//MAY/JUNE 2015 /

{ JAVA GROWS UP }

Twenty years of language evolution and JVM innovation

ORACLE.COM/JAVAMAGAZINE

ORACLE

//table of contents /

EDITORIAL

Editor in Chief
Andrew Binstock

Interim Associate Editor
Kay Keppler

Copy Editors
Claire Breen, Karen Perkins

Section Development
Blair Campbell – Java in Action
Michelle Kovac – Java in Action, Features
Bob Larsen – Java Nation
Yolande Poirier – Java Nation

Technical Reviewers
Stephen Chin, Reza Rahman, Simon Ritter, James Weaver

DESIGN

Senior Creative Director
Francisco G Delgadillo

Design Director
Richard Merchán

Contributing Designers
Jaime Ferrand, Diane Murray, Arianna Pucherelli

Production Designers
Sheila Brennan, Kathy Cygnarowicz

PUBLISHING

Publisher
Jennifer Hamilton +1.650.506.3794

Associate Publisher and Audience Development Director
Karin Kinnear +1.650.506.1985

Audience Development Manager
Jennifer Kurtz

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 at no cost to qualified subscribers by Oracle, 500 Oracle Parkway, MS OPL-3A, Redwood City, CA 94065-1600.

Digital Publishing by GTxcel

BIO

Warning: We're Going Technical

Java Magazine heads in a new direction.

Several weeks ago, I was hired by Oracle to head up *Java Magazine*. This happy event means that I get to write about Java from the very heart of Java. *How cool is that?* The joy I feel is akin to the excitement of setting out on a journey for which one has long prepared. I've been an editor of developer publications for many years, most recently heading up *Dr. Dobb's*. I've also spent much of my free time during the last 10 years programming in Java and, to a lesser extent, Groovy.

Mixed with the joy of this new work is the awareness of the long road ahead, defined by the endpoint I am driving toward; so, let me share my plans for the new direction of our magazine.

In simple terms, I plan to take *Java Magazine* in a radically more technical direction. The editorial team and I will be catering squarely to Java developers who work regularly with code. The magazine will both reflect and cater to our shared love of programming, our passion for great tools and technologies, and the abiding enjoyment we all derive from Java and JVM languages.

To do this, we'll enlist more Java experts, accomplished programmers, algorithm junkies, design mavens, build wizards, bad-code scolds, experienced teachers, and skilled architects to share their wisdom in our pages. We'll also explore tools—especially open source tools—and important open source projects.

I also expect to remove some sections, mainly those that don't focus tightly on programming. There will be fewer case histories in *Java Magazine*—and those that do appear will bear directly on a relevant technical topic. We'll also

PHOTOGRAPH BY BOB ADLER/GETTY IMAGES

transform other sections. For example, the section that used to contain book blurbs will run detailed reviews, signed by the reviewers. (The first book review appears on page 11 of this issue.)

Finally, you'll see fewer articles telling you how great Java is. There's no need for that. You get it. I get it. Let's move forward.

You can expect to see these changes roll out over the next few issues. Changing magazines is not a stop-the-world-while-I-do-this kind of project. It's more like changing a tire on a moving truck. In this issue, you'll see some initial tweaks. For example, code is now presented in a monospaced font (hey, hey!). There's also a new last page about how to contact us and propose articles. The next issue will have more-substantial changes, and the one after that will have a lot more of the content I have promised. By year's end, the transition will be nearly complete and we should be rolling in high gear.

A word of thanks to my predecessor, Caroline Kvitka, for bringing *Java Magazine* to this point (where we have nearly a quarter of a million readers), and to Kay Keppler, who was the key to making this transition possible.

Finally, allow me to emphasize one more point: I really like hearing from you. Comments, plaudits, grumbles, contrasting opinions, corrections, proposals for articles—these are all welcome and make for a better magazine. My address is at the end of this editor's note, and more contact information appears on the Contact Us page at the end of the magazine. Our staff reminds me that the last paragraph of that page should not be overlooked.

Looking forward to this journey together,

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

PS: For more on my background, you can click on the link marked "Bio" on the previous page.

INTRODUCING

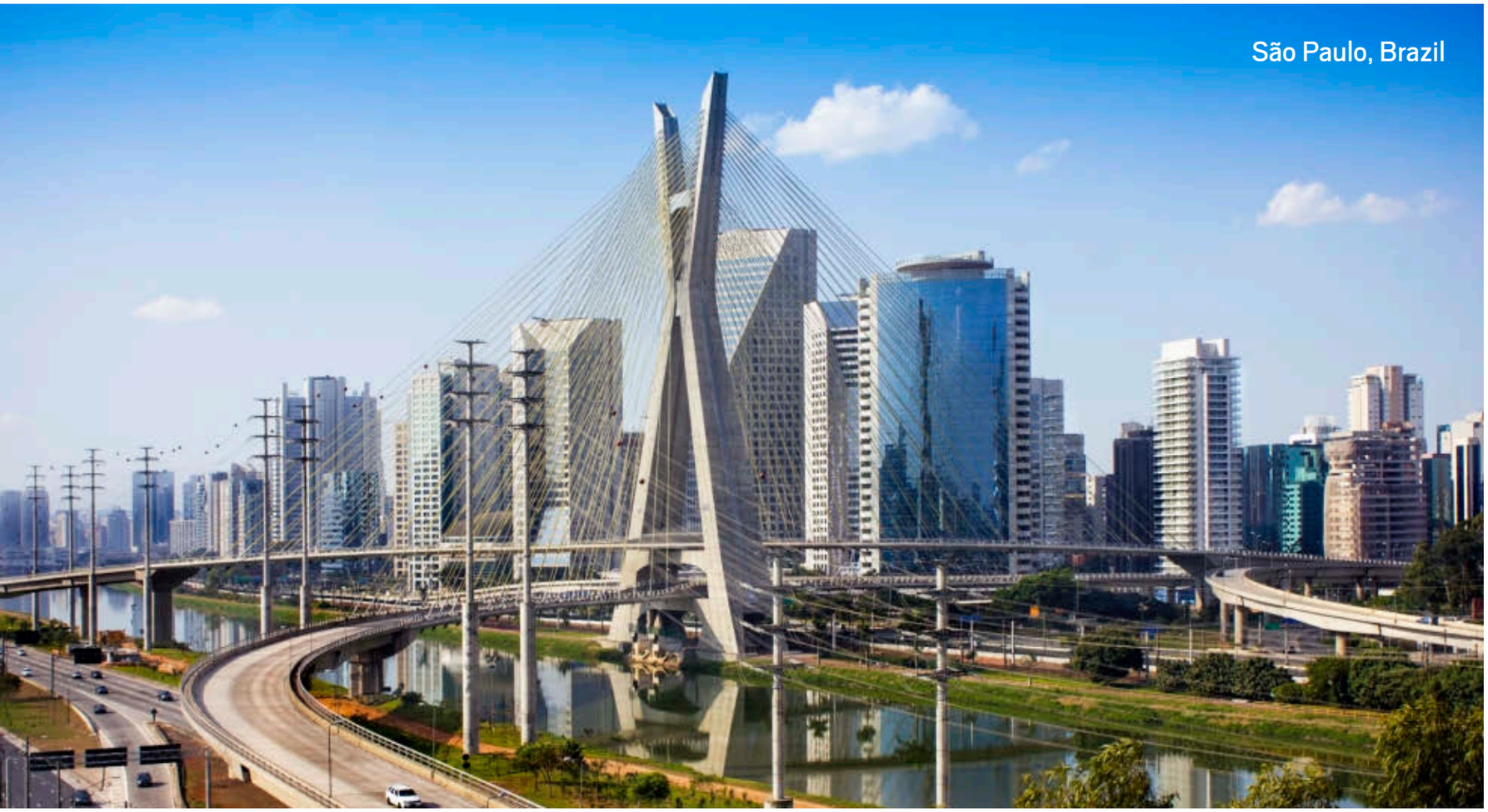
Atalasoft MobileImage®

Image Processing Controls Camera Capture Control Barcode Reading Controls

iOS/Android Capture SDK

Atalasoft
A Kofax Company

DOWNLOAD FREE 30 DAY TRIAL


São Paulo, Brazil

EVENTS

JavaOne Latin America *JUNE 23–25*
SÃO PAULO, BRAZIL
JavaOne Latin America is the premier Java conference in Latin America. Visionaries and world-renowned speakers will present conference and hands-on sessions in five tracks, including clients and the user interface; core Java platform; Java and the Internet of Things; server-side Java; and a new track about Java, DevOps, and the cloud.

QCon New York
JUNE 8–12
NEW YORK, NEW YORK
QCon is designed for technical team leads, architects, engineering directors, and project managers who influence innovation. The event includes two days of tutorials followed by a three-day conference. One track focuses on Java innovations.

Devoxx UK
JUNE 17–19
LONDON, ENGLAND
The innovative London Java User Group organizes this conference, and puts developers front and center with the theme "The League of Extraordinary Developers." The schedule is packed with presentations, hands-on-labs, and Birds-of-a-Feather (BOF) sessions. Developers can meet top British and international experts in a casual atmosphere.

Devoxx Poland
JUNE 22–25
KRAKOW, POLAND
Seven tracks focus on Java technology. Participants can dive deep in the conference sessions, evening BOFs, and workshops and labs led by world-class trainers.

Java EE Summit
JUNE 24–26
MUNICH, GERMANY
The Java EE Summit includes 14 intensive workshops led by top German-speaking Java EE experts. Developers and software architects will receive in-depth knowledge to plan and implement successful Java EE 6 and Java EE 7 projects.

PHOTOGRAPH BY GETTY IMAGES

EVENTS

FISL
JULY 8–11
PORTO ALEGRE, BRAZIL
The International Free Software Forum (FISL) event is jam-packed with lectures, workshops, demonstrations, and presentations.

Java Forum
JULY 9
STUTTGART, GERMANY
Organized by the Stuttgart Java User Group, Java Forum attracts more than 1,000 participants who learn about Java and the Java environment. The forum includes lectures, presentations, workshops, demos, and BOF sessions.

OSCON
JULY 20–24
PORTLAND, OREGON
Open source pioneers, experts, and innovators have gathered at OSCON over the past decade. With a dozen tracks and hundreds of sessions, the conference has practical tutorials, inspirational keynotes, and a wealth of information on open source languages and platforms.

ÜberConf
JULY 21–24
DENVER, COLORADO
ÜberConf is part of the No Fluff Just Stuff Software Symposium Series that explores the ever-evolving ecosystem of the Java platform. The conference offers 100 technically focused sessions including more than 25 hands-on workshops about architecture, cloud, security, enterprise Java, languages on the JVM, build/test, mobility, microservices, Docker, and agility.

The Developer's Conference (TDC)
JULY 21–25
SÃO PAULO, BRAZIL
One of Brazil's largest conferences for developers, IT professionals, and students offers Java tracks over five days. Java-focused content includes introductory sessions in the Java University track, and advanced and intermediate sessions in the Java EE, mobile, and embedded tracks.

PHOTOGRAPH BY GETTY IMAGES

Fix This—Solution

In the March/April 2015 issue's Fix This, Stephen Chin showed us a problem of the new TemporalAdjuster class introduced with the Java SE 8 Date and Time API: What if you want to do something specific to your application, such as finding the next occurrence of Friday the 13th? Here's a quick explanation of each possible issue:

1) Variations in month. By using the `temporal.plus` method, variances in month length are automatically fixed by the underlying `LocalDate` instance.
2) A time zone issue. The underlying implementation of a `Temporal` passed in to the `TemporalAdjuster` is responsible for taking care of compensating for time zones (which would be the case if a `ZonedDateTime` was used, for example).
3) An error in the formula. This is the problem. The formula works fine if the given day of the month is on or before the 13th or the given month has no Friday the 13th. If both of those cases coincide, then the `TemporalAdjuster` incorrectly returns a date in the past. To fix this, you can simply add a check for the day of the month and increment the month in advance of running the loop.

```
if (temporal.get(
  ChronoField.DAY_OF_MONTH) > 13)
    temporal =
      temporal.plus(1, ChronoUnit.MONTHS);
```

A simple fix, and thanks to the new Date and Time API you can focus on debugging your algorithm rather than second-guessing whether your date fields are 0, 1, or 1900 based. (For an explanation, check out slide 10 of Chin's ZombieTime talk.)

FEATURED JAVA SPECIFICATION REQUEST

JSR 363: Units of Measurement API

JSR 363: Units of Measurement API seeks to support "the programmatic handling of physical quantities and their expression as numbers of units." It provides type-safe support for units of measure that is necessary for machine-to-machine communication and the Internet of Things, especially when you're working with sensors or in the manufacturing, engineering, or scientific fields. You have no need to worry if the length passed to or from that API is in feet or meters—or even leagues!

The Specification Leads for JSR 363—**Jean-Marie Dautelle**, **Werner Keil**, and **Leonardo Lima**—are working with the support of government organizations, private companies, Java user groups, and individuals like you. If you'd like to comment on or contribute to the work on JSR 363, or even just track the progress, visit java.net and help steer the future of Java.

FEATURED JAVA SPECIAL INTEREST GROUP: NEW YORK

Long ago, on September 21, 1995—back when the years still began with a *1* and flannel was considered the epitome of fashion—a group of Java developers gathered after a Java developer conference. Because the term *Java user group* didn't exist yet, they called themselves the New York Java Special Interest Group (NYJavaSIG).

Since that time, membership in the NYJavaSIG has grown to almost 8,000 of what **Frank Greco**, NYJavaSIG founder and leader as well as lead guitarist for the NullPointers (the now world-famous band composed entirely of Java developers), describes as "extremely opinionated technologists." He quickly follows up with "But hey...it's New York." Monthly meetings average about 240 members, and bimonthly hands-on workgroups with titles such as "Cloud/HPC and Languages" ensure that something will pique the interest of almost any Java developer.

The future of NYJavaSIG looks to be every bit as promising as its past. According to Greco, "The local NY Java community is enjoying a renaissance thanks to the release of Java 8 with new functionality and power. Every meeting seems to add many new NYJavaSIG members. And it appears to be a larger phenomenon with Java excitement growing globally. With more cloud and IoT [Internet of Things] functionality coming and with the Java ecosystem growing even bigger, there's still a lot of excitement yet to come."

If you plan to be in the New York area and would like to participate in an NYJavaSIG event, look them up at the NYJavaSIG website or on Twitter @nyjavasig.

JAVA CHAMPION PROFILE

ARUN GUPTA

***Arun Gupta**, the director of technical marketing and developer advocacy at Red Hat, promotes Red Hat technologies that focus on JBoss middleware. He has been an avid proponent of the Java community and technologies since his days as a founding member of the Java EE team at Sun Microsystems. He travels the world speaking about a wide variety of topics and is a JavaOne Rock Star and best-selling author.*

Java Magazine: Where did you grow up?
Gupta: Delhi, India

Java Magazine: When and how did you first become interested in computers and programming?
Gupta: My elder sister was doing a Master of Science degree in computers when I was growing up, and I found that intriguing and engaging. My first interaction with computers was when I got a master's degree in computing myself.

Java Magazine: What was your first computer and programming language?
Gupta: Intel 386, 4 MB RAM, 200 MB disk. First programming language: Pascal.

Java Magazine: What was your first professional programming job?
Gupta: My first job was way back in 1996 doing Java programming.

Java Magazine: What do you enjoy for fun and relaxation?
Gupta: I like doing long-distance running, teaching STEM [science, technology, engineering, and mathematics] workshops to kids, and cooking with family.

Java Magazine: What happens on your typical day off from work? Do you spend time with family and/or friends? Do you wander off alone?
Gupta: A day off is typically packed with kids' activities, watching a movie with the family, Devoxx4Kids workshops, and socializing with friends.

Java Magazine: What "side effects" of your career do you enjoy the most?
Gupta: Meeting people of different cultures from all around the world. I've been to more than 40 countries so far.

Java Magazine: Has being a Java Champion changed anything in your daily life?
Gupta: Bragging rights. I also feel a social responsibility to grow the Java Champion circle bigger and create more of us.

Java Magazine: What are you looking forward to in the coming years?
Gupta: I want 9 million Java developers to grow to 10, 11, 12, and 13 million, and keep growing.

Follow Arun Gupta's blog or reach him on Twitter @arungupta.

TeamCity

TeamCity builds your day

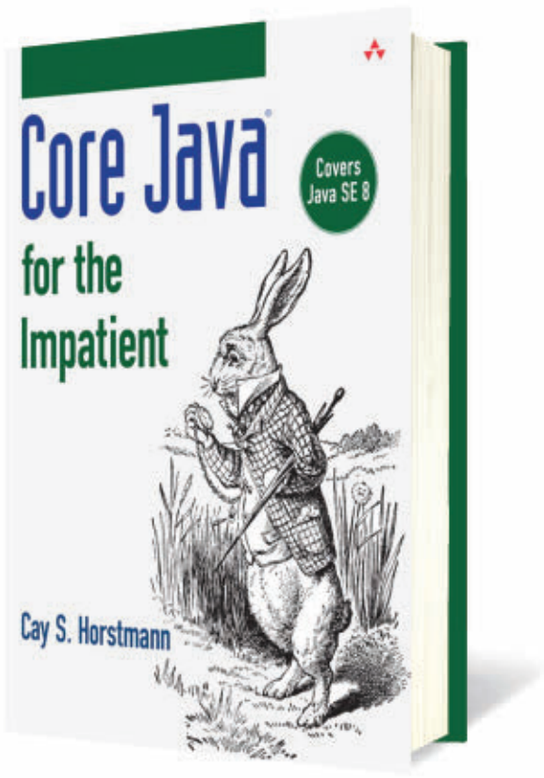
Hassle-free continuous integration and deployment server by JetBrains

Try free. Use free.

START BUILDING TODAY

JetBRAINS

//java books /



CORE JAVA FOR THE IMPATIENT

By Cay S. Horstmann
Addison-Wesley Professional, February 2015

This book follows on the slimmer volume by Horstmann, *Java SE 8 for the Really Impatient*, which came out in early 2014. That book targeted existing Java developers and explained concisely what was new in Java SE 8 (as well as some important but overlooked features in Java SE 7).

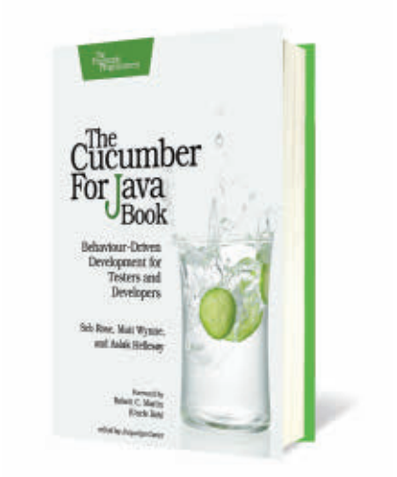
This 500-page volume is an introduction to the full Java SE 8 language, aimed at developers coming to Java from another language, but not at novices new to programming. It is a bridge between the earlier Java SE 8 book and the upcoming revision of *Core Java*—the two-volume masterpiece that Horstmann coauthors with Gary Cornell—which should see light later this year.

To my eyes, the added introduction to the language seems rushed and a bit too thin. For example, Horstmann writes, "Java has ... five integral types." Six pages later, he asserts that "four [primitive types] are integer types." Only careful comparison reveals that the char type has been reclassified from its correct group (integral primitives) to an invented one (integer primitives) without explanation. It's not a big point, but it suggests that a reader looking to learn Java would find the early sections confusing, whereas they should be the simplest.

Once into the meat of Java SE 8, however, Horstmann displays his widely recognized skill at explaining difficult concepts clearly and choosing code examples that cogently support his textual presentation. The book also contains many wise and pragmatic recommendations about the right way to do things (a central strength of his previous writings), so that the work is not merely an exposition but truly a guide.

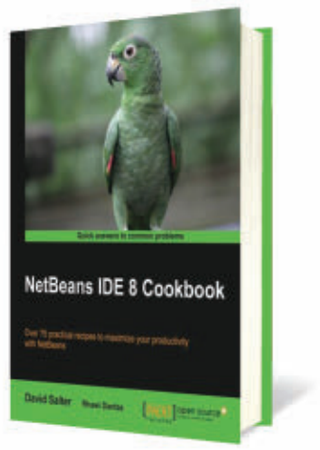
There is a fair amount of duplication between the examples in this book and Horstmann's earlier one on Java SE 8, so if you bought that book, I'd recommend waiting for the arrival of *Core Java* before upgrading. However, for all other readers, this is an excellent tutorial for Java SE 8.
—Andrew Binstock

> ***Over the next few issues,*** *the book section of* Java Magazine *will evolve from descriptive blurbs to critical reviews, signed by the respective authors. Due to potential conflicts of interest, books from Oracle Press will be presented but not reviewed. —Ed.*



THE CUCUMBER FOR JAVA BOOK: BEHAVIOUR-DRIVEN DEVELOPMENT FOR TESTERS AND DEVELOPERS
By Seb Rose, Matt Wynne, and Aslak Hellesoy
Pragmatic Bookshelf, February 2015

Teams working on the Java Virtual Machine (JVM) can now say goodbye forever to misunderstood requirements, tedious manual acceptance tests, and out-of-date documentation. Cucumber—the popular open source tool that helps teams communicate more effectively with their customers—now has a Java version, and the best-selling *Cucumber Book* has been updated to match. It has the same good advice as the earlier book about how to deliver rock-solid applications collaboratively, but with all code completely rewritten in Java. New chapters cover features unique to the Java version of Cucumber, and reflect insights gleaned by the Cucumber team since the original book was published.



NETBEANS IDE 8 COOKBOOK
By David Salter and Rhawi Dantas
Packt Publishing, October 2014

This book will show you how to perform many key tasks with the NetBeans IDE, uncovering more about mobile, desktop, and enterprise Java along the way. You will start by creating Java projects and learning how to refactor and use NetBeans tools to increase developer efficiency. You then get a walkthrough of how to create a desktop application, and then JavaFX, mobile applications, and how to use external services. Having seen how to create many different types of applications, you are then shown how to test and profile them before storing them in revision control systems such as Git or Subversion. Finally, you will learn how to extend NetBeans itself by adding new features to the IDE.

JAVA EE 7: THE BIG PICTURE
By Danny Coward
Oracle Press, October 2014

Java EE 7: The Big Picture explores the entire Java EE 7 platform in an all-encompassing style while examining each tier of the platform in enough detail so that you can select the right technologies for specific project needs. In this authoritative guide, Java expert Danny Coward walks you through the code, applications, and frameworks that power the platform. Take full advantage of the new capabilities of Java EE 7, increase your productivity, and meet enterprise demands with help from this Oracle Press resource.

```
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());
      }
```

```
List<Integer> transactionsIds =
      transactions.stream()
      .filter(t -> t.getType() == Transaction.GROCERY)
      .sorted(comparing(Transaction::getValue).reversed())
      .map(Transaction::getId)
      .collect(toList());

// as a parallel stream:

List<Integer> transactionsIds =
      transactions.parallelStream()
      .filter(t -> t.getType() == Transaction.GROCERY)
      .sorted(comparing(Transaction::getValue).reversed())
      .map(Transaction::getId)
      .collect(toList());

List<Integer> transactionsIds =
      transactions.stream()
      .filter(t -> t.getType() == Transaction.GROCERY)
```


//20 years of java//

{THE MASTERS SPEAK}

The code masters behind Java's success discuss what has surprised them most over the years and what innovations are coming in the years ahead.

In the history of language development, "crossing the chasm" refers to the stage at which a language breaks out of a niche in which it was used only by aficionados and hobbyists and begins its journey into the programming mainstream.

Languages that cross the chasm become widely used. While they are assured of a prolonged period of popularity, they are not guaranteed good health and long life. Perl, for example, was *the* language 15 years ago, but has seen its popularity decline rapidly during the last decade—largely under pressure from Python.

Java, however, crossed the chasm and continued to thrive in the mainstream. It has been robustly healthy for two decades. In fact, by almost all measures, its popularity continues to grow. I attribute this success to the substantial investment Sun, Oracle, and various partners made in two key areas: advancing the language and optimizing the Java Virtual Machine (JVM).

With this work in mind, we sat down with John Rose, JVM architect at Oracle, and Mark Reinhold, chief architect of the Java Platform Group at Oracle, and asked them to tell us what has most surprised them in their work on Java during these last 20 years and what lies immediately ahead for the language. —*Ed.*

JRebel

ZEROTURNAROUND

RELOAD CODE CHANGES INSTANTLY

TRY IT NOW!

Get a free t-shirt!

A CONVERSATION WITH JOHN ROSE

With Projects Valhalla and Panama, the JVM continues to add useful features. **BY TIMOTHY BENEKE**

Any celebration of Java would be incomplete without a discussion of the remarkable versatility of the JVM, which now hosts numerous languages such as Clojure, Groovy, JRuby, Jython, Kotlin, and Scala. JVM Architect John Rose, whose history with Java dates back to his days at Sun Microsystems in 1995, contributed to the design of Java inner classes, the initial port of Java HotSpot VM to SPARC, and the Unsafe API. His work enhanced the performance of the HotSpot and OpenJDK stack, affecting everything from hardware architecture to code generators, libraries, and programming languages. He is currently the lead engineer of the Da Vinci Machine Project, a part of the OpenJDK effort, and the JSR 292 Specification Lead, responsible for specifying new support in the JVM standard for dynamic invocation and related facilities, such as type profiling and improved compiler optimizations. We spoke with him to get his take on the past and future of the JVM.

Rose's first project on the JVM was designing and implementing inner classes for Java 1.1.

PHOTOGRAPHY BY BOB ADLER/GETTY IMAGES

Java Magazine: Tell us a little about your history with Java and the JVM.

Rose: I sometimes refer to myself as a "Lisp refugee" because I spent some of my early years happily programming in Lisp, which is a managed language with garbage collection and safe dynamicity, with safe typing in a rich library. Like many language geeks, I invented my own Lisp VM, which I used successfully at Sun on a couple of projects.

Then Java exploded on the scene in the mid-1990s and I discarded my own VM and joined the Java project as one of the engineers who onboarded the HotSpot performance engine. I remember Bill Joy calling me "fresh meat" when I joined the project and saying, "We need someone to work on the Java language." James Gosling had a Java problem—how to do method pointers and delegation—and asked for my help. So I designed and implemented inner classes for Java 1.1. So I got onboard fairly early.

Because I've mentioned two of the main designers of Java who are not currently on the Java team, I should also mention Guy Steele, who is at Oracle Labs and has been instrumental and is even now helping us think through the value type proposals for the JVM, which are part of Project Valhalla. So the influence of the original designers of Java is still being felt.

Java Magazine: Do you still do any coding yourself?

Rose: I still do code, even in C++. At Oracle I've been primarily concerned with the Java HotSpot VM and especially with its just-in-time [JIT] compiler and, of course, with the implementation. I was the main implementer of JSR 292, the `invokedynamic` construction. I wish I could say that I code a lot. I mostly content myself with doing small prototypes to prove that things are possible, and I review other people's core code. I have fun hammering out architectural documents with some really brilliant people in the organization. We have a good community, not only of coders whom I interact with all the time, but also hardware and software designers, theorists, and other folks who are trying to figure out the next great thing for Java. I've had two really good coding sessions in the last couple of months. One was a proof of concept in Java for a way of doing value types, and another involved solving a C++ puzzle that should let us structure our Java HotSpot VM code better.

There's a good prospect that Java will naturally grow to assume more tasks that are currently being done with C++. When I'm frustrated with using C++, it's usually because I'm building some part of the JVM that needs to be done right. When people on the team commiserate with each other about C++ and how come we have to do C++ when everybody around us is using Java on the VM, we are fond of saying that we do C++ coding *so that* millions of other people won't have to.

Java Magazine: Give us some historical perspective on the JVM.

Rose: The original authors of Java understood that they could not aggressively solve problems all at once, so they addressed them in a layered manner. They not only invented Java but also the virtual machine and, in particular, the bytecode and object architecture that the Java compiler compiles to. We think of these things as bytecodes for Java, but there's a lot more to them—they have their own identity and their own special powers, so we can think of bytecodes as a very powerful and tastefully designed execution model in its own right.

The earliest versions of the JVM manual made it clear that the bytecodes were not just replicating Java but could actually support other languages. And whether people took the hint from the architecture manual or because they perceived that the bytecodes were very inviting for such purposes, from the beginning, developers began writing other languages for the JVM. Just a few years after Java began, there were literally hundreds of known language implementations running on top of the JVM. There were versions of Scheme, like Kawa, that came out very early, and even today developers are continually inventing new languages to run on the Java bytecodes. It's extremely exciting.

JAVA VS. C++

"People on the team commiserate with each other about C++: 'How come we have to do C++ when everybody around us is using Java on the VM?'"

Java Magazine: How has the JVM developed and changed over the years?

Rose: Around 2005, Sun Microsystems got serious about adding a new invocation mode in the JVM to support dynamic languages. We realized that our invocation modes were all organized toward static typing and Java in particular, and we wanted to add a new invocation mode or methods that would more directly support dynamic languages and semantics. So before we had any details, we called it the "invokedynamic" instruction.

But it took a good five years to implement, because we had a lot of ideas about what dynamic invocation might look like and the design went in various directions. Our first prototype took some wrong turns, as the experts in the JSR 292 Expert Group [*in the Java Community Process —Ed.*] pointed out. So it was redesigned and came out in JDK 7 and has been an important part of the stack ever since then. `invokedynamic` is a completely general invocation Swiss army knife that language implementers can use to implement not only dynamic method invocation but also some kinds of static invocations that were not possible in the previous instructions. So it hit a sweet spot. It gives you full access to the JVM's capabilities and full flexibility to the language implementer who's working on top of the JVM. To make a proof of concept, we had early involvement with Charlie Nutter, the lead on JRuby, who accelerated several prototype implementations of JRuby using `invokedynamic`. So we had dynamic language people involved extremely early.

In recent years, Oracle has created a brand-new JavaScript implementation called Nashorn [*part of JDK 8 —Ed.*], which is a high-performing JavaScript engine that runs on top of the JVM, and was designed from the start to build on top of `invokedynamic`.

I want to make it clear that when you add a change to the bytecode architecture—and this holds for the future—it can take years to think around and finally come to a good design. And once we're committed to an implementation, it can take more years to establish the optimization dynamics of the design. That's one reason we go very slow in JVM evolution. As James Gosling said recently at JavaOne, "We don't want to do things until we know we can do them right—in Java and on the JVM."

Java Magazine: Where is the JVM headed?
Rose: Oracle and the Java community are making serious investments in two new OpenJDK projects, Valhalla and Panama. Project Valhalla is ultimately aimed at closing the divergence between references and primitives so that users can define new primitive-like types, which we call "value types." Java doesn't have a complex number or an unsigned int type. It doesn't have a 128-bit integer type, but with value types we will be able to create all of these as library code, not as VM intrinsics. That's a big vision. A basic physical reason why we need this is because value types are natural units for flatter, more compact data structures. You can look at our current Java architecture and say, "Suppose you don't have value types, and suppose you're not allowed to make new primitives. Surely you can use classes to make 128-bit integers and unsigned ints and complex numbers." This is something people do. But the problem is that it's tied up with this object-oriented data model that is native to Java, which puts pointers everywhere. And it turns out that once you make an object that's reachable by a pointer, it's hard to get rid of the pointer. There's a class of effects called "identity semantics" that makes it difficult or impossible for compilers to ignore the pointer. If this were not a problem, Java wouldn't need primitive types with everything being an object. With value types, the same performance feel that you get from an int, you'll get from a complex number. The same things that make you hesitate before using a `Java.lang.Integer` on your data also push you away from using a Java map complex class. But once the complex becomes a value type instead of an object type, it loses its indirection—it gains flat data representation, and the performance profile feels like an int.

JVM LANGUAGES
"Clojure is a mad-scientist library of how to rethink concurrent programming with persistent data structures in the setting of an idiosyncratic Lisp-like language."

In fact, that's our slogan for value types: "Codes like a class, works like an int!" You define your complex in a library construct that looks like a class, but when you use it, it feels like an int.

The addition of value types also makes it easier for other languages to migrate to the JVM. It no longer forces languages that have other datatypes to conform them to the JVM. They can create datatypes to reflect their natural datatypes. They can do that today, but at the cost of a pointer and a storage allocation.

Java Magazine: What is Project Panama?
Rose: With Project Panama, we are creating a more robust interconnection with native code. Developers like Java because it's a friendly language to write in. But sometimes they have to program in C or C++ because, for example, there's a low-level system library with the C API. So they have to pull out their C editor and include some header files and start coding up in C. To weave that into their Java project, they have to write a bunch of Java Native Interface [JNI] code. But JNI is intentionally difficult to use. In its early days, Java was fighting to differentiate itself and needed to maintain a certain amount of integrity and control over its own execution. So the JVM was pretty unwilling to share trust with native libraries. If you're going to attach a native library, you might have to sign up for months of engineering work to create the necessary JNI bindings.

So to enter the JVM world from the C world, you had to either leave your C behind or bring it in at great cost. It's like an import duty. Java now has a stable place in the computing world, so it would be helpful to Java to have a more tax-free entry from the C world to the Java world—that's Project Panama.

Here's a picture: If Java is the Pacific Ocean and C is the Atlantic Ocean, Panama offers an easy bridge or canal to connect the two great bodies. I posted a manifesto on my blog a few months ago called "The Isthmus in the VM" that lays out the risks and rewards of connecting Java and C. I'm happy to say that Project Panama is making good progress. So JVM users should one day enjoy not only a wide variety of

languages with more and more powerful datatypes, but also the ability to resort to C libraries with much less overhead.

Java Magazine: What's happening with bytecode transformers?

Rose: Let me first provide some background. The idea of having a flexible virtual machine that runs at high speeds is fairly old. Pascal achieved some portability using p-code and if you look at the early Java papers, they acknowledge a debt to previously existing bytecode and virtual-machine instructions. One reason that Java's bytecode design has been an extremely successful VM instruction set is that it hides enough detail about the underlying machine, so that it can be easily moved and re-executed from machine to machine. It also has security properties that were not present in previous bytecodes. In other words, the information-hiding aspect of the bytecodes implies a secure ability. It means that your bytecodes can, in certain situations, be proven by a verifier to not subvert the type system. We don't allow unsigned or possibly hostile code to run on the JVM, because the bytecode is hardened against such attacks. By the way, we take very seriously any reports of loopholes or cracks in the armor, and we patch them up as vigorously as we can. It's one of our priorities to keep the attack surface as clean as possible.

Our bytecode set is not just an executable format but is also a clear expression of the abstract structure of a program. Users can perform a neat trick and transform bytecodes. So, people have written all sorts of really ingenious bytecode transformers, which do not execute the bytecodes but modify them incrementally to make them more useful from their perspective. So, bytecode-to-bytecode transforms are practical and interesting in part because of the good vision of the original designers of the VM to not be too closely bound either to hardware or to a language.

You will see all sorts of interesting annotation-driven transformers in the market today, because the bytecodes are not only a good execution format but they're a good representation of a program that can be manipulated in its own right.

IN THE JVM

"Bytecodes are not only a good execution format but they're a good representation of a program that can be manipulated in its own right."

Java Magazine: Tell us about the role of the JVM Language Summit.

Rose: When we started doing the JVM Language Summit in 2008, we realized that there was a substantial community doing very interesting things just on the bytecodes. So it was a fun idea to have them all gather together. They talk in a very specialized kind of technical language—these are people who speak bytecode and are interested in what the bytecodes could do for them. They are knowledgeable about compilation techniques that would take a high-level language into bytecodes. We've met once a year since 2008, and it's usually the best conference that I go to each year because the attendees care passionately, not only about Java but also about efficiently implementing languages on top of Java bytecodes. As a result, we've been able to slowly evolve the virtual machine so that it will serve more and more languages.

Of course, now we have Clojure, Groovy, Jython, JRuby, Scala, and other established languages. Scala is a wonderful mad-scientist library of excellent language ideas. And Clojure is a mad-scientist library of how to rethink concurrent programming with persistent data structures in the setting of an idiosyncratic Lisp-like language that some people find extremely compelling when it fits their problem space. `</article>`

Timothy Beneke is a freelance author whose work has previously appeared in this magazine. He has written extensively about Java.

LEARN MORE

- John Rose's blog

USING JAVA

"Despite being internally complex, Java has a feeling of initial simplicity. It has a high 'DWIM' factor—'Do What I Mean.'"

A CONVERSATION WITH MARK REINHOLD

The changes that have driven Java's wide use, and the changes to come **BY TIMOTHY BENEKE**

Mark Reinhold, chief architect of the Java Platform Group at Oracle, has made many contributions to the Java platform in the past 19 years, working first at Sun Microsystems and then at Oracle: character-stream readers and writers, reference objects, shutdown hooks, the NIO high-performance I/O APIs, library generification, and service loaders. He was the lead engineer for the Java 1.2 and 5.0 releases, the Specification Lead for Java SE 6, and both the project and Specification Lead for JDK 7 (Java SE 7) and JDK 8 (Java SE 8). He currently leads the JDK 9 and Jigsaw projects in the OpenJDK community, where he also serves on the governing board.

Reinhold, who calls himself an old Lisp hacker in disguise, holds a PhD in computer science from the Massachusetts Institute of Technology, where he worked on garbage collection, compilation techniques, type systems, semantics, and the visualization and analysis of program performance.

PHOTOGRAPHY BY BOB ADLER/GETTY IMAGES

Java Magazine: What has surprised you most about Java over the years?
Reinhold: I'd have to say it's the strength of its foundation. I don't think anybody understood at the beginning what a strong foundation James Gosling and the rest of the original Java team built into the Java Virtual Machine [JVM] and Java language specifications. They're clear but evolvable designs that are not so self-contained and interdependent that you wouldn't dare change them.

If you look at the VM specification, for example, it's very clear about the structure of the VM, how class files are defined, what all the bytecodes are, and what they mean. Yet there's room to grow because there are plenty of unused bytecodes and the conceptual framework in which they exist has room for flexibility. The language specification has the same characteristics.

I've also been surprised at the number of times people have declared Java to be dead, which has not yet proved to be true.

Java Magazine: You have spoken about the importance of retaining what James Gosling calls the "feel of Java," which includes readability, simplicity, and universality.
Reinhold: Those properties are really important. We recognized their value early on, and we've worked very hard to preserve them. Readability is the most critical because, with software, the amount of time you spend writing a piece of code that goes into production is microscopic compared with the time that you and your successors will spend reading it in order to maintain it, fix bugs in it, and enhance it.

CLEAR CODE
"Readability is the most critical because, with software, the amount of time you spend writing a piece of code that goes into production is microscopic compared with the time that you and your successors will spend reading it."

Simplicity refers to the comprehensibility of the platform. Any production-quality language and surrounding platform must have a certain amount of internal complexity in order to address the inherent complexities that we find in modern computing systems. But despite being internally complex, Java has a feeling of initial simplicity. It has a high "DWIM" factor—"Do What I Mean." Once newcomers with a good IDE learn the basic plumbing, they can get around pretty easily. You don't have to go read the VM specification or memorize the language specification. Those references are available and you might eventually need them, but you can often just "write what you mean"—and it turns out to be what you meant a large fraction of the time. As a Java programmer, you can get into the zone and solve your problem—and the language and platform tend to get out of your way.

There are some people who disagree with this because they prefer other styles of languages, but for this style of language Java does that pretty well.

Java Magazine: Why have you devoted so much of your professional life to Java?
Reinhold: What attracted me to Java initially is what keeps me here still. As someone deeply interested in programming languages, the opportunity to work on one that's used every day by millions of developers is exceedingly rare. It's exciting and rewarding—there's a kind of visceral thrill that's hard to describe. I'm enjoying it as much now as I did when I started nearly 20 years ago.

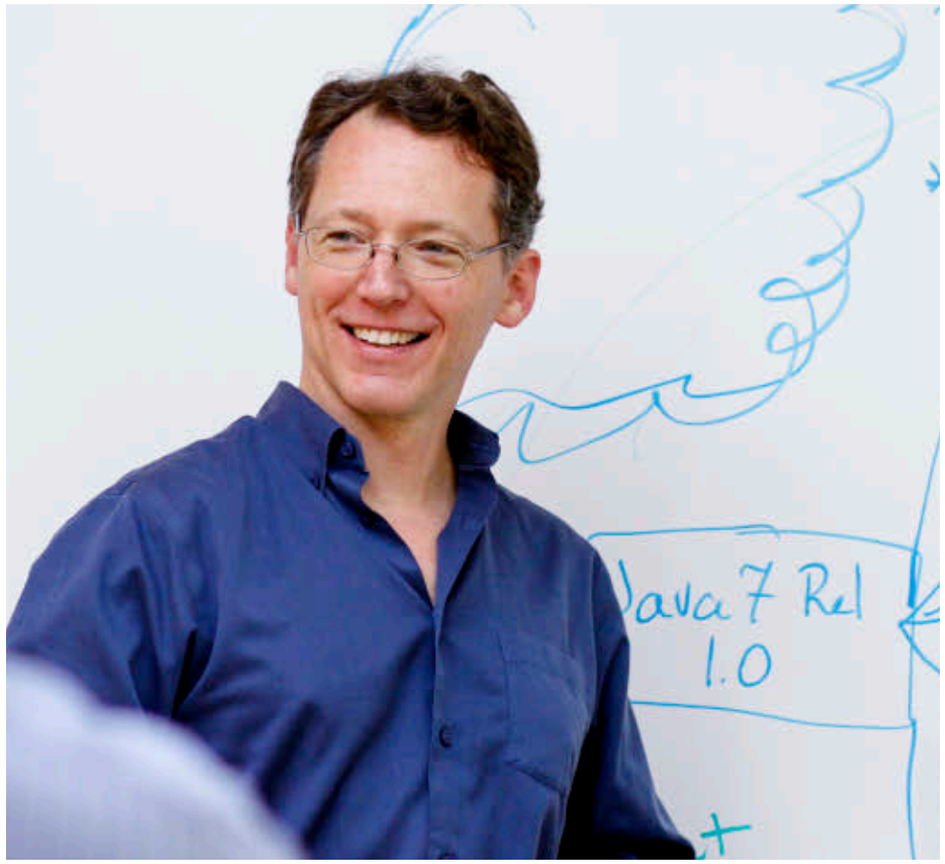
Java Magazine: Tell us a little about what you do all day. Do you still write code?
Reinhold: I still write code and I've had code in every release starting with JDK 1.1, although sometimes not a lot. I spend a fair amount of time mentoring junior engineers and advising people on various things. As one of the old-timers, I know a lot of obscure facts, so I've wound up being a steward of history of the platform. I also do a fair amount of design work. I'm the Specification Lead for the module system that we're working on for Java SE 9, and that involves coding and design and architecture and language changes.

Java Magazine: It has been more than five years since Oracle acquired Sun Microsystems. How do you feel about Oracle's stewarding of Java?
Reinhold: I feel pretty good about it. Once we got through the initial challenge of merging these two large organizations, Oracle astutely decided that keeping Java vibrant was the top priority. The investments and deliveries over the last five years demonstrate that Oracle is serious about that commitment. There was a five-year hiatus when Sun basically failed to deliver Java SE 7, and then Oracle picked up the pieces and delivered it. Oracle then funded the development of Java SE 8 with the very successful lambda and streams features. Oracle is continuing to invest in the future: We've got Java SE 9 coming with modules, and we're already investing in some deep language features for releases beyond Java SE 9.

Java Magazine: You remarked that Java "evolves by taking on the next big pain point." Could you give us some examples?
Reinhold: Our basic recipe for evolving Java is to identify a pain point, figure out which abstractions are missing from the platform, and then add them. We make sure they fit in with everything that's already there so that the new features feel like they were there from the beginning.

One example is generics, introduced in Java SE 5 in 2004. Prior to that release, any complex data structure was, essentially, dynamically typed. You could put a bunch of objects of some type T into a list, for example, but when you got them back out they'd just be raw objects rather than objects of type T. So you'd have to insert cast operations wherever you did that, and that was error-prone and difficult to scale. The generics feature addressed this pain point by introducing the ability to abstract over types, also known as *parametric polymorphism*. You can now declare that a list is a list of objects of a particular type, and you no longer have to insert cast operations when accessing its elements.

The next big change to the language was lambda expressions, which were our response to another specific pain point: It was too awkward to express computations that needed to



pass code around as data. Lambdas introduced a better way of abstracting over behavior and also introduced the Stream API, which makes it much easier to take advantage of multicore processors.

Java Magazine: What pain point will Java SE 9 address?
Reinhold: There really are two pain points, which turn out to share a common solution. One pain point is that the Java platform is a huge, monolithic thing that can't be made any smaller. The other pain point is that large applications today are fairly brittle—that is, they're constructed by throwing tens or hundreds or even thousands of JAR files onto the classpath, which is prone to all kinds of difficult-to-diagnose pathologies.

The missing abstraction here is modules—that is, a way to gather multiple packages of Java code and ancillary data, such as native code and resource files, and treat them as one unit

that declares explicit dependencies upon other modules and controls exactly which packages are exported for use by other modules. Once we have a module system, we can apply that to the platform itself, dividing the platform into a set of modules so you can pick and choose the ones you need for your application. Applications themselves, rather than using the classpath, can migrate to modular form so that they can be reliably and safely configured.

Looking beyond Java SE 9, we're working on value types and specialization. There, the pain point is the performance of applications that need very large amounts of data, especially data that would normally be held in very small objects. When you have billions of small objects you waste a lot of memory space and also a lot of time accessing them, due to all the cache misses. The missing abstraction here is value types, which are a lighter-weight way to aggregate data values that does not require instantiation in the heap so that the JVM can, for example, pass them around in registers and lay them out efficiently in arrays.

JAVA SE 9

"Once we have a module system, we can divide the platform into a set of modules so you can pick and choose the ones you need."

Java Magazine: Could you describe how your work on Java differs from the standard bread-and-butter programming that most developers engage in?

Reinhold: If you're a developer and your job is to deliver a working system that has to meet some requirements and run within one organization, then, well, life is actually pretty straightforward. You deliver code that meets the requirements, it's put into production, and you move on to the next thing. That's not to say the work itself is easy, just that it's not hard to understand your ultimate goal.

Working on a widely used platform is very different—for example, there isn't just one customer. When we design a new feature, especially one that's far-reaching such as generics, or lambdas, or modules, we have to think about all the different kinds of developers who are going to use it. What is it we're asking them to learn? What is it that they can learn later, when they get around to it? What's the best way to structure it so that it's something they can learn and remember in a way that's consistent with what they've learned and remembered about Java so far?

We also have to consider the uptake of new versions of the platform. People often ask, "Why didn't you just add the module system in an update release of Java SE 8?" Well, we couldn't, for many reasons. One is that Oracle is not the sole implementer of the Java platform. IBM, Red Hat, and SAP, among many others, also implement it, and while many of those implementations share some code, they can still be very different. Qualifying an implementation takes a lot of time and is not something that an implementer can do at the drop of a hat.

Another issue is uptake in enterprises. People running large websites, for example, are not keen to update the underlying Java platform in a big way very frequently, because that, too, takes a lot of time and effort.

Java Magazine: Finally, do you have any advice for younger developers?

Reinhold: Yes—learn how to write clear prose in English, or whatever your native language happens to be. If you can write clearly, then you can think clearly—and that's more important than learning any specific programming language. `</article>`

Timothy Beneke is a freelance author whose work has previously appeared in this magazine. He has written extensively about Java.

LEARN MORE

- Mark Reinhold's blog

What's New in JPA

Attribute conversion, schema generation, enhanced SQL queries, and a host of other improvements highlight JPA 2.1.

JOSH JUNEAU

BIO

The Java Persistence API (JPA) is a fundamental component for relational database–driven Java EE applications. Although JPA 2.1 was a minor release, some of the new features that were added pack a punch, and they can vastly improve overall developer productivity.

This article covers some of the latest features in JPA 2.1, which was introduced with Java EE 7. Throughout the article, we will delve into the new features of JPA 2.1, examining examples of each in a real-world application, so that you can begin applying these new features to your Java EE applications today.

The AcmePools application we will use for the examples in this article was developed using Java EE 7 with JavaServer Faces (JSF) for the front end and using Enterprise JavaBeans (EJB) beans for working with data. The application was originally built in an article entitled "PrimeFaces in the Enterprise," which was published on Oracle Technology Network.

Database Schema and Type Mapping

Developers face several challenges when modeling database tables with Java objects. JPA 2.1 brings various enhancements in database schema and type mapping, helping to ease the transition between a relational database and Java objects. This section covers two of the new features that help in this area: attribute conversion and schema generation.

Attribute conversion. In order to represent database tables as objects, you must map each database type to a Java type, aka *type mapping*. For some cases, JPA does not automatically convert from a Java type to a database type or vice versa. For instance, perhaps JPA contains a specified mapping, and an application requires something different. JPA 2.1 allows you to write an attribute converter in such cases, to specify which database type should correspond to a specified Java type.

Similar to a custom Hibernate type, attribute conversion can be applied to a basic attribute or to an element of a collection type in order to convert between a database attribute (column) type and a Java object type. For instance, in the example application, we would rather see a String value of `N` or `Y` in the database than a `0` or `1` for the representation of a Java Boolean value. We can code a converter into the application to convert a Java `false` value to an `N` and a Java `true` value to a `Y`, and then store the values in `VARCHAR` fields. To do so, you must create a converter class, which implements the `javax.persistence.AttributeConverter` interface. This interface accepts an `X` and a `Y` value, with `X` being the Java type and `Y` being the corresponding database type.

A type converter should not be applied to `Id` attributes, version attributes, relationship attributes, or attributes denoted as `Enumerated` or `Temporal`, because doing so makes an application nonportable.

SCHEMA

Automatic schema generation is achieved in JPA 2.1 via a combination of annotations and by specifying properties.

Listing 1 demonstrates a converter for Boolean-to-String conversion.

A converter can be applied automatically, or it can be applied using the `@Convert` annotation, specifying the class name of the converter. In the example application, the `@Convert` annotation is specified on the entity class attribute that should use the converter when being persisted or retrieved, as seen in **Listing 2**.

Attribute conversion makes it possible to easily configure a custom mapping between a database field type and a Java type without muddying up entity classes with conversion logic.

Schema generation. In the past, developers relied upon manual generation of entity classes for a given database, or they utilized a nonstandard method of autogeneration, such as an IDE or provider-specific implementation. Manual generation of entity classes can be cumbersome and error-prone, and the use of an IDE or provider-specific implementation can cause vendor lock-in. These issues have been resolved in JPA 2.1 via the introduction of automatic schema generation. Automatic schema generation is achieved in JPA 2.1 via a combination of annotations and also by specifying properties within the persistence context of an application.

To accommodate schema generation, properties can be added to the `persistence.xml` file to designate how to perform the schema generation. The properties shown in **Listing 3** are some of the most often used, which can be specified to generate a database schema. For a complete listing of the different properties, see the Java EE 7 tutorial.

Listing 4 demonstrates an example of a `persistence.xml` file containing JPA schema-generation properties. The properties that designate a script source point to SQL scripts that contain the SQL code that is required to create, load, or delete the database objects. There is also an API for programmatically generating a database schema. Oftentimes, these scripts are placed within the application `META-INF` folder.

Another benefit that JPA schema generation has over most IDE- and provider-specific implementations is that it provides support for indexes and foreign keys via the `@Index` and `@ForeignKey` annotations, respectively. To designate an index on a database attribute, specify the indexes attribute of the `@Table` annotation, and then list each index within the indexes attribute using `@Index`.

JPA 2.1

An annotation was introduced with the release of JPA 2.1, providing a more convenient way to invoke procedures stored in a database.

LISTING 1 | LISTING 2 | LISTING 3 | LISTING 4 | LISTING 5 | LISTING 6

```
@Converter
public class BooleanToCharacterConverter implements
    AttributeConverter<Boolean, String> {

    @Override
    public String convertToDatabaseColumn(Boolean x) {

        if(x){
            return "Y";
        } else {
            return "N";
        }
    }

    @Override
    public Boolean convertToEntityAttribute(String y) {
        String val = (String) y;
        if(val.equals("Y")){
            return true;
        } else {
            return false;
        }
    }

}
```

Download all listings in this issue as text

Listing 5 shows how to specify an index on the `Customer` entity. The `@ForeignKey` annotation is placed within the `@JoinColumn` annotation, which is specified on an entity attribute (see **Listing 6**). Note that if it is specified, the `@ForeignKey` annotation overrides the defaults provided by the persistence provider.

Invocation of Database Constructs

JPA 2.1 brings new techniques to invoke database constructs, enabling both passing parameters and retrieving values. Now it is easier and less error-prone to invoke database procedures and functions. This section covers a couple of these new features.

Named stored procedures. In some instances, applications need to call into the database and invoke stored procedures. In the past, developers relied upon native SQL calls to invoke stored procedures in a database, but the `@NamedStoredProcedureQuery` annotation was introduced with the release of JPA 2.1, providing a more convenient and standard technique for invoking procedures that are stored within a database. Using this annotation, you can map a stored procedure to an identifier and map the `resultSet` to one or more entity classes.

To demonstrate this functionality, consider invoking a stored procedure within the AcmePools database that retrieves pool information based upon a specified `Customer` ID. The stored procedure might look something like that shown in **Listing 7**. The stored procedure can be declared within an entity by specifying `@NamedStoredProcedure` and providing a `name`, `resultClass`, `procedureName`, and `parameters` within the annotation.

In the example shown in **Listing 8**, the `OBTAIN_CUSTOMER_POOL_INFORMATION` stored query is assigned to a name of `ObtainCustomerPoolInformation`, and it returns a `Pool` object. A list of `@StoredProcedureParameter` annotations can be specified to assign parameters or return values to the stored procedure.

In more-complex scenarios, a stored procedure might require more than one result mapping. In these cases, the `resultSetMappings` attribute can be specified within the annotation, passing a list of `@SqlResultSetMapping` names.

As seen in **Listing 9**, to invoke the stored procedure that is declared via `@NamedStoredProcedureQuery`, call upon the Entity Manager's `createNamedStoredProcedureQuery()`, passing the assigned name of the stored procedure. Parameters can be passed to the resulting `StoredProcedureQuery` object by calling `setParameter()` and passing a position/value pair. Results can be returned by calling the `StoredProcedureQuery` `getOutputParameterValue()` method. If you wish to manually register parameters, or perhaps do it in a more dynamic manner, the `registerStoredProcedureParameter(position, type, ParameterMode)` method can be used.

Stored procedure query via Entity Manager. Stored

LISTING 7 LISTING 8 / LISTING 9

```
public static void obtainCustomerPoolInformation(
        long customerId,
        String[] poolInformation) throws SQLException {
    Connection conn = null;
    PreparedStatement stmt = null;
    ResultSet res = null;

    try {
        // "jdbc:default:connection" tells the
        // DriverManager to use the existing connection.
        conn = DriverManager
           .getConnection("jdbc:default:connection");
        // prepare the query
        String sql = "SELECT STYLE, SHAPE, LENGTH, " +
                "WIDTH, RADIUS, GALLONS " +
                "FROM POOL P, CUSTOMER CUST " +
                "WHERE P.ID = CUST.POOL_ID " +
                "AND CUST.CUSTOMER_ID = ?";
        stmt = conn.prepareStatement(sql);
        stmt.setLong(1, customerId);
        res = stmt.executeQuery();

        poolInformation[0] =
          (res.next()) ? res.getString(1) : "N/A";
        . . .

    } finally {
        // Perform cleanup
      . . .
    }
}
```

Download all listings in this issue as text

procedures can also be mapped via the Entity Manager's `createStoredProcedureQuery()` method, passing the name of the database-stored procedure. You can then register parameters or assign return values using the `setParameter()` methods in the same manner as seen in **Listing 9**.

To demonstrate, **Listing 10** utilizes the same stored procedure that was used in the previous example. Only this time, there is no requirement to annotate any entity class with `@NamedStoredProcedureQuery`. Instead, simply initiate the stored procedure call using the Entity Manager's `createStoredProcedureQuery()` method, passing the name of the database-stored procedure.

SQL Queries, Updates, and Query Enhancements

Perhaps the area that received the most updates with JPA 2.1 was the SQL realm. Several improvements were added, facilitating more options than ever before for querying, updating, and removing data from an underlying datastore.

Constructor result mapping. Constructor result mapping allows you to map the results of a query to the constructor of a Java object utilizing the `@SqlResultSetMapping` annotation. As a result, constructor result mapping allows you to populate entities via the results of a query. To implement such a solution using constructor result mapping, the `@SqlResultSetMapping` annotation must specify a `name`, and it should contain the `classes` attribute with an assigned `@ConstructorResult`.

In essence, the `@ConstructorResult` annotation maps a target class to a series of columns that are noted via `@ColumnResult` annotations. The columns must be listed in the order in which they are listed as arguments for the target class constructor. **Listings 11a** and **11b** demonstrate how to construct an entity that facilitates constructor mapping, and **Listing 12** demonstrates how to populate the objects via a native query.

Dynamic named queries. In some circumstances, named queries need to be built dynamically, depending upon the specified application. Under these circumstances in pre-JPA 2.1 days, it was not possible to create a named query on the fly using dynamic tactics. This is now possible thanks to `EntityManagerFactory.addNamedQuery()`.

The `addNamedQuery(String, Query)` method accepts a String as the name that you wish to assign to the query, and it accepts a `Query` object itself, which can be generated dynamically. An example of generating a dynamic named query is shown in **Listing 13**.

LISTING 10 LISTING 11a LISTING 11b LISTING 12 LISTING 13

```
StoredProcedureQuery spq =
     em.createStoredProcedureQuery(
        "OBTAIN_CUSTOMER_POOL_INFORMATION"
     );
spq.registerStoredProcedureParameter(
      1, Integer.class, ParameterMode.IN);
spq.setParameter(1, customerId);
return spq.getResultList();
```

Download all listings in this issue as text

JPA Query Language enhancements. There are several improvements in the JPA Query Language (JPQL) area, providing the ability to

perform more-complex queries. For example, to achieve a finer degree of control over queries, the `ON` keyword can be used to query tables based upon an outer `JOIN` condition. An outer join allows you to extend the results that can be achieved with a standard join. Outer joins return all rows satisfying the join condition, along with additional rows from one table that do not satisfy the condition.

The syntax for performing an outer join using JPQL can be seen in **Listing 14**, which returns a count of customers that have a pool, along with a current maintenance contract. In the example, two entities are joined based upon their primary/foreign key relationship. Each `Pool` includes a collection of `Customer` objects, because there is a one-to-many relationship. In the example, the join is based `ON` the `currentMaintenance` field.

Another example is the `FUNCTION` keyword, which can be utilized within a JPQL query to call upon a database function, passing parameters if needed. This new addition to JPQL provides a concise and easy-to-use syntax for calling upon and returning the results of predefined or user-defined database functions.

To utilize the `FUNCTION` keyword, embed it into a SELECT query, passing two arguments: the function name and a comma-separated list of parameters, as follows:

```
FUNCTION('func_name',a1,a2)
```

Listing 15 demonstrates a method that utilizes a function to query the `Pool` entity for all entities that have a volume greater than 25,000 gallons.

A final example is the `TREAT` keyword. In the object-relational world, there are oftentimes classes that have subclasses to extend functionality. Therefore, it is possible for one entity to be extended by another, thereby producing a subclass of an entity. JPA 2.1 makes it possible to query an entity and retrieve all its attributes, along with any attributes of entities that are subclasses and explicitly called out in the query.

To provide this functionality, the `TREAT` keyword can be used to support downcasting within path expressions in the `FROM` or `WHERE` clauses of a query. The entity being extended should contain a discriminator column that is used for `SINGLE_TABLE` and `JOINED` mapping inheritance. The `Pool` entity in our example will be extended by other entities, so it contains the `@DiscriminatorColumn` annotation (see **Listing 16**). This annotation should specify the name and type of the discriminator column.

LISTING 14 LISTING 15 / LISTING 16 / LISTING 17

```
public List obtainAllCustomers(){
    TypedQuery<Object[]> qry =
         em.createQuery("SELECT c.name, count(p) "
            + "FROM Customer c LEFT JOIN c.pool p "
            + "ON c.currentMaintenance = true "
            + "GROUP BY c.name", Object[].class);
        List data = new ArrayList();
        if (!qry.getResultList().isEmpty()) {
            List<Object[]> tdata = qry.getResultList();
            for (Object[] t : tdata) {
                HashMap resultMap = new HashMap();
                resultMap.put("customerName", t[0]);
                resultMap.put("count", t[1]);
                data.add(resultMap);
            }
        }
        return data;
    }
```

Download all listings in this issue as text

In our example, the `Pool` entity is extended by two entities entitled `AboveGround` and `InGround`. Each subclass should contain the `@DiscriminatorValue` annotation, specifying the value that should be used for the discriminator column in the subclass. The `AboveGround` entity looks as follows:

```
@Entity
@DiscriminatorValue("ABOVE")
public class AboveGround
    extends Pool {
    public AboveGround(){
        ...
    }
}
```

Where the `TREAT` syntax comes in is when you wish to build a query that returns all `Pool` entities that are treated as `AboveGround` objects. **Listing 17** demonstrates its usage.

Criteria API bulk operations. The Criteria API has been enhanced, adding the ability to perform bulk updates and deletions. To make use of the Criteria API, you must obtain a `CriteriaBuilder` object from

Figure 1

the `EntityManager`, and then use the builder to generate objects, which can then be used to work with the specified entity classes. In JPA 2.1, the `CriteriaUpdate` and `CriteriaDelete` objects were added to the API, and you can now call upon the `CriteriaBuilder` to generate these objects for performing bulk update and delete operations.

The code in **Listing 18** contains a method named `updateMaintenanceByCity()`, and its implementation demonstrates performing a bulk update operation using the Criteria API. In the listing, all `Customer` instances that correspond to a specified city are updated, setting the current maintenance status accordingly.

Figure 1 shows the user interface for making the changes in the AcmePools application. When a user clicks one of the buttons, the `updateMaintenanceByCity()` method is invoked, updating all `Customer` instances that belong to the selected city.

A deletion can occur in much the same manner, utilizing a `CriteriaDelete` object rather than `CriteriaUpdate`. These enhancements to the Criteria API make it possible to develop an application entirely using the Criteria API without using any JPQL, if desired.

Entity graphs. In the past, an entity had to specify whether its relationships should be loaded using `EAGER` or `LAZY`, meaning an entity was statically bound to make all relations load using one technique or the other. There are many cases where an application is required to perform many similar queries against the same entity throughout its lifecycle, changing only a subset of the data that needs to be returned for each. Using `EAGER` loading could place an unnecessary tax on the database if relationships are unnecessarily loaded each time the entity is queried. Entity graphs seek to solve this issue by allowing you to have finer-grained control over which attributes are returned, based upon a specified entity graph name.

An entity graph is specified on an entity class using the `@NamedEntityGraph` annotation. The entity graph is given a name, followed by a list of attributes to be loaded, each designated with the `@NamedAttributeNode` annotation. The code in **Listing 19** demonstrates an entity graph on the `Customer` entity, which specifies that the relationship within the entity identified as `discountCode` should be loaded.

Once an entity graph has been assigned on an entity, it can be used by specifying a hint on an entity manager query and passing by name the entity graph that is to

LISTING 18 LISTING 19

```
public void updateMaintenanceByCity(String city,
      boolean enabled) {
   CriteriaBuilder builder = em.getCriteriaBuilder();
   CriteriaUpdate<Customer> customerUpdate =
            builder.createCriteriaUpdate(Customer.class);
   Root<Customer> c = customerUpdate.from(Customer.class);
   customerUpdate.set(c.get("currentMaintenance"), enabled)
            .where(builder.equal(c.get("city"), city));
   Query q = em.createQuery(customerUpdate);
   q.executeUpdate();
   em.flush();
}
```

Download all listings in this issue as text

be loaded. The hint can be of type `javax.persistence.fetchgraph` or `javax.persistence.loadgraph`. The `fetchgraph` type treats all specified attributes as `FetchType.EAGER`, and it treats unspecified attributes as `FetchType.LAZY`. The `loadgraph` type also treats all specified attributes as `FetchType.EAGER`, but it then reverts to the default specified fetch type of the other attributes. **Listing 20** demonstrates the use of the `customersWithDiscount` entity graph.

What happens if you wish to access a relationship that exists within the relationships that have been loaded via an entity graph? In such a case, it is possible to define subgraphs within an entity graph to load relationships within those that are loaded by an entity graph. Even though this might sound confusing, it is quite easy to achieve. Consider the case where a `DiscountCode` entity included a relationship with a `SpecialSale` entity. **Listing 21** demonstrates the syntax that is used to load the `SpecialSale` entities as a subgraph.

There is a lot more to know about entity graphs, such as how to use the programmatic API. Entity graphs are a very powerful addition to JPA once you get used to using them.

THERE'S MORE
Read the Java EE 7 tutorial for more details.

Unsynchronized persistence context. Applications might contain a series of forms that are related to each other, allowing a user to enter information into each form. An example is a shopping cart or an application wizard, for which a user needs to complete a series of tasks prior to the completion of a transaction. In such instances, it can be beneficial to wait until all the updates have been successfully made before flushing them to the database.

What happens if the power goes out while a user is in the midst of making a purchase online? It would be best if the entire transaction were either committed at once or rolled back, instead of being only partially completed. By default, when a database transaction has been made, the changes are automatically committed by the persistence context.

The new `UNSYNCHRONIZED` enumerated value allows you to indicate that a specified persistence context should be committed only after all transactions that belong to a specified conversation have been completed, which is indicated when the `EntityManager` instance's `joinToTransaction()` method is invoked. **Listing 22** demonstrates how to specify whether an `EntityManager` should be marked as `UNSYNCHRONIZED`. It also shows an example of completing the transaction.

LISTING 20 LISTING 21 / LISTING 22

```
public List<Customer> listCustomerDiscounts(){
  EntityGraph eg =
      em.getEntityGraph("customerWithDiscount");
  return em.createQuery("select o from Customer o")
        .setHint("javax.persistence.fetchgraph", eg)
        .getResultList();
}
```

Download all listings in this issue as text

Conclusion

The feature list described in this article does not cover the entire set of new features that were introduced with JPA 2.1. There are several others, including the ability to inject beans into entity listeners and implement `@PreDestroy` and `@PostConstruct` methods. Moreover, the features in this article were covered only briefly, and there is much more to learn about each. I recommend that you work with the examples, and read the Java EE 7 tutorial for more details.

Although JPA 2.1 is not considered a major release, there are several additions that help to solidify Java EE 7 as one of the most productive platforms available. Applying some of the techniques that were covered in this article will not only make you more productive as a developer, but will also allow you to achieve more functionality. `</article>`

LEARN MORE

- Java EE 7 tutorial
- AcmePools source code on GitHub

STEPHEN CHIN

BIO

Brewing Java with the Raspberry Pi

Interact with a USB scale to accurately measure your coffee beans and brew the perfect cup.

The Raspberry Pi comes preloaded with Java SE Embedded 8, so getting started with Java is as simple as typing the `java` command at the prompt. However, the ability to run Java on your Raspberry Pi wouldn't be complete without an application to help perfect your coffee-brewing skills. This article takes a scientific approach to coffee brewing in order to obtain the perfect "cup of Joe."

Communicating with a USB Scale

In order to precisely measure weight for coffee brewing, we are going to use a USB shipping scale. USB communication is based on a series of pipes (logical channels) that are connected to the device via endpoints. Furthermore, these endpoints are grouped into a set of interfaces. The protocol used by most shipping scales is simple one-way communication where the current scale value is repeatedly broadcast out over a single endpoint on the first interface.

To use the scale, simply plug it into one of the host ports on your Raspberry Pi and turn it on. It consumes only 16 mA, so it can be safely powered right off the Raspberry Pi USB bus.

The `vendorId` and `productId` show up when you plug in the scale and run the `dmesg` command. For the Dymo M10 scale they are 0x0922 and 0x8003, respectively. For its sister scale, the Dymo M25, they are 0x0922 and 0x8004, respectively. However, any compatible Stamps.com or DymoStamp scale should work fine as long as you change the `vendorId` and `productId` in the code.

For communication with the USB scale, we are going to use the `usb4java` library. This is an open source, JSR 80–compliant implementation of the standard JavaX-USB specification that has support for ARM Linux distributions such as the Raspberry Pi. You can download the latest version of `usb4java` from the project website.

USB DEVICE

The concept of stability is important, **because scales do not immediately register new weights,** so the value can fluctuate quite a bit before it settles down.

Make sure to download both the core library as well as the javax extension. From the core distribution you need to grab the following JAR files:

- `usb4java-1.2.0.jar`
- `libusb4java-1.2.0-linux-arm.jar`
- `commons-lang3-3.2.1.jar`

And you need these additional JAR files from the `usb4java-javax` distribution:

- `usb-api-1.0.2.jar`
- `usb4java-javax-1.2.0.jar`

Place all these files in a new folder called `lib` in your project directory, and add them to your project dependencies in your IDE of choice.

You also need to create a properties file in the root package that tells the JavaX-USB wrapper to use `usb4java` as the implementation. To accomplish this, create a file called `javax.usb.properties` in the root package (under `src`), and give it the contents shown in **Listing 1**.

To test the data returned from the scale, we are going to use a single-class implementation of the USB scale protocol that returns 60 data points and prints out the results on the command line.

This will be the basis for our coffee-brewing application.

The main method for our application is shown in **Listing 2** and can be summarized as follows:

- Find a connected USB scale.
- Open a connection to the scale.
- For 60 data points, submit a request to retrieve data.
- Close the scale.

To ensure the scale is closed properly even in the case of an error or exception, we are using a `try finally` block.

Listing 3 shows the implementation of the first method, `findScale`, which discovers Dymo M10 or M25 scales using the JavaX-USB API. This calls the `findDevice` method, which contains code that traverses the USB device tree (see **Listing 4**).

To read data from the scale, we need to connect to the correct interface and endpoint. Fortunately, the USB scale protocol is fairly simple, so you can simply grab the first interface and endpoint and then start listening to the data coming in from the scale directly, as shown in **Listing 5**.

Notice that we had to use the `claim` method that accepts a `UsbInterfacePolicy`. This allows us to force the kernel to detach from the USB interface so we can claim it for our application.

The implementation of `syncSubmit` is trivial, calling the same-named method on the `UsbPipe`:

```
private void syncSubmit()
   throws UsbException {
      pipe.syncSubmit(data);
}
```

However, the real work happens in the callback. To receive the callback, our class needs to implement the UsbPipeListener class, which has two required methods. The first is `dataEventOccurred`, which will be called as a result of invoking `syncSubmit` and contain the data returned from the scale. The second is `errorEventOccurred`, which will be invoked when there is a problem interfacing with the scale.

The data sent by these shipping scales is a simple byte array that contains six values. The protocol is not well documented, but it has been reverse-engineered by the open source community. The data is as follows:

- Byte 0—Unused
- Byte 1—Special flags: empty=2, overweight=6, negative=5 (The conditions indicated by overweight and negative are described later.)
- Byte 2—Unit of measure: grams=2, ounces=11
- Byte 3—Weight scale
- Byte 4—Base weight low-order byte
- Byte 5—Base weight high-order byte

Listing 6 shows the implementation of `dataEventOccurred`, which takes apart the byte array returned and prints out a human-readable scale value to the command line.

Besides doing a little bit of bit-shifting to get the weight magnitude, we also need to scale the weight by the `scalingFactor` returned in byte 3, as shown in **Listing 7**.

In the case of an error, the best we can do is log the error and continue (see **Listing 8**).

And the last method to finish the scale test is the `close` algorithm. This simply closes the pipe and interface cleanly so that the next application that tries to use the scale will be able to access it:

```
public void close() throws
    UsbException {
       pipe.close();
```

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

```
javax.usb.services = org.usb4java.javax.Services
```

Download all listings in this issue as text

```
        iface.release();
    }
```

When you run the completed `UsbScaleTest` class, it will compile and deploy the JAR file as before, but this time it will show some more-interesting output. If you have your scale connected and turned on, it should output the first 60 readings from the scale, including information about whether the scale is empty, overweight, or negative, as shown in **Listing 9**.

Try adding a few objects of different weights while the scale is returning data to get a feel for how the USB interface works. If you want to get a negative value, add an object to the scale, press the TARE button (to zero the scale), and then remove the object. To get an overweight reading, add an object that is heavier than the scale supports (for example, an 11-kilogram object on the Dymo M10 scale). Be careful not to use an object that is too heavy, because it can break the scale's sensitive internal components.

Now it's time to apply this to brew some perfectly proportioned coffee, and for that we need a coffee calculator.

Calculating Perfect Coffee

The science behind our application is that we will apply the Specialty Coffee Association of America (SCAA) Coffee Brewing Control Chart to select the optimum amount of ground coffee for our beverage. This chart is based on research in the 1960s led by Dr. Earnest Earl Lockhart, who is best known for his graphical representation of coffee brewing parameters along with a recommended guide for optimum strength and extraction. The North American version of this chart is shown in **Figure 1**.

There are three different units represented on this chart. The strength of the coffee is represented on the vertical axis as a percentage of the total dissolved solids (TDS), the extraction rate is represented on the horizontal axis as a percentage, and the brewing ratio is represented in grams per liter by the diagonal lines.

Our target density for a "regular" cup of coffee will be 55 grams per liter, which gives the highest chance of hitting the SCAA's optimum balance for a given extraction percentage.

There is also a Specialty Coffee Association of Europe (SCAE), which recommends the same 18 percent to 22 percent extraction rate, but moves the optimum strength up from 1.2 percent to 1.45 percent soluble concentration. This will define our "rich" cup of coffee at 58 grams per liter. Finally, the Norwegian Coffee Association further moves the optimum strength up from 1.3 percent to 1.5 percent soluble concentration, which will define our "strong" cup of coffee at 62 grams per liter.

LISTING 7 LISTING 8 / LISTING 9 / LISTING 10

```
private double scaleWeight(int weight, int scalingFactor) {
  return weight * Math.pow(10, scalingFactor);
}
```

Download all listings in this issue as text

Listing 10 shows a `CoffeeCalculator` class that takes these principles and turns them into code that will help us calculate the perfect brew.

Given a target liquid volume (cup size), the `CoffeeCalculator` class will help us determine how much ground coffee to use. We are assuming all values are in grams, because this is what most scales use and there is a straightforward conversion from volume to mass of water in the metric system.

A simple example of using this class would be to calculate how much ground coffee to use for a 300-gram cup of water:

```
grindWeight(REGULAR, 300)
```

This code returns 17.4 grams, which should sound about right if you have ever followed a coffee-

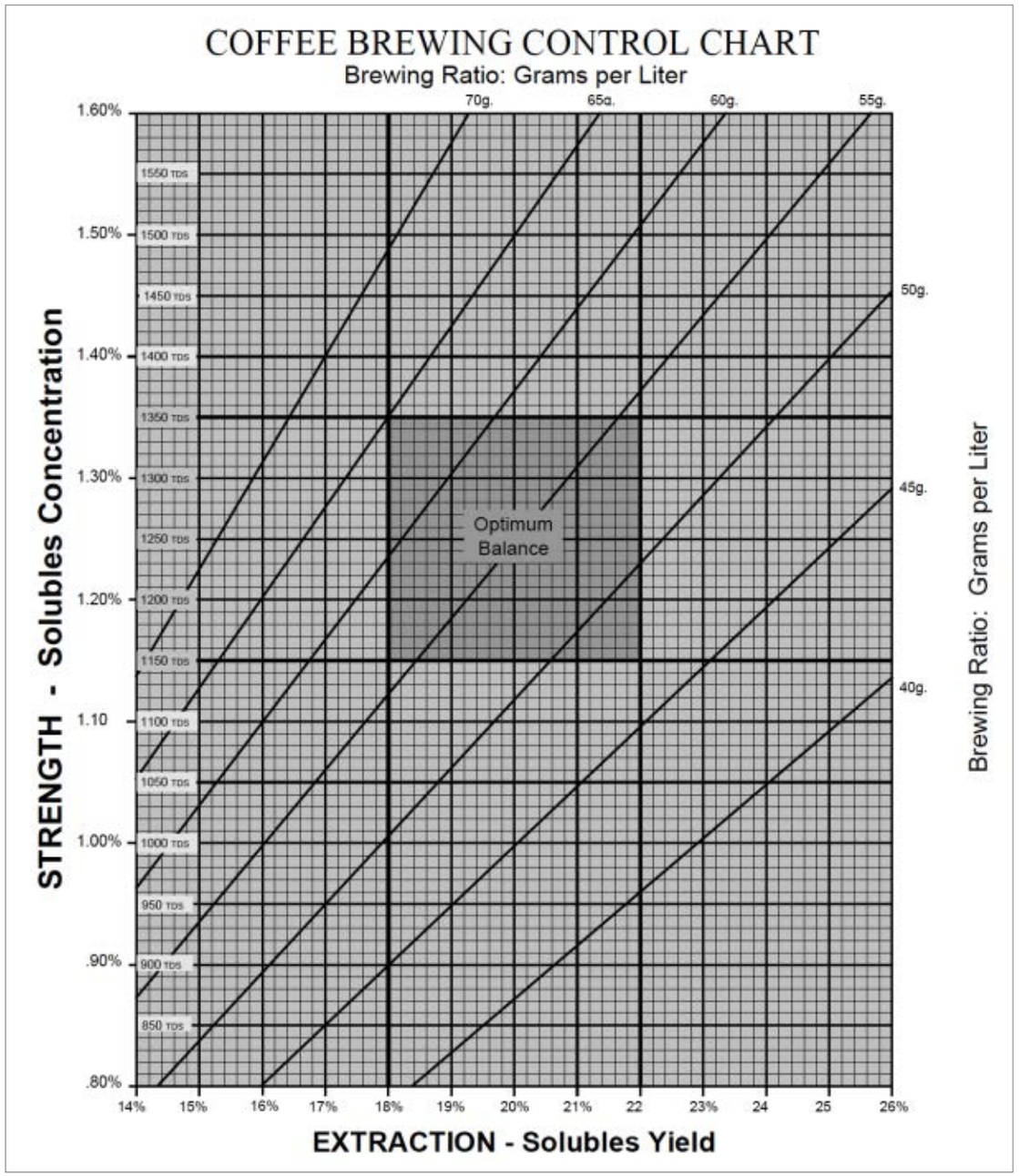


Figure 1

brewing recipe from one of the popular gourmet coffee roasters.

Asynchronous Communication

While we used a synchronous algorithm for reading the scale in our earlier example, this turns out to be a poor choice for our coffee algorithm. The reason is that USB scale data is buffered, so reading on demand returns stale data initially. If we wait until we need to know the current weight to take a reading, it is impossible to differentiate between the stale buffered data and the current scale weight unless we read and discard a fixed amount of data. By reading data continuously, we have an instantaneous snapshot of the current scale value in memory at all times, and we can simply block our thread until the next reading is available if we are waiting for a specific value.

The code in **Listing 11** implements a new `USBPipeListener` callback, which processes the returned data, submits a new request for data, and notifies any threads blocked on the `scalePhaser` that we have new data.

LISTING 11

```
public void dataEventOccurred(UsbPipeDataEvent upde) {
  processData();
  if (closing) {
    busy = false;
  } else {
    try {
      pipe.asyncSubmit(data);
    } catch (UsbException ex) {
      errorEventOccurred(
        new UsbPipeErrorEvent(upde.getUsbPipe(), ex));
    }
  }
  scalePhaser.arrive();
}
```

Download all listings in this issue as text

`Phaser` is one of the new thread control classes introduced in Java SE 7, and it is ideally suited for this purpose, because it allows the notification of arrival from a single thread to unblock potentially multiple threads. To create a `Phaser` for this purpose, simply construct one with a single party like this:

```
private final Phaser scale-
Phaser =
    new Phaser(1);
```

Because there is only one registered party, the `Phaser` fires on each call that arrives. Then, blocking on it is as simple as calling `awaitAdvance` with the current

phase value, as shown in **Listing 12**.

The `waitFor` method demonstrates the use of the `DoublePredicate` functional interface introduced in the Java 8 release to support lambdas. This allows it to take an arbitrary condition as a lambda expression to wait for, such as waiting for the scale to read 80 grams or more:

```
scale.waitFor(w -> w >= 80);
```

This is the basis for an improved version of the USB scale communication logic refactored into a new class called `UsbScale.java`. It also implements a unified `Scale` interface that can be used to create an alternate implementation for serial scales. The full set of methods

Figure 2

for the Scale method is shown in **Listing 13**.

There are two additional scale concepts introduced here in order to improve the reliability and consistency of our coffee recipe: a *stable* scale value and a programmatic *tare*. The concept of stability is important, because scales do not immediately register new weights, so the value can fluctuate quite a bit before it settles down. Some scales implement the concept of stability directly in their data reporting; however, USB scales do not do this. As a workaround, when updating the weight, we will compare the last few results to see whether we have reached a steady state, as shown in **Listing 14**.

This allows us to have more-reliable results when executing the recipe, because we know that when we hit a target value, the value is no longer bouncing but has actually stabilized. You can control the number of weights to compare for stability by adjusting the size of the `pastWeights` array. There is a trade-off between speed and accuracy, but a good number seems to be comparing two past readings against the current weight measurement:

```
private final int[] pastWeights =
    new int[] {-1, -1};
```

LISTING 12 LISTING 13 / LISTING 14

```
public void waitFor(DoublePredicate condition) {
  while (!condition.test(getWeight())) {
    scalePhaser.awaitAdvance(scalePhaser.getPhase());
  }
}
```

Download all listings in this issue as text

Resetting the scale value to zero, also called *taring*, is an important part of executing our coffee recipe. Some scales allow you to send a command that programmatically tares; however, these USB scales do not, so the best we can do is to prompt the user to press the TARE button manually. To indicate that programmatic taring is not supported by our scale, we are going to reuse the built-in `UnsupportedOperationException`:

```
@Override
public void tare() {
  throw new
UnsupportedOperationException();
}
```

The full source code for the `Scale` and `UsbScale` classes can be downloaded from the GitHub repository.

Coffee Brewing Recipe

Now that we can asynchronously communicate with a scale hooked up via USB, the next step is to build a coffee recipe that uses our `CoffeeCalculator` and `UsbScale` classes. While you can use any coffee brewing method of your choice, we are going to recommend trying the Aerobie AeroPress brewing system shown in **Figure 2**. It is a simple and inexpensive coffee

Figure 3

brewing device that also travels well as a portable coffee maker.

The way the AeroPress works is that you put your ground coffee and a portion of the heated water into the main chamber. The extraction occurs directly in the main chamber as you stir the grounds and brew for 60 seconds. Then you insert the plunger and create a pressurized space to force the water through the filter. The resulting liquid is a concentrated form of coffee closely resembling espresso, which can then be diluted to the desired coffee strength by adding additional hot water.

The AeroPress combines the benefit of uniform extraction provided by a French press with a sediment-free filtered output similar to what you get with filtered brewing. As a result, it is much less sensitive to grind size and uniformity; however, we still recommend using a high-quality conical burr grinder that will not heat up your beans. If you don't have an electric grinder already, a good choice that pairs well with the travel-friendly nature of the AeroPress is a Japanese-made hand grinder, such as the Porlex Mini shown in **Figure 3**. Not only does it produce very fine and consistent grounds, but it also perfectly fits in the AeroPress chamber once the crank handle is removed.

To support reusable recipes, we are going to build a little bit of infrastructure first. To start, we need

LISTING 15

```
public AeroPressCoffee(double strength) {
  beans = Ingredient.byWeight(CoffeeCalculator.grindWeight(
      strength, CUP_SIZE), "Coffee Beans");
  brewingWater = Ingredient.byWeight(beans.getWeight() /
                        BREW_RATIO, "Water");
  extraWater = Ingredient.byWeight(CUP_SIZE –
                        brewingWater.getWeight(), "Water");
}

@Override
public Ingredient[] ingredients() {
  return new Ingredient[]{beans, brewingWater, extraWater};
}
```

Download all listings in this issue as text

a `Recipe` interface to describe how we are going to specify our AeroPress brewing method and other recipes you might create in the future. The `Recipe` interface has four methods:

```
public interface Recipe {
  String name();
  String description();
  Ingredient[] ingredients();
  Step[] steps();
}
```

The first two methods provide the name and description of the `Recipe` we will be making. The third returns a list of ingredients using a wrapper class, and the final method is the actual set of steps. While it would be possible to simply write the steps as straight Java code, this method allows us to later make improvements in recipe flow, such as skipping or repeating steps.

The `Ingredient` and `Step` classes are fairly straightforward, and both follow a factory pattern where they supply static methods that can be used to construct an instance of the class. In the case of the ingredients, we will specify them by weight, making use of our `CoffeeCalculator` class, as shown in **Listing 15**, which shows the initialization of ingredients for the `AeroPressCoffee` recipe.

This algorithm is designed to

create a fixed volume of liquid, which is set by the `CUP_SIZE` constant in grams/milliliter, and it will use the strength passed in to the constructor, which should be one of the constants defined in the `CoffeeCalculator` class for a regular, rich, or strong cup of coffee.

Because we are using an AeroPress, there is an additional constant called `BREW_RATIO`, which determines how much water we mix with the ground coffee initially and affects the extraction process. If you notice that your coffee is bitter (overextracted from too much water) or sour (underextracted from not enough water), you can tweak this parameter to change the volume of water pressed through the ground coffee. Our recommended targets for these constants are shown in **Listing 16**.

GOOD COFFEE

The important part: The recipe consists of 19 steps that walk you through the process of creating **the perfect cup of Java.**

And finally, we come to the important part: the `AeroPressCoffee` recipe. This consists of 19 steps that walk you through the process of creating the perfect cup of Java, as shown in **Listing 17**. To give this a try, download the full source code from the GitHub repository.

The main class is called `JavaScale`, and it executes a `CommandLineRecipeRunner` that will fire off our `AeroPressCoffeeRecipe` for a strong cup of coffee (see **Listing 18**). **Listing 19** shows the output of running the `JavaScale` application with the default settings.

Once you have tried it with the default settings, here are some things you might want to try tweaking:

- Change the strength of the cup of coffee (`REGULAR` or `RICH`).
- Adjust the volume of coffee created (but remember that the AeroPress has a physical limit on how much liquid it can use during the extraction process).
- Fine-tune the `BREW_RATIO` for your bean type and preference.

Conclusion

With luck, you have enjoyed programming your brew as much as you did drinking it! For more examples of what you can do with embedded Java, check out the upcoming McGraw-Hill book entitled *Raspberry Pi with Java: Programming the Internet of Things (IoT)*, from which this article was adapted. It includes this coffee recipe example as well as examples for general-purpose input/output (GPIO) devices, networking, drones, 3-D printing, and more. `</article>`

LISTING 16 LISTING 17 / LISTING 18 / LISTING 19

```
private static final double CUP_SIZE = 300; // grams
private static final double BREW_RATIO = .2; // beans/water
```

Download all listings in this issue as text

LEARN MORE

- GitHub repository
- usb4java, a Java library for accessing USB devices
- JSR 80 (Java USB API)

ANTONIO GONCALVES

BIO

Part 1

Contexts and Dependency Injection: the New Java EE Toolbox

Using strong typing in dependency injection

Dependency injection solutions have been around for a long time, even before Java. Now commercial and open source products in the Java ecosystem include Spring, JBoss Seam, and Guice. These frameworks were successful but never standardized. Contexts and Dependency Injection (CDI) 1.0 (JSR 299), which is bundled in Java EE 6, provided a strongly typed approach and adopted many JBoss Seam design principles. Since then, CDI was updated to version 1.1 (JSR 346) and included in Java EE 7, and it has had a maintenance release. It has several implementations (Weld, Apache OpenWebBeans, and Caucho Resin CanDI) and is enriched by the Apache DeltaSpike project, which provides several CDI extensions.

With type-safe injection, alternatives, interception, decoration, event management, scopes, and the contextual APIs, CDI has become the Java EE toolbox. We can use it to ease the development of business applications or to extend our in-house technical framework.

Let's focus on type-safe injection and see what strong typing means in dependency injection.

Injecting Programmatically

Applications are made of business logic, interaction with other systems, user interfaces, external APIs, and so on. In fact, object-oriented programming summarizes these functions by classes depending on other classes. These classes can be strongly coupled (for example, depend on an implementation) or loosely coupled (that is, depend on an interface). Let's see a concrete example.

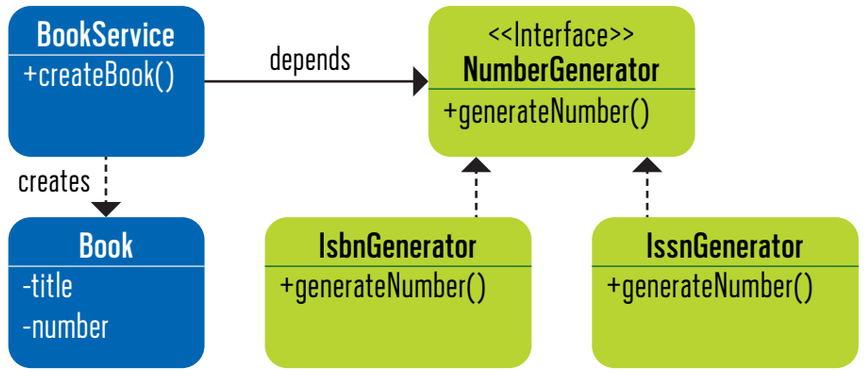


Figure 1

Figure 1 shows a BookService class whose job is to create a Book object. A book contains a title and a number. This number can be an ISBN number (a thirteen-digit number generated by the IsbnGenerator class) or ISSN number (an eight-digit number generated by the IssnGenerator class).

BookService depends on the NumberGenerator interface, which has one method called generateNumber. The BookService can then choose from the IsbnGenerator or IssnGenerator implementation according to some condition or environment. In terms of code, one solution is to pass the implementation to the constructor (see **Listing 1**) and leave an external class (the Main class) to choose which implementation it wants to use:

```
public class Main {
  public static void
   main(String[] args) {
    Book book =
new BookService(
     new IsbnGenerator()).
      createBook("H2G2");
    System.out.println(book);
  }
}
```

In the Main class, the implementation of the NumberGenerator is passed as a parameter of the constructor. So if you need a BookService that generates an ISBN number, you just pass the IsbnGenerator implementation to the constructor. If you need to generate an ISSN number, you change the implementation to be IssnGenerator. This is what's called Inversion of Control (IoC): the control of choosing the dependency is inverted because it's given to an external class. But at the end of the day, you end up constructing dependencies programmatically, and you can leave an injector to do it. And that's where CDI can help.

Injecting with CDI

CDI is a standard solution that manages dependencies between classes. Injection is made using strongly typed annotations, called qualifiers. CDI removes boilerplate code by using a simple API, so we don't have to do construction of dependencies by hand. To see how this works, let's re-examine our example.

In CDI, there's an annotation you need to remember: it's @Inject, which is used by the runtime to inject the reference of an implementation. So, change the code of the BookService, get rid of the constructor, and use @Inject on the NumberGenerator interface, as shown in **Listing 2**.

Which implementation is injected: the IsbnGenerator, which generates a thirteen-digit ISBN number, or the IssnGenerator, which generates an eight-digit ISSN number? Neither. CDI considers this injection ambiguous, won't deploy the application, and throws an exception. At system initialization, the CDI runtime must validate that exactly one bean satisfies each injection point. So, if two implementations of NumberGenerator were available, the container would inform you of an unsatisfied dependency and wouldn't deploy the application.

To solve this ambiguous dependency, the solution is to have different qualifiers: a different one for each implementation. That's when you need to create your own annotations and give them a specific name that suits your business. In this case, to qualify the IsbnGenerator implementation, I created a qualifier called @ThirteenDigits:

LISTING 1 LISTING 2 / LISTING 3 / LISTING 4

```
public class BookService {

    private NumberGenerator generator;

    public BookService(NumberGenerator generator) {
        this.generator = generator;
    }

    public Book createBook(String title) {
        return new Book(title, generator.generateNumber());
    }
}
```

Download all listings in this issue as text

```
@Qualifier
@Retention(RUNTIME)
@Target({FIELD, TYPE, METHOD,
 PARAMETER})
public @interface
 ThirteenDigits {
}
```

I also created a qualifier called @EightDigits for the IssnGenerator:

```
@Qualifier
@Retention(RUNTIME)
@Target({FIELD, TYPE, METHOD,
 PARAMETER})
public @interface EightDigits
{
}
```

Then, if you need to inject the IsbnGenerator, qualify the injection point on the BookService (see **Listing 3**) and the IsbnGenerator implementation (see **Listing 4**).

As you might have guessed, if you change the injection point to @EightDigits, CDI injects the IssnGenerator implementation.

A CDI qualifier is basically a Java annotation with an extra @Qualifier annotation. A qualifier represents some semantics associated with a type that is satisfied by some implementation of that type. It's a user-defined annotation and can be called whatever you want to call it. You can introduce a qualifier to represent a thirteen-digit generator or an eight-digit generator. The idea is that you can create as many qualifiers as your application

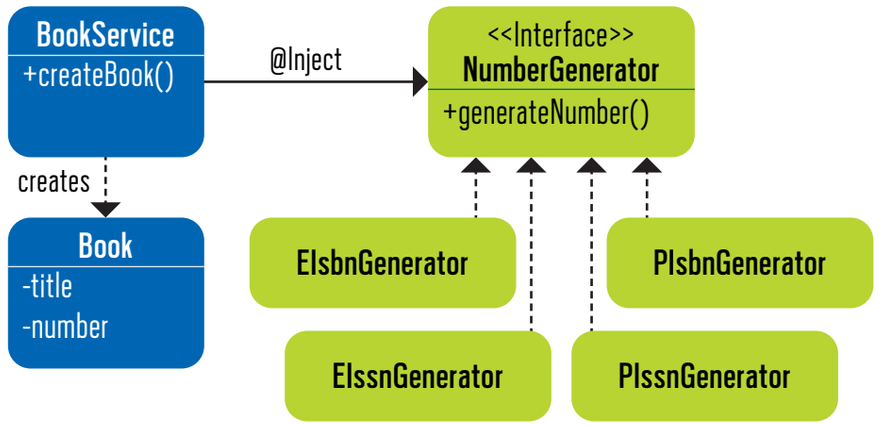


Figure 2

LISTING 5 LISTING 6 / LISTING 7

```
public class BookService {

    @Inject @Default
    private NumberGenerator generator;

    public Book createBook(String title) {
        return new Book(title, generator.generateNumber());
    }
}
```

Download all listings in this issue as text

needs. After you define the needed qualifiers, they must be applied on the appropriate implementation and injection point.

The default qualifier. Now let's see why the dependency in **Figure 1** was ambiguous. Assume that you haven't qualified anything yet, and that @ThirteenDigits and @EightDigits don't exist.

Whenever a bean, or injection point, doesn't explicitly declare a qualifier, CDI assumes the qualifier @javax.enterprise.inject.Default. In fact, the BookService in **Listing 2** is identical in having a default injection point, as seen in **Listing 5**. @Default is a built-in qualifier that informs CDI to inject the default bean implementation. If you define a bean with no qualifier, the bean automatically has the qualifier @Default. Both implementations IsbnGenerator and IssnGenerator have the same @Default qualifier, which is why the dependency is ambiguous. CDI doesn't know which implementation to choose from. That's why you had to create specific qualifiers.

The injection point. The @Inject annotation defines an injection point that is injected during bean instantiation. Injection can occur by three mechanisms: property, setter, or constructor. The code in **Listing 2** uses property injection. With this mechanism, you don't have to create a getter and a setter. CDI can access an injected field directly, even if it's private. But instead of annotating the attributes, you can add the @Inject annotation on a constructor (see **Listing 6**). In this case, the rule is that you can only have one constructor injection point. The other choice is to use setter injection, which looks like constructor injection.

The deployment descriptor. Nearly every Java EE specification has an optional XML deployment descriptor. It usually describes how a component, module, or application should be configured. With CDI, the deployment descriptor is called beans.xml (see **Listing 7**) and is also optional.

In the beans.xml, you can set bean-discovery-mode to change the CDI bean discovery resolution. The default is annotated, in which case CDI discovers only annotated beans. But it can be changed to all (so all the beans are discovered) or none (to disable CDI discovery on a particular archive). The beans .xml descriptor can also be used to configure certain functionalities such as alternatives, decorators, or interceptors.

Qualifiers with members. Imagine that our NumberGenerator interface has several implementations (see **Figure 2**): EIsbnGenerator generates thirteen-digit ISBN numbers for electronic documents, and EIssnGenerator generates eight-digit ISSN numbers for electronic documents. For any document that is printed on paper, you need different generators: the PIssnGenerator generates an eight-digit number, and PIsbnGenerator generates a thirteen-digit number.

You could imagine having even more implementations. You could create as many qualifiers as you have implementations. But if you create a qualifier each time you

need to inject something, your application will be very verbose with lots of empty qualifiers. That's when qualifiers with members can help.

A qualifier is an annotation, so it can have as many members of any type as needed. Here, I got rid of the @ThirteenDigits and @EightDigits qualifiers and replaced them with a more-generic one called @Generator (see **Listing 8**). This @Generator qualifier has a first member of type NumberOfDigits. With this enumeration, you can differentiate beans that generate 13 digits or 8 digits. The second member is called printed and is of type boolean. It is set to true to represent a number for printed documents, and it is set to false for electronic documents.

To differentiate the four implementations, use your Generator qualifier and set different values on each member. EIsbnGenerator generates a thirteen-digit number for electronic documents, so the member numberOfDigits has the value THIRTEEN and the member printed is false (see **Listing 9**). Each implementation is uniquely defined, thanks to the values of the qualifier.

The way you use qualifiers with members on injection points doesn't change from what we've seen. The injection point qualifies the needed implementation by setting the annotation members. In **Listing 10**, you inject the thirteen-digit generator for electronic documents, and just by changing the printed value to true, CDI injects a different implementation. Change the enumeration value and you get an eight-digit number generator for printed documents.

Multiple qualifiers. Another way of qualifying a bean and an injection point when there are many implementations is to specify multiple qualifiers. So, you could keep @ThirteenDigits to qualify a thirteen-digit number generator and @EightDigits to qualify an eight-digit number generator. But because this is not enough, you could create other qualifiers for electronic documents and printed documents. The idea is always to identify each implementation uniquely so CDI can wire dependencies. Another way to qualify EIsbnGenerator would be to aggregate both @ThirteenDigits and @Electronic (see **Listing 11**).

CDI QUALIFIERS

Qualifiers are a type-safe way to resolve ambiguous dependency.

LISTING 8 LISTING 9 LISTING 10 LISTING 11 LISTING 12 LISTING 13

```
@Target({FIELD, TYPE, METHOD})
@Retention(RUNTIME)
@Qualifier
public @interface Generator {

    NumberOfDigits numberOfDigits();
    boolean printed();

    public enum NumberOfDigits {
        EIGHT,
        THIRTEEN
    }
}
```

Download all listings in this issue as text

Alternatives

With qualifiers, you can choose between multiple implementations of an interface at development time in a type-safe manner. But sometimes you want to inject an implementation depending on a particular deployment scenario. For example, you might want to use a mock number generator but only in a testing environment. That's when you can use alternatives. Alternatives are beans annotated with the special qualifier @Alternative, which are disabled by default. You need to enable alternatives in the beans.xml descriptor to make them available for instantiation and injection.

Let's add a new MockGenerator implementation. In terms of code, the novelty is that MockGenerator is annotated with the @Alternative qualifier (see **Listing 12**). This means that CDI treats it as the default alternative of the NumberGenerator.

In terms of the injection point, nothing changes. BookService is not affected (see **Listing 2**). The code injects the default implementation of a number generator, because alternatives are disabled by default. So until this point, CDI would not inject the mock implementation. To enable the @Alternative, you need to add it to the beans.xml deployment descriptor (see **Listing 13**). It's just a matter of adding the fully qualified class name of the MockGenerator inside an alternatives element. You can

LISTING 14

```
@Alternative
@ThirteenDigits
public class MockGenerator implements NumberGenerator {

    public String generateNumber() {
        return "mock-" + Math.abs(new Random().nextInt());
    }
}
```

Download all listings in this issue as text

have several beans.xml files declaring several alternatives, depending on your environment—for example, one beans.xml file for testing with all the mock alternatives, one for production without any mocks, and so on.

Alternatives can be qualified. So you could define the MockGenerator as the alternative for only IsbnGenerator (thirteen digits). **Listing 14** says that if the alternative is enabled in beans.xml, and if you inject the @ThirteenDigits implementation, CDI always injects the mock.

Conclusion

CDI accommodates simple injection cases, such as the default injection, but it can also solve more-complex cases, thanks to qualifiers. Qualifiers are used as a type-safe approach for solving ambiguous dependency. Qualifiers are annotations, so they can have members or be aggregated with other qualifiers to form a uniquely defined injection point. Alternatives can be used to affect injection points at deployment time, thanks to the beans.xml deployment descriptor. As you can see, CDI is rich in terms of injection, is type-safe, and has an easy-to-use API.

This article is the first of four about CDI. The next article explains how to integrate third-party frameworks using producers. The third article focuses on how you get loose coupling with interceptors, decorators, and events. The last article covers the integration of CDI in Java EE. `</article>`

LEARN MORE

- CDI specification
- Weld reference implementation
- *Beginning Java EE 7* (book)

PLURALSIGHT

INSPIRE. TEACH. TRANSFORM.

Share your knowledge with the world.
Join the largest tech & creative training
resource on the planet.

PLURALSIGHT.COM/TEACH

MICHAEL KÖLLING

BIO

Part 2

Create a Simple Paddle Game with Microsoft Kinect

More fun with using Greenfoot to drive Kinect from Java

Greenfoot is a Java-based graphical programming environment aimed at novice developers. In the September/October 2014 issue of *Java Magazine*, we started programming with Greenfoot and Microsoft Kinect, a sensor board that lets you track body movements of people in front of a camera. If you aren't familiar with either Greenfoot or Kinect, read the previous article first to learn more about the setup of the hardware and the system, the basics of accessing Kinect from Greenfoot, and the programming tasks required for a painting program that let us paint onscreen by moving our hands in the air.

In this article, we take a traditional, keyboard-controlled Pong-style game and turn it into a game controlled with gestures—using two paddles, one held in each hand (see **Figure 1**).

Old-Style Pong

Start by downloading the base project so you can program along as you read through this article: kinect-pong.zip. This Greenfoot scenario implements a basic Pong-style game: Players bounce a ball up and down with paddles at the top and bottom of the screen. The bottom paddle is controlled by the player using the cursor keys on the keyboard, and the top paddle is computer controlled.

Make sure you have Greenfoot installed, and then open the scenario and play it. Read through the code of the existing classes—they should be easy to understand.

Then set up your Kinect and start the Kinect server, as described in the previous article. Now you're ready to get started with the Kinect programming for this scenario.

Detecting Users

The first thing we want to do is to detect when a user steps

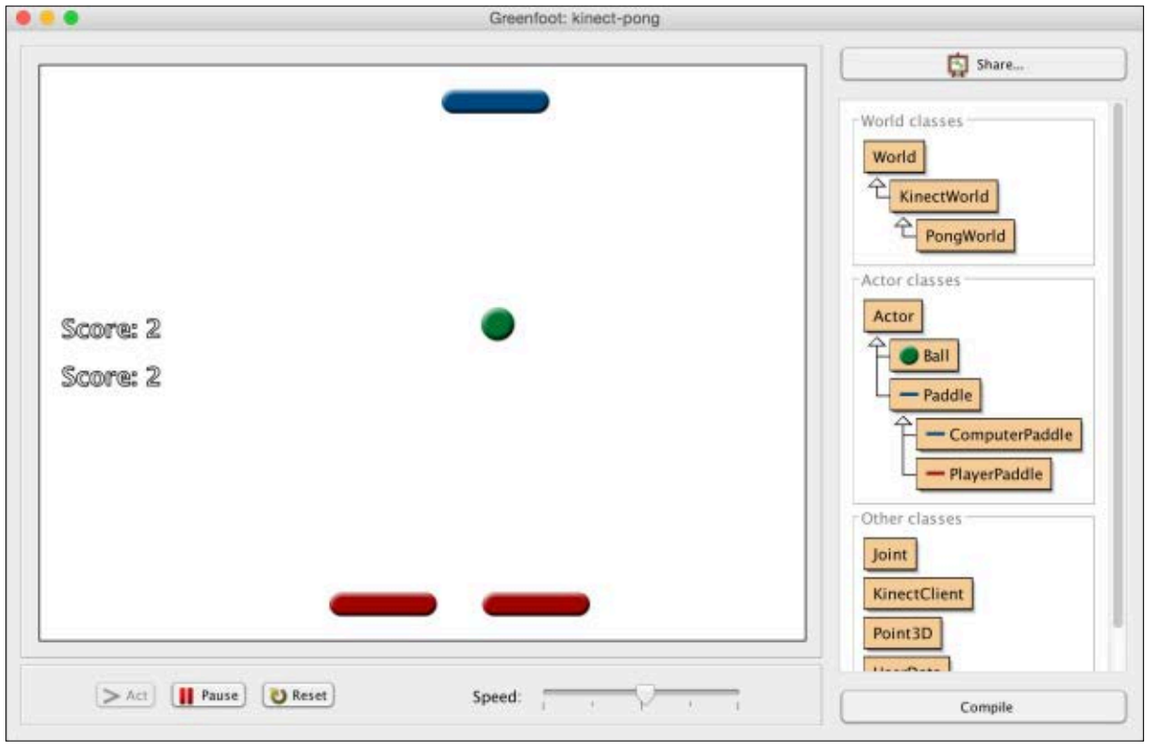


Figure 1

PHOTOGRAPH COURTESY OF UNIVERSITY OF KENT

in front of the Kinect camera. Our game should be in an idle state (waiting) as long as there is no user, and then starts when a user enters Kinect's view. In our project, we implement this in the `PongWorld` class.

We can get an array of all users being tracked by using the `getTrackedUsers` method:

```
UserData[] users =
    getTrackedUsers();
```

We can then check whether any users are present by checking whether the length of this array is zero:

```
if (users.length > 0) {
   ...
}
```

Start by moving the code that creates the `PlayerPaddle` and the `Ball` from the `PongWorld` constructor into a new private method called `startGame`. Then, we can call this method from our `act` method when a user enters the game:

```
if (users.length > 0) {
   startGame();
}
```

This code is somewhat problematic, though: Because the `act` method is called repeatedly while the scenario is running, many balls and paddles would be created. We need to make sure that this happens only once, when the player first enters the view. Use the `idle` flag, which is already defined in the class, for this purpose:

```
if (idle) {
   if (users.length > 0) {
      startGame();
      idle = false;
   }
}
```

You can see the code completed so far in **Listing 1**.

Controlling the Paddle

At the moment, our game detects a player in front of the camera, but the player can't control the paddle yet. This is what we will do next.

For the paddle to make use of the player's information, we need to pass the tracked user data into the `PlayerPaddle` object. Because this is a single-player game, we can just use the first tracked user out of our array (ignoring all others if there is more than one user) and pass it to our `startGame` method:

```
startGame(user[0]);
```

We then add the appropriate parameter to the definition of our

LISTING 1

```
public class PongWorld extends KinectWorld
{
    private boolean idle;

   /* Create the game world with the game objects in it. */
    public PongWorld()
    {
        super ();
        addObject(new ComputerPaddle(), 200, 30);
        idle = true;
    }

    /* Nothing to do. */
    public void act()
    {
        super.act();
        UserData[] users = getTrackedUsers();

        if (idle) {
            // check whether a user has arrived
            if (users.length > 0) {
                // discovered a new user
                startGame();
                idle = false;
            }
        }
    }

    private void startGame()
    {
        addObject(new PlayerPaddle(), 200, 450);
        addObject(new Ball(), getWidth()/2, getHeight()/2);
    }
}
```

Download all listings in this issue as text

`startGame` method, and pass the parameter into the `PlayerPaddle` constructor. The new version of the `startGame` method doing this is shown in **Listing 2**.

Now we'll move on to the `PlayerPaddle` class. In the `PlayerPaddle` constructor definition, we add a parameter to receive the `UserData` object and then store it in an instance field:

```
private UserData user;

public PlayerPaddle(
    UserData user)
{
    this.user = user;
}
```

Now we have the user data available to rewrite the `act` method.

The current `act` method contains code that moves the paddle in reaction to cursor keys. Remove this code.

Instead, we now want to add code that sets the position of the paddle according to the right hand of the user. We can achieve this by getting the user's hand's x-coordinate and using that as the paddle's x-coordinate. We'll leave the y-coordinate of the paddle unchanged. The code to do this is as follows:

```
Joint hand = user.getJoint(
    Joint.RIGHT_HAND);
setLocation( hand.getX(),
    getY());
```

Listing 3 shows the complete class at this stage. Try it out: You should be able to play the game now and control the paddle with your hand.

Stopping the Game

Currently, we start the game when a player enters the view of the camera, but we don't stop it when the player leaves. That makes our game unplayable for anyone after the first player has finished.

Let's add some code that detects when a player leaves the game and have it respond appropriately. For this, go back to our `PongWorld` class.

In this class, add a new private method called `stopGame`, which removes the `PlayerPaddle` and `Ball` objects when the game stops. **Listing 4** shows an implementation. You must also modify your `act` method so that it calls `stopGame` when the world isn't idle and no users are visible. (Look ahead to **Listings 6a** and **6b** if you have trouble doing this on your own.)

After this much coding, it's good to test. When you start the scenario, there should be no ball and no player paddle. Once you step into Kinect's view, the paddle and ball should appear, and the game should start. When you leave the view of the camera, Kinect should attempt for another few seconds to track you, and then give up. The paddle and ball should then disappear.

LISTING 2 LISTING 3 / LISTING 4

```
private void startGame(UserData user)
{
    addObject(new PlayerPaddle(user), 200, 450);
    addObject(new Ball(), getWidth()/2, getHeight()/2);
}
```

Download all listings in this issue as text

The Two-Paddle Version

The last task is to extend our game to use both our hands by giving the player two paddles, one controlled by each hand.

In the current version of the `PlayerPaddle` class, we used the constant `Joint.RIGHT_HAND` to specify which hand we want to track. We now generalize this class so that it can track either hand, and then we can create two paddle objects—one for each hand.

The type of this constant is an `int`. We, therefore, add an `int` parameter to specify the hand to the `PlayerPaddle` constructor, and store the value we receive in a field. Then, when we want to retrieve the hand's x-coordinate in the `act` method, we use this field to specify which hand to use. The full code for this is shown in **Listing 5**.

After setting the paddle like this, we must adapt the code that creates the paddle to pass the additional parameter for the hand. Previously, the instruction to create the paddle looked like this:

```
new PlayerPaddle(user)
```

Now we add the second parameter to make it look like this:

```
new PlayerPaddle(user,
      Joint.RIGHT_HAND)
```

Adding a second paddle for the other hand is now easy. Just duplicate the line of code that creates the paddle and adds it to the world, and then use the left-hand constant to create the second paddle:

```
new PlayerPaddle(user,
      Joint.LEFT_HAND)
```

Listings 6a and **6b** show the complete `PongWorld` class, including these additions.

Conclusion

This example shows how to use the Microsoft Kinect's input to control a little game. If you programmed along as you read this article, you've seen that using the location data of tracked users to control your program isn't difficult.

The Kinect-based examples work very well to demonstrate how easy it can be to program in Java. Greenfoot makes it easy to get started with Java programming, including working with Kinect.

The example presented here is taken from *Introduction to Programming with Greenfoot* (Pearson, 2015), which discusses it in more detail. You can also find other Greenfoot Kinect projects there. `</article>`

LEARN MORE

- Greenfoot download page
- Instructions for Greenfoot and Kinect setup
- Greenfoot book

LISTING 5 LISTING 6a / LISTING 6b

```
import greenfoot.*;

/**
 * A pong game paddle controlled by the player.
 *
 * @author Michael Kölling
 * @version 1.0
 */
public class PlayerPaddle extends Paddle
{
    private UserData user;
    private final int HAND;  // right or left hand

    /**
     * Create a player paddle.
     * The parameters are the user data of the controlling
     * user, and a constant specifying which hand to track.
     */
    public PlayerPaddle(UserData user, int hand)
    {
        this.user = user;
        this.HAND = hand;
    }

    /**
     * Place the paddle at the x-location of the
     * user's hand.
     */
    public void act()
    {
        Joint hand = user.getJoint(HAND);
        setLocation(hand.getX(), getY());
    }
}
```

Download all listings in this issue as text

BRUNO SOUZA AND
EDSON YANAGA

BIO

Part 2

More Ideas to Boost Your Developer Career

Study code, go to conferences, but use the conferences to build your network.

In the first article of this series (January/February 2015 issue, p. 25), we introduced three broad areas that you could focus on to improve your career as a developer. *Code* is the most important, because this is the basic skill for a developer. But the other two, which we named *Community* and *Think Differently*, can't be left behind. They are the things that will help you break through the boundaries of your career.

Spending some weekly practice time in those three areas will help you become a better developer. It will make you a more complete professional, and enable you to go after new opportunities. All this helps you boost your career and find new paths. And the suggestions we present here are not hard things to do, nor time consuming. You can squeeze them into your busy day.

Code: Open Source Software

We have already seen that to be good at coding, you must practice a lot. But even with practice, how do you know if you're writing good code or bad code? We still have not reached the point of having a practical and noncontroversial way of measuring code quality. Code quality is still most of the time a qualitative, rather than quantitative, subject. You can't say for sure if your code is good, but you can always recognize better or worse code. To make sure you're on the right path to becoming a better developer, your best bet is to assess your progress against other developers' code.

In the software world, the way to watch others perform is by looking at their software, and the source code they wrote. That allows us to study it, experiment with it, and even improve it. But for that, we need access to the source code of great developers.

This is the beauty of open source: It's freely available for you to study and compare with your own code. You can find a huge variety of different projects on GitHub and Bitbucket, ranging from small to enterprise-level projects. You can learn new techniques, and you can explore different points of view. You can be awed by great code and learn what not to do from developers who are not accomplished.

Want to explore different ways of testing code? Dig into open source projects and see how other developers do it. Don't know the proper way of doing logging in your systems? Get an open source project you use that provides great logging information and learn to do it the same way.

Contributing to open source enables you to practice some skills that maybe you don't (or can't) use in your daily work. Many of us are limited at work to creating traditional enterprise systems with create, read, update, and delete (CRUD) features. But in our spare time, we can choose any open source project that sparks our interest to learn or contribute to. Sometimes even at work, we can take a relevant open source tool or library and fix a bug or implement a new feature. It certainly is exciting, and it helps us to think outside the box.

BRUNO SOUZA PHOTOGRAPH BY BOB ADLER/GETTY IMAGES

You can start small, probably just adding some documentation or opening an issue at the bug tracker.

Later maybe you'll get to submit your own pull requests with bug fixes or new features. At that point, you'll get another great benefit of open source: the feeling you get when your contribution is accepted and gets in the hands of dozens (or thousands) of developers in the next version of the project. Bragging rights included! Recognition, more networking, new friends, maybe even a job offer. Open source puts your code in the open, and the light that shines on it helps your developer career to blossom.

ACTION ITEM: Choose one open source project that you use in your daily job, and look at the code behind it. It can be a framework, a library, maybe the Java Virtual Machine (JVM) in the OpenJDK project or the code in a JSR reference implementation. But do not only stare at the code; learn how it works, and experiment with it.

ACTION ITEM: Choose one open source project in an area that you love. It could be technical, such as database internals or memory allocation. Maybe it is something else, for example a music project, games, or something you always wanted to do as a kid—such as space exploration or soccer. Learn the code. See if there is anything you would like changed or improved.

Community: Events

Even with the great technology innovations we have had in the last few years, nothing compares to face-to-face conversations. To many people, the value of participating in events is questionable. They can be large time and money investments, while sessions can be seen online instead, and there is no guarantee of good content—not to mention being away from family while work keeps piling up back at the office.

FACE-TO-FACE

These days on the internet, being seen is easy. **The hard part is being remembered.** Go out and enjoy the interactions.

Don't be misled—the real value of events is in the people and relationships. Face-to-face events are a great place to increase networking and visibility.

"Whoever is not seen is not remembered" is a useful observation. These days on the internet, being seen is easy. The hard part is being remembered—and we are best remembered by those we meet in person and whom we share moments with.

That's why when you go to an event, large or small, you should decide to enjoy the interactions: a conversation in the registration line, sitting down to eat with strangers in the crowded lunchroom, late night dinner or drinks. At an event, there are countless opportunities to just say hi and expand your network. It is amazing how easy it is to find people who are having problems similar to yours, or who have experience with things you would love to try.

And you can maximize your experience by going a step above most attendees. Go say hello to a speaker you just watched. Track him or her later in the halls for a friendly conversation. Skip a couple of sessions to have a discussion or enjoy a drink with someone you just met. Sit down to show your solution, or ask someone to show you their work. Share moments with other developers.

And bring those moments back to your own team! Present in your words the best talks you have seen. Introduce people you meet to those on your team who would be interested in their work. Blog about something cool you have learned, or share that little tidbit of a hall conversation that probably no one else heard but you.

You can also submit a talk to become a speaker at an event. That will give you another perspective on event participation—a topic we'll explore in the future.

ACTION ITEM: Register and attend a local event on a topic that interests you. While there, focus on the people. Make a point of meeting as many people as you can. Set out an event schedule that gives you room to talk with speakers and hold extended hall conversations. Do not spend all the time with your friends. Meet new people every day.

ACTION ITEM: When meeting people, ask what projects they have been working on recently. Be prepared to share what you have been working on. It's a much better conversation starter than asking, "How are you?"

Think Differently: The Internet of Things

Research indicates that the Internet of Things (IoT) will change the world in the next decades. According to Forbes, by 2020, an estimated 50 billion devices around the globe will be connected to the internet. The worldwide impact will come from the evolution of experiments done today by developers and hobbyists. We are in the early days of IoT, and most projects are seen as mere toys used for fun. But we already have a large array

of tools, kits, boards, and sensors readily available for anyone to use. We are seeding and fertilizing the ground, and when the crops start to grow, it will be an explosion of opportunity everywhere.

But as we have been saying in this series, you don't need to be interested directly in IoT for it to help improve your career. Most of us will only be on the receiving end of the IoT explosion, being users of products and services enabled by those ideas. More probable than not, your company or project will have nothing to do with IoT for quite some time. Maybe you won't have a real opportunity to use it careerwise anytime soon. So, why bother?

IoT is one of those ideas that forces us to think differently. Maybe you remember the Sun Microsystems slogan that "the network is the computer," and how the emergence of the network in the 1990s changed how we create software and services. We are still riding that wave today. Being able to experiment and peek into the future right now does change how we do things today, and how we choose our path for tomorrow.

IoT makes us think differently because it changes the basic tool of our trade: the computer. It is the opposite side of virtualization—when computers become part of our world, and are able to read reality and directly manipulate it.

This changes how we interact with those computers and how the software we write interacts with us. It's a different kind of software, in terms of usability, longevity, security, and even size and performance. When we deal with the cloud, we try to think how our application would be different if everything were infinite. When dealing with IoT, we think how our application needs to be different if it never stops, if it does one thing only and is never upgraded. Network access can be an issue, and there may be different constraints that will sharpen your thinking in ways that will help in your daily work. Working with IoT might well open up your career for new opportunities in the future. It will also show a fun side of development, one that we sometimes forget exists.

TO DO
"Use Java on the Raspberry Pi."

ACTION ITEM: Buy a Raspberry Pi kit (they are cheap these days) and load some software on it to get a feeling for what can be done. Bonus points if you get a few sensors!

ACTION ITEM: Use Java on the Raspberry Pi, and get started exploring some ideas of what you can do. The Java.net IoT Developer Challenge has a series of great videos that will get you going in no time.

Put It All Together

One of the best things about events is that they give us a chance to meet people and participate in activities that can expand our networking. How about joining an Adopt-a-JSR hackathon? You can learn, code, and contribute to an open source reference implementation. Why not attend JavaOne, the premier Java conference in the world? Or easier, join a local Java user group (JUG) event about IoT. There's no need to spend all your time in sessions. Sit in the hall, and hack on projects together with other attendees.

Does being bold and introducing a new topic to a user group seem intimidating? In the next article, we'll discuss how you, too, can become a speaker and how that can help you improve your developer career. `</article>`

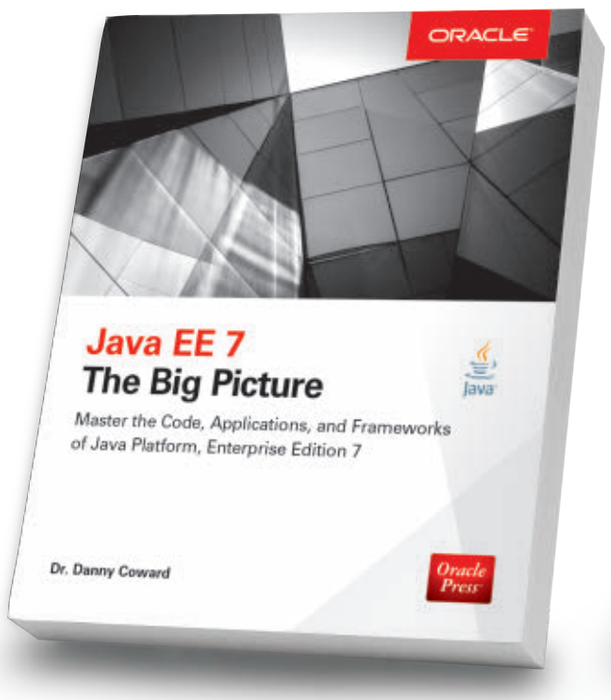
LEARN MORE

- Join a JUG
- "Code Java on the Raspberry Pi," November/December 2014 issue, p. 35
- "Brewing Java with the Raspberry Pi," in this issue

Oracle Press™

Your Destination for Java Expertise

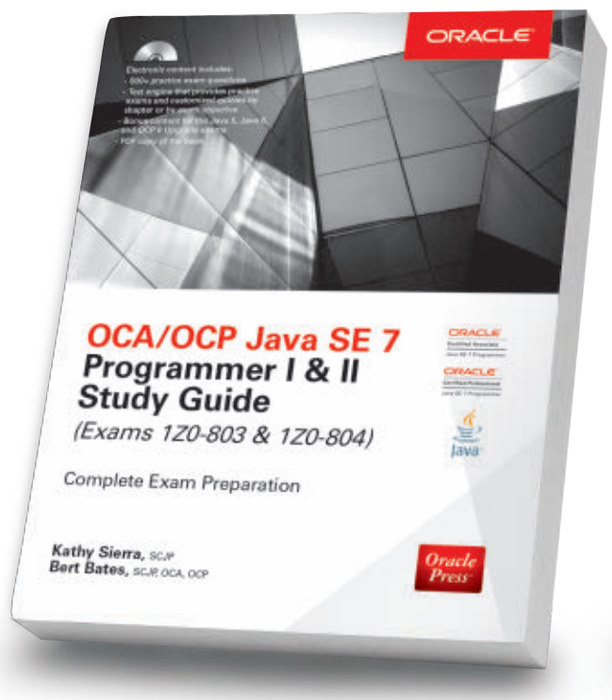
Written by leading Java experts, Oracle Press books offer the most definitive, complete, and up-to-date coverage of Java available.



Java EE 7: The Big Picture

Danny Coward

Master the code, applications, and frameworks that power Java Platform, Enterprise Edition 7.



OCA/OCP Java SE 7 Programmer I & II Study Guide

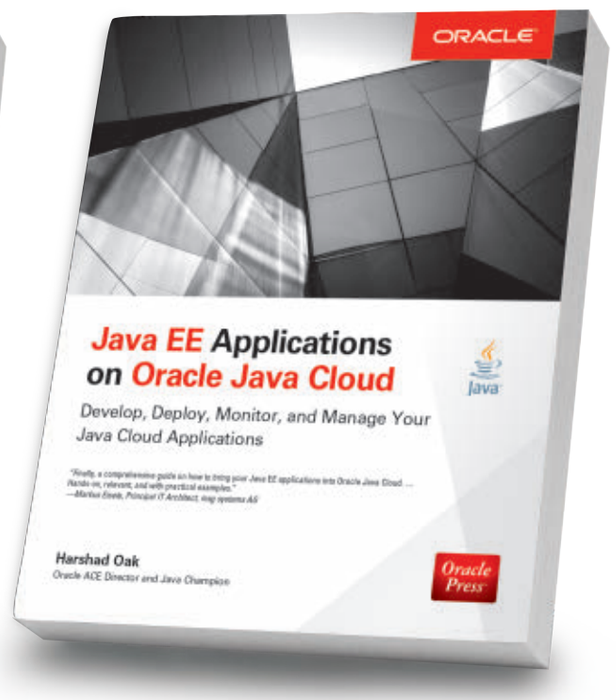
Kathy Sierra, Bert Bates

This self-study tool offers full coverage of OCA/OCP Exams 1Z0-803 and 1Z0-804. Electronic content includes 500+ practice exam questions.

Mastering JavaFX 8 Controls

Hendrik Ebbers

Create custom Java FX 8 controls and deliver state-of-the-art applications with visually stunning UIs.



Java EE Applications on Oracle Java Cloud

Harshad Oak

Build highly available, scalable, secure, distributed applications on Oracle Java Cloud.

Available in print and as eBooks

Oracle Press™

www.OraclePressBooks.com • @OraclePress

Magnusson consulting with developer colleagues at SourcePoint

JCP Executive Series

Working on Java from Within the JCP

SourcePoint's **Geir Magnusson Jr.** discusses the advantages of building systems with Java, and how the JCP has affected the platform's evolution. **BY STEVE MELOAN**

PHOTOGRAPHY BY MATT MILLS MCKNIGHT/GETTY IMAGES

Currently the CTO of SourcePoint, Geir Magnusson Jr. was a cofounder of the Apache Geronimo J2EE project and the Apache Harmony J2SE project. He is a member of and has served on the board of directors of the Apache Software Foundation. He is currently an individual representative on the Java Community Process (JCP) Executive Committee, and previously represented the Apache Software Foundation on the JCP Executive Committee.

Java Magazine: Given that we're celebrating 20 years of Java, what has surprised you most about the evolution of the platform during that time frame?

Magnusson: I've been impressed by how Java has remained so relevant. I can't think of many developer-focused technologies that have survived and blossomed in the same way.

There are obviously other programming

languages that have been in the game for a long time, like C and C++. But in my opinion, they are not adapting to new computing niches and opportunities with the same flexibility as Java. In every iteration of technology—from smartphones to tablets to whatever will come next—I believe Java will be there.

Java Magazine: Why do you think Java has continued to flourish?
Magnusson: I build software systems every day, and I've done it for a variety of industries during my career. Java is not the only tool I use, because other languages are more appropriate for certain parts of the architecture. But when Java is a good fit, it's a really great fit. I can depend on it for reliability, scalability, and performance. Java is a main tool in my toolbox. And that usability underpins my continued interest in participating in the broader ecosystem around Java, such as the JCP.

Java Magazine: What would you like younger developers to know about the 20th anniversary of Java? And do you have any advice for students and new developers regarding Java technologies, or software technologies in general?
Magnusson: Developers are very much creatures of fashion—not necessarily in their attire, but certainly in the tools and technologies they use. Because of this, we've seen the meteoric rise of some platforms, and then a descent.

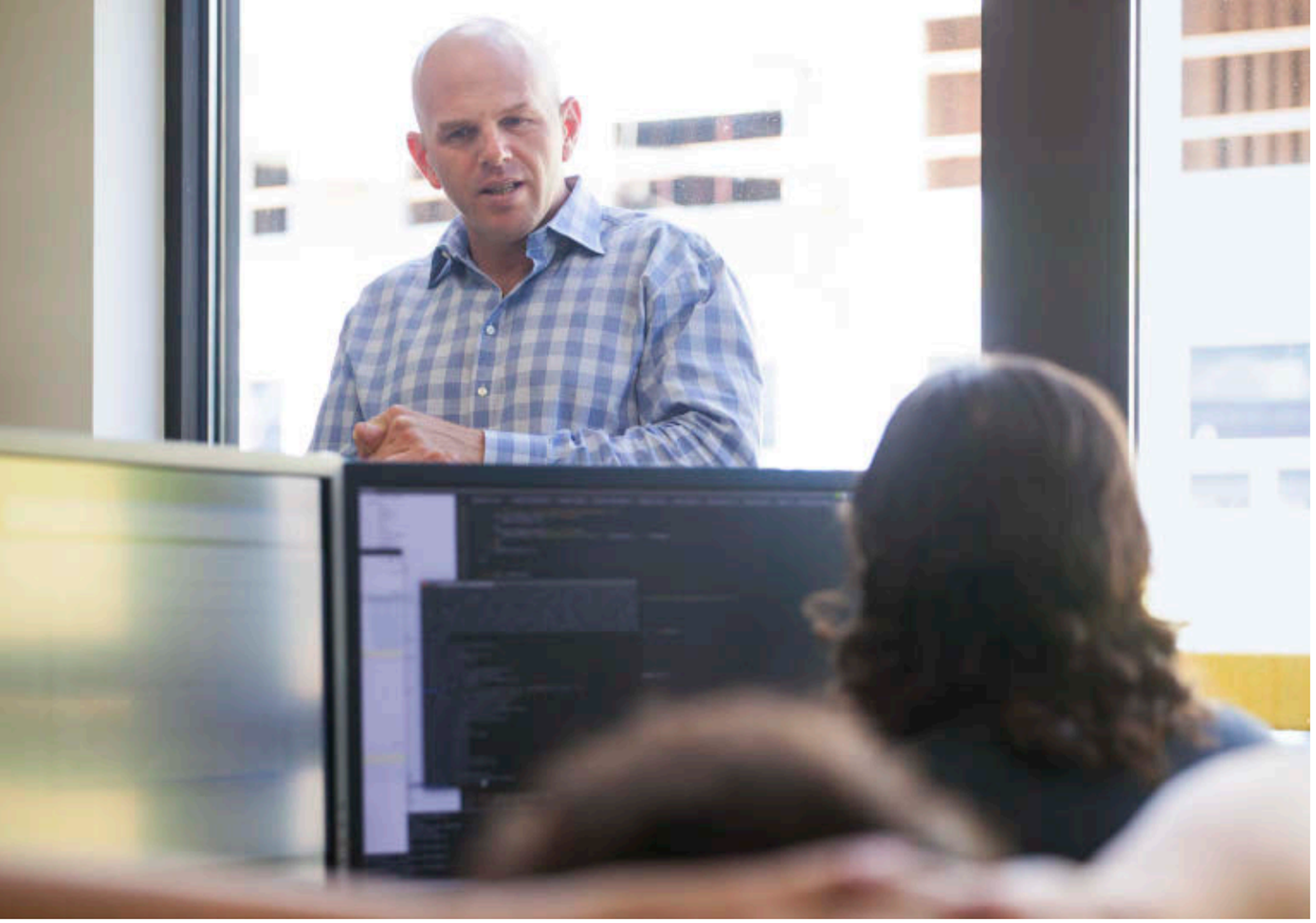
But I think the fact that we're celebrating the 20th anniversary of this general-purpose application development platform indicates that Java technologies should be known and understood by any modern coder or system architect.

Except for a few niche markets, virtually every developer will need to write, interface with, or maintain Java software.

And regarding students and young developers, I think it's important for them to understand that the JVM [Java Virtual Machine] is one of the most sophisticated pieces of software they will encounter. It's something they can rely on as they build their careers. With languages like Scala, Groovy, and Jython also running on the JVM, there's tremendous flexibility for building systems. So Java is a great skill-set investment for young professional developers.

Java Magazine: What is the importance of the JVM in the world of Java technologies?
Magnusson: I would call the JVM the crown jewel of the Java ecosystem. It provides a computing model that

Magnusson started his career in R&D at Bloomberg.



Says Magnusson, "Being able to write code that is very clear to the reader on the first time through is incredibly important."

handles concurrency and memory management in a consistent and highly reliable fashion.

The garbage collector, for instance, simplifies the life of a programmer immensely. If I need large heap, I have no concerns, due to the fine-grained control over garbage collection. I don't have to worry about memory management.

In addition, just-in-time [JIT] compilation has stopped seeming like magic. I now take it for granted that the JVM will convert bytecodes into very fast machine code that is completely optimized. As I've mentioned, a variety of languages have blossomed to take advantage of this amazingly sophisticated piece of software technology.

And it continues to improve all the time. I've used other garbage-collected languages, but the JVM is head and shoulders above the rest.

Java Magazine: Are there any misconceptions about Java that need to be dispelled?

Magnusson: There is a common misperception that Java is a slow language, which is completely untrue. It's used in the high-frequency trading community and other domains that require excellent performance and very low latency.

Java, the language, is also sometimes viewed as being overly verbose. But programming languages are for humans to read, not computers. Being able to write code that is very clear to the reader on the first time through is incredibly important for maintainability and clarity.

I've also heard the occasional claim that software development in Java is a slow process. But in my experience, writing, compiling, executing, and testing can all be done in very fast iterations. Modern Java IDEs like IntelliJ, Eclipse, and NetBeans do most of the housekeeping, and streamline the entire endeavor.

If you look at the flourishing big data area, most of those technologies are written in Java: the open source software framework Hadoop and complementary technologies; the open source distributed database system Cassandra; and many others, such as the processing and querying library Cascalog. Java is an environment that you can control, tune, and understand. The tooling and profiling are excellent. Endless application areas are evolving because Java is getting the job done.

OPEN SOURCE
"From my point of view, the bulk of innovation today is being driven by open source communities and open source projects."

Java Magazine: You've previously represented the Apache Software Foundation as a JCP Executive Committee member, but now you're involved as an individual. Will you explore those different roles and experiences?

Magnusson: When I represented the Apache Software Foundation, my personal concerns were certainly in the mix. But at the end of the day, Apache's issues were first and foremost.

Now that I'm a JCP individual member, I want to represent people who are using Java daily to build innovative software, in open

SourcePoint is producing a set of tools that allow publishers to detect ad blockers, and react in the manner that they choose.

source, for internal use or as commercial products.

I have a very strong interest in ensuring that the Java ecosystem is a safe and compatible place for open source. From my point of view, the bulk of innovation today is being driven by open source communities and open source projects.

Years ago, there was significant trepidation in the legal departments of many companies about open source. But now I think the world has become fairly comfortable with it and the myriad advantages it delivers. Open source is just another dimension of the tools and resources that developers have at their disposal. The JCP has made significant progress in this area. But there is still more work to be done.

Java Magazine: As a voice for the practitioner within the JCP, what is the balance between establishing standards while still fostering and encouraging innovation?

Magnusson: Having a set of standards that mandate predictable behaviors is extremely important to me as a developer. It's essential for efficiently building reliable and maintainable systems. If I'm

using software from commercial entities such as Oracle or IBM, or open source projects that build to the standards, I can be assured that standard Java components will perform consistently. And that assurance is one of the things that make Java so compelling. As a developer, you have confidence that all the elements will interoperate smoothly.

In terms of innovation, there are a wide variety of implementations of the JVM that focus on various aspects of performance, system integration, monitoring, and so on. Many different players are in this space. But it's still a standard Java environment, guaranteeing predictable and reliable performance.

But not everything needs to be standardized. The Hadoop software library has established a de facto standard in the marketplace. There would be nothing to gain from making it a formal standard.

For the most part, the JCP has done a very good job in this area. And I think that's one of the reasons Java has remained so relevant during the past 20 years.

The formal standards process can slow down innovation, and sometimes that's a good thing—so that evolving technologies can be established as the right way to do things, not just the newest way.

Java Magazine: How can the JCP better serve the community going forward?

Magnusson: The most pressing issue right now is making JCP participation more interesting and accessible to a broad swath of developers in the ecosystem. Certainly technology vendors need continued representation, and some individuals currently have a voice and a role. But we need to increase the breadth of participation. We must find new ways to make the JCP more relevant in the day-to-day lives of developers.

> **JCP**
> "The most pressing issue is making JCP participation more interesting to developers."

Most software is written by developers at companies that are not in the software business—construction companies, banks, retailers, and so forth. We need to make sure that Java remains relevant for all of these businesses and their related application areas.

Java Magazine: Tell us a bit about your startup venture, and how Java technology is part of the picture.

Magnusson: The company is called SourcePoint, and it is focused on developing technology for web and mobile publishers.

In the near term, we're producing a set of tools that allow publishers to detect ad blockers, and react in the manner that they choose—for example, by reminding the user that the content they are consuming is paid for by advertising and asking the consumer to turn off the ad blocker. For the longer term, we are building technologies that enable publishers to engage in a deeper conversation with the consumer.

Regarding Java's role—it has been key. Our front-end web pages are implemented using JavaScript and Ruby on Rails. But virtually our entire back-end infrastructure is written in Java. Data collection from the internet is handled by Java servlets that route everything into a data pipeline written in Java and running on the JVM. Then we use custom software written in Java to process the data, and store the results in databases written in C and Java.

For a very long time, Java has been an indispensable tool in my development processes. And the JCP will ensure that it will continue to evolve and thrive. `</article>`

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

- Becoming an individual JCP member

Cofounder Adam Gibson, center, meets with Christian Nicholson, left, and George Takeo to go over Skymind features.

DEEP LEARNING

Startup Skymind looks to take deep learning technology out of the labs and into the enterprise with an open source framework built on Java and Hadoop.

BY PHILIP J. GILL

PHOTOGRAPHY BY BOB ADLER/GETTY IMAGES

Deep learning is a form of machine learning that seeks to enable intelligent devices of all shapes and sizes to emulate learning capabilities of the human brain.

Although such capabilities could offer enormous payoffs for big data and the Internet of Things—enabling

Like many tech startups in San Francisco, Skymind is headquartered in a former industrial building that has been converted to office use.

SKYMIND
skymind.io

Location:
San Francisco, California

Industry:
Software

Employees:
Four

Java technology used:
Java Development Kit 7

applications such as the driverless car—production-grade deep learning still remains largely confined to a few universities and corporate research labs. But Adam Gibson, cofounder and chief developer of a year-old San Francisco, California–based business intelligence and enterprise software startup called Skymind, hopes to make this technology more accessible.

"There's a huge gap between academia and the enterprise when it comes to the deep learning tools each deploys in such areas as big data and big data analytics to solve such problems as facial recognition, speech recognition, and similar applications," explains Gibson, 25. "Enterprise developers want a way to use deep learning technology on these big data problems and others, and to deploy it in the same production environment as their day-to-day operations."

With that goal in mind, the Skymind team has developed and released as open source technology the Deep Learning for Java framework, also called Deeplearning4J (DL4J), and made it available for free download at deeplearning4j.org. The company describes DL4J as the first commercial-grade, open source, distributed deep learning library written in Java and designed for business environments.

EMULATING THE BRAIN

Many common computing tasks today, such as spam filters, optical character recognition, and search engines, incorporate aspects of *machine learning*. "At a very high level, machine learning is driving advances in big data analytics," Gibson says.

But big data has also hit a wall, as certain unstructured datatypes—images, video, speech recognition, and text—that have the potential to offer enterprises large benefits remain beyond

The DL4J Framework

Deep Learning for Java (DL4J) is an open source framework for developing deep learning algorithms in Java for enterprise computing environments. This framework includes a variety of artificial neural networks (ANNs) for developers to deploy on a Java Virtual Machine (JVM), depending on what kind of data they want their deep learning application to understand. ANNs are statistical learning algorithms that loosely approximate the capabilities of biological neural networks, such as the central nervous system of animals—particularly the human brain.

There are several common types of ANNs. *Convolutional networks* are often used to learn images, while *recursive neural tensor networks* can be deployed in natural-language processing, text mining, and parsing sentences.

Deep-belief networks, which are simply stacked restricted Boltzmann machines, work with multiple datatypes. *Deep autoencoder networks* are useful for data compression, reducing the dimensionality of complex input. *Stacked denoising autoencoders* utilize a denoising technique that helps them generalize over new data, while *recurrent nets* are typically used to learn sound and times series.

Also included in the DL4J framework is N-Dimensional Arrays for Java (ND4J), a scientific computing library that brings NumPy and MATLAB-type data analysis tools to the JVM. NumPy is an extension to the Python programming language that supports multidimensional arrays and a large library of high-level math functions—while MATLAB, from MathWorks, is a multiparadigm numerical computing environment and fourth-generation programming language.

DL4J also supports Scala, a language that runs on the JVM.

the understanding of today's big data tools. Deep learning could potentially solve that problem. Also known as *deep structural learning*, this type of machine learning combines artificial neural networks (ANNs), pattern recognition, statistics, mathematical optimization, and other technologies associated with artificial intelligence.

In deep learning, a *signal* is relevant information that a deep learning algorithm running on an ANN can use to properly classify a piece of data and give it meaning—for example, recognizing facial features from a photo or a video, spoken words from a recording, or written words from a document or website. Everything else is the opposite: noise.

HISTORY

The first deep learning artificial neural network is believed to have been the neocognitron, developed by Kunihiko Fukushima in 1980.

"A deep learning algorithm automatically extracts a signal from data and solves problems with that data in a way that doesn't require preprogramming. The more signals the algorithm is able

DOMAIN NAME

The .io top-level domain is particularly popular with Silicon Valley startups these days, because it calls to mind the term input/output.

to strike from that data, the more it will be able to make inferences, just like in big data analytics. These are things that humans can do easily that are very difficult for computers. Deep learning algorithms teach computers to parse information automatically in much the same way humans do," says Gibson.

Most current deep learning algorithms and ANNs are written in other programming languages, such as Python. But for Gibson, Java was the natural choice for DL4J because of the prominent place it holds in the enterprise today. "Developing DL4J in Java makes us first-class citizens in the enterprise," he says. "We are able to use the same security mechanisms and operate in an environment that enterprise programmers already know."

DL4J is integrated with Hadoop and Spark—both de facto standards for big data production environments in the enterprise that are available from the Apache Software Foundation—whose 2.0 license Skymind is using for DL4J. Hadoop is written in Java, and Spark in Scala (a language that runs on the JVM). "Along with Java, Hadoop and Spark are the leading open source technologies in the enterprise, and they are all going to stick around," says Gibson.

BIG DATA AND BEYOND

Because of the broad number of neural network types that DL4J supports, the framework can be used as a platform to build applications to parse just about any datatype (see sidebar, "The DL4J Framework"). Skymind, like other open source companies before it, offers fee-based training, implementation, customization, and ongoing support services. Its major areas of focus include text analysis and images.

"Text analytics is a common office problem and something a lot of people come to us with," Gibson says. "A lot of companies have customer service logs

Gibson, Nicholson, and Takeo meet frequently to discuss product features and schedules.

5,000 and Counting

Since the first stable release of the Deep Learning for Java framework (DL4J) in November 2014, Skymind has seen 5,000 downloads from unique IP addresses, and the company's website gets roughly 21,000 visits per month, says Cofounder Adam Gibson.

"Our Google Group community has nearly 600 members," Gibson continues, "and more than 250 software engineers have forked our GitHub repository to create their own version to work on."

that they don't know what to do with, and they don't know how to extract a lot of signals from that data. For instance, they can't tell when customers are unhappy, because they can't mine the text to see if there are exclamation points that show customers are angry and excited."

The same algorithms that can give meaning to the customer service logs can also help improve another common text-mining chore: "Deep learning algorithms can automatically recognize that two words are used in a similar context. The algorithm can learn that, and thus help improve search results."

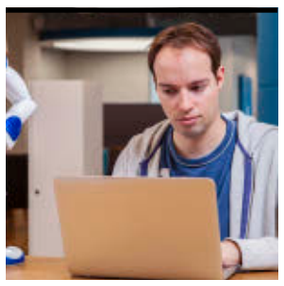
Facial recognition in particular has many important applications, especially in security. If someone walks up to a company entrance, for example, a deep learning–enabled system can use facial recognition algorithms to identify that person.

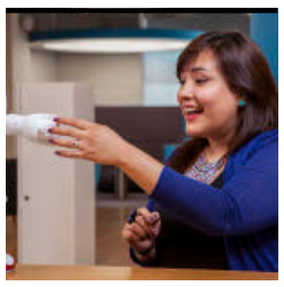
"If you want to do facial recognition for security purposes, the system will keep taking pictures until it figures out who the person is."

Deep learning algorithms, Gibson adds, are also essential to the success of many applications in the Internet of Things, in which virtually every machine or device will have some form of embedded intelligence that will read and parse signals in real time and on-the-fly, and communicate with other devices through the cloud.

Gibson concludes, "While the self-driving car is the 'Big One,' there will be many other Internet of Things applications in e-commerce and other areas that will rely on deep learning capabilities to understand and learn from a steady stream of incoming data." `</article>`

Philip J. Gill is a San Diego, California–based writer and editor who has followed Java technology since its inception.





ERWAN PINAULT AND THALÍA CRUZ

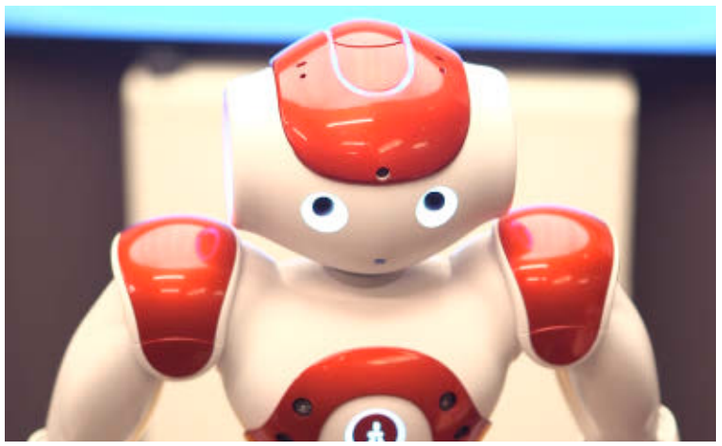
BIO

Robots Running Wild

Using the Java NAOqi SDK to program companion robots.

Aldebaran (SoftBank Group) designs humanoid and programmable robots. In less than 10 years, we've created three companion robots: NAO, Pepper, and Romeo.

These three humanoid robots run the same operating system: NAOqi, a GNU/Linux distribution based on Gentoo. This distribution contains all the required libraries and programs to make NAOqi the main software that gives life to Aldebaran's robots. In other words, NAOqi is a framework that allows you to create distributed robotic applications. These applications are normally written in Python or C++, but you can use other programming languages through the `libqi` library, which provides cross-platform and cross-language support. Thus, NAOqi can support other platforms (Windows, Linux, and Mac OS) and any language (with its corresponding bindings).



A robot responding to voice commands (video)

PHOTOGRAPHS COURTESY OF ALDEBARAN

The Java NAOqi SDK offers Java bindings and dedicated wrapper classes so you can use NAOqi easily. This SDK is available as a JAR file and can be used as any other third-party Java library is used.

The Java NAOqi SDK

This article shows how to create a new robot client application using the Java NAOqi SDK. The robot can listen to your orders and move accordingly: turn right or left, move forward, walk faster or slower, and stop.

To begin, download the JAR files from Aldebaran's Community website. (Please note that these JAR files are available for customers only.) There is one JAR file per platform and NAOqi version. The example in this article was done with NAOqi version 2.1.4, and we tested it on Linux x64 and Mac OS X. Download the JAR file that fits your platform and add it to your build path.

The first step is to create an `Application` in the `main` method, as shown in **Listing 1**. For this, you must pass your robot's IP address as an argument in the constructor of the `Application`. The `Application` is in charge of initializing the framework and connecting it to a `Session`, which connects services together, locally or over the network. When the `Application` is started, a `Session` is created and connected to your robot.

Once the `Session` is ready, you can get services. A service is an interface that exposes what the robot can do. Each robot has a `ServiceDirectory` that lists all its available services. For this example, you need the following services, as shown in **Listing 2**:

- `ALSpeechRecognition`: A service that gives the robot the ability to recognize

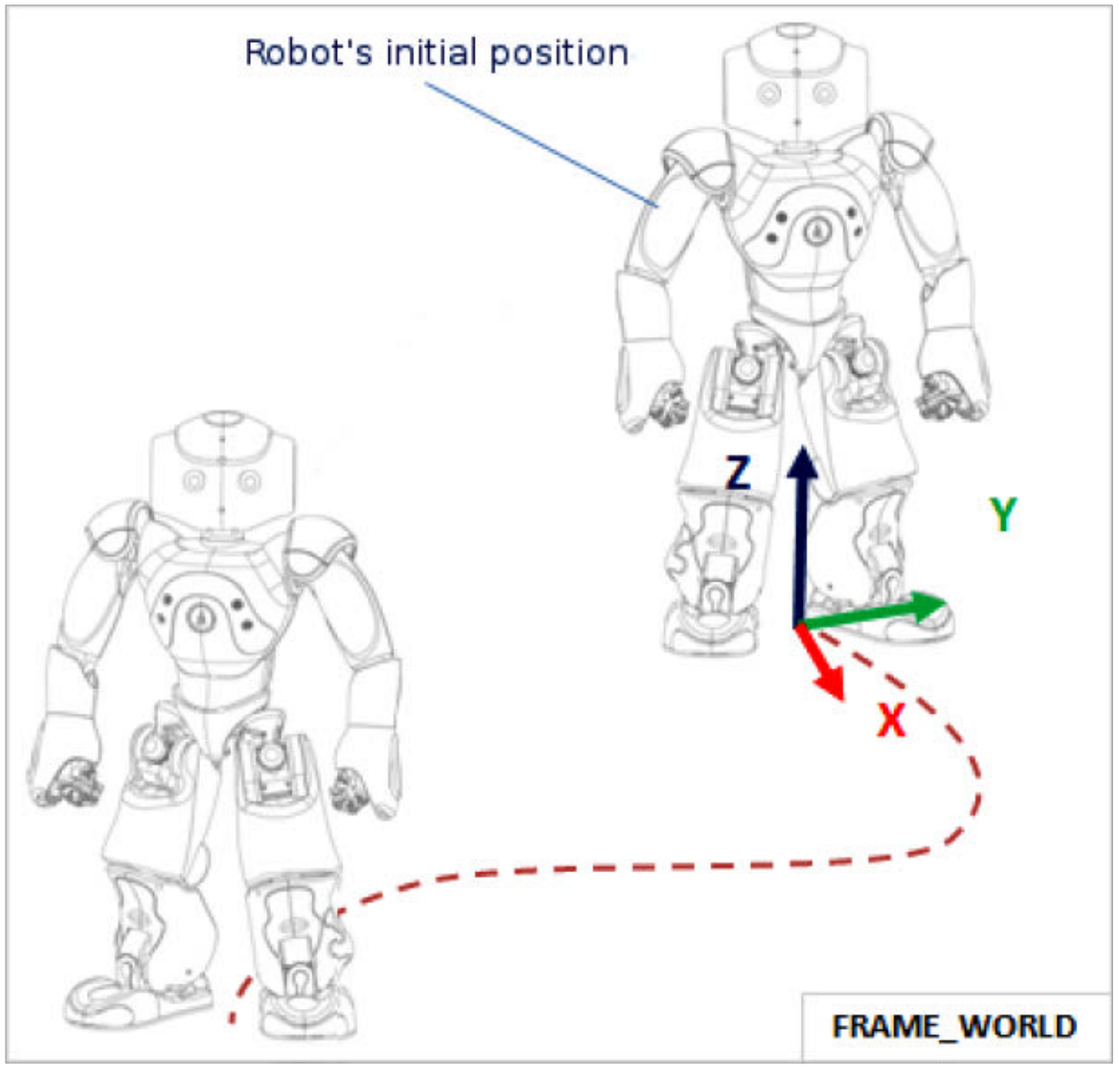


Figure 1

predefined words or phrases in several languages.

- `ALMemory`: A service that provides centralized memory used to store and retrieve key information related to the hardware configuration, which acts also as a hub for the distribution of event notifications.
- `ALMotion`: A service that facilitates making the robot move.
- `ALTextToSpeech`: A service that allows the robot to speak.
- `ALAutonomousMoves`: A service that enables subtle movements that the robot does autonomously.

Let's start by using `ALSpeechRecognition`. To make the robot listen and understand what you say, you must establish a vocabulary (see **Listing 3**). For this example, the vocabulary is a list of the possible directions that you can give to the robot. Leave "word spotting" disabled, because we want the robot to understand specific given words: nothing more, nothing less.

Then, you need to subscribe to the `ALSpeechRecognition` service, as shown in **Listing 4**, to enable the robot to listen to your voice inputs.

Each time the robot hears a word, the `ALMemory WordRecognized` event is raised. This event contains the recognized word and the estimated probability that this was, in fact, what you said. For handling this input, you need to subscribe to this event and set a callback, as shown in **Listing 5**. When you subscribe to an event, you get a subscription ID, so you can unsubscribe later.

Because the `EventCallback` is a functional interface, you can also use lambda expressions, as shown in **Listing 6**.

Each time the robot recognizes a word, you make the robot move according to the given order. To do so, use the `ALMotion` service, taking into account the following axis definitions. The X axis is positive toward the robot's front, the Y axis is from right to left, and the Z axis is vertical, as shown in **Figure 1**.

LISTING 1 | LISTING 2 | LISTING 3 | LISTING 4 | LISTING 5 | LISTING 6

```
public static void main(String[] args) throws Exception {
   String robotUrl = "tcp://nao.local:9559";
   // Create a new Application
   Application application =
       new Application(args, robotUrl);

   // Create a session and connect it to the robot.
   application.start();

   // Retrieve the created session.
   Session session = application.session();
...
```

Download all listings in this issue as text

Because you want the robot to move forward, turn left or right, go faster or slower, and stop, use the `moveToward(float x, float y, float theta)` method of `ALMotion` (see **Listing 7**), where

- `x` is the normalized, unitless velocity along the X axis. +1 and -1 correspond to the maximum velocity in the forward and backward directions, respectively.
- `y` is the normalized, unitless velocity along the Y axis. +1 and -1 correspond to the maximum velocity in the left and right directions, respectively.
- `theta` is the normalized, unitless velocity around the Z axis. +1 and -1 correspond to the maximum velocity in the counterclockwise and clockwise directions, respectively.

Because the robot moves autonomously while listening, disable "expressive listening" to reduce the fall-down risk. In effect, if the robot tries to follow one of your orders while moving autonomously, the mix of both movements might make it fall down. Therefore, add the code in **Listing 8** before enabling listening.

FRAMEWORK

NAOqi is a powerful framework that allows homogeneous programming, communication between different services, and information sharing.

For this example, you also want to subscribe to the `MiddleTactilTouched` event (*Tactile* is misspelled in the event's name), so that every time you touch the tactile sensor on the robot's head, the robot starts listening, if it was not already listening, or stops listening. See **Listing 9**.

The same can be done with lambda expressions, as shown in **Listing 10**.

Use the `ALTextToSpeech` service to make the robot give feedback about its state, for example, in the `disableListening()` method shown in **Listing 11**.

You're almost ready to start testing this Java application. Because you want the application to keep listening to the subscribed events, add the following lines to the `main` method, which blocks the main thread until the application is stopped:

```
// Keep the application run-
// ning.
application.run();
```

Thus, once the application is no longer running, it's important to unsubscribe from everything you were subscribed to and leave the robot in a "clean" state, as shown in **Listing 12**.

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

```
public void handleOrder(String order) throws CallError,
            InterruptedException {
        float x = 0;
        float y = 0;
        float theta = 0;
        if (order.equals("forward")) {
            x = 0.6f;
        } else if (order.equals("left")) {
            theta = 0.4f;
        } else if (order.equals("right")) {
            theta = -0.4f;
        } else if (order.equals("stop")) {
            x = 0f;
            y = 0f;
            theta = 0f;
        } else if (order.equals("faster")) {
            if (x > 0)
                x += 0.2;
            else
                x -= 0.2;
        } else if (order.equals("slower")) {
            if (x < 0)
                x += 0.2;
            else
                x -= 0.2;
        }
        motion.moveToward(x, y, theta);
}
```

Download all listings in this issue as text

Running Your Application

To better see what happens when you run this application, use the Choregraphe Memory watcher and Robot view so you can monitor the `ALMemory` events, what the robot says, and its moves.

1. First, connect your robot to Choregraphe, add the `MiddleTactilTouched` and

`WordRecognized` events to the Memory watcher, and enable the Robot view. Then, run your Java application. See **Figure 2**.

2. Touch the robot's middle tactile head sensor. The `MiddleTactilTouched` event is raised and the robot's eye LEDs turn navy blue (this change isn't visible in the Robot view). This means that the robot is listening to what you are saying.
3. Say "forward." The `WordRecognized` event is raised containing the word the robot heard and its probability, as shown in **Figure 3**. The robot moves forward.

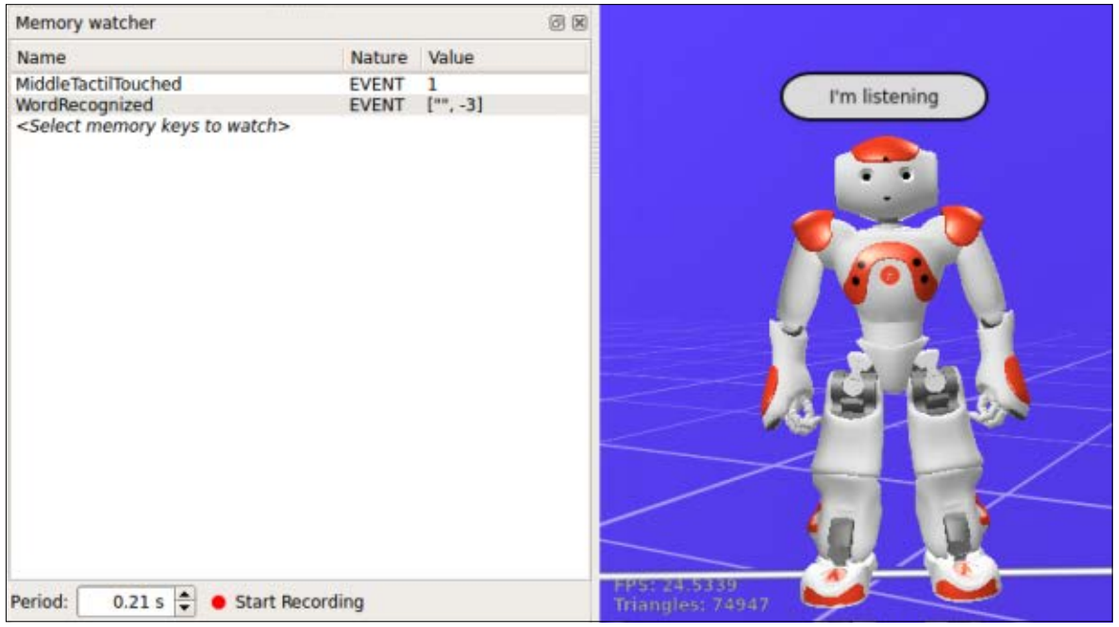


Figure 2

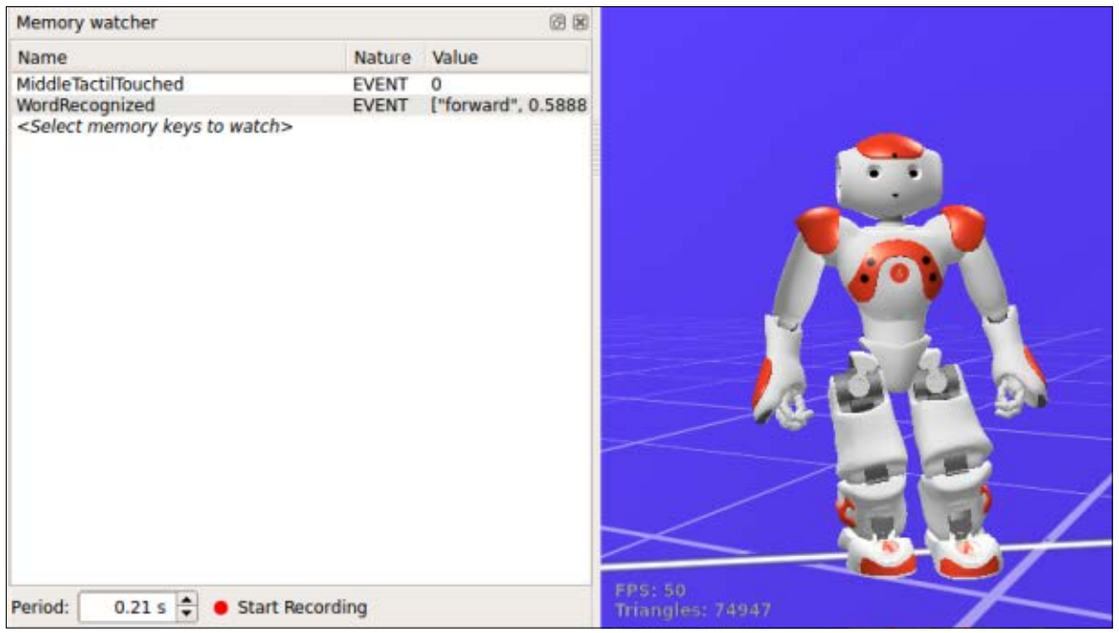


Figure 3

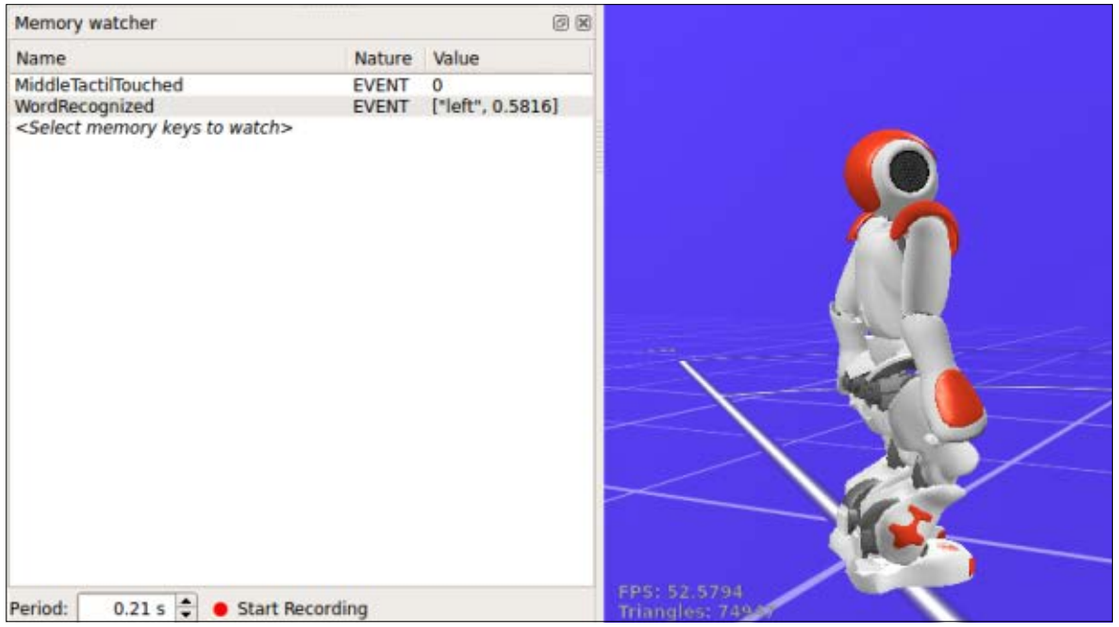


Figure 4

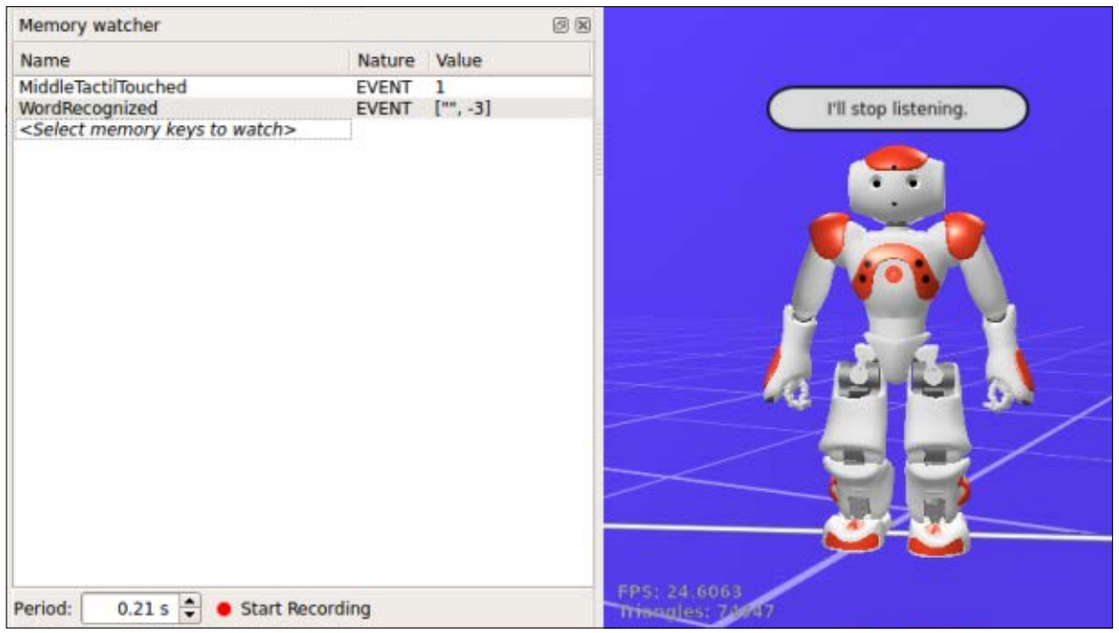


Figure 5

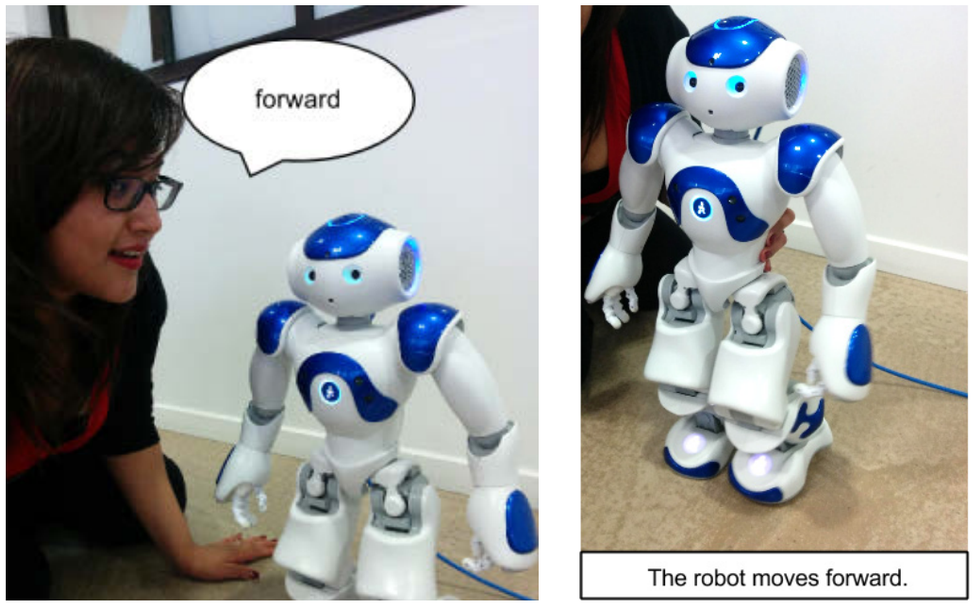


Figure 6

4. Say "left." The `WordRecognized` event is raised containing `left` and the robot turns left (see **Figure 4**).
5. Try it out with the other words contained in our vocabulary: "right," "faster," "slower," or "stop." If you say something else, the robot tries to match your input with one of the words it knows.
6. Touch the robot's middle tactile head sensor again. The robot stops moving and announces that it will stop listening (see **Figure 5**), and its eye LEDs return to their normal color.
7. Try saying one of the known words; the robot does nothing. To make it listen again, touch its middle tactile head sensor once again.

Everything seems to be working correctly. **Figure 6** is what it should really look like.

Creating a Service

For now, the code that you wrote can be accessed only by your current application. Let's say you want to reuse it and be able to call it from another application. For this, you need to create your own service.

First, create a new class that extends from `QiService`. In this case, call it `VoiceCommandService`, as shown in **Listing 13**. It contains your previously defined `handleOrder(String order)` method.

Then, as you did in the previous

LISTING 13

```
public class VoiceCommandService extends QiService {
    ALMotion motion;

    public VoiceCommandService(Session session) {
        motion = new ALMotion(session);
    }

    public void handleOrder(String order) throws CallError,
            InterruptedException {
        float x = 0;
        float y = 0;
        float theta = 0;
        if (order.equals("forward")) {
            x = 0.6f;
        } else if (order.equals("left")) {
            theta = 0.4f;
        } else if (order.equals("right")) {
            theta = -0.4f;
        } else if (order.equals("stop")) {
            x = 0f;
            y = 0f;
            theta = 0f;
        } else if (order.equals("faster")) {
            if (x > 0)
                x += 0.2;
            else
                x -= 0.2;
        } else if (order.equals("slower")) {
            if (x < 0)
                x += 0.2;
            else
                x -= 0.2;
        }
        motion.moveToward(x, y, theta);
    }
}
```

Download all listings in this issue as text

section, create a new `Application`, connect it to your robot, and retrieve the created `session`, as shown in **Listing 14**. This `Application` is in charge of creating and registering your `VoiceCommandService` for the `session`. The sevice is registered by using a `DynamicObjectBuilder`, which makes the service compatible with the other supported languages. This same object allows it to advertise its methods by specifying their signature and a brief description. When registering the service to the session, you can give it any name—for example, `VoiceCommand`.

Last, create another `Application`, whose goal is to demonstrate that you can call the advertised methods of your `VoiceCommand` service. For this, you need to create an `AnyObject` object, which functions as a bridge between this new `Application` and your service. See **Listing 15**.

Let's test it:

1. Run `MyRegisterService Application`. As long as this application is running, you'll be able to call the methods contained in your service.
2. Run `ServiceConsumer Application` (without stopping the first application).

The robot starts moving forward, then turns to the left, and finally stops. It works!

The `VoiceCommand` service is also accessible from applications written in other supported programming languages. For example, in Python it would look as shown in **Listing 16**. The same example can also be written in C++.

Conclusion

NAOqi is a powerful framework that allows homogeneous programming, communication between services, and information sharing. This example shows how a desktop Java application can interact with services, mainly written in C, and control the robot as it would do for any other supported language: C++, Python, or JavaScript.

The Java NAOqi SDK is still a work in progress. We want to make it more Java friendly. We know that we still have a long way to go, and we will continue improving the Java bindings because we believe our robots have even greater potential if Java is well supported. We hope that eventually Java will be directly integrated into our robots. `</article>`

LEARN MORE

- Aldebaran's website
- Javadoc
- Java NAOqi SDK documentation
- libqi library
- libqi-java on GitHub

LISTING 14 LISTING 15 / LISTING 16

```
public class MyRegisterServiceApplication {

   public static void main(String[] args) throws Exception
{

        Application application = new Application(args,
                "tcp://nao.local:9559");

        application.start();
        Session session = application.session();

        VoiceCommandService vcService =
              new VoiceCommandService(session);

        // Create a DynamicObjectBuilder, that will make
        // your service compatible with other supported
        // languages.
        DynamicObjectBuilder objectBuilder =
            new DynamicObjectBuilder();
        vcService.init(objectBuilder.object());

        // Advertise the handleOrder method contained in
        // your VoiceCommandService service.
        // You need to specify its signature.
        objectBuilder.advertiseMethod("handleOrder::v(s)",
          vcService,
           "Robot will move according to the given order:
            forward, left, right, faster, slower or stop");
        // Register the service as "VoiceCommand"
        session.registerService("VoiceCommand",
        objectBuilder.object());

        application.run();
    }
}
```

Download all listings in this issue as text

//contact us /

Comments.

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

Article Proposals.

We welcome article proposals on all topics regarding Java the language; the Java Virtual Machine (JVM); languages that run on the JVM that would be of interest to Java developers; discussion of specific Java utilities, preferably open source; and programming algorithms implemented in Java. Articles should be in Microsoft Word or .odt format and should run between 1,200 and 2,000 words, not including code. Listings should be short; if necessary, break out snippets of 30 lines or fewer and make the rest available as a download.

Customer Service.

If you're having trouble with your subscription, please contact the folks at java@halldata.com (phone +1.847.763.9635), who will do whatever they can to help.

Where?

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

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