// MARCH/APRIL 2014 /

Java™ magazine

By and for the Java community

JAVA EXPLORE THE POSSIBILITIES

IN THIS ISSUE: Lambda Expressions / Nashorn / Date and Time / Embedded Java

ORACLE.COM/JAVAMAGAZINE

ORACLE

COVER ART BY I-HUA CHEN

EDITORIAL

Editor in Chief
Caroline Kvitka

Community Editors
Cassandra Clark, Sonya Barry, Yolande Poirier

Java in Action Editor
Michelle Kovac

Technology Editors
Janice Heiss, Tori Wieldt

Contributing Writer
Kevin Farnham

Contributing Editors
Claire Breen, Blair Campbell, Karen Perkins

DESIGN

Senior Creative Director
Francisco G Delgadillo

Senior Design Director
Suemi Lam

Design Director
Richard Merchán

Contributing Designers
Jaime Ferrand, Arianna Pucherelli

Production Designers
Sheila Brennan, Kathy Cygnarowicz

PUBLISHING

Vice President
Jeff Spicer

Publisher
Jennifer Hamilton +1.650.506.3794

Associate Publisher and Audience Development Director
Karin Kinnear +1.650.506.1985

ADVERTISING SALES

President, Sprocket Media
Kyle Walkenhorst +1.323.340.8585

Western and Central US, LAD, and Canada, Sprocket Media
Tom Cometa +1.510.339.2403

Eastern US and EMEA/APAC, Sprocket Media
Mark Makinney +1.805.709.4745

Advertising Sales Assistant
Cindy Elhaj +1.626.396.9400 x 201

Mailing-List Rentals
Contact your sales representative.

RESOURCES

Oracle Products
+1.800.367.8674 (US/Canada)

Oracle Services
+1.888.283.0591 (US)

Oracle Press Books
oraclepressbooks.com

ARTICLE SUBMISSION

If you are interested in submitting an article, please e-mail the editors.

SUBSCRIPTION INFORMATION

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

MAGAZINE CUSTOMER SERVICE

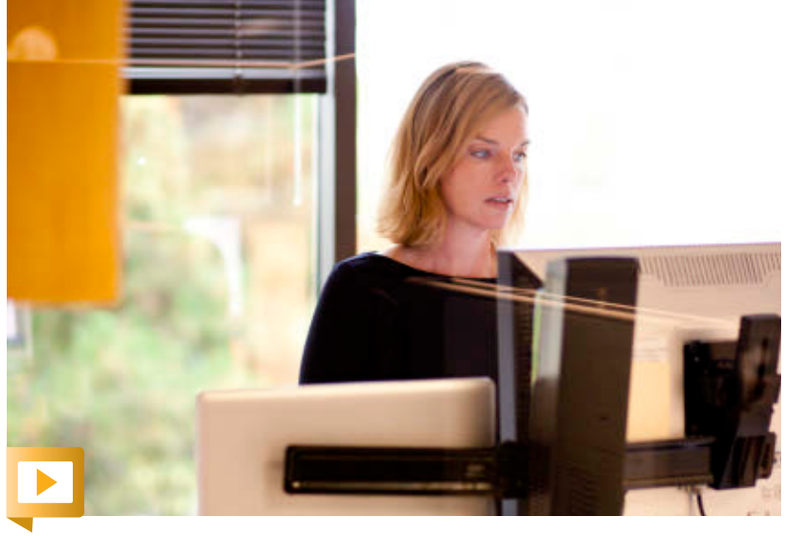
java@halldata.com **Phone** +1.847.763.9635

PRIVACY

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

Java Magazine is published bimonthly with a free subscription price by Oracle, 500 Oracle Parkway, MS OPL-3C, Redwood City, CA 94065-1600.

Digital Publishing by GTxcel



Change is good—when it comes with benefits. In the case of Java 8, those benefits come in the form of more productivity for developers plus expanded opportunities to develop applications that span from embedded devices to the cloud. Java 8 is here—you can download Java SE 8 today, while Java ME 8 is available in Early Access form. We've dedicated most of this issue to Java 8—just look for the theme icon for articles on this topic.

Java 8 Is Here

The biggest change in Java SE 8—what some have called the most significant upgrade to the Java programming language ever—is lambda expressions, or closures. "Lambda expressions are anonymous methods that provide developers with a simple and compact means for representing behavior as data," explains Brian Goetz, Java language architect at Oracle. "In a few years, developers will wonder how they ever lived without [them]."

There's plenty more to get excited about, including Nashorn, a next-generation JavaScript engine; Compact Profiles, which offer a convergence of the Java ME Connected Device Configuration (CDC) product with Java SE 8; and a new date and time API. You, the Java community, also played a huge role in bringing Java 8 to fruition. We examine all that and more in our Java 8 cover story.

In "Java ME 8 and the Internet of Things," we explore the new features that make Java ME 8 a comprehensive platform for connected devices. And if you're ready to get going with embedded development, Angela Caicedo shows you how.

This issue is jam-packed with information to get you started with Java 8 (including articles on streams and type annotations). So go on . . . embrace change and explore the possibilities. Let us know where you take Java 8.

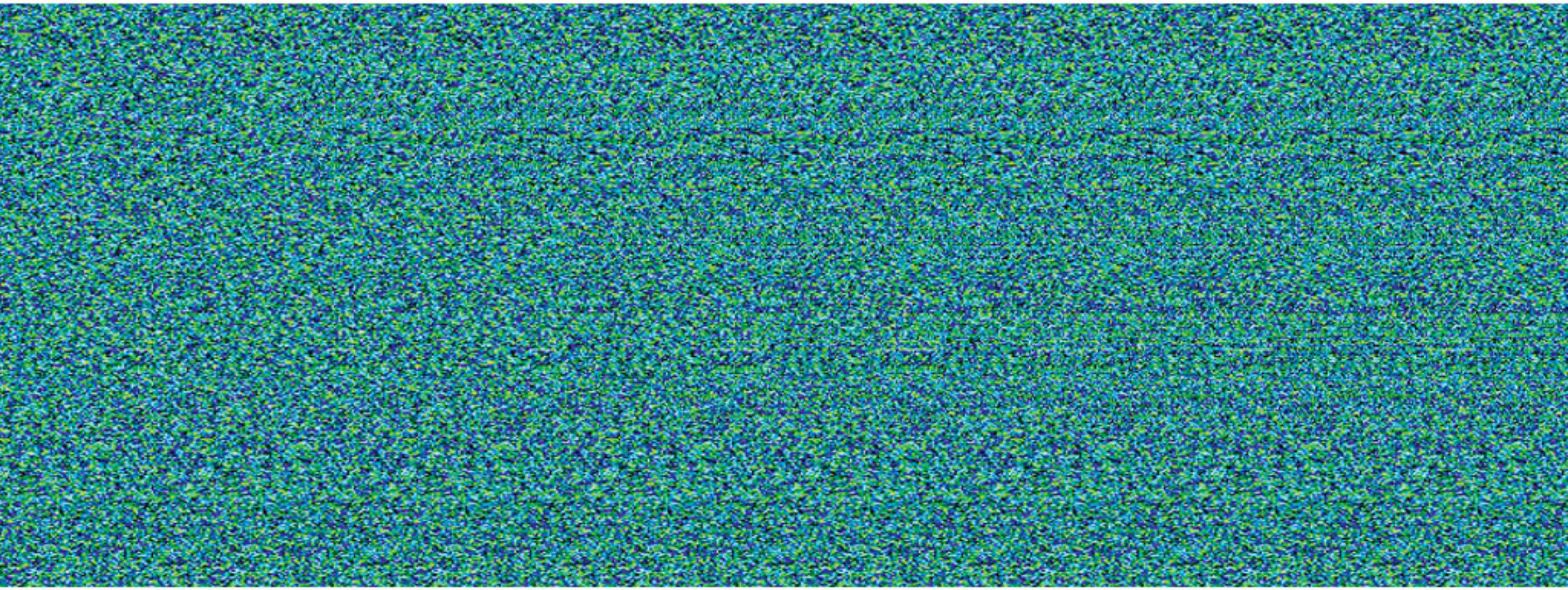
Caroline Kvitka, Editor in Chief BIO

//send us your feedback /

We'll review all suggestions for future improvements. Depending on volume, some messages may not get a direct reply.

PHOTOGRAPH BY BOB ADLER

The answer is right in front of you



Java Image Enabling SDKs that Help You See the Big Picture

At first glance it may seem difficult, but it's really quite simple. Atalasoft's JoltImage product is a proven SDK for image enabling your Java-based web applications, easily. Image enabling helps to add dimension to your data, so you can uncover insights such as correlations and causations hidden inside your 2-dimensional documents. Our SDK does the heavy lifting for you, saving time, money, and the headaches of figuring it out yourself. Backed by our highly knowledgeable & caffeinated support engineers, JoltImage will enable your success and make the big picture so much easier to see.

Atalasoft JoltImage

Click for tips on viewing the stereogram

Image enabling experts & bacon connoisseurs. Visit us online to see our full line of SDK products for .NET and Java

Atalasoft
A Kofax Company
www.atalasoft.com

From embedded to the cloud, get there with Java 8.

BY TIMOTHY BENEKE

ART BY I-HUA CHEN

1. LAMBDA EXPRESSIONS
2. NASHORN AND JAVASCRIPT
3. COMPACT PROFILES
4. DATE AND TIME API
5. JAVAFX 8
6. COMMUNITY INVOLVEMENT
7. JAVA ME 8
8. JAVA SE 8 AND THE JVM

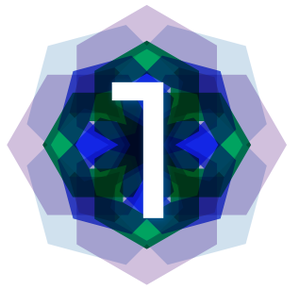
Java 8 Is Here

Java 8, which encompasses both Java SE 8 and Java ME 8, might be the most significant expansion of the Java platform yet. Lambda expressions and the Stream API increase the expressive power of the platform and make it easier for developers to take advantage of modern, multicore processors. Compact Profiles in Java SE 8 allow developers to use just

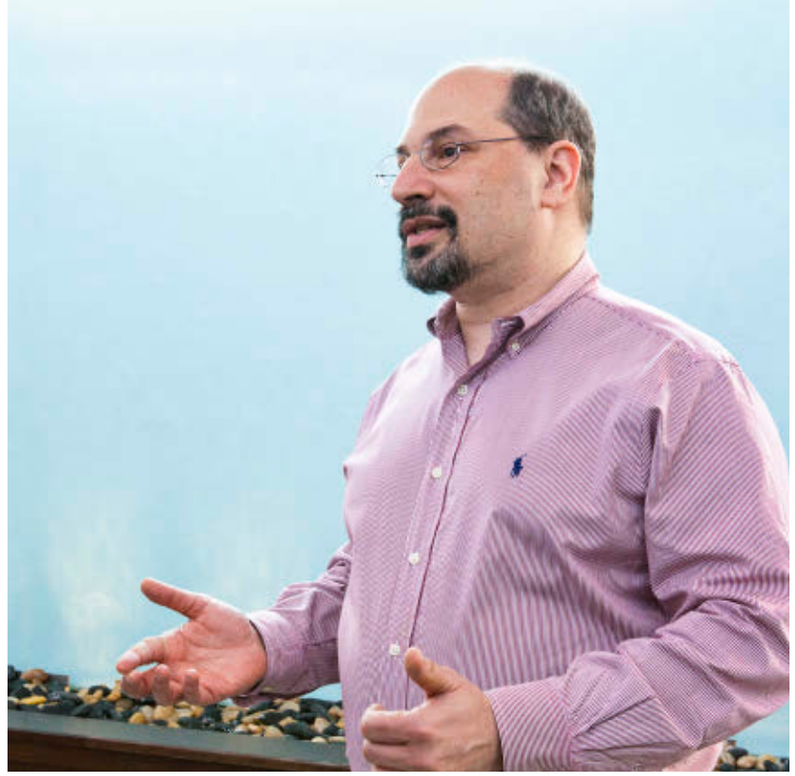
a subset of the platform, and are a significant step toward the convergence of Java SE and Java ME. Java ME 8 itself, meanwhile, has been updated to include more-recent Java Virtual Machine (JVM), language, and library features while retaining the focus on small embedded devices. Java 8 allows developers to apply the same skill set across a wide range of scenarios, from the smallest embedded Internet of Things (IoT) devices to enterprise servers in the cloud.

As Mark Reinhold, chief architect of the Java Platform Group at Oracle, put it, "Java 8 is the first truly revolutionary release we've done in a very long time."

Here, we take a look at eight key pieces of Java 8.



LAMBDA EXPRESSIONS



Lambda expressions are at the heart of Java SE 8. "Lambda expressions, also known as closures, are anonymous methods that provide developers with a simple and compact means for representing behavior as data," explains Brian Goetz, Oracle's Java language architect and Specification Lead for JSR 335, Lambda Expressions for the Java Programming Language. "This enables the development of libraries that do a better job of abstracting over behavior, which in turn leads to more-expressive, less error-prone code."

While Java has always provided reasonable tools such as classes, inheritance, and generics for abstracting over data, Java SE 8 provides new tools for abstracting over behavior. As Goetz explains, "If we want to model a workflow such as 'do A before you start, do B for every file in this group, do C if you encounter an error, and do D when you're done,' we don't have ideal tools for expressing the behaviors A through D, and this affects the sort of APIs we design. We have to break up the phases of the workflow, and the client code has to be directly involved in each phase, rather than saying 'here's what I want; go do it.' This negatively affects reusability, readability, and performance."

According to Goetz, the typical developer's initial experience with lambda expressions is likely to be through the powerful new APIs for manipulating collections. Business logic typically is full of ad hoc query-like calculations over collections; the core libraries now make it easier to express the "what" of such a query without getting bogged down in details about the "how."

> "This is the most significant upgrade of the Java programming model ever. In a few years, developers will wonder how they ever lived without it. I certainly do!"
>
> —Brian Goetz, Java Language Architect, Oracle

The other major language feature of Java SE 8 is default methods, which

PHOTOGRAPH BY TIM GRAY/GETTY IMAGES

COMMUNITY SPOTLIGHT
ADOPT-A-JSR

JSR 335: Lambda Expressions for the Java Programming Language

ADOPTED BY: Jozi JUG (Johannesburg, South Africa), London Java Community, and SouJava (São Paulo, Brazil)

WHAT: Jozi JUG organized a Java 8 hackathon and presented a talk on lambda expressions to its members.

London Java Community held three hackdays and presented a "Lambdas: Myths and Mistakes" talk to 140 Java developers.

SouJava held several lambda expressions sessions for its members and evangelized Java 8, Adopt-a-JSR, and the JCP in Brazil, Colombia, and beyond.

Brian Goetz does a live hacking session on lambda expressions.

enable interfaces to evolve over time. Default methods are necessary for Java SE 8 given that some core library classes, such as Collections, are more than 15 years old. Without default methods, some core library classes would be unable to support lambdas.

"More generally," explains Goetz, "if you want libraries to stay relevant over long periods of time, they need to be flexible, so we needed to address this problem while maintaining our commitment to compatibility."

When asked about the challenges Java developers face in working with lambdas, Goetz is optimistic. "The big challenge," he says, "is accepting that this is not simply a matter of new syntax; there are some new concepts that developers will have to learn just to be able to read Java code, even if they do not plan to take advantage of these new features in the code they write. There are also big additions to the core libraries that developers will have to learn—but these costs should be more than offset by increases in productivity and expressiveness."

Goetz makes no effort to hide his enthusiasm for lambdas: "I think this is the most significant upgrade of the Java programming model ever—and yet, it still feels like the Java we know and love," he insists. "In a few years, developers will wonder how they ever lived without it. I certainly do!"

MORE ON LAMBDAS

- Project Lambda
- Lambda Expressions tutorial
- "Looking Ahead to Project Lambda"

EASIER QUERIES AND CALCULATIONS WITH LAMBDAS

Java Language Architect

Brian Goetz shows how the Java core libraries have been enhanced to enable more-query-like calculations over collections.

"Consider something like 'find artists with albums that have fewer than eight tracks.' This is easy enough to write with a for loop, but the resulting code is full of accidental detail about 'how,' which obfuscates the 'what,'" he says. In Java SE 8, it looks like this:

```
Set<Artists> artists =
  albums.stream()
      .filter(album -> album.
getTracks().size() < 8)
      .map(album -> album.
getArtist())
      .collect(toSet());
```

"Here, we've said 'give us the elements of albums, select the ones with fewer than eight tracks, for each of those get the artist, and then collect them into a set.' The code reads like the problem statement, which is good because code that is easy to read also is more likely to be correct," says Goetz.

This shift from hand-written loops to aggregate operations using lambdas and streams inverts the control of the computation. "With the for loop, the client is in control at every step, asking the Iterator for the next element on each iteration," Goetz says. "With lambdas and streams, the library is in control, but still allows the computation to be easily customized by the client."

He goes on to explain, "In the albums calculation above, it might look like we're doing three passes on the data, but in fact the three are fused into a single pass. This is possible only because the client was able to express everything it wanted in one go, parameterizing the steps with various bits of behavior, rather than being involved in every loop iteration."

NASHORN AND JAVASCRIPT

"Nashorn is a part of Java SE 8 that is intended to provide a version of JavaScript that would run as part of the JVM—a more modern version of JavaScript using newer JVM technologies," observes Jim Laskey, engineering lead for multilanguages at Oracle. With Java SE 8, Nashorn is being shipped as a javax.script engine embedded in the JDK, which means that any Java application can now contain components written in JavaScript. JavaScript is a popular language with features similar to Java's features that is ideal for transmitting code across networks. In addition, there is now a command-line tool (jjs) that allows developers to use Nashorn as a scripting tool. Nashorn, which replaces Rhino, also an open source JavaScript engine, is both faster and lighter and integrates JavaScript and Java more tightly.

As a consequence, JavaScript developers can take advantage of the wealth of libraries normally only available to Java developers, and can implement portions of their application in a simpler, lightweight programming language. Developers can now implement runtime dynamic features and bypass build cycles, while users and field engineers can make configuration changes without having to rebuild the application. "Nashorn will allow JavaScript to be used in ways that have not been available before," he says.

Laskey is quick to point out, "Although Nashorn JavaScript is not the same as browser JavaScript, and currently lacks DOM components, it is a full-featured programming language with access to all of the available Java libraries."

MORE ON NASHORN

- Project Nashorn
- "Oracle Nashorn: A Next-Generation JavaScript Engine for the JVM"

PHOTOGRAPH BY DAN CALLIS/GETTY IMAGES

> "Nashorn will allow JavaScript to be used in ways that have not been available before."
> —Jim Laskey, Engineering Lead for Multilanguages, Oracle

"With the introduction of Compact Profiles, we now have a single Java SE standard with all of the productivity that Java is famous for that can truly scale from small embedded IoT devices to large enterprise server installations."

—Bob Vandette, Consulting Engineer, Oracle

3 COMPACT PROFILES

Compact Profiles, three well-defined API subsets of the Java 8 specification, offer a convergence of the Java ME Connected Device Configuration (CDC) with Java SE 8. With full Java 8 language and API support, developers now have a single specification that will support the Java ME CDC class of devices under the Java SE umbrella.

For many years, embedded developers have wanted to allow subsetting of the Java SE platform in order to deploy smaller binaries in their embedded devices. Compact Profiles enable the creation of applications that do not require the entire platform to be deployed and run on small devices. Because Compact Profiles are much smaller than the full Java SE Java runtime environment (JRE), they enable applications to be deployed on platforms with limited storage. The smallest of these runtimes is 11 MB, which is more than four times smaller than the equivalent traditional JRE.

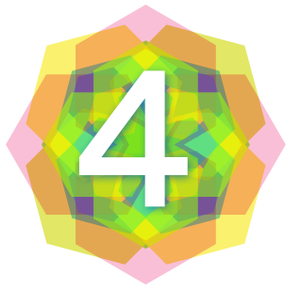
Java SE applications can now be designed to run on resource-constrained devices by targeting the specific APIs that are available in Compact Profiles. The Compact Profiles were primarily designed for embedded developers who wish to use Java for the creation of embedded solutions with limited static and dynamic storage. By keeping the amount of RAM and flash to a minimum, the bill of materials (BOM) cost for embedded devices is also kept to a minimum, resulting in increased profit for device distributors.

Bob Vandette, consulting engineer at Oracle, sums it up: "With the introduction of Compact Profiles, we now have a single Java SE standard with all of the productivity that Java is famous for that can truly scale from small embedded IoT devices to large enterprise server installations."

MORE ON COMPACT PROFILES

- "An Introduction to Java 8 Compact Profiles"

PHOTOGRAPH BY TIM GRAY/GETTY IMAGES



DATE AND TIME API

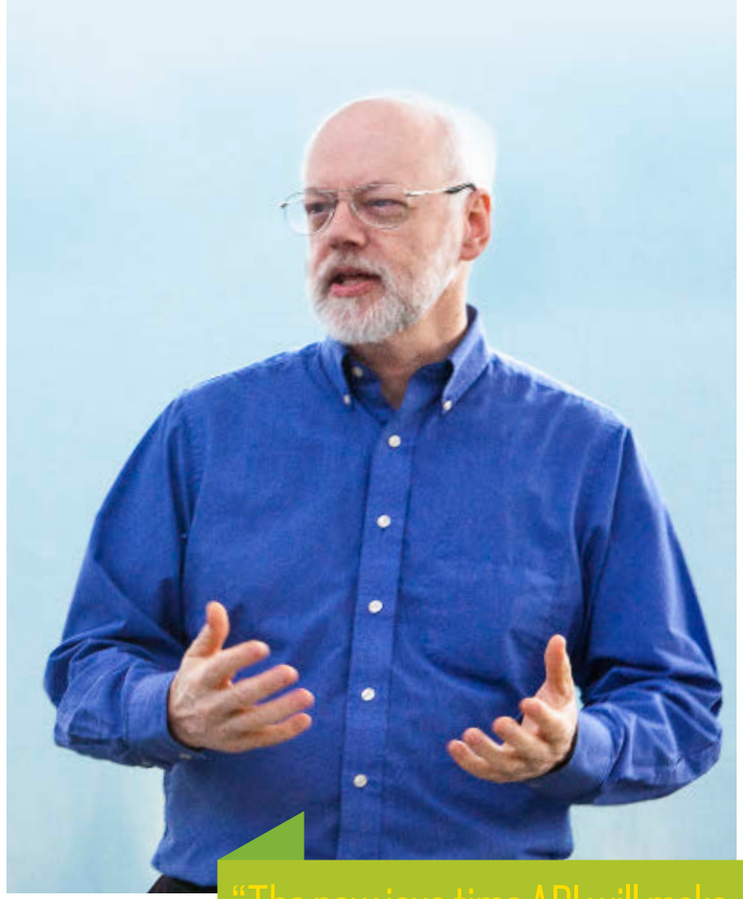
Java SE 8 introduces a new java.time API with a fluent and familiar style that is easy to read and write without the aid of IDEs. It provides excellent support for the international ISO 8601 time standard that global businesses use and also supports the frequently used Japanese, Minguo, Hijrah, and Thai Buddhist calendars. Each of the new core Java classes—for date, time, date and time combined, time zones, instants, duration, and clocks—has a specific purpose and has explicitly defined behavior without side effects. The types are immutable to simplify concurrency issues when used in multitasking environments. In addition, the extensibility of the API enables the addition of new calendars, units, fields of dates, and times. Developers can focus on completing a particular task without being concerned about unrelated functions.

Jim Gough of the London Java Community hacks on date and time.

PHOTOGRAPH BY TIM GRAY/GETTY IMAGES

The new API uses immutable values, with each computation producing a new value. With immutable objects, an instance can be shared with a library or a concurrent task without concern that the value will change unexpectedly, explains Roger Riggs, consulting member of the technical staff at Oracle. This contrasts with the current Date and Calendar types that hold a mutable value and are not multithread safe. Using immutable values eliminates many problems for multithreaded programming.

"The current APIs for Date and Calendar combine many functions in each type, which often results in unexpected interactions that increase programming complexity," explains Riggs. "The new APIs offer greater clarity of purpose, ease of use, and maintenance."



> "The new java.time API will make developers more productive, whether they are using the global date and time standards or one of the regional calendars."
>
> —Roger Riggs, Consulting Member of Technical Staff, Oracle

MORE ON DATE AND TIME

- Package java.time
- "Java SE 8 Date and Time"

COMMUNITY SPOTLIGHT
ADOPT-A-JSR

JSR 310: Date and Time API

ADOPTED BY: Guadalajara JUG (Guadalajara, Mexico), Jozi JUG (Johannesburg, South Africa), London Java Community, and SouJava (São Paulo, Brazil)

KEY CONTRIBUTIONS:

Guadalajara JUG held a live date and time coding session.

Jozi JUG organized a hackathon with more than 25 participants.

London Java Community separated out technology compatibility kit (TCK) tests versus implementation-specific tests.

SouJava held several talks at their meetings and provided feedback to the JSR 310 Expert Group.

Mark Reinhold (left), chief architect of the Java Platform Group at Oracle, and Richard Bair, client Java architect at Oracle, talk about JavaFX 8 and lambdas.

"JavaFX 8 is integrated with Java SE 8 and works well with lambda expressions. It simplifies many kinds of code, such as event handlers, cell value factories, and cell value factories on TableView."
—Richard Bair, Client Java Architect, Oracle

5 JAVAFX 8

"JavaFX 8 is integrated with Java SE 8 and works well with lambda expressions. It simplifies many kinds of code, such as event handlers, cell value factories, and cell value factories on TableView," explains Richard Bair, client Java architect at Oracle. "It brings some much-requested new UI controls and APIs such as TreeTableView and APIs that enable modal dialogs."

PHOTOGRAPH BY BOB ADLER

In addition, says Bair, JavaFX 8 offers a powerful rich text node for the scene graph, and the Java community is fast at work on several projects building rich text and code editors on top of this support. In addition, the default look of JavaFX applications has been refreshed with a new theme, called "Modena." JavaFX 8 also offers increased support for third-party components, many of which can now work out of the box. All of the above will lead to enhanced productivity for JavaFX developers, he says.

Much progress has also been made on the performance front, largely due to the effort to bring JavaFX to embedded devices such as the Raspberry Pi and Freescale i.MX 6 boards. Increased 3-D support in JavaFX has opened new opportunities for developers, with AMD contributing COLLADA importer functionality to OpenJFX, which is a lively and thriving community, Bair adds.

"The new Scene Builder 2, the first Scene Builder actually built with Scene Builder, has an excellent CSS inspector, so developers can better understand why things are styled as they are," explains Bair. It is also modular and can be embedded in any IDE. Scene Builder 2 works with FXML, an open and published format, so developers can hack the FXML file directly.

"I encourage everybody to download JavaFX 8, try it out, and provide feedback to the team," he adds.

MORE ON JAVAFX 8

- JavaFX 8 Overview

6 COMMUNITY INVOLVEMENT

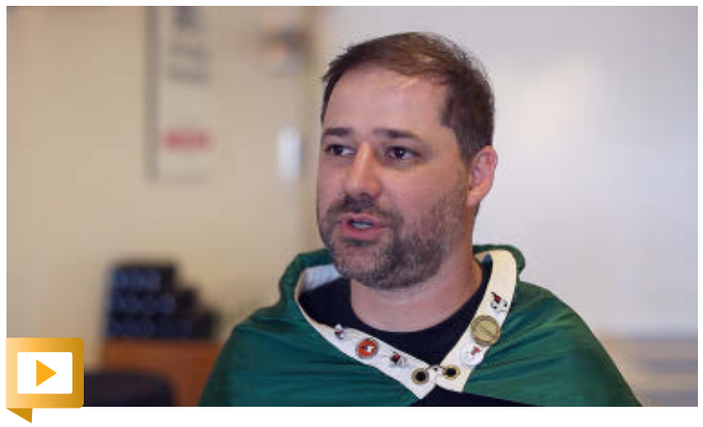
The Java community has been vital in the creation of Java 8. It has provided ongoing feedback to Specification Leads, made requests for new Java functionality, uncovered bugs, contributed to code fixes, and much more.

Java user groups (JUGs), which are centers of Java community learning and participation, have been essential. As Java Champion and SouJava JUG Leader Bruno Souza explains, "Java 8 has set a new standard of community participation. There are Java user groups all over the world running programs and providing feedback for the JCP specifications and being active in OpenJDK." The OpenJDK community is where developers collaborate on the open source reference implementation of the Java SE platform.

The Adopt-a-JSR program, created to encourage JUG members, individuals, and other organizations to get involved in JSRs, has resulted in increased grassroots and developer participation in not only Java 8 but other emerging standards as well. More than 26 JUGs have participated in the program.

The Java Community Process (JCP), with its enhanced openness and flexibility, has made it easier for both organizations and individuals to contribute to the Java platform. "The JCP now has a much more transparent process," says Souza. "It's a lot easier for someone to go to the JCP site and get involved. This increased transparency has made it easier for the community to participate."

> "Java 8 has set a new standard of community participation. There are Java user groups all over the world running programs and providing feedback for the JCP specifications and being active in OpenJDK."
>
> —Bruno Souza, Java Champion and SouJava JUG Leader



SouJava's Bruno Souza explains how community participation has changed with Java 8.

PHOTOGRAPH AND VIDEO BY BOB ADLER

JAVA ME 8

PHOTOGRAPH BY TON HENDRIKS

> "Java ME 8 is a major step in the convergence of Java ME and Java SE and the unification of the Java ecosystem."
>
> —Terrence Barr, Senior Technologist and Product Manager, Oracle

"Java ME 8 is a major step in the convergence of Java ME and Java SE and the unification of the Java ecosystem, enabling Java developers to more easily deploy their existing skills across a range of embedded devices, drawing upon the richness and portability of the Java platform," explains Terrence Barr, senior technologist and product manager at Oracle who focuses on the IoT and embedded technologies. "This results in a faster time to market, as well as cross-platform compatibility and embedded device scalability."

With the advent of Java ME 8, developers can take their existing skills and begin writing applications for the rapidly developing realm of embedded devices—using the same familiar platform, language, programming model, and tools. With the embedded space and the fast-developing IoT in a state of flux, Java ME 8 offers a flexible and scalable development/deployment environment.

See "Java ME 8 and the Internet of Things" in this issue for details.

MORE ON JAVA ME 8

- Oracle Java ME Embedded 8 Early Access
- Connected Limited Device Configuration 8
- Generic Connection Framework 8
- Micro Edition Embedded Profile 8
- Device Access API 8

JAVA SE 8 AND THE JVM

"Java SE 8 offers an improved, more mature G1 garbage collector. G1's scalability is getting better, and more work is being done concurrently," explains Jon Masamitsu, principal member of the technical staff at Oracle. "G1's basic design is not changing, but incremental improvements toward more-dependable pause times have been made in many areas.

More generally, in all the garbage collectors the command-line flags are now more consistent, with flag discrepancies corrected. This should make it easier for users to switch among the different garbage collectors and have the command-line flags have a common meaning where possible."

In addition, Java SE 8 removes the need to tune the permanent generation. The Java HotSpot VM's internal representation of Java classes has been re-engineered and moved out of the permanent generation (which has been removed entirely) and moved into native memory and/or the Java heap. Previously, tuning the size of the permanent generation was a tiresome, trial-and-error process. The maximum size of the permanent generation had to be set at startup, and there was no intuitive way to know how large it should be. Developers need no longer guess about a size for the permanent generation.

Finally, Java ME 8 has been updated with the JVM, the language, and the libraries to be aligned with Java SE 8.

"Java SE 8 offers an improved, more mature G1 garbage collector. G1's scalability is getting better, and more work is being done concurrently."
—Jon Masamitsu, Principal Member of Technical Staff, Oracle

PHOTOGRAPH BY BOB ADLER

COMMUNITY SPOTLIGHT
TEST JAVA 8

The Test Java 8 program was initiated by the London Java Community as part of the Adopt-a-JSR program. The aim of the program was for Java user group members to help with testing Java 8 on several open source software projects prior to the launch of Java 8.

TAKE IT FOR A SPIN

All in all, Java 8 offers a new opportunity for enhanced innovation for Java developers who operate on anything from tiny devices to cloud-based systems. We can expect increases in developer productivity and application performance through the reduced boilerplate code and increased parallel programming that lambdas offer. Java 8 offers best-in-class diagnostics, with a complete tool chain to continuously collect low-level and detailed runtime information.

By bringing the advantages of Java SE to embedded development, developers can transfer their Java skills to fast-evolving new realms in the IoT, enabling Java to support any device of any size. It is an exciting time to be a Java developer, so take it for a spin, and tell us about the ride.

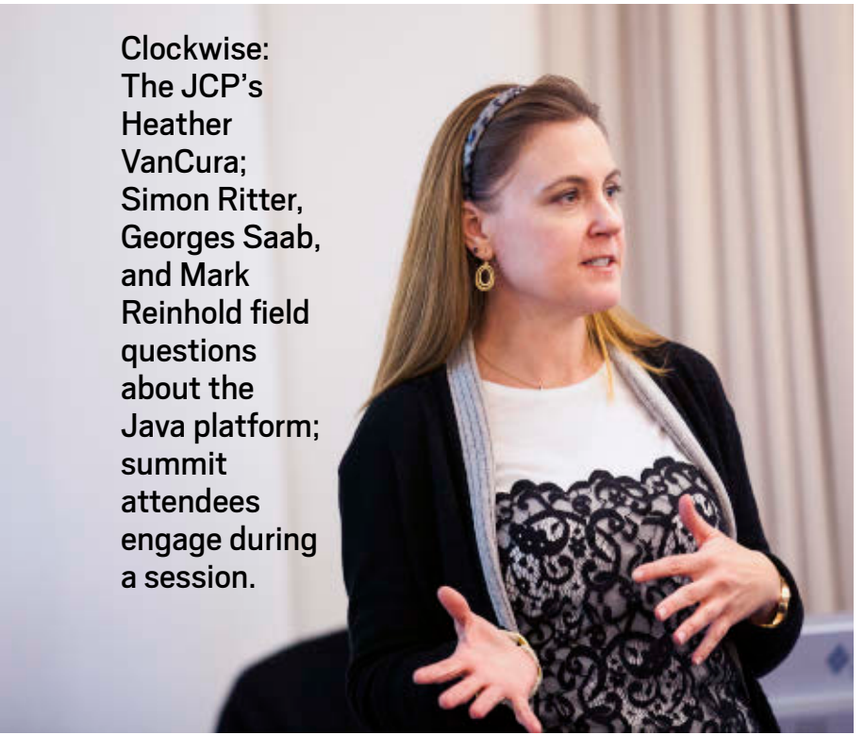
MORE ON TOPIC:

Java 8 Is Here

Timothy Beneke is a freelance writer and editor, best known for his books on gender.

DOWNLOAD JAVA 8

- Java SE 8
- Java ME 8



Clockwise: The JCP's Heather VanCura; Simon Ritter, Georges Saab, and Mark Reinhold field questions about the Java platform; summit attendees engage during a session.

ORACLE USER GROUP LEADERS' SUMMIT

PHOTOGRAPHS BY BOB ADLER

More than 20 Java user group (JUG) leaders from all over the world attended the International Oracle User Group Community (IOUC) Annual Summit January 21–23, 2014, in Redwood Shores, California. The IOUC is a community of leaders representing Oracle user groups worldwide. User group leaders from all over the world, including more than 20 JUG leaders, attended the summit to learn about Oracle products and technologies, provide feedback to product groups, network, and share best practices.

The unofficial kickoff to the Java track was a Brazilian barbecue at the home of **Stephen Chin**, a Java evangelist at Oracle. **Bruno Souza**, leader of SouJava in Brazil and a member of the Java Community Process (JCP) Executive Committee, acted as executive chef, manning the barbecue for hours after just stepping off a flight from Brazil.

The official Java track began the following day,

Summit attendees have a healthy discussion over lunch.

with a session on the future of Java. Oracle's **Mark Reinhold**, **Georges Saab**, **Henrik Stahl**, and **Simon Ritter** hosted an interactive discussion of what's next for the Java EE, Java SE, and Java ME platforms (beyond Java EE 7, Java SE 8, and Java ME 8). Topics included the Internet of Things, big data, Project Sumatra, OpenJDK, alternative languages for the Java Virtual Machine, modularity, adoption of the Java platform, and more.

In a session on JUGs and the JCP, **Heather VanCura** and **Patrick Curran** of the JCP led a discussion about how JUGs can be more involved with the JCP and JSRs, including the Adopt-a-JSR program. "Adopt-a-JSR has been a great push and has put the JCP back in the spotlight," said VanCura. "There is more interest from JUGs, more people are involved, and there are more discussions." One way that many JUGs are involving their members is through hackathons. **Badr Elhouari** of the Morocco Java User Group said that getting a hackathon going is easy. "Have a speaker," he said, "and then do a hands-on activity at your JUG meeting."

Other topics of discussion were barriers to participation.

PHOTOGRAPH BY BOB ADLER

HOT TOPIC:
INTERNET OF THINGS

Clothes that monitor your baby's vital signs. **Prescription medicine bottles** that remind you to take your pills (and can tell others when you don't). **Wind turbines** that turn themselves on in anticipation of high energy usage. Traffic lights that adjust to traffic conditions. **Trash cans** that send a text when they are full. These are all examples of the rapidly growing world of the Internet of Things (IoT). **There is a lot of interest in this topic, so it wasn't surprising that the Internet of Things session was full at the IOUC Summit.**

Oracle's **Sharat Chander** assembled a panel of experts to discuss IoT: **Bruno Souza**, SouJava president and Java Community Process (JCP) Executive Committee member; **Stephen Chin** and **Jai Suri** of Oracle; **Ian Ferguson** of ARM; and **Richard Niemiec**, Midwest Oracle Users Group leader.

The panel started by defining the *Internet of Things*. Most computers already talk to a "back end," so what makes IoT different? "IoT makes computers invisible," Souza explained. "They are everywhere and all connected. You don't think of your refrigerator as a computer." Suri talked about the multitude of devices and networks involved, the challenge of data in many different formats, and powering small devices.

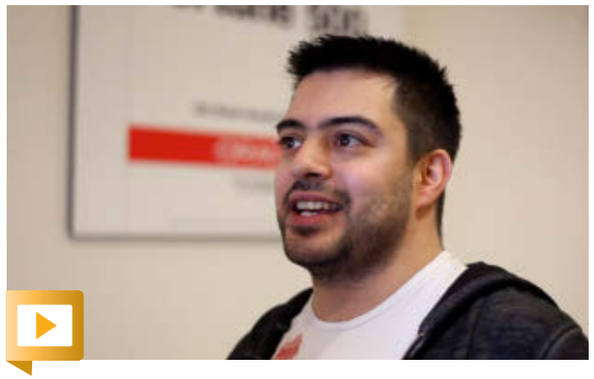
The panel also discussed the challenges of IoT. Chin asked, "How many people have a wireless router at home?" and many hands went up. But when he asked how many people had updated their router firmware in the last six months, most hands dropped. With more devices come more software, more management, and more vulnerability. Security is a real issue, especially for devices and industries that haven't had to consider it before. And it's not *if* you get hacked, but *when*. "One good thing about Java in the IoT space," Suri commented, "is that it provides a level of abstraction that allows for better security and quicker updates. That's especially important in devices that are out in the field for years."

Whatever it is, IoT is coming fast. New applications are happening daily. The coolest IoT innovation hasn't even been thought of yet.

SouJava's Bruno Souza (center) grilled up a traditional Brazilian feast to welcome Java user group leaders to the summit.

ticipation in the JCP for those who don't speak English and ways to work around this; lack of awareness of the JCP; and complexity in getting involved. However, Adopt-a-JSR is helping to lower the barriers to participation and to promote experimentation. By participating in the Adopt-a-JSR program, said SouJava's Souza, your members will feel like they are participating and they will learn more about Java.

Other session topics during the summit included gamification, managing your community, getting the most out of Java.net, and feedback about JavaOne.



Java user group leaders share their reasons for attending the summit.

PHOTOGRAPH COURTESY OF STEPHEN CHIN; VIDEO BY BOB ADLER

GAMIFICATION FOR USER GROUPS

ArabOUG Leader Mohamed Chargui talks about his experience using gamification.

At the Gamification session at the IOUC Summit, user group leaders discussed how to drive membership. Typically, they give away software licenses, books, and goodies to encourage attendance at monthly meetings. Others have used gamification to get their communities to brainstorm on mascot names, or post pictures and comments on social media. Hackathons also require the use of similar techniques to keep attendees motivated to create applications over several days. SouJava Leader **Bruno Souza** said that his Java user group (JUG) successfully ran hackathons that combined brainstorming, team building, training, hacking sessions, and prizes to keep participants engaged.

Gamification lets you turn life into a game, said Houston JUG Leader **Jim Bethancourt**. The key advantages are driving audience engagement, making the user experience more enjoyable, and getting users to come back. The forum platform Stack Overflow is a great example of running a thriving community of developers with its point systems, he said. Contributors get rewarded with points for their useful entries, and visitors easily find the most-relevant and best-rated entries.

The ArabOUG has implemented a point system to keep its community active. The group gives out points to its members, who contribute applications, articles, and translations. It partnered with training organizations and other services to give its members free training and services in exchange for points. As a result, members don't have to pay for services using online payments, which governments in many countries in North Africa and the Middle East don't allow.

HACKING AT CAMPUS PARTY

Clockwise from top left: Duke is ready to hack; Yara Senger welcomes participants; hackers work on their projects.

Brazilian Java user group SouJava ran a Raspberry Pi and Java hackathon from January 30–February 1, 2014, at Campus Party, the weeklong technology gathering of geeks, developers, gamers, scientists, and students in Brazil. Sponsored by Oracle Technology Network, the hackathon was designed for enthusiasts who wanted to create Internet of Things (IoT) projects with the Raspberry Pi and Java. The objectives were for attendees to learn, practice, and innovate.

Oracle Java evangelist **Angela Caicedo** opened the hackathon with an overview of IoT and Java development. Over three days, participants formed teams, brainstormed, attended training, received a kit from the organizers, and hacked on their own projects. Onsite experts were available to help participants. These veteran Java developers of web, enterprise, and embedded development included GlobalCode Founder **Vinicius Senger**, Senior Developer at Vitae Futurekids Brazil **Rubens Saraiva**, SouJava Leader **Bruno Souza**, Java Champion **Yara Senger**, Oracle Product Manager **Bruno Borges**, and Senior Mobile Developer at Mobilidade é tudo **Ricardo Ogliari**.

PHOTOGRAPHS COURTESY OF SOUJAVA

IoT Developer Challenge

From March 3 to May 30, 2014, Oracle Technology Network is hosting an Internet of Things (IoT) Developer Challenge. Open to online submissions, the challenge gives developers the chance to submit an innovative, groundbreaking, business-ready IoT project using embedded Java and any hardware, whether it is computer boards such as the Raspberry Pi, sensors, or other IoT technologies. Members of winning teams will each receive a JavaOne 2014 pass, plus up to US$2,000 for flight and hotel expenses. A student category with prizes is also available, thanks to Oracle Academy.

During the challenge period, Oracle Technology Network will offer free online training on IoT, the Raspberry Pi, embedded Java, and more. By registering, you will enter a chance to win a Raspberry Pi.

ART BY I-HUA CHEN

FEATURED JAVA USER GROUP

JUG Münster

Java User Group Münster was formed in December 2008 by **Thomas Kruse** and **Gerrit Grunwald** (inset), who continue to lead the group today.

Münster's geographic location posed some challenges as the Java user group (JUG) was getting started. Münster is only about 50 km away from Germany's populous Ruhr area, which has some long-established larger JUGs that included members from the Münster area.

Nevertheless, the JUG continued to meet; Kruse and Grunwald continued spreading the word; and JUG Münster's membership steadily grew to the current level of 280 members.

JUG Münster has had an active meeting and events schedule from the start. "Since we started in 2008, we've met every two weeks for a so-called 'Java Roundtable,'" says Grunwald. "In addition, we arrange 10 events each year where we invite external speakers; and we have an annual event for Software Freedom Day."

Developers who come to Java after having worked with other languages are often surprised by Java's community aspect. "Java is not a young language/environment anymore, but still has the greatest community that I know, and one reason for this is the Java user groups... so let's keep this alive," says Grunwald.

JAVA CHAMPION PROFILE

HOWARD LEWIS SHIP

Howard Lewis Ship *is a Java EE and Clojure developer, author, blogger, and speaker. The creator of the Apache Tapestry open source web framework, he became a Java Champion in February 2010.*

Java Magazine: Where did you grow up?

Ship: I grew up south of Boston, Massachusetts, and got my CS degree at Worcester Polytechnic Institute.

Java Magazine: When did you first become interested in computers and programming?

Ship: My dad bought a Burroughs B800 computer, a beast I called "HAL," for doing payrolls for his CPA business. It came with a "Battleship" game you played by printing the map on the line printer and entering shots using the keyboard.

Java Magazine: What was your first computer and programming language?

Ship: I had access to our high school's PDP-11 when I was in junior high. I started out by entering and running BASIC code from David Ahl's *BASIC Computer Games*. I also experimented with Pascal and Logo on an Apple, and even wrote some games that had a smidgen of 6502 assembly code.

Java Magazine: What was your first professional programming job?

Ship: I helped my dad after school, customizing an invoicing, inventory, accounts payable, and accounts receivable business suite. An early programming accomplishment was developing an automatic zip-code-to-UPS-zone lookup that saved my Dad's order entry team lots of time and tedious effort.

Java Magazine: What do you enjoy for fun and relaxation?

Ship: I have a four-year-old and a one-year-old, so fun and relaxation are mostly "something I used to do." I enjoy skiing, photography, reading, flying stunt kites, and playing board and card games.

Java Magazine: What "side effects" of your career do you enjoy the most?

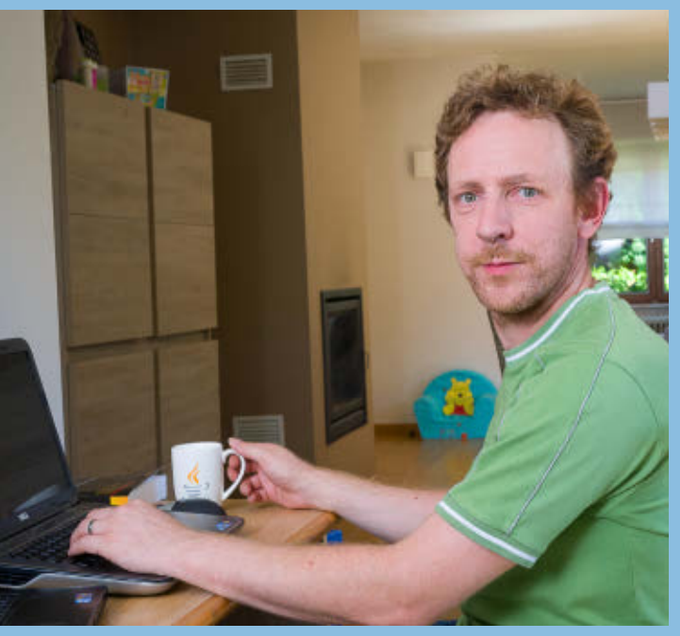
Ship: I love to speak at conferences and see the work people are doing. I'm especially interested to see how people are using my open source projects. It is fascinating that so much innovative work happens in smaller cities, not just the larger technology centers. It's everywhere!

Java Magazine: What are you looking forward to?

Ship: It will be interesting to see the functional programming aspects of Java SE 8 go mainstream. For some developers, it will open their eyes to the limitations that have been around them. I fear, though, that for many others it will mostly provide a new way to do things wrong.

But what I'm really looking forward to is the kids being old enough to take care of themselves in the morning, so that I can sleep in every once in a while.

THE JAVAFX ANDROID COMMUNITY PROJECT



The JavaFX Android Community Project seeks to make JavaFX code runnable on Android devices. The project was started by **Johan Vos** (@johanvos) after he talked with JavaFX architect **Richard Bair** (@richardbair) at Devoxx 2013. The project got off to a fast start due to work that Oracle's **Tomáš Brandalík** had already done on the native parts, and the Java 7 backport maintained by **Stefan Fuchs**. "After only a few weeks, we delivered a JavaFX runtime for the Android platform," says Vos.

Vos blogged about the project's background, present, and future in the post "JavaFX and Android." There he notes that with respect to porting software to unsupported platforms, "developers are usually better in doing it themselves than in complaining. With the JavaFX code being open-sourced over the past years, and with lots of work already being done by Oracle people, a community effort for porting JavaFX to mobile platforms turned out to be a viable option."

You can find instructions for building and deploying JavaFX applications on the Android platform on the project site. The project's source code is available in the Bitbucket android-graphics-rt repository. You can download a prebuilt JavaFX Android runtime, or you can build it yourself.

The project has come a long way in a relatively short time. In his blog, Johan says, "Most of the JavaFX 8 Ensemble applications are now working out of the box on Android devices. I've seen many reports from JavaFX developers demonstrating their applications on Android, which is very cool."

The JavaFX Android Community Project currently has 4 active developers. About 20 developers have contributed feedback that has resulted in patches and new code. But, as the project's task list reveals, there's a lot more work to do. So, the project is actively seeking broader participation.

"All contributors are very welcome," Vos says. "Tool support is high on our list. People who can help with integrations with IDEs and packaging tools are very welcome. Also, native Android developers are very much appreciated, especially people with feedback on how to complete the last mile toward sending an application to the Play store. Developers with Gradle knowledge are also very welcome."

If you'd like to contribute to the JavaFX Android Community Project, visit the JavaFXAndroid Google Group, or contact Vos.

PHOTOGRAPH BY TON HENDRIKS

JUG Tour

Oracle Technology Network is hosting a Java user group (JUG) tour following the launch of Java 8. Events will be held at JUGs across the globe, with Java evangelists and Java Champions talking about key features in Java 8. Follow @java for more information.

EVENTS

Devoxx France *APRIL 16–18*
PARIS, FRANCE

Now in its third year, Devoxx France brings together 1,500 Java enthusiasts in an adrenaline rush of technical content, training, and community networking. The conference focuses on Java, web technologies, mobile, cloud, agile, Java Virtual Machine languages, and entrepreneurship. This year's theme, "Born to Be," is about being and becoming a developer.

PHOTOGRAPH COURTESY OF WILHELM LAPPE ON FLICKR

4Developers
APRIL 7
WARSAW, POLAND
4Developers is a technical conference intended for developers, architects, testers, coders, team leaders, project managers, IT departments, as well as all IT students. The event is designed for both professionals and amateurs working in different technological fields.

Great Indian Developer Summit (GIDS)
APRIL 22–25
BANGALORE, INDIA
Twenty-five thousand developers are expected at this four-day conference. At the dedicated Java track, top experts Stephen Chin, Venkat Subramaniam, Erik Hatcher, Kito Mann, Pratik Patel, and others will talk about Java 8, lambdas, InvokeDynamic, the Raspberry Pi, JavaFX, and more.

JAX 2014
MAY 12–16
MAINZ, GERMANY
This conference on Java and the enterprise focuses on current and future trends in web, Android, software architecture, cloud, agile management methods, big data, and more. More than 180 speakers will present 230 talks and 20 day-long sessions on Java 8, Java EE, JavaFX, embedded Java, Java core, and agile.

GeeCON 2014
MAY 14–16
KRAKOW, POLAND
With its motto "Let's move the Java world," the conference focuses on Java technologies, dynamic languages, rich internet applications, enterprise architectures, patterns, distributed computing, software craftsmanship, and more.

JEEConf
MAY 23–24
KIEV, UKRAINE
JEE is the largest developer conference in Ukraine. It focuses on Java technologies for application development.

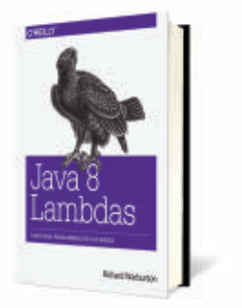
Devoxx UK
JUNE 12–13
LONDON, ENGLAND
Devoxx UK is part of the Devoxx family of conferences. Run by the local Java community, this developer conference consists of a packed schedule of presentations, hands-on labs, quickies, and Birds-of-a-Feather (BOF) sessions. Top local and international speakers present an array of Java topics on web, mobile, Java Virtual Machine languages, agile, and more.

NightHacking Tour

Java Evangelist **Stephen Chin** is making his way across Europe—visiting Java user groups, conducting interviews, and broadcasting it all. Chin will be talking with community experts about Java 8, the community, the future of Java, the Internet of Things, and more. Join him at an event or watch his sessions online at nighthacking.com.

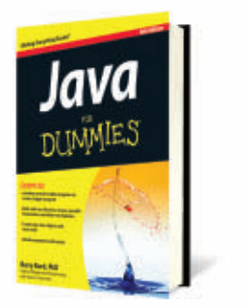
JAVA BOOKS

JAVA SE 8 FOR THE REALLY IMPATIENT
By Cay S. Horstmann
Addison-Wesley Professional (January 2014)
Author of *Core Java* and internationally renowned Java expert Cay S. Horstmann concisely introduces Java SE 8's most valuable new features (plus a few Java SE 7 innovations that haven't gotten the attention they deserve). If you're an experienced Java programmer, Horstmann's practical insights and sample code will help you quickly take advantage of these and other Java language and platform improvements.

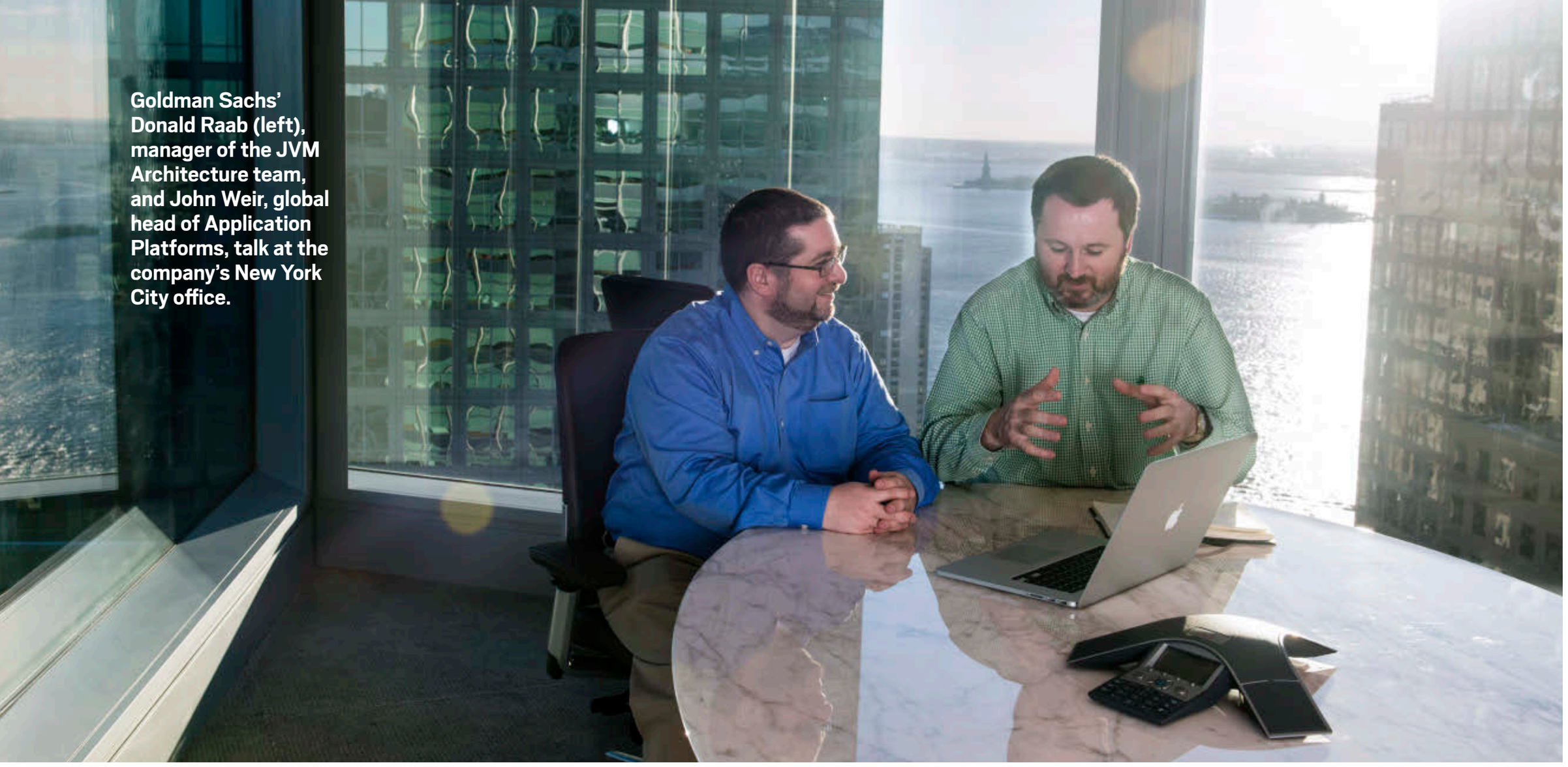


JAVA 8 LAMBDAS
By Richard Warburton
O'Reilly (April 2014)
If you're an experienced Java programmer, *Java 8 Lambdas* shows you how to make use of your existing skills to adapt your thinking and your codebase to use lambda expressions properly. Starting with basic examples, this book is focused solely on Java SE 8 language changes and related API changes. Lambdas make a programmer's job easier, and this book will teach you how. Coverage includes introductory syntax for lambda expressions, method references that allow you to reuse existing named methods from your codebase, and the collection library in Java SE 8.

JAVA: THE COMPLETE REFERENCE, NINTH EDITION
By Herbert Schildt
Oracle Press (April 2014)
In *Java: The Complete Reference, Ninth Edition*, Herbert Schildt uses the clear, concise, uncompromising style that has made the previous eight editions so successful worldwide. This comprehensive volume covers the entire language, including its syntax, keywords, and fundamental programming principles. Schildt also presents detailed information about Java's core libraries and key programming techniques. Of course, coverage of new features added by Java SE 8, such as lambda expressions and modules, is included.



JAVA FOR DUMMIES, 6TH EDITION
By Barry Burd
Wiley (March 2014)
In this update of a best-selling book, veteran author Barry Burd shows you how to create basic Java objects and clearly explains when you should simply reuse existing code. He explores how Java SE 8 offers more-robust functionality and new features such as closures to keep Java competitive with more syntax-friendly languages like Python and Ruby. He also covers object-oriented programming basics with Java, code reuse, the essentials of creating a Java program using the new JDK 7, creating basic Java objects, and new Eclipse features.



Goldman Sachs' Donald Raab (left), manager of the JVM Architecture team, and John Weir, global head of Application Platforms, talk at the company's New York City office.

JCP Executive Series

Goldman Sachs' GS Collections Leverages Lambdas

Goldman Sachs' John Weir and Donald Raab discuss the creation of GS Collections, the impact of Java SE 8's lambda expressions, and how their work interacts with the JCP. **BY STEVE MELOAN**

PHOTOGRAPHY BY LAURA BARISONZI/GETTY IMAGES

Java 8 Is Here

John Weir is the global head of Application Platforms at Goldman Sachs. In 2011, Goldman Sachs was elected to the Executive Committee of the Java Community Process (JCP), with Weir as the firm's representative. He previously represented Goldman Sachs on the FpML Standards Committee for Credit Derivatives and was also on the FpML Architecture Committee. He joined

Goldman Sachs in 1997 and has held a number of senior technology roles. He was named technology fellow in 2004 and managing director in 2008.

Donald Raab manages the JVM Architecture team, which is part of the Enterprise Platforms group in the Technology division at Goldman Sachs. Raab serves as a member of the JSR 335 Expert Group (Lambda Expressions for the Java Programming Language) and is one of the alternate representatives for Goldman Sachs on the JCP Executive Committee. He joined Goldman Sachs in 2001 as a technical architect on the PARA team. He was named technology fellow in 2007 and managing director in 2013.

Java Magazine: Tell us about the genesis of GS Collections. When did you begin work, and what were your initial goals?

Raab: GS Collections began in 2004 as a Goldman Sachs internal project called CARAMEL, which stood for collections, arrays, map iteration library. We were dealing with very large Java heaps, on the order of ten gigabytes to a hundred gigabytes. And in these spaces, we operated on large collections to process lists, sets, and maps that had millions or even hundreds of millions of elements. The codebases were very large—with thousands and sometimes tens of thousands of iteration patterns implemented with handwritten for loops.

Prior to Java SE 8, in the collections space, there was no rich API. Developers had to re-create patterns again and again. Our goal was to provide a functional API that would reduce duplicate code by separating the "what" from the "how" in a developer's code. We also wanted to leverage parallelism using the fork/join library, allowing developers to optimize performance across multiple processor cores.

The API for GS Collections was heavily influenced by my experiences programming in Smalltalk, which had useful methods like select, reject, and collect. Over the years, we've added methods like flatCollect, groupBy, partition, and zip, which were influenced by our experiences with languages like Haskell, Ruby, and Scala. There are also optimized replacements for the JDK Collections classes, like HashMap, ArrayList, and HashSet. GS Collections is tuned for both large- and small-scale performance. It's ready for Java SE 8 lambdas today, but it also works with Java 7.

REALITY CHECK

"Innovation needs to be carefully vetted by the community through the JSR process."

—John Weir, Global Head of Application Platforms, Goldman Sachs

Java Magazine: *Internal iteration* is a key concept for GS Collections. Why is it so powerful?

Weir tackles a problem on the whiteboard, while Raab looks on.

Raab: Internal iterators reduce errors and hide the implementation details of collections they're operating across. They relieve the developer from the responsibility of optimizing the various datatypes being processed. An internal iterator is passed an operation to perform, and the iterator applies that operation to every element in the data collection.

For example, if one collection is being transformed to another, and I know the target is the same size as the source, it can be presized in the algorithm, so the developer doesn't have to handle implementation details. This approach simplifies code and promotes reuse.

In the Java space, most developers implement what's called *eager iteration*,

Weir and Raab fuel up and catch up on projects.

where computation happens immediately. But there are patterns where it's advantageous to perform computation *lazily*, meaning that processing can be done a bit later. This type of processing is not easily implemented by developers. But using internal iteration, processing details are handled internally by the collections classes.

Internal iteration is also advantageous for code maintenance. It makes large codebases much more readable. Over the long life of an application, this can have tremendously important productivity ramifications.

Java Magazine: How will Java SE 8's lambda expressions interact with GS Collections? Are there other Java SE 8 features of interest?

Raab: GS Collections was developed at the outset around the concepts embodied in lambda expressions. Having lambdas built into Java SE 8 is absolutely terrific; we will be able to leverage them extensively across our APIs. We have 90+ functional methods that can be called by passing lambdas.

This Java SE 8 feature will deliver a huge upside. We'll be able to delete a lot of code. I imagine a savings between 20 percent and 30 percent.

For example, the Select pattern handwritten in Java SE 7 looks like this:

```
List<Integer> results = new ArrayList<>();
for (Integer each : list)
{
  if (each > 50)
  {
    results.add(each);
  }
}
```

While the Select pattern in GS Collections with Java SE 8 looks like this:

```
list.select(each -> each > 50);
```

Another interesting development in Java 8, virtual extension methods, will benefit GS Collections. This feature allows behaviors to be added to interfaces, which will help reduce the amount of duplicate code in the GS Collections testing hierarchy by building-in default methods and behaviors. This will allow the mixing and matching of different testing concerns across the framework.

Method references, another Java SE 8 feature closely tied to lambdas, will also benefit GS Collections. They provide an even more concise and readable format for leveraging functional APIs.

The following is a method reference code example from our GS Collections Kata Java 8 Solutions:

```
/**
 * Aggregate the total order values by
      item.
 */
@Test
public void totalOrderValuesByItem()
{
  MutableMap<String, Double> map =
      this.company
        .getOrders()
        .flatCollect(Order::getLineItems)
        .aggregateBy(LineItem::getName,
            () -> 0.0, (result, lineItem) ->
            result + lineItem.getValue());
  Verify.assertSize(12, map);
  Assert.assertEquals(100.0, map
      .get("shed"), 0.0);
  Assert.assertEquals(10.5, map
      .get("cup"), 0.0);
}
```

Method references make maintenance and review cleaner and easier.

Optimizations have been made within the language and the JVM [Java Virtual Machine] to take advantage of these innovations.

Java Magazine: When did GS Collections become open source? Tell us about that process.

Raab: We open-sourced GS Collections in January 2012. Before doing that, we open-sourced it internally. At Goldman Sachs we have a large number of developers and applications; it's a very collaborative and consensus-driven organization. We wanted to find out if there was a viable community of potential users for GS Collections. And we discovered that the demand was definitely there.

In addition to open-sourcing the collections library, we decided to include the associated training libraries, the GS Collections Kata, which is a great way to learn not just GS Collections but also Java SE 8 lambda expressions and method references.

Java Magazine: How has Goldman Sachs contributed to the JCP process?

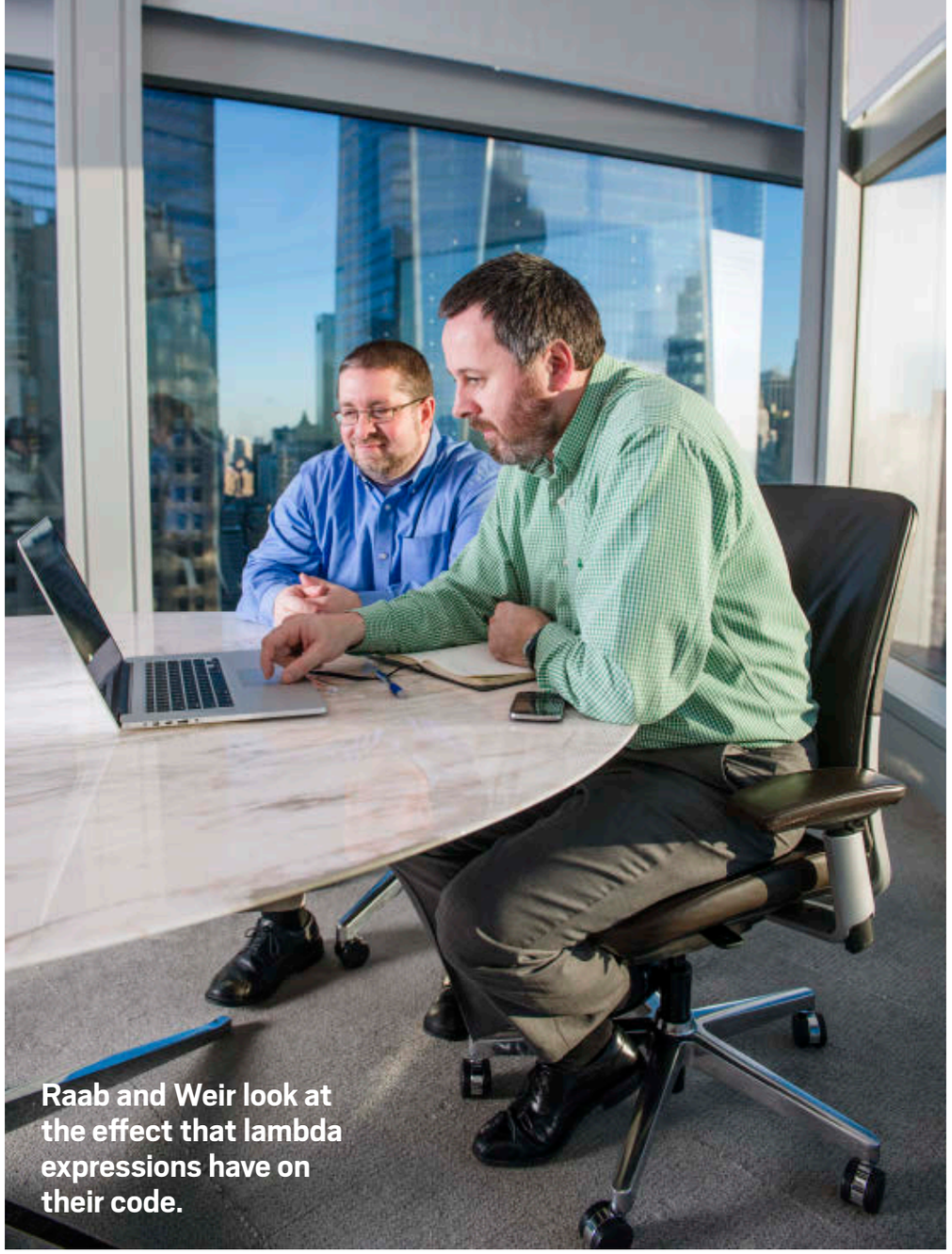
Weir: We're engaged in the JCP process at several different levels. Our role with the Executive Committee involves reviewing JSRs at various stages. And internally we leverage our network to interact with several thousand Goldman Sachs developers to get their JSR feedback. For example, on JSR 310, the new and improved Date and Time API, we uncovered a problem with a method that replaces getDateTimeMillis from Joda-Time. The new method to extract time in milliseconds ran about three and a half times slower and created some garbage on the JVM. The Goldman Sachs community uncovered this due to our extensive codebase, and we were able to offer feedback through our role in the Executive Committee.

We also participate directly in the JSR process. We've been involved as contributing members in lambda expressions, the Collections API, the JCache API, and also the Identity API. We also participate in debates and discussions as to how the JCP process should operate, evolve, and move forward.

Java Magazine: JCP member companies have diverse interests and priorities. How are these needs best orchestrated within the JCP?

Weir: One of the great strengths of the Executive Committee is its breadth of organizational participation—from vendors and redistributors of the Java platform, to framework and library providers, to Java user groups [JUGs] and end users. It's an excellent representative community all sitting at the virtual table with the goal of guiding Java's evolution.

That goal unifies the community, in spite of some vigorous competition on the vendor side. We try to maintain a cohesive Java platform that doesn't become fragmented and continues to accommodate a broad array of functional demands.



Raab and Weir look at the effect that lambda expressions have on their code.

Java Magazine: What is the proper balance between promoting standards and promoting innovation?

Weir: We all want innovation. And innovation is always there. It's a fundamental ingredient when developers solve problems. Donald talked earlier about CARAMEL evolving into GS Collections. That software resulted from developers striving for a simplified, transpar-

CHANGE MAKER

"[Lambda expressions] will deliver a huge upside. We'll be able to delete a lot of code. I imagine a savings between 20 percent and 30 percent."

—Donald Raab, JVM Architecture Team Manager, Goldman Sachs

ent, and more engaging API. In the Java ecosystem, we have a great degree of openness, which spurs innovation. There is a collective wisdom that comes from talented people solving real-world problems. But innovation needs to be carefully vetted by the community through the JSR process, to make sure that a diverse range of needs are being met. Millions of developers are affected by these standards. The platform has to function as an integrated whole.

The work that was done in JSR 348 to improve transparency across the JSR process has been very beneficial. It requires that Expert Groups use public mailing lists and maintain a public Issue Tracker. JSR 348 mandates that all JCP members and members of the public must have the opportunity to view, comment on, and participate in the process. These developments will make the JSR process a lot less opaque, and will extend the reach to as broad a community as possible.

Java Magazine: What are your views on improving JCP participation?

Weir: The Executive Committee is currently debating the topic of how to improve participation. A wide range of enterprises—from pharmaceuticals, to financial services, to consumer products—have significant numbers of Java developers. For an individual to join the JCP and participate in the process, their employer must sign the Java Specification Participation Agreement (JSPA). Because this is a legal contract, there can be restrictions for some companies. Consequently, some Java developers don't have a mechanism to become involved. Having seen the benefit from large numbers of developers engaged in software evolution, with GS Collections, we're particularly invested in the process as a core value. So we're intent on broadening involvement, but it must be done in a thoughtful and methodical way that reflects the best interests of the community. JSR 348 is tackling that. I believe that increasing participation will increase the rate at which new innovations can be integrated into the Java platform. There are millions of Java developers out there, and we'd like to give them all the opportunity to participate in the process.

Java Magazine: Any closing remarks?

Weir: We enjoy engaging in the JCP process. Our role on the Executive Committee is a high-priority task. Frequently we leverage our internal JUG to comment on JSRs.

JSR participation is beneficial for us, and also beneficial for the broader Java enterprise. Donald mentioned the GS Collections Kata. People are using the training materials we created for GS Collections and applying them to general training for lambda expressions and Java SE 8. We're pleased to have made contributions to the greater community.

Raab: I would echo John's comments regarding our commitment to Java. We're working with JUGs globally, including groups in New York, London, and Hong Kong. The community is huge and diverse, which is fantastic.

We've also just signed the agreement to participate in OpenJDK, and we look forward to making direct contributions. We want to give back to the open source community through this process.

The Java platform is a very important element of the Goldman Sachs infrastructure. We're committed to making sure that it continues to evolve and develop to provide benefits for the community and our firm. `</article>`

MORE ON TOPIC:

Java 8 Is Here

Steve Meloan is a former C/UNIX software developer who has covered the web and the internet for such publications as *Wired*, *Rolling Stone*, *Playboy*, *SF Weekly*, and the *San Francisco Examiner*. He recently published a science-adventure novel, *The Shroud*, and regularly contributes to *The Huffington Post*.

LEARN MORE

- GS Collections
- lambda expressions

Guilherme Spina (left), managing director at V2COM, talks with Leonardo De Moura Rocha Lima, chief technology officer, at the company's headquarters in São Paulo, Brazil.

Java 8 Is Here

FROM SMART GRIDS TO MICROBREWS

V2COM uses Java to connect millions of intelligent devices.

BY DAVID BAUM AND ED BAUM

PHOTOGRAPHY BY PAULO FRIDMAN

The science fiction films of yesteryear often depict a world of pervasive automation. Computers are everywhere, and people are constantly interacting with them. Other, perhaps more-prescient cinematic visions foresee chic living environments without a CPU in sight. The gadgets work autonomously, with circuits and controllers hidden from view.

Fast-forward to the present. The technologies that we use every day are fading into the background as the Internet of Things ushers in a new era of intelligent, seamlessly connected devices. With embedded microprocessors and a wireless communications fabric, today's "computers" require less intervention than ever before, even as they enable a higher degree of automation. At the forefront of this transformation is V2COM, a leading Latin American provider of smart grid technologies and advanced metering infrastructure.

"The Internet of Things is here, now," says Guilherme Spina, managing director at V2COM. "This is an interesting phase as IT moves from the data cen-

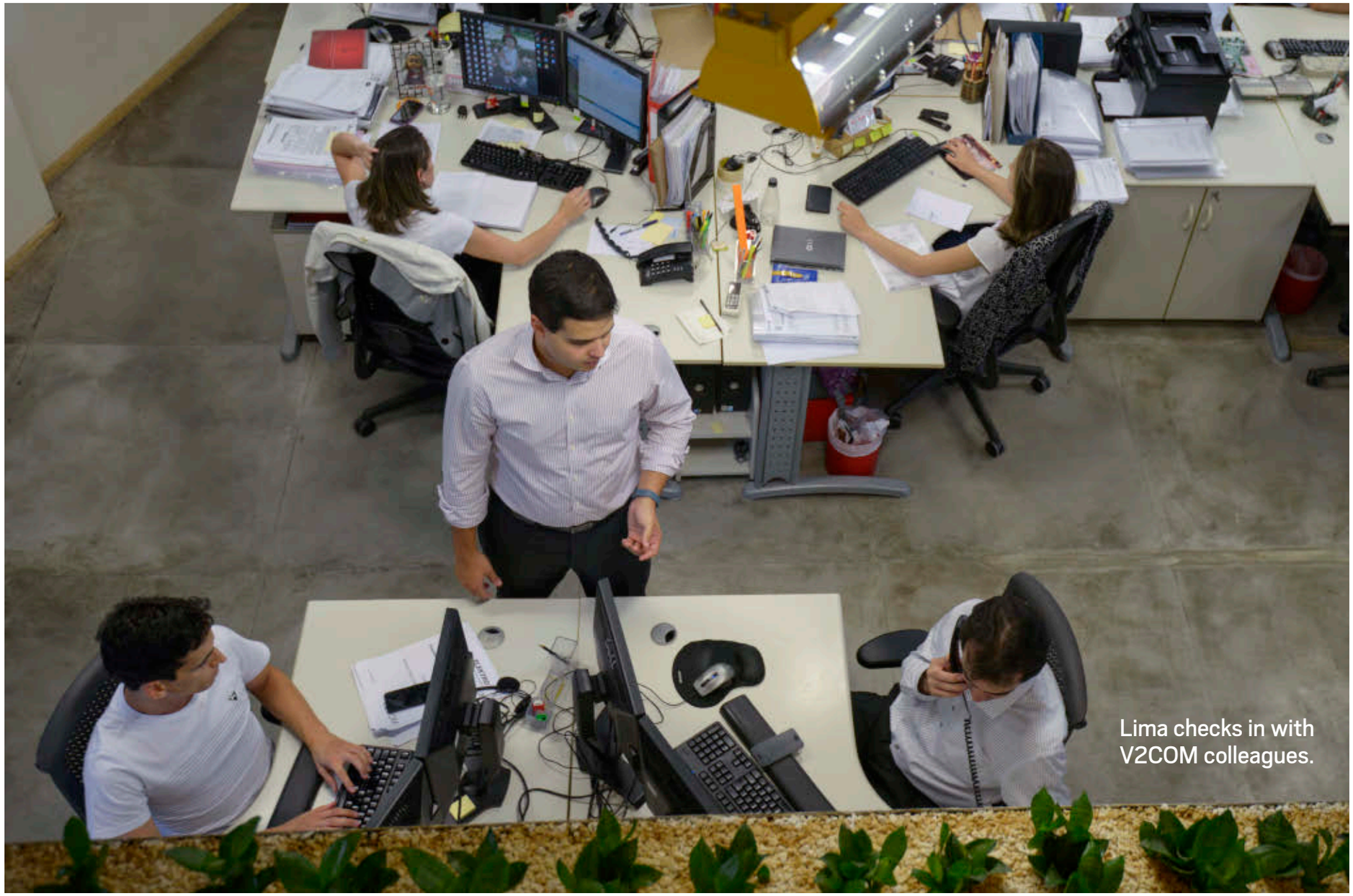
V2COM
v2com.mobi

Industry:
High technology/ energy

Location:
São Paulo, Brazil

Employees:
50

Oracle technologies used:
Oracle Java ME Embedded, Oracle Java SE Embedded, Oracle Utilities Meter Data Management, GlassFish Server Open Source Edition, Oracle Database, Oracle WebLogic Server



Lima checks in with V2COM colleagues.

ter and the desktop out to the streets to be embedded in all types of things. For example, in the energy industry, companies are replacing dumb meters with smart meters that not only enable people to conserve energy but also to play an active role in providing other services, such as internet connectivity and in-home automation solutions. Java is key to creating and delivering these solutions to our clients."

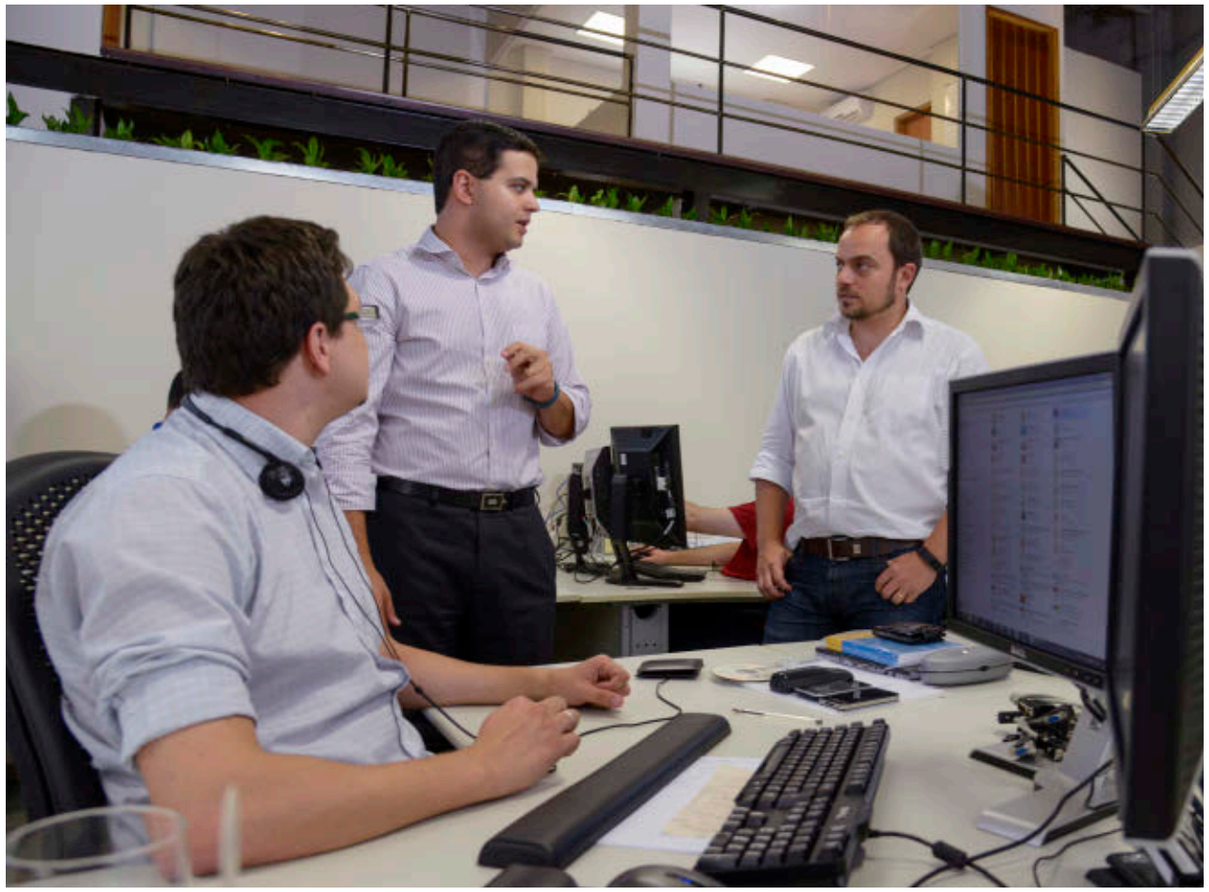
FROM STARTUP TO INDUSTRY LEADER WITH JAVA

When it was formed in 2002, V2COM was—in Spina's words—a "bootstrap startup." The fundamental ideas upon which the company is based were formulated during the mobile phone phenomenon that swept through Brazil in the 1990s. Spina and his colleagues realized they could leverage the same communications infrastructure that connects people to automate remote devices.

One of the first orders of business for V2COM's founders was to choose a software platform to manage the interactions between these remote devices and legacy systems. "We chose Java from the beginning," Spina says. "We liked the speed of development and ability to reuse components by writing code once and deploying it many times,

supporting functionality on both the mobile side and the server side. Also, we found that the majority of the people we hired were proficient with Java."

Spina and his colleagues also favored Java for its high level of support for object-oriented programming and for its open source development environment, which makes it easier to add intelligence to a wide variety of end units such as meters, transformers, and circuit breakers. These components work together to manage energy flow at homes and businesses and report back to the energy distribution systems at a utility company's head office.



Lima (center) and Spina (right) get a project update from a V2COM team member.

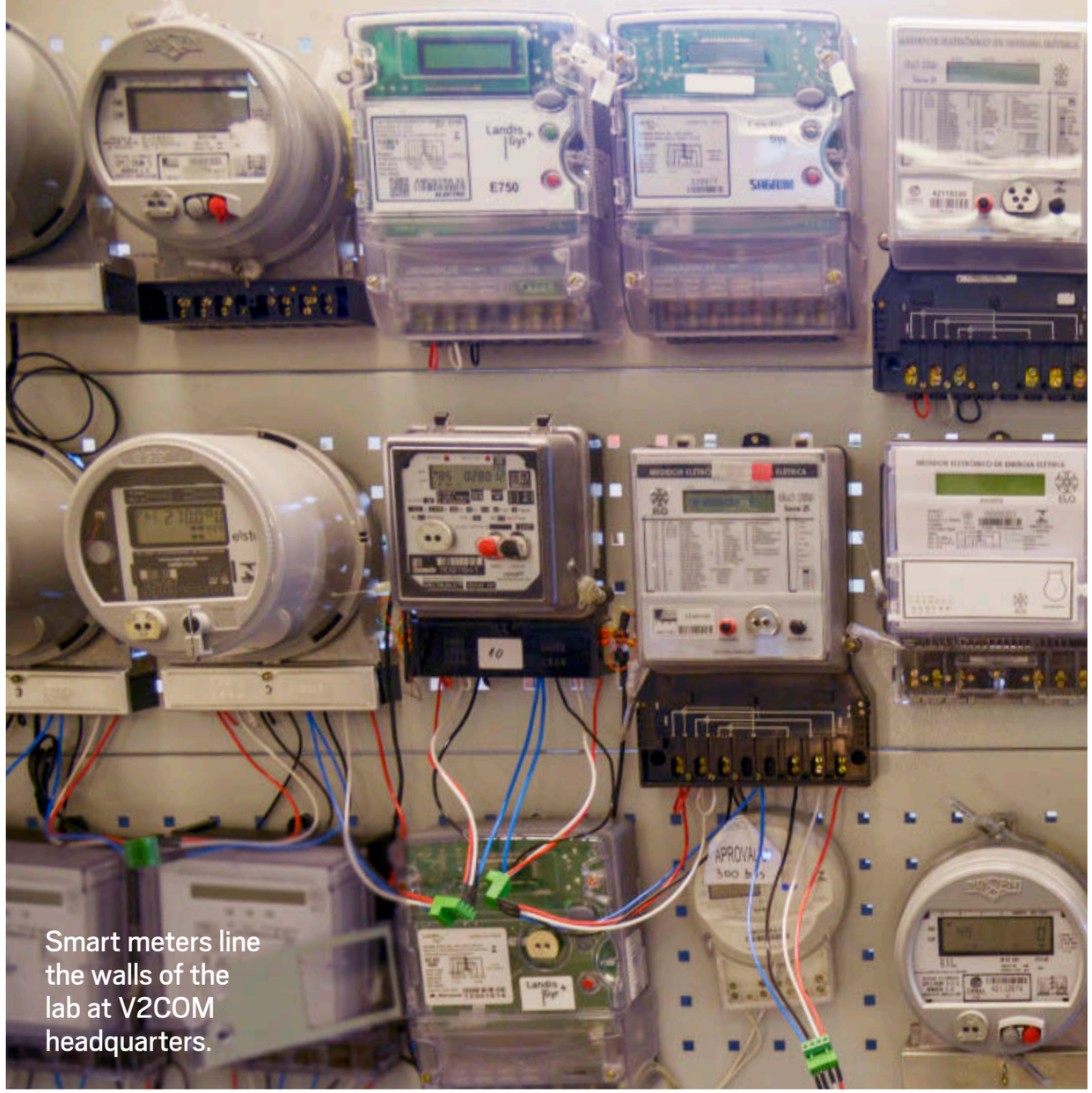
"A complex set of rules governs these devices, so we need a software infrastructure that allows us to code in a way that is manageable, expandable, scalable, and not too complex," Spina says. "Our system embeds Java in all these different computing platforms, from the back-end applications to the edge devices and pole-top smart concentrators, creating an environment of distributed processing."

Today V2COM offers hardware, software, and services that can reduce losses and increase water and energy efficiency, connecting more than 1 million devices on its platform. Spina believes the electric power industry is ripe for disruption, as analog distribution systems, control mechanisms, and usage meters are replaced by computerized devices. Smart grids can support millions of remote intelligent devices that handle complex operations in the field, all connected to a data center through smart concentrators. These massive, intelligent networks allow electric companies to better manage the flow and consumption of energy, with less equipment and fewer IT and field resources. And for that, Spina explains, you need dynamic management of energy distribution—something for which Java is innately suited.

LAYING THE GROUNDWORK FOR SMART CITIES

In the energy sector, V2COM's innovative solutions use Gemalto M2M's Cinterion modules with Oracle Java ME Embedded to share energy usage data over cellular wireless networks. These modules communicate with V2COM's Intelligenceware Suite, which uses Oracle Java SE Embedded, Oracle Utilities Meter Data Management, and GlassFish Server Open Source Edition to transmit meter and sensor data to back-end utility systems.

One of V2COM's most successful projects was for Elektro, an energy distribution company in Brazil controlled by Iberdrola Group. Elektro, which serves 2.3 million clients and reaches 5.5 million people, enlisted V2COM to automate its commercial and industrial metering processes. Together the companies created a flexible solution that is modernizing electrical power delivery in Latin America. The project includes intelligent communication modules connected to legacy electronic meters through the public cellular network. In addition to improving energy efficiency and decreasing energy loss, the solution has helped Elektro improve its remote monitoring capabilities and respond to incidents faster. It also helps the company detect fraud and field installation problems that weren't visible before.



Smart meters line the walls of the lab at V2COM headquarters.

V2COM and Elektro consider the venture to be an important step toward a much more extensive set of distributed computing solutions. "The Elektro engagement is exciting because it lays the groundwork for smart-city projects," says Leonardo De Moura Rocha Lima, chief technology officer at V2COM. "Cities face huge challenges including congestion, pollution, blackouts, crime, debt, and rising costs, while competing with each other for investments, jobs, and talent. Technology is the best answer to manage these challenges."

V2COM's smart grid solutions are the first vector for smart cities that will have smart street lighting, smart traffic control, smart video monitoring, and other remote intelligent devices sharing the same architecture and working in an orchestrated fashion. Elektro is using V2COM's solution as a test bed for these highly automated cities.

V2COM and the Java Community Process

All Java technology is developed through the Java Community Process (JCP), an open, industry-led organization chartered with evolving the Java platform. V2COM is on the JCP Executive Committee, through which it has influenced the Java ME 8 specification.

V2COM owes its involvement with the JCP to a fortuitous seating arrangement that placed V2COM's chief technology officer, Leonardo De Moura Rocha Lima, next to Terrence Barr, principal product manager for Java ME, at an Oracle Technology Network community event for Java developers in Brazil a few years ago.

"As we chatted about V2COM's product strategy, technical requirements, and use cases, it became clear to me that the company's general concept and approach to the solution was something that Oracle should support in its architecture," Barr recalls.

Barr and Lima kept in touch. When the JCP program office later contacted Barr looking for candidates for the Executive Committee, he suggested V2COM as a likely member. "They have a lot of experience with embedded Java," Barr says. "We like their vision for the Internet of Things, and we are confident the community will benefit from their contributions to the standardization process."

For example, V2COM's work with modularization makes it easier to partition an application across a wide range of remote devices and reuse the code in different projects. It also simplifies software updates because developers only have to update a single module, rather than multiple modules tailored to each type of field device.

"V2COM's input has been extremely valuable," Barr says. "They are helping adapt the Java ME platform to support a variety of Internet of Things use cases."

"The backbone of tomorrow's smart cities is the distribution network of the electrical utility," Lima explains. "Java is the connecting fabric."

Spina predicts that Brazil will deploy as many as 60 million smart meters over the next 10 years. "Java is a good fit for this massive project because it is available both on the embedded devices and on the server side," Lima adds. "It's the same technology, the same language, and the same feature set."

JAVA 8 IN THE WORLD OF CONNECTED DEVICES

V2COM has come a long way since 2002. Today, with more than 1 million devices connected to its platform, the company is a significant contributor to the Internet of Things, which Gartner predicts will include 26 billion connected devices by 2020.

Beyond the modernization of Brazil's energy infrastructure, says Spina, Java will play a key role in connecting and enabling the Internet of Things on a global scale. In the meantime, as V2COM's distributed computing model transforms the energy sector, embedded Java technology is establishing new patterns and precedents for many industries.

For example, just as Java took complexity out of processes and operating systems, it will eventually simplify networks and network protocols. "Lots of devices utilize proprietary networks. But with Java, we can standardize," says Spina. "The almost 10 million Java developers can easily start programming for the Internet of Things without having to learn a lot of complex network protocols. Today programming mobile apps is cool. Tomorrow it will be Internet of Things solutions."

New features in Java ME 8 are particularly important to this development. "Java 8 permits a distinct separation of services," Spina continues, "along with modularization so we can run different services on the same virtual machines, with a clearer boundary between them."

It's no coincidence that these features that are so valuable in delivering smart grid technology are now part of Java ME 8. V2COM shares its expertise as a member of the Executive Committee of Oracle's Java Community Process (JCP). Through the JCP, V2COM contributed key concepts to the Java ME 8 specification. (See the "V2COM and the Java Community Process" sidebar.)

V2COM's work with Java took a refreshing turn when company directors realized they could use the same distributed infrastructure to accurately monitor and control refrigerator temperatures—a crucial variable in brewing beer.

"Our home-brew project is just one example of how flexible this technology can be," Lima says. "In addition to monitoring beer temperature, it could be used in medical laboratories to control the temperature of vaccines or in other sensitive environments."

Whether V2COM is monitoring electrical usage, the temperature of beer, or movements in a video field, it uses the same basic Java infrastructure to communicate between the end devices and the central controllers.

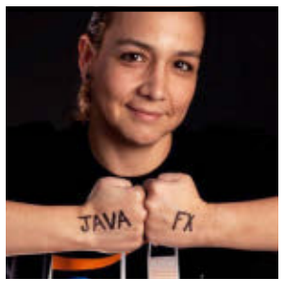
"It's the same set of Java services up to the application layer on the embedded device," Lima summarizes. "In the fast-growing Internet of Things, Java can be the common connecting technology for all these possible use cases." `</article>`

Left to right: Guilherme Vallerini, Lima, and Spina taste the beer they make at V2COM's headquarters.

MORE ON TOPIC:

Java 8 Is Here

Based in Santa Barbara, California, **David Baum** and **Ed Baum** write about innovative businesses, emerging technologies, and compelling lifestyles.



How to Become an Embedded Developer in Minutes

Jump into programming the next big thing using embedded Java.

ANGELA CAICEDO

BIO

In my 10 years as a Java evangelist, I've never been more excited about a Java release. Not only is Java 8 a great release with cool new features in the language itself, but it also provides great things on the embedded side, great optimizations, and really small specifications. If you are a Java developer and ready to jump with me into the new wave of machine-to-machine technology—better said, into programming the next big thing—let's get started with the Internet of Things (IoT).

Before you get started with embedded programming, you need to understand exactly what you are planning to build and where you are planning to run your application. This is critical, because there are different flavors of embedded Java that will fit your needs perfectly.

If you are looking to build applications similar to the ones you run on your desktop, or if you are looking for great UIs, you need to take a look at Oracle Java SE Embedded, which is derived from Java SE. It supports the same platforms and functionality as Java SE. Additionally, it provides specific features and supports additional platforms, it has small-footprint Java runtime environments (JREs), it supports headless configurations, and it has memory optimizations.

SHARED ROOTS

Oracle Java SE Embedded supports the same platforms and functionality as Java SE.

On the other hand, if you are looking for an easy way to connect peripherals, such as switches, sensors, motors, and the like, Oracle Java ME Embedded is your best bet. It has the Device Access API, which defines APIs for some of the most common peripheral devices that can be found in embedded platforms: general-purpose input/output (GPIO), inter-integrated circuit (I^2C) bus, serial peripheral interface (SPI) bus, analog-to-digital converter (ADC), digital-to-analog converter (DAC), universal asynchronous receiver/transmitter (UART), memory-mapped input/output (MMIO), AT command devices, watchdog timers, pulse counters, pulse-width modulation (PWM) generators, and generic devices.

The Device Access API is not present in Oracle Java SE Embedded (at least not yet), so if you still want to use Oracle Java SE Embedded and also work with peripherals, you have to rely on external APIs, such as Pi4J.

In term of devices, embedded Java covers an extremely wide range from conventional Java SE desktop and server platforms to the STMicroelectronics STM32F4DISCOVERY board, Raspberry Pi, and Windows. For this article, I'm going to use the Raspberry Pi, not only because it's a very powerful credit-card-size single-board computer, but also because it's very affordable. The latest model costs only US$35.

Getting Your Raspberry Pi Ready

In order to boot, the Raspberry Pi requires a Linux image on a Secure Digital

Java 8 Is Here

(SD) memory card. There is no hard drive for the computer. Instead, an SD card stores the Linux image that the computer runs when it is powered on. This SD memory card also acts as the storage for other applications loaded onto the card.

To configure your SD card, perform the following steps:

1. Format your SD card.
2. Download Raspbian, a free operating system based on Debian that is optimized specifically for the Raspberry Pi hardware.
3. Create a bootable image. You can use applications such as Win32 Disk Imager to easily create your image.

Once you have your SD card ready, you can turn on your Raspberry Pi. The first time you boot your Raspberry Pi, it will take you to the Raspberry Pi Software Configuration Tool to perform some basic configuration. Here are the additional tasks you should perform:

1. Ensure that all the SD card storage is available to the operating system by selecting the Expand Filesystem option.
2. Set the language and regional setting to match your location by selecting the Internationalisation option.
3. Set up the Raspberry Pi as a headless (that is, without a monitor attached) embedded device by allowing access via Secure Shell (SSH). To configure this, select the Advanced option from the main menu.
4. Ensure that the Raspberry Pi always has the same IP address by setting a static IP address. Although this is not required, I find it very useful when using the Raspberry Pi headless. To set up the static IP address, edit the /etc/network/interfaces file. An example of this file is shown in **Figure 1**.

Now you are ready to connect to the Raspberry Pi. One option is to use PuTTY. An example of how to connect is shown in **Figure 2**.

```
  GNU nano 2.2.6         File: /etc/network/inter

auto lo

iface lo inet loopback
#iface eth0 inet dhcp

# Added to make this Raspberry use a static IP
iface eth0 inet static
address 192.168.1.105
network 192.168.1.0
netmask 255.255.255.0
broadcast 192.168.1.255
gateway 192.168.1.1

allow-hotplug wlan0
iface wlan0 inet manual
wpa-roam /etc/wpa_supplicant/wpa_supplicant.conf
iface default inet dhcp
```

Figure 1

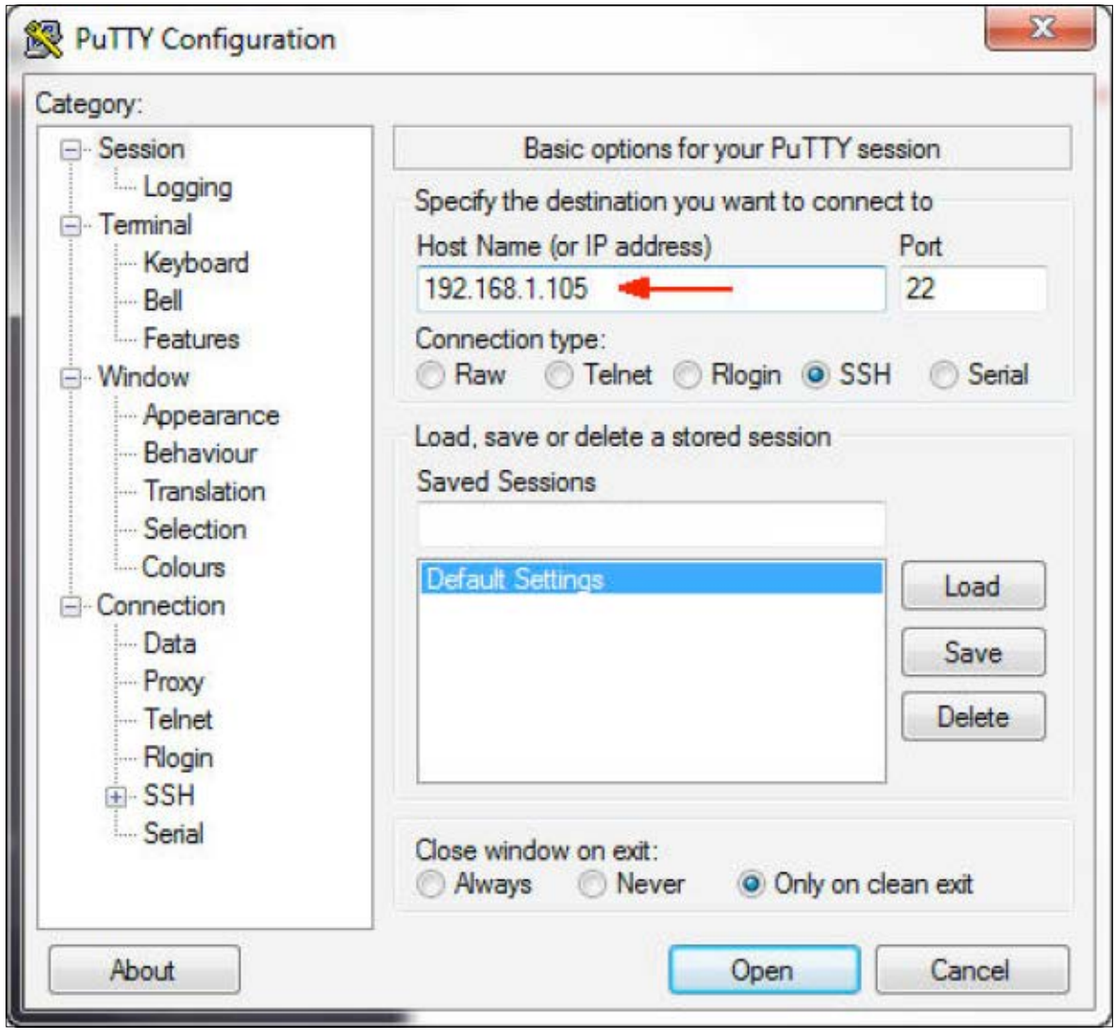


Figure 2

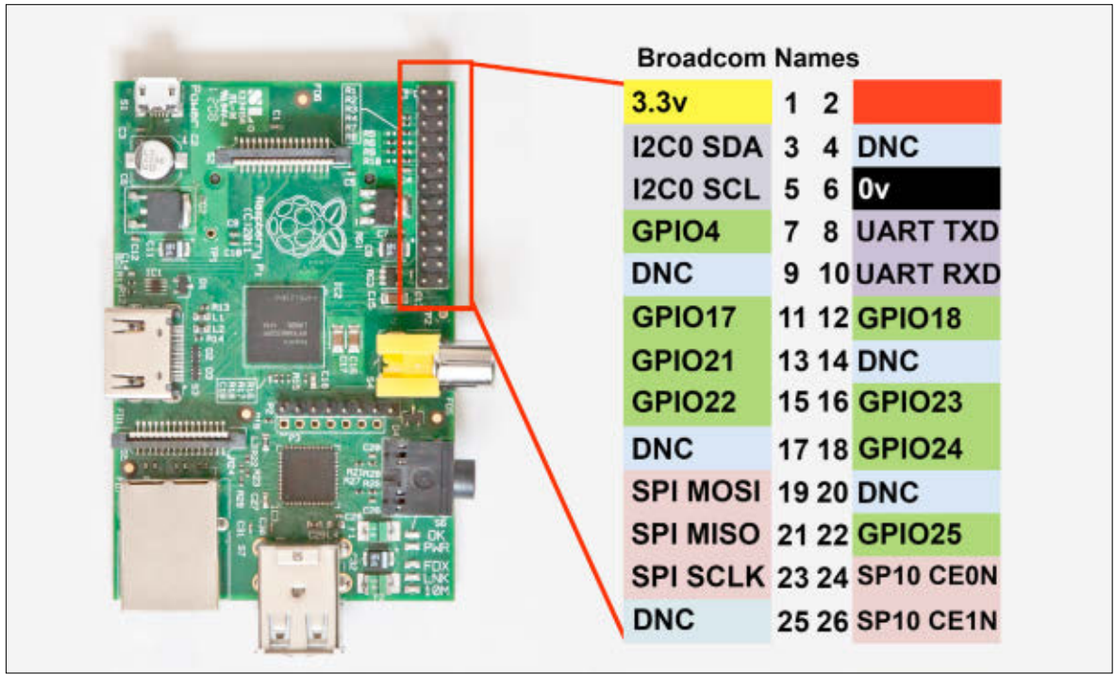


Figure 3

LISTING 1

```
public class Midlet extends MIDlet {

  @Override
  public void startApp() {
    System.out.println("Started...");
  }

  @Override
  public void destroyApp(boolean unconditional) {
    System.out.println("Destroyed...");
  }
}
```

Download all listings in this issue as text

Installing Embedded Java on the Raspberry Pi

Now, this is where you decide the kind of application you want to run on your device. I personally love having fun with peripherals, so in this article I'm going to use Oracle Java ME Embedded, so I can leverage the Device Access API. But remember, you can also run Oracle Java SE Embedded on your Raspberry Pi.

Installing the Oracle Java ME Embedded binary on the Raspberry Pi is very simple. Just use FTP to transfer the Raspberry Pi distribution zip file from your desktop to the Raspberry Pi over the SSH connection. Then unzip the file into a new directory, and you're done.

Putting Things Together

One great option for creating your embedded application is using the NetBeans IDE with the Java ME SDK. Combining these two allows you to test your application, even before you run it on your device, by using an emulator. You will be able to automatically transfer your code and execute it on your Raspberry Pi, and you can even debug it on the fly. All you need to ensure is that the Java ME SDK is part of the Java platforms on your IDE. You need to enable the SDK in the NetBeans IDE by selecting Tools->Java Platforms, clicking Add Platform, and then specifying the directory that contains the SDK.

In order to remotely manage your embedded applications on your Raspberry Pi, you need to have the Application Management System (AMS) running. Through SSH, simply execute the following command:

```
pi@raspberrypi sudo
javame8ea/bin/usertest.sh
```

Your First Embedded App

Oracle Java ME Embedded applications look exactly like other Java ME applications. **Listing 1** shows the simplest example you can have.

Your application must inherit from the MIDlet class, and it should override two lifecycle methods: startApp and destroyApp. These two methods will be invoked when the application gets started and just before it gets destroyed. The code in **Listing 1** just prints a text message on the device console.

Turning on the Lights!

Now let's do something a bit more interesting, such as turning an LED on and off by pressing a switch. First, let's have a look at the GPIO pins on the Raspberry Pi (see **Figure 3**).

The GPIO connector has a number of different types of connections on it:

- GPIO pins
- I²C pins
- SPI pins
- Serial Rx and Tx pins

This means that we have several options for where to connect our

LED and switch; any of the GPIO pins will work fine. Just make a note of the pin number and ID for each device, because you will need this information to reference each device from your code.

Let's do some basic soldering and create the circuit shown in **Figure 4**. Note that we are connecting the LED to pin 16 (GPIO 23) and the switch to pin 11 (GPIO 17). A couple of resistors are added to make sure the voltage levels are within the required range.

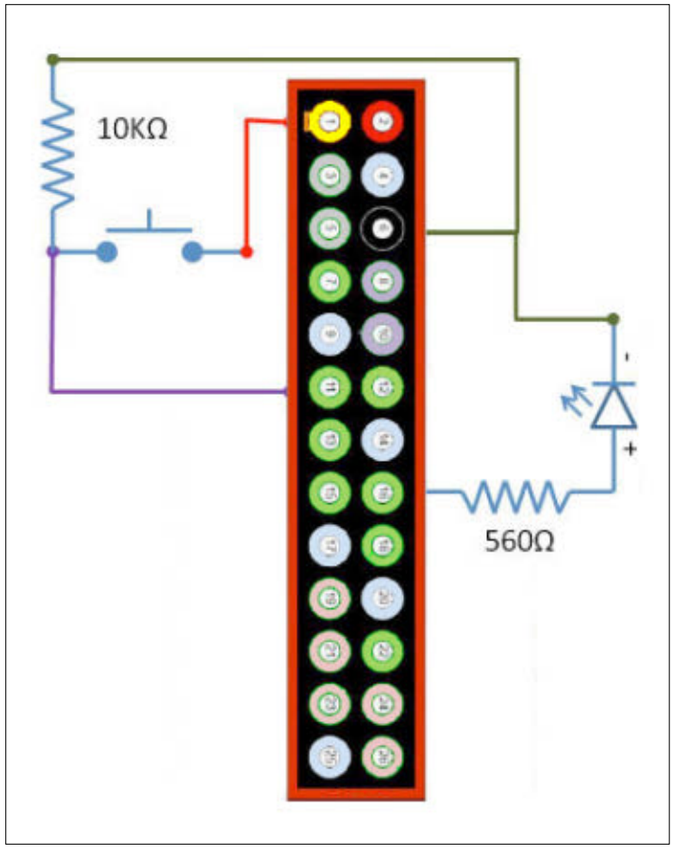


Figure 4

Now let's have a look at the program. In the Device Access API, there is a class called PeripheralManager that allows you to connect to any peripheral (regardless of what it is) by using the peripheral ID, which simplifies your coding a lot. For example, to connect to your LED, simply use the static method open, and provide the pin ID 23, as shown in **Listing 2**. Done!

To change the value of the LED (to turn it on and off), use the setValue method with the desired value:

```
// Turn the LED on
led1.setValue(true);
```

It really can't get any easier.

To connect the switch, we could potentially use the same open method on PeripheralManager, but because we would like to set some configuration information, we are going to use a slightly different approach. First, we create a GPIOPinConfig object (see **Listing 3**), which contains information such as the following:

- Device name
- Pin number
- Direction: input, output, or both
- Mode: pull-up, pull-down, push-pull, or open-drain

LISTING 2 LISTING 3 LISTING 4 LISTING 5 LISTING 6

```
private static final int LED1_ID = 23;
...
GPIOPin led1 = (GPIOPin) PeripheralManager.open(LED_ID);
```

Download all listings in this issue as text

- Trigger: none, falling-edge, rising-edge, both edges, high-level, low-level, or both levels
- Initial value

Then, we call the open method using this configuration object, as shown in **Listing 4**.

We can also add listeners to the pins, so we will get notified every time a pin value changes. In our case, we want to be notified when the switch value changes, so we can set the LED value accordingly:

```
button1.setInputListener(this);
```

Then implement the valueChanged method that will be called when events occur, as shown in **Listing 5**.

It's also important that you close the pins when you are done, and also make sure you turn your LED off (see **Listing 6**).

The whole class can be found here.

Now, all we are missing is the main MIDlet that invokes our code. The startApp method shown in **Listing 7** will create an object to control our two GPIO devices (LED and switch) and listen to our inputs, and the stopApp method will make sure everything is closed (stopped) properly.

Sensing Your Environment

LEDs and switches are nice, but what is really interesting is when we start sensing our surrounding environment. In the following example, I want to show how to get

started with sensors that use the I²C protocol.

I²C devices are perhaps the most widely available devices, and their biggest advantage is the simplicity of their design. I²C devices use only two bidirectional open-drain lines: Serial Data Line (SDA) and Serial Clock Line (SCL).

Devices on the bus will have a specific address. A master controller sets up the communication with an individual component on the bus by sending out a start request on the SDA line, followed by the address of the device. If the device with that address is ready, it responds with an acknowledge request. Data is then sent on the SDA line, using the SCL line to control the timing of each bit of data.

NOW IS THE TIME

If you are a Java developer and you are ready to jump into the new wave of machine-to-machine technology—**let's get started with the Internet of Things (IoT).**

When the communication with a single device is complete, the master sends a stop request. This simple protocol makes it possible to have multiple I²C devices on a single two-line bus.

Getting I²C Working on the Raspberry Pi

If you look at the Raspberry Pi pins again (see **Figure 3**), you will see that there are two pins for I²C: pin 3 is the data bus (SDA) and pin 5 is the clock (SCL). I²C is not enabled by default, so there are a few steps we need to follow in order to make it available to our application.

First, use a terminal to connect to your Raspberry Pi, and then add the following lines to the /etc/modules file:

```
i2c-bcm2708
i2c-dev
```

It's very useful to have the i2c-tools package installed, so the tools will be handy for detecting devices and making sure everything works properly. You can install the package using the following commands:

```
sudo apt-get install
python-smbus
sudo apt-get install i2c-tools
```

Lastly, there is a blacklist file called /etc/modprobe.d/raspi-blacklist.conf; by default SPI and I²C are part of this blacklist. What this means is that unless we remove or comment out these lines, I²C and SPI won't work on your Raspberry Pi. Edit the file and

LISTING 7

```
public class Midlet extends MIDlet{

 private MyFirstGPIO gpioTest;

 @Override
 public void startApp() {
  gpioTest = new MyFirstGPIO();
  try {
   gpioTest.start();
  } catch (PeripheralTypeNotSupportedException |
        PeripheralNotFoundException|
        PeripheralConfigInvalidException |
        PeripheralExistsException ex) {
    System.out.println("GPIO error:"+ex.getMessage());
  } catch (IOException ex) {
    System.out.println("IOException: " + ex);
  }
 }

 @Override
 public void destroyApp(boolean unconditional) {
  try {
    gpioTest.stop();
  } catch (IOException ex) {
     System.out.println("IOException: " + ex);
   }
  }
 }
}
```

Download all listings in this issue as text

remove the following lines:

```
blacklist spi-bcm2708
blacklist i2c-bcm2708
```

Restart the Raspberry Pi to make sure all the changes are applied.

Adding Sensors

The BMP180 board from Bosch Sensortec is a low-cost sensing solution for measuring barometric pressure and temperature. Because pressure changes with altitude, you can also use it as an altimeter. It uses the I²C protocol and a voltage in the range of 3V to 5V, which is perfect for connecting to our Raspberry Pi.

Let's go back to soldering to connect the BMP180 board to your Raspberry Pi using the diagram shown in **Figure 5**. Normally, when you use an I²C device, a pull-up resistor is required for the SDA and SCL lines. Fortunately, the Raspberry Pi provides pull-up resistors, so a simple connection is all you need.

Once we connect the board to the Raspberry Pi, we can check whether we can see an I²C device. On your Raspberry Pi run the following command:

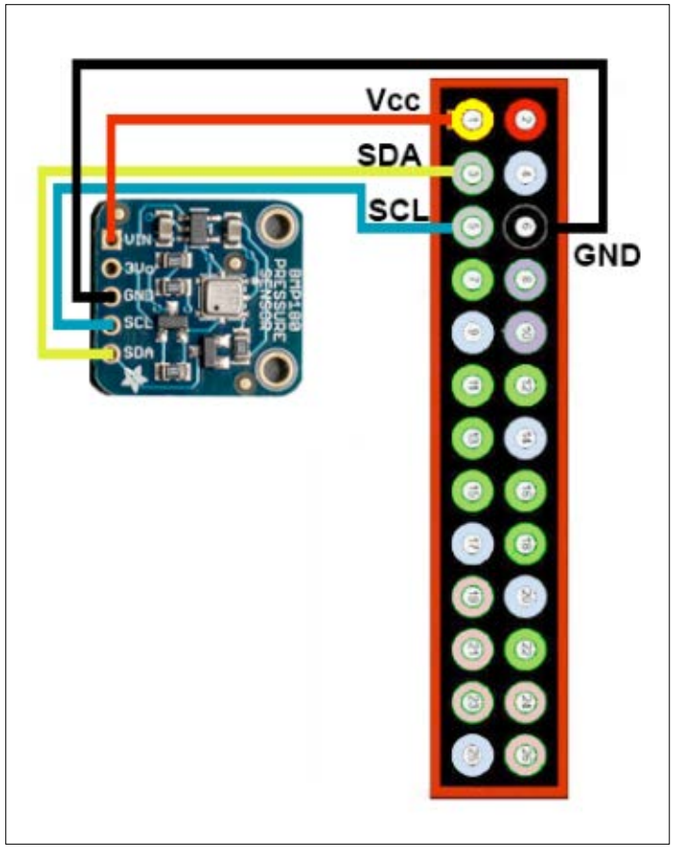


Figure 5

```
sudo i2cdetect -y 1
```

You should be able to see your device in the table. **Figure 6** shows two I²C devices: one at address 40 and one at address 70.

Using an I²C Device to Get the Temperature

There are a few things you need to know before you programmatically connect to an I²C device:

- What is the device's address? I²C uses 7 bits for a device address, and the Raspberry Pi uses I²C bus 1.

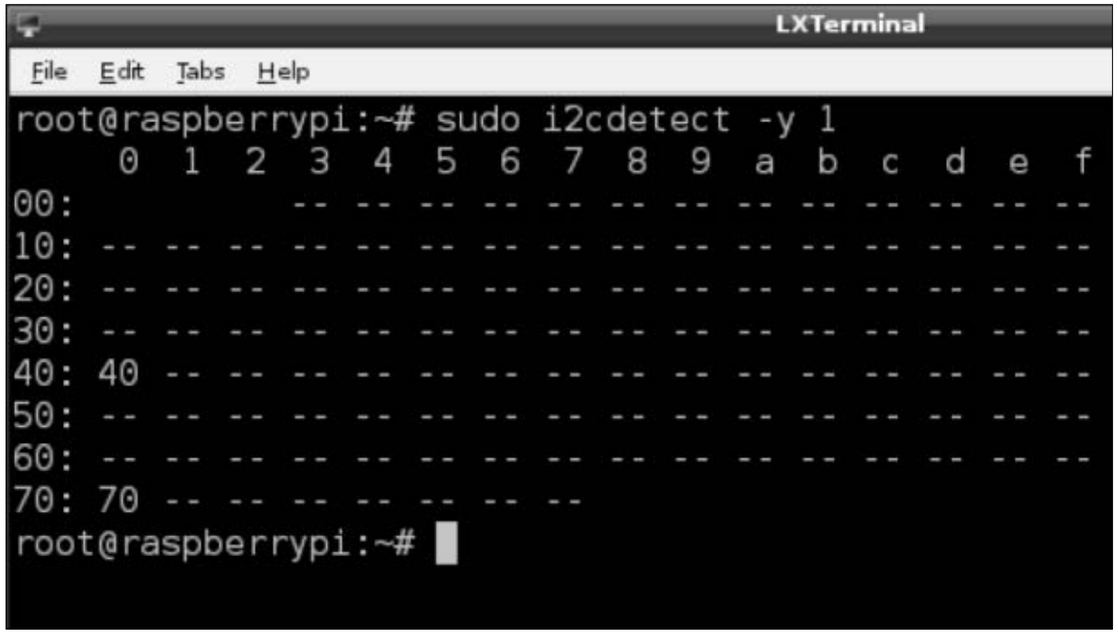


Figure 6

- What is the register's address? In our case, we are going to read the temperature value, and this register is located at address 0xF6. (This is specific to the BMP180 board.)
- Do you need to configure any control registers to start sensing? Some devices are in a sleep mode by default, which means they won't be sensing any data until you wake them up. The control register for our device is address 0xF4. (This is specific to the BMP180 board.)
- What is the clock frequency for your device? In our case, the BMP180 board uses 3.4 MHz.

Listing 8 defines the values for the BMP180 as static variables to be used later in the code.

Once again, the way we connect programmatically to the device is using the PeripheralManager's static method open. In this case, we will provide an I2CDeviceConfig object that is specific to an I²C device (see **Listing 9**). The I2CDeviceConfig object allows us to specify the device's bus, address, address size (in bits), and clock speed.

To take a temperature reading, we need to follow three steps:

1. Read calibration data from the device, as shown in **Listings 10a** and **10b**. This is specific to the BMP180 board, and you might not need to perform this step when using other temperature sensors.
2. Write to a control register on the device to initiate a tem-

LISTING 8 LISTING 9 LISTING 10a LISTING 10b LISTING 11 LISTING 12

```
//Raspberry Pi's I2C bus
private static final int i2cBus = 1;
// Device address
private static final int address = 0x77;
// 3.4MHz Max clock
private static final int serialClock = 3400000;
// Device address size in bits
private static final int addressSizeBits = 7;
...

// Temperature Control Register Data
private static final byte controlRegister = (byte) 0xF4;
// Temperature read address
private static final byte tempAddr = (byte) 0xF6;
// Read temperature command
private static final byte getTempCmd = (byte) 0x2E;

...
// Device object
private I2CDevice bmp180;
```

Download all listings in this issue as text

perature measurement (see **Listing 11**).

3. Read the uncompensated temperature as a two-byte word, and use the calibration constants to determine the true temperature, as shown in **Listing 12**. (Once again, this is specific to this sensor.)

Finally, the temperature in Celsius will be stored in the celsius variable. You can find the entire program here.

As an exercise, you can extend the program to read the pressure, the altitude, or both.

Conclusion

This article took you through the steps required to start creating embedded Java applications by showing real examples of how to use GPIO and the I²C devices. Now it's your turn to find more devices that you would like to connect to your Raspberry Pi so you can have fun with embedded Java on the Raspberry Pi. `</article>`

MORE ON TOPIC:

Java 8 Is Here

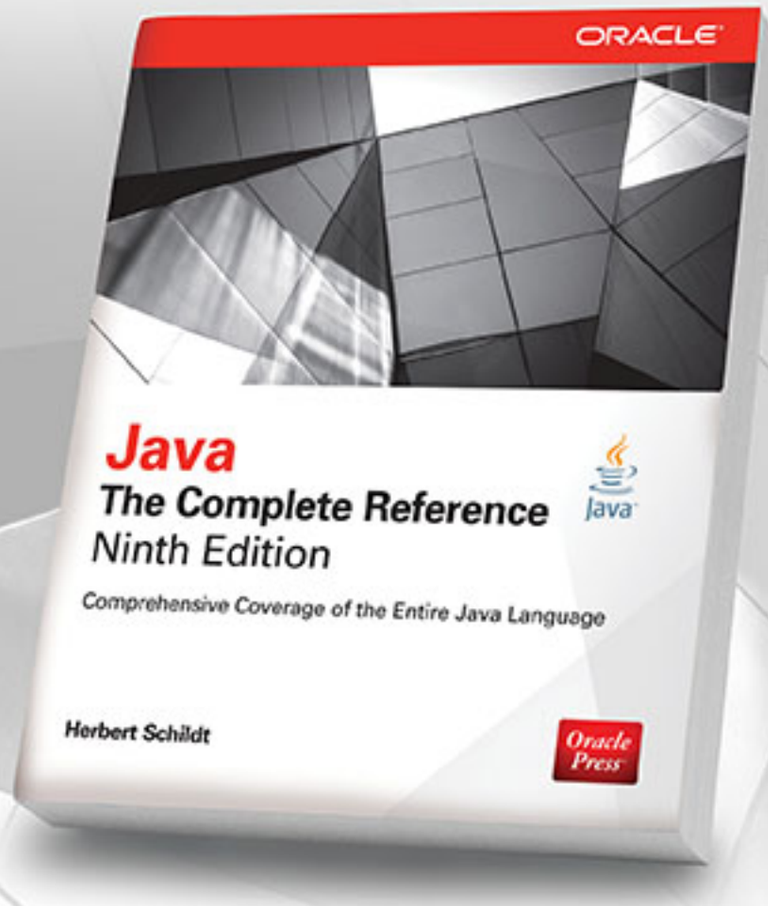
LEARN MORE

- Getting Started with Oracle Java ME Embedded 3.3 on the Keil Board

Oracle Press™

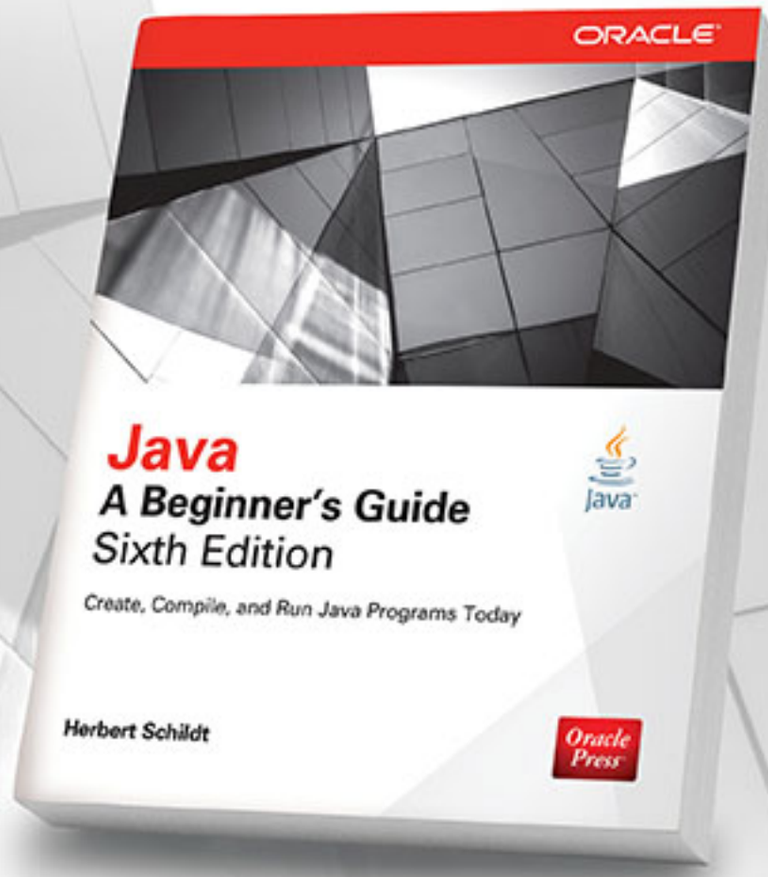
Your Destination for Java Expertise

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



Java: The Complete Reference, Ninth Edition
Herbert Schildt

Fully updated for Java SE 8, this definitive guide explains how to develop, compile, debug, and run Java programs.



Java: A Beginner's Guide, Sixth Edition
Herbert Schildt

This fast-paced Java programming tutorial is revised to cover Java SE 8.

Quick Start Guide to JavaFX
J.F. DiMarzio

Create and deploy dynamic enterprise client applications in no time.

Oracle Press™

Available in print and as eBooks

Visit us at OraclePressBooks.com • Follow us @OraclePress and Like us OraclePress

JOSH JUNEAU

BIO

Part 3

Three Hundred Sixty–Degree Exploration of Java EE 7

Learn how to use the Java API for Batch Processing, utilize Java Message Service, and create a chat client using WebSockets.

This article is the third in a three-part series demonstrating how to use Java EE 7 improvements and newer web standards, such as HTML5, WebSocket, and JavaScript Object Notation (JSON) processing, to build modern enterprise applications.

In this third part, we will improve the movieplex7 application, which we started in the first two articles, by adding the ability to view ticket sales, accumulate movie points, and chat with others.

Note: The complete source code for the application can be downloaded here.

Overview of the Demo Application

In the first two articles, we learned how to download, install, and configure NetBeans and GlassFish 4, which will also be used for building and deploying the application in this article. We built the application from the ground up, using JavaServer Faces (JSF) 2.2 and making use of new technology, such as the Faces Flow for navigation. We demonstrated how to utilize Java Persistence API (JPA) 2.1, Java API for RESTful Web Services (JAX-RS), and JSON Processing (JSON-P) 1.0 for implementing business logic and data manipulation.

For the remainder of this article, please follow along using the Maven-based project that you created using NetBeans IDE in Part 1 and Part 2 of this series.

Movie Ticket Sales

Java API for Batch Processing 1.0 (JSR 352) will be used in this section to import movie ticket sales for each show and populate the database by reading cumulative sales from a comma-separated values (CSV) file.

Batch processing is a method for executing a series of tasks or *jobs*. Most often, these jobs do not require intervention, and they are bulk-oriented and long-running. For details regarding batch processing terminology, please refer to the "Java EE 7 Tutorial."

Generate classes. First, let's create the classes that will be used to process movie sales and persist them to the database.

1. Right-click **Source Packages**, select **New** and **Java Package**, and then specify the name of the package as org.glassfish.movieplex7.batch. Click **Finish**.
2. Right-click the newly created package, select **New** and **Java Class**, and then provide the name SalesReader. Change the class definition to extend AbstractItemWriter, and resolve imports.

   AbstractItemReader implements the ItemReader interface, which defines methods for reading a stream of items.
3. Add @Named and @Dependent as class-level annotations, and resolve imports.

   The @Named annotation allows the bean to be injected into the job XML, and @Dependent makes the bean available for injection.
4. Add the following field

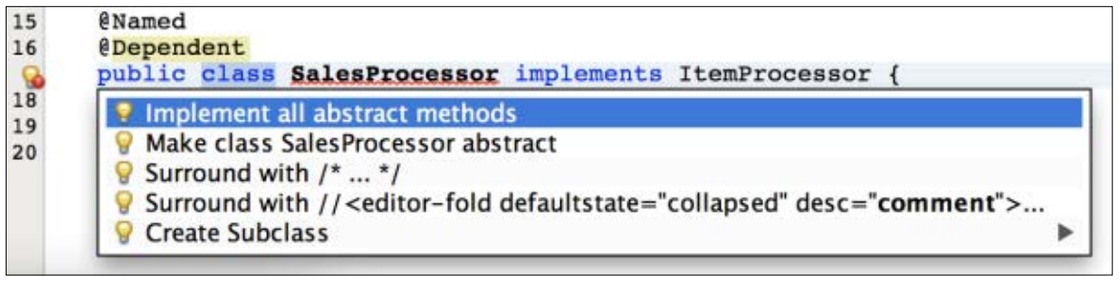


Figure 1

to the class to declare the reader:

```
private BufferedReader reader;
```

5. Provide an implementation for reading the CSV file by adding the open method shown in **Listing 1**.

   For this example, the CSV file must be placed within the META-INF directory of the application, which resides at the root of the application source packages.
6. Override the readItem method and replace with the code shown in **Listing 2**. Resolve imports.

   The readItem method is responsible for reading the next item from the stream, with a null indicating the end of the stream.
7. Add a new Java class to the org.glassfish.movieplex7.batch package, and name it SalesProcessor. Implement ItemProcessor by adding the following to the class definition:

```
implements ItemProcessor
```

   Implementing ItemProcessor allows the class to operate on an input item and produce an output item.
8. Add @Named and @Dependent class-level annotations, and resolve imports. Click the yellow lightbulb, and choose **Implement all abstract methods**, as shown in **Figure 1**.
9. Change the implementation of processItem to match that shown in **Listing 3** and resolve imports.

   The processItem method creates a new Sales object, and then it creates a StringTokenizer that is used to parse through each item within the object passed in. The Sales object is populated with the processed item value and then returned.

LISTING 1 LISTING 2 LISTING 3 LISTING 4

```
public void open(Serializable checkpoint) throws Exception {
    reader = new BufferedReader(
        new InputStreamReader(Thread.currentThread()
        .getContextClassLoader().getResourceAsStream
        ("META-INF/sales.csv")));
}
```

Download all listings in this issue as text

10. Create a new Java class within the org.glassfish.movieplex7.batch package, and name it SalesWriter. Extend AbstractItemWriter by adding the following to the class definition:

```
extends AbstractItemWriter
```

11. Add @Named and @Dependent class-level annotations, and then resolve imports.
12. Click the yellow lightbulb to add the import for javax.batch.api.chunk.AbstractItemWriter. Once the import has been added, click the lightbulb and choose **Implement all abstract methods** to add the writeItems method to the class.
13. To gain access to the data store, inject an EntityManager instance into the class by adding the following declaration:

```
@PersistenceContext
EntityManager em;
```

14. Replace the code within the writeItems method with a for loop, which will be used for persisting all the Sales objects that have been aggregated from the batch runtime, as shown in **Listing 4**.
15. Add the @Transactional annotation to the method to incorporate transaction control into your method. Resolve the imports.

Create a batch job. The next task is to implement a procedural task within XML. In this case, the job will consist of one chunk that contains three items. Those items are the <reader>, <processor>,

and <writer> elements, and their respective class implementations are SalesReader, SalesProcessor, and SalesWriter. To create the task within XML, use the following procedure:

1. Create a new folder to contain the batch XML by right-clicking META-INF and selecting **New** and then **Folder**. Name the folder batch-jobs. Click **Finish**.
2. Create the XML file by right-clicking the batch-jobs folder and selecting **New** and then **XML Document**. Name the file eod-sales. Click **Next** and accept all default values. Click **Finish**.

   Note the following about the XML file:
   - The item-count attribute of <chunk> indicates that there are three items to be given to the writer.
   - The skip-limit attribute of <chunk> indicates the number of exceptions that will be skipped.
   - <skippable-exception-classes/> lists the set of exceptions to be skipped by chunk processing.
3. Add the job definition to the XML by replacing the contents of the newly created file with the code shown in **Listing 5**.

Invoke the batch job. Create the implementation for invoking the batch job.

1. Right-click the org.glassfish.movieplex7.batch package, select **New** and then **Session Bean**, and then provide the name SalesBean. Click **Finish**.
2. Add the runJob method, as shown in **Listing 6**, to the new session bean.

   In the code, the BatchRuntime is used to obtain a new JobOperator. The new JobOperator is then used to start the batch process outlined within eod-sales, and it could also be used for stopping or restarting the job.
3. Add the @Named annotation to the class to make it injectable. Resolve imports.
4. Inject EntityManager into the class, as seen below:

```
@PersistenceContext
EntityManager em;
```

5. Add a method named getSalesData to the class, which will use an @NamedQuery to return all rows within the table, as shown in **Listing 7**. Resolve imports.
6. To create a view to show the sales, right-click **Web Pages** and select **New** and then **Folder**. Specify batch for the folder name. Click **Finish**, and right-click the newly created folder and select **New** and then **Faces Template Client**. Specify sales for the File Name, and browse to WEB-INF/template.xhtml to specify the template. Then click **Finish**. Replace all the content within the <ui:composition> tags with the code shown in **Listing 8**. Repair namespace prefix/URI mapping by clicking the yellow lightbulb.

   This page contains a dataTable, which will be used to display the contents of the sales file once the batch process has populated the database. It also contains two commandButtons: one to run the job and one to refresh the page.
7. Edit the template.xhtml file to incorporate a link to the sales.xhtml view by adding the code shown in **Listing 9**.
8. Finally, run the project, and click the **Sales** link in the left navigational menu, which will open the Movie Sales view. Click the **Run Job** button; the batch job will be initiated,

LISTING 5 LISTING 6 LISTING 7 LISTING 8 LISTING 9

```
<?xml version="1.0"?>
<job id="endOfDaySales" xmlns=
"http://xmlns.jcp.org/xml/ns/javaee" version="1.0">
  <step id="populateSales" >
    <chunk item-count="3" skip-limit="5">
      <reader ref="salesReader"/>
      <processor ref="salesProcessor"/>
      <writer ref="salesWriter"/>
      <skippable-exception-classes>
        <include class="java.lang.NumberFormatException"/>
      </skippable-exception-classes>
    </chunk>
  </step>
</job>
```

Download all listings in this issue as text

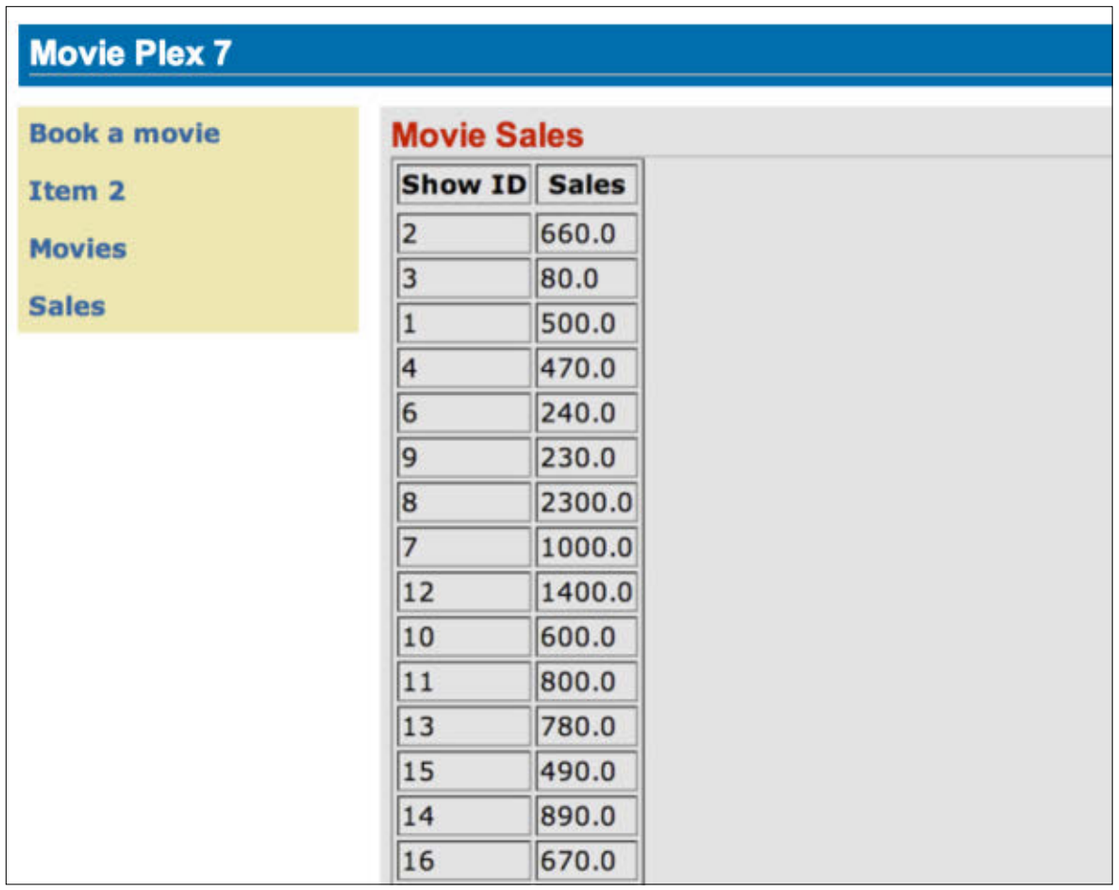


Figure 2

processing the CSV file and inserting the amounts into the SALES database table. Click the **Refresh** button to display the list of sales that have been processed, as shown in **Figure 2**.

Movie Points

To implement a movie point system for the movieplex7 application, we'll use Java Message Service (JMS) 2.0 and its new API. JMS 2.0 increases developer productivity by decreasing the amount of code and complexity that are necessary for sending and receiving messages compared with prior releases.

Note: In order to work with JMS, a topic or queue must be created within the application server container. This can be done with code or via the application server's administrative utilities. In this example, you will learn how to create a queue using code.

Create a JSF managed bean. First, let's create a JSF managed bean that will be bound to a JSF view for collecting movie points data and sending that data to the queue.

1. Create a new package in the movieplex7 application by right-clicking **Source Packages** and then selecting **New** and **Java Package**. Specify the name org.glassfish.movieplex7.points, and then click **Finish**.
2. Create a class within the new package by right-clicking the package and selecting **New** and then **Java Class**. Specify the name SendPointsBean. Implement Serializable, and add the following class-level annotations to make the bean Expression Language (EL)–injectable and session-scoped. Resolve imports.

```
@Named
@SessionScoped
```

3. Add the String field shown in **Listing 10** to the class, which will be bound to a JSF inputText field to capture point data. Generate the getters/setters for the field by right-clicking the editor pane and selecting **Insert Code** and then **Getter and Setter**. Select the field, and click **Generate**.

Note: This field uses bean validation annotations to ensure that the text entered into the field adheres to the required format.

4. Inject an instance of JMSContext and a Queue into the class by adding the following code. Then resolve imports, taking care to import javax.jms.Queue.

```
@Inject
JMSContext context;

@Resource(mappedName =
 "java:global/jms/pointsQueue")
Queue pointsQueue;
```

JMSContext is a new JMS 2.0 interface that combines

LISTING 10

```
@NotNull
@Pattern(regexp = "^\\d{2},\\d{2}",
  message = "Message format must be 2 digits, comma,

2 digits, e.g. 12,12")
private String message;
```

Download all listings in this issue as text

Connection and Session objects into a single object.

Note: Java EE–compliant application servers contain a default JMS connection factory under the name java:comp/DefaultJMSConnectionFactory, which is used when no connection factory is specified.

5. To send the message to the JMS queue, add the method shown in **Listing 11** to the class.

The JMSContext createProducer method can be used to create a JMSProducer, which provides methods to configure and send messages synchronously and asynchronously. The send method of the JMSProducer is used to send a message to the queue instance, which we will create in the next step.

6. Right-click the org.glassfish.movieplex7.points package, select **New** and then **Java Class**, and specify the name ReceivePointsBean. Implement Serializable, and add the class-level annotations shown in **Listing 12**.

Introduced in JMS 2.0, the @JMSDestinationDefinition annotation is used to reduce the administrative overhead of application configuration by programmatically provisioning required resources. In this case, the annotation is used to create a javax.jms.queue named java:global/jms/pointsQueue.

7. Inject the JMSContext and Queue resources that will be used within the class by adding the following code. Then resolve imports, being sure to import the correct classes: javax.jms.Queue and so on.

```
@Inject
JMSContext context;
@Resource(mappedName=
 "java:global/jms/pointsQueue")
 Queue pointsQueue;
```

Note: Although we are creating the queue by using the @JMSDestinationDefinition annotation, we still need to inject the queue into the class via pointsQueue to make it usable.

8. Add the receiveMessage() method shown in **Listing 13**.

This method uses the JMSContext to create a consumer for the pointsQueue and then synchronously receive a String message from the queue.

9. Add a method named getQueueSize, which is shown in **Listing 14**, for returning the number of messages currently in the queue. Resolve imports.

This method creates a QueueBrowser and then uses it to traverse through each message within the queue and add it to the total.

LISTING 11 LISTING 12 / LISTING 13 / LISTING 14 / LISTING 15

```
public void sendMessage() {
    System.out.println("Sending message: " + message);
    context.createProducer().send(pointsQueue, message);
}
```

Download all listings in this issue as text

Create a Facelet. Next, we need to create a Facelet view that will be used for entering movie points and simulating the send/receive message functionality.

1. Add a new folder by right-clicking **Web Pages** and selecting **New** and then **Folder**. Name the folder points, and then click **Finish**.
2. Create a new XHTML file named points.xhtml within that folder. Add the code shown in **Listing 15** to the view, replacing the code within the <ui:composition> elements. Click the yellow lightbulb to resolve the namespace prefix/URI mapping, as needed.

This view contains a field for entering a message, along with a commandButton that invokes the sendMessage method. When the method is invoked, the message contained within the inputText is added to the queue, and the queueSize is increased by 1. Another commandButton in the view invokes the receiveMessage method,

which will read a message from the queue and decrement the queueSize by 1.

3. Add the code shown in **Listing 16** to template.xhtml along with the other outputLink components, exposing the points.xhtml view to the application.

Run the movieplex7 application.

1. Click the **Points** link (see **Figure 3**).

   Currently, the queue contains zero messages.
2. Add the text 1212 to the field and click **Send Message**.

   An error will be displayed (see **Figure 4**). This message was produced by the bean validation that was added to SendPointsBean.
3. Insert a comma so the contents of the field is 12,12 and click **Send Message**.

   This time the error message disappears and the queue is increased by 1 (see **Figure 5**).
4. Click the **Receive Message** button, and note that the queue is decremented by 1, because a message has been read from the queue (see **Figure 6**).

Continue to send and receive messages in any order. You should notice the queue number being incremented each time a message is sent and decremented with every message received.

LISTING 16

```
<p/>
<h:outputLink value=
  "${facesContext.externalContext.requestContextPath}

/faces/points/points.xhtml">
  Points</h:outputLink>
```

Download all listings in this issue as text

Movie Chat

In this section, we will build a chat room for movieplex7 visitors using

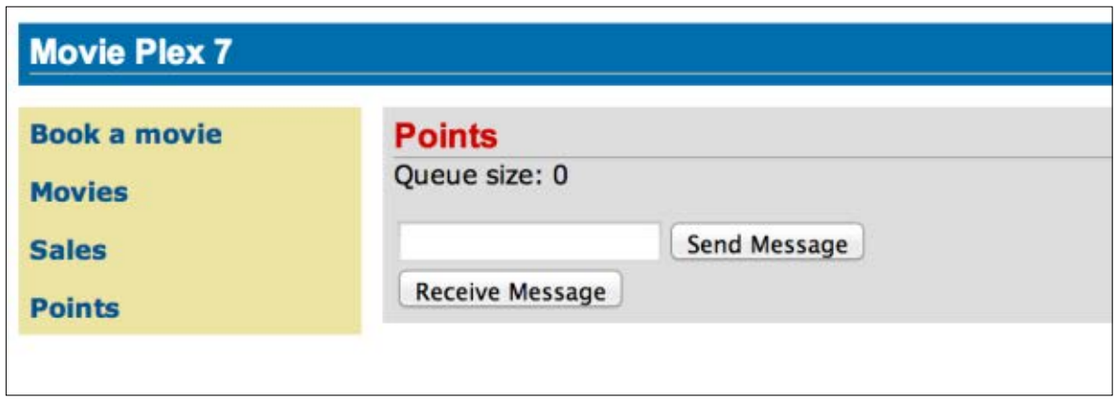


Figure 3

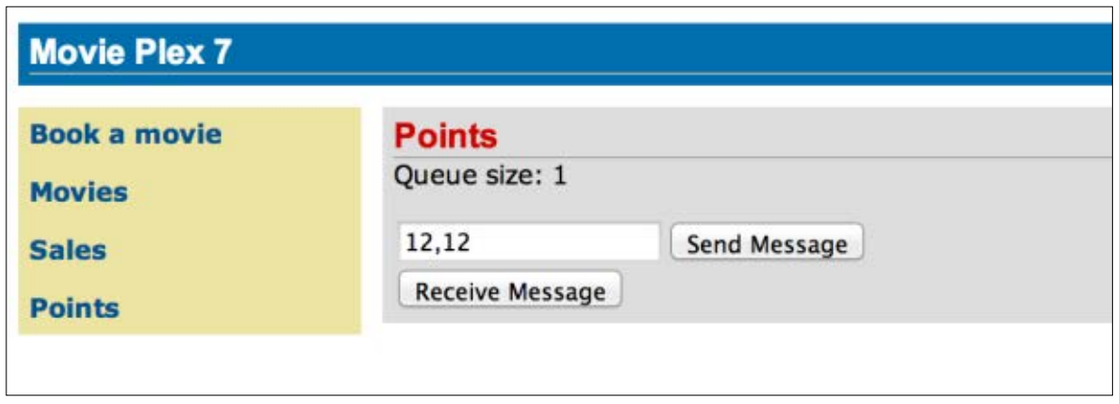


Figure 5

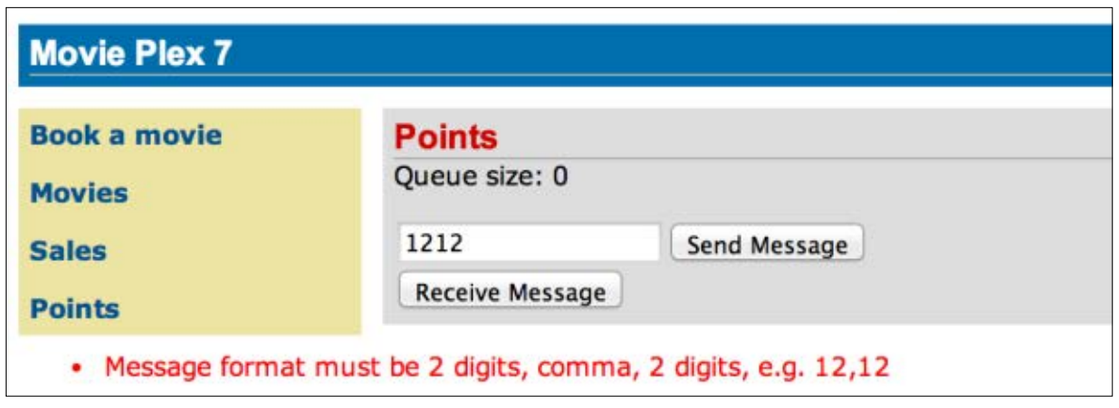


Figure 4

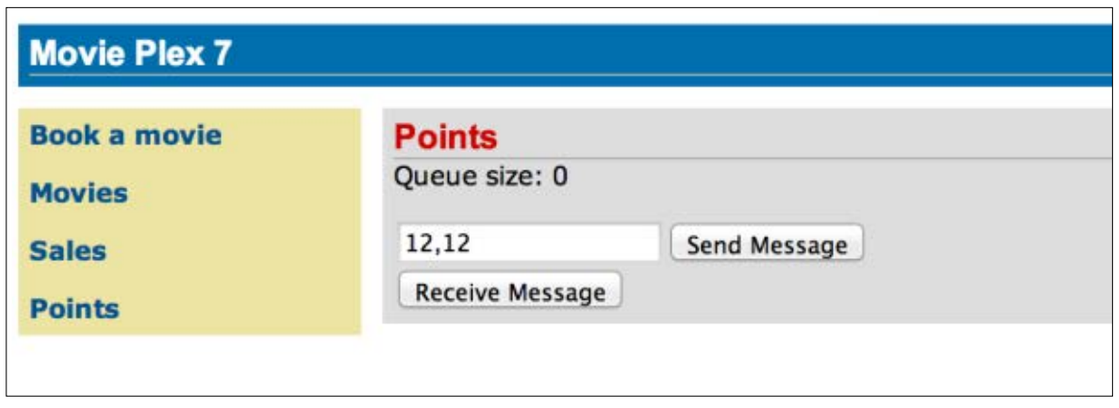


Figure 6

the WebSocket 1.0 API, which provides a full-duplex, bidirectional communication protocol over a single TCP connection that defines an opening handshake and data transfer.

A standard API for WebSocket, under JSR 356, has been added to Java EE 7, enabling developers to create WebSocket solutions using a common API, rather than customized frameworks.

Create a class to implement WebSocket communication. To begin, let's create the Java class for implementation of the WebSocket communication.

1. Create a new package by right-clicking **Source Packages** and selecting **New** and then **Java Package**. Name the package org.glassfish.movieplex7.chat.
2. Create the class by right-clicking **Source Packages** and selecting **New** and then **Java Class**. Name the class ChatServer.
3. Replace the code within the new class with the code shown in **Listing 17**. Resolve imports, being careful to ensure the import of javax.websocket.Session, among others.

The code for the ChatServer class does the following:

- @ServerEndpoint is used to indicate that this class is a WebSocket endpoint. The value attribute defines the URI to be used for accessing the endpoint.
- @OnOpen and @OnClose annotate the methods that are invoked when a session is opened or closed, respectively. The Session parameter defines the client that is requesting the initiation or termination of a connection.
- @OnMessage annotates the method that is invoked when a message is received. The first parameter is the payload of the message, and the second is the client session that defines the other end of the connection. This method broadcasts the received message to all the connected clients.

Generate the web view. Now, let's generate the web view for the chat client.

1. Right-click **Web Pages** and select **New** and then **Folder**. Provide the name chat, and click **Finish**.
2. Create a new XHTML file, specifying chatroom for the File Name, and browse to WEB-INF/template.xhtml to specify the template. Keep the other defaults and click **Finish**.
3. Replace all the content within the <ui:composition> tags with the code shown in **Listing 18**.

This code constructs the HTML form that will be used for the chat client, using two textareas that will be used to display the chat log and current user list and an input textField for entering a username and messages. There are three buttons: one to open a WebSocket session, another to send a message, and the last to disconnect the session. All three are bound via the onclick attribute to JavaScript functions, which implement the WebSocket communication. Near the end of the HTML is a <script> element,

LISTING 17 **LISTING 18**

```
public class ChatServer {

  private static final Set<Session> peers =
    Collections.synchronizedSet(new HashSet<Session>());

  @OnOpen
  public void onOpen(Session peer) {
    peers.add(peer);
  }

  @OnClose
  public void onClose(Session peer) {
    peers.remove(peer);
  }

  @OnMessage
  public void message(String message, Session client)
      throws IOException, EncodeException {
    for (Session peer : peers) {
      if (!peers.equals(client)) {
        peer.getBasicRemote().sendObject(message);
      }

    }
  }
}
```

Download all listings in this issue as text

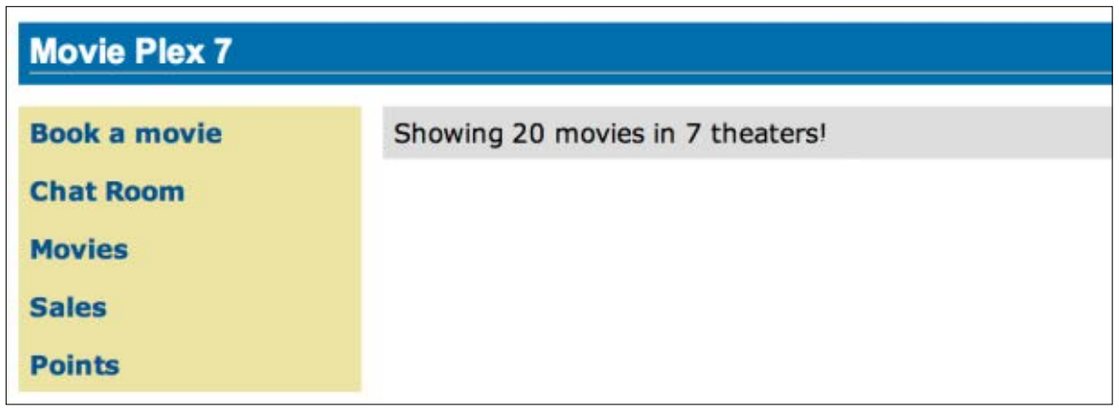


Figure 7

which includes the websocket.js JavaScript file.

4. To create the JavaScript file that will contain the WebSocket communication implementation, right-click **chat** in **Web Pages**, select **New** and then **Web** and **JavaScript File**. Name the file websocket, and click **Finish**. Edit websocket.js so it contains the code shown in **Listings 19a** and **19b**.

The websocket.js implementation constructs the endpoint URI by appending the URI specified in the ChatServer class. A new WebSocket object is then created, and each WebSocket event function is assigned to a JavaScript function that is implemented within the file. When a user clicks the Join button on the page, the username is captured, and the initial WebSocket send method is invoked, passing the username.

When messages are sent, any relevant data is passed as a parameter to the send_message function, which appends the username to the message and broadcasts to all clients. The onMessage method is invoked each time a message is received, updating the list of logged-in users. The Disconnect button on the page initiates the closing of the WebSocket connection.

5. Edit WEB-INF/template.xhtml and overwrite the outputLink element for Item 2 with the code shown in **Listing 20**.
6. Run the project and navigate to the chat room by clicking the **Chat Room** link in the left menu, as shown in **Figure 7**.

You will see the CONNECTED message presented at the

LISTING 19a / LISTING 19b / LISTING 20

```
var wsUri = 'ws://' + document.location.host
      + document.location.pathname.substr(0,
           document.location.pathname.indexOf
           ("/faces")) + '/websocket';
console.log(wsUri);
var websocket = new WebSocket(wsUri);
var username;
websocket.onopen = function(evt) {
  onOpen(evt);
};
websocket.onmessage = function(evt) {
  onMessage(evt);
};
websocket.onerror = function(evt) {
  onError(evt);
};
websocket.onclose = function(evt) {
  onClose(evt);
};
var output = document.getElementById("output");
function join() {
  username = textField.value;
  websocket.send(username + " joined");
}
function send_message() {

  websocket.send(username + ": " + textField.value);
}
```

Download all listings in this issue as text

bottom of the chat room (see **Figure 8**).

7. Open a separate browser and enter the URI localhost:8080/movieplex7, and then open the chat room in that browser. Click **Join** in one of the browser windows (we will refer to it as "browser 1"), and you will be joined to the room as Duke. You should see the user list updated in the other browser window (browser 2) as well. Join the session in browser 2 under a different name (for example, Duke2), and you should see the user list in each of the windows updated (see **Figure 9**).

Note: Chrome Developer Tools can be used to monitor WebSocket traffic.

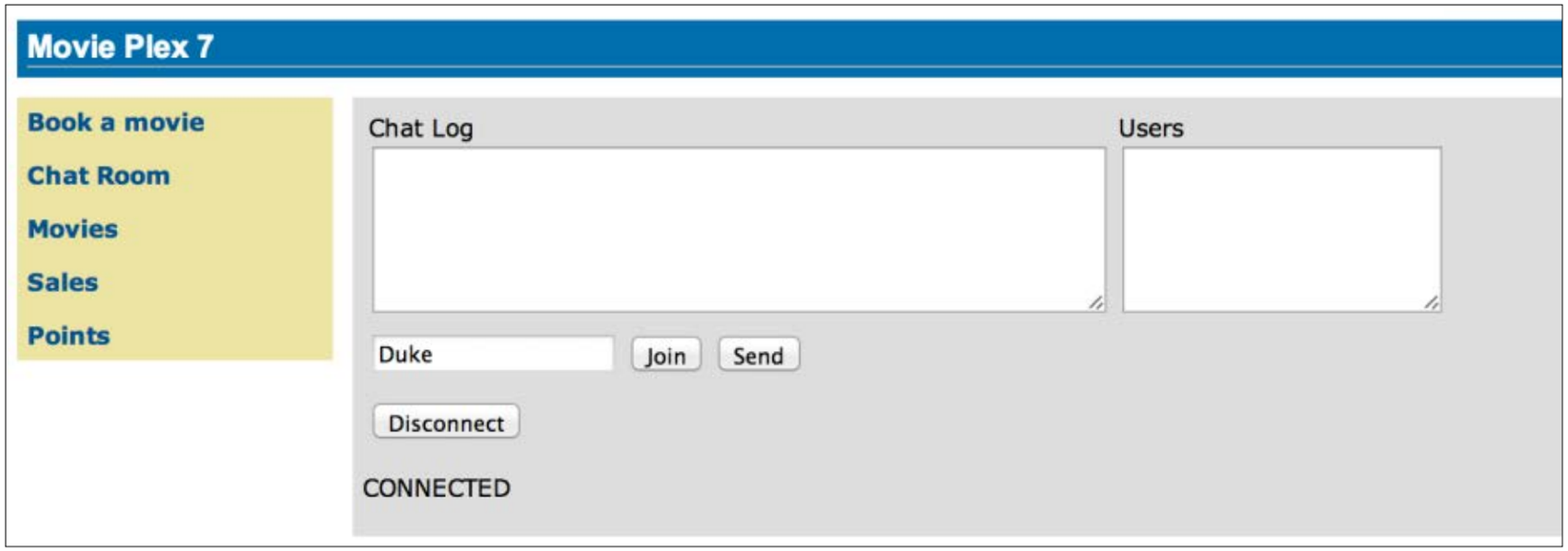


Figure 8

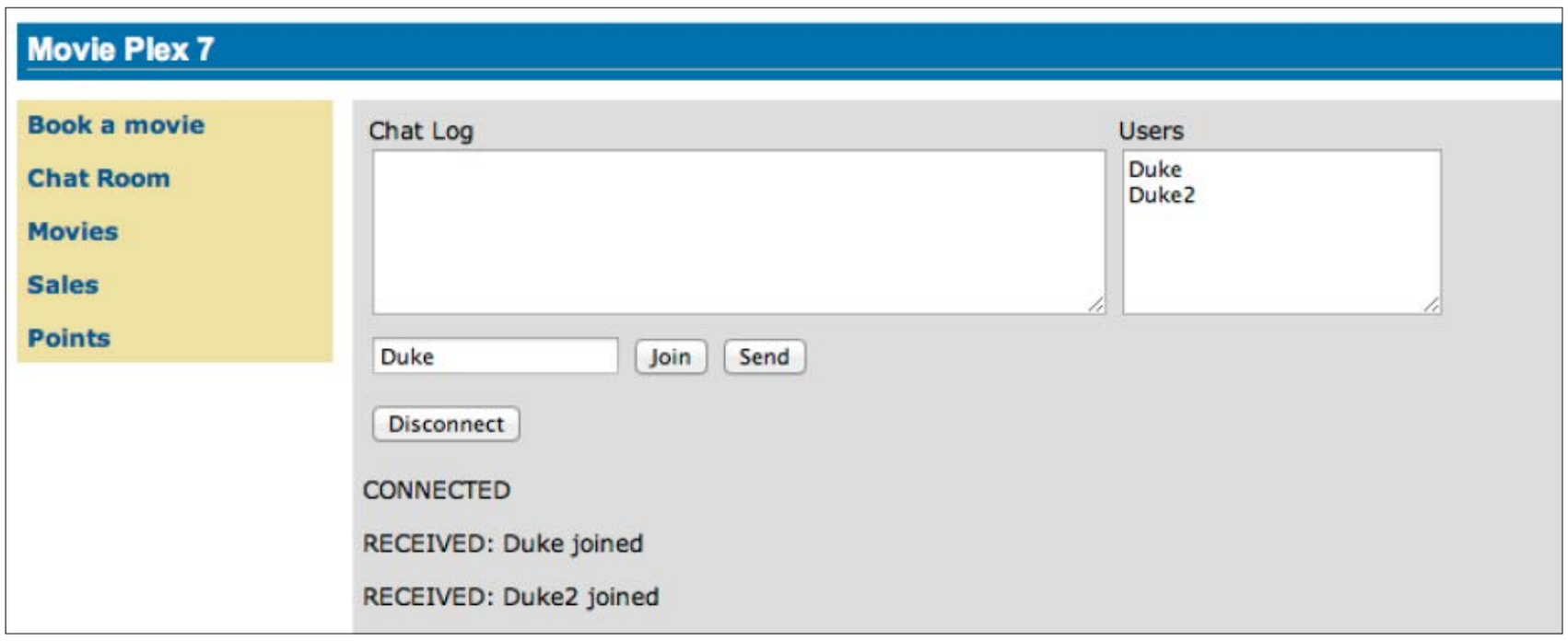


Figure 9

Conclusion

In this article, we modified the movieplex7 application that was started in Part 1 and Part 2, providing the ability to calculate movie sales, assign movie points, and chat with peers. The article covered the following technologies:

- Batch processing for Java EE
- JMS 2.0
- WebSocket 1.0

This three-part article series has taken you on a whirlwind tour of the new features that are available in Java EE 7. To learn more about Java EE 7, take a look at the online tutorial. `</article>`

LEARN MORE

- Part 1 of this series
- Part 2 of this series

RAOUL-GABRIEL URMA

BIO

Part 1

Processing Data with Java SE 8 Streams

Use stream operations to express sophisticated data processing queries.

What would you do without collections? Nearly every Java application *makes* and *processes* collections. They are fundamental to many programming tasks: they let you group and process data. For example, you might want to create a collection of banking transactions to represent a customer's statement. Then, you might want to process the whole collection to find out how much money the customer spent. Despite their importance, processing collections is far from perfect in Java.

First, typical processing patterns on collections are similar to SQL-like operations such as "finding" (for example, find the transaction with highest value) or "grouping" (for example, group all transactions related to grocery shopping). Most databases let you specify such operations declaratively. For example, the following SQL query lets you find the transaction ID with the highest value: "SELECT id, MAX(value) from transactions".

As you can see, we don't need to implement *how* to calculate the maximum value (for example, using loops and a variable to track the highest value). We only express *what* we expect. This basic idea means that you need to worry less about how to explicitly implement such queries—it is handled for you. Why can't we do something similar with collections? How many times do you find yourself reimplementing these operations using loops over and over again?

Second, how can we process really large collections efficiently? Ideally, to speed up the processing, you want to leverage multicore architectures. However, writing parallel code is hard and error-prone.

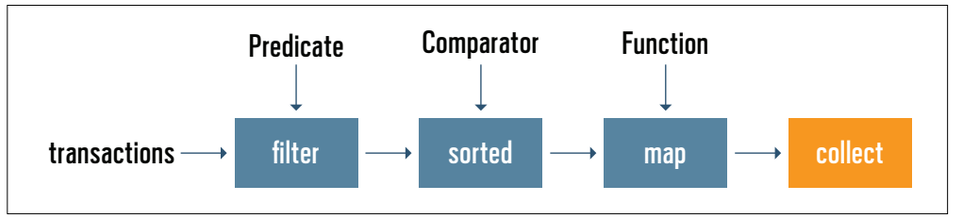


Figure 1

Java SE 8 to the rescue! The Java API designers are updating the API with a new abstraction called *Stream* that lets you process data in a declarative way. Furthermore, streams can leverage multicore architectures without you having to write a single line of multithread code. Sounds good, doesn't it? That's what this series of articles will explore.

Before we explore in detail what you can do with streams, let's take a look at an example so you have a sense of the new programming style with Java SE 8 streams. Let's say we need to find all transactions of type grocery and return a list of transaction IDs sorted in decreasing order of transaction value. In Java SE 7, we'd do that as shown in **Listing 1**. In Java SE 8, we'd do it as shown in **Listing 2**.

Figure 1 illustrates the Java SE 8 code. First, we obtain a stream from the list of transactions (the data) using the stream() method available on List. Next, several operations (filter, sorted, map, collect) are chained together to form

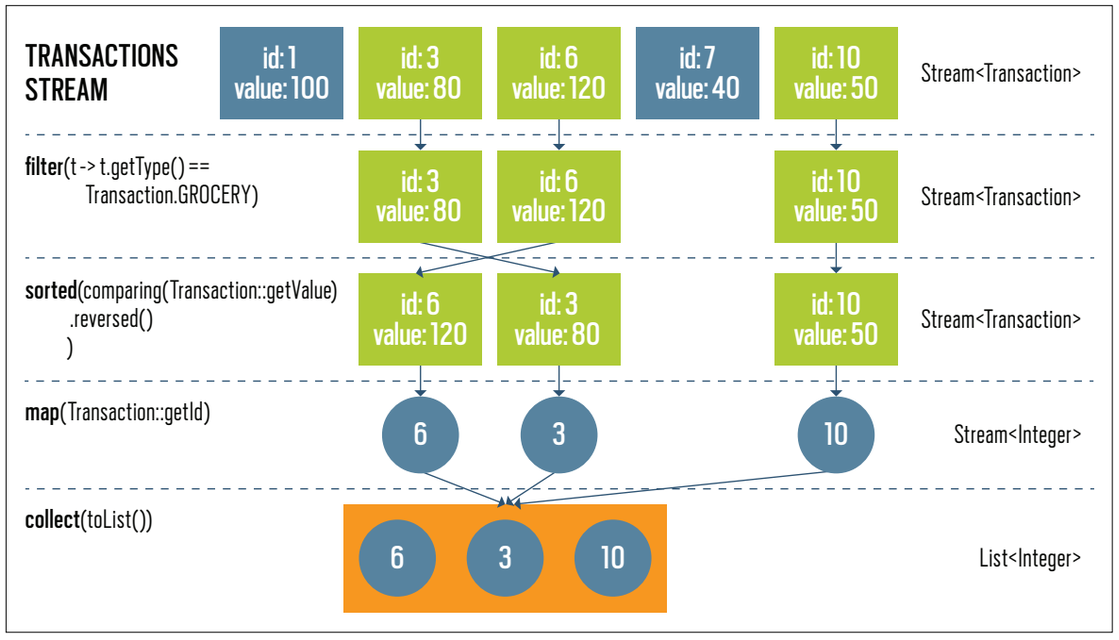


Figure 2

a pipeline, which can be seen as forming a query on the data.

So how about parallelizing the code? In Java SE 8 it's easy: just replace stream() with parallelStream(), as shown in **Listing 3**, and the Streams API will internally decompose your query to leverage the multiple cores on your computer.

Don't worry if this code is slightly overwhelming. We will explore how it works in the next sections. However, notice the use of lambda expressions (for example, t-> t.getCategory() == Transaction.GROCERY) and method references (for example, Transaction::getId), which you should be familiar with by now. (To brush up on lambda expressions, refer to previous *Java Magazine* articles and other resources listed at the end of this article.)

For now, you can see a stream as an abstraction for expressing efficient, SQL-like operations on a collection of data. In addition, these operations can be succinctly parameterized with lambda expressions.

At the end of this series of articles about Java SE 8 streams, you will be able to use the Streams API to write code similar to **Listing 3** to express powerful queries.

Getting Started with Streams

Let's start with a bit of theory. What's the definition of a stream? A short definition is "a sequence of elements from a source that supports aggregate operations." Let's break it down:

- **Sequence of elements:** A stream provides an interface to a sequenced set of values of a specific element type. However, streams don't actually store elements; they are computed on demand.
- **Source:** Streams consume from a data-providing source such as collections, arrays, or I/O resources.
- **Aggregate operations:** Streams support SQL-like operations and common operations from functional programing languages, such as filter, map, reduce, find, match, sorted, and so on.

Furthermore, stream operations have two fundamental characteristics that make them very different from collection operations:

- **Pipelining:** Many stream operations return a stream themselves. This allows operations to be chained to form a larger pipeline. This enables certain optimizations, such as *laziness* and *short-circuiting*, which we explore later.
- **Internal iteration:** In contrast to collections, which are iterated explicitly (*external iteration*), stream operations do the iteration behind the scenes for you. Let's revisit our earlier code

LISTING 1 LISTING 2 / LISTING 3

```
List<Transaction> groceryTransactions = new Arraylist<>();
for(Transaction t: transactions){
 if(t.getType() == Transaction.GROCERY){
  groceryTransactions.add(t);
 }
}
Collections.sort(groceryTransactions, new Comparator(){
 public int compare(Transaction t1, Transaction t2){
  return t2.getValue().compareTo(t1.getValue());
 }
});
List<Integer> transactionIds = new ArrayList<>();
for(Transaction t: groceryTransactions){
 transactionsIds.add(t.getId());
}
```

Download all listings in this issue as text

example to explain these ideas. **Figure 2** illustrates **Listing 2** in more detail.

We first get a stream from the list of transactions by calling the stream() method. The datasource is the list of transactions and will be providing a sequence of elements to the stream. Next, we apply a series of aggregate operations on the stream: filter (to filter elements given a predicate), sorted (to sort the elements given a comparator), and map (to extract information). All these operations except collect return a Stream so they can be chained to form a pipeline, which can be viewed as a query on the source.

No work is actually done until collect is invoked. The collect operation will start processing the pipeline to return a result (something that is not a Stream; here, a List). Don't worry about collect for now; we will explore it in detail in a future article. At the moment, you can see collect as an operation that takes as an argument various recipes for accumulating the elements of a stream into a summary result. Here, toList() describes a recipe for converting a Stream into a List.

Before we explore the different methods available on a stream, it is good to pause and reflect on the conceptual difference between a stream and a collection.

BEHIND THE SCENES

In contrast to collections, which are iterated explicitly (*external iteration*), **stream operations do the iteration behind the scenes for you.**

Streams Versus Collections

Both the existing Java notion of collections and the new notion of streams provide interfaces to a sequence of elements. So what's the difference? In a nutshell, collections are about data and streams are about computations.

Consider a movie stored on a DVD. This is a collection (perhaps of bytes or perhaps of frames—we don't care which here) because it contains the whole data structure. Now consider watching the same video when it is being streamed over the internet. It is now a stream (of bytes or frames). The streaming video player needs to have downloaded only a few frames in advance of where the user is watching, so you can start displaying values from the beginning of the stream before most of the values in the stream have even been computed (consider streaming a live football game).

LISTING 4 LISTING 5

```
List<String> transactionIds = new ArrayList<>();
for(Transaction t: transactions){
  transactionIds.add(t.getId());
}
```

Download all listings in this issue as text

In the coarsest terms, the difference between collections and streams has to do with *when* things are computed. A collection is an in-memory data structure, which holds all the values that the data structure currently has—every element in the collection has to be computed before it can be added to the collection. In contrast, a stream is a conceptually fixed data structure in which elements are computed on demand.

Using the Collection interface requires iteration to be done by the user (for example, using the enhanced for loop called foreach); this is called external iteration. In contrast, the Streams library uses internal iteration—it does the iteration for you and takes care of storing the resulting stream value somewhere; you merely provide a function saying what's to be done. The code in **Listing 4** (external iteration with a collection) and **Listing 5** (internal iteration with a stream) illustrates this difference.

In **Listing 4**, we explicitly iterate the list of transactions sequentially to extract each transaction ID and add it to an accumulator. In contrast, when using a stream, there's no explicit iteration. The code in **Listing 5** builds a query, where the map operation is parameterized to extract the transaction IDs and the collect operation converts the resulting Stream into a List.

You should now have a good idea of what a stream is and what you can do with it. Let's now look at the different operations supported by streams so you can express your own data processing queries.

Stream Operations: Exploiting Streams to Process Data

The Stream interface in java.util.stream.Stream defines many operations, which can be grouped in two categories. In the example illustrated in **Figure 1**, you can see the following operations:

- filter, sorted, and map, which can be connected together to form a pipeline

- collect, which closed the pipeline and returned a result

Stream operations that can be connected are called *intermediate operations*. They can be connected together because their return type is a Stream. Operations that close a stream pipeline are called *terminal operations*. They produce a result from a pipeline such as a List, an Integer, or even void (any non-Stream type).

You might be wondering why the distinction is important. Well, intermediate operations do not perform any processing until a terminal operation is invoked on the stream pipeline; they are "lazy." This is because intermediate operations can usually be "merged" and processed into a single pass by the terminal operation.

For example, consider the code in **Listing 6**, which computes two even square numbers from a given list of numbers. You might be surprised that it prints the following:

```
filtering 1
filtering 2
mapping 2
filtering 3
filtering 4
mapping 4
```

This is because limit(2) uses *short-circuiting*; we need to process only part of the stream, not all of it, to return a result. This is similar to evaluating a large Boolean expression chained with the and operator: as soon as one expression returns false, we can deduce that the whole expression is false without evaluating all of it. Here, the operation limit returns a stream of size 2. In addition, the operations filter and map have been merged in the same pass.

To summarize what we've learned so far, working with streams, in general, involves three things:

- A datasource (such as a collection) on which to perform a query
- A chain of intermediate operations, which form a stream pipeline
- One terminal operation, which executes the stream pipeline and produces a result

Let's now take a tour of some of the operations available on streams. Refer to the java.util.stream.Stream interface for the complete list, as well as to the resources at the end of this article for more examples.

Filtering. There are several operations that can be used to filter elements from a stream:

- filter(Predicate): Takes a predicate (java.util.function.Predicate) as an argument and returns a stream including all elements that match the given predicate
- distinct: Returns a stream with unique elements (according to the implementation of equals for a stream element)
- limit(n): Returns a stream that is no longer than the given size n
- skip(n): Returns a stream with the first n number of elements discarded

Finding and matching. A common data processing pattern is determining whether some elements match a given property. You can use the anyMatch, allMatch, and noneMatch operations to help you do this. They all take a predicate as an argument and return a boolean as the result (they are, therefore, terminal operations). For example, you can use allMatch to check that all elements in a stream of transactions have a value higher than 100, as shown in **Listing 7**.

In addition, the Stream interface provides the operations findFirst and findAny for retrieving arbitrary elements from a stream. They can be used in conjuction with other streams operations such as filter. Both findFirst and findAny return an Optional object (see **Listing 8**).

The Optional<T> class (java.util.Optional) is a container class to represent the existence or absence of a value. In **Listing 8**, it is possible that findAny doesn't find any transaction of type grocery. The Optional class contains several methods to test the existence of an element. For example, if a transaction is present, we can choose to apply an operation on the optional object by using the ifPresent method, as

LISTING 6 LISTING 7 / LISTING 8 / LISTING 9

```
List<Integer> numbers = Arrays.asList(1, 2, 3, 4, 5, 6, 7, 8);
List<Integer> twoEvenSquares =
  numbers.stream()
      .filter(n -> {
          System.out.println("filtering " + n);
          return n % 2 == 0;
        })
      .map(n -> {
          System.out.println("mapping " + n);
          return n * n;
        })
      .limit(2)
      .collect(toList());
```

Download all listings in this issue as text

shown in **Listing 9** (where we just print the transaction).

Mapping. Streams support the method map, which takes a function (java.util.function.Function) as an argument to project the elements of a stream into another form. The function is applied to each element, "mapping" it into a new element.

For example, you might want to use it to extract information from each element of a stream. In the example in **Listing 10**, we return a list of the length of each word from a list.

Reducing. So far, the terminal operations we've seen return a boolean (allMatch and so on), void (forEach), or an Optional object (findAny and so on). We have also been using collect to combine all elements in a Stream into a List.

However, you can also combine all elements in a stream to formulate more-complicated process queries, such as "what is the transaction with the highest ID?" or "calculate the sum of all transactions' values." This is possible using the reduce operation on streams, which repeatedly applies an operation (for example, adding two numbers) on each element until a result is produced. It's often called a *fold operation* in functional programming because you can view this operation as "folding" repeatedly a long piece of paper (your stream) until it forms one little square, which is the result of the fold operation.

LEVERAGE CORES

The Streams API will internally decompose your query to leverage the multiple cores on your computer.

It helps to first look at how we could calculate the sum of a list using a for loop:

```
int sum = 0;
for (int x : numbers) {
  sum += x;
}
```

Each element of the list of numbers is combined iteratively using the addition operator to produce a result. We essentially "reduced" the list of numbers into one number. There are two parameters in this code: the initial value of the sum variable, in this case 0, and the operation for combining all the elements of the list, in this case +.

Using the reduce method on streams, we can sum all the elements of a stream as shown in **Listing 11**. The reduce method takes two arguments:

- An initial value, here 0
- A BinaryOperator<T> to combine two elements and produce a new value

LISTING 10 LISTING 11 LISTING 12 LISTING 13 LISTING 14 LISTING 15

```
List<String> words = Arrays.asList("Oracle", "Java", "Magazine");
 List<Integer> wordLengths =
   words.stream()
      .map(String::length)
      .collect(toList());
```

Download all listings in this issue as text

The reduce method essentially abstracts the pattern of repeated application. Other queries such as "calculate the product" or "calculate the maximum" (see **Listing 12**) become special use cases of the reduce method.

Numeric Streams

You have just seen that you can use the reduce method to calculate the sum of a stream of integers. However, there's a cost: we perform many boxing operations to repeatedly add Integer objects together. Wouldn't it be nicer if we could call a sum method, as shown in **Listing 13**, to be more explicit about the intent of our code?

Java SE 8 introduces three primitive specialized stream interfaces to tackle this issue—IntStream, DoubleStream, and LongStream—that respectively specialize the elements of a stream to be int, double, and long.

The most-common methods you will use to convert a stream to a specialized version are mapToInt, mapToDouble, and mapToLong. These methods work exactly like the method map that we saw earlier, but they return a specialized stream instead of a Stream<T>. For example, we could improve the code in **Listing 13** as shown in **Listing 14**. You can also convert from a primitive stream to a stream of objects using the boxed operation.

Finally, another useful form of numeric streams is numeric ranges. For example, you might want to generate all numbers between 1 and 100. Java SE 8 introduces two static methods available on IntStream, DoubleStream, and LongStream to help generate such ranges: range and rangeClosed.

Both methods take the starting value of the range as the first parameter and the end value of the range as the second parameter. However, range is exclusive, whereas rangeClosed is inclusive. **Listing 15** is an example that uses

rangeClosed to return a stream of all odd numbers between 10 and 30.

Building Streams

There are several ways to build streams. You've seen how you can get a stream from a collection. Moreover, we played with streams of numbers. You can also create streams from values, an array, or a file. In addition, you can even generate a stream from a function to produce infinite streams!

Creating a stream from values or from an array is straightforward: just use the static methods Stream.of for values and Arrays.stream for an array, as shown in **Listing 16**.

You can also convert a file in a stream of lines using the Files.lines static method. For example, in **Listing 17** we count the number of lines in a file.

Infinite streams. Finally, here's a mind-blowing idea before we conclude this first article about streams. By now you should understand that elements of a stream are produced on demand. There are two static methods—Stream.iterate and Stream.generate—that let you create a stream from a function. However, because elements are calculated on demand, these two operations can produce elements "forever." This is what we call an *infinite stream*: a stream that doesn't have a fixed size, as a stream does when we create it from a fixed collection.

BLOW YOUR MIND
Here's a mind-blowing idea: these two operations can produce elements "forever."

LISTING 16 / LISTING 17 / LISTING 18 / LISTING 19

```
Stream<Integer> numbersFromValues = Stream.of(1, 2, 3, 4);
int[] numbers = {1, 2, 3, 4};
IntStream numbersFromArray = Arrays.stream(numbers);
```

Download all listings in this issue as text

Listing 18 is an example that uses iterate to create a stream of all numbers that are multiples of 10. The iterate method takes an initial value (here, 0) and a lambda (of type UnaryOperator<T>) to apply successively on each new value produced.

We can turn an infinite stream into a fixed-size stream using the limit operation. For example, we can limit the size of the stream to 5, as shown in **Listing 19**.

Conclusion

Java SE 8 introduces the Streams API, which lets you express sophisticated data processing queries. In this article, you've seen that a stream supports many operations such as filter, map, reduce, and iterate that can be combined to write concise and expressive data processing queries. This new way of writing code is very different from how you would process collections before Java SE 8. However, it has many benefits. First, it makes use of techniques such as laziness and short-circuiting to optimize your data processing queries. Second, streams can be parallelized automatically to leverage multicore architectures. In Part 2, we will explore more-advanced operations, such as flatMap and collect. Stay tuned. `</article>`

MORE ON TOPIC:

Java 8 Is Here

LEARN MORE

- *Java 8 Lambdas in Action*
- GitHub repository with Java SE 8 code examples
- "Java 8: Lambdas, Part 1" by Ted Neward

JOSH JUNEAU

BIO

JSR 308 Explained: Java Type Annotations

The benefits of type annotations and example use cases

Annotations were introduced into Java in Java SE 5, providing a form of syntactic metadata that can be added to program constructs. Annotations can be processed at compile time, and they have no direct effect on the operation of the code. However, they have many use cases. For instance, they can produce informational messages for the developer at compile time, detecting errors or suppressing warnings. In addition, annotations can be processed to generate Java source files or resources that can be used to modify annotated code. The latter is useful for helping to cut down on the amount of configuration resources that must be maintained by an application.

JSR 308, Annotations on Java Types, has been incorporated as part of Java SE 8. This JSR builds upon the existing annotation framework, allowing type annotations to become part of the language. Beginning in Java SE 8, annotations can be applied to types in addition to all of their existing uses within Java declarations. This means annotations can now be applied anywhere a type is specified, including during class instance creation, type casting, the implementation of interfaces, and the specification of throws clauses. This allows developers to apply the benefits of annotations in even more places.

Annotations on types can be useful in many cases, most notably to enforce stronger typing, which can help reduce the number of errors within code. Compiler checkers can be written to verify annotated code, enforcing rules by generating compiler warnings when code does not meet certain requirements. Java SE 8 does not provide a default type-checking framework, but it is possible to write custom annotations and processors for type checking. There are also a number of type-checking frameworks that can be downloaded, which can be used as plug-ins to the Java compiler to check and enforce types that have been annotated. Type-checking frameworks comprise type annotation definitions and one or more pluggable modules that are used with the compiler for annotation processing.

NEED INFO?

Annotations have no direct effect on code operation, but at processing time, they can cause an annotation processor to generate files or provide informational messages.

This article begins with a brief overview of annotations, and then you'll learn how to apply annotations to Java types, write type annotations, and use compile-time plug-ins for type checking. After reading this article, you'll be able to use type annotations to enforce stronger typing in your Java source code.

Overview of Built-in Annotations

Annotations can be easily recognized in code because the annotation name is prefaced with the @ character.

Java 8 Is Here

Annotations have no direct effect on code operation, but at processing time, they can cause an annotation processor to generate files or provide informational messages.

In its simplest form, an annotation can be placed in Java source code to indicate that the compiler must perform specific "checking" on the annotated component to ensure that the code conforms to specified rules.

Java comes with a basic set of built-in annotations. The following Java annotations are available for use out of the box:

- @Deprecated: Indicates that the marked element should no longer be used. Most often, another element has been created that encapsulates the marked element's functionality, and the marked element will no longer be supported. This annotation will cause the compiler to generate a warning when the marked element is found in source code.
- @Override: Indicates that the marked method overrides another method that is defined in a superclass. The compiler will generate a warning if the marked method does not override a method in a superclass.
- @SuppressWarnings: Indicates that if the marked element generates warnings, the compiler should suppress those warnings.
- @SafeVarargs: Indicates that the marked element does not perform potentially unsafe operations via its varargs parameter. Causes the compiler to suppress unchecked warnings related to varargs.
- @FunctionalInterface: Indicates that the type declaration is intended to be a functional interface.

Listing 1 shows a couple of examples using these built-in annotations.

Use Cases for Annotations on Types

Annotations can exist on any Java type declaration or expression to help enforce stronger typing. The following use cases explain where type annotations can be of great value.

Generation of new objects. Type annotations can provide static verification when creating new objects to help enforce the compatibility of annotations on the object constructor. For example:

```
Forecast currentForecast =
new @Interned Forecast();
```

Generics and arrays. Generics and arrays are great candidates for type annotations, because they can help restrict the data that is to be expected for these objects.

LISTING 1 LISTING 2 / LISTING 3

```
@Override
public void start(Stage primaryStage) {
  ...
}

@Deprecated
public void buttonAction(ActionEvent event){
  ...
}
```

Download all listings in this issue as text

Not only can a type annotation use compiler checking to ensure that the correct datatypes are being stored in these elements, but the annotations can also be useful as a visual reminder to the developer for signifying the intent of a variable or array, for example:

```
@NonEmpty Forecast []
```

Type casting. Type casts can be annotated to ensure that annotated types are retained in the casting. They can also be used as qualifiers to warn against unintended casting uses, for instance:

```
@Readonly Object x; ...
 (@Readonly Date) x ...
```

or

```
Object myObject =
 (@NotNull Object) obj
```

Inheritance. Enforcing the proper type or object that a class extends or implements can significantly reduce the number of errors in application code. **Listing 2** contains an example of a type annotation on an implementation clause.

Exceptions. Exceptions can be annotated to ensure that they adhere to certain criteria, for example:

```
catch (@Critical Exception e) {
 ...
}
```

Receivers. It is possible to annotate a receiver parameter of a method by explicitly listing it within the parameter list. **Listing 3** shows a demonstration of type annotations on a method receiver parameter.

Applying Type Annotations

Type annotations can be applied on types in a variety of ways. Most

often, they are placed directly before the type to which they apply. However, in the case of arrays, they should be placed before the relevant part of the type. For instance, in the following declaration, the array should be read-only:

```
Forecast @Readonly [] fiveDay =
 new Forecast @Readonly [5];
```

When annotating arrays and array types, it is important to place the annotation in the correct position so that it applies to the intended element within the array. Here are a few examples:

- Annotating the int type:

```
@ReadOnly int [] nums;
```

- Annotating the array type int[]:

```
int @ReadOnly [] nums;
```

- Annotating the array type int[][]:

```
int @ReadOnly [][] nums;
```

- Annotating the type int[], which is a component type of int[][]:

```
int [] @ReadOnly [] nums;
```

Using Available Type Annotations

To enforce stronger type checking, you must have a proper set of annotations that can be used to enforce certain criteria on your types. As such, there are a number of type-checking annotations that are available for use today, including those that are available with the Checker Framework.

Java SE 8 does not include any annotations specific to types, but libraries such as the Checker Framework contain annotations that can be applied to types for verifying certain criteria. For example, the Checker Framework contains the @NonNull annotation, which can be applied to a type so that upon compilation it is verified to not be null. The Checker Framework also contains the @Interned annotation, which indicates that a variable refers to the canonical representation of an object. The following are a few other examples of annotations available with the Checker Framework:

- @GuardedBy: Indicates a type whose value may be accessed only when the given lock is held
- @Untainted: Indicates a type that includes only untainted, trusted values
- @Tainted: Indicates a type that may include only tainted, untrusted values; a supertype of @Untainted
- @Regex: Indicates a valid regular expression on Strings

To make use of the annotations that are part of the Checker Framework, you must download the framework, and then add the annotation source files to your CLASSPATH, or—if you are using Maven—add the Checker Framework as a dependency.

If you are using an IDE, such as NetBeans, you can easily add the Checker Framework as a dependency using the Add Dependency dialog box. For example, **Figure 1** shows how to add a dependency to a Maven project.

Once the dependencies have been added to the application, you can begin to use the annotations on your types by importing them into your classes, as shown in **Listing 4**.

If you have a requirement that is not met by any of the existing

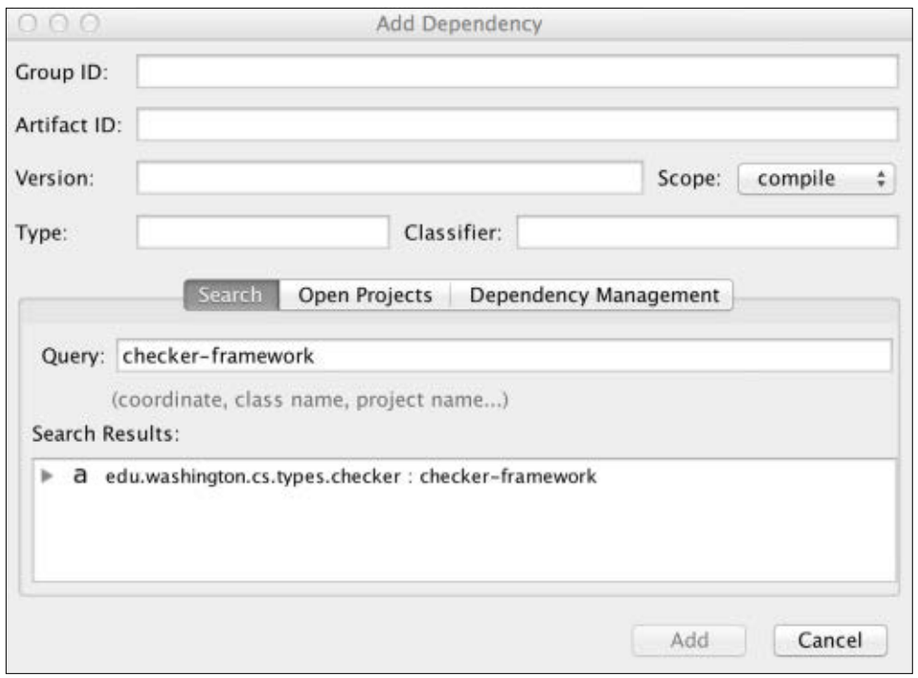


Figure 1

LISTING 4

```
import checkers.interning.quals.Interned;
import checkers.nullness.quals.NonNull;
...
@ZipCode
@NonNull
String zipCode;
```

Download all listings in this issue as text

implementations, you also have the option of creating custom annotations to suit your needs.

Defining Custom Annotations

Annotations are a form of interface, and an annotation type definition looks very similar to an interface definition. The difference between the two is that the annotation type definition includes the interface keyword prefixed with the @ character. Annotation definitions also include annotations themselves, which specify information about the type definition. The following list of annotations can be used when defining an annotation:

- @Retention: Specifies how the annotation should be stored. Options are CLASS (default value; not accessible at runtime), RUNTIME (available at runtime), and SOURCE (after the class is compiled, the annotation is disregarded).
- @Documented: Marks the annotation for inclusion in Javadoc.
- @Target: Specifies the contexts to which the annotation can be applied. Contains a single element, value, of type java.lang.annotation.ElementType[].
- @Inherited: Marks the annotation to be inherited by subclasses of the annotated class.

The definitions for standard declaration annotations and type annotation look very similar. The key differentiator is in the @Target specification, which denotes the kind of elements to which a particular annotation can be applied. Declaration annotations target fields, whereas type annotations target types.

A declaration annotation may contain the following meta-annotation:

```
@Target(ElementType.FIELD)
```

A type annotation must contain the following meta-annotation:

```
@Target(ElementType.TYPE_USE)
```

If the type annotation is to target a type parameter, the annotation must contain the following meta-annotation:

```
@Target
(ElementType.TYPE_PARAMETER)
```

A type annotation may apply to more than one context. In such cases, more than one ElementType can be specified within a list. If the same ElementType is specified more than once within the list, then a compile-time error will be displayed.

Listing 5 shows the complete listing for an @NonNull type annotation definition. In the listing, the definition of @NonNull includes two targets, which means that the annotation may be applied to either a type or a type parameter.

LISTING 5 LISTING 6

```
@Documented
@Retention(RetentionPolicy.RUNTIME)
@Target({ElementType.TYPE_USE, ElementType.TYPE_PARAMETER})
@TypeQualifier
public @interface NonNull {
}
```

Download all listings in this issue as text

Similar to standard declaration annotations, type annotations can also contain parameters, and they may contain default values. To specify parameters for type annotations, add their declarations within the annotation interface. **Listing 6** demonstrates a type annotation with one parameter: a string field identified by zip.

It is possible that an annotation can be applied on both a field and a type simultaneously. For instance, if an annotation of @Foo was applied to a variable declaration, it could also be applied to the type declaration at the same time if the @Target contains both elements. Such a scenario might look like the following declaration, which applies to both the type String and the variable myString:

```
@Foo String myString;
```

Processing Type Annotations

After you apply type annotations to the code, you must use a compiler plug-in to process the annotations accordingly. As mentioned previously, annotations have no operational effect on application code. The processor performs the magic as it parses the code and performs certain tasks when annotations are encountered.

If you write custom annotations, you must also write custom compiler plug-ins to process them. JSR 269, Pluggable Annotation Processing API, provides support for developing custom annotation processors. Developing an annotation processor is out of scope for this article, but the Pluggable Annotation Processing API makes it easy to do. There are also annotation processors available for download, such as the Checker Framework processors.

Using a type-qualifier compiler plug-in. The Checker Framework

is a library that can be used within your applications, even if you are using older releases of Java. The framework contains a number of type annotations that are ready to be utilized, along with annotation processors that can be specified when compiling your code. Annotation support in Java SE 8 enables the use of third-party libraries such as the Checker Framework, making it easy to incorporate prebuilt type annotations into new and existing code.

Previously, we saw how to incorporate the Checker Framework into a project in order to make use of the type annotations that come with it. However, if you do not also use a custom annotation processor, these annotations will not be processed, and they'll be useful only for documentation purposes.

The Checker Framework contains a number of custom processors for each of the type annotations that are available with the framework. Once the framework is installed onto a machine, a custom javac compiler that is packaged with the framework can be used to compile annotated applications.

To make use of the Checker Framework, simply download the .zip file and extract it to your machine. Optionally, update your execution path or create an alias to make it easy to execute the Checker Framework binaries. Once installed, the framework can be verified by executing a command similar to the one shown in **Listing 7**.

It helps to know which annotations are available for use, so users of the Checker Framework should first review the framework documentation to read about them. The Checker Framework refers to type annotations as *qualifiers*. Once you are familiar with the qualifiers that are available for use, determine which of them might be useful to incorporate on types within existing applications. If you are authoring a new application, apply the qualifiers to types accordingly to take advantage of the benefits.

To check your code, the compiler must be directed as to which processors to use for type checking. This can be done by executing the custom javac normally and specifying the –processor flag along with the fully qualified processor name.

For instance, if the @NonNull annotation is used, then the nullness processor must be specified when compiling the code. If you have installed the Checker Framework, use the custom javac that is distributed with the framework, and specify the checkers.nullness.NullnessChecker processor to process the annotations.

Listing 8 contains a sample class that makes use of the @NonNull annotation. To compile this class, use the command in **Listing 9**. If the class shown in **Listing 8** is compiled, then no warnings will be noted.

However, assigning null to the annotated variable declaration will cause the nullness checker to provide a warning. **Listing 10** shows the modified class, and **Listing 11** shows the warning that the compiler will produce.

Rather than using the custom javac binary, you can use the standard JDK installation and run checkers.jar, which will utilize the Checker compiler rather than the standard compiler. **Listing 12** demonstrates an invocation of checkers.jar, rather than the custom javac.

The Checker Framework contains instructions for adding the custom javac command to your CLASSPATH and creating an alias, and it describes more ways to make it easy to integrate the framework into your compile process. For complete details, please see the documentation.

Compilation using multiple processors at once. What if you wish to specify more than one processor at compilation time? Via autodiscovery, it is possible to use mul-

CONFORMIST CODE

In its simplest form, an annotation can be placed in Java source code to indicate that the compiler must perform specific "checking" on the annotated component to ensure that the code conforms to specified rules.

LISTING 7 / LISTING 8 / LISTING 9 / LISTING 10 / LISTING 11 / LISTING 12

```
java -jar binary/checkers.jar -version
javac 1.8.0-jsr308-1.7.1
```

Download all listings in this issue as text

tiple processors when compiling with the Checker Framework. To enable auto-discovery, a configuration file named META-INF/services/javax.annotation.processing.Processor must be placed within the CLASSPATH. This file must contain the name of each Checker plug-in that will be used, one per line. When using auto-discovery, the javac compiler will always run the listed Checker plug-ins, even if the –processor flag is not specified.

To disable auto-discovery, pass the –proc:none command-line option to javac. This option disables all annotation processing.

Using a Maven plug-in. A plug-in is available that allows the Checker Framework to become part of any Maven project. To use the plug-in, modify the project object model (POM) to specify the Checker Framework repositories, add the dependencies to your project, and attach the plug-in to the project build lifecycle. **Listing 13**, **Listing 14**, and **Listing 15** show an example for accomplishing each of these tasks.

Using the Maven plug-in makes it easy to bind the Checker Framework to a project for use within an IDE as well. For instance, if the plug-in is configured on a NetBeans Maven project, the Checker Framework will process annotations each time the project is built within NetBeans.

Distributing Code Containing Type Annotations

To make use of a particular type annotation, its declaration must be within the CLASSPATH. The same holds true when distributing applications that contain type annotations. To compile or run source code containing type annotations, minimally the annotation declaration classes must exist within the CLASSPATH.

If you've written custom annotations, they might already be part of the application source code. If not, a JAR file containing those annotation declarations should be packaged with the code distribution. The Checker Framework includes a JAR file, checkers-quals.jar, which includes the declarations of the distributed qualifiers (annotations). If you are using the Checker Framework annotations within an application, you should package this JAR file with the distribution.

Conclusion

Java SE 8 adds support for type annotations. Type annotations can provide a stronger type-checking system, reducing the number of errors and bugs within code. Applications using type annotations are also backward compatible, because annotations do not affect runtime operation.

Developers can opt to create custom type annotations, or use annotations from third-party solutions. One of the most well-known type-checking frameworks is the Checker Framework, which can be used with Java SE 8 or previous releases. To begin making your applications less error-prone, take a look at the Checker Framework documentation. `</article>`

LISTING 13 LISTING 14 / LISTING 15

```
<!-- Add repositories to POM -->
<repositories>
 <repository>
  <id>checker-framework-repo</id>
  <url>http://types.cs.washington.edu/m2-repo</url>
 </repository>
</repositories>
<pluginRepositories>
 <pluginRepository>
  <id>checker-framework-repo</id>
  <url>http://types.cs.washington.edu/m2-repo</url>
 </pluginRepository>
</pluginRepositories>
```

Download all listings in this issue as text

MORE ON TOPIC:

Java 8 Is Here

LEARN MORE

- Annotations lesson from The Java Tutorials
- Checker Framework

JOHAN VOS

BIO

JavaFX and Near Field Communication on the Raspberry Pi

Use your Java skills to create end-to-end applications that span card readers on embedded devices to back-end systems.

JavaFX has already proven to be an excellent platform running on the Raspberry Pi. If you are working on client-side Java development on embedded devices, you probably know that JavaFX and the Raspberry Pi make for a great combination. In a number of cases, embedded devices are used in kiosk environments. A card reader is connected to an embedded device, and users have to scan a card to initiate an action—for example, to be allowed to enter a room, pay for something, or gain some credits.

There are different types of card readers, different types of cards, and different protocols. In this article, we look at how Java SE 8, which is available for the Raspberry Pi, can be used to communicate with a near field communication (NFC) card reader, show a status message on a screen, or send information to a back-end system.

Note: The source code for the examples shown in this article is available at https://bitbucket.org/johanvos/javafx-pi-nfc.

YOU'RE READY

You can use your existing Java skills to create end-to-end applications that span different environments.

Components of Our System

The NFC specification defines a number of standards for the near field communication between devices, including radio communication between devices that are close to each other. Various cards are equipped with a chipset. In this article, we will use a MIFARE DESFire card.

A card reader needs to be chosen that works out of the box on the Raspberry Pi. The USB-based ACR122U card reader from Advanced Card Systems Ltd. is a widely used device that can easily be connected to the Raspberry Pi and allows it to read MIFARE cards.

Note: It is recommended that a powered USB hub be used for connecting the card reader to the Raspberry Pi, because card readers consume a lot of power.

In order to have visual feedback, we need to attach a screen to the Raspberry Pi. A variety of screens are available, including a regular HDMI monitor and smaller screens suited for cars, kiosks, and copiers.

Although this is the setup we will use in this article, different setups are possible.

Set Up the Raspberry Pi

In order to run the code provided for this project, you need a working setup. We will use an image for the Raspberry Pi that is available for download here.

In order to have the ACR122U card reader work with this image, you have to install two additional packages. Run the following commands to do this:

```
sudo apt-get update
sudo apt-get install libpcsclite1
pcscd
```


PHOTOGRAPH BY TON HENDRIKS

Install Java

Download Java SE 8 and install on the Raspberry Pi.

Java SE comes with a package, javax.smartcardio, that allows developers to communicate with smart card readers. By default, this package will look for a personal computer/smart card (PC/SC) implementation on the system.

Now that you have installed the required Linux packages on your Raspberry Pi, the only remaining task is that we have to point the Java Virtual Machine to the location of the pcsclite library.

This is done by setting the system property sun.security.smartcardio.library to the location of the pcsclite library as follows:

```
java -Dsun.security.smartcardio
.library=/path/to/libpcsclite.so
```

The JavaFX Application

Making a connection with a card reader and reading cards are operations that can take some time. Hence, it is not a good idea to execute those tasks at the JavaFX application thread level. Instead, we will create a dedicated thread that will deal with the card reader and uses the standard JavaFX way to update the user interface.

Creating the user interface. We will create a very simple user interface that shows the identifier of the latest successfully scanned card. This identifier is stored in StringProperty latestId, as follows:

```
private StringProperty latestId =
new SimpleStringProperty(" --- ");
```

Our simple user interface contains only an HBox with a label containing some static information followed by a label containing the identifier of the latest scanned card. It is constructed as shown in **Listing 1**.

Communicating with the card reader. The thread responsible for the communication with the card reader is created in a function called doCardReaderCommunication, which is called in the start() method of the JavaFX application. A JavaFX Task is created, assigned to a new thread, and started, as follows:

```
Task task = new Task() {
  ...
};
Thread thread = new Thread(task);
thread.start();
```

This Task performs the following work:

1. It detects a card reader.
2. It detects a card being recognized by the card reader.
3. It reads the identifier from the card.
4. It updates the value of latestId.
5. It waits for the card to be removed.
6. It goes back to Step 2.

Detecting a card reader. Before we can access a card reader we need a TerminalFactory. The javax.microcard package provides a static method for getting a default TerminalFactory, and it allows us to create more-specific TerminalFactory instances for different types of card readers. The default TerminalFactory will use the PC/SC stack that we installed on the Raspberry Pi, and it is obtained by calling the code shown in **Listing 2**. Using the code shown in **Listing 3**, we can query terminalFactory to detect the card readers that are installed.

In this case, our setup was successful, and cardTerminal List contains one element. This CardTerminal is obtained by running the code shown in **Listing 4**.

Detecting cards. Once we have a reference to the card reader, we can start reading cards. We will do this using the infinite loop shown in **Listing 5**. In real-world cases, a dummy card could be used to stop reading.

The thread will block until the reader detects a card. Once a card is detected, we will handle it as discussed in the next section. Then, the thread will block again until the card is removed.

Handling cards. The handleCard method is called when we know that a card is presented to the card reader. Note that a card can be removed during the process of reading, and proper exception handling is required.

A connection to the card is obtained by calling the code shown in **Listing 6**.

Before we can read the identifier on the card, we need to know what type of card we are reading.

LISTING 1 / LISTING 2 / LISTING 3 / LISTING 4 / LISTING 5 / LISTING 6

```
HBox hbox = new HBox(3);
Label info = new Label("Last checkin by: ");
Label latestCheckinLabel = new Label();
latestCheckinLabel.textProperty().bind(latestId);
hbox.getChildren().addAll(info, latestCheckinLabel);
Scene scene = new Scene(hbox, 400, 250);
primaryStage.setScene(scene);
primaryStage.show();
```

Download all listings in this issue as text

Different card types use different addresses to store the identifier. In our simple demo, we will limit ourselves to MIFARE DESFire cards, but the example can easily be extended to other card types.

Information on the type of card we are reading can be obtained by inspecting the ATR (answer-to-reset) bytes of the card as follows:

```
ATR atr = card.getATR();
```

We compare the bytes in this atr object with the bytes we expect on a DESFire card, as follows:

```
Arrays.equals(
desfire, atr.getBytes());
```

As shown in **Listing 7**, desfire is a static byte array containing the information we expect.

If we have a supported card type (that is, a DESFire card), we can query the identifier. In order to do so, we need to transmit an application protocol data unit (APDU) command over the basic logic channel of the card, and then read the response. Using the javax.microcard package, this is done as shown in **Listing 8**.

KIOSK 101

In a kiosk, a card reader is connected to an embedded device, and **users scan a card** to initiate an action.

The getAddress parameter we are passing is a static byte array that defines how to read the identifier for a DESFire card, and it is declared as shown in **Listing 9**. Different card types might require a different byte array.

The function readable(byte[]) translates the byte array into a more human-readable format by converting each byte to its hexadecimal String representation (see **Listing 10**).

Now that we have a readable identifier for the card, we need to show the identifier in the user interface. Because the identifier is obtained on a thread different from the JavaFX application thread (that is, from the communication thread), we need to make sure that we don't change the latestId StringProperty directly from the communication thread. Rather, we need to put the change on the JavaFX application thread as shown here.

```
Platform.runLater(new Runnable() {
  public void run() {
    latestId.setValue(uid);
  }
});
```

This separation of concerns also guarantees that the communication with the card reader is not disturbed by activity in the user interface and vice versa.

LISTING 7 LISTING 8 / LISTING 9 / LISTING 10 / LISTING 11 / LISTING 12

```
static byte[] desfire =
new byte[]{0x3b, (byte) 0x81, (byte) 0x80,
    0x01, (byte) 0x80, (byte) 0x80};
```

Download all listings in this issue as text

Sending the data to a back end. In many cases, visual feedback is required when a reader scans a card. In other cases, the information needs to be sent to a back-end system. In the next section, we will send data to the back end using a very simple client-server module by which identifiers are sent from the Raspberry Pi to a Java EE 7 back end and then visualized on a web page. Sending data from a JavaFX application to a REST-based back-end system is easily done using DataFX.

The code shown in **Listing 11** will send the identifier uid to a REST endpoint at http://192.168.1.6:8080/webmonitor/rest/card/checkin.

Note that we will use the POST method, because a form parameter is specified in the REST request. This method should be called whenever a new identifier is read. Because DataFX deals with the threading, we have to call this method from the JavaFX application thread. Hence, we can do this in the same code block in which we set the value of the latestId property (see **Listing 12**).

In real-world applications, the server might send additional information, such as the real name of the user of the card, back to the client. The client application can use this information to make the user interface more personal.

The Back End

We will now create a very simple back end that accepts the REST requests and shows them on a

web page. Because it is not assumed that the user of the web page knows when cards are being scanned, it makes sense to dynamically update the web page whenever a card has been scanned. This can easily be done leveraging the WebSocket API in Java EE 7. It is surprising how little code is required for doing this.

We will start with the web page, which is a simple HTML page with a division named checkins that will contain the list of identifiers and a time stamp indicating when each card scan happened.

When the HTML page is loaded, a WebSocket is created, pointing to our simple back end. The content of the checkins division is populated whenever a message arrives over the WebSocket. This is easily achieved using the code shown in **Listing 13**.

Note that the HTML page will open a WebSocket toward localhost:8080/webmonitor/endpoint, while the JavaFX application will push identifiers to localhost:8080/webmonitor/rest/card/checkin. The glue for this is our single-file back-end system, implemented in a class named CardHandler.

Because CardHandler provides a REST endpoint, it extends javax.ws.rs.core.Application, and it contains an @ApplicationPath annotation specifying that the REST interface is located at the rest value. The handler itself is registered at a path card, and the particular method for receiving identifiers is associated with the path checkin. This method also takes a form parameter named id, which contains the identifier (see **Listing 14**).

If we want to send this information to WebSocket clients, for example, to the simple web page we created in the previous section, we need to register a WebSocket endpoint as well. We can leverage CardHandler for this, because a class can have multiple annotations. Doing so, we put all the back-end code into a single file. This is probably not the best idea for a more complex production system, but by doing it here, we can see how easy it is to create an enterprise application in Java, combining REST and WebSockets.

GIVE IT A TRY

It's easy to create an embedded Java application, enrich it with a simple JavaFX user interface, connect it to a card reader, connect it to a Java EE system, and visualize the information using HTML5.

LISTING 13 LISTING 14 / LISTING 15 / LISTING 16

```
function openConnection() {
  connection =
new WebSocket('ws://localhost:8080/webmonitor/endpoint');

  connection.onmessage = function(evt) {
    var date = new Date();
    var chld = document.createElement("p");
    chld.innerHTML = date + " :  " + evt.data;
    var messages = document.getElementById("checkins");
    messages.appendChild(chld);
  };
}
```

Download all listings in this issue as text

When the WebSocket endpoint receives a connection request, the created session is stored in a Set. When the connection is closed, the session is removed from the Set, as shown in **Listing 15**.

We can now modify the checkin method created in **Listing 14**, and send the identifier to the connected WebSocket clients, as shown in **Listing 16**.

Conclusion

In this article, we saw how easy it is to create an embedded Java application, enrich it with a simple JavaFX user interface, connect it to external hardware (a card reader), connect it to a Java EE system, and visualize the information using HTML5.

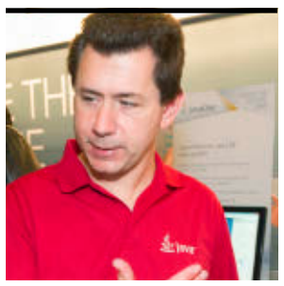
The underlying systems all use the same Java SE code. As a consequence, developers can use their existing Java skills to create end-to-end applications that span different environments. `</article>`

MORE ON TOPIC:

Java 8 Is Here

LEARN MORE

- Raspberry Pi
- JavaFX



STEVE MELOAN AND **TERRENCE** BARR

BIO

Java 8 Is Here

BARR PHOTOGRAPH BY TON HENDRIKS

Java ME 8 and the Internet of Things

Top features of Java ME 8

A major theme of the 2014 Consumer Electronics Show in Las Vegas, Nevada, was *wearable computing*. Under the hood of such diverse hardware devices as sleep-monitoring infant clothes, sports-enhancing wristbands, and pet accelerometer collars, as well as mainstay technologies such as connected vehicles, smart appliances, medical sensors, smart meters, and industrial controllers, lies a broad array of embedded and connected computations devices—increasingly termed the *Internet of Things*. By many estimates, this decade will see billions of such connected *things*, spanning a broad range of hardware and memory specifications.

But every good story begins with a challenge of some kind. For all the promise and possibility of the exploding space of embedded devices, there are also a number of obstacles. The ubiquity and pervasiveness of such technologies, along with the growing fragmentation of hardware architectures, devices, system software, and infrastructure, cries out for unified development standards in terms of programming language, software platform, tools, testing, deployment, scalability, and developer community. The hardware and software fragmentation of the current embedded space often requires piecing together the entire development stack for a given hardware platform—including the runtime, the tools, the languages, the APIs, the protocols, and the connectivity.

Java ME 8 provides a purpose-built, scalable, and flexible development and deployment environment for the embedded space, including language, standards, platform, tools, security, scalability, and a 9-million-plus community of Java developers already versed in the overall language and platform—all poised to help facilitate what many predict will be a third IT revolution. **Figure 1** shows an overview of the Java ME 8 platform. The following sections explore the top new features of Java ME 8.

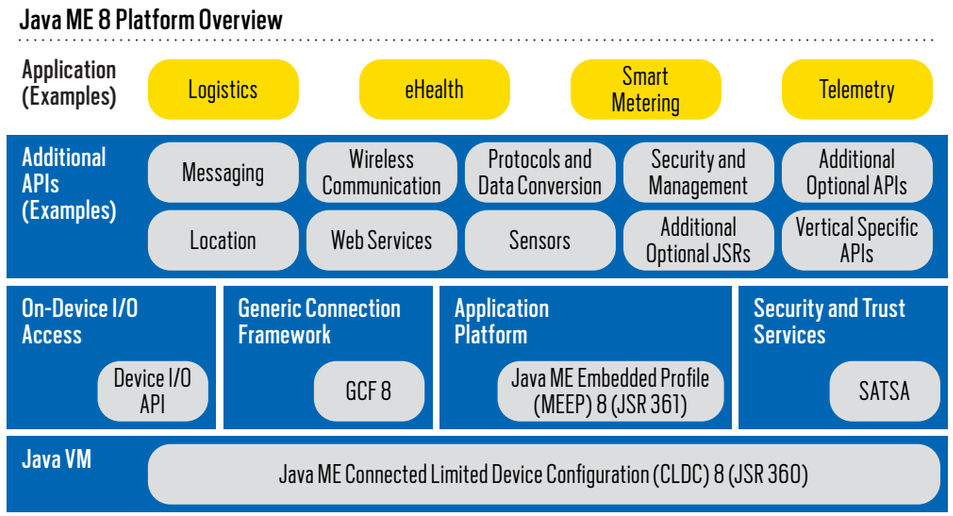


Figure 1

Connected Limited Device Configuration 8

Connected Limited Device Configuration 8 (CLDC 8)—the configuration underpinnings of Java ME 8—is an extended, strict subset of Java SE 8 that provides an evolutionary update to CLDC 1.1.1 while bringing the virtual machine (VM), Java language, and core API libraries into alignment with Java SE 8. CLDC 8 aligns the two platforms in terms of tools and programming model, while providing many new features specifically targeting the embedded space, as well as ensuring backward binary compatibility. CLDC 8

also provides such new Java language features as assertions, generics, annotations, and more. Java ME 8 is the first major step in the convergence of Java SE and Java ME, with further alignment planned for future releases.

Java developers can now more easily take their existing skills and begin writing applications for the exploding realm of embedded devices—using the same familiar platform, language, programming model, and tools. In so doing, they achieve a much faster time to market as well as cross-platform compatibility and embedded device scalability. Being mindful of appropriate language and API subsets, it's possible to create an application or library that will run unmodified across a range of hardware, from very small CLDC 8 devices up to larger Java SE 8 type devices.

Generic Connection Framework 8

In the desktop or server world, one typically finds a simple and stable Ethernet pipe into the application or system. But the embedded space often presents highly varied connection options and needs—cellular, Wi-Fi, and wired (and often multiple connectivity options in a single device)—in order to achieve the required flexibility for a specific use case.

The Generic Connection Framework 8 (GCF 8) AccessPoint API provides fine-grained and optimized connectivity control—depending upon use case, available connection options, roaming specifications and costs, and data transfer needs.

And in a predicted era of billions of connected embedded devices, the ability to locate and intercommunicate via IP address is essential. GCF 8 provides full IPv6 support to tackle the long-anticipated issue of IPv4 address exhaustion. There is also support for IP multicast—not only for installing a device into the network and detecting and connecting to peers, but also the ability to act as a server while using fully encrypted Secure Server Socket connections.

Because security is important in today's wirelessly connected world, and will be even more so with millions or billions of devices in a connected Internet of Things, GCF 8 also provides the latest security standards—including Transport Layer Security (TLS) 1.2 and secure datagram connections via Datagram Transport Layer Security (DTLS) 1.2—offering the highest levels of networked encryption and authentication.

EMBEDDED-READY

With a stable of 9-million-plus Java developers already in the field, the Java ME 8 release offers not only **vast new career possibilities**, but provides an industry-growth catalyst for the entire burgeoning embedded space.

Micro Edition Embedded Profile 8

While CLDC 8 provides the basic Java platform—in terms of the core runtime, core language features, and core APIs—it does not fully define the embedded application platform. Micro Edition Embedded Profile 8 (MEEP 8) sits on top of CLDC 8 and provides the application model and container, a means to provision applications to a system, to share code among applications, and to update software components during the lifetime of the system, for example, to add new functionality or to deploy updates and bug fixes.

MEEP 8 also provides for partitioning and modularizing the individual functional components of an application—an interface to a sensor, data filtering logic, connection to a server, and so on. In this way, processing granularity is established at the *service level* rather than at the application level, making provisioning, management, and updating of the embedded application much simpler and more elegant.

Meanwhile, *right sizing* via profile sets provides highly specific customization of memory footprint for given application use cases. A particular application might not need or use a given set of functionality. Selecting the appropriate profile set can, thereby, eliminate unnecessary functionality and memory uses. If a later use case expands functionality, additional optional packages (technology-specific APIs) can be added—offering seamless scaling.

Pertinent to both minimizing memory footprint and modularizing application functionality, MEEP 8 also provides shared libraries (LIBlets)—where multiple applications can share common library code. Read-only code thus resides in one shared place in memory, rather than within each individual application space. As an added bonus, such shared library space can be updated or altered just one time for all applications accessing it.

Meanwhile, so that applications running on embedded devices can maximally collaborate and communicate, MEEP 8 offers both asynchronous event-based messaging (akin to a publish-subscribe mechanism) and synchronous

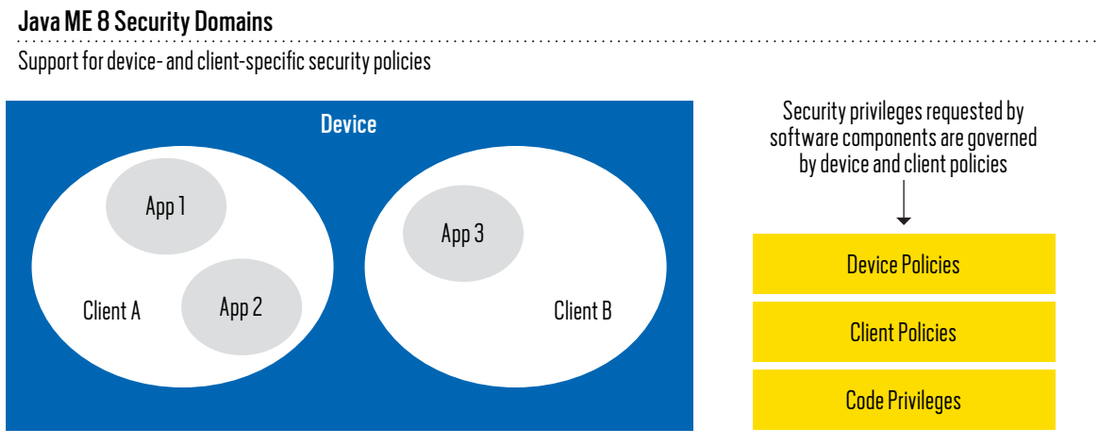


Figure 2

Inter-MIDlet Communication (IMC; akin to a UNIX pipe).

MEEP 8 also provides a sophisticated and fine-grained security model, enabling use case–specific security policies for authentication and authorization (see **Figure 2**). Sandboxed execution ensures secure code loading, verification, datatyping, and configurable permission controls for access to resources and data. Individual software components are associated with specific clients and inherit their allowed privileges. And such permissions are confirmed at every access event.

Device Access API

The Device Access API for Java ME 8, already present in previous Java ME releases, is critical functionality that enables embedded Java applications to access peripheral devices such as buttons, switches, sensors, LEDs, LCD screens, audio, and more in a platform-neutral way. It now expands access to more devices and adds new functionality. And *late binding* functionality allows for the addition of new peripheral types without changing the API.

Java ME SDK 8

Java ME SDK 8 offers a full-featured and purpose-built toolset for embedded development, including emulation and simulation environments for a variety of devices—so applications can be developed in parallel with hardware, thereby vastly improving time to market. Developers can simulate, test, and debug in emulation mode, and then once the actual hardware devices and peripherals are in place, have much greater confidence in the final testing of applications.

In addition to security, memory, and network monitoring tools, the Developer Agent in Java ME SDK 8 provides levels of access and control beyond those of typical embedded development environments, including the ability to reset devices, access file systems (on screenless devices), make configuration changes, and more—all from Java ME SDK 8. Meanwhile, NetBeans plug-ins offer full-featured, IDE integration with the Java ME SDK—providing a familiar and unified development environment across Java ME, Java SE, and Java EE, with support for Eclipse becoming available alongside the Eclipse support timeline for Java SE 8, Java ME 8, and Java ME SDK 8.

Conclusion

Earlier eras of the cell phone space saw fragmented hardware that was, in many ways, similar to today's embedded technology space. Part of what ultimately helped spawn the explosive growth of smartphones and mobile apps was the establishment of standard development platforms, languages, and tools.

Java ME 8 now offers just such a comprehensive, purpose-built platform for connected embedded devices and the Internet of Things. And with a stable of 9-million-plus Java developers already in the field, the release offers not only vast new career possibilities, but provides an industry-growth catalyst for the entire burgeoning embedded space.

So get started today. Oracle Java ME Embedded 8 (the Oracle implementation of the Java ME 8 standard) is now available in Early Access form for the Raspberry Pi Model B (ARM11) and STMicroelectronics STM32F4DISCOVERY (ARM Cortex-M4) platforms.

And an Early Access version of the Oracle Java ME SDK 8 is also now available—supporting development of Java ME 8 Early Access–based applications in an emulation runtime for Windows 7 as well as for the Raspberry Pi and STMicroelectronics devices mentioned previously.

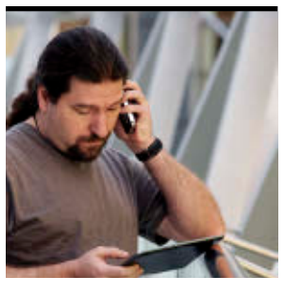
The Oracle Java ME Embedded 8 and Oracle Java ME SDK 8 Early Access releases are available at no cost for evaluation and development purposes under the Oracle Technology Network Developer License. `</article>`

MORE ON TOPIC:

Java 8 Is Here

LEARN MORE

- Oracle Java ME Embedded FAQ
- JSR 360 (Connected Limited Device Configuration 8)
- JSR 361 (Java ME Embedded Profile)



Part 2

Take Time to Play

Learn how to take advantage of the client tier.

TED NEWARD

BIO

In Part 1, we examined the Play Framework, a lightweight, stateless, server-side web framework that is written in Scala and is usable from either Scala or Java. We explored a simple "hello world" application, examined Play's model-view-controller (MVC) components, talked about its routing scheme and syntax, and played with HTML form input and processing. We noted that the code was minimalistic, straightforward, and relatively easy to understand.

However, web applications built for 2013 are either entirely "client-side" using JavaScript technologies such as AngularJS or Ember.js, or they are a hybrid of client-side JavaScript using components and scripts (usually, either based on jQuery or using that library directly) and server-side processing. Now, with the rise of more-powerful mobile devices, we're seeing mobile clients that live natively on the device and do significant work on the device, rather than on the device's web browser.

While such an approach has its strengths, there are significant advantages to pushing more of the display and processing work into the client tier. For starters, the fewer round-trips the application has to take back to the server, the more responsive the application appears. In a traditional, entirely server-side HTML-based application, each form submission—and the subsequent processing and response—requires a complete round-trip to the server. No matter how fast the web server responds, there's going to be a noticeable pause and lag. There's also little the application can do to improve this, because it depends on the network in between the browser and the server.

On top of the responsiveness gained from using the client tier, browsers can be insecure. Many applications work with sensitive data that shouldn't be in the browser until the user has demonstrated proof of identity and has the appropriate rights to see that data. Unfortunately, between the plain-text nature of HTTP and the fact that it's trivial to install developer tools into any browser, anything hosted in the browser is a short step from being displayed in plain text. (Yes, you can use HTTPS to secure the data in between the browser and the server; however, you can't stop a malicious user from using developer tools to view the running code, and you can't stop a malicious user from using Telnet to view the HTML and JavaScript in raw text.)

EASY ENDPOINTS

Play makes it nearly trivial to build RESTful endpoints that offer data-only processing elements. In fact, Play makes it easier to build these endpoints than to build a traditional web application.

These reasons, combined with a basic desire to offer APIs over HTTP as an integration effort, or even as a product itself, yield a new trend in server-side Web development: building RESTful (or, if you prefer, "REST-ish" or "REST-like") endpoints that offer data-only processing elements. This, in many respects, is the "new web" and any modern server-side web developer should be prepared to build these kinds of processing points if they're going to support mod-

PHOTOGRAPH BY PHIL SALTONSTALL

ern web development. Fortunately, Play makes it nearly trivial to build these kinds of endpoints. In fact, Play makes it easier to build these API endpoints than to build a traditional web application.

Revisiting the MVC Pattern

Recall that in a traditional MVC application, models contain the data and entities the application manipulates, controllers do the actual heavy lifting, and views handle the display of that data and work. Because all of this is executing on the server, the user generally sees only the views, and the models are generally close to the database structures they represent, or the database structures are generated off of those models. (Which approach the application takes usually depends on whether the architect is of the "business objects first" or the "database first" camp.)

In an application where the views are done on the client tier, the server-side perspective changes slightly. While it's simple to pronounce that the server-side MVC still works exactly as before—with the views now being pure data and with JavaScript Object Notation (JSON) or XML being the preferred formats—things start to get confusing. The "views" are handing back entities of pure data—which to the client-side technology might be "models" in their MVC lexicon. Throw in the cost of traversing the network, and it's not always reasonable to expect that client-side "models" will follow server-side models on a one-to-one basis.

Consider the ubiquitous shopping cart in an e-commerce application. In a server-side scenario, the Customer model object provides the customer's name, and linked Address objects provide the shipping and billing addresses for the order, while individual Item objects provide the price, quantity, and description for each item in the cart. All this information can be neatly partitioned into the individual objects, complete with business logic to ensure that tax is calculated correctly, shipping-and-handling charges are dealt with appropriately, and so on. The server-side view can access each of these objects in turn while constructing the HTML code for the checkout page, and when it's all assembled, shoot back the complete HTML view for the user.

NIFTY STUFF

The Play Framework is a great example of how **rethinking the server-side web stack** can lead to some really nifty enhancements.

When we move that view processing to the client, however, it's not as simple as it was before. If the client-side view processing tries to access each of the data elements from the relevant objects on the server, the application will spend nearly a dozen round-trips to the server just to assemble the view for a single shopping cart. Not only will this crush the application's performance, it will kill scalability.

Instead, we need to start thinking about a different model: a client-side model that is sometimes called a *view model*. We can take either a big-picture view and say that the MVC pattern is still in effect—with the views being on the client, displaying view models, and sending input to controllers on the server, where they are transformed into *server models* and manipulated as necessary—or we can take a more fractal view and suggest that MVC is now happening in two places: once on the server (with the server views being the JSON/XML that is shipped back and forth) and once on the client (with the client controllers performing only the work that it makes sense to do there, such as formatting and basic data validation). Neither approach is entirely right or wrong, but attempting to mix and match the two styles usually results in chaos and confusion.

This perspective of different models for different tiers is sometimes called the *Model-View-ViewModel* (MVVM) pattern, and it originated around Microsoft's Silverlight and Windows Presentation Foundation, but there are enough subtle differences between view model and MVVM that I don't care to use the name. Regardless of what the model is called, it requires rethinking how an application is built, beyond what Play can provide. But what Play can provide is a lot of help for building the server-side endpoints.

Endpoint Design

Let's create a new application that will supply information about computer systems across history. (This is a variant of the Play computer-database sample, which ships as part of the Play distribution. For a preview, look at the samples/java/computer-database subdirectory of the Play distribution and do a trial run to see that sample in action.) These computer systems have a fairly flat structure consisting of four fields: the name of the computer, its introduction date, its discontinuation date (if there is one), and the company that offered the computer.

From an API design perspective, for the endpoint URLs, we have one resource—computers. (You might argue that companies should be another resource, but for simplicity, I'm limiting it to just computers.)

We'll want to view the entire list of computers and obtain the data for a particular computer. So at a minimum, we want two endpoints:

- We'll use a GET to /computers to get a list of all computers.
- We'll use a GET to /computers/<id> (where <id> is a unique identifier) to get the data for a particular computer.

Historically, in the database world, the identifier has been an incrementing integer, but some in the REST camp will argue that URLs should always be meaningful and, therefore, the identifier should be the unique name of the computer. For simplicity, I'm going to stick with an integer, but Play won't care either way. So if you really want /computers/thinkpad-w510 as a URL pattern, you can have it.

Models, Views, and Controllers

Because we installed Play in Part 1 of this series, type play new computer-database at the command prompt to start the Play console and create a new application called computer-database. Then we will start creating model classes. We'll start with Computer, and just to prove a point, let's break Company out as a separate server-side model object, as shown in **Listing 1**. (These will obviously need to be in separate files; I'm combining them here because they share the same package and import statements.)

For simplicity, I'm not worrying about database storage, but Play works well with Java Persistence API (JPA) and ships with two other data-access APIs (Anorm for Scala developers, Ebean for Java developers). And it will work with just about any other Java data-access layer.

Now we want to have two controller actions that respond to two URL routes: one for the home page and one for the computers endpoint we described earlier (see **Listing 2**). I leave the home page controller action in place because people often browse to the root of a domain expecting to see something. I modify the index.scala.html view to display a simple "There's nothing to see here" message. Because I like the convention, I've prefixed the computers endpoint with /api; that makes it clearer that this is intended to be a developer API, not a set of human-friendly web pages.

Next, it's time to write the controller action, but first, I need a simple database of computers and companies to work with (see **Listing 3**). (Yes, I'm aware that the

LISTING 1 LISTING 2 / LISTING 3

```
package models;

import java.util.*;
import play.data.format.*;
import play.data.validation.*;

public class Company {
  public Long id;
  public String name;
  public Company(Long i, String n) {
    id = i; name = n;
  }
}

public class Computer {
  public Long id;
  public String name;
  public Date introduced;
  public Date discontinued;
  public Company company;
  public Computer(Long id, String n, Date i, Date d, Company c) {
    this.id = id;
    this.name = n;
    this.introduced = i;
    this.discontinued = d;
    this.company = c;
  }
}
```

Download all listings in this issue as text

use of the Date constructor will generate a deprecation warning, but I don't care for this example. And yes, these "introduced" and "discontinued" dates are fictitious.)

Browsing to the root of the application (http://localhost:9000, if you're running the server on the default port from the Play console) shows the message specified in index(), so that's all good.

Adding a controller action for the /api/computers endpoint requires just another static method (see **Listing 4**), and while we could create a computers.scala.html view for it, which iterates through the collection of Computer objects in the computers field, it's actually easier to ask the Jackson JSON library embedded in Play to do the conversion for us.

Note that the "model" being sent back here is not the same as the model objects from which it was generated—in particular, it's being flattened out, and the Company objects are being embedded "by value" within the JSON generated from the Computer objects referencing them, as shown in **Listing 5**.

This is common for client models, both because it eases the work needed for this model in the client, and because the JSON syntax doesn't really support object references. But, as you can also tell, the toJson method didn't really do much for us with the Date objects—it decided to hand back the millisecond representation of the Date, rather than a more-sane "Jan 15 2012" type of representation. This isn't a tutorial on the Jackson JSON library—so I'll defer fixing that to readers.

If your users (or clients) prefer XML over JSON, Play doesn't have a handy library bundled with it to handle XML, but Java ships with Java API for XML Binding (JAXB), which you could use to generate XML out of Java objects governed by attributes on the model classes. Numerous other libraries for generating XML out of Java objects also litter the web—pick your favorite. On top of that, Scala has some great built-in support for XML primitives, so this might be a case where using Scala instead of Java yields a specific and noticeable advantage.

One note about the use of JSON versus XML for endpoints—although enterprise-class organizations voted with their pocketbooks for XML (and SOAP, Web Services Description Language [WSDL], and a whole library of WS* specifications), the rest of the web seems to have voted with their feet for JSON. Some systems offer both by routing based on the suffix of the URL pattern requested. So, for example, if the client wants a JSON representation, a request is issued to /api/computers.json, but if the client wants an XML representation, a request is issued to /api/computers.xml. Supporting this would be trivial to do in the routes file (see **Listing 6**).

Alternatively, the controller action could examine the incoming request for the suffix on the path, but this involves parsing, is less declarative, and represents unnecessary work that the framework could do for us if we declare these routes explicitly here. (Besides, some representations might not be supported for all resource types or there might be only two such representations for this system, so do the simplest thing that could work.)

What we've done so far handles the case where we want to see the complete list of computers, but what about the scenario where we want to see only a single computer using JSON? Traditional web applications would suggest that this is a query parameter to the URL, for example, /api/computers?id=2, but the preferred approach for API endpoints is to embed the query parameter as a meaningful part of the URL, such as /api/computers/2. Before the panic or heavy sigh sets in about having to parse the URL path, check out **Listing 7**.

In essence, the routes file allows the declaration of *bound wildcards* (my term), meaning the framework will map a value in a URL request such as /api/computers/2 to a corresponding placeholder in the routing pattern (the 2 will be mapped into the placeholder id), and then pass that value to your controller action method. This, then, makes

LISTING 4 LISTING 5 / LISTING 6 / LISTING 7

```
public static Result getComputers() {
  return ok(Json.toJson(computers));
}
```

Download all listings in this issue as text

the controller's action method nearly trivial to write (see **Listing 8**). In fact, it's more work to convert the incoming Long parameter to an int for use in the List.get() method.

We Are Now Accepting Input

But endpoints like this aren't read-only most of the time—the server usually needs some facility for accepting changes to existing resources (either a PUT or a POST operation, depending on which church of REST you attend), creating new resources (either a POST or a PUT operation; the opposite of whatever you chose for changes), and deleting resources (DELETE).

DELETE maps well to a standalone route using a bound wildcard similar to the earlier GET example. Accepting changes to existing resources and creating new resources, however, require that the controller action accept incoming values from the HTTP request, and usually this means either a slew of query parameters or accepting a JSON (or XML) document in the body of the request.

Query parameters are fairly easy to extract out of the URL, because they're bound to parameters to the controller action method from the routing declaration in the routes file. So, for example, if we want to add a "add a new computer to the system" endpoint, we can map this in the routes file as shown in **Listing 9**.

Assuming that clients make a GET request to /api/computers?name=MacBook%20Air2&companyID=1, the name query string parameter will be bound to the name parameter in addComputer(), companyID will be similarly matched against company ID, and MacBook Air2 and 1 will be passed, respectively.

The query parameter approach has its limitations, however, particularly when there are more than just a few fields, or when a field consists of even a small amount of human-readable text (something beyond a name or title). For those cases, the preference is to send a body in the HTTP request, usually consisting of JSON or XML describing the resource we want to create or update.

(From an API perspective, realistically, the only difference is that creating a resource will be a POST or a PUT to /api/computers, whereas an update will be a POST or a PUT to /api/computers/<id>, where <id> is the identifier of the resource being updated. Arguably, if the ID is part of the document being sent, PUT or POST to /api/computers could be an "upsert," meaning the controller action would examine the document and either update or insert, depending on whether an ID is present. But this sometimes feels too subtle and occasionally yields an odd bug.)

LISTING 8 LISTING 9 / LISTING 10 / LISTING 11 / LISTING 12

```
public static Result getComputer(Long id) {
  int index = (int)((long)id + 1);
  return ok(Json.toJson(computers.get(index)));
}
```

Download all listings in this issue as text

Parsing a request body in Play is straightforward and easy. Assuming that we want to add a computer by accepting POST requests at /api/computers with a JSON body that looks like **Listing 10**, the route would look like **Listing 11**, and the controller action method, using the Jackson library to parse the incoming JSON and using its methods to pick apart the JSON, would look like **Listing 12**.

Although you wouldn't likely do this in your own "create" endpoints, here I'm returning the whole list of Computer instances upon a successful insert, so I can verify that it's there and demonstrate that these controller action methods are, just as they've always been, standard Java methods that can be daisy-chained on top of one another when it makes sense to do so.

Unfortunately, it's harder to test a POST from the browser, and I don't really want to set up a form in the application just to test the POST functionality (particularly since I'd have to do some serious JavaScript

LISTING 13

```
curl --header "Content-type: application/json" -X POST
http://localhost:9000/api/computers -d
"{\"name\":\"MacBook Air2\",\"companyID\":1}"
```

Download all listings in this issue as text

Fu to transform the form fields into a JSON body). Thankfully, curl, a command-line utility for sending HTTP requests of forms, comes to the rescue. (If you're on a Mac or Linux box, it's already installed. If you're on Windows and you haven't downloaded it, it's a trivial download, and it's so useful in web programming, you should grab it now.)

In **Listing 13**, the server will take the incoming JSON, parse it, add it to List<Computer>, and redirect the request to the getComputers() controller action, which will then show the full list, including the new computer just added. (If you're on Linux or Mac, you don't need to backslash-escape the quotes in the command—that is necessary only on Windows.)

Conclusion

Play is definitely an interesting piece of work, and these two articles have only scratched the surface of the framework. Thanks to its integration with Akka, for example, it's nearly trivial to take an incoming HTTP request and farm the processing out to an Actor for better concurrent execution.

We haven't touched any of the data-access APIs that ship with Play; two ship with the framework (Anorm for Scala and Ebean for Java), plus there's a sample demonstrating the use of JPA. Then there are filters and action composition (using attributes to daisy-chain controller actions without having to explicitly chain the calls yourself), WebSocket support, and much more.

The Play Framework is a great example of how rethinking the server-side web stack can lead to some really nifty enhancements. Even if you're not ready to give up Java EE, looking at Play can offer some useful insights and inspiration for the design of your code.

Most of all, remember Play's mantra: Have fun! `</article>`

LEARN MORE

- the Play Framework
- "Take Time to Play" (Part 1 of this series)

//fix this /

In the January/February 2014 issue, Romanian polyglot developer Attila Balazs presented us with an error-logging challenge. He gave us code that requires all callsites that submit runnables to be modified and asked for a simpler solution. The correct answer is #2: Extend ThreadPoolExecutor and override the afterExecute method. The javadoc for this method gives the exception logging as an example for why the afterExecute method might be overriden. All we need to do is to replace System.out.println with the proper call to the logger.

Answer #1 won't work because the exception isn't unhandled—it is caught by the ThreadPoolExecutor, and it doesn't make its way to the UncaughtExceptionHandler. #3 and #4 might or might not work, depending on the way tasks are submitted to the executor. #3 doesn't cover the methods from #4, and #4 doesn't cover the void execute(Runnable command) method. An implementation overriding the submission methods needs to override all four of them to cover all situations.

In this issue, Balazs offers us another challenge. This time the problem is around a concurrency bug.

1 THE PROBLEM

A programmer needs a collection of counters. Each counter is identified by a name and starts at 42. The programmer needs to implement an increment method that takes the name of the counter, increments it, and returns the new value or initializes it to 42 if it's the first time the particular name was used.

2 THE CODE

The programmer comes up with the following code:

```
private final ConcurrentMap<String, Integer> cnts = new
ConcurrentHashMap<>();

int increment(String name) {
    if (cnts.containsKey(name)) {
          Integer cnt = cnts.get(name);
          cnts.put(name, cnt + 1);
        return cnt;
    } else {
          cnts.put(name, 42);
        return 42;
    }
}
```

However, the programmer finds that sometimes this code mysteriously "drops" incrementation attempts (that is, the increment method is called twice but the counter is incremented only once).

3 WHAT'S THE FIX?

1) Add the keyword synchronized to the method declaration.
2) Use a Hashtable instead of a ConcurrentHashMap.
3) Implement an optimistic concurrency-style algorithm.
4) Use an AtomicLongArray instead of a ConcurrentHashMap.

GOT THE ANSWER?

Look for the answer in the next issue. Or submit your own code challenge!

ART BY I-HUA CHEN