HTTP/2 39 | LANGUAGE QUIZ 47

JULY/AUGUST 2017

11 NINE NEW FEATURES OF JDK 9

17 MIGRATING FROM JAVA 8 TO JAVA 9

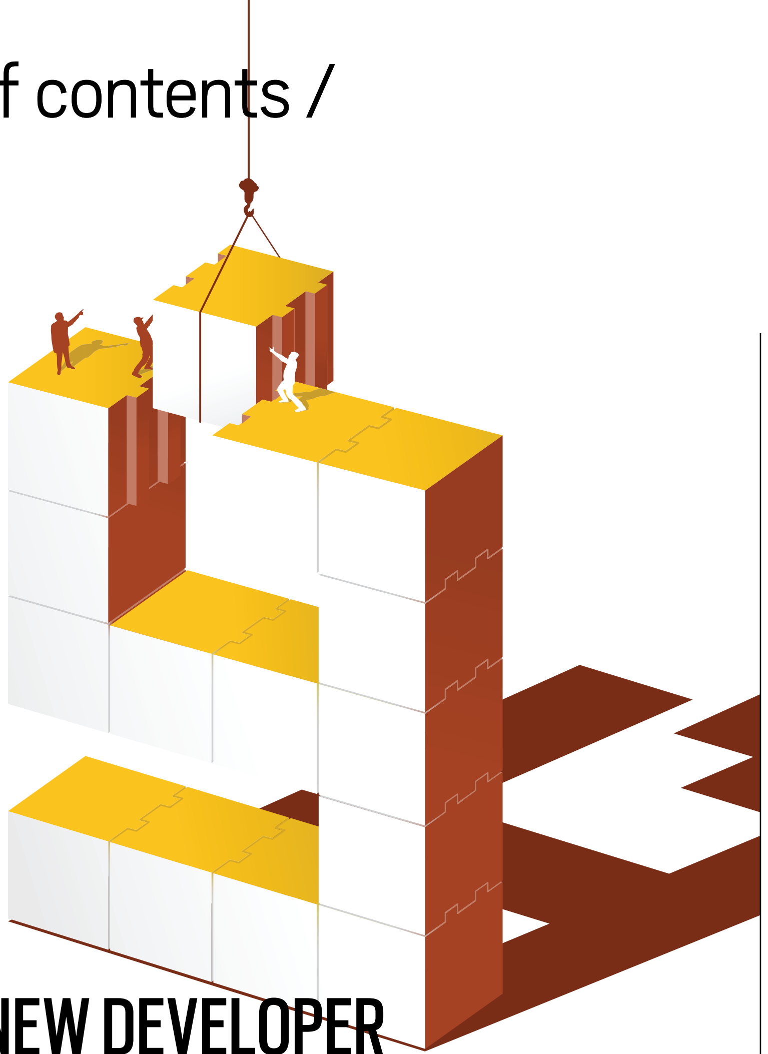
21 ENHANCEMENTS TO COLLECTIONS, STREAMS, AND ITERATORS

28 JSHELL: THE NEW REPL

ORACLE.COM/JAVAMAGAZINE

ORACLE®

//table of contents /



EDITORIAL
Editor in Chief
Andrew Binstock
Managing Editor
Claire Breen
Copy Editors
Karen Perkins, Jim Donahue
Technical Reviewer
Stephen Chin

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

PUBLISHING
Associate Publisher and Audience Development Director
Karin Kinnear +1.650.506.1985
Audience Development Manager
Jennifer Kurtz

ADVERTISING SALES
Sales Director
Tom Cometa
Account Manager
Mark Makinney
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.

ORACLE CODE

Register Now

Oracle Code

New One-Day, Free Event | 20 Cities Globally

Explore the Latest Developer Trends:

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

Find an event near you:
developer.oracle.com/code

Live for the Code

ORACLE®

The Noisy, Successful Undertaking of Collaborative Work

Delays in the delivery of major releases such as JDK 9 are common when community is valued.

This issue of *Java Magazine* is all about the release of Java 9. When I was originally laying out the editorial schedule, this special issue was assigned to a slot in last year's lineup. As many of you know, that intended delivery date was postponed until late July of this year, which is why you have this issue in your hands now. We anticipated—with excitement—that this issue and the Java 9 release would occur simultaneously.

But this happy scenario was thwarted by an unexpected development: a disagreement within the Java Community Process (JCP) that pushed the release back until late September of this year.

For some in the Java community, this postponement and the fact that it came after several previous delays triggered a great sense of frustration. Not for me. Sure, I would have loved to have the release and this magazine come out simultaneously, and certainly I would have liked to begin the transition to the new version with a final release of the JDK. Those sentiments, I believe, are universal in the community.

However, what lets me abide the delay without complaint is that it derives from collaboration with the community. As anyone who has served on industry committees can tell you, collaborative work is difficult, noisy, and supremely frustrating. There is nothing quite like it, except perhaps parenting. Yet despite the frustrations, there is a conviction held by all parties that it is better to

PHOTOGRAPH BY BOB ADLER/THE VERBATIM AGENCY

ORACLE

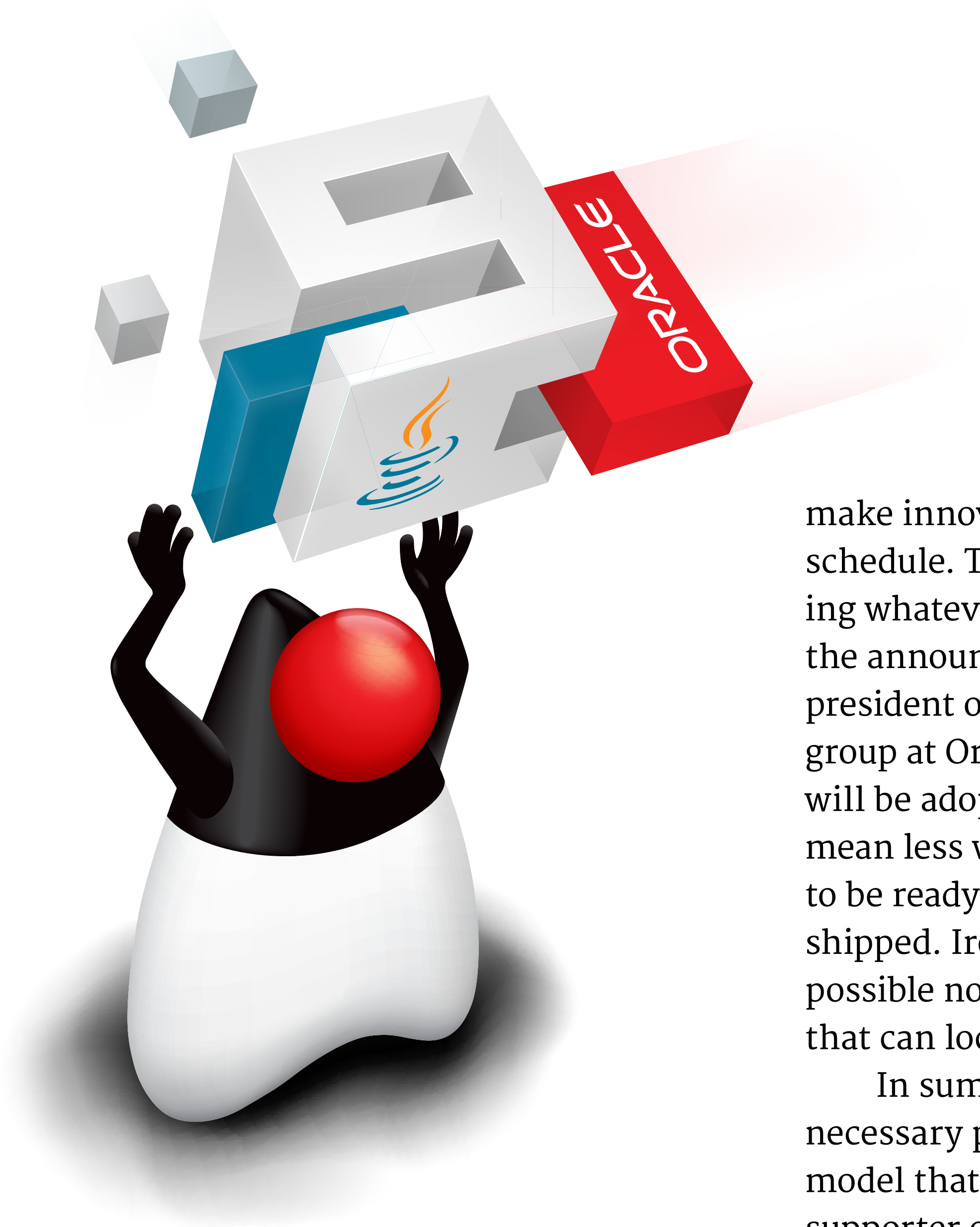
Level Up at Oracle Code

Step up to modern cloud development. At the Oracle Code roadshow, expert developers lead labs and sessions on PaaS, Java, mobile, and more.

Get on the list for event updates:
go.oracle.com/oraclecoderoadshow

developer.oracle.com

#developersrule



work through the hard parts of the collaboration than to abandon the effort. We are all better served by the collaboration than by insisting on our own preferences.

The understanding of collective benefit has long driven Java's history and is indisputably integral to its success. It can be easy to forget that the development of Java and the JVM is itself a collaborative effort. The development work is done as open source in public repositories, and the discussions about feature selection, code implementations, release dates, and other points of contention are held in public mailing lists. In fact, the recent decision regarding the new release date and the specific reasons for the delay were all posted and replied to on public forums.

Now, let me ask you, do you know of any other language whose principal corporate sponsor assigns more than 100 engineers to work on the language and yet defers on matters of release date to a community of partners? There are only three companies outside of Oracle that have made so large a commitment to language development: Apple, Google, and Microsoft. But none of them have adopted this open, consensual approach. (I'm not here deprecating those companies' work on their languages or their preferred approach, but rather I'm trying to underscore how unique Oracle's approach with Java is.)

I recognize that my acceptance of delay in the name of collaboration is borne of an insider's view of Java community operations. Many Java developers have no interest in the politics behind release dates and couldn't care less whether discussions are held in public or not; they just want the new technology to be released and so are frustrated by the succession of delays.

This perspective is driving an initiative to make innovations available on a predictable schedule. This new approach proposes publishing whatever technology is good to go when the announced date arrives. Georges Saab, vice president of development for the Java Platform group at Oracle, states that this new approach will be adopted in post–Java 9 releases. It will mean less waiting for a single central feature to be ready before lesser improvements are shipped. Ironically enough, this strategy is possible now *because* of the modularity in Java 9 that can localize the effects of a given change.

In sum, the delays of this release are the necessary product of the open, collaborative model that underlies Java's success. If you're a supporter of open source and open collaboration, then you surely recognize that such delays are a sacrifice that the process demands. In that sense, it is the antithesis of releasing by unilateral fiat. Nonetheless, I am excited that modularity makes possible future releases on a defined schedule while maintaining the commitment to high levels of collaboration.

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

MAY/JUNE 2017
Java magazine
By and for the Java community
Libraries
10 LOMBOK: ANNOTATIONS FOR CLEANER CODE
16 JDEFERRED'S ASYNC EVENT MANAGEMENT
34 DATABASE ACCESS WITH STREAMS
28 BEST PRACTICES FOR LIBRARY DESIGN
ORACLE.COM/JAVAMAGAZINE
ORACLE

MAY/JUNE 2017

Exception Swallowing in Project Lombok

In your excellent May/June 2017 issue, I want to comment on your article "Project Lombok: Clean, Concise Code," in which Josh Juneau presents a concise guide to using Lombok. On page 14, Juneau states, "The `@SneakyThrows` annotation can be placed on a method to essentially 'swallow' the exceptions." I think this sentence is somewhat misleading. To me the term *swallow* means to catch the exception, do nothing with it, and continue execution of the code.

However, this is far from what Lombok is doing. From the Lombok site, "Lombok will not ignore, wrap, replace, or otherwise modify the thrown checked exception; it simply fakes out the compiler." This means we will still get an exception bubbling through should one occur, but we don't need to explicitly handle it in calling code, much like runtime exceptions.

—Syed Asghar

Author Josh Juneau responds: "Thanks for sending me this useful feedback. I am glad that you found the article useful. I can certainly see why you find the term swallow *misleading with respect to the way in which the* `@SneakyThrows` *annotation handles checked exceptions. I do agree, after reassessing, that this was not an ideal choice of words. The Lombok* `@SneakyThrows` *annotation allows you to omit a try-catch clause and throw a checked exception—it does not silently swallow it. Perhaps the term* hides *would be more precise, as Lombok hides the requirement to code a try-catch clause by allowing you to throw a checked exception without it. As stated in the Lombok Javadoc,* `@SneakyThrows` *does not silently swallow, wrap into RuntimeException, or otherwise modify any exceptions of the listed checked exception types. The JVM does not check for the consistency of the checked exception system; javac does, and this annotation lets you opt out of its mechanism."*

Downloading Java Magazine

Two readers inquired why they could not see the download icon for *Java Magazine* when they accessed the new page. Downloading the PDF version of the magazine is a subscriber privilege. So you must log in to access the PDF. As we have increasingly opened the magazine (making articles available without login), you might be reading the issue without logging in. To force a login, go to the quiz (which is also a subscriber-only benefit) and you'll be asked to log in. When you do so from a laptop or desktop, you'll see the download icon in the right margin.

Contact Us

We welcome comments, suggestions, grumbles, kudos, article proposals, and chocolate chip cookies. All but the last two might be edited for publication. If your note is private, indicate this in your message. Write to us at javamag_us@oracle.com. For other ways to reach us, see the last page of this issue.

PHOTOGRAPH BY BOB ADLER/THE VERBATIM AGENCY

JVM Language Summit

JULY 31–AUGUST 2
SANTA CLARA, CALIFORNIA
The JVM Language Summit is an open technical collaboration among language designers, compiler writers, tool builders, runtime engineers, and architects who target the JVM. The schedule consists of a single track of traditional presentations (about six each day) interspersed with workshop discussion groups. Each registrant is invited to suggest a few topics of interest for the workshops.

JCrete

JULY 16–21
KOLYMBARI, GREEECE
This loosely structured "unconference" involves morning sessions discussing all things Java, combined with afternoons spent socializing, touring, and enjoying the local scene. There is also a JCrete4Kids component for introducing youngsters to programming and Java. Attendees often bring their families.

ÜberConf

JULY 18–21
DENVER, COLORADO
ÜberConf 2017 will be held at the Westin Westminster in downtown Denver. Topics include Java 8, microservice architectures, Docker, cloud, security, Scala, Groovy, Spring, Android, iOS, NoSQL, and much more.

JavaZone 2017

SEPTEMBER 12, WORKSHOPS
SEPTEMBER 13–14, CONFERENCE
OSLO, NORWAY
JavaZone is a conference for Java developers created by the Norwegian Java User Group, javaBin. The conference has existed since 2001 and now consists of around 200 speakers and 7 parallel tracks over 2 days, plus an additional day of workshops beforehand. You will be joined by approximately 3,000 of your fellow Java developers. Included in the ticket price is a membership in javaBin.

Strange Loop

SEPTEMBER 28–30
ST. LOUIS, MISSOURI
Strange Loop is a multidisciplinary conference that brings together the developers and thinkers building tomorrow's technology in fields such as emerging languages, alternative databases, concurrency, distributed systems, security, and the web. Talks are, in general, code-heavy, not process-oriented. A preconference day on September 28 is optional and not included in the conference rate.

NFJS Boston

SEPTEMBER 29–OCTOBER 1
BOSTON, MASSACHUSETTS
Since 2001, the No Fluff Just Stuff (NFJS) Software Symposium Tour has delivered more than 450 events with more than 70,000 attendees. This event in Boston

PHOTOGRAPH BY MASSMATT/FLICKR

covers the latest trends within the Java and JVM ecosystem, DevOps, and agile development environments.

JavaOne

OCTOBER 1–5

SAN FRANCISCO, CALIFORNIA

Whether you are a seasoned coder or a new Java programmer, JavaOne is the ultimate source of technical information and learning about Java. For five days, Java developers gather from around the world to talk about upcoming releases of Java SE, Java EE, and JavaFX; JVM languages; new development tools; insights into recent trends in programming; and tutorials on numerous related Java and JVM topics.

JAX London

OCTOBER 9–12

LONDON, ENGLAND

JAX London is a four-day conference for cutting-edge software engineers and enterprise-level professionals, bringing together the world's leading innovators in the fields of Java, microservices, continuous delivery, and DevOps. Conference sessions, keynotes, and expo happen October 10–11. Hands-on workshops take place the day preceding and the day following the main conference.

Oracle Code Events

Oracle Code is a free event for developers to learn about the latest development technologies, practices, and trends, including containers, microservices and API applications, DevOps, databases, open source, development tools and low-code platforms, machine learning, AI, and chatbots. In addition, Oracle Code includes educational sessions for developing software in Java, Node.js, and other programming languages and frameworks using Oracle Database, MySQL, and NoSQL databases.

ASIA PACIFIC

JULY 18, Sydney, Australia

AUGUST 10, Bangalore, India

AUGUST 30, Seoul, South Korea

O'Reilly Software Architecture Conference

OCTOBER 16–18, CONFERENCE AND TUTORIALS

OCTOBER 18–19, TRAINING

LONDON, ENGLAND

For four days, expert practitioners share new techniques and approaches, proven best practices, and exceptional technical skills. At this conference, you'll hear about the best tools to use and why, and the effect they can have on your work. You'll learn strategies for meeting your company's business goals, developing leadership skills, and making the conceptual jump from software developer to architect.

KotlinConf
NOVEMBER 2–3
SAN FRANCISCO, CALIFORNIA
KotlinConf is a JetBrains event that provides two days of content from Kotlin creators and community members.

Devoxx
NOVEMBER 6–10
ANTWERP, BELGIUM
The largest gathering of Java developers in Europe takes place again this year in Antwerp. Dozens of expert speakers deliver hundreds of presentations on Java and the JVM. Tracks include server-side Java, cloud, big data, and extensive coverage of Java 9.

W-JAX
NOVEMBER 6–10
MUNICH, GERMANY
W-JAX is a conference dedicated to cutting-edge Java and web development, software architecture, and innovative infrastructures. Experts share their professional experiences in sessions and workshops. This year's focus is on Java core and enterprise technologies, the Spring ecosystem, JavaScript, continuous delivery, and DevOps.

QCon San Francisco
NOVEMBER 13–15, CONFERENCE
NOVEMBER 16–17, WORKSHOPS
SAN FRANCISCO, CALIFORNIA
Although the content has not yet been announced, recent QCon conferences have offered several Java tracks along with tracks related to web development, DevOps, cloud computing, 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.

ORACLE

JavaOne™

JavaYourNext
(Breakthrough)

October 1–5, 2017
San Francisco, California

#JavaOne

REGISTER NOW

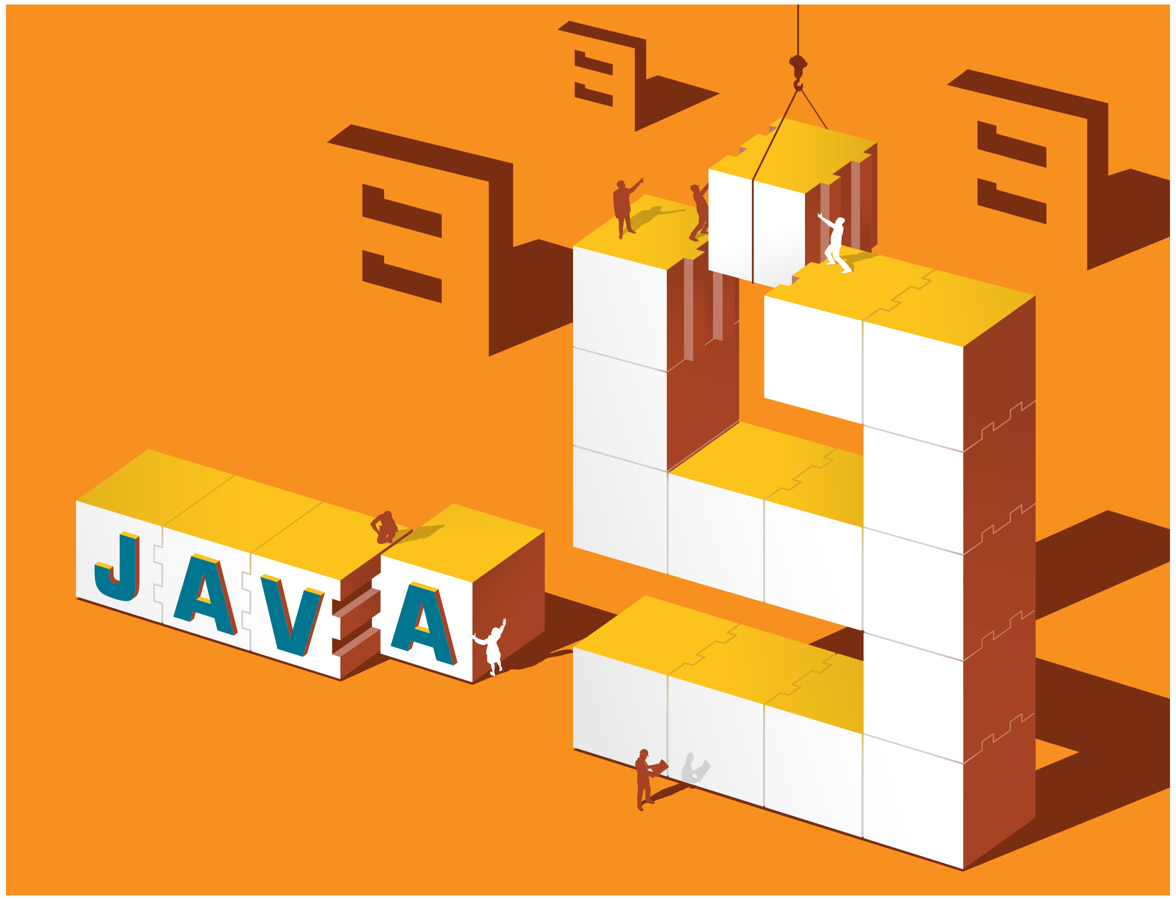
Inside JDK 9

This single-topic issue of *Java Magazine* focuses on the benefits of the new JDK 9 release other than the Java Platform Module System (JPMS). The modularity feature, often touted as the central part of this release, was not yet in final approved form when we went to press. So, rather than provide early information that might be wrong later, we've chosen to focus on the other parts of JDK 9, which have been formalized and finished and will ship whenever modules are officially approved. At present, that shipping date is expected to be in late September. The reasons for this delay are discussed in the editorial (page 3). Shortly after that date, *Java Magazine* will dedicate a second issue to Java 9, with a deep focus on the new modular architecture and how best to use it.

As the articles in this issue demonstrate, there is a lot of goodness in Java 9 outside of modules. The language and platform teams have created dozens of convenient new features that make Java programming more succinct and enjoyable. Simon Ritter's article (page 11) provides an overview of many of these useful additions. His work is complemented by an in-depth examination (page 21) of the new features in Collections, Streams, and iterators. Also, Trisha Gee explains (page 17) how to compile and run Java 8 code on Java 9, even if you're not using modules.

An alternative way to run Java 9 code is with JShell, which is a new read-evaluate-print loop (REPL) bundled with this release. Our introduction to JShell (page 28) shows the basics, while our article on HTTP/2 (page 39) provides additional examples of JShell usage. The HTTP/2 technology, which facilitates network programming, is part of a new incubator system introduced in Java 9 that presents developers with technologies that are likely to be bundled in future releases. If you regularly use HTTP, take a long look at this article.

In addition to these articles, we also have our usual language quiz, events calendar, and letters to the editor. Enjoy, and let us know if there are other Java 9 topics you'd like us to cover in the future.



ART BY I-HUA CHEN

SIMON RITTER

Nine New Developer Features in JDK 9

There's a lot more to this release than modules.

The big new feature in JDK 9 is the Java Platform Module System coupled with the introduction of the modular JDK. However, there are plenty of other new features in JDK 9, and in this article, I focus on nine that are of particular interest to developers. Where applicable, I've included the relevant JDK Enhancement Proposal (JEP) number so you can find more information.

Factory Methods for Collections (JEP 269)

Collections provide a well understood way for you to gather together groups (I was going to say *sets*, but that could be a bit misleading) of data items in your applications and then manipulate the data in a variety of useful ways.

At the top level, there are interfaces that represent the abstract concepts of a List, Set, and Map.

The problem, until now, has been that Java doesn't provide a simple way to create a collection with predefined data. If you want a collection to be structurally immutable (that is, you can't add, delete, or change references to elements), you need to do more work.

Let's look at a simple example using JDK 8:

```
List<Point> myList = new ArrayList<>();
myList.add(new Point(1, 1));
myList.add(new Point(2, 2));
myList.add(new Point(3, 3));
myList.add(new Point(4, 4));
myList = Collections.unmodifiableList(myList);
```

It's not terrible, admittedly, but to create an immutable list of four `Point`s required six lines of code. JDK 9 addresses this through factory methods for collections.

This feature makes use of a change introduced in JDK 8 that enabled static methods to be included in interfaces. That change means that you can add the necessary methods at the top-level interfaces (`Set`, `List`, and `Map`) rather than having to add them to a large group of classes that implement those interfaces.

Let's rewrite our example using JDK 9:

```
List<Point> list =
    List.of(new Point(1, 1), new Point(2, 2),
    new Point(3, 3), new Point(4, 4));
```

The code is now much simpler.

The rules that apply to the use of the different collections also apply (as you would expect) when using these factory methods. So, you cannot pass duplicate arguments when you create a `Set`, nor can you pass duplicate keys when you create a `Map`. A `null` value cannot be used as a value for any collection factory method. The Javadoc documentation provides full descriptions of how the methods may be called. [Collections

PHOTOGRAPH BY BOB ADLER/
THE VERBATIM AGENCY

are discussed further in the article "Java 9 Core Library Updates: Collections and Streams" (page 21). —*Ed.*]

Optional Class Enhancements

The Optional class was introduced in JDK 8 to reduce the number of places where a NullPointerException could be generated by code (and it was frequently used to make the Stream API more robust).

JDK 9 adds four new methods to Optional:

- ifPresent(Consumer action): If there is a value present, perform the action using the value.
- ifPresentOrElse(Consumer action, Runnable emptyAction): Similar to ifPresent, but if there is no value, it executes the emptyAction.
- or(Supplier supplier): This method is useful when you want to ensure that you always have an Optional. The or() method returns the same Optional if a value is present; otherwise, it returns a new Optional created by the supplier.
- stream(): Returns a stream of zero or one elements, depending on whether there is a value.

Stream API Enhancements

It's always useful to be able to create a stream source from a collection of data, and JDK 8 provided several methods to do this outside the Collections API (BufferedReader.lines(), for example). Several new sources are being added in JDK 9, such as java.util.Scanner and java.util.regex.Matcher.

JDK 9 adds four methods to the Stream interface.

First, there are two related methods: takeWhile(Predicate) and dropWhile(Predicate). These methods are complementary to the existing limit() and skip() methods, but they use a Predicate rather than a fixed integer value. The takeWhile() method continues to take elements from the input stream and pass them to the output stream until the test() method of the Predicate returns true. The dropWhile() method does the opposite; it *drops* elements from the input stream until the test() method of the Predicate returns true. All remaining elements of the input stream are then passed to the output stream.

JDK 9 includes a new read-eval-print loop (REPL) command-line tool, jshell, that allows you to develop and test Java code interactively.

Be careful when using either of these methods when you have an unordered stream. Because the predicate needs to be satisfied only once to change the state of elements being passed to the output, you might get elements in the stream that you don't expect, or you might miss ones you thought you would get.

The third new method is ofNullable(T t), which returns a stream of zero or one elements, depending on whether the value passed is null. This can be very useful to eliminate a null check before constructing a stream, and it is similar in a sense to the new stream() method in the Optional class discussed in the previous section.

The last new stream method is a new version of the static iterate() method. The version in JDK 8 took one parameter as the seed and created an infinite stream as output. JDK 9 adds an overloaded method that takes three parameters, which effectively gives you the ability to replicate the standard for loop syntax as a stream. For example, Stream.iterate(0, i -> i < 5, i -> i + 1) gives you a stream of integers from 0 to 4.

Read-Eval-Print Loop: jshell (JEP 222)

JDK 9 includes a new read-eval-print loop (REPL) command-line tool, jshell, that allows you to develop and test Java code interactively, unlike when you use an IDE, where code must be edited, compiled, and then run. Developers can quickly prototype sections of code as jshell continually reads user input, evaluates it, and prints either the value of the input or a description of the state change the input caused.

To make the tool easier to use, it includes features such as an editable history, tab completion, automatic addition of terminal semicolons when needed, and configurable predefined imports and definitions. It is also possible to declare variables whose type is defined after the declaration (but this is still *not* dynamic typing: once the type is set, it can't be changed).

The introduction of private methods in interfaces will allow common code to be extracted to methods that will remain encapsulated within the interface.

There is even a module, `jdk.jshell`, that can be used if you want to build your own "snippet" evaluation tool.

One of the most significant reasons for including a REPL in the JDK is to make it easier to use Java as a teaching language, lowering the amount of boilerplate code new developers need to write before they get results.

Concurrency Updates (JEP 266)

Java 5 introduced concurrency utilities to simplify writing Java code that has multiple cooperating threads. Subsequent releases have improved these features by adding things such as the fork/join framework in JDK 7 and, most recently, parallel streams in JDK 8. Now, JDK 9 provides enhancements for concurrency in two areas.

The first is a reactive streams publish-subscribe framework. Processing streams of elements in an asynchronous fashion can lead to problems when the rate at which elements are submitted to the processing code rises sharply. If the processor is unable to handle the load, the element supplier can be blocked or a buffer overflow situation can occur.

Reactive streams use a publish-subscribe model in which the stream processor (the subscriber) subscribes to the supplier of elements (the publisher). By doing this, the supplier knows how many elements it can pass to the processor at any time. If the supplier needs to send more than this number, it can use techniques such as local buffering or it can use an alternative processor (this is a typical microservice approach that delivers horizontal scalability by spinning up new service instances to handle increased load).

JDK 9 includes a new class, `Flow`, that encloses several interfaces: `Flow.Processor`, `Flow.Subscriber`, `Flow.Publisher`, and `Flow.Subscription`. A `Subscription` is used to link a `Publisher` with a `Subscriber`.

The `Subscription` interface contains two methods:

- `cancel()`: This method causes the `Subscriber` to stop receiving messages (eventually).
- `request(long n)`: Add *n* elements to the number that the `Subscriber` is able to process. This method can be called repeatedly as the `Subscriber` processes elements and is ready to process more.

By implementing the `Processor` interface, a class can act as both a publisher and a subscriber, thereby acting as an intermediate operation in the stream.

The second new concurrency feature in JDK 9 consists of enhancements to the `CompletableFuture` class, which was introduced in JDK 8. Several new methods relate to adding time-based functionality. These enhancements enable the use of time-outs via methods such as `completeOnTimeout()`, `delayedExecutor()`, and `orTimeout()`. Other new methods include `failedFuture()` and `newIncompleteFuture()`.

Milling Project Coin (JEP 213)

One of the most significant changes in JDK 7 was Project Coin, which was a set of small language changes to smooth out some of the common tasks developers undertake repeatedly in Java. The project included things such as the ability to use Strings as constants in switch statements. JDK 9 builds on this and adds a few more small changes to the language syntax of Java:

- Effectively final variables can be used in try-with-resources without being reassigned. Currently, each resource that is used must be assigned to a new variable, even if that resource has already been defined in the method. With this change, it is now possible to use an existing variable (as long as it is effectively final) without reassigning it. Here is an example of the change:

  ```
  try (Resource r = alreadyDefinedResource) ...
  ```

  In JDK 9, it becomes this:

  ```
  try (alreadyDefinedResource) ...
  ```

- Private methods can be used in interfaces. Interfaces changed in a big way in JDK 8. First, there was the introduction of default methods, which allowed new methods to be added to existing interfaces without forcing a break in backward compatibility. Because this meant that behavior could be included in an interface (giving Java multiple-inheritance of behavior for the first time), it was also logical to include static methods. In JDK 9, the introduction of private methods in interfaces will allow common code to be extracted to methods that will remain encapsulated within the interface.
- The diamond operator can be used with anonymous classes. One of the things included in JDK 8 and now JDK 9 is better type inference. Being able to use lambda expressions without explicitly specifying the types of parameters is a good example of this. This ability required a substantial rewrite of how the Java compiler processes type inference, and the diamond operator change in JDK 9 makes use of that compiler change. The problem, as it existed before, was that using the diamond operator with an anonymous class could result in a type being inferred that is not denotable. A type that is not denotable can be represented by the compiler, but it cannot be expressed to the JVM using the class signature attribute. In JDK 9, it is now possible to use the diamond operator with an anonymous class as long as the inferred type *is* denotable, for example:

  ```
  List<String> myList = new ArrayList<>() {
    // Overridden methods
  };
  ```

The ProcessHandle interface identifies and provides control of native processes.

- A single underscore will no longer be valid as an identifier. I often ask, during Java 9 presentations, if anyone has ever used a single underscore as a variable name and, thankfully, few people have. The reason for doing this is that in a later release of the JDK (presumably JDK 10), it will be valid to use a single underscore as a variable name *only* in lambda expressions. This makes sense. You could use a single underscore if your lambda expression takes only one argument and you don't use the argument in the body of the lambda, for example:

  ```
  _ -> getX()
  ```

  (For those of you who like the idea of underscores as variable names, you can still use two or more as an identifier.)
- Extended use of the @SafeVarargs annotation is allowed. Currently, this annotation can be used only on constructors, static methods, and final methods, none of which can be overridden. Because a private method cannot be overridden by a subclass, it is logical also to enable private methods to use this annotation.

Spin-Wait Hints (JEP 285)

This is a small change because it adds only a single method, but its addition to the Thread class is significant. The other interesting aspect of this feature is that it is the first JEP to be included that was proposed by a company other than Oracle (in this case, Azul Systems).

The onSpinWait() method provides a hint to the JVM that the thread is currently in a processor spin loop. If the JVM and hardware platform support optimizations when spinning, such hints can be used; otherwise, the call is ignored. Typical optimizations include reducing thread-to-thread latencies and reducing power consumption.

Variable Handles (JEP 193)

One of the most significant changes in JDK 9 caused by modularizing the core JDK libraries is the encapsulation of the internal APIs by default. Probably the most well known of these is sun.misc.Unsafe. To make some of the functionality of this private API available through a public API, JDK 9 introduces the VarHandle class.

Variables are used in Java all the time. They are implicit pointers to areas of memory that hold values. A variable handle is a typed reference to a variable (so for those who've used C and C++, it's effectively a pointer to a pointer).

In order to get a reference to a VarHandle, you use the MethodHandle.Lookup class that has been extended in JDK 9. You can retrieve references to either static or nonstatic variables, as required, as well as to arrays.

Once you have a VarHandle, you can execute low-level memory ordering operations on the variable it references. This capability allows you to perform atomic operations, such as compare-and-set operations, but without the performance overhead that is associated with using the equivalent classes and methods of the java.util.concurrent.atomic package.

With features such as these, you should consider migrating your development to this release ASAP.

You can also use a VarHandle to "fence" operations, giving you fine-grained control over the memory ordering of operations such as read and write or store and load. This can be useful in situations where you don't want the Java compiler or JVM to reorder operations, which is often done to optimize the performance of code paths.

Process API Updates (JEP 102)

JDK 9 contains enhancements to the Process and ProcessBuilder classes and introduces a new ProcessHandle interface. ProcessBuilder now includes the method startAsPipeline(). As the name suggests, this method constructs a UNIX-style pipeline of processes using a list of ProcessBuilders. As each process is started, its standard output is connected to the standard input of the next process.

Here's a simple example that pipes the output of the UNIX/Linux ls command on /tmp through wc -l to get a count of the number of files in the tmp directory:

```
ProcessBuilder ls = new ProcessBuilder()
        .command("ls")
        .directory(Paths.get("/tmp").toFile());
ProcessBuilder wc = new ProcessBuilder()
        .command("wc", "-l")
        .redirectOutput(Redirect.INHERIT);
List<Process> lsPipeWc = ProcessBuilder
        .startPipeline(asList(ls, wc));
```

The ProcessHandle interface identifies and provides control of native processes. It provides methods to retrieve information about a process, such as the process ID as well as any child and descendant processes. Also, there is a static method that returns an Optional<ProcessHandle> for a given process ID.

The subinterface `ProcessHandle.Info` provides a range of process-specific details, such as the command line used to start the process, when the process started, and the identity of the user that started the process.

The `Process` class has seven new methods that provide access to more information about the native process. Some of these overlap with those available through the `ProcessHandle`:

- `children()` and `descendants()`: Lists the process's children or dependents, respectively
- `getPid()`: Returns the ID of the given process
- `info()`: Returns a snapshot of information as a `ProcessHandle.Info` instance
- `onExit()`: Is a `CompletableFuture` that can be used to perform tasks when the process terminates
- `supportsNormalTermination()`: Determines whether `destroy()` terminates the process normally
- `toHandle()`: Returns the `ProcessHandle` of this process

Conclusion

As you can see, there are many useful and powerful new features included in JDK 9 that are particularly targeted at developers. With features such as these, you might want to consider migrating your development to this release as soon as you can. `</article>`

Simon Ritter (@speakjava) is the deputy CTO of Azul Systems. He has been in the IT business since 1984 and holds a BS in physics from Brunel University in the UK. He joined Sun Microsystems in 1996 and spent time working in both Java development and consultancy. Ritter has been presenting Java technologies to developers since 1999, focusing on the core Java platform as well as client and embedded applications.

FEATURED JDK ENHANCEMENT PROPOSAL

JEP 241: Veteran Tool jhat Reaches Retirement

The release of JDK 9 contains lots of goodies, as this issue of *Java Magazine* has reported in detail. However, due to the approval of JDK Enhancement Proposal 241 (JEP 241), the jhat heap analysis tool will no longer be bundled. jhat originally made its way into the JDK with the release of Java 6. It came from a project called HAT and was a part of the now defunct javatools projects before its incorporation into the JDK.

Despite its inclusion in recent releases of the JDK, it was always categorized as an experimental tool and labeled as such in all official documentation. Over the years, other heap profilers have surpassed it, and so jhat has been in an extended decline with little maintenance being accorded to it.

If you're still using jhat, you should consider other tools. Two worthy options are VisualVM and, at a programmatic level, the Java HotSpot Serviceability Agent API. The latter is thoroughly explored and documented in the book *Java Performance Companion*, which was reviewed in the September/October 2016 issue.

Standalone commercial offerings are also available, notably JProfiler and YourKit. In addition, leading IDEs today bundle tools that enable you to examine the JVM heap in developer-friendly ways.

One way or the other, you should be able to find good substitutes for jhat.

TRISHA GEE

PHOTOGRAPH BY JOHN BLYTHE

Migrating from Java 8 to Java 9

Step-by-step work can lead to the adoption of all or some of the features even without using modules.

This article shows the steps you take to run Java 8 code and use some of the new features of Java 9 without migrating to modules. I principally examine what you need to do to get the code compiling and running.

Getting Started

First, download and install the latest version of JDK 9. At the time of this writing, it is still an Early Access release and can be found here (in this article, I'm using build 9-ea+166).

Until the impact of Java 9 on your system is understood, you probably don't want it to be the default Java version. So, instead of updating $JAVA_HOME to point to the new installation, you might want to create a new environment variable, $JAVA9_HOME, instead. I'll be using this approach throughout this article.

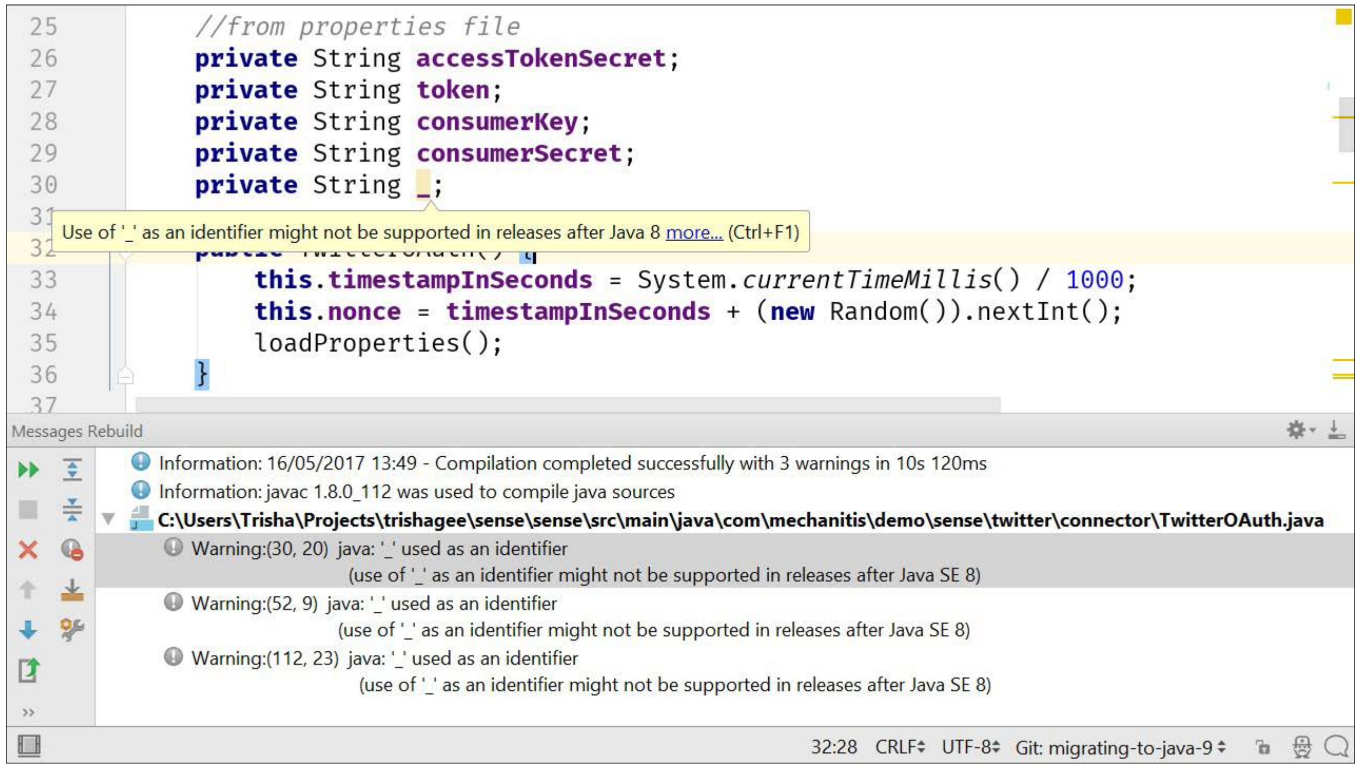
You should also look at this detailed migration guide to get a feel for what steps might be needed.

Before Jumping In

You can determine the amount of effort that might be required to migrate to Java 9 without even downloading or using it. There are several tools available in Java 8 that will give you appropriate hints.

Compiler warnings. First, of course, you can look at any compiler warnings, because some of those will tell you about potential problems (see **Figure 1**).

I'm using my IDE, IntelliJ IDEA, to compile the code and to show me the warnings, but what you see in the messages box in **Figure 1** is the same as what you'd see using javac from the command line. Other IDEs, such as NetBeans and Eclipse, will have similar functionality. In **Figure 1**, you can see that the code has a field with a single underscore as a name. This single-underscore name has been removed as a legal identi-



```
    //from properties file
    private String accessTokenSecret;
    private String token;
    private String consumerKey;
    private String consumerSecret;
    private String _;

    public TwitterOAuth() {
        this.timestampInSeconds = System.currentTimeMillis() / 1000;
        this.nonce = timestampInSeconds + (new Random()).nextInt();
        loadProperties();
    }
```

```
Use of '_' as an identifier might not be supported in releases after Java 8 more... (Ctrl+F1)
```

```
Messages Rebuild
Information: 16/05/2017 13:49 - Compilation completed successfully with 3 warnings in 10s 120ms
Information: javac 1.8.0_112 was used to compile java sources
C:\Users\Trisha\Projects\trishagee\sense\sense\src\main\java\com\mechanitis\demo\sense\twitter\connector\TwitterOAuth.java
    Warning:(30, 20)  java: '_' used as an identifier
                      (use of '_' as an identifier might not be supported in releases after Java SE 8)
    Warning:(52, 9)  java: '_' used as an identifier
                      (use of '_' as an identifier might not be supported in releases after Java SE 8)
    Warning:(112, 23)  java: '_' used as an identifier
                      (use of '_' as an identifier might not be supported in releases after Java SE 8)

32:28  CRLF  UTF-8  Git: migrating-to-java-9
```

Figure 1. Example compiler warning about a potential problem

fier name in Java 9 as part of JDK Enhancement Proposal (JEP) 213. This is because underscores might be used in a later version of Java to make working with lambda expressions even easier. If you come across identifier names that consist of only a single underscore, you'll need to rename them. Note that names containing underscores, for example `_fieldName`, are still valid; the problem is identifier names that contain *only* an underscore and nothing else.

Identify problematic dependencies with jdeps. One of the goals of this release, and surely the most famous Java 9 feature, is to allow the JDK developers to hide internal implementation details. In the past, developers were expected to follow guidelines to reduce the risk of depending on an unsupported class in the JDK; but it's much better if a language, framework, or library can hide things that developers shouldn't use. To make sure your code isn't using internal APIs that will be hidden in Java 9, you can use a tool called jdeps.

Figure 2 shows where my compiled classes live in my project.

Many tutorials suggest you use jdeps with a JAR file, but you can also use it with your class files. For my application, I navigate to the location of my class files via the command line (Figure 2 shows that this is `%PROJECT%\%MODULE%\build\classes`) and from there, I run jdeps with the `-jdkinternals` flag. I'm using Windows, so my command line is the following, where `main` is the name of the directory containing my classes:

```
> %JAVA_HOME%\bin\jdeps -jdkinternals main
```

When I run this command, I get the output in Figure 3.

These messages indicate that my `TwitterOAuth` class is using the internal classes `sun.misc.BASE64Encoder` and `sun.misc.Unsafe`. The output suggests a replacement for the encoder, `java.util.Base64`, which was a new class in Java 8. `unsafe` is a special case, because there is no suitable replacement for all of its functionality yet.

JEP 260 talks about which internal APIs are going away and which will remain accessible. Java team member Mark Reinhold provided the following summary in an email to one of the principal JDK mailing lists to explain how each API will be handled by the Java team:

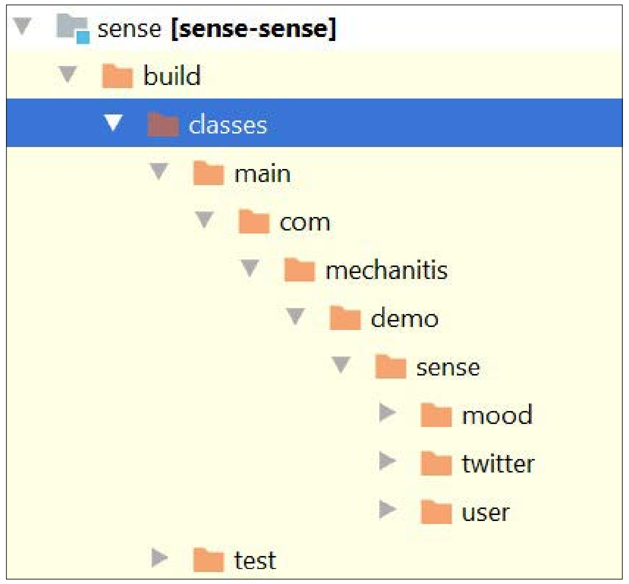
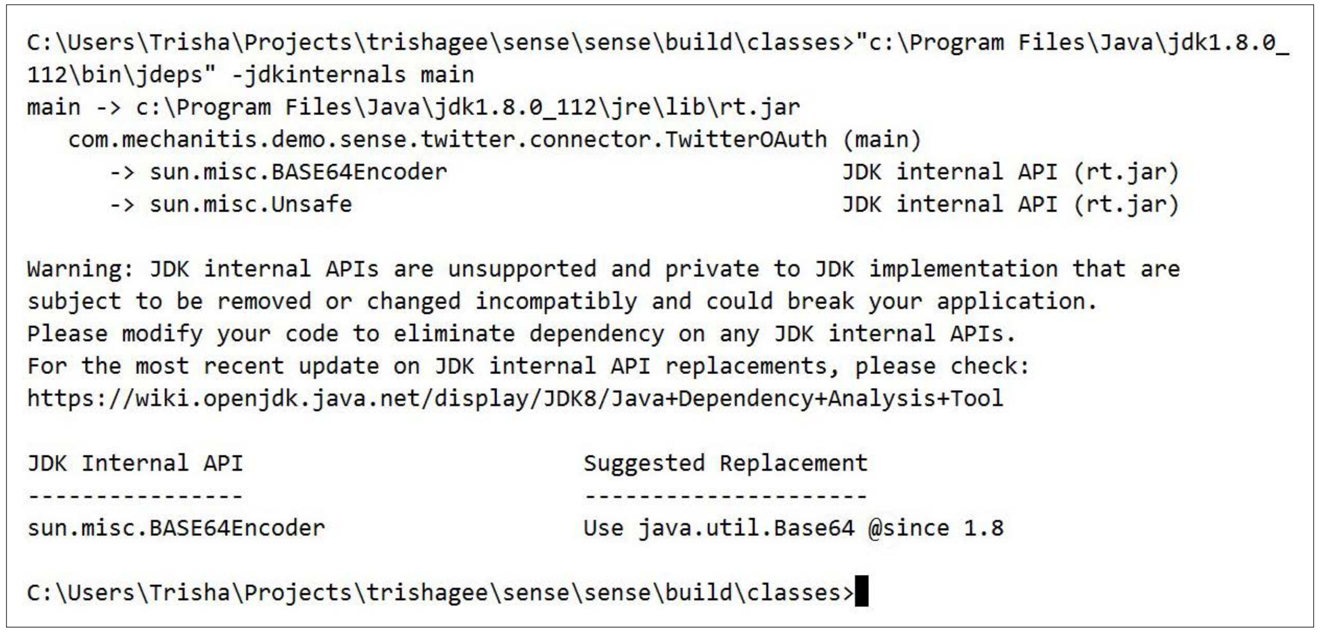


Figure 2. Location of compiled classes in the project



```
C:\Users\Trisha\Projects\trishagee\sense\sense\build\classes>"c:\Program Files\Java\jdk1.8.0_
112\bin\jdeps" -jdkinternals main
main -> c:\Program Files\Java\jdk1.8.0_112\jre\lib\rt.jar
   com.mechanitis.demo.sense.twitter.connector.TwitterOAuth (main)
      -> sun.misc.BASE64Encoder                           JDK internal API (rt.jar)
      -> sun.misc.Unsafe                                  JDK internal API (rt.jar)

Warning: JDK internal APIs are unsupported and private to JDK implementation that are
subject to be removed or changed incompatibly and could break your application.
Please modify your code to eliminate dependency on any JDK internal APIs.
For the most recent update on JDK internal API replacements, please check:
https://wiki.openjdk.java.net/display/JDK8/Java+Dependency+Analysis+Tool

JDK Internal API                         Suggested Replacement
----------------                         ---------------------
sun.misc.BASE64Encoder                   Use java.util.Base64 @since 1.8

C:\Users\Trisha\Projects\trishagee\sense\sense\build\classes>
```

Figure 3. Output from running jdeps

- If it has a supported replacement in JDK 8, then we will encapsulate it in JDK 9.
- If it does not have a supported replacement in JDK 8, then we will not encapsulate it in JDK 9, so that it remains accessible to outside code.
- If it has a supported replacement in JDK 9, then we will deprecate it in JDK 9 and encapsulate it, or possibly even remove it, in JDK 10.

Compiler warnings and jdeps can show you areas in your code that might be problematic in Java 9 and even suggest solutions. These problems can be fixed while you're still running with an earlier version of Java (provided the suggested replacement classes are available in that version), and doing so will ease your transition to Java 9.

Running with Java 9

The migration guide I mentioned earlier suggests running your application with Java 9 before recompiling or making any other changes. One place you might want to do this testing is in a continuous integration (CI) environment: you can use the artifacts built with your current version of Java but run the application and its tests with Java 9.

However, at the time of this writing, many of the common CI servers and some build tools, such as Gradle, do not yet fully support Java 9. If your project uses Maven, a few CI servers, such as TeamCity, will be able to compile or run your code with Java 9. But because the JDK and JRE were restructured as part of Java 9, some tools will need to be updated to work correctly.

Java 9 contains several changes that might affect your application; some might not cause compilation errors but instead will demonstrate different behavior.

You should find out which CI server and build tool your team uses, and determine whether they and the related parts of the toolchain presently support running with Java 9.

Compiling with Java 9

If you followed my previous advice, compiling with Java 9 should be simple. In my code, if I compile with Java 9 without fixing those errors identified in **Figure 1** and **Figure 3**, I will get the two errors I was expecting: one error stating I cannot use underscore as a field name and one that shows I can no longer access `BASE64Encoder`. Both of these are straightforward to fix. For the first, I rename the field. For the second, I import `java.util.Base64` and replace the following old line of code:

```
String encodedString =
    new BASE64Encoder().encode(bytes);
```

with this:

```
String encodedString =
    Base64.getEncoder().encodeToString(bytes);
```

The use of `sun.misc.Unsafe` won't give me an error or warning (at this point), because it's still accessible in Java 9. See "Removed or Changed APIs" for more examples of changes that might affect your code.

The external libraries my application depends on all work with Java 9, so after I apply the two fixes above, everything compiles and runs as expected. However, you might find that some of the libraries you're using aren't compatible with Java 9. There is a list of major open source libraries that have been testing with Java 9. If you have a problem with a library, you should check whether there is an updated version of the library that supports Java 9.

Unexpected Behavior

Java 9 contains several changes that might affect your application; some might not cause compilation errors but instead will demonstrate different behavior. The migration guide mentioned earlier covers these changes. Here's a summary of some of the changes that might affect you. I've listed them as links with the associated JEP number, so you can look up any of them if they become suspects in what you perceive to be behavioral changes.

- JEP 231: Remove Launch-Time JRE Version Selection
- JEP 240: Remove the JVM TI hprof Agent
- JEP 241: Remove the jhat Tool
- JEP 260: Encapsulate Most Internal APIs
- JEP 289: Deprecate the Applet API
- JEP 298: Remove Demos and Samples
- JEP 214: Remove GC Combinations Deprecated in JDK 8
- JEP 248: Make G1 the Default Garbage Collector
- JEP 271: Unified GC Logging
- JEP 158: Unified JVM Logging
- JEP 223: New Version-String Scheme
- JEP 245: Validate JVM Command-Line Flag Arguments

Conclusion

In this article, I have examined how to port Java 8 code to Java 9, without getting into the question of modularity. This explanation is intended for readers who want to use many of the innovations in Java 9 discussed here and in the accompanying article by Simon Ritter, "Nine New Developer Features in JDK 9." [As this issue goes to press, modules have not been finalized by the Java Community Process, so coverage of working with modules will appear in a forthcoming issue of *Java Magazine*. —*Ed.*] </article>

Trisha Gee is a Java Champion with expertise in Java high-performance systems. She is a leader of the Seville Java User Group (SVQJUG) and dabbles in open source development. Gee is the IntelliJ IDEA developer advocate for JetBrains.

RAOUL-GABRIEL URMA
AND
RICHARD WARBURTON

Java 9 Core Library Updates: Collections and Streams

Collections, Streams, and iterators have all added new capabilities.

Java 9 introduces many refinements to Java 8 features, such as Streams, Collectors, Optional, and CompletableFuture, as well as enhancements to the Collections API. This article focuses on using the new features in Collections, Streams, and Collectors. In the next issue, we will focus on reliability using Optional and CompletableFuture.

Collection Factories

Java 9 adds a set of new factory methods to the Collections Framework. Let's start by looking at the problem these methods are trying to solve, by instantiating a list with a few String values:

```
List<String> values = new ArrayList<>();
values.add("Java 9");
values.add("is");
values.add("here");
```

Let's face it: this is bulky for such a simple and common thing to do. We admit that this isn't the only way to instantiate a List though. `Arrays.asList()` has been around since before Java 5, and it originally took just an array. In Java 5, it was converted to accept varargs and is in common use.

```
List<String> values =
    Arrays.asList("Java 9", "is", "here");
```

This is a helpful improvement. The List returned by `Arrays.asList()` is a little strange, however. If you try to add an element to the List, it'll throw an UnsupportedOperationException. You might think that is OK—after all, it's a List that cannot be mutated. Not so fast: it's actually a list that wraps an array. So the `set()` operation will modify the backing array and, in fact, if you hold onto the array that is wrapped, it can be modified. If you want to create a Set or a Queue, you are out of luck as well; there's no `Arrays.asSet()`. The normal way to solve this problem is to use the Collection constructor overload:

```
Set<String> values =
    new HashSet<>(Arrays.asList(
        "Java 9", "is", "here"));
```

Again, this is fairly verbose. Note that some programming languages offer a feature to solve this problem by adding collection literals to the programming language. This gives you some syntax that instantiates a collection from specific values. Here's an example in Groovy:

```
def values = ["Hello", "World", "from", "Java"] as Set
```

One solution to implementing this is to provide some methods that construct collections from values and then use a varargs constructor to make them syntactically shorter, similar to collection literals. This is the approach used in Java 9, so you can do the following, which is a lot more concise:

```
List<String> list =
    List.of("Java 9", "is", "here");
Set<String> set =
    Set.of("Hello", "World", "from", "Java");
```

Maps now also have similar factory methods. They work differently because Maps have keys and values rather than a single type of element. For up to 10 entries, Maps have overloaded constructors that take pairs of keys and values. For example, you could build a map of people and their ages, like this:

```
Map<String, Integer> nameToAge
    = Map.of("Richard", 49, "Raoul", 47);
```

The varargs case for Maps is a little bit harder: you need to have both keys and values, but in Java, methods cannot have two varargs parameters. Therefore, the general case is handled by taking a varargs method of `Map.Entry<K, V>` objects and adding a static `entry()` method that constructs them:

```
Map<String, Integer> nameToAge =
    Map.ofEntries(entry("Richard", 49),
                  entry("Raoul", 47));
```

The goal here isn't just to reduce verbosity; it is also to reduce the possibility of programmer errors. All the collections added in recent years have banned the use of nulls as elements within collections, and these collections follow suit. This helps reduce the scope for bugs related to referring to null values in collections. It also simplifies the internal implementation.

A bigger difference, compared to most collections in the JDK, is that these collections are immutable. Immutability reduces the scope for bugs by removing the ability for one part of an application to cause problems by modifying state that another component is relying on. Immutability is a concept that has been advocated by functional programming for a long time. Speaking of functional programming, let's look at the updates to the Streams API in Java 9.

Streams

Streams were a great addition to Java 8. The code that developers write using streams tends to read a lot more like the problem they are trying to solve, and less code is usually required. Java 9 brings some small improvements to streams.

ofNullable. The Stream interface has a pair of factory methods called `of()` that allow you to create streams from prespecified values: one is an overload for a single value, and the other takes a varargs parameter. These are very useful when you're trying to test streams code and when you want to just instantiate a stream with a few values. Java 9 adds an `ofNullable()` factory. Let's see how you might use this functionality.

Let's suppose you're trying to find a location where you can put some configuration files in a Java application. You want to use a couple of different properties: let's say `app.config` and `app.home`. Let's write this code in Java 8:

```
String configurationDirectory =
    Stream.of("app.config", "app.home", "user.home")
        .flatMap(key -> {
            final String property =
                System.getProperty(key);
             if (property == null)
             {
                 return Stream.empty();
            }
            else
            {
                return Stream.of(property);
            }
        })
        .findFirst()
        .orElseThrow(IllegalStateException::new);
```

That's a little ugly. What's going on here? The code is looking up each property in the Stream and using the `flatMap` operation. `flatMap` is used here because it allows you to map an element to zero or one elements in the Stream. If you can look up the system property, a Stream is returned containing only it, but if you can't look it up, an empty Stream is returned in its place, which results in no element being added to the Stream.

Unfortunately, a large statement-style lambda expression with a null check is in the middle of the code. One alternative would be to use a ternary operator:

```
String configurationDirectory =
   Stream.of("app.config", "app.home", "user.home")
         .flatMap(key -> {
              String prop = System.getProperty(key);
              return prop == null ? Stream.empty() :
                  Stream.of(property);
         })
         .findFirst()
         .orElseThrow(IllegalStateException::new);
```

Even after this refactoring, however, the code reads slightly inelegantly. The Java 9 `ofNullable` would allow you to write the same pattern much more succinctly and readably:

```
String configurationDirectory =
   Stream.of("app.config", "app.home", "user.home")
         .flatMap(key ->
           Stream.ofNullable(System.getProperty(key)))
         .findFirst()
         .orElseThrow(IllegalStateException::new);
```

takeWhile and dropWhile. Suppose you have an application that is processing payments being made on an ecommerce website and you're maintaining a list of all payments for the current day that are sorted from the largest down to the smallest. You have a business requirement to produce a report on every payment that is £500 or greater at the end of the day. A natural way of writing this code using Java 8 Streams might be:

```
List<Payment> expensivePayments = paymentsByValue
   .stream()
   .filter(transaction ->
      transaction.getValue() >= 500)
   .collect(Collectors.toList());
```

Unfortunately, the downside of this approach is that if you start processing many transactions in a day, the `filter` operation is applied to every transaction in your input list. You know that your input list is sorted by descending value of the transaction, so once you have found a transaction that fails your predicate, every transaction after that point can be filtered out. If your list of transactions grew to be very large, this would take an increasing amount of time to complete and incur unnecessary inefficiency. Java 9 solves this problem with the addition of the `takeWhile` operation.

```
List<Payment> expensivePayments = paymentsByValue
   .stream()
   .takeWhile(transaction ->
      transaction.getValue() >= 500)
   .collect(Collectors.toList());
```

While filter retains all elements in the Stream that match its predicate, takeWhile stops once it has found an element that fails to match. The dropWhile operation does the inverse: it throws away the elements at the start for which the predicate is false.

So far, we've talked about streams that have a defined order: an *encounter order*. The order of a stream can be defined at its source. For example, if you're streaming from a list of values, the order in the list is the encounter order. It is also possible to have Stream operations that introduce an encounter order into their pipeline, for example, sorted(). Most, but not all, of the practical use cases of takeWhile() and dropWhile() rely upon their input streams having a defined encounter order.

One use case for applying takeWhile() on an unordered stream is if you want to be able to stop the Stream operation. For example, perhaps you have a Stream operation that may operate on an infinite stream, processing all the data in it, but you want to be able to stop the Stream when your application shuts down or a user needs to cancel the stream pipeline. You can do this with a takeWhile() operation that reads from a piece of external state, such as a volatile boolean flag. When you want to stop the stream pipeline, you simply set the state to false.

iterate. A related update is the introduction of an alternative iterate() method for creating streams. The vintage iterate method from Java 8 takes an initial value and a function that provides the next value in the stream. Look at the following example:

```
IntStream.iterate(3, x -> x + 3)
         .filter(x -> x < 100)
         .forEach(System.out::println);
```

This code prints all the numbers that are divisible by 3 and less than 100. It starts with 3, which is divisible by 3, and adds 3 on every iteration. It then filters to ensure that the numbers are less than 100 and uses a method reference to print the resulting numbers. It looks straightforward, but if you run it, you'll find that there's a big problem. Go on: try it!

That's right: the program never terminates. It keeps adding 3 in a loop infinitely. That's because there's no way to know in the filter that the numbers continue to increase. You can solve that problem with the new version of iterate in Java 9, which takes a predicate as its second argument that indicates at what point to stop iterating up. So, rewrite the code as follows:

```
IntStream.iterate(3, x -> x < 100, x -> x + 3)
         .forEach(System.out::println);
```

It now stops running after it has printed the number 99. The sample code here used the IntStream interface because it was operating on primitive int values, but the iterate() methods appear on both the primitive and regular Stream interfaces. Streams weren't added on their own in Java 8; they came with a powerful Collectors class that has also been improved in Java 9, as discussed next.

Collectors

Collectors were another important addition to Java 8. Collectors let you specify data processing queries by aggregating the elements of a Stream into various containers such as Map, List, and Set. For example, you can create a map of the sum of expenses for each year by using the groupingBy and summingLong collector from the Collectors class. In the rest of this article, assume that there are static imports when a static method is referred to from the Collectors class.

```
Map<Integer, Long> yearToSum
    = purchases.stream()
               .collect(groupingBy(Expense::getYear,
                 summingLong(Expense::getAmount));
```

So what's new in Java 9? In Java 9, two new Collectors have been added to the Collectors utility class: `Collectors.filtering` and `Collectors.flatMapping`.

We will show you how to use the new filtering and flatMapping functionality with a running example used throughout the rest of the article. Here are the `Expense` and `Tag` class definitions we use.

```
public class Expense {
   private final long amount;
   private final int year;
   private final List<Tag> tags;

   public Expense(long amount, int year,
                  List<Tag> tags) {
       this.amount = amount;
       this.year = year;
       this.tags = tags;
   }

   public long getAmount() {
       return amount;
   }

   public int getYear() {
       return year;
   }

   public List<Tag> getTags() {
       return tags;
   }
}

public class Tag {
   private final String content;

   public Tag(String content) {
       this.content = content;
   }
}
```

Collectors.filtering. Let's revisit the example at the start of the "Collectors" section and say that you now need to build a map of the list of expenses for each year but only for expenses that are higher than £1,000.

From the immediately preceding discussion, you already know how to generate a Map of the list of expenses for each year, as follows:

```
Map<Integer, List<Expense>> yearToExpenses =
    purchases.stream()
             .collect(groupingBy(Expense::getYear));
```

So, you could add a filter to the streams, as follows:

```
Map<Integer, List<Expense>> yearToExpenses =
   purchases.stream()
            .filter(expense ->
                 expense.getAmount() > 1_000)
            .collect(groupingBy(Expense::getYear));
```

Unfortunately, this means that if all the expense amounts for a certain year were below £1,000, the resulting map would not contain an entry for that year (that is, no key and no value).

Instead, you can use the filtering collector, as follows, which would preserve the year in the resulting Map and produce an empty list. This would be a confusing thing in our report for a user to read. Is a given year missing because there is some missing data, or is it simply a software bug? We want to make it clear that there are no entries that meet their filter criteria in a given year and thus return an empty list.

```
Map<Integer, List<Expense>> yearToExpensiveExpenses =
    purchases.stream()
             .collect(
                groupingBy(
                  Expense::getYear,
                  filtering(expense ->
                     expense.getAmount() > 1_000,
                  toList())));
```

flatMapping. The `flatMapping` collector is the big brother of the mapping collector. Let's say you need to produce a map of years with a set of tags from the expenses for each year. In other words, you need to produce `Map<Integer, Set<Tag>>`.

A first attempt might look like this:

```
expenses.stream()
        .collect(
           groupingBy(
             Expense::getYear,
             mapping(Expense::getTags, toSet())));
```

Unfortunately, this query will return a `Map<Integer, Set<List<Tag>>>`.

By using `flatMapping`, you can flatten the intermediate Lists into a single container. The `flatMapping` collector takes two arguments: a function from one element to a Stream of elements and a downstream collector to collect the single flattened stream into a container. With this, you can solve the query, as follows:

```
Map<Integer, Set<String>> =
    expenses.stream()
            .collect(
               groupingBy(
                  Expense::getYear,
                  flatMapping(expense ->
                     expense.getTags().stream(),
                     toSet())));
```

Note that the `flatMapping` collector is related to the `flatMap` method from the Stream API. That method takes a function producing a Stream of zero or more elements for each element in the input Stream. The result is then flattened into a single Stream.

Conclusion

Java 9 adds new goodies to improve patterns by introducing new operations to Collections, Streams, and Collectors. These additions help you write code that reads closer to the problem statement and is easier to understand. Often when there are major new releases, developers focus only on the flagship features, but in practice, many developer productivity improvements can be found in the enhancement features of recent releases. Java 8 was a fantastic release, and Java 9 now makes some common patterns even simpler to use. </article>

Raoul-Gabriel Urma (@raoulUK) is the CEO and cofounder of Cambridge Spark, a leading learning community for data scientists and developers in the UK. He is also chairman and cofounder of Cambridge Coding Academy, a community of young coders and students. Urma is coauthor of the bestselling programming book *Java 8 in Action* (Manning Publications, 2015). He holds a PhD in computer science from the University of Cambridge.

Richard Warburton (@richardwarburto) is a software engineer, teacher, author, and Java Champion. He is the author of the best-selling *Java 8 Lambdas* (O'Reilly Media, 2014) and helps developers learn via Iteratr Learning and at Pluralsight. Warburton has delivered hundreds of talks and training courses. He holds a PhD from the University of Warwick.

NOTHING IS WORSE
THAN AN ITCH...
Programming
FOR
8 BIT DEVICES
DEVOXX™
6 - 10 NOV 2017 - BELGIUM

CONSTANTIN DRABO

JShell: Read-Evaluate-Print Loop for the Java Platform

Testing code snippets is now part of the JDK.

JShell, a new read-evaluate-print loop (REPL), will be introduced in JDK 9. Motivated by project Kulla (JEP 222), JShell is intended to provide developers an API and an interactive tool that evaluates declarations, statements, and expressions of the Java programming language.

In this article, I present a brief overview of JShell, explain its use, and demonstrate its benefits for developers.

Overview

JShell is a new tool in JDK 9 that offers a basic shell for Java that uses a command-line interface. It is also the first official REPL implementation for the Java platform, even though this concept has existed in many languages (for example, Groovy and Lisp) and in Java third-party tools (such as Java REPL and BeanShell). So, it is not a new language, nor is it an IDE or a new compiler for Java.

JShell acts like a UNIX shell: it reads the instructions, evaluates them, prints the result of the instructions, and then displays a prompt while waiting for new commands. It is built around several core concepts—snippets, state, wrapping, instruction modification, forward references, and snippet dependencies—that I'll explain.

How to Run JShell

In order to run JShell, you need to download and install the latest Early Access preview build for JDK 9 for your environment. Then, set your JAVA_HOME environment variable and run `java -version` to verify your installation. The output of the command should show something similar to the following:

```
java version "9-ea"
Java(TM) SE Runtime Environment (build 9-ea+173)
Java HotSpot(TM) 64-Bit Server VM...
```

To run JShell, type `jshell` at the command line:

```
[pandaconstantin@localhost ~]$ jshell
|  Welcome to JShell -- Version 9-ea
|  For an introduction type /help intro
```

When the prompt is available, you can get help on several useful commands by typing `/help` at the command line. **Figure 1** shows the partial output from that command, with many of the principal commands explained succinctly.

To understand how JShell works, let's look at a few code snippets. A *snippet* is an instruction that uses standard Java syntax. It represents a single expression, statement, or declaration. The following is a simple snippet. The text below the command is JShell output:

```
System.out.println("My JShell snippet");
My JShell snippet
```

(In my examples in this article, the characters in blue indicate text entered at the command line into JShell, and the resulting output is shown in black monospace.)

Like Java code, JShell allows you to declare variables, methods, and classes:

```
int x, y, sum
```

```
/list [<name or id>|-all|-start]]
    list the source you have typed
/edit <name or id>
    edit a source entry referenced by name or id
/drop <name or id>
    delete a source entry referenced by name or id
/save [-all|-history|-start] <file>
    Save snippet to a source file.
/open <file>
    open a file as source input
/vars [name or id|-all|-start]
    list the declared variables and their values
/methods [name or id|-all|-start]
    list the declared methods and their signatures
/types [name or id|-all|-start]
    list the declared types
/imports
    list the imported items
/exit
    exit the jshell
/history
    History of what you have typed
/!
    Re-run last snippet
```

Figure 1. Partial list of JShell commands

```
x ==> 0
y ==> 0
int sum ==> 0

x = 10 ; y = 20 ; sum = x + y;
x ==> 10
y ==> 20
sum ==> 30

System.out.println("Sum of " + x + " and " + y +
  " = " + sum);
Sum of 10 and 20 = 30
```

Here's an example of a valid class, which I use later:

```
class Student {
private String name ;
private String classRoom ;
private double grade ;

public Student() {

}

public String getName() {
return name ;
}

public void setName(String name) {
this.name =  name ;
}

public String getClassRoom() {
return classRoom ;
}
public void setClassRoom(String classRoom) {
this.classRoom  = classRoom ;
}

public double getGrade() {
return grade ;
}

public void setGrade(double grade) {
this.grade  = grade ;
}

}
created class Student
```

The indentation looks different than in Java, because this code was typed at the JShell command line. Some normal Java statements are not needed at this initial declaration. For example, JShell automatically imports many typical packages. In our example, the following imports were done automatically:

```
import java.io.*
import java.math.*
import java.net.*
import java.nio.file.*
import java.util.*
import java.util.concurrent.*
import java.util.function.*
import java.util.prefs.*
import java.util.regex.*
import java.util.stream.*
```

At any given point, you can see the already imported packages with the `/import -all` command listed earlier.

State. Each statement in JShell has a state. The state defines the execution status of snippets and of variables. It is determined by results of the `eval()` method of the JShell instance, which evaluates code. There are seven status states:

- DROPPED: The snippet is inactive.
- NONEXISTENT: The snippet is inactive because it does not yet exist.
- OVERWRITTEN: The snippet is inactive because it has been replaced by a new snippet.
- RECOVERABLE_DEFINED: The snippet is a declaration snippet with potentially recoverable unresolved references or other issues in its body.
- RECOVERABLE_NOT_DEFINED: The snippet is a declaration snippet with potentially recoverable unresolved references or other issues. (I discuss the difference between this and the previous state shortly.)
- REJECTED: The snippet is inactive because it failed compilation upon initial evaluation and it is not capable of becoming valid with further changes to the JShell state.
- VALID: The snippet is a valid snippet.

When a snippet is not declared, it is considered inactive and not part of the state of the JShell instance nor is it visible to the compilation of other snippets. At this stage, it is a NONEXISTENT snippet.

If the snippet is submitted to the `eval()` method and there are no errors, it becomes part of the state of the JShell instance and the status is VALID. Querying JShell gives `isDefined == true` and `isActive == true`.

In the case where the signature of the snippet is valid but the body contains issues or unresolved references, the status is RECOVERABLE_DEFINED and a JShell query states `isDefined == true` and `isActive == true`.

If the signature of the snippet is wrong and the body also contains issues or unresolved references, the snippet's status is RECOVERABLE_NOT_DEFINED and the status is `isDefined = false` even though the snippet stays active (`isActive == true`).

A snippet becomes REJECTED when compilation fails, and it is no longer a valid snippet. This is a final status and will not change again. At this stage, both isDefined and isActive are false. You can also deactivate and remove a snippet from the JShell state with an explicit call to the JShell.drop(jdk.jshell.PersistentSnippet) method. At that point, the snippet status changes to DROPPED. This is also a final status and will not change in the future.

Sometimes a snippet type declaration matches another one. In this case, the previous snippet is inactive and it is replaced by the new one. The status of the old snippet becomes OVERWRITTEN and the snippet is no longer visible to other snippets (isActive == false). OVERWRITTEN is also a final status.

Using JShell from a Program

The JDK offers APIs to developers to access JShell programmatically rather than by using the REPL. The code below creates an instance of JShell, evaluates a snippet, and provides the status of the instructions.

```
import java.util.List;
import jdk.jshell.*;
import jdk.jshell.Snippet.Status;

public class JShellStatusSample {
  public static void main(String... args) {
    //Create a JShell instance
    JShell shell  = JShell.create();
    //Evaluate the Java code
    List<SnippetEvent> events =
      shell.eval( "int x, y, sum; " +
        "x = 15; y = 23; sum = x + y; " +
        "System.out.println(sum)" );
    for(SnippetEvent event : events) {
      //Create a snippet instance
      Snippet snippet = event.snippet();
      //Store the status of the snippet
      Snippet.Status snippetstatus  =
         shell.status(snippet);
      if(snippetstatus  == Status.VALID) {
        System.out.println("Successful ");
      }
    }
  }
}
```

The result of the execution of this code is

```
Successful
Successful
Successful
```

Wrapping

You are not obliged to declare variables or define a method within a class. Classes, variables, methods, expressions, and statements evolve within a synthetic class (as an artificial block). You can define them in the top-level context or within a class body, as you wish. Also, if you're a person who prefers concision, you can at times skip using semicolons.

```
String firstName , lastName
firstName ==> null
lastName ==> null

String concatName(String firstName,
String lastName) {
return firstName + lastName ;
}
|  created method concatName(String,String)
```

The following code shows the declaration of variables and a method in the top-level context. As discussed previously, you cannot modify classes at the top level; however, as seen in the following code, you can modify methods within classes.

```
class Person {

private String firstName ;
private String lastName ;

public String concatName(String firstName,
String lastName) {
return firstName + lastName;
}

}
|  created class Person
```

Because each statement or expression is created in its own unique namespace, modifications can be applied at any time without disturbing the overall functioning of the code.

Forward References and Dependencies

Within the body of a class, you can refer to members that will be defined later. During evaluation of the code, the references produce errors. But because JShell works sequentially, the issue can be resolved by writing the missing member before actually calling the snippets.

When a snippet A depends on a second snippet B, any changes in snippet B are immediately propagated in A. Then, if the dependent snippet is updated, the main snippet is also updated. If the dependent snippet is invalid, the main snippet becomes invalid.

If you declare variables and then initialize them, you can see them by using the list command, for example:

```
String firstname;
firstname ==> null

String lastname;
lastname ==> null

double grade;
grade ==> 0.0

String getStudentFullName(String fn, String ln) {
return fn + " " + ln ; }
| created method getStudentFullName(String,String)

firstname  =  "Wolfgang";
firstname ==> "Wolfgang"

lastname  = "Mozart";
firstname ==> "Mozart"

System.out.println("Hello " +
getStudentFullName(firstname,lastname));
Hello Wolfgang Mozart
```

The output of the list command shows the following:

```
1 : String firstname ;
2 : String lastname ;
3 : double grade ;
4 : String getStudentFullName
(String firstname, String lastname) {
     return firstname + " " + lastname ;
 }
5 : firstname = "Wolfgang" ;
6 : lastname  = "Mozart" ;
7 : System.out.println("Hello" +
getStudentFullName(firstname,lastname));
```

The numbers in the output are the snippet identifiers. They are useful for manipulating a snippet (editing, dropping, and so on). You can also list all the variables, methods, and classes that are in the code. Here's an example of listing all the variables:

```
/vars
|    String firstname = "Wolfgang"
|    String lastname = "Mozart"
|    double grade = 0.0
```

If you decide to change the values of variables or edit a specific snippet, you run `/edit` with the snippet identifier, for example:

```
/edit 6
```

A dialog box, as shown in **Figure 2**, appears to allow you to modify the value.

If I use the editor to change lastname to my last name, I get the following result response:

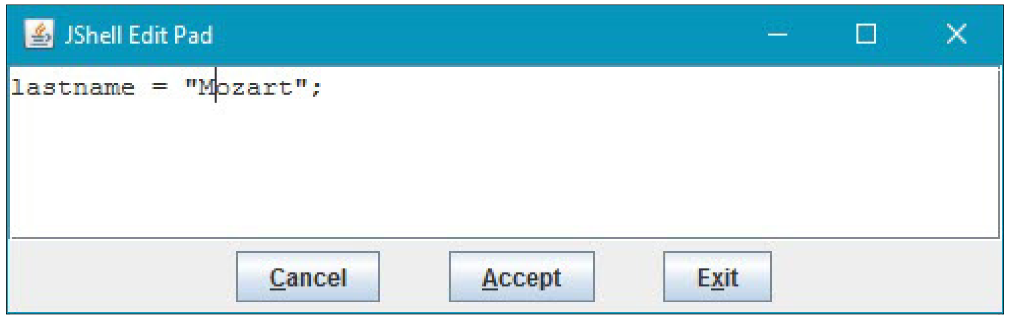
```
lastname ==> "Drabo"
```

If I change the firstname to my first name, then I can rerun the print-out function by simply referring to the snippet identifier, in this case, #7:

```
/7
System.out.println("Hello" +
getStudentFullName(firstname, lastname));
Hello Constantin Drabo
```

You can use the `/save` command to save your snippets to a file, and the `/open` command allows you to open and run the file:

```
/save StudentName.jsh
```



```
lastname = "Mozart";
```

Figure 2. The built-in snippet editor

```
/open StudentName.jsh
Hello Wolfgang Mozart
Hello Constantin Drabo
```

JShell also offers some keyboard shortcuts. You can obtain the navigation history by using the up and down arrow keys. And you can use the tab key to show you the options you have for the text you've typed so far—a kind of IntelliSense feature.

Conclusion

JShell has many possible uses. The first is for beginners to try out Java code without having to write a full program. In this sense, it is a terrific learning tool. It's also a great tool for trying out small functions, validating that a web service is available and seeing what it returns, and so on. In addition, it's an excellent tool for trying out some quick layouts in JavaFX.

Whether it is used from the command line or programmatically, JShell is likely to become one of the most widely used features of JDK 9. </article>

Constantin Drabo is a software engineer living in Burkina Faso. He is a NetBeans Dream Teamer and a Fedora Ambassador for the Fedora Project. Drabo is also the founder of FasoJUG, the first Java user group in Burkina Faso (the former Upper Volta).

[This article is a substantially updated version of the JShell 9 tutorial that appeared in the July/August 2016 issue. —*Ed.*]

JIM LASKEY

Nashorn JavaScript Engine in JDK 9

Handy additions and support for ES6 make Nashorn even more useful.

Nashorn, the JDK's built-in JavaScript engine, has undergone a variety of improvements in this new release. Before diving into those enhancements, let me briefly set the context for these changes so that their addition makes the most sense.

Background

When I started the development of Nashorn in late 2010, I was just looking for a way to experiment with the newly minted `invokedynamic` (JSR 292) byte-code instruction. The JVM team later adopted Nashorn as a test bed for `invokedynamic`, and the Nashorn Project drove much of the performance improvements made to the `invokedynamic` implementation.

While this was going on, the Java group was discussing how JavaScript would likely grow to dominate client-side development and that integrating JavaScript with Java would be a critical element of the client/server equation. Rhino—an open source implementation of JavaScript from the Mozilla Foundation that was then the JDK offering for JavaScript—was getting long in the tooth, and further development was winding down. In November 2012, JDK Enhancement Proposal (JEP) 174, "Nashorn JavaScript Engine," was approved, which enabled work to begin in earnest to provide a fresh, robust, secure, full-featured implementation of ECMAScript-262 Edition 5.1 (ES5) to run on the Java platform.

Initially, Nashorn found a home in a wide variety of applications, such as app servers, JavaFX applications, utilities, shell scripts, embedded systems, and so on.

Nashorn continues to have broad usage, but its use appears to have settled into three main areas:

- The development of JavaScript applications that can be run on both the client and the server. In the JavaScript world, this is known as *isomorphic development*. The advantages are huge for smaller shops that want to build front ends for both desktop and mobile services—a single programming language with a common codebase and rapid delivery. Isomorphic development also scales well for larger systems.
- Runtime adaptive or dynamic coding. The term I like to use is *soft coding*, where portions of an application can be modified after the application/server is deployed. This capability is used for everything from stored procedures in databases to application configuration management.
- Shell scripting. This consists of using JavaScript in areas where bash or Python would traditionally be used.

There is often a lot of debate about Nashorn's performance on the JVM compared to native JavaScript performance on platforms such as Google's V8. Nashorn starts out slower because it translates the JavaScript to bytecode. After that, Nashorn is at the mercy of HotSpot to deliver native code. This approach works well for long-running server applications but not as well for small and run-once scripts. The main reason we developed Nashorn in the first place was to run JavaScript and Java together seamlessly. Most Nashorn applications do a great job of orchestrating JavaScript and Java, something that V8 was not designed to do.

ES6 Support

The most important issue for JavaScript developers doing isomorphic development is the need to have client/server source code compatibility. With most browsers adopting ECMAScript 6 (ES6) as the standard level of language support for JavaScript, it was essential to bring Nashorn in line. While the Nashorn team didn't have time to deliver the complete ES6 feature set for the JDK 9 release, more ES6 features will follow in future updates. To activate ES6 support, use `--language=es6` on the command line.

New keywords. Here is what you'll find initially: the implementation of the new keywords, `let` and `const`, follow ES6 block scoping, with `const` creating an immutable local. For backward compatibility, `var` retains its ES5.1 semantics:

```
const a = 10;
a = 20; // TypeError trying to set a constant

let x = 10;
{
    let x = 20; // different scope
    print(x); // prints 20
}
print(x); // prints 10 from the outer scope
```

Support for symbols. Symbols are string-like objects introduced in ES6. They are primitives that are compared by reference instead of by value. This makes every symbol unique, which enables developers to create private object properties.

```
// a new, unique symbol
let unique = Symbol('optionalName');
myobj[unique] = 'foo';
// a shared symbol for the given name
let shared = Symbol.for('name');
myobj[shared] = 3;
```

The first example creates a distinct symbol, which can be used to create a property that is private to the current scope. The second example shows how symbols can be shared (*interned*) across an entire context. ES6 uses shared symbols to define iterators and default `toString` functions. Symbol-keyed properties are not exposed to any reflective operations.

New iterator protocol. ES6 provides a new protocol for iterating over objects:

```
// assign an iterator function to the
// Symbol.iterator property
myobj[Symbol.iterator] = function() {
     return {
        next: function() { ... }
     }
};
// Iterate over myobj using a for..of statement
for (let id of myobj) {
    print(myobj[id]);
}
```

Note the introduction of `for..of` loops. Nashorn's syntax `for each` will still be available, but the team recommends switching to `for..of` even though they are functionally equivalent.

New collections. New collection classes were added in ES6 and in Nashorn: `Map` and `Set`. These collection classes implement the new iterator protocol.

```
let map = new Map();
map.set('foo', 'bar');
map.get('foo');  // -> 'bar'
map.clear();
map.get('foo');  // -> undefined
```

The weak reference versions of Map and Set have also been implemented; WeakMap and WeakSet. An entry in one of these collections is removed by the garbage collector when the entry value is no longer referenced by any other variable or object.

Templates. Templates are a new form of string literals using back-ticks as delimiters, which allow embedded expressions and multiline strings. Nashorn also supports ES6 "tagged" strings rendered by a function.

```
// Multiline string
'string text line 1
 string text line 2'
// Embedded expression
'string text ${expression} string text'
// Rendered by function "tag"
tag 'string text ${expression} string text'
```

Note that in -scripting mode, back-ticks are still used for $EXEC expressions.

Binary and octal numbers. Finally, there is a new syntax for binary and octal number literals.

```
// Binary number literal
0b111110111 === 503
// Octal number literal
0o767 === 503
```

In addition to these features from ES6, new capabilities enhance Nashorn's usefulness.

JavaScript Parsing API

In JDK 9, Nashorn's Parser API has been expanded to include full ES6 syntax support. The following example shows how to parse a JavaScript sample:

```
// load parser.js from Nashorn resources
//// Load the parser library
// Sample script containing an ES6 class declaration
var script = "class XYZ {}";

// Parse the script and  build an abstract syntax tree
var json = parse(script);

// Convert the abstract syntax tree to JSON and print
print(JSON.stringify(json, undefined, 4));
```

The result is the following JSON output:

```
{
   "type":"Program",
   "body":[
      {
         "type":"VariableDeclaration",
         "declarations":[
            {
               "type":"VariableDeclarator",
               "id":{
                  "type":"Identifier",
                  "name":"XYZ"
               },
               "init":{

               }
            }
         ]
      }
   ]
}
```

The Parsing API can also be used to analyze code or inject instrumentation, and so offers a developer lots of flexibility. NetBeans uses this API to handle most of its JavaScript functionality.

$EXEC 2.0

One of the most popular features of Nashorn is the ability to fork processes with a simple back-tick expression. This unusual capability enables developers to create shell scripts written in JavaScript.

```
var listing = `ls -l`;
```

In this example, `listing` will contain the output string from the `ls -l` command.

The back-tick expression is just shorthand for a call to the function $EXEC. That is, `` `ls -l` `` is shorthand for `$EXEC('ls -s')`.

$EXEC in JDK 9 is greatly improved over the previous version. You can now pass in an array of strings for the first argument, which allows you to pass arguments containing special characters, such as spaces.

```
$EXEC(["/bin/echo", "my.java", "your java"]);
```

The $EXEC function now has a `throwOnError` property that, if set to `true`, will raise a RangeError if the command fails. It is still possible to check the $EXIT global to check the status of the command.

```
$EXEC.throwOnError = true
`javac my.java`
<shell>:1 RangeError:
    $EXEC returned non-zero exit code: 2
```

[The last two lines appear as a single line. —*Ed.*] It is possible to better manage command I/O operations with standard redirection directives as command arguments.

```
$EXEC("echo 'my argument has spaces' > tmp.txt");
```

It is also possible to override `stdin`, `stdout`, and `stderr` by passing input and output streams as the last three $EXEC arguments.

Multiple commands can be issued by using semicolons or new lines.

```
$EXEC(<<EOD);
echo this ; echo that
echo whatever you want
EOD
```

Or, you can pipe the results of commands to the next command using the vertical bar symbol.

```
$EXEC("echo 'my argument has spaces' | cat");
```

Finally, you can change the values of environment variables on the fly with pseudo cd, `setenv`, and `unsetenv` built-in commands.

```
$EXEC("setenv PATH ~/bin:${ENV.PATH}; mycmd");
$EXEC("cd ~/bin; ls -l");
```

jjs as a REPL

The release of Nashorn with JDK 8 introduces a new command-line tool: Java JavaScript (jjs). jjs made it easier for developers to test out Nashorn features and to launch JavaScript applications without writing Java code.

With JDK 9, jjs has replaced its input and output API with jline2, a new library for handling console input. This change means that developers can use all the standard controls that are expected when they are using a shell:

- Left and right arrows to move through the input and an option arrow to jump past symbols
- Forward and backward deletion capability

- Ctrl-k to delete the rest of a line and Ctrl-y to restore the line
- Up and down arrows to scroll through the input history
- Tab completion to expand globals, properties, and Java types
- VT100 escape sequences to format the screen

With all these small changes, life is made a little bit easier.

Performance

In JDK 9, optimistic type optimization is on by default, which means that, over time, the performance of code on Nashorn improves as stronger typing is determined by the engine.

```
function f(array, i) {
    return array[i] + array[i + 1];
}
```

In the example above, Nashorn will initially optimize the function `f` assuming that `i` and the contents of `array` are integers (because of the plus sign.) If it turns out that these assumptions are not correct, Nashorn will create a new edition of the function with the actual type used and hot-swap the old code with the newer version. This optimization is specialized per call site, so the best solution is always used.

Conclusion

We've seen here that the Nashorn engine in JDK 9 provides many convenient new features. This summary covers what's new in this release, but the team has a lot more in the pipeline that it hopes to show you soon. </article>

Jim Laskey (@wickund) is a senior development manager in the Java Platform group at Oracle. Laskey has been a compiler-runtime developer since the mid-1970s and has worked at Symantec, Apple, and Azul Systems.

THE LONDON JUG

The London Java Community (LJC), also known as the LJC JUG (@ljcjug), comprises developers in Europe and in the international scene. The LJC was created in 2006 when Barry Cranford from the London Java recruitment firm RecWorks sought to bring together like-minded Java developers for sharing knowledge and skills. Since then, the LJC has grown to more than 6,000 members.

From the early days, the LJC had the good fortune to have leaders such as Zoe Slattery, Ben Evans, Martijn Verburg, Simon Maple, John Stevenson, Trisha Gee, and many other seasoned Java developers.

The LJC has been actively involved with the Java Community Process (JCP) and has won the 11th Duke's Choice Award and co-won the JCP Member of the Year honor.

With help and support from staff from RecWorks, the LJC organizes three to four events every month. One of these is a regular hack day called HackTheTower, where participants gather in groups hacking on OpenJDK, doing projects in Scala or Clojure or working independently on pet projects. The LJC also runs the annual Open Conference (UnConf) and supports Devoxx UK and Devoxx4Kids every year. These conferences involve close cooperation between the LJC and other JUGs in the UK, such as those in Bristol and Manchester, England.

In addition, LJC members share homegrown libraries and frameworks, answer questions on the mailing list, and participate in JUG-organized projects such as Adopt-a-JSR and Adopt OpenJDK. If you think that sounds fun, join up!

Working with the New HTTP/2 Client

An incubating technology in JDK 9 promises to make HTTP communication a lot simpler.

GASTÓN HILLAR

Java 9 introduces a new HTTP client API defined in Java Enhancement Proposal (JEP) 110 that implements HTTP/2 and WebSocket. This new HTTP client is included as an incubator module, and its goal is to replace the legacy Http-URLConnection API. In this article, I explain how to work with the asynchronous API provided by the new incubator HTTP client. Specifically, I show how to use the new HTTP client API with JShell, the new read-eval-print loop (REPL) included with JDK 9.

I start with an introduction to this new client showing basic synchronous usage. Then, I move to the asynchronous version and I use the client to perform a basic GET request; I also work with HTTP/2 over TLS. This way, you can see how to work with the new HTTP client to take advantage of the language features that provide nonblocking behavior.

A Modern HTTP Client

Whenever it was necessary to use an HTTP client in Java, it was a common practice to use third-party clients. The new incubator HTTP client is capable of working with HTTP/1.1 and HTTP/2. It is possible to work with HTTP/2 over TLS (known as *h2*) and HTTP/2 over TCP using cleartext (*h2c*).

The new client is capable of including an upgrade header field with the h2c token to request an upgrade from HTTP/1.1 to HTTP/2 over TCP. If the server doesn't support h2c, there won't be an upgrade and everything will go on working with HTTP/1.1. h2c stands for "HTTP/2 cleartext" and, therefore, it is important and easy to remember that h2c is not encrypted.

Presently, most configurations that support HTTP/2 provide support only for h2. Thus, it is very important to understand how to configure the client to work with modern versions of TLS. I include a complete example to cover this important scenario.

The combination of the new client with the upgrades in the TLS stack makes it possible to provide support for Application-Layer Protocol Negotiation (ALPN) and, therefore, the client can use this TLS extension to negotiate HTTP/2 with fewer round-trips. Based on the configuration settings you use for the HTTP client, it can negotiate the previously explained upgrade from HTTP/1.1 to h2c or select HTTP/2 (h2) from scratch. In addition, the client provides support for WebSocket conformity with RFC 6455. In this example, I focus on the HTTP features.

The new incubator module is included in jdk.incubator .httpclient in JDK 9. It is very important to take into account that this incubator module will be moved to another module in future JDK versions. In previous prerelease versions of the JDK, the module had a different name. You need to make sure you are running the latest JDK 9 version for the following code samples to work as expected.

I use JShell to make it easier to demonstrate the usage of this HTTP client without all the necessary specific configurations for build systems or IDEs to work with JDK 9. When I was writing this article, many IDEs and build systems were still not 100 percent compatible with JDK 9, so you might see unexpected issues. However, once the IDEs and build systems provide full compatibility with JDK 9, you can use the code samples in a Java application built with any IDE.

The following command launches JShell with the jdk.incubator.httpclient module specified as a value for the --add-modules option. This way, JShell will resolve the jdk.incubator.httpclient and you will be able to use it within the JShell session. If you have multiple JDK versions installed and you don't have JDK 9 in your path, you need to run the command in the bin folder for JDK 9. In macOS or Linux, you might need to replace jshell with ./jshell.

```
jshell --add-modules=jdk.incubator.httpclient
```

JShell doesn't require you to use semicolons (;) at the end of the statements. However, in order to make the code compatible with the Java code for building real-life applications, I prefer to use semicolons. Enter the following import statement in JShell:

```
import jdk.incubator.http.*;
```

JShell includes many import statements by default. However, if you don't work with JShell, you need to add the following additional import statements for the sample code to work:

```
import java.lang.*;
import java.net.URI;
import java.net.URISyntaxException;
```

The new module separates requests from responses. The following are the principal classes with which I work in the sample code to perform HTTP operations:

- HttpClient: Represents an immutable HTTP client with a specific configuration and allows you to send requests and receive responses
- HttpRequest: Represents an HTTP request
- HttpResponse: Represents an HTTP response

The API provides builders to create instances and configure the different pieces. These are static methods that start with the prefix new. However, the API doesn't provide builders for the headers represented with the HTTPHeader class. Unfortunately, the URI is still specified as a java.net.URI instance. Thus, if you need to use query parameters, it is necessary to format or concatenate strings.

The following lines build an HttpClient instance named client by chaining many method calls after the call to the HttpClient.newBuilder method that creates a new HttpClient builder. For example, the call to the followRedirects method with HttpClient.Redirect.ALWAYS as an argument specifies that I want the client to always follow redirects. In this case, I want to follow redirects because I want to perform an HTTP GET request to the following URI: http://www.oracle.com. I do not specify any desired HTTP version for the client; therefore, the client will be created as an HTTP/1.1 client that won't request an upgrade to HTTP/2 in the request header. The final chained call to the build method finishes the code for the builder.

```
HttpClient client = HttpClient.newBuilder()
    .followRedirects(HttpClient.Redirect.ALWAYS)
    .build();
System.out.println(client.version());
URI uri = new URI("http://www.oracle.com");
HttpRequest request = HttpRequest
    .newBuilder()
    .uri(uri)
    .GET()
    .build();
```

After the HttpClient instance is built, I print the result of calling the client.version() method. JShell displays HTTP_1_1 because I used the default configuration. Its default value is HttpClient.Version.HTTP_1_1, and it makes the client work only with HTTP/1.1.

The next line of code creates a new URI instance using the URI to which I want to make the HTTP GET request, and it saves it in uri. Then, the code builds an HttpRequest instance named request by chaining many method calls after the call to the HttpRequest.newBuilder method that creates a new HttpRequest builder.

The call to the URI method with uri as an argument specifies the URI to which I want to make the request. Then, the chained GET method indicates that I want to make an HTTP GET request to the specified URI. The final chained call to the build method finishes the code for the builder. The code is very easy to read. The *GET* HTTP verb is clearly indicated as a method that I can easily recognize in the code that builds the HttpRequest instance.

Then, the code shown next calls the client.send method that runs synchronously and blocks the execution until a response is retrieved. The code passes the previously created HttpRequest instance, request, and HttpResponse.BodyHandler.asString() as arguments.

```
HttpResponse<String> response =
    client.send(request,
        HttpResponse.BodyHandler.asString());
System.out.println(
    String.format("Status code: %d",
        response.statusCode());
System.out.println(
    String.format("Body length: %d",
        response.body().length());
```

The second argument specifies the response body handler I want to use. A body handler takes the response status code and the response headers and returns an HttpResponse.BodyProcessor instance. In this case, HttpResponse.BodyHandler.asString() returns a body processor that stores the response body as a String with the default charset. After the HTTP GET request is successfully processed, the send method returns an HttpResponse<String> that is saved in response.

With just a few lines of code in JShell, I was able to use the new HTTP client with a very basic configuration and its synchronous API.

The final lines print the results of the statusCode() and body().length() methods that provide the HTTP status code returned by the response and the length of the body retrieved as a String.

(These lines of code won't work in JShell if you enter them as they are displayed. It is necessary to enter each statement in a single line. Unfortunately, when I was writing this article, JShell didn't allow entering code with multiple lines without throwing some unexpected errors. However, putting all the code in a single line in this article would make it very difficult to read.)

The previous code retrieves the body for a web page with an HTTP GET request to a String and uses HTTP/1.1. This is the simplest usage you might have for the new HTTP client. If you read the code again, you will notice that both the HttpClient and the HttpRequest are built in a similar way. The specification of the BodyHandler makes it easy to understand that I want the HttpResponse as a String. You just need to think about working with builders and configuring them based on your goals. The API is clear and chainable.

If you are used to modern Java code, you will find the API very easy to use. With just a few lines of code in JShell, I was able to use the new HTTP client with a very basic configuration and its synchronous API.

Working with Asynchronous Execution

Before demonstrating the usage of the client to work with HTTP/2 and TLS, I'll work with the asynchronous API. JShell includes many import statements by default. However, if you don't work with JShell, you need to add the following additional import statement for the next sample code to work:

```
import java.util.concurrent.CompletableFuture;
```

In the previous lines that worked with synchronous execution, I called the `client.send` method. Now, I will write similar code to build the `HttpRequest`, and this time I will call the `client.sendAsync` method that returns a `CompletableFuture<HttpResponse<String>>`. This way, you can take advantage of the support of dependent functions and actions that a `java.util.concurrent.CompletableFuture<T>` triggers upon its completion and write code that uses the `HttpResponse<String>`. [If you're not familiar with completable futures, have a look at the explanation of them in Andrés Almiray's article on JDeferred in the May/June 2017 issue of *Java Magazine*. —*Ed.*]

In the following code, I call `response.whenComplete` to run code that prints the results of the `statusCode()` and `body().length()` methods that provide the HTTP status code returned by the response and the length of the body retrieved as a String, if no exception was thrown. Because the asynchronous task generated under the hood can throw an exception, I specify `HttpResponse<String> response` and `Throwable exception` when I declare the lambda expression that will be executed when the asynchronous task is completed. In the lambda expression body, I work with the `response` after making sure that the `exception` argument is null.

```
HttpClient client = HttpClient.newBuilder()
    .followRedirects(HttpClient.Redirect.ALWAYS)
    .build();
URI uri = new URI("http://www.oracle.com");
HttpRequest request = HttpRequest
    .newBuilder()
    .uri(uri)
    .GET()
    .build();
CompletableFuture<HttpResponse<String>> response =
    client.sendAsync(request,
      HttpResponse.BodyHandler.asString());
response.whenComplete((HttpResponse<String> response,
      Throwable exception) -> {
      if (exception == null) {
            System.out.println(
                  String.format("Status code: %d",
                         response.statusCode()));
            System.out.println(String.format(
                  "Body length: %d",
                  response.body().length()));
      } else {
            System.out.println(String.format(
                  "Something went wrong. %s",
                  exception.getMessage()));
      }
});
```

With just these few lines of code in JShell, I was able to use the new HTTP client with a very basic configuration and its asynchronous API.

Working with HTTP/2 over TLS

The HTTP client uses standard Java TLS mechanisms to enable work with HTTP/2 over TLS (h2). To do this programmatically, it is necessary to create and initialize a `javax.net.ssl.SSLContext` instance and pass it as an argument to the `sslContext` method chained to the `HttpClient` builder. The name chosen for the sslContext method is confusing, because it makes you think that the HttpClient will use the old and deprecated SSL instead of TLS to work with h2. It is important to avoid confusion about the usage of "SSL" and the various names. The sslContext method will configure a TLS context, not an SSL context. I work with TLS version 1.2 (TLSv1.2) in my next example.

For software development tasks, you will find the new HTTP/2 client to be extremely useful, especially when you need to work with an interactive REPL such as JShell.

I also use the Bouncy Castle libraries to make it easy to load certificates. I end up generating an SSLContext instance configured for TLSv1.2. These Bouncy Castle libraries are very popular when developers are working with TLS in Java. Using them, I combine the new HTTP client with the usage of Bouncy Castle libraries to work with h2 in JShell. When I was in the process of writing this article, the latest version of the Bouncy Castle libraries was 1.57 and, therefore, I use the names for the JAR files that include this version number.

I won't be working with any specific build system. I will continue to make the examples work with JShell. However, you can easily use the code as a baseline to configure any build system to work with the Bouncy Castle libraries.

Download the following JAR files from the Bouncy Castle site and save them in a folder:

- bcmail-jdk15on-157.jar
- bcpkix-jdk15on-157.jar
- bcprov-jdk15on-157.jar

The following line launches JShell with the jdk.incubator.httpclient module specified as a value in the --add-modules option and the previously enumerated JAR files specified as values for the --class-path option. This way, JShell will resolve the jdk.incubator.httpclient and load the specified class files that allow us to work with the Bouncy Castle libraries. Make sure you launch JShell in the path in which you saved the JAR files. If you don't have jshell included in the path, you need to specify its full path:

```
jshell --add-modules=jdk.incubator.httpclient --class-
path=bcpkix-jdk15on-157.jar;bcmail-jdk15on-157
.jar;bcprov-jdk15on-157.jar
```

The following code contains all the import statements. I'm also including many import statements that JShell runs by default, to make it easy to run the code when you don't use JShell. Import statements that duplicate ones JShell brings in automatically do not generate errors.

```
import jdk.incubator.http.*;
import java.util.concurrent.CompletableFuture;
import java.lang.*;
import java.net.URI;
import java.net.URISyntaxException;
import jdk.incubator.http.*;
import java.security.KeyFactory;
import java.security.KeyStore;
import java.security.KeyStoreException;
import java.security.NoSuchAlgorithmException;
import java.security.PrivateKey;
import java.security.SecureRandom;
import java.security.Security;
import java.security.UnrecoverableKeyException;
import java.security.cert.Certificate;
import java.security.cert.CertificateException;
import java.security.cert.CertificateFactory;
import java.security.cert.X509Certificate;
import java.security.spec.InvalidKeySpecException;
import java.security.spec.PKCS8EncodedKeySpec;
import java.io.File;
import java.io.FileInputStream;
import java.io.FileReader;
import java.io.IOException;
import java.io.InputStream;
import javax.net.ssl.KeyManager;
import javax.net.ssl.KeyManagerFactory;
import javax.net.ssl.SSLContext;
import javax.net.ssl.SSLSocketFactory;
import javax.net.ssl.TrustManager;
import javax.net.ssl.TrustManagerFactory;
import
  org.bouncycastle.jce.provider.BouncyCastleProvider;
import org.bouncycastle.util.io.pem.PemObject;
import org.bouncycastle.util.io.pem.PemReader;
```

The `TLSContextHelper` class declares many static methods that will allow me to work with TLSv1.2. It is about 150 lines, so you need to download it from *Java Magazine*'s download area. The methods of importance in this code are

- `getKeyFactoryInstance`: Returns a `java.key.KeyFactory` instance that converts public and private keys of the RSA algorithm.
- `createX509CertificateFromFile`: Takes a certificate filename, loads the file contents, generates an instance of X509Certificate from the file, and returns the instance.
- `createPrivateKeyFromPemFile`: Takes a key filename in the PEM format, loads the file contents, generates an instance of `java.Security.PrivateKey`, and returns the instance.
- `createKeyManagerFactory`: Takes a certificate filename, a client key filename, and a client key password; and calls the `createX509CertificateFromFile` and `createPrivateKeyFromPemFile` methods. Subsequently, the code uses the instances returned by these methods to create an instance of the `java.net.ssl.KeyManagerFactory` class, which it then returns.
- `createTrustManagerFactory`: Takes a certificate authority certificate filename and calls the `createX509CertificateFromFile` method. The code uses the X509Certificate instance, and creates and returns an instance of the `java.net.ssl.TrustManagerFactory` class.
- `createAndInitTLS12Context`: Takes a certificate authority certificate filename, a client certificate filename, and a client key filename, and creates and initializes an `SSLContext` instance with the desired TLS version (TLSv1.2). The code uses the `BouncyCastleProvider` and calls the previously explained `createKeyManagerFactory` and `createTrustManagerFactory` methods.

The `certificatesPath` variable in the following code declares a base path for the certificates that you need to run this example. You should replace the contents of this string with the path in which you have a certificate authority certificate file, a client certificate file, and a client key file. You must use files that will be compatible with the HTTP request you are going to process. I use a Windows path, `D:\JavaMagazine\http2`, as an example. However, the code that defines the variables that specify the filenames for the certificate authority certificate, the client certificate, and the client key are compatible with any platform in which you are running the code, such as Linux, Oracle Solaris, or macOS. The code calls `String.join` and uses `java.io.File.separator` to build the filenames by combining the previously explained path with filenames. Make sure you replace `ca.crt`, `server.crt`, and `server.key` with the appropriate filenames. The last line creates and initializes the `SSLContext` that I will use with the `HttpClient` to work with h2.

```
String certificatesPath = "D:\\JavaMagazine\\http2";
String caCertificateFileName =
        String.join(java.io.File.separator,
                certificatesPath,
                "ca.crt");
String clientCertificateFileName =
        String.join(java.io.File.separator,
                certificatesPath,
                "server.crt");
String clientKeyFileName =
        String.join(java.io.File.separator,
                certificatesPath,
                "server.key");
SSLContext sslContext =
        SecurityHelper.createAndInitSSLContext(
                caCertificateFileName,
                clientCertificateFileName,
                clientKeyFileName);
```

The following lines build an HttpClient instance named client by chaining many method calls after the call to the HttpClient.newBuilder method that creates a new HttpClient builder. Some parts of the code are similar to the previous examples. However, in this case, notice that the call to the sslContext method (with the previously created SSLContext instance named sslContext as an argument) makes it possible to work with HTTP/2 over TLSv1.2. Then, the call to the version method with HttpClient.Version.HTTP_2 as an argument forces the use of HTTP/2, specifically, HTTP/2 over TLSv1.2 because I chained the call to SSLContext, too.

```
HttpClient client = HttpClient.newBuilder()
       .sslContext(sslContext)
       .version(HttpClient.Version.HTTP_2)
       .followRedirects(HttpClient.Redirect.ALWAYS)
       .build();
System.out.println(client.version());
```

```
URI uri =
  new URI("https://your-rest-api-url-for-get-method");
HttpRequest request = HttpRequest
    .newBuilder()
    .uri(uri)
    .GET()
    .build();
CompletableFuture<HttpResponse<String>> response =
    client.sendAsync(request,
      HttpResponse.BodyHandler.asString());
response.whenComplete((HttpResponse<String> response,
      Throwable exception) -> {
      if (exception == null) {
            System.out.println(
                  String.format("Status code: %d",
                        response.statusCode()));
            System.out.println(String.format(
                  "Body length: %d",
                  response.body().length()));
      } else {
            System.out.println(String.format(
                  "Something went wrong. %s",
                  exception.getMessage()));
      }
});
```

After the HttpClient instance is built, the next line prints the result of calling the client.version() method. JShell displays HTTP_2 because I forced the usage of HTTP/2. In this case, you have to replace https://your-rest-api-url-for-get-method with the URI for your REST API that allows a GET method and provides a response with HTTP/2 over TLSv1.2. Remember that you are using certificates, so any wrong certificate will make the TLS handshake (reported as an SSL handshake in the old Java names for exceptions) fail. With just a few additional lines, the code performs an HTTP GET request with an asynchronous execution, as done in the previous examples. However, in this case, if the REST API is compatible with HTTP/2 over TLSv1.2, the `HttpClient` will work with this protocol instead of HTTP/1.1.

```
HttpClient client = HttpClient.newBuilder()
    .sslContext(sslContext)
    .version(HttpClient.Version.HTTP_2)
    .followRedirects(HttpClient.Redirect.ALWAYS)
    .build();
System.out.println(client.version());
URI uri = new URI("https://your-rest-api-url-for-get-
method");
HttpRequest request = HttpRequest
    .newBuilder()
    .uri(uri)
    .GET()
    .build();
CompletableFuture<HttpResponse<String>> response =
    client.sendAsync(request,
      HttpResponse.BodyHandler.asString());
response.whenComplete((HttpResponse<String> response,
      Throwable exception) -> {
      if (exception == null) {
            System.out.println(
                  String.format("Status code: %d",
                        response.statusCode()));
            System.out.println(String.format(
                  "Body length: %d",
                  response.body().length()));
      } else {
            System.out.println(String.format(
                  "Something went wrong. %s",
                  exception.getMessage()));
      }
});
```

Conclusion

These simple examples demonstrated how to use the new incubator module HTTP/2 client with HTTP/2 over TLS in JShell. The module provides many additional features that you can also use from JShell and that make code easy to read. For software development tasks, you will find the new HTTP/2 client to be extremely useful, especially when you need to work with an interactive REPL such as JShell. However, the current disadvantage is that the module is included as an incubator and is subject to change. Nonetheless, if you can tolerate some API changes, the benefits of the new features justify exploring the technology. </article>

Gastón Hillar (@gastonhillar) has been working as a software architect with Java since its first release. He has twenty years of experience designing and developing software and is the author of many books related to software development, hardware, electronics, and the Internet of Things. Hillar has been awarded the Intel Black Belt Software Developer award eight times.

learn more

[Home page for HTTP/2]
[JEP 110: HTTP/2 Client (Incubator)]
[JEP 222: jshell: The Java Shell]
[RFC 6455 (WebSocket Protocol)]
[Bouncy Castle Crypto APIs]

SIMON ROBERTS

Quiz Yourself

Intermediate and advanced test questions

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 material 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.

Let me re-emphasize that these questions rely on Java 8. I'll begin covering Java 9 in a future column and will make that transition quite clear when it occurs.

Question 1 (intermediate). Given this code:

```
interface ParentIF {}
interface ChildIF extends ParentIF {}
interface OtherIF {}
class ParentCL {}
class ChildCL extends ParentCL {}
class OtherCL {}
```

and this code:

```
ChildIF cI = null;
ParentIF pI = null;
OtherIF oI = null;
ChildCL cC = null;
ParentCL pC = null;
OtherCL oC = null;
cI = (ChildIF)oI; // line n1
cC = (ChildCL)pC; // line n2
cC = (ChildCL)oC; // line n3
cI = (ChildIF)oC; // line n4
```

Which is true? Choose one.

a. Line n1 and line n3 both fail to compile.
b. The casts are unnecessary in both line n1 and line n3.
c. Line n3 fails to compile.
d. The casts are unnecessary in both line n2 and line n4.
e. Line n4 fails to compile.

Question 2 (intermediate). Given the following:

```
class P {
  private int value;
  // line n1
  public P(int v) {
    value = v;
  }
}

class S extends P {
  private int value;
  // line n2
  public S(int v, int u) {
    // line n3
    value = u;
  }
}
```

Which of these statements is true? Choose one.

- **a.** The code compiles without errors.
- **b.** Adding the following at line n2 allows the code to compile without errors: `public S(int v) {}`.
- **c.** Adding the following at line n3 allows the code to compile without errors: `this(v);`.
- **d.** Adding the following at line n3 allows the code to compile without errors: `super(v);`.
- **e.** Adding the following at line n1 allows the code to compile without errors: `private P(){}`.

Question 3 (advanced). You are writing a program that needs to respond to changes in a directory, such as a new version of a file being written or a file being added or deleted.

Which of these features from the standard Java SE APIs would you use in addressing this requirement? Choose one.

- **a.** `java.nio.file.Path`
- **b.** `java.nio.file.Files`
- **c.** `java.nio.file.FileVisitor`
- **d.** `java.nio.channels.AsynchronousChannel`
- **e.** `java.io.File`

Question 4 (advanced). Given the following code:

```
public static void delay() {
    try { Thread.sleep((int) (Math.random() * 10)); }
    catch (InterruptedException ie) {}
}
public static void main(String[] args) {
    int[] x = {0};
    boolean[] hold = {true};
    new Thread(() -> {
        delay();
        x[0] = 99;
        hold[0] = false;
    }).start();
    new Thread(() -> {
        delay();
        while (hold[0])
            ;
        System.out.println("value is " + x[0]);
    }).start();
}
```

And, choosing from these behaviors:

1. The program prints `value is 0`.
2. The program prints `value is 99`.
3. The JVM exits (the program stops).
4. The JVM does not exit (the program does not stop).

Which describes all the outcomes allowed by the specification? Choose one.

- **a.** 2 and 3
- **b.** Neither 1 nor 2, and 3
- **c.** Either 1 or 2, and 3
- **d.** Either 2 and 3; or neither 1 nor 2, and 4
- **e.** 3 and either 1 or 2; or neither 1 nor 2, and 4

Question 1. The correct answer is option C. This question examines some aspects of assignment compatibility.

As a summary, the compiler recognizes three situations during an assignment. First, the expression on the right has an is-a relationship with the type of the expression being assigned to. The expression could be an exact instance or an instance of a class that is a subclass of the class that's the target of the assignment, or it could be an implementation of the interface that's the target of the assignment. In this situation, the assignment works directly and no cast is necessary.

The second situation is where the expression on the right might refer to something that is assignment-compatible, but it's not definite. This happens, for example, when the expression on the left is a subclass of the expression on the right. It also happens in many situations where interfaces are involved. In the latter case, the compiler allows the assignment only if a cast is used, and it lets the runtime system verify the validity of the actual usage.

If a constructor does not explicitly state how to pass control to a superclass constructor, the compiler implicitly generates code in the constructor that invokes the zero-argument constructor of the parent class.

The third situation is where the compiler can prove that the assignment cannot possibly work, such as with classes that don't share any class hierarchy. In this situation, the compiler rejects the assignment even if a cast is used.

None of the options in this question assigns from something that has an is-a relationship to a more general type in the way that does not require a cast. Therefore, if any of the casts is removed, what's left is not a compilable assignment. Because of this, options B and D are both incorrect.

Given that every case has an explicit cast, all that remains is to determine whether the assignment is plausible and, therefore, whether it will be permitted by the compiler. Let's look at these one at a time.

Line n1 takes a reference to an `OtherIF` and attempts to cast it to a `ChildIF`. There's no relationship between these interface types, but what matters to the compiler is whether it's possible that the reference being cast might actually implement the `ChildIF` interface. In general, such a cast to an interface type is plausible. For example, imagine this additional class definition exists:

```
class BothImpl implements ChildIF, OtherIF {}
```

Now suppose the variable `oI` refers to an instance of `BothImpl`. That's possible, because `BothImpl` has an is-a relationship to `OtherIF`. In that case, the cast would succeed when executed. Because a scenario is possible in which the cast would succeed, the compiler allows the code, deferring to the runtime system to determine if it actually works. This tells you that option A is false, because line n1 compiles (even though I've not discussed line n3 yet).

The code in line n2 takes a reference of `ParentCL` type and casts it to `ChildCL`. This is always plausible, simply because the `ChildCL` type is assignment-compatible with the `ParentCL` type. Because you can assign `pC = cC` without any cast, this means that the opposite, `cC = (ChildCL) pC`, while needing the cast, is definitely plausible, because you'd simply be putting the original value back into `cC`. Therefore, `line n2` is definitely compilable, even though it doesn't move the search for the right answers forward—option D was already rejected, because the cast is necessary.

Line n3 attempts to cast a reference to an instance of `OtherCL` to a `ChildCL`. This is not plausible. A simple instance of `OtherCL` is an `OtherCL` and is an `Object` but, because Java doesn't permit multiple class inheritance, it's not possible to have any object that has both `OtherCL` and `ChildCL` as parents. Because of this, line n3 does not compile and option C is correct.

Line n4 takes a reference to `OtherCL` and casts it to `ChildIF`. Similar to the consideration in line n1, this is plausible. Just imagine that you have another declaration:

```
class AnotherImpl extends OtherCL implements ChildIF {}
```

Clearly it's possible then to have the oC reference point at one of these objects, and that would successfully cast to the ChildIF type as required. As a result, option E is incorrect.

As a side note, if the definition of OtherCL were final, guaranteeing there could be no subclasses of OtherCL, then line n4 would fail to compile, because no scenario like the one outlined above would be possible.

Question 2. The correct answer is option D. This question investigates the initialization of parent class elements that occurs as part of the initialization of subclasses. Java takes considerable pains to ensure that things are properly initialized. (It's not always possible to enforce that, but it's a clear goal.)

Three particular behaviors are relevant to this question. First, any time a subclass is instantiated, it must call up to a specific parent class constructor, passing the required arguments, so that the parent has a chance to be properly initialized.

Second, in the unique situation that the source code for a class does not define any explicit constructor, the compiler creates one by default. That constructor takes zero arguments, and it calls the zero-argument constructor of its parent class, the superclass, even if that class is Object.

Third, if a constructor does not explicitly state how to pass control to a superclass constructor, the compiler implicitly generates code in that constructor that invokes the zero-argument constructor of the parent class. This rule means that the constructor shown for S is equivalent to this:

```
public S(int v, int u) {
    super();
    value = u;
}
```

In this example, the child class S—as written—has no explicit call to any constructor in the parent class P, so it will invoke the zero-argument constructor of the parent. However, P *has* an explicit constructor that takes a single argument. Because of this, the child constructor cannot compile, because it implicitly calls a (nonexistent) zero-argument constructor in P. Therefore, option A is false.

There are two likely solutions to the problem. Either you provide the parent class P with a suitable zero-argument constructor to accept the child's call, or you modify the child constructor to explicitly call the constructor that does exist in the parent class. It could call the existing constructor, or you could make another constructor in P with different arguments, and call that explicitly from S. None of the options proposes that latter route, so you can ignore it.

Option B adds a single-argument constructor to the child, but that doesn't actually help. The provided constructor still has no explicit constructor call; therefore, a zero-argument constructor is still required *in the parent*. Adding such a constructor to the child does not help this situation. Note that mere matching of constructor signatures—that is, matching the argument lists—has no value here. Therefore, option B is false.

Option C is actually valid syntax, but it is incorrect for solving the problem at hand. The use of `this(v);` would try to delegate to a constructor that takes one argument, but that target constructor would have to be in the same class, not the parent. Because of this, option C is false. Also, note that constructors, unlike instance methods, are *not* inherited, so there really is no target for the call proposed here.

Option D is the correct syntax for invoking the single-argument parent class constructor. It satisfies the need for the single argument by passing the value of v up to the parent class constructor. From there, that value will be stored in the private member variable value that is a member of P. Because of this, option D results in successful compilation, and it is the correct answer.

Option E might look tempting, creating a zero-argument constructor in the parent, but the constructor in this option is private. A private zero-argument constructor would not be accessible to the child class and, for this reason, it does not solve the essential problem: the constructor in S tries to call a zero-argument constructor in P. S must have access to the target constructor, and the private form fails in that respect. Because of this, option E is incorrect. A private constructor might seem strange, but it can be very useful for taking control of how objects are created, for example, in implementing the singleton, static factories, and builder patterns.

Programmers need a mental model of how a computer works that allows reasoning about their daily work.

It's also interesting to consider the private fields called `value` that exist in both `P` and `S`. Is this a really bad idea? Are they actually the same field? In fact, neither is true. Because the fields are private, there's no naming collision; each is visible only within the class that contains it. In effect, they're not visible outside their classes, so nobody can be affected by them, nor can the names collide with any other variable of that name. Also, they really do define separate fields; they're not somehow the same field. Therefore, while it might look odd when you are looking at both classes at once, this code is actually fine and you wouldn't even notice it if you didn't see both pieces of source code at once. However, if these fields were accessible outside the class—for example, if they allowed default access—the code would be pretty horrible. Such a situation is typically called *shadowing*, or sometimes variable *hiding*, and although it's manageable with careful syntax use, allowing it is just asking for maintenance trouble and can cause all kinds of misunderstandings.

Question 3. The correct answer is option A. This is almost one of those troublesome "learn the API" questions, except that, in this question, you don't have to learn method names; you just have to learn about the facilities the API can offer. That's actually not a waste of time, because failing to learn such things often results in duplicating behavior that has already been provided for you. On that basis, I make no apology for this question.

So, what do these various features do? First, the `java.nio.file.Path` interface is the modern way for a Java program to represent "path and filename" as they relate to disk storage. This formerly was done using the `java.io.File` class, but that class lacked expressive power for representing less-universal features of file systems, such as permissions. Notably, `Path` is an interface, allowing entirely independent implementations on different operating systems or even file system types, whereas `File` is a class, which made such flexibility harder to achieve. Along with many handy and predicable features for interacting with the path segments, and for dealing with relative and absolute paths, the `Path` interface defines two methods called `register`, which allow code to determine easily when changes are made to a file system. Given this description, it's clear that option A is the correct answer. It's also safe to infer that option E is incorrect, because the `File` class is of limited functionality and it is considered to be a legacy feature.

The `Files` class is a container for static methods that perform useful operations such as copying and moving files, creating and deleting files and directories, reading and manipulating permissions, and traversing directory trees. The class also provides utility methods that can simplify access to the data in files, for example, the `lines` method that reads a text file as a `Stream<String>` directly. Although this is a very useful class—well worth looking at, if you haven't already—it's not directly suitable for solving the problem of watching for change in a directory structure, and so option B is incorrect.

The `FileVisitor` interface is used with features such as the `Files.walkFileTree` method. The visitor is used to perform operations on some or all of the files and directories in a directory tree. The interface can, therefore, examine the contents of directories, but it does so on a one-time basis, and it is not directly suited to looking for changes. Therefore, option C is incorrect.

The `AsynchronousChannel` interface is the base interface from which some interesting classes are derived indirectly. These interesting classes allow "callback" type asynchronous I/O operations, which are potentially valuable in highly concurrent systems that seek to minimize the number of threads they use. However, this capability has no direct relationship with the issue of monitoring changes in a directory. Therefore, option D is incorrect.

Finally, it's fair to point out that although the `Path` interface defines the `register` method that facilitates monitoring changes on the file system, it's not sufficient on its own. You'll also need to pass an instance of `WatchService` to that method. You can get an instance of `WatchService` from a static factory method: `FileSystem.newWatchService()`.

Question 4. The correct answer is option E. This question delves into one of the most commonly misunderstood areas of Java's specification—the memory model—and what it means for the behavior of threaded code. Two particular points affect the behavior of the given program, and they are permitted by the specification: the visibility of written data and the perceived ordering of the write operations. Notice that one of these points is unlikely to be manifested if you run the code, but that doesn't mean the answer I give is incorrect; it just means your system doesn't happen to behave in this way.

Programmers need a mental model of how a computer works that allows reasoning about their daily work. Generally, the kind of model that serves well on a daily basis is a fairly significant simplification and can get you into trouble where concurrency is concerned. Hardware engineers have some radical tricks up their sleeve to make their processors perform faster, and Java needs to allow every host it runs on to use as many of these tricks as possible to ensure good performance. Consequently, Java's specification is not written in terms of implementations but in terms of what memory effects can be relied upon. The specification uses an idea called a *happens-before relationship* to allow reasoning about which data written by one thread must be visible to another thread and when that data must be visible.

Perhaps strangely, a happens-before relationship does not actually say that one thing happens before another, at least not in the sense you would expect. It does not really relate to, nor does it actually mandate, execution order; it relates only to particular visibilities of an effect. That's perhaps surprising, but one rationale for this is that optimizing compilers have for decades indulged in the reordering of instructions. Reordering can be safe in particular situations. Let's look at a simple example:

```
double x = heavyComputation();
double y = otherComputation();
if (x > 3) doSomethingWith(x);
```

Notice that changing the order of the first two lines would make no difference to the result, but it's possible that a compiler might generate more-efficient code as a result of this reordering. It would be able to execute `otherComputation` and store the result, and then perform `heavyComputation` and immediately use the value of `x`, rather than storing `x` and having to fetch it back again for the third operation. That might seem like a small improvement, but many small improvements can add up to a large improvement.

What does a happens-before relationship tell us? If A happens before B, *and* B reads something written by A, the happens-before relationship tells you that B will read the value written by A. Importantly, unless both the happens-before relationship and the "observation" exist, no guarantees are given. Further, just because the happens-before relationship exists in one context (for example, between two threads T1 and T2), that guarantees nothing about the visibilities of effects elsewhere where no such relationship exists (for example, a third thread T3). As you would expect, the happens-before relationship is created by the order of the code lines executed by a single thread, and happens-before relationships are transitive. Therefore, if A happens before B and B happens before C, then A happens before C. But that simple line-by-line expectation holds only for a single thread, although the transitive effect is not limited to single threads. Special steps must be taken to create any necessary relationship across threads.

In the code example in the question, nothing is done to create any happens-before relationships between the two threads that are created and started inside the `main` method. This means that the second thread—the one doing the reading—might or might not see the change from 0 to 99 in the array `x`. Similarly, it might or might not see the change from `true` to `false` in the array `hold`. It might see both changes or it might see neither, but it also might see either one without seeing the other one. (That often comes as a surprise to the developer.) Therefore, the second thread might see the 99, yet not see the change to `false`, or it might see the change to `true`, but not see the change to 99. That second possibility, which results in the program printing `value is 0` and then exiting, is often a bit unexpected. (To be fair, it's also a fairly unlikely result in typical computation hardware, but the specification allows it, which means that this is a bug in the code and that's not OK.)

Therefore, let's consider the possible outcomes. What matters is what the second thread sees. If it sees the boolean value change, it prints something and the program exits. But, if it doesn't see the boolean change, it never prints and it doesn't stop. That allows you to eliminate options A, B, and C, all of which fail to acknowledge that the program might never stop.

To evaluate the last two possibilities, you must consider the value that will be printed if the second thread sees the change in the boolean. You now know that it could correctly print either `value is 0` or `value is 99`. Therefore, option D, which suggests that if it stops, it must have printed `value is 99`, but admits that it might not stop at all, is incorrect. Option E, which admits that it could either print a value and stop or print nothing and not stop, is the correct answer.

It's likely the notion that "the specification allows this" is a bit unsatisfying, so let's look at one way this might actually happen in real hardware. But first, it's important to realize that trying to reason about Java memory behavior in terms of hardware implementation is dangerous and often misleading. The *only* reliable approach is to consider the memory model and reason using that. Anyway, just to quiet that nagging "but that could never happen" feeling, here's one possible way it might happen on real hardware. First, suppose that both the threads created in the `main` method start running on their own CPUs, and those CPUs have independent caches (which is not unreasonable). Now, imagine that the writing thread runs, and it writes both 99 and `false` to its cache, not to main memory. Now, given that no happens-before relationship has been enforced between the threads, neither the underlying hardware nor the JVM has any obligation to maintain any kind of cache coherence between those threads—and whether those values end up in main memory or get moved to the other CPU's cache is undefined. Therefore, imagine that the boolean value happens to be flushed out of the cache, before the 99. (Caches usually flush a row at a time, and these two data items might happen to be in different rows. Then it would not be unreasonable that they could be flushed in the opposite order from the order in which they were written.) In this situation, write operations that were executed "in order" by one thread can be seen as occurring out of order by another thread. That is sufficient to explain why the memory model is the only safe way to reason about what does, and does not, constitute reliable code in Java.

It's probably fair to discuss briefly how this code could be made to behave in the expected way. I venture to suggest that the expected behavior—that is, the behavior implied by the source code—is that it should print `value is 99` and then stop. To achieve this, you must create a happens-before relationship between the write to `hold[0]` by the first thread and the read from it by the second thread.

One simple way to create a happens-before relationship between two threads depends on the fact that a write to a volatile variable happens before a subsequent read from that same variable. It's probably tempting, then, to modify the `hold` variable to be volatile. But that would not work, because the modifier would relate to the variable, and that variable is a *reference* to the array, not the storage in the array. In other words, you never write to the volatile variable itself. You also cannot convert the array to a simple variable, because the access to the value from the nested Runnable instances mandates that the variable be effectively final. You could, however, make `hold` into a simple static field in the enclosing class and then label that static field as volatile. That's not the only way, of course, but it would be sufficient to solve this particular problem. `</article>`

Simon Roberts joined Sun Microsystems in time to teach Sun's first Java classes in the UK. He created the Sun Certified Java Programmer and Sun Certified Java Developer exams. He wrote several Java certification guides and is currently a freelance educator who teaches at many large companies in Silicon Valley and around the world. He remains involved with Oracle's Java certification projects.

ORACLE

Java Is Just the Beginning

Your Java applications need high-performance and battle-tested platform and infrastructure services to build, test, deploy, and monitor. Oracle Cloud delivers.

Start with Java in the cloud—or choose whatever language, database, compute service, and OS option you need. Rapid scalability. True portability. It's all here. Now.

Oracle Cloud.
Built for modern app dev.
Built for you.

Start here:
developer.oracle.com

#developersrule

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 it 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.

- Subscription application
- Download area for code and other items
- *Java Magazine* in Japanese