Java™ magazine

By and for the Java community

// JULY/AUGUST 2014 /

THE JAVA VIRTUAL MACHINE

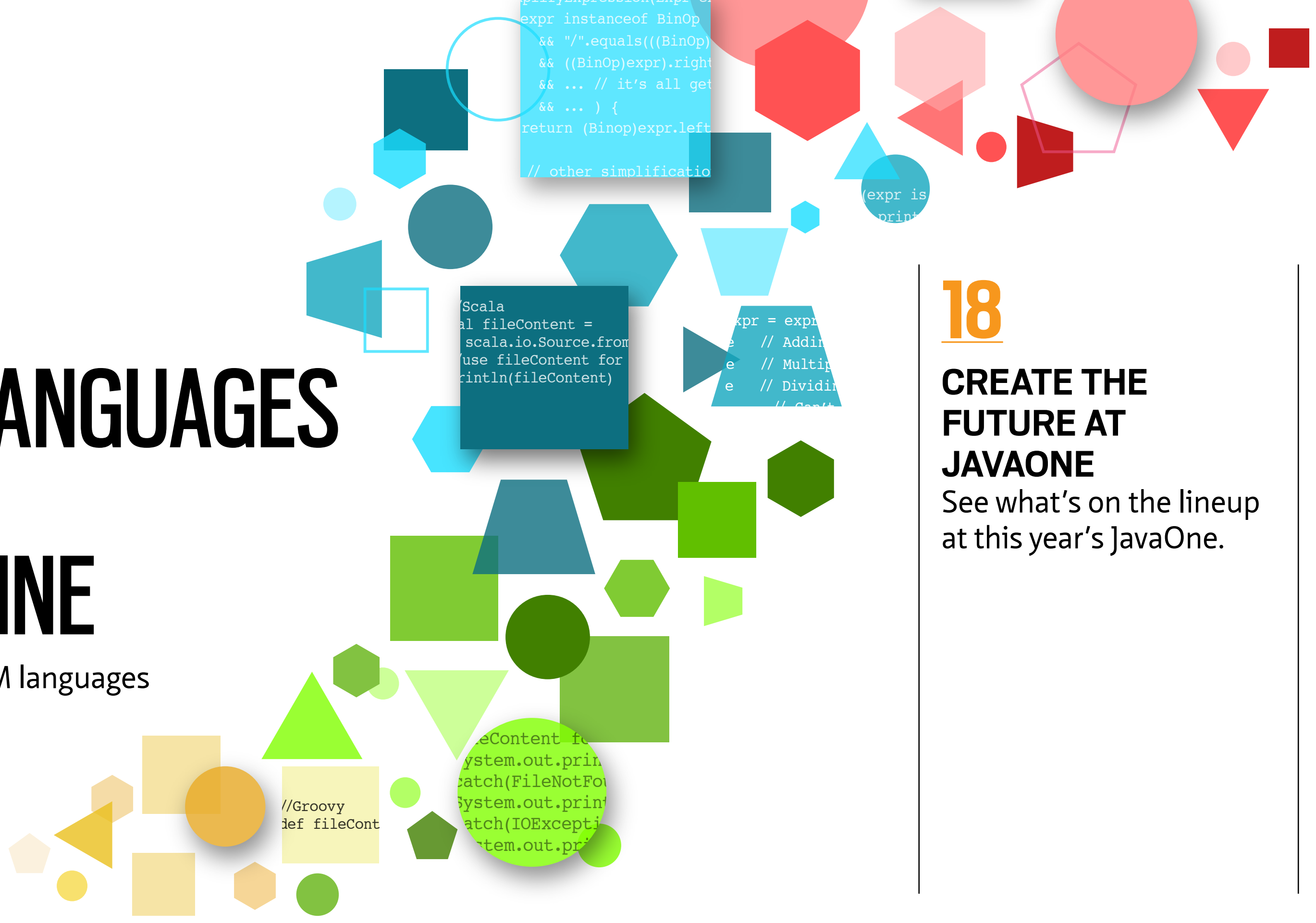
ALTERNATIVE LANGUAGES

Explore the flexibility of the JVM with the language of your choice

ORACLE.COM/JAVAMAGAZINE

ORACLE®

//table of contents /



COVER ART BY I-HUA CHEN

EDITORIAL

Editor in Chief
Caroline Kvitka
Community Editor
Yolande Poirier
Java in Action Editor
Michelle Kovac
Technology Editor
Tori Wieldt
Contributing Writer
Kevin Farnham
Contributing Editors
Claire Breen, Blair Campbell, Kay Keppler, Karen Perkins

DESIGN

Senior Creative Director
Francisco G Delgadillo
Senior Design Director
Suemi Lam
Design Director
Richard Merchán
Contributing Designers
Jaime Ferrand, Arianna Pucherelli
Production Designers
Sheila Brennan, Kathy Cygnarowicz

PUBLISHING

Publisher
Jennifer Hamilton +1.650.506.3794
Associate Publisher and Audience Development Director
Karin Kinnear +1.650.506.1985

ADVERTISING SALES

President, Sprocket Media
Kyle Walkenhorst +1.323.340.8585
Western and Central US, LAD, and Canada, Sprocket Media
Tom Cometa +1.510.339.2403
Eastern US and EMEA/APAC, Sprocket Media
Mark Makinney +1.805.709.4745
Advertising Sales Assistant
Cindy Elhaj +1.626.396.9400 x 201
Mailing-List Rentals
Contact your sales representative.

RESOURCES

Oracle Products
+1.800.367.8674 (US/Canada)
Oracle Services
+1.888.283.0591 (US)
Oracle Press Books
oraclepressbooks.com

ARTICLE SUBMISSION

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

SUBSCRIPTION INFORMATION

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

MAGAZINE CUSTOMER SERVICE

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

PRIVACY

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

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

Digital Publishing by GTxcel

JavaOne™

SEPTEMBER 28 – OCTOBER 2, 2014
SAN FRANCISCO
oracle.com/javaone

REGISTER NOW

Respond Early, Save $400*

Discover All Things Java

- Learn all about Java 8
- 500+ educational sessions
- 100+ product demos
- Meet with 500+ Java experts

Diamond Sponsor
IBM

Silver Sponsor
AVAYA

Bronze Sponsors
GridGain IN-MEMORY COMPUTING
JFrog
New Relic.
Restlet

*Save $400 over onsite registration. Visit oracle.com/javaone/register for early bird registration dates and details.

ORACLE®

While often overshadowed by the Java language, the Java Virtual Machine (JVM) is the cornerstone of the Java platform. It gives Java its hardware and operating system independence, small compiled code size, and protection from malicious programs. We've dedicated almost an entire issue to the JVM, so if you haven't paid much attention to it lately, buckle up and get ready for the ride.

Java Virtual Machine

One of the most notable things about the JVM is that it can run hundreds of programming languages besides the Java language—providing developers with flexibility and freedom of choice. In our cover story, "Alternative Languages for the JVM," Raoul-Gabriel Urma takes a look at eight JVM languages and some of the benefits of using them. In "Performing 10 Routine Operations Using Different JVM Languages," Venkat Subramaniam further explores the benefits of alternative languages—while also showing how Java SE 8 handles certain tasks. Developing JavaFX applications? Check out Josh Juneau's article, "JavaFX with Alternative Languages," to discover how alternative languages and custom APIs can speed development.

We also focus on things you can do to make the JVM work better for you. Julien Ponge shares advice on avoiding benchmarking pitfalls on the JVM, and Cas Saternos gets us up to speed on server-side deployment on the JVM. In addition, Marcus Hirt introduces us to a profiling and diagnostics tool for Java applications running on the JVM and Ben Evans gives us a primer on just-in-time compilation.

Want more? Make sure to register for JavaOne, September 28–October 2 in San Francisco, California, which features a dedicated JVM Languages track (and check out our conference preview). Hope to see you there!

Caroline Kvitka, Editor in Chief BIO

//send us your feedback /

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

PHOTOGRAPH BY BOB ADLER

Java magazine

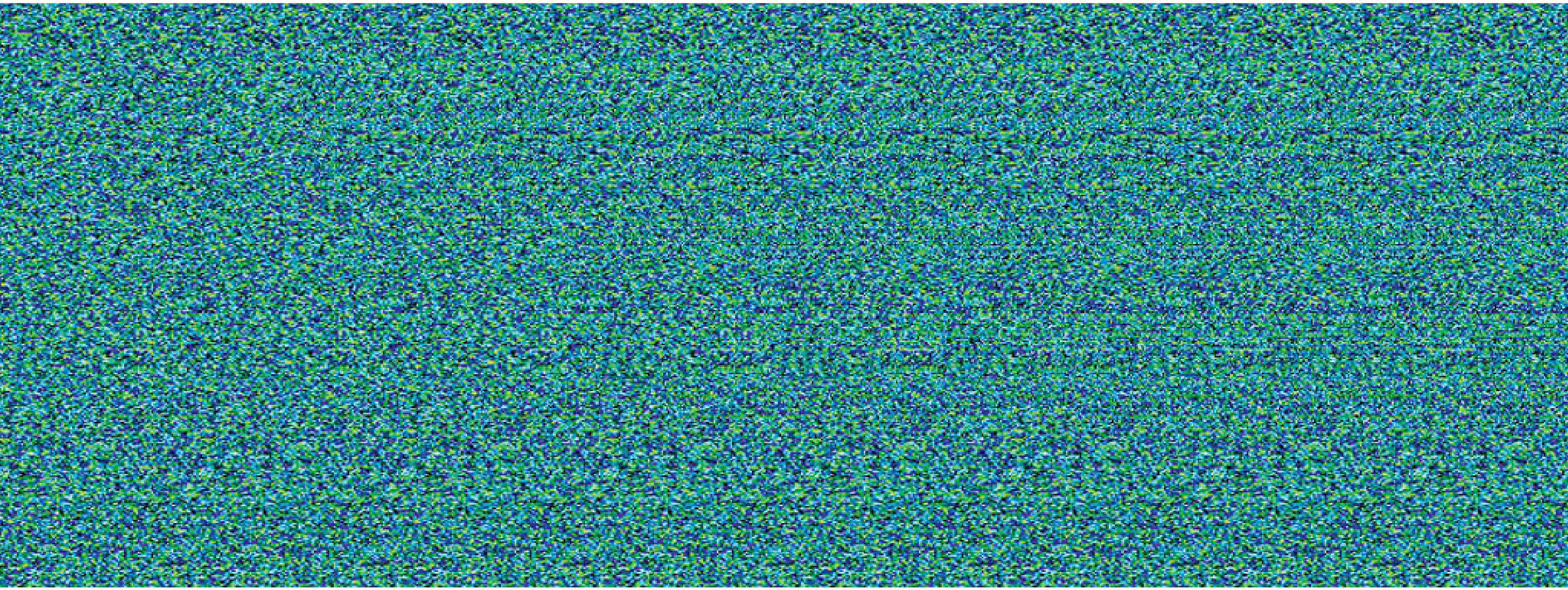
Find the Most Qualified Java Professionals for your Company's Future

Introducing the *Java Magazine* Career Opportunities section – the ultimate technology recruitment resource.

Place your advertisement and gain immediate access to our audience of top IT professionals worldwide including: corporate and independent developers, IT managers, architects and product managers.

For more information or to place your recruitment ad or listing contact:
tom.cometa@oracle.com

The answer is right in front of you



Java Image Enabling SDKs that Help You See the Big Picture

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

Atalasoft JoltImage

Click for tips on viewing the stereogram

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

Atalasoft A Kofax Company

www.atalasoft.com

ALTERNATIVE LANGUAGES FOR THE JVM

A look at eight features from eight JVM languages

BY RAOUL-GABRIEL URMA

The Java Virtual Machine (JVM) isn't just for Java anymore. Several hundred JVM programming languages are available for your projects. These languages ultimately compile to bytecode in class files, which the JVM can then execute. As a result, these programming languages benefit from all the optimizations available on the JVM out of the box.

The JVM languages fall into three categories: They have features that Java doesn't have, they are ports of existing languages to the JVM, or they are research languages.

Java Virtual Machine

ART BY I-HUA CHEN

The first category describes languages that include more features than Java and aim to let developers write code in a more concise way. Java SE 8 introduced lambda expressions, the Stream API, and default methods to tackle this issue of conciseness. However, developers love many other features—such as collection literals, pattern matching, and a more sophisticated type inference—that they can't find in Java yet. The languages we'll look at in this first category are Scala, Groovy, Xtend, Ceylon, Kotlin, and Fantom.

The second category is existing languages that were ported to the JVM. Many languages, such as Python and Ruby, can interact with Java APIs and are popular for scripting and quick prototyping. Both the standard implementation of Python (CPython) and Ruby (Ruby MRI) feature a global interpreter lock, which prevents them from fully exploiting a multicore system. However, Jython and JRuby—the Python and Ruby implementations on the JVM—get rid of this restriction by making use of Java threads instead. (You can read more about JRuby and JRubyFX in this issue's "JavaFX with Alternative Languages" article by Josh Juneau. Juneau also covers Jython extensively on his blog.)

Another popular language ported to the JVM is Clojure, a dialect of Lisp, which we'll look at in this article. In addition, Oracle recently released Nashorn, a project that lets you run JavaScript on the JVM.

The third category is languages that implement new research ideas, are suited only for a specific domain, or are just experimental. The language that we'll look at in this article, X10, is designed for efficient programming for high-performance parallel computing. Another language in this category is Fortress from Oracle Labs, now discontinued.

For each language we examine, one feature is presented to give you an idea of what the language supports and how you might use it.

1 | SCALA

Scala is a statically typed programming language that fuses the object-oriented model and functional programming ideas. That means, in practice, that you can declare classes, create objects, and call methods just like you would typically do in Java. However, Scala also brings popular features from functional programming languages such as pattern matching on data structures, local type inference, persistent collections, and tuple literals.

The fusion of object-oriented and functional features lets you use the best tools from both worlds to solve a particular problem. As a result, Scala often lets programmers express algorithms more concisely than in Java.

Feature focus: pattern matching. To illustrate, take a tree structure that you would like to traverse. **Listing 1** shows a simple expression language consisting of numbers and binary operations.

Say you're asked to write a method to simplify some expressions. For example "5 / 1" can be simplified to "5." The tree for this expression is illustrated in **Figure 1**.

In Java, you could deconstruct this tree representation by using instanceof, as shown in **Listing 2**. Alternatively, a common design pattern for separating an algorithm from its domain is the visitor design pattern, which can allevi-

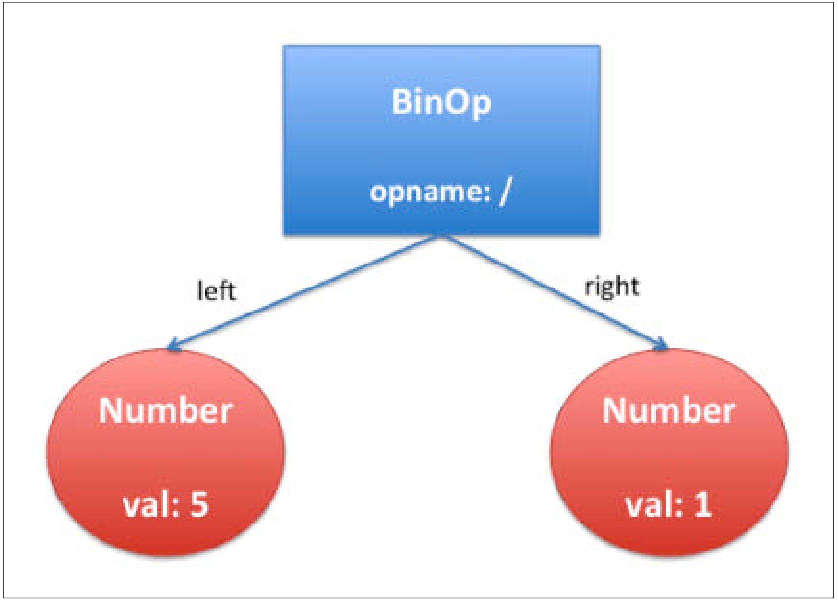


Figure 1

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

```
[Java]

class Expr { ... }
class Number extends Expr { int val; ... }
class BinOp extends Expr { String opname; Expr left, right; ... }
```

Download all listings in this issue as text

ate some of the verbosity. See **Listing 3**.

However, this pattern introduces a lot of boilerplate. First, domain classes need to provide an accept method to use a visitor. You then need to implement the "visit" logic.

In Scala, the same problem can be tackled using pattern matching. See **Listing 4**.

THREE TYPES
The JVM languages fall into three categories: They have features that Java doesn't have, they are ports of existing languages to the JVM, or they are research languages.

2 | GROOVY

Groovy is a dynamically typed object-oriented language. Groovy's dynamic nature lets you manipulate your code in powerful ways. For example, you can expand objects at runtime (for example, by adding fields or methods).

However, Groovy also provides optional static checking, which means that you can catch errors at compile time (for example, calling an undefined method will be reported as an error before the program runs, just as in Java). As a result, programmers who feel that they are more productive without types getting in their way can embrace Groovy's dynamic nature. Nonetheless, they can also opt to gradually use static checking later if they wish. In addition, Groovy is friendly to Java programmers because almost all Java code is also valid Groovy code, so the learning curve is small.

Feature focus: safe navigation. Groovy has many features that let you write more-concise code compared to Java. One of them is the *safe navigation operator*, which prevents a NullPointerException. In Java, dealing with null can be cumbersome. For example, the following code might result in a NullPointerException if either person is null or getCar() returns null:

```
Insurance carInsurance =
person.getCar().getInsurance();
```

To prevent an unintended NullPointerException, you can be defensive and add checks to prevent null dereferences, as shown in **Listing 5**.

However, the code quickly becomes ugly because of the nested checks, which also decrease the code's readability. The safe navigation operator, which is represented by ?., can help you navigate safely through potential null references:

```
def carInsurance =
person?.getCar()?.getInsurance()
```

In this case, the variable carInsurance will be null if person is null, getCar() returns null, or getInsurance() returns null. However, no NullPointerException is thrown along the way.

3 | CLOJURE

Clojure is a dynamically typed programming language that can be seen as a modern take on Lisp. It is radically different from what object-oriented programmers might be used to. In fact, Clojure is a fully functional programming language, and as a result, it is centered on immutable data structures, recursion, and functions.

LISTING 6 / LISTING 7 / LISTING 8 / LISTING 9

```
[Clojure]

(unless false (println "ok!!") (println "boo!!"))
; prints "ok!!"

(if false (println "boo!!") (println "ok!!"))
; prints "ok!!"
```

Download all listings in this issue as text

Feature focus: homoiconicity. What differentiates Clojure from most languages is that it's a *homoiconic* language. That is, Clojure code is represented using the language's fundamental datatypes—for example, lists, symbols, and literals—and you can manipulate the fundamental datatypes using built-in constructs. As a consequence, Clojure code can be elegantly manipulated and transformed by reusing the built-in constructs.

Clojure has a built-in if construct. It works like this. Let's say you want to extend the language with a new construct called unless that should work like an inverted if. In other words, if the condition that is passed as an argument evaluates to false, Clojure evaluates the first branch. Otherwise—if the argument evaluates to true—Clojure evaluates the second branch. You should be able to call the unless construct as shown in **Listing 6**.

To achieve the desired result you can define a macro that transforms a call to unless to use the construct if, but with its branch arguments reversed (in other words, swap the first branch and the second branch). In Clojure, you can manipulate the code representing the branches that are passed as an argument as if it were data. See **Listing 7**.

In this macro definition, the symbol branches consists of a list that contains the two expressions representing the two branches to execute (println "boo!!" and println "ok!!"). With this list in hand, you can now produce the code for the unless construct. First, call the core function reverse on that list. You'll get a new list with the two branches swapped. You can then use the core function conj, which when given a list, adds the remaining arguments to the front of the list. Here, you pass the if operation together with the condition to evaluate.

4 | KOTLIN

Kotlin is a statically typed object-oriented language. Its main design goals are to be compatible with Java's API, have a type system that catches more errors at compile time, and be less verbose than Java. Kotlin's designers say that Scala is a close choice to match its design goals, but they dislike Scala's complexity and long compilation time compared to Java. Kotlin aims to tackle these issues.

Feature focus: smart casts. Many developers see the Java cast feature as annoying and redundant. For an example, see **Listing 8**.

Repeating the cast to Number shouldn't be necessary, because within the if block, expr has to be an instance of Number. The generality of this technique is called *flow typing*—type information propagates with the flow of the program.

Kotlin supports *smart casts*. That is, you don't have to cast the expression within the if block. **See Listing 9**.

5 | CEYLON

Red Hat developed Ceylon, a statically typed object-oriented language, to give Java programmers a language that's easy to learn and understand (because of syntax that's similar to Java) but less verbose. Ceylon includes more type system features than Java. For example, Ceylon supports a construct for defining type aliases (similar to C's typedef; for example, you could define Strings to be an alias for List<String>), flow typing (for example, no need to cast the type of an expression in a block if you've already

LISTING 10 / LISTING 11 / LISTING 12

```
[Java]

List<Integer> numbers = IntStream.rangeClosed(1, 10).mapToObj(
x -> x * 2).collect(toList());
```

Download all listings in this issue as text

done an instanceof check on it), union of types, and local type inference. In addition, in Ceylon you can ask certain variables or blocks of code to use dynamic typing—type checking is performed at runtime instead of compile time.

Feature focus: for comprehensions. for comprehensions can be seen as syntactic sugar for a chain of map, flatMap, and filter operations using Java SE 8 streams. For example, in Java, by combining a range and a map operation, you can generate all the numbers from 2 to 20 with a step value of 2, as shown in **Listing 10**.

In Ceylon, it can be written as follows using a for comprehension:

```
List<Integer> numbers =
[for (x in 1..10) x * 2];
```

Here's a more-complex example. In Java, you can generate a list of points in which the sum of the x and y coordinates is equal to 10. See **Listing 11**.

Thinking in terms of flatMap and map operations using the Stream API might be overwhelming. Instead, in Ceylon, you can write more simply, as done in the code shown in **Listing 12**, which produces [(1, 9), (2, 8), (3, 7), (4, 6), (5, 5), (6, 4), (7, 3), (8, 2), (9, 1)].

The result: Ceylon can make your code more concise.

6 | XTEND

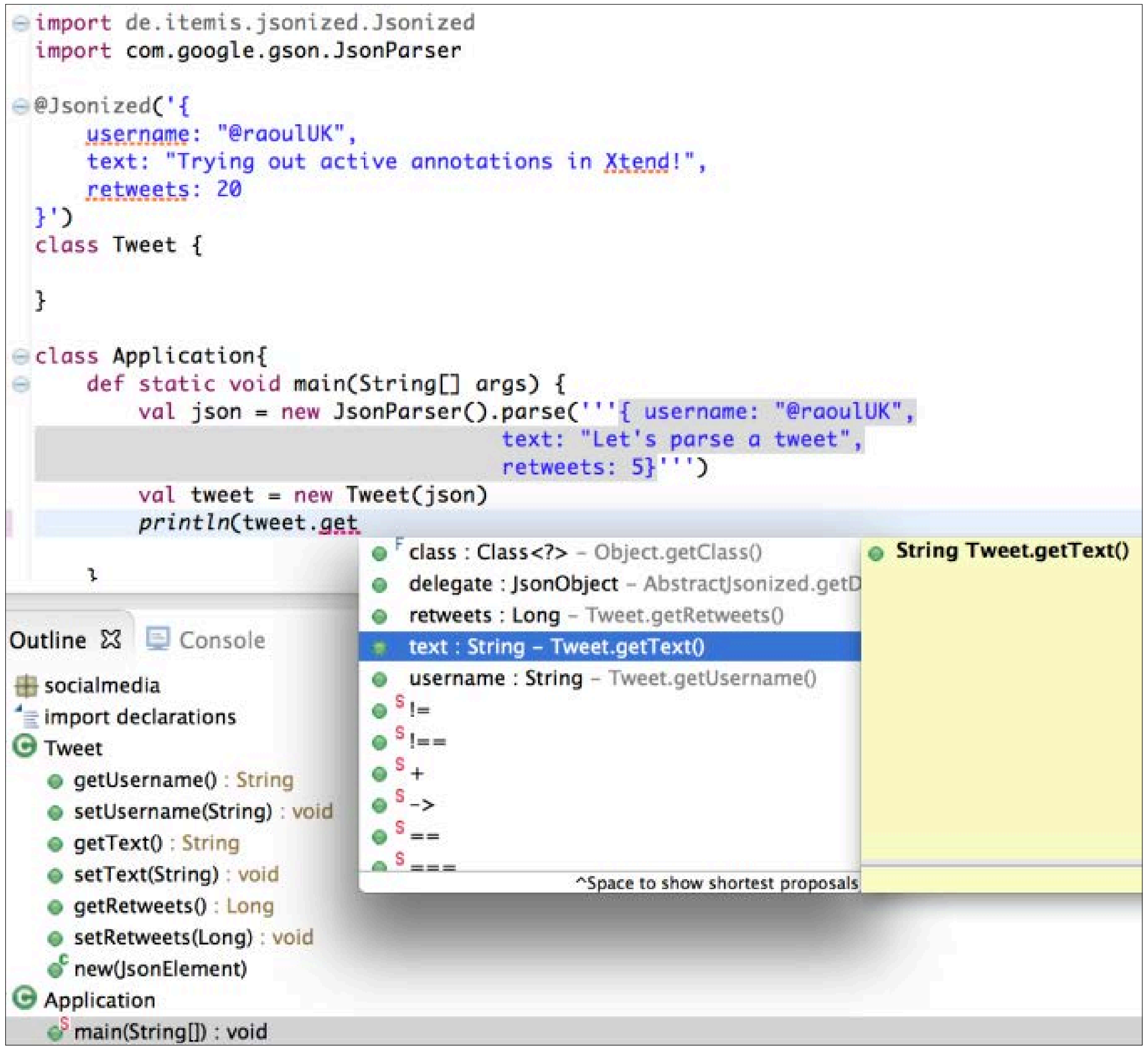
Xtend is a statically typed object-oriented language. One way it differs from other languages is that it compiles to pretty-printed Java code rather than bytecode. As a result, you can also work with the generated code.

Xtend supports two forms of method invocation: default Java dispatching and multiple dispatching. With multiple dispatching, an overloaded method is selected based on the runtime type of its arguments (instead of the traditional static types of the arguments, as in Java). Xtend provides many other popular features available in other languages such as operator overloading and type inference.

One unique feature is template expressions, which are a convenient way to generate string concatenation (similar to what template engines provide). For example, template expressions support control-flow constructs such as IF and FOR. In addition, special processing of white space allows templates to be readable and their output to be nicely formatted.

Feature focus: active annotations. Xtend provides a feature called active annotations, which is a way to do compile-time metaprogramming. In its simplest form, this feature allows you to generate code transparently, such as adding methods or fields to classes with seamless integration in the Eclipse IDE for example. New fields or methods will show up as members of the modified classes within the Eclipse environment. More-advanced use of this feature can generate a skeleton of design patterns such as the visitor or observer pattern. You can provide your own way to generate code using template expressions.

Here's an example to illustrate this feature in action. Given sample JSON data, you can automatically generate a domain class in your Xtend program that maps JSON properties into members. The Eclipse IDE will recognize these members, so you can use features such as type checking and autocompletion. All you have to do is wrap the JSON sample within an @Jsonized annotation. **Figure 2** shows an example within the Eclipse IDE using a JSON sample representing a tweet.



```
import de.itemis.jsonized.Jsonized
import com.google.gson.JsonParser

@Jsonized('{
    username: "@raoulUK",
    text: "Trying out active annotations in Xtend!",
    retweets: 20
}')
class Tweet {

}

class Application{
    def static void main(String[] args) {
        val json = new JsonParser().parse('''{ username: "@raoulUK",
                                              text: "Let's parse a tweet",
                                              retweets: 5}''')
        val tweet = new Tweet(json)
        println(tweet.get
    }
```

```
class : Class<?> - Object.getClass()
delegate : JsonObject - AbstractJsonized.getD
retweets : Long - Tweet.getRetweets()
text : String - Tweet.getText()
username : String - Tweet.getUsername()
!=
!==
+
->
==
===
^Space to show shortest proposals
```

```
String Tweet.getText()
```

```
Outline    Console
socialmedia
import declarations
Tweet
    getUsername() : String
    setUsername(String) : void
    getText() : String
    setText(String) : void
    getRetweets() : Long
    setRetweets(Long) : void
    new(JsonElement)
Application
    main(String[]) : void
```

Figure 2

7 | FANTOM

Fantom is an object-oriented language featuring a type system that takes an alternative view compared to most other established, statically typed languages. First, it differentiates itself by not supporting user-defined generics. However, three built-in classes can be parameterized: List, Map, and Func. This design decision was made to let programmers benefit from the use of generics (such as working with collections—see the link to an empirical study conducted by Parnin et al. in "Learn More") without complicating the overall type system. In addition, Fantom provides two kinds of method invocations: one that goes through type checking at compile time (using a dot notation: .) and one that defers checking to runtime (using an arrow notation: ->).

Feature focus: immutability. Fantom encourages immutability through language constructs. For example, it supports const classes—once created, an instance is guaranteed to have no state changes. Here's how it works. You can define a class Transaction prefixed with the const keyword:

```
const class Transaction {
  const Int value
}
```

The const keyword ensures that the class declares only fields that are immutable, so you won't be able to modify the field named value after you instantiate a Transaction. This is not much different than declaring all fields of a class final in Java. However, this feature is particularly useful with nested structures. For example, let's say the Transaction class is modified to support another field of type Location. The compiler ensures that the location field can't be reassigned and that the Location class is immutable.

For instance, the code in **Listing 13** is incorrect and will produce the error Const field 'location' has non-const type 'hello_0::Location'. Similarly, all classes extending a const class can be only const classes themselves.

8 | X10

X10 is an experimental object-oriented language that IBM developed. It supports features such as first-class functions and is designed to facilitate efficient programming for high-performance parallel computing.

To this end, the language is based on a programming model called the *partitioned global address space*. In this model, each process shares a global address space, and slices of this space are allocated as private memory for local data and access. To work with this model, X10 offers specialized built-in language constructs to work with concurrency and distributed execution.

Compared to popular object-oriented languages, a novel feature in its type system is support for *constraint types*. You can think of constraint types as a form of contracts attached to types. What makes this useful is that errors are checked statically, eliminating the need for more-expensive runtime checks. For example, one possible application of constraint types is to report out-of-bound array accesses at compile time.

Feature focus: constraint types. Consider a simple Pair class, with a generated constructor:

```
class Pair(x: Long, y: Long){}
```

You can create Pair objects as follows:

```
val p1 : Pair = new Pair(2, 5);
```

However, you can also define explicit constraints (similar to contracts) on the properties of a Pair at use-site. Here, you want to ensure that p2 holds only symmetric pairs (that is, the values of x and y must be equal):

```
val p2 : Pair{self.x == self.y}
= new Pair(2, 5);
```

Because x and y are different in this code example, the assignment will be reported as a compile error. However, the following code compiles without an error:

```
val p2 : Pair{self.x == self.y}
= new Pair(5, 5);
```

CONCLUSION

In this article, we examined eight features from eight popular JVM languages. These languages provide many benefits, such as enabling you to write code in a more concise way, use dynamic typing, or access popular functional programming features.

I hope this article has sparked some interest in alternative languages and that it will encourage you to check out the wider JVM ecosystem. </article>

Acknowledgements. I'd like to thank Alex Buckley, Richard Warburton, Andy Frank, and Sven Efftinge for their feedback.

LISTING 13

```
[Fantom]

const class Transaction {
  const Int value
  const Location location := Location("Cambridge")
}
class Location{
  Str city
  new make(Str city) { this.city = city }
}
```

Download all listings in this issue as text

MORE ON TOPIC:

Java Virtual Machine

Raoul-Gabriel Urma started his PhD in computer science at the University of Cambridge at the age of 20. He is a coauthor of *Java 8 in Action: Lambdas, Streams, and Functional-Style Programming* (Manning Publications, 2014). In addition, he has given more than 20 technical talks at international conferences. He holds a MEng degree in computer science from Imperial College London and graduated with first-class honors, having won several prizes for technical innovation.

LEARN MORE

- "Java Generics Adoption: How New Features Are Introduced, Championed, or Ignored"

Clockwise from left: the Internet of Things demo wall; the DIY demo area; the bull's-eye toss

MADE FOR MAKERS

Java was in full swing May 17–18, 2014, in San Mateo, California, at Maker Faire, a festival of invention, creativity, and resourcefulness—and a celebration of the Maker Movement. At the Java booth, faire attendees learned how they could create the future with Java. The booth included a do-it-yourself (DIY) demo area, a bull's-eye toss, and an Internet of Things (IoT) demo wall. In addition, ORACLE TEAM USA crew member **Brad Webb** was on hand to talk about the America's Cup and sign autographs next to a replica of the winning sailboat.

PHOTOGRAPHS BY RON SELLERS

Clockwise from top: MakerCon's innovation showcase; dozens of makers displaying their products; Oracle's Jeremy Ashley; Maker Media's Dale Dougherty; an attendee checking out a maker's device

PHOTOGRAPHS BY RON SELLERS

In the DIY area, faire attendees got hands-on with Java on workstations connected to a Raspberry Pi with a dual-sensor board that could detect proximity and luminosity. Using a simple icon-based application, users created "if-this-then-that" actions to turn on a light, move a robotic arm, turn on a table fan, send a tweet or text, take a picture, and more.

In the bull's-eye area, attendees threw foam stress balls at a smart target powered by a Raspberry Pi. If they hit the target, a sound played and a camera connected to the Raspberry Pi took a picture and uploaded it to a photo wall.

The demo wall included an IoT Java panel and a Minecraft Java IoT 4-D cube. The IoT Java panel used a collection of development boards running Oracle Java SE Embedded and Oracle Java ME Embedded to control lights using a guitar or a flute, ask about sensor values using Twitter and Google Voice, and call a phone number to turn on a lamp or a popcorn machine. The 4-D cube was integrated with Minecraft to make TNT explode in the game and to change the background and brightness of the Minecraft world.

MakerCon, a two-day conference and workshop for those at the forefront of the Maker Movement, was held at Oracle headquarters in Redwood Shores, California, May 13–14, 2014. The conference connected experts in digital manufacturing, technology and tools providers, accelerators that facilitate taking a prototype to market, and a broad swath of makers. Conference themes included tools of innovation, the IoT, the business side of making, and community building. An innovation showcase was also held the first evening of the event, with more than two dozen makers displaying their products and devices in a casual atmosphere. Keynote speakers included Oracle's **Jeremy Ashley** on designing new enterprise experiences for the IoT; Arduino's **Massimo Banzi**, who updated the audience on new Arduino boards and cloud services; and Autodesk's **Carl Bass**, who announced an Autodesk 3-D printer.

DEVOXX FRANCE: BORN TO BE

Devoxx France, part of the Devoxx family of conferences, took place in Paris April 16–18, 2014. Organized by the Paris Java User Group, the event attracted 1,400 developers to hundreds of sessions about Java SE 8, Java EE, Java Virtual Machine (JVM) languages, future and upcoming technologies, agile and DevOps methodologies, startups and innovations, mobile, and infra-structure—especially cloud, big data, and NoSQL.

Keynote speakers reflected on this year's theme of "Born to Be" a developer in the digital era. They invited developers to embrace a strategic and central role in this revolution and to see themselves as makers of it.

Several new activities were added to the lineup of Birds-of-a-Feather (BOF) sessions, conference sessions, and hands-on labs. New this year were the future lab and hackathons, where developers programmed a JavaFX game, flew the Crazyflie Quadcopter, watched the Nao Robot (shown at left with Oracle's **Stephen Chin**) dance, and checked out home automation applications and a robotic xylophone.

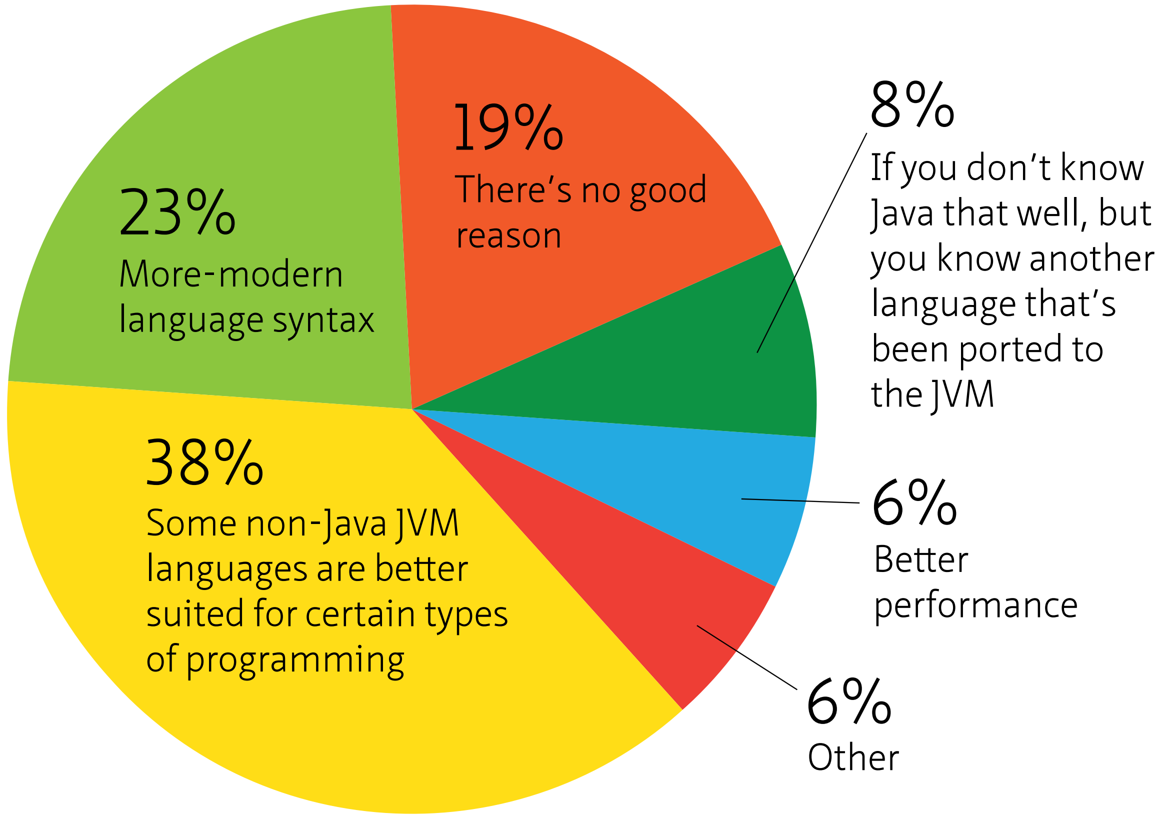
Devoxx France Organizer Antonio Goncalves discusses the activities planned for the conference.

PHOTOGRAPH BY YOLANDE POIRIER

JAVA.NET POLL

Why Use a Non-Java JVM Language?

In a recent Java.net poll, Java and Java Virtual Machine (JVM) developers highlighted multiple reasons for using a non-Java JVM language. A total of 181 votes were cast, as developers responded to the prompt **"The best reason to use a non-Java JVM language is . . . ."** Here are the results:



One voter who selected "Other" said, "I think it is a set of options, and not a single one," and listed modern syntax, proper support for type inference, reified generics, and value types as some top reasons.

While it's likely that most of those who selected "There's no good reason" probably haven't spent that much time investigating non-Java JVM languages, the remaining selections hint at the diversity of features and capabilities that modern JVM languages provide.

Java Tour 2014

The Java Tour 2014 is a series of events (77 and counting) for Java developers held around the world. The Oracle Events team is producing the tour in coordination with Oracle Technology Network and Java user groups (JUGs). Oracle Java evangelists and Java Champions provide top-notch content and a chance to network with other developers. The Java Tour 2014 allows both Oracle and the Java community to address local, regional-specific Java topics. You'll hear the latest on Java 8, Java EE, and embedded Java. Check the tour page for dates and cities.

JAVA CHAMPION PROFILE

CHRISTIAN ULLENBOOM

***Christian Ullenboom** is a Java trainer, a best-selling author, and a blogger from Germany. He was selected to be a Java Champion in September 2005.*

Java Magazine: Where did you grow up?

Ullenboom: I grew up in a small town in Germany—about 8,000 people. But can you really call a guy who is still fascinated about a jumping deforming sphere on a C64 grown up?

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

Ullenboom: I was born in the 1970s, the beginning of the home computer era. My first computer was a C64 that had no external storage. Without the ability to save any programs, I had to type them over and over again, and so I learned programming.

Java Magazine: What was your first professional programming job?

Ullenboom: In my youth, I did a lot of performance-related Assembler demo programming (for fun, not for cash), mainly on Amiga, some on a SPARCstation at the university. After I moved from Assembler to C++, I got a part-time job while studying, and was programming sewing machines.

Java Magazine: What do you like about the Java ecosystem?

Ullenboom: The enormous range of open source libraries and frameworks makes Java a great choice for programming. Although I did some Python development—and I love this language, too—I never saw a compelling reason to move from Java to another language or runtime.

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

Ullenboom: I like to combine my work as an author with traveling around the globe. I'm writing this after a bus trip from north Luzon [Philippines] to Manila, during which I updated my book on Java 8 features while my darling was sleeping.

Java Magazine: Has being a Java Champion changed anything for you with respect to your daily life?

Ullenboom: Not in terms of stalkers or groupies. I became a Java Champion in 2005 because of my books on Java, which are known to almost every German-speaking Java developer. In 2013 "Java Is an Island Too" [translated title] was the most sold computer book of all IT books in Germany. However, because my books and also my blog are written in my mother language only, I am almost unknown internationally.

Java Magazine: What are you looking forward to in the coming years?

Ullenboom: Over the last years, my collection of home computers and game consoles has become very comprehensive, and I am in the process of setting up a 2,000-square-meter building for a museum where people can play with the good old stuff. Donations are welcome.

You can find Christian Ullenboom on Twitter (@javabuch).

EVENTS

JavaOne 2014 *SEPTEMBER 28–OCTOBER 2*
SAN FRANCISCO, CALIFORNIA

Top Java experts from around the world gather for this weeklong conference that includes practical hands-on content. Track topics this year range from the stronger-than-ever core Java platform to in-depth and timely explorations of Java and security, Java and the cloud, and other mission-critical Java topics. Tracks will also focus on the tools and techniques that help create outstanding user experiences that can be delivered through a variety of channels, including personal devices, smartcards, embedded environments, and intelligent equipment.

PHOTOGRAPH BY JOE PARKS AT FLICKR

OSCON
JULY 20–24
PORTLAND, OREGON
The Open Source Convention (OSCON) is an immersive five days of all things open source: new and innovative projects, major enterprisewide deployments, and—from icons of the open source movement—deep perspective on where we've been and where we're headed.

JVM Language Summit
JULY 28–30
SANTA CLARA, CALIFORNIA
The 2014 JVM Language Summit is an open technical collaboration among language designers, compiler writers, tool builders, runtime engineers, and virtual machine (VM) architects.

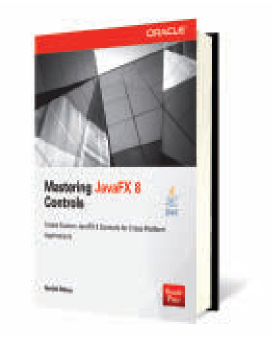
The Developer's Conference
AUGUST 3–9
SÃO PAULO, BRAZIL
One of Brazil's largest developer conferences, this event offers 35 tracks, several of them exclusively on Java.

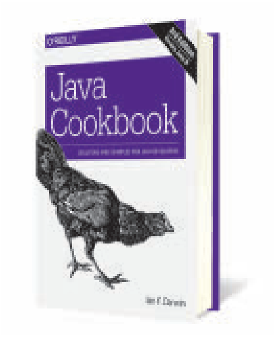
JCrete
AUGUST 25–29
CHANIA, GREECE
JCrete is a Java unconference for Java experts and evangelists and Java Champions. It combines the benefits of an open-spaces conference with Greek hospitality and seaside location.

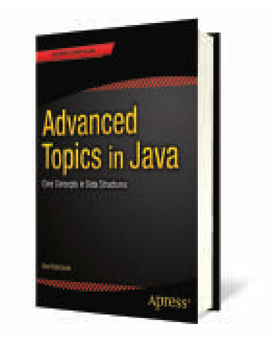
JCertif
SEPTEMBER 8–14
BRAZZAVILLE, CONGO
JCertif is the biggest IT community event in central Africa. International speakers present talks and labs about Java technologies, web apps, cloud apps, and more.

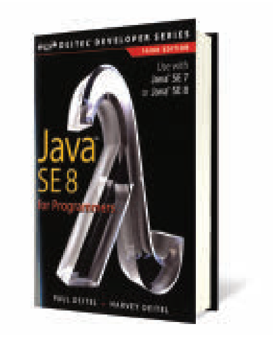
JavaZone
SEPTEMBER 9–11
OSLO, NORWAY
The conference offers a combination of technical talks and panels in an informal atmosphere with an expected attendance of more than 2,500. JavaZone 2014 will be the 13th consecutive JavaZone conference.

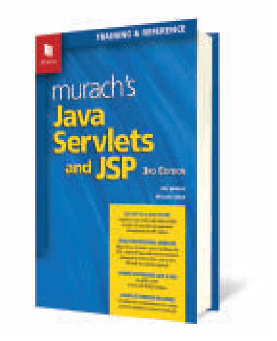
JAVA BOOKS











MASTERING JAVAFX 8 CONTROLS

By Hendrik Ebbers
Oracle Press, July 2014

This Oracle Press guide shows you how to create custom JavaFX 8 controls and master the development of rich clients and huge applications. It introduces JavaFX controls and the basic JavaFX APIs to handle them. It then reviews available controls and provides clear instructions on how to alter them as well as how to create new, custom controls specified to the user's needs. Developers new to JavaFX and those ready to start advanced work will benefit from this book.

JAVA COOKBOOK

By Ian Darwin
O'Reilly Media, January 2014

Java Cookbook is a comprehensive collection of common problems, solutions, and practical examples for anyone programming in Java. This third edition replaces irrelevant information, such as RMI and applets, with recipes updated for Java 8. The book covers a lot of ground and offers Java developers short, focused pieces of code that can be easily incorporated into other programs. The recipes focus on things that are useful, tricky, or both.

ADVANCED TOPICS IN JAVA

By Noel Kalicharan
Apress, January 2014

Advanced Topics in Java teaches concepts that any budding software developer should know. You'll delve into topics such as sorting, searching, merging, recursion, random numbers, and simulation, among others. Increase the range of problems you can solve by manipulating versatile and popular data structures such as binary trees and hash tables. This book assumes that you have a working knowledge of basic programming concepts such as variables, constants, assignment, selection, and looping.

JAVA SE 8 FOR PROGRAMMERS, THIRD EDITION

By Paul Deitel and Harvey Deitel
Prentice Hall, March 2014

Written for programmers with a background in high-level language programming, this book applies the Deitel signature live-code approach to teaching programming and explores the Java language and Java APIs in depth. The book presents concepts in the context of fully tested programs, complete with syntax shading, code highlighting, line-by-line code walkthroughs, and program outputs.

MURACH'S JAVA SERVLETS AND JSP

By Joel Murach and Michael Urban
Murach, June 2014

Now in its third edition, this Java web programming book will help you to master the many interrelated concepts and skills that you need to create web applications using servlets and JavaServer Pages (JSPs). Early chapters cover using servlets and JSPs and taking advantage of HTML5 and CSS3. The book also covers JavaBeans, the JSP Expression Language, the JSP Standard Tag Library (JSTL), database programming, JavaMail, listeners and filters, and much more.

PHOTOGRAPH BY GETTY IMAGES

CREATE THE FUTURE AT JavaOne

The conference returns to San Francisco, September 28–October 2.

By Tom Caldecott

What draws developers from around the world to San Francisco in September? JavaOne, of course. The technology talks are always a main attraction. This year, topics will range from the stronger-than-ever core Java platform, to in-depth and timely explorations of Java and security, Java and the cloud, and other mission-critical Java topics. Tracks will also focus on the tools and techniques that help create outstanding user experiences that can be delivered through a variety of channels, including personal devices, smart cards, embedded environments, and intelligent equipment. "This year, we have an Internet of Things track to help developers interact with the physical world using Java," says Stephen Chin, JavaOne content chair. "We're also adding an agile development track, and we have a huge focus on Java 8."

Other main attractions include product introductions, demos, networking, and fun. In addition, Devoxx4Kids Bay Area is hosting an event on the Saturday before JavaOne. "It's a great way to get the younger generation interested in technology," says Chin.

And, of course, people will come to JavaOne for the speakers, including Jim Manico, Hendrik Ebbers, Rebecca Parsons, and David Blevins. Read what these technology experts have to say about their upcoming sessions and what they're looking forward to at the conference. Get their advice on how to make the most of your JavaOne 2014 experience.

MEET A SPEAKER

JIM MANICO

Jim Manico (@manicode) is an author and educator of developer security–awareness training. He is a frequent speaker on secure software practices, a global board member for the Open Web Application Security Project (OWASP), and a member of the JavaOne Rock Star Wall of Fame.

Java Magazine: Why focus on security?

Manico: Application and software security is my passion. It's the ultimate puzzle and challenge. It's also something that is really important right now. We're in the golden age of application-layer hacking, and we are seeing a radically increased need for developers to care about and learn more about secure coding.

Java Magazine: What's a big misconception about security?

Manico: The security community tells developers that if you do input validation right—if you validate data, and you say it's good data—your application is secure. There are cases where data that's been validated is still incredibly dangerous to the application, requiring deeper defenses in addition to input validation. Another important misconception is that traditional security professionals are the right folks to be securing your applications. Having a traditional security professional tell developers to "do security" is often like developers telling security professionals to "ship a product." I feel we need to close this gap between security teams and developer teams.

Java Magazine: What aspect of security will you address in your JavaOne talk?

Manico: I'll be speaking about the top coding techniques and essential tools, including several Oracle, OWASP, Apache, and Google open source Java projects that will help developers build low-risk, high-security applications.

Java Magazine: What advice do you have for developers attending JavaOne?

Manico: Go to as many tracks as you possibly can! The amount of knowledge you can gain in a short time is really off the charts.

Java Magazine: How do you spend your time at JavaOne?

Manico: I go to a lot of the security tracks that talk about different ways to defend Java applications. This year, I want to go to the new Java 8 talks that discuss the changes in the various security APIs—including the new cipher suites being offered and better random number generation—which I think are awesome. Random number generation is the heart of all cryptography. So, I'm really excited to see all the enhancements in Java 8.

Don't miss Manico's session, "Third-Party Java Libraries for Secure Development."

Get on Track

What's your interest when it comes to development and Java? Security? Java EE? Core Java platform? JavaFX 8? Tools? Cloud? Agile? Java and the Internet of Things? We have you covered at JavaOne 2014.

Check out this year's tracks:

- Clients and UI
- Core Java Platform
- Internet of Things
- JVM Languages
- Java and Security
- Tools and Techniques
- Server-Side Java
- Java in the Cloud
- Agile Development

Learn more about tracks and register for JavaOne.

MEET A SPEAKER

HENDRIK EBBERS

Hendrik Ebbers is senior Java architect at Materna GmbH in Dortmund, Germany, focusing on research and development, Swing, JavaFX, middleware, and DevOps. He is also founder and leader of the Java User Group Dortmund and just wrote the book Mastering JavaFX 8 Controls *(Oracle Press, 2014). Follow Ebbers on his blog and Twitter @hendrikEbbers.*

Java Magazine: What made you decide to write a book?

Ebbers: First of all, Oracle Press asked me. I blog, write articles, and talk at international conferences on JavaFX 8, so I thought the next step should be a book. It introduces JavaFX controls and the basic JavaFX APIs to handle them. It also reviews available controls and provides instructions on how to alter them and create new, custom controls.

Java Magazine: What will your JavaOne session cover?

Ebbers: I will discuss how developers can use JavaFX in production and in enterprise and business applications. They'll get an overview of the various best practices on solving problems such as client/server communications, asynchronous versus synchronous tasks, MVC/MVP [model-view-controller/model-view-presenter] framework–patterned approaches, and how to design complex dialog window flows.

Java Magazine: What is your advice to developers attending JavaOne? What should they look for, and what should they expect?

Ebbers: There will be some very cool JavaFX, community, open source, and Internet of Things talks this year. The talks are always very professional; I have never experienced a bad one. So don't miss the talks.

Java Magazine: What are you looking forward to at the conference?

Ebbers: I love all things around JavaOne. There are a lot of people whom I see only one or two times a year. JavaOne is the best place to meet with people because everyone is there. The conference has interesting topics. And I like seeing San Francisco and going to parties. All that stuff.

Don't miss Ebbers' session, "JavaFX Enterprise."

Get Your Fix

Get **COFFEE** without leaving the vicinity of the conference. This list of java joints is arranged by walking time from JavaOne at the Hilton Union Square.

Starbucks at the Hilton Union Square
333 O'Farrell Street

Taylor Street Coffee Shop
375 Taylor Street
2-minute walk
Although it offers only plain coffee and no specialty drinks, it also serves a hearty breakfast.

Café Encore
488 Post Street
3-minute walk

Barbary Coast
55 Cyril Magnin Street
4-minute walk

PHOTOGRAPH BY GREENCOLANDER/CC BY

MEET A SPEAKER

REBECCA PARSONS

Dr. Rebecca Parsons is chief technology officer at ThoughtWorks, a provider of software delivery, pioneering tools, and consulting services. She has more than 20 years of application development experience in industries ranging from telecommunications to emergent internet services. She has written articles for language and artificial intelligence publications and is chair of the Agile Alliance Board. Follow Parsons on her blog.

Java Magazine: ThoughtWorks has an unusual mission: "To better humanity through software and help drive the creation of a socially and economically just world." How is that achieved?

Parsons: Our core competence is software development. We were one of the very early adopters of agile, and we're really the first to do agile on a large scale. In terms of our mission, we make our services available not only to Fortune 50 companies globally but also to NGOs [nongovernmental organizations], hospitals, and other organizations in emerging economies. I'm personally committed to helping with IT capacity development in those areas.

Java Magazine: What will be the mission for your JavaOne talk?

Parsons: The title is "Principles of Evolutionary Architecture." An adaptable architecture is critical to allow systems to respond to change without the need to predict the future. This is an area that doesn't get much attention. But it's a tension point in a lot of organizations. It's about getting architecture and development to work more closely together. I'll talk about various strategies to allow an architecture to evolve as the understanding of an organization's business problem evolves.

Java Magazine: What is your advice to developers attending JavaOne?

Parsons: I think they ought to come with an open mind to learn new things, to connect with the people who go to JavaOne, and to connect with the different components of the Java ecosystem. It's a community, and JavaOne can help them see how they can get more deeply embedded in that community.

Java Magazine: And what will you do at JavaOne?

Parsons: Talk with people and learn new things. I have an insatiable appetite for learning.

Don't miss Parsons' session, "Principles of Evolutionary Architecture."

Get Your Fix

Café Madeleine
43 O'Farrell Street
5-minute walk
This café is known for its mochas, which are made with real ganache.

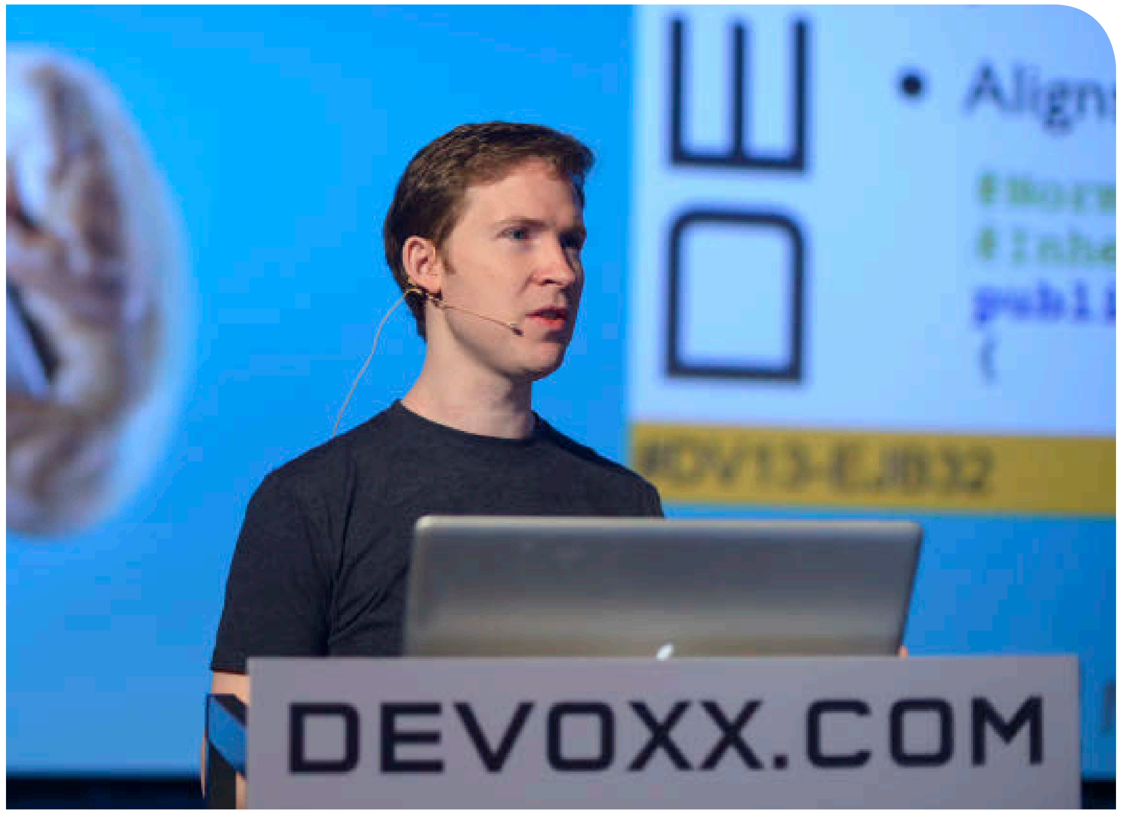
Sugar Café
679 Sutter Street
7-minute walk
Try the iced caramel macchiato on a warm day.

Blue Bottle Coffee
66 Mint Plaza
7-minute walk

The Coffee Bean & Tea Leaf
773 Market Street
7-minute walk
Try the lightest roast black coffee.

—Curran Mahowald

PHOTOGRAPH BY KENNYMATIC/CC BY



MEET A SPEAKER

DAVID BLEVINS

David Blevins is a founder of the Apache TomEE, OpenEJB, and Geronimo projects. In 2013, he cofounded Tomitribe, an open source Java EE company that focuses on Apache TomEE. He is a member of the EJB 3.2 and Java EE 7 Expert Groups. Follow Blevins on his blog.

Java Magazine: You're giving two talks at JavaOne. With one, you appear to be taking on the role of Java EE champion/promoter.

Blevins: A lot of people checked out of Java EE in the 2006 time frame and haven't looked back. When I'm presenting on TomEE, I often find that there are several changes that we've made to Java EE that most people aren't aware of.

Java Magazine: So that inspired the talk "Java EE Game Changers"?

Blevins: Yes. I want to get people up to speed on the major high-level, macro-level changes we've made to realign Java EE to what is relevant. And I want to get people involved and up to speed on where it could go and really inspire them to participate in that.

Java Magazine: And what will your second talk cover?

Blevins: The title is "Apache TomEE, Java EE Web Profile and More on Tomcat." I'm one of the creators of Apache TomEE. I'll discuss where TomEE comes from, how it works, and what advantages a Tomcat developer can get by using TomEE.

Java Magazine: What advice do you have for JavaOne attendees?

Blevins: Look at the sessions, get inspired, act on that inspiration, and connect with the speakers. Connect with the other attendees and become part of the community. That's the best part about [it].

Java Magazine: Why do you attend?

Blevins: I love to talk with the people. It's about sitting down in person and having a conversation; getting that level of interaction is not possible online. That's a special aspect of JavaOne.

Don't miss Blevins' sessions, "Java EE Game Changers" and "Apache TomEE, Java EE Web Profile and More on Tomcat."

Tom Caldecott is a writer in Oracle Brand Communications.

Quench Your Thirst

Check out the eclectic **BAR SCENE** in San Francisco.

The Buena Vista Café
2765 Hyde Street
Try the famous Irish coffee, which was purportedly invented here.

Bourbon and Branch
501 Jones Street
Experience a speakeasy from the Prohibition Era (make reservations to get the password).

Martuni's
4 Valencia Street
Sip a martini and sing along with the piano player.

Jasper's Corner Tap
401 Taylor Street
Want a beer? They have 18 on tap.

—*Curran Mahowald*

PHOTOGRAPH BY GETTY IMAGES

From left to right: VNIIRA's Alexandre Teterin, Anton Fedorov, and Mikhail Kuznetsov at the Pulkovo-1 Airport in Saint Petersburg, Russia

PHOTOGRAPHY BY ARTEM LEZHEPEKOV/GETTY IMAGES

JAVA GETS ITS WINGS

Russian research institute creates custom air traffic control system with NetBeans and Java. **BY DAVID BAUM**

The Boeing 737 made a wide turn above the expansive Siberian landscape and began its final descent into Talakan Airport. A rousing cheer went up from the air traffic control crew as the flight from Surgut, Russia, touched down with 26 people on board. It was the first passenger flight to land at Talakan, a privately constructed airport in the Sakha Republic of Russia.

The successful landing was an especially proud moment for Anton Fedorov, chief of software development at the Scientific Research Center of Air Traffic Management, who spearheaded the development of the airport's Java-based air traffic control system. "For me and my team, it was the culmination of a four-year effort that resulted in a customized system called SINTEZ-KSA ATC," Fedorov says.

Fedorov and his team have had plenty of experience with air traffic control systems in their work for the Scientific Research Center of Air Traffic Management, a division of the

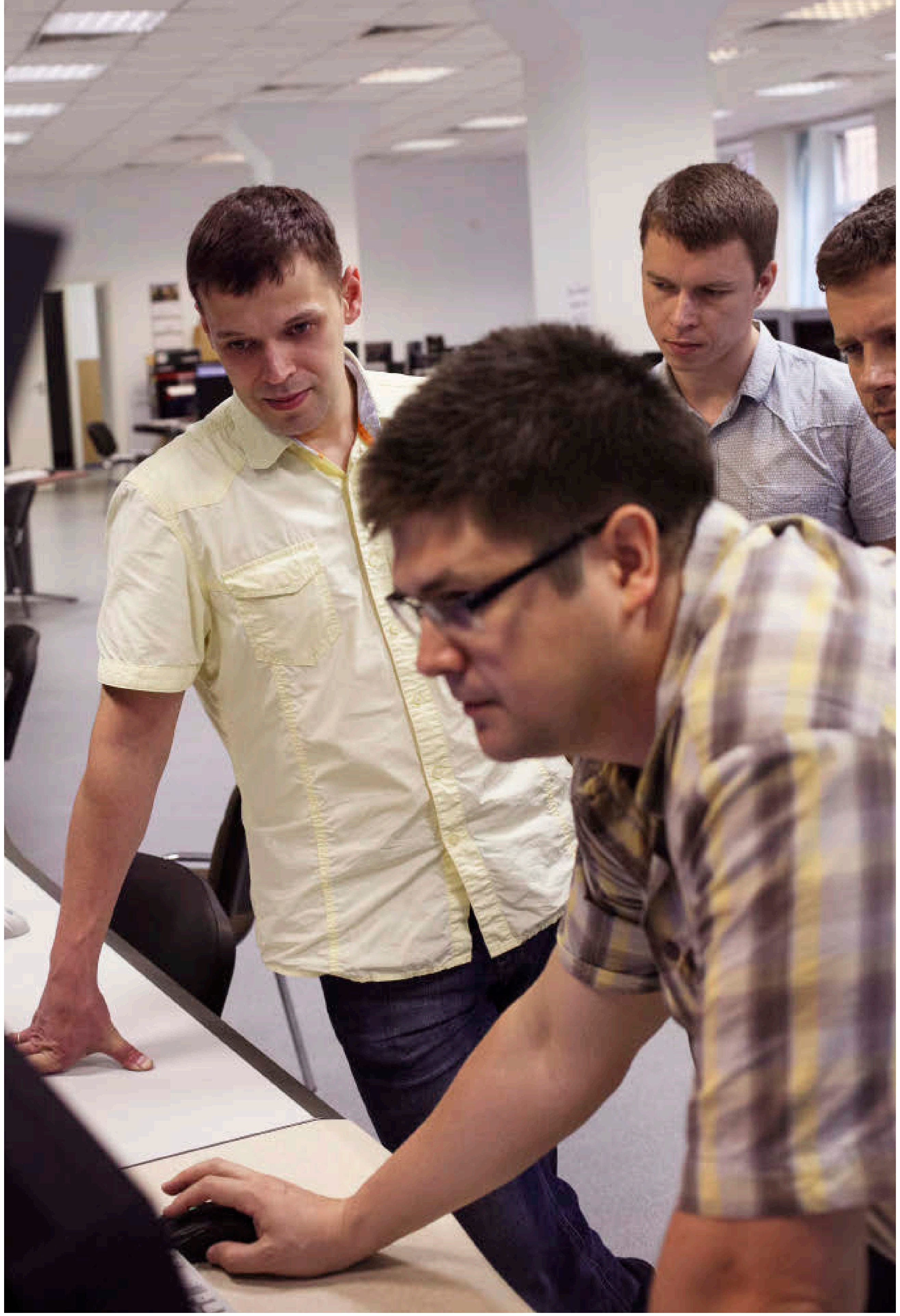
VNIIRA
vniira-ovd.com

Industry:
Aerospace

Location:
Saint Petersburg, Russia

Employees:
1,500

Java technologies used:
Java SE 1.7,
NetBeans IDE 7.4


Teterin checks in on the SINTEZ air traffic control system at the test area at the Pulkovo-1 Airport.

All-Russian Scientific Research Institute of Radio Equipment (VNIIRA). Located in Saint Petersburg, VNIIRA specializes in the development, production, commissioning, and maintenance of landing systems, air traffic control automation systems, airborne equipment, and weather radar systems. The company is well known in Russia for the air traffic control systems it has installed in Moscow, Saint Petersburg, Khabarovsk, Sochi, and other locations.

The task of ensuring safe operations of commercial and private aircraft falls on air traffic controllers. They must coordinate the movements of thousands of aircraft, keep them at a safe distance from each other, direct them during takeoff and landing, route them around bad weather, and ensure that traffic flows smoothly with minimal delays.

Air traffic control systems are often customized for each airport based on the airport's size, type of traffic, local regulations, and the accompanying hardware and software environment. In addition to private and commercial planes, lots of helicopters use Talakan Airport as a base of operations to reach the oil, gas, and other mining deposits in resource-rich Western Siberia. Coal, gold, silver, tin, tungsten, and many other natural resources are mined here. Sakha produces 99 percent of all Russian diamonds and more than 25 percent of the diamonds mined in the world.

"The airport is absolutely in the middle of nowhere," explains Fedorov. "The closest small town is more than 100 kilometers away. Despite its remote location, the airport meets the latest safety requirements in an area with challenging weather conditions."

"Challenging" is an understatement. About 40 percent of Sakha lies above the Arctic Circle. Winter low temperatures average –30 degrees Fahrenheit, and record lows of –76 have been recorded—some of the coldest temperatures on Earth, often with blizzard conditions. Precise tracking and surveillance of flights in and out of Talakan Airport are essential.

VNIIRA was a key contractor for the construction of the Talakan Airport due to the company's experience developing customized navigation and flight solutions. This multifaceted project included development, installation, and commissioning of up-to-date navigation, surveillance, and communication equipment along with equipping the control tower, developing the air traffic control system, and installing the airport's weather and lighting equipment.

AN EXPERIENCED TEAM

Fedorov has worked for VNIIRA since 2003 and has been using Java since 2006. When he started developing the SINTEZ-KSA ATC series of systems several years ago, he decided to base it on the NetBeans Platform. NetBeans is an integrated development environment (IDE) and application platform framework for Java desktop applications and other platforms.

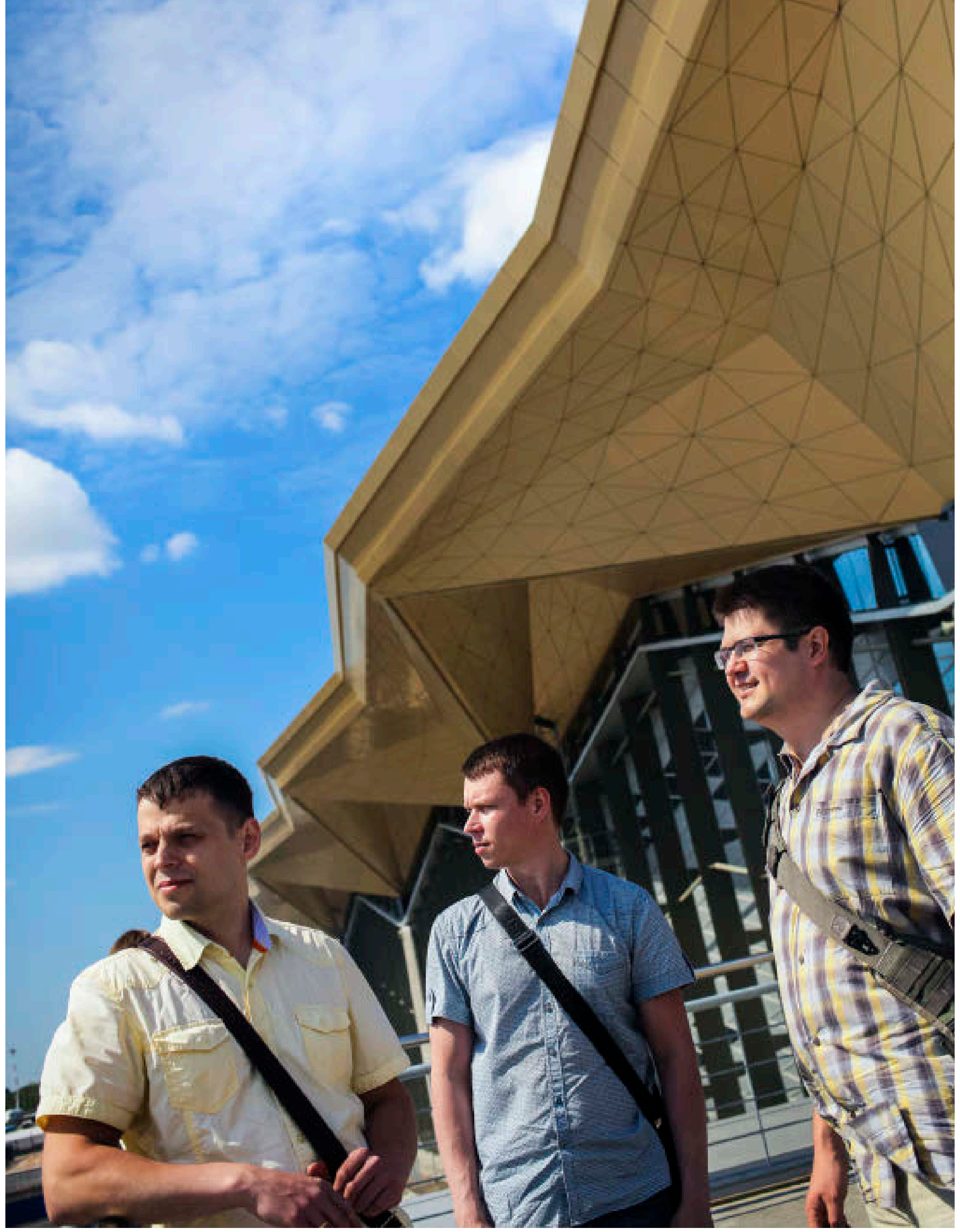
"Air traffic control systems must collect information from a number of sources, process that information, and present it to the controller," says Fedorov. "All those tasks can be easily accomplished with Java."

Once the software development project was underway, Fedorov hired two other Java programmers to help him complete the NetBeans Platform application: Alexandre Teterin and Mikhail Kuznetsov.

"Java helped us to develop reliable applications," says Teterin. "We have tremendous uptime, and that's essential for this domain. It is a very stable environment. Java also streamlines real-time communications and complies with the response time requirements. We used the standard, commercial Java implementation."

Teterin served in the military for about 15 years and earned a Candidate of Science degree (roughly equivalent to a PhD) at the Military Academy of Communication in Saint Petersburg in 2007. In 2009 he started listening to podcasts by Java Champion Yakov Fain. "Yakov's passion was one of the reasons that I decided to become a Java developer," recalls Teterin, who adds that he was also influenced by the open source project JTalks, the largest and most popular Java forum in Russia.

Kuznetsov joined the VNIIRA team in 2010, having graduated from Saint Petersburg State Electrotechnical University with a master's degree in 2008. "I started to learn Java at the university, where there is a growing swell of interest in the language," he says.


From left to right: Fedorov, Kuznetsov, and Teterin at the new terminal of the Pulkovo-1 Airport

MODULAR CONNECTIONS

More than 40 air traffic control centers are equipped with VNIIRA's unified series of SINTEZ air traffic control facilities, both in Russia and abroad. Flight executive officers and air traffic controllers are the primary users of these systems, which track thousands of daily flights using primary and secondary surveillance radars.

Like most air traffic control systems, SINTEZ handles airspace organization and management, aerodrome operations, demand and capacity balancing, traffic synchronization, conflict management, and airspace user operations. The airport relies on the custom system to process and display surveillance data and flight data, including information regarding airspace management, meteorological data, and air safety.

"SINTEZ predicts aircraft flight paths over a wide area of interest with great precision, allowing air traffic controllers to detect and forecast potential conflicts," explains Teterin. The system also helps air traffic controllers resolve these conflicts in an expedient manner. Application features include flight tracking, conflict detection and resolution, and data display from various sources, together with playback and recording of all system events and user actions.

SINTEZ consists of dozens of software modules to execute and coordinate these functions. Some of these components are implemented only in Java, others in Java and C++, and others only in C++. Fedorov and his team used Java with the NetBeans IDE because of the modular, cross-platform nature of this system.

"This modular open architecture made it easy to accommodate to shifting needs throughout the development lifecycle, as well as to interface with many types of radar systems and other sources of information," Fedorov says. "We were able to compose features from multiple modules to provide the necessary functionality tailored to various roles, from flight executive officers to air traffic controllers."

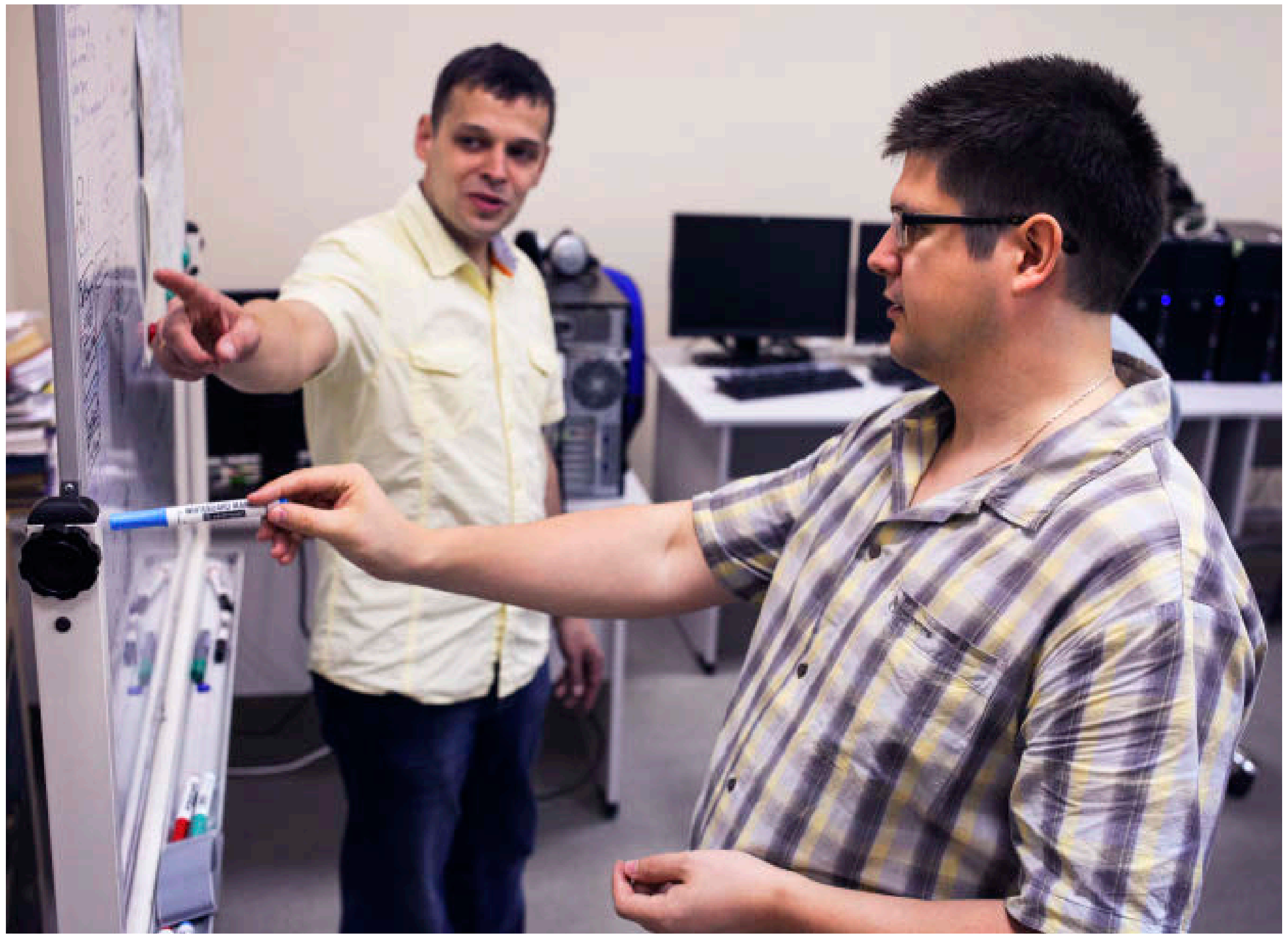
For example, the NetBeans Visual Library permitted the developers to create custom UI controls. According to Teterin, utilizing different sets of modules makes it possible to deploy automated workstations, which ensure execution of various tasks: flight management, air traffic controlling, engineering, and so forth.

The team also favored Java for its rapid application development cycle and innate portability. The NetBeans IDE can run on Windows, Macintosh OS X, Linux, Oracle Solaris, and other

VNIIRA's Credentials

VNIIRA has a great depth of experience developing customized navigation and flight solutions—and a proud list of accomplishments:

- 65 years as a pioneer of air safety
- 150 prototypes of radio technical systems, navigation systems, and radio instruments
- 1,300 inventor certificates and patents
- 60 complexes of air traffic control systems and facilities in Russia and other countries
- 100 types of aircraft employing VNIIRA's airborne equipment, navigation, and landing facilities


Fedorov (left) and Teterin brainstorm about updates to the SINTEZ system.

Teterin (left), shown with Kuznetsov and Fedorov, says that Java helped the team to develop reliable applications.

platforms supporting a compatible Java Virtual Machine, ensuring flexibility in the future. "If our management decides to develop a new version of the air traffic control system for another airport, it will be relatively easy to adapt to the new requirements," Fedorov says. "In addition, our target platform was Oracle Solaris, and since Java and Oracle Solaris come from the same vendor, they are guaranteed to fit each other unconditionally."

AN AWARD-WINNING IMPLEMENTATION

Fedorov's decision to select a portable software environment was prescient: Java's platform independence made it easy to port the SINTEZ systems to Linux when the requirements changed partway through the development cycle. He describes the transition as "absolutely seamless" and then goes on to talk about Java's versatility.

"Air traffic control systems collect a lot of information from many different sources, such as weather equipment and radar systems," Fedorov explains. "The system is based on standard protocols, but these standards can vary. Each airport has its own types of information. A modular design is useful for configuring the system for each specific airport. NetBeans lets us combine different models to achieve the necessary functionality. This rich client platform allows us to solve domain tasks with readily available APIs and libraries such as a module system, window management system, and widget system. We used NetBeans 7.4 but we are planning to switch to the latest version, so as to make extensive use of all the latest NetBeans capabilities."

The developers also favor Java for its mature ecosystem of third-party tools and utilities such as VisualVM profiler, Oracle Java Mission Control profiler, and Gradle. They use Apache JMeter for performance and load testing, SonarQube for quality testing, and Crucible overview for code review.

Oracle honored VNIIRA with a Duke's Choice Award during the opening ceremony of the 2013 JavaOne conference in Moscow for this custom air traffic control system. These prestigious awards were established in 2002 to acknowledge innovative projects in the world of Java-related technologies and to recognize regional software developers.

"I felt many of the same emotions as I experienced when the Science Council of the Military Academy of Communication awarded me with a science degree," says Teterin, recalling the JavaOne event. "I believe our other team members have similar emotions." </article>

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

MICHAEL KÖLLING

BIO

PHOTOGRAPH BY
JOHN BLYTHE

Part 2

Interactive Objects with BlueJ

Visualization and interaction tools illustrate aspects of object-oriented programming.

In this second part of a two-part series, we continue our discussion of BlueJ's interaction features—the main aspect that differentiates BlueJ from other environments. In Part 1, we discussed why interaction and visualization are important for learners, and we started by demonstrating the first (and most fundamental) examples of the visualization of object-oriented concepts in BlueJ: the depiction of classes and objects. Both are visualized graphically and allow direct interaction that illustrates and reinforces their roles, characteristics, and behavior.

However, while the class diagram and object bench are the most immediately visible and obvious features in BlueJ, they are not the only tools available for developing or reinforcing an understanding of object-oriented concepts. In this article, we examine a number of additional BlueJ design elements and tools that help learners develop consistent mental models and explore aspects of object orientation.

Working at a Conceptual Level

When learners start working in BlueJ, they can work at a conceptual level that is abstracted from some of the underlying implementation details. For example, users of BlueJ create *classes*, not *files*. The distinction might be subtle at first, but it is meaningful. Users do not need to be concerned with the file system structure, or, for example, with the rule that the filename has to match the class name. In fact, they do not even need to know, initially, that the source code for classes is stored in files. In BlueJ, when a programmer changes the class name in the class' source code, BlueJ automatically changes the underlying filename accordingly. Similarly, when a programmer changes the package name, BlueJ moves the source file to the right location.

At this point, experienced programmers often say "But it is important to know how Java classes are stored," to which my answer is, "No, it

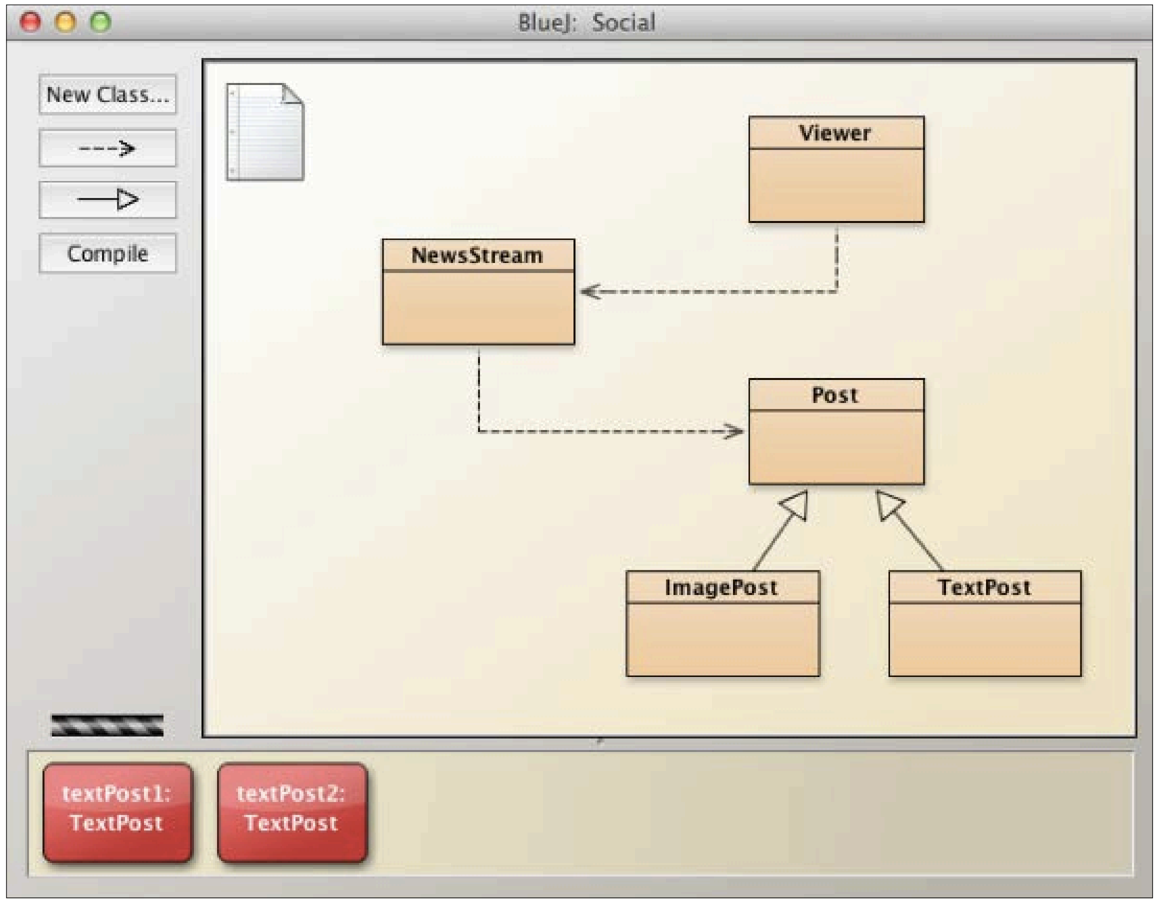


Figure 1

isn't—at least not at first." The principle of named, interacting classes is fundamental to object-oriented programming; this is what beginners should concentrate on. The file system structure, filenaming rules, and other storage details are merely a coincidental detail of Java—there is no fundamental principle here. Classes might just as well be stored in a database or in any other persistent storage—it is really not important while you are still struggling with programming fundamentals.

Dual Editing Options

A similar mechanism for working at a higher level exists for the relationships of classes in the class diagram: Inheritance and client relationships are depicted by two different kinds of arrows (see **Figure 1**). These arrows can be drawn in the diagram graphically, or the relationship can be defined textually in the source code, and the two representations will be kept in sync. If, for instance, a user inserts an inheritance arrow from class A to class B, the extends B clause will be automatically inserted into class A's source code. If the extends clause is edited in the text, the diagram is updated to reflect this.

Parameters

As discussed in Part 1 of this article, public methods of objects can be called interactively without the need to write test drivers. This mechanism illustrates communication with objects. However, the mechanism also helps users experiment with another concept: the passing of parameters.

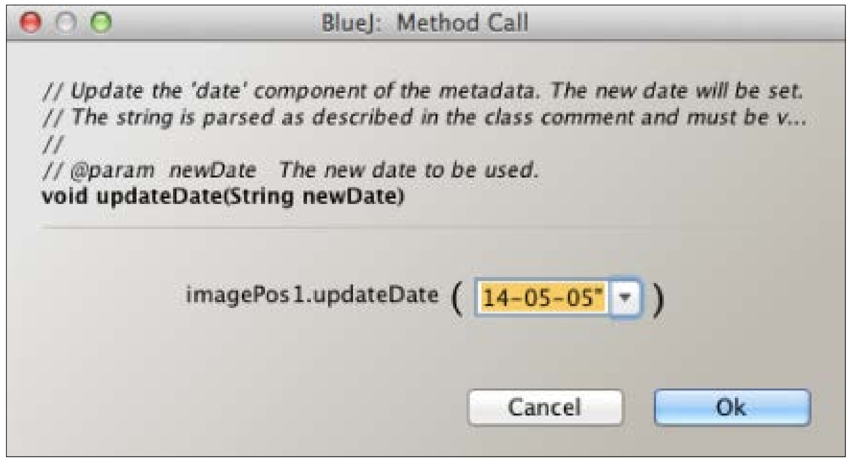
When a method that expects parameters is invoked, a dialog box opens that shows the message signature and its comment, and the user is prompted to enter values for the expected parameters (see **Figure 2**). This interaction allows students to experience and understand the relationship between the parameter specification in a method signature and the actual parameter values that must be passed. A first encounter of this feature usually also includes a first discussion of datatypes.

An interactive invocation in BlueJ is internally translated into a transient class, which is then compiled using the standard compiler. This ensures that possible errors in an interactive invocation are reported using the same mechanism and the same message as equivalent errors in source code.

Objects as parameters. The interactive entry of parameters is not restricted to primitive types. *Object composition*—the passing of one object as a parameter to another—is also supported. When a parameter of an object class is expected, the user can click an object on the object bench to cause the object to be used as the parameter.

Return Values

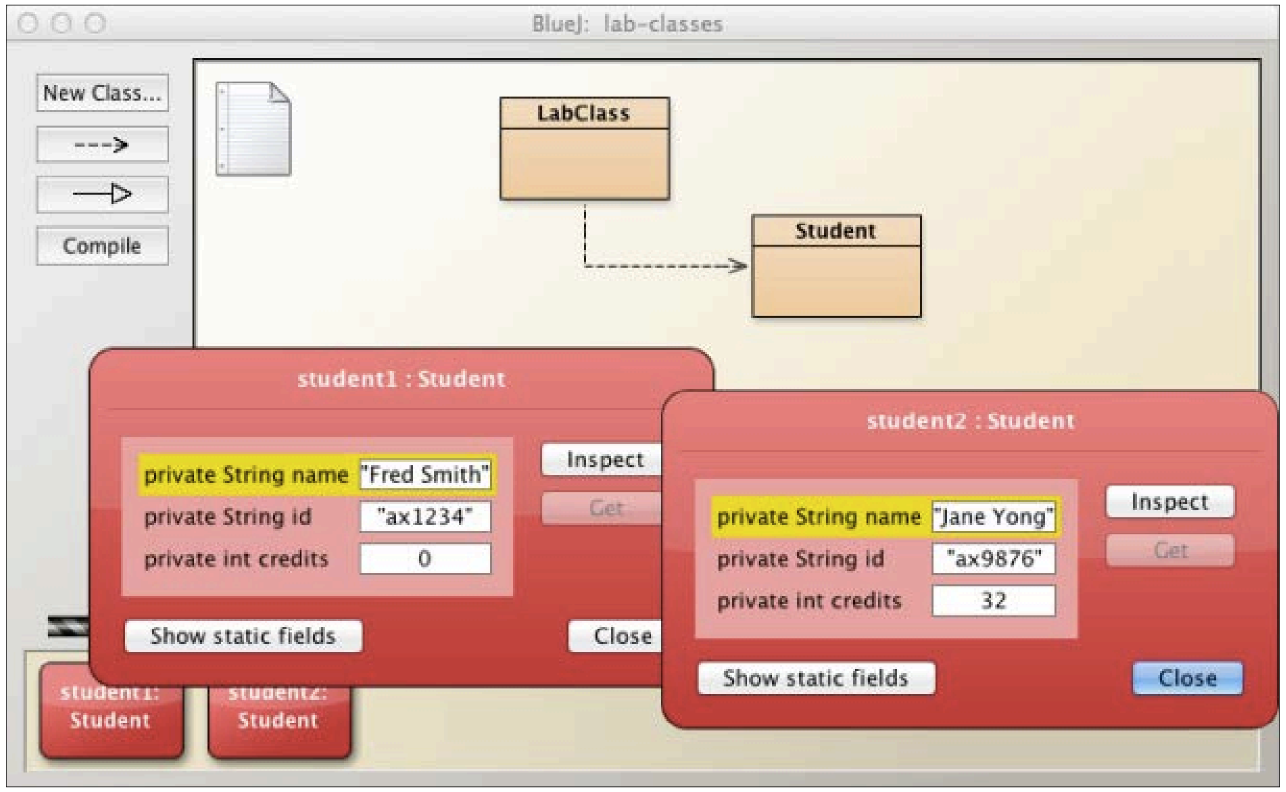
The last missing element related to method calls is the display of return values. If a method returns



```
// Update the 'date' component of the metadata. The new date will be set.
// The string is parsed as described in the class comment and must be v...
//
// @param  newDate   The new date to be used.
void updateDate(String newDate)

imagePos1.updateDate ( 14-05-05" )
```

Figure 2



student1 : Student

private String name	"Fred Smith"
private String id	"ax1234"
private int credits	0

student2 : Student

private String name	"Jane Yong"
private String id	"ax9876"
private int credits	32

Figure 3

a result, the result is displayed in a separate dialog box. The user can then inspect the result or—if it is an object—place it on the object bench. If the user is in the process of recording a unit test, an assertion can also be added.

Taken together, interactive method invocation, parameter passing, and return value display provide a consistent and meaningful illustration of object communication in object-oriented systems. Performing this process interactively imbues students with a sense of how this interaction works, and it helps them gain a good understanding of the mechanism.

Inspection

Methods, however, do not always return values; their effect might be a state change rather than an explicit result. *State* is the third of the three characteristics that define objects in object-oriented systems. (The other two are *identity* and *behavior*—and we have already seen how these are illustrated through interactive object creation, object display on the object bench, and method invocation.)

State is also visualized in BlueJ. Each object's context menu includes an **Inspect** option, which displays a visualization of the object's internal state (see **Figure 3**). Fields of the objects are listed with their type and name, and current values are displayed.

In the early phases of learning, it is especially enlightening to inspect two objects of the same type side by side and observe the similarity in fields, but the differences in values. Object inspectors can also remain open during interactive method calls to observe a state change more immediately.

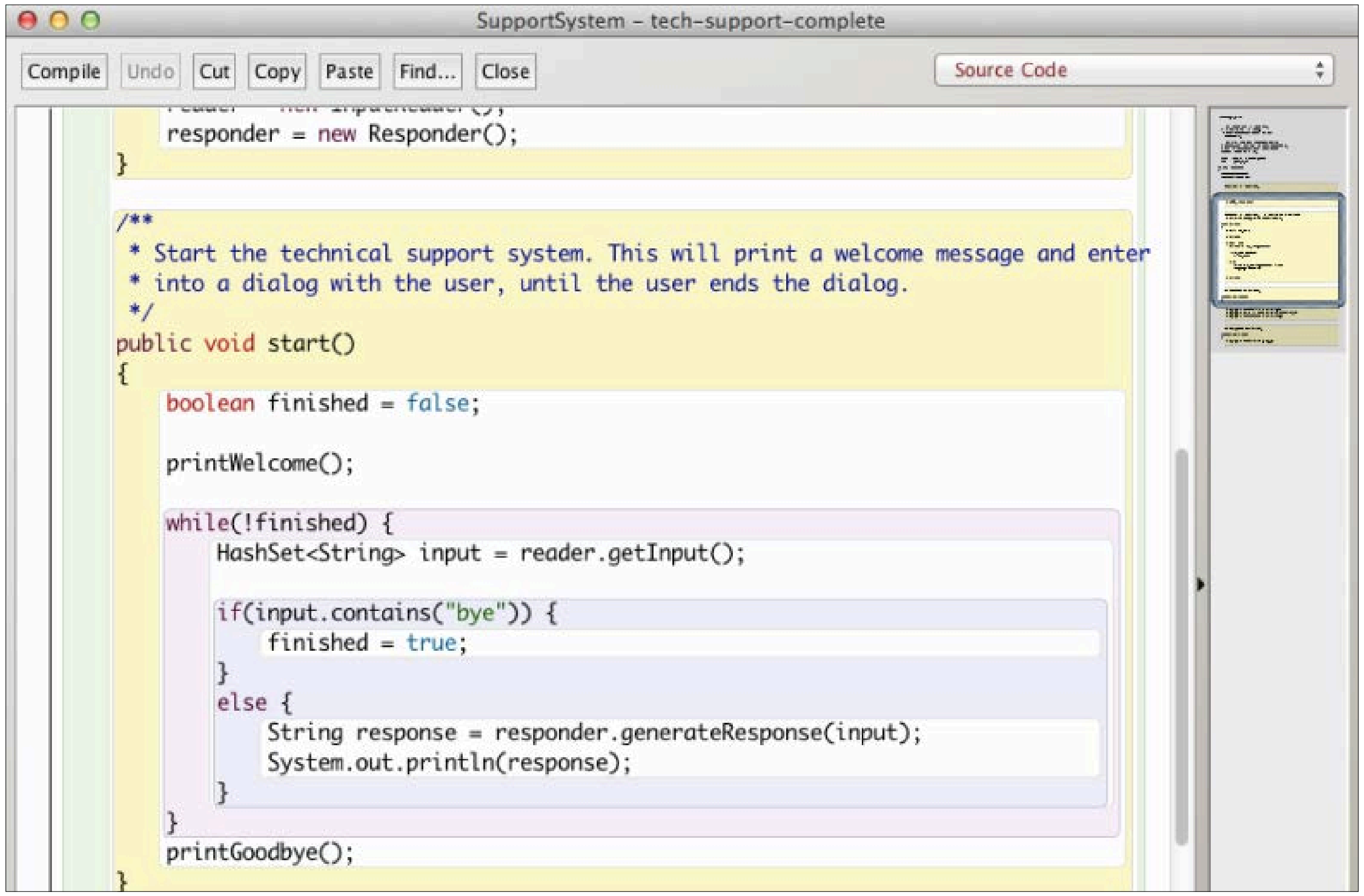
Again, this tool visualizes an important principle of object-oriented programming and supports obtaining a valid model of machine behavior.

Static methods and inspection. Static methods can also be invoked interactively. This is done by selecting them from the class'—rather than the object's—context menu. Similarly, static fields are displayed by inspecting the class rather than the object. This distinction in the interface reinforces the differences in definition between static attributes and object attributes.

The Editor: Scope Highlighting

As every teacher knows, understanding the object model is not the only difficulty in learning to program in Java. Beginning students also struggle with details of Java's syntax when writing their programs.

One of the most common problems, especially for young learners, is the correct balancing of curly brackets to define scopes. The very concept of nested scopes—so seemingly natural for all of us who have programmed for some time—is difficult to grasp for many beginners, and the placement of matching opening and closing scope brackets is prone to errors. Professional environments have tried to help by automating the insertion of bracket pairs and by highlighting matching brackets. However, this has not caused a great reduction in beginners' errors.



```
        responder = new Responder();
    }

    /**
     * Start the technical support system. This will print a welcome message and enter
     * into a dialog with the user, until the user ends the dialog.
     */
    public void start()
    {
        boolean finished = false;

        printWelcome();

        while(!finished) {
            HashSet<String> input = reader.getInput();

            if(input.contains("bye")) {
                finished = true;
            }
            else {
                String response = responder.generateResponse(input);
                System.out.println(response);
            }
        }
        printGoodbye();
    }
```

Figure 4

BlueJ goes a step further by automatically highlighting scopes in the editor using colored boxes (see **Figure 4**). Different types of scope (class, method, loop, and so on) are shown using different background colors, which serves two purposes. First, in syntactically valid programs, this visualization helps illustrate the concept of nested scopes, and it aids understanding. Second, if scopes are ill-defined—for example, because a bracket is missing—this display makes this fact immediately visually obvious. Scope highlights are updated in real time with every keystroke.

Code Experimentation: The Code Pad

The object interaction discussed above supports a better appreciation of the object model, but students also need to develop an understanding of small-scale constructs: the individual statements and expressions of the programming language. Again, experimentation can help with developing an insight into these constructs.

To support this kind of experimentation, BlueJ provides the *Code Pad* (see **Figure 5**). The Code Pad is an interactive interpreter of individual textual Java statements or expressions. Statements are executed, and expression results are displayed.

In the Code Pad, variables can be defined, assigned, and inspected. The namespace is the same as the object bench, so references can be made to the classes in the project and the objects currently on the object bench. If the result of a Code Pad expression is itself an object, it is shown using a small object icon that can then be inspected or dragged to the object bench.

In terms of teaching and learning, the availability of the Code Pad (and the other experimentation features) fundamentally changes how we as instructors can interact with our students. Frequently in a class, I am asked questions of the type "What happens when I do X?" or "Can I do Y?" (for example, "What happens if I use the modulo operator with a negative number?" or "Is the second parameter in substring the end or the length?").

Having the Code Pad available, my answer invariably is, "Try it!"

Being able to give this answer, from a teaching point of view, is highly valuable for two reasons: Not only do students remember the answer better if they have found it through their own experiments rather than by being told, but they are also given tools and strategies for answering future questions independently.

Test Recording

The last tool that I want to mention is the interactive recording of unit tests. In BlueJ, tests can be performed manually and interactively, and this interaction can be recorded to automatically create JUnit test classes.

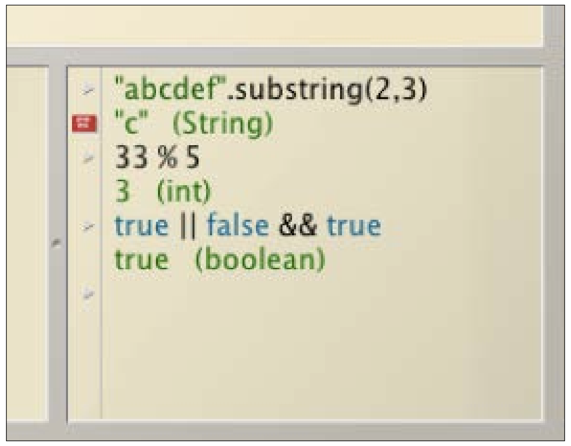
This mechanism was discussed in detail in an earlier issue of *Java Magazine*, so I won't repeat a description here. However, this tool supports the same spirit as the other functionality we've discussed: Interaction allows experimentation, which leads to better understanding.

Conclusion

Environments designed for beginning programmers cannot assume that users already have a well-formed mental model of the constructs and principles involved in their programming systems. On the contrary, they must help users acquire such models.

BlueJ uses a range of carefully designed visualization and interaction tools to illustrate many aspects of object-oriented programming. They are integrated closely and complement each other. These tools include some that illustrate aspects—such as the overall object model—at the conceptual level and others that illustrate small-scale concepts, such as scope or the functionality of individual operators.

An educational environment such as BlueJ does not compete with professional development environments. It provides a transitional step, and it is hoped that every BlueJ user will eventually outgrow BlueJ and move on to use one of the more traditional environments. However, BlueJ has an important place in the learning of programming. Developing a thorough understanding of the principles before getting bogged down with professional tools can turn every learner into a better programmer. </article>



```
"abcdef".substring(2,3)
"c"   (String)
33 % 5
3   (int)
true || false && true
true   (boolean)
```

Figure 5

LEARN MORE

- Java SE 8 API
- BlueJ website

VENKAT SUBRAMANIAM

Performing 10 Routine Operations Using Different JVM Languages

JVM languages offer unique capabilities, powerful constructs, and specialized libraries.

Daily coding activities are often punctuated by several routine operations, from reading a file to processing XML documents. Performing these operations with ease, using just a few lines of code, can help get those tasks out of our way, so we can focus on more-important, application-specific tasks.

In this article, we will use a polyglot approach to look for elegant, concise solutions for performing routine tasks using different Java Virtual Machine (JVM) languages.

Note: The source code for the examples described in this article can be downloaded here.

Leveraging the Polyglot Ecosystem

In the past few years, the JVM platform has flourished with quite a few powerful languages. Many developers have made use of different languages to program on the JVM. Each day, I come across more developers who are curious, willing, and eager to learn and apply techniques and solutions from different languages.

The different languages on the JVM do not merely offer a different syntax. Each of them comes with unique capabilities, a few powerful constructs, or specialized libraries that can make some routine tasks easy or almost trivial to perform. For example, Groovy's metaprogramming capability or Scala's ability to handle XML as a first-class citizen can come in handy. Furthermore, some things that were rather onerous in Java before just got easier with the recent release of Java SE 8.

In this article, we will explore solutions from Groovy, Scala, and Java SE 8 for the following operations:

- Quickly reading a file (Groovy and Scala)
- Easily starting a process (Groovy)
- Joining values (Java SE 8)
- Using delegation over inheritance (Groovy)
- Performing pattern matching (Scala)
- Easily parsing XML (Groovy)
- Generating XML documents (Groovy)
- Creating expressive code (Groovy)
- Making deep recursion possible (Scala)
- Guaranteeing resource cleanup (Java SE 8)

To try out the provided examples in a specific language, ensure the latest version of the language is installed on your system and the PATH environment variable is set up appropriately.

Quickly Reading a File Using Groovy and Scala

Let's take a look at the effort needed to read the entire contents of a small file using Java.

In **Listing 1**, we first open the /etc/networks file using the BufferedReader class (replace /etc/networks with a filename that is valid for your system). Then we loop through each line to concatenate the content into a StringBuilder. The noise in the code comes from two parts: the lack of a convenience method in BufferedReader and the need to handle checked exceptions. We can reduce the noise just a bit by using the BufferedReader's lines() method in Java 8; however, the noise from checked exceptions is here to stay.

We can quickly read the entire contents of a relatively small file using one line of code in both Groovy and Scala. Let's take a look at an example in Groovy first.

In **Listing 2**, File is the all-too-familiar class from the JDK java.io package. Groovy has extended this class with a getText() method, which we can access using the text property. Furthermore, Groovy does not have checked exceptions, so we're not forced to handle any exceptions. (Any exception we don't handle will propagate much like unchecked exceptions do in Java.)

To run the code shown in **Listing 2**, type the following command on the command line, where readFile.groovy is the file in which the Groovy code is saved:

```
groovy readFile.groovy
```

In case we mistype the filename, we can place a try and catch block around the code, if we desire, to handle any exceptions such as "file not found." Replace the constructor argument /etc/networks with other filenames to get the contents of different files.

Let's take a look at similar code in Scala. In **Listing 3,** the fromFile() method of the Source class can read the entire content of a given file and return an iterator. The mkString method concatenates the lines into a single string. Instead of using this method, we could iterate over each line, processing one line at a time.

To run the code in **Listing 3**, type the following command on the command line, where readFile.scala is the file in which the Scala code is saved:

```
scala readFile.scala
```

Easily Starting a Process Using Groovy

On a project I was working on, I was able to replace 50 lines of code for starting and interacting with an external program with a mere three lines of Groovy code, thanks to some convenience methods in Groovy.

In the previous section, we saw how Groovy took the File class and extended it with a convenience method. Likewise, Groovy adds some convenience methods to the java.lang.Process class, which we can use to easily start external processes.

Let's take a look at an example to call an external program. To keep the output short, we will simply call git but ask only for the version number. **Listing 4** shows the code.

To run an external process, we first create a string of the command we want to run, such as git version in this example. Then we call the execute() method on it. This is a Groovy-created extension method on the java.lang.String class. This method returns a java.lang.Process instance on which we call the getText() method (using the text property). This method returns the standard output of the program we invoked.

To run the program, save it in a file named callGit.groovy and type the following command on the command line:

```
groovy callGit.groovy
```

Here's the output from the code:

```
git version 1.9.2
```

In this example, we merely printed the content from the stan-

LISTING 1 / LISTING 2 / LISTING 3 / LISTING 4

```
//Java
import java.io.*;

public class ReadFile {
 public static void main(String[] args) {
  try {
   BufferedReader reader = new BufferedReader(
                          new FileReader("/etc/networks"));

   StringBuilder fileContent = new StringBuilder();
   String line;
   while((line = reader.readLine()) != null) {
    fileContent.append(String.format("%s%n", line));
   }

   //use fileContent for something practical here...
   System.out.println(fileContent);
  } catch(FileNotFoundException ex) {
   System.out.println(ex.getMessage());
  } catch(IOException ex) {
   System.out.println(ex.getMessage());
  }
 }
}
```

Download all listings in this issue as text

dard output. However, by storing a reference to the result of the execute() method, we can attach to standard input and standard error if we want more control over the interaction with the external process.

Joining Values Using Java SE 8

Given a list of names, let's look at the effort that is required to print the values in comma-separated format (see **Listing 5**).

Let's compile and run the code in **Listing 5** from the command line:

```
javac PrintList.java
java PrintList
```

Here is the output from the code:

```
[Marlin, Gill, Greg, Deb, Bruce]
```

It's not quite what we wanted; the names are comma-separated, but we have to get rid of those square brackets. Let's try again using the code in **Listing 6**, which uses the Java SE 5 for-each construct to loop through the values and print them in comma-separated form. Now we get this output:

```
Marlin, Gill, Greg, Deb, Bruce,
```

We got rid of the square brackets, but there is a silly comma at the end. This simple task seems to be a slippery slope. Let's take yet another stab at it, this time to get rid of that trailing comma (see **Listing 7**).

Here is the output of this version:

```
Marlin, Gill, Greg, Deb, Bruce
```

That worked! But, all of us have gone through similar exercises and ended up with code that we wouldn't be proud to flaunt. Such experiences leave us wondering why such a simple task should be so difficult to do.

Thankfully, the way we do this task has changed in Java SE 8. The String class has a new join method that can save us from all this trouble. **Listing 8** makes use of this new method.

With one call to the join method, we instructed Java to concatenate the elements in the list into a single string. We went from the verbose version that had a primitive obsession to a declarative version in which we concisely specified our intention. If we want to perform some operations on the list of elements and then concatenate the result, we can use the joining method of the Collectors utility class with the collect method of the Stream interface.

Java SE 8 added a number of similar new convenience methods to the classes and interfaces in the JDK to make our lives easier. Take a few minutes to revisit the JDK classes and interfaces to see what's new in Java SE 8.

Using Delegation in Groovy

Inheritance is quite useful for substitutability when an instance of a class can be used in place of an instance of a base class. However, if we desire class reuse more than substitutability, it is better to use delegation instead of inheritance.

Yet, in Java we often use inheritance more than delegation. The fact that delegation is difficult to write while inheritance is easy to implement is one of the reasons. If delegation were just as easy, we might rely on it more.

Using compile-time metaprogramming, Groovy makes delegation quite easy to use. Let's look at an example. The two classes in **Listing 9** are written in Groovy; however, they could be written in any JVM language.

The Worker class has one method, work, which prints out a message when called. The Analyst class has two methods, one with the same name and signature as the method in the Worker class.

Let's now create a Groovy class named Boss, which—as you might expect—does nothing. However, this is one smart class; it knows how to tactfully delegate (see **Listing 10**).

Within the Boss class, we first use the @Delegate annotation on an instance of the Worker class. This tells the Groovy compiler to generate in the Boss class a method named work that merely routes the

LISTING 5 / LISTING 6 / LISTING 7 / LISTING 8 / LISTING 9 / LISTING 10

```
//Java
import java.util.*;

public class PrintList {
  public static void main(String[] args) {
    List<String> names = Arrays.asList(
       "Marlin", "Gill", "Greg", "Deb", "Bruce");

    System.out.println(names);
  }
}
```

Download all listings in this issue as text

call to the same method of instance worker. Then we use the annotation again, this time on an instance of Analyst, which instructs the compiler to bring in methods from the Analyst class that are not already in the Boss class. Since the compiler just synthesized the work method, it skips that method of the Analyst class and generates a delegation method for the analyze method.

Let's use the Boss class now:

```
//Groovy
def boss = new Boss()
boss.work()
boss.analyze()
```

We created an instance of Boss and invoked the methods work and analyze. Before we run this code, let's take a look at the compiler magic. Save the Worker class, the Analyst class, and the Boss class plus the code above for using the Boss class in a file named delegate.groovy. Then, as shown below, compile the file using the groovyc compiler, which will produce Java bytecode for the three classes:

```
groovyc delegate.groovy
```

Let's take a peek at the bytecode for the Boss class using the javap tool:

```
javap -c Boss.class
```

If you examine the output of this tool, you will find that the compiled Boss class contains a method named work where the implementation routes the call to the worker instance's work method. See the excerpt shown in **Listing 11**.

Likewise, the analyze method will route the call to the analyze method of the analyst instance.

Run the delegate.groovy file from the command line:

```
groovy delegation.groovy
```

Here is the output from the code:

```
working...
analyst analyzing...
```

Groovy has a number of annotations such as @Delegate that perform compile-time abstract syntax tree (AST) transformations. Since these are done at compile time, there is no overhead at runtime for using these features. So, they provide convenience without compromising runtime performance.

Performing Pattern Matching in Scala

There's often a need to parse the content of a file or content received through a messaging service. The pattern matching capability of Scala is one of my favorite features of the language. It provides a concise way to not only take actions based on the types of data, but also to extract the contents from select types. Let's look at the example shown in **Listing 12**.

The process method receives an object (Any in Scala is like Object in Java). We invoke the match method on the instance and provide it several case statements to match against. In the first case, we look for an instance of String. If that match does not succeed, Scala will continue looking further in the sequence. We then look for the literal 5 followed by any instance of Int.

In the next case, we look for a list that starts with two elements: apple and peach. We capture the remaining elements in a fruits variable and print it comma-separated using the mkString() method (this method is the Scala equivalent of the Java SE 8 joining method we discussed earlier).

In the final case, we look for an XML fragment and parse the child of the greet element. In each of the case matches, we print a message with details about the parsed content.

As an aside, in **Listing 12** we see how Scala treats XML as a first-

LISTING 11 LISTING 12

```
Compiled from "delegate.groovy"
public class Boss implements groovy.lang.GroovyObject {

...
  public void work();
    Code:
       0: aload_0
       1: getfield      #29                  // Field worker:LWorker;
       4: invokevirtual #45                  // Method Worker.work:()V
       7: aconst_null
       8: pop
       9: return
...
```

Download all listings in this issue as text

class citizen. We can place any well-formed XML directly into Scala code, and we can embed Scala expressions into XML for easy generation of XML documents.

Let's call the process method with a few different values, as shown in **Listing 13**.

Now, let's look at the output first and then discuss each of the calls. To run the code, save the code in **Listing 12** and **Listing 13** in a file named match.scala and then type the following on the command line:

```
scala match.scala
```

Listing 14 shows the output.

The call to process with hello as an argument was picked up by the first case since there was a direct type match. The second call with a value of 5 was matched by the literal in the second case. The argument of 2, on the other hand, was matched by the case with the Int type. The List argument was matched by the case with the list, and the fruit variable was bound to the values grapes and kiwi.

Further, the XML fragment we passed in was matched by the last case and the child of the greet element, howdy, was bound to the msg variable. The last call with argument 2.2 was unmatched since no case exists for that value or that type. Therefore, the last call generated a runtime MatchError exception, which we handled on the call side.

In applications where we want to match against different types, values, or both, we can make use of the Scala matching capability to succinctly compare and process the values.

Easily Parsing XML Using Groovy

Groovy's XmlSlurper, combined with the dynamic nature of the language, makes parsing XML documents effortless. Let's get a feel for its capabilities using an example XML document, languages.xml, which is shown in **Listing 15**.

The root element, languages, has a bunch of child elements named language. Each of the child elements contains a name attribute and an author child element. Let's use the code in **Listing 16** to parse this XML content and print the name and author of each of the languages mentioned in languages.xml.

Listing 16 creates an instance of the XmlSlurper and points its parse method at the languages.xml file. The parser creates an instance of a node representation. We can dynamically access the properties of this object using the names of the known elements in languages.xml. To access an attribute, we prefix the name of the attribute with the @ symbol, for example, @name.

LISTING 13 / LISTING 14 / LISTING 15 / LISTING 16 / LISTING 17

```
//Scala
process("hello")
process(5)
process(2)
process(List("apple", "peach", "grapes", "kiwi"))
process(<greet>howdy</greet>)
try {
  process(2.2)
} catch {
  case ex : MatchError => println(s"Error: $ex")
}
```

Download all listings in this issue as text

Let's run this code and take a look at the output it produces:

```
Languages and authors:
C++         Stroustrup
Java        Gosling
Scala       Odersky
Ruby        Matz
Lisp        McCarthy
```

In addition to simple iteration over the elements, we can also apply powerful filter and search operations to extract the desired contents or parts of the document.

Generating XML Documents Using Groovy

In the previous section, we saw how to parse an XML document using Groovy. Groovy also makes it quite easy to create an XML document from data in memory (or data read from a file or from a database). Let's create a hashmap with sample data for languages and authors, and then create code to produce the languages.xml file we used earlier.

In **Listing 17**, the langs reference holds a hashmap of some languages as keys and their authors as values. The code creates an instance of groovy.xml.MarkupBuilder and uses the variable builder to hold a reference to it.

MarkupBuilder provides a highly fluent interface—with syntax similar to a domain-specific language (DSL)—for creating XML documents. If we call a property on an instance of MarkupBuilder and MarkupBuilder does not recognize the property, it assumes we're referring to an element. Depending on the level, which is specified by the curly braces ({), the property becomes an XML element at that level. Attributes for an XML element can be specified like a method argument. For example, language (name: key) defines an attribute named name for an element whose name is language. The variable key's value is substituted in the document as the value of the attribute name.

The output from the code in **Listing 17** is the same as the content of the languages.xml file that we used as an input for the parsing example.

In addition to Groovy, we could also fluently create XML documents using Scala, since that language treats XML as a first-class citizen and allows you to embed expressions in XML content.

Creating Expressive Code in Groovy

We saw the fluency of MarkupBuilder in the previous section. With Groovy metaprogramming, we can inject special methods into classes to create very fluent and expressive code. For example, we can facilitate the execution of code such as 2.days.ago quite easily. Let's try exactly that example by creating a series of classes and methods, as shown in **Listing 18**.

The DateUtil class has a method named getAgo, which returns a LocalDateTime instance that is a number of days prior to now. The number property is initialized using the constructor. This is a pretty standard class that we're used to writing. The core of the solution is in the code that follows this class.

In Groovy, we can inject methods into classes using a special metaClass property. In **Listing 18**, Integer.metaClass.getDays is a way to inject a getDays method into the java.lang.Integer class. We attach a closure (a lambda expression) as an implementation for this method. In this implementation, we create a new instance of DateUtil with the current Integer object—indicated by the delegate property—as the constructor argument.

As a final step, we're ready to use the facility we've created. Instead

LISTING 18 LISTING 19

```
//Groovy
import java.time.*

class DateUtil {
 int number

 DateUtil(aNumber) { number = aNumber }

 def getAgo() {
  LocalDateTime.now().minusDays(number)
 }
}

Integer.metaClass.getDays = { -> return new DateUtil(delegate) }

println 2.days.ago
```

Download all listings in this issue as text

of calling methods such as getDays and getAgo, we can use Groovy's JavaBeans facility to use the corresponding property names, such as days and ago, as in the last line.

Run this code to see it print out the date of the day that was two days before today.

Making Deep Recursion Possible Using Scala

Using a recursive approach, we can solve some problems by using solutions to their subproblems. Let's first take a look at the small Scala example shown in **Listing 19**, and discuss the problems we will soon run into with this approach.

In **Listing 19**, the factorial() method uses recursion to compute the factorial of a given number. For small positive arguments, this code will yield the right result. However, for large values, it will run into StackOverflowError. The reason for this error is that the last operation performed in the recursive call is multiplication. This final operation forces the partial result to be held on the stack while the method waits for the result of the subsequent recursive call to be completed.

This problem can be avoided using a special technique called *tail-call optimization*. A tail-call is when the last call in a method is a call to itself. In other words, in **Listing 19**, we could rearrange the code so the last operation is not multiplication, but a call to the method factorial. This rearrangement would change the simple recursion into tail-recursive form.

Merely converting a recursion to a tail-call in any arbitrary language will not solve the problem, however. We need the language's compiler or its library to support tail-call optimization to produce the desired effect. Thankfully, languages such as Scala and Clojure provide tail-call optimization through their compile support. In Scala, we can further enforce the tail-recursive form at compile time using a special annotation. Let's convert the code to tail-recursive form, as shown in **Listing 20**.

In this new version, the factorial method takes two parameters: the partial factorial result in the fact parameter and the number for which the factorial needs to be computed. Unlike the code in **Listing 19**, the last operation in **Listing 20** is not multiplication. We multiply the partial result in fact with the current number and pass it as an argument to the factorial method. This recursive call is the last operation in the method when the recursion has to continue.

At compile time, the tailrec annotation will ensure that the method is in tail-recursive form. The compiler will optimize this code "under the hood" by converting the recursion into a mere iteration. Even for large input parameters, this code will not run into a stack overflow situation.

With this approach in mind, when working on applications where we want to employ recursive algorithms for large input sizes, we can benefit from tail-call optimization techniques.

GO EXPLORING

By looking beyond the syntax—**by exploring the libraries and key capabilities of the languages we adopt**—we can leverage the languages to perform various routine tasks with greater ease.

Guaranteeing Resource Cleanup in Java

With the introduction of lambda expressions in Java, we can make use in Java of some good old patterns from the Smalltalk days. One such pattern is the Execute Around Method pattern, which is useful for wrapping a piece of code around some logic that we want to ensure is run. Let's first discuss the problem and then look into why this pattern is a good option.

Suppose we have an object that uses extensive external resources and we want to clean up the object deterministically. Java SE 7 provides a special language feature called try-with-resources. This feature is also called Automatic Resource Management (ARM). Let's look at an example of this feature, discuss some problems with it, and then look at an alternative solution.

Listing 21 shows a ResourceARM class that uses the ARM feature in Java SE 7.

To use the try-with-resources feature on a class, the class should implement the AutoCloseable interface. The ResourceARM class implements that interface and the required close method that would actively clean up the resources.

LISTING 20 LISTING 21 LISTING 22

```
//Scala
@scala.annotation.tailrec
def factorial(fact: BigInt, number: Int) : BigInt = {
  if(number == 1)
    fact
  else
    factorial(fact * number, number - 1)
}
```

Download all listings in this issue as text

Let's use this class in an example.

In **Listing 22**, we placed the instance creation in the special form of try. This form removes quite a bit of noise in code; the finally block is not required because that part is synthesized by the compiler. In the synthesized finally block, the compiler automatically calls the close method on the instance we created in the try statement.

With ARM, we have two benefits: the code is reduced—since we don't have to write the finally block—and the resource cleanup is guaranteed at the end of the try block. This try-with-resources feature is useful when such deterministic cleanup is necessary.

The disadvantage of this feature, however, is that we have to use this form of the try block. ARM is a suggestive feature; it does not force us to do the right thing.

If we want to enforce a clear "use and clean up" boundary for code, we can use lambda expressions in Java SE 8 and employ the Execute Around Method pattern. Let's create a code example for this approach (see **Listing 23**).

The Resource class in **Listing 23** is much like the ResourceARM class in **Listing 22**, except it does not implement the AutoCloseable interface. The constructor and the close methods are declared private, which will prevent the user of the class from directly creating an instance or invoking the cleanup code. We provide a use method, which is marked static and takes as a parameter an instance of the new Java SE 8 Consumer interface.

Within the use method, we create an instance of Resource and in the safe haven of the try and finally block, we pass the instance to the accept method of Consumer.

The use method is the implementation of the Execute Around Method pattern. It executes the proper creation and cleanup operation around whatever operation we intend to perform when accept is invoked.

EASY DOES IT

Tasks that are arduous in one language **might be quite easy** to perform in another.

With Java SE 8, we don't have to create an instance of single, abstract method interfaces; we can use lambda expressions instead. Here is the code to use Resource:

```
//Java
Resource.use(resource -> {
 resource.op1();
 resource.op2();
});
```

We invoke the use method of Resource and pass a lambda expression as an argument. In the lambda expression, we receive an instance of Resource that was passed to us from within the use method. We make use of this resource by calling methods such as op1() on it. When we return from the lambda expression, back in the use method, the cleanup operation takes place.

Since we marked the constructor private, the use of the pattern is forced on programmers using the Resource class. Unlike the ARM feature, the Execute Around Method pattern is useful when the pre- and post-operations have to be guaranteed and the usage of the instance has to be rather narrow and enforced.

Conclusion

Simple tasks should be simple, and thanks to some highly capable languages available on the JVM, several routine tasks are now quite approachable. In this article, we looked at 10 different tasks and examined how to solve them using the capabilities of three different languages (Java SE 8, Scala, and Groovy).

Tasks that are arduous in one language might be quite easy to perform in another due to a special capability, a library, or a special construct, as we saw in this article. Each of these languages has capabilities and features that can help us with tasks far beyond the 10 tasks discussed in this article.

By looking beyond the syntax—by exploring the libraries and key capabilities of the languages we adopt—we can leverage the languages to perform various routine tasks with greater ease. </article>

LISTING 23

```
//Java
import java.util.function.Consumer;

public class Resource {
 private Resource() { System.out.println("Resource created..."); }
 public void op1() { System.out.println("op1"); }
 public void op2() { System.out.println("op2"); }
 private void close() { System.out.println("cleanup logic goes here..."); }

 public static void use(Consumer<Resource> block) {
  Resource resource = new Resource();
  try {
   block.accept(resource);
  } finally {
   resource.close();
  }
 }
}
```

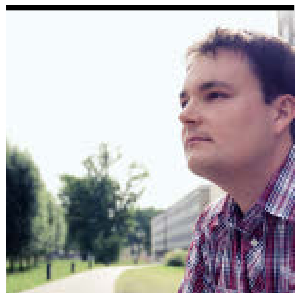
Download all listings in this issue as text

MORE ON TOPIC:

Java Virtual Machine

LEARN MORE

- Java SE 8
- Groovy
- Scala



Avoiding Benchmarking Pitfalls on the JVM

Use JMH to write useful benchmarks that produce accurate results.

JULIEN PONGE

BIO

Benchmarks are an endless source of debates, especially because they do not always represent real-world usage patterns. It is often quite easy to produce the outcome you want, so skepticism is a good thing when looking at benchmark results.

Yet, evaluating the performance of certain critical pieces of code is essential for developers who create applications, frameworks, and tools. Stressing critical portions of code and obtaining metrics that are meaningful is actually difficult in the Java Virtual Machine (JVM) world, because the JVM is an adaptive virtual machine. As we will see in this article, the JVM does many optimizations that render the simplest benchmark irrelevant unless many precautions are taken.

In this article, we will start by creating a simple yet naive benchmarking framework. We will see why things do not turn out as well as we hoped. We then will look at JMH, a benchmark harness that gives us a solid foundation for writing benchmarks. Finally, we'll discuss how JMH makes writing concurrent benchmarks simple.

TAKE NOTE
This is the first lesson: **mixing benchmarks** within the same JVM run is wrong.

A Naive Benchmarking Framework

Benchmarking does not seem so difficult. After all, it *should* boil down to measuring how long some operation takes, and if the operation is too fast, we can always repeat it in a loop. While this approach is sound for a program written in a statically compiled language, such as C, things are very different with an adaptive virtual machine. Let's see why.

Implementation. Let's take a naive approach and design a benchmarking framework ourselves. The solution fits into a single static method, as shown in **Listing 1**.

The bench method executes a benchmark expressed as a java.lang.Runnable. The other parameters include a descriptive name (name), a benchmark run duration (runMillis), the inner loop upper bound (loop), the number of warm-up rounds (warmup), and the number of measured rounds (repeat).

Looking at the implementation, we can see that this simple benchmarking method measures a throughput. The time a benchmark takes to run is one thing, but a throughput measurement is often more helpful, especially when designing microbenchmarks.

Sample usage. Let's use our fresh benchmarking framework with the following method:

```
static double distance(
double x1, double y1,
double x2, double y2) {
 double dx = x2 - x1;
 double dy = y2 - y1;
 return Math.sqrt((dx * dx) +
(dy * dy));
}
```

The distance method computes the Euclidean distance between two points (x1, y1)

PHOTOGRAPH BY MATT BOSTOCK/GETTY IMAGES

and (x2, y2).

Let's introduce the following constants for our experiments: 4-second runs, 10 measurements, 15 warm-up rounds, and an inner loop of 10,000 iterations:

```
static final long RUN_MILLIS =
4000;
static final int REPEAT = 10;
static final int WARMUP = 15;
static final int LOOP = 10_000;
```

Running the benchmark is done as follows:

```
public static void main(
String... args) {
 bench("distance", RUN_MILLIS,
LOOP, WARMUP, REPEAT, () ->
distance(0.0d, 0.0d, 10.0d,
10.0d));
}
```

On a test machine, a random execution produces the following shortened trace:

```
Running: distance
 (...)
[ ~30483613 ops/ms ]
```

According to our benchmark, the distance method has a throughput of 30483613 operations per millisecond (ms). Another run would yield a slightly different throughput. Java developers will not be surprised by that. After all, the JVM is an adaptive virtual machine: bytecode is first interpreted, and then native code is generated by a just-in-time compiler. Hence, performance results are subject to random variations that tend to stabilize as time increases.

Great; but still . . . is 30483613 operations per ms for distance a meaningful result?

What Could Possibly Go Wrong?

The raw throughput value does not give us much perspective, so let's compare our result for distance with the throughput of other methods.

Looking for a baseline. Let's take the same method signature as distance and return a constant instead of doing a computation with the parameters:

```
static double constant(
 double x1, double y1,
 double x2, double y2) {
 return 0.0d;
}
```

We also update our benchmark as shown in **Listing 2**. The constant method will give us a good baseline for our measurements, since it just returns a constant. Unfortunately, the results are not what we would intuitively expect:

LISTING 1 LISTING 2

```
public class WrongBench {

 public static void bench(String name, long runMillis, int loop,
int warmup, int repeat, Runnable runnable) {
  System.out.println("Running: " + name);
  int max = repeat + warmup;
  long average = 0L;
  for (int i = 0; i < max; i++) {
   long nops = 0;
   long duration = 0L;
   long start = System.currentTimeMillis();
   while (duration < runMillis) {
    for (int j = 0; j < loop; j++) {
     runnable.run();
     nops++;
    }
    duration = System.currentTimeMillis() - start;
   }
   long throughput = nops / duration;
   boolean benchRun = i >= warmup;
   if (benchRun) {
    average = average + throughput;
   }
   System.out.print(throughput + " ops/ms" + ([
!benchRun ? " (warmup) | " : " | "));
  }
  average = average / repeat;
  System.out.println("\n[ ~" + average + " ops/ms ]\n");
 }
}
```

Download all listings in this issue as text

```
Running: distance
  (...)
[ ~30302907 ops/ms ]

Running: constant
  (...)
[ ~475665 ops/ms ]
```

The throughput of constant appears to be lower than that of distance, although constant is doing no computation at all.

To give more depth to this observation, let's benchmark an empty method (see **Listing 3**). The results get even more surprising.

```
Running: distance
  (...)
[ ~29975598 ops/ms ]

Running: constant
  (...)
[ ~421092 ops/ms ]

Running: nothing
  (...)
[ ~274938 ops/ms ]
```

nothing has the lowest throughput, although it is doing the least.

Isolating runs. This is the first lesson: mixing benchmarks within the same JVM run is wrong. Indeed, let's change the benchmark order:

```
Running: nothing
  (...)
[ ~30146676 ops/ms ]

Running: distance
  (...)
[ ~493272 ops/ms ]

Running: constant
  (...)
[ ~284219 ops/ms ]
```

We get the same relative throughput drop figures, albeit with a different benchmark ordering.

Let's run a first benchmark alone, as shown in **Listing 4**. By repeating the process for each benchmark, we get the following results:

```
Running: nothing
  (...)
[ ~30439911 ops/ms ]

Running: distance
  (...)
[ ~30213221 ops/ms ]

Running: constant
[ ~30229883 ops/ms ]
```

In some runs distance could be faster than constant. The general observation is that all these measurements are very similar, with nothing being marginally faster. In itself, this result is suspicious, because the distance method is doing computations on double numbers. So we would expect a much lower throughput. We will come back to this later, but first let's discuss why mixing benchmarks within the same JVM process was a bad idea.

The main factor in why benchmarks get slower over runs is the Runnable.run() method call in the bench method. While the first benchmark runs, the corresponding call site sees only one implementation class for java.lang.Runnable. Given enough runs, the virtual machine speculates that run() always dispatches to the same target class, and it can generate very efficient native code. This assumption gets invalidated with the second benchmark, because it introduces a second class to dispatch run() calls to. The virtual machine has to deoptimize the generated code. It eventually generates efficient code to dispatch to either of the seen classes, but this is slower than in the previous case. Similarly, the third benchmark introduces a third implementation of java.lang.Runnable. Its execution gets slower because Java HotSpot VM generates efficient native code for up to two different types at a call site, and then it falls back to a more generic dispatch mechanism for additional types.

This is not the sole factor, though. Indeed, the bench method's code and the Runnable objects' code blend when seen by the virtual machine. The virtual machine

LISTING 3 LISTING 4

```
static void nothing() {

}

// (...)

public static void main(String... args) {
  bench("distance", RUN_MILLIS, LOOP, WARMUP, REPEAT, () ->
distance(0.0d, 0.0d, 10.0d, 10.0d));
  bench("constant", RUN_MILLIS, LOOP, WARMUP, REPEAT, () ->
constant(0.0d, 0.0d, 10.0d, 10.0d));
  bench("nothing", RUN_MILLIS, LOOP, WARMUP, REPEAT,
WrongBench::nothing);
}
```

Download all listings in this issue as text

tries to speculate on the entire code using optimizations such as loop unrolling, method inlining, and on-stack replacements.

Calling System.currentTimeMillis() has an effect on throughput as well, and our benchmarks would need to accurately subtract the time taken for each of these calls. We could also play with different inner-loop upper-bound values and observe very different results.

The OpenJDK wiki Performance Techniques page provides a great overview of the various techniques being used in Java HotSpot VM. As you can see, ensuring that we measure only the code to be benchmarked is difficult.

More pitfalls. Going back to the performance evaluation of the distance method, we noted that its throughput was very similar to the throughput measured for a method that would do no computation and return a constant.

In fact, Java HotSpot VM used dead-code elimination; since the return value of distance is never used by our java.lang.Runnable under test, it practically removed it. This also happened because the method has no side effect and has a simple control flow that is recursion-free.

To convince ourselves of this, let's modify the java.lang.Runnable lambda that we pass to our benchmark method, as shown in **Listing 5**.

Instead of just calling distance, we now assign its return value to a field and eventually print it, to force the virtual machine not to ignore it. The benchmark figures are now quite different:

```
Running: distance_use_return
 (...)
[ ~18865939 ops/ms ]
```

We now have a more meaningful result, because the constant method had a throughput of about 30229883 operations per ms on our test machine.

Although it's not perceptible in this example, we could also highlight the effect of *constant folding*. Given a simple method with constant arguments and a return value that is evidently dependent on those parameters, the virtual machine is able to speculate that it is not useful to evaluate each call. We could come up with an example to illustrate that, but let's instead focus on writing benchmarks with a good harness framework.

Indeed, the following should be clear by now:

- Our simple benchmarking framework has flaws.
- The virtual machine does so many optimizations that it is difficult to ensure that what we are benchmarking is actually what we expect to benchmark.

LISTING 5 / LISTING 6 / LISTING 7

```
// (...)

static double last = 0.0d;

public static void main(String... args) {
  bench("distance_use_return", RUN_MILLIS, LOOP, WARMUP,
REPEAT, () -> last = distance(0.0, 0.0, 10.0, 10.0));
  System.out.println(last);
}
```

Download all listings in this issue as text

Introducing JMH

JMH is a Java harness library for writing benchmarks on the JVM, and it was developed as part of the OpenJDK project. JMH provides a very solid foundation for writing and running benchmarks whose results are not erroneous due to unwanted virtual machine optimizations. JMH itself does not prevent the pitfalls that we exposed earlier, but it greatly helps in mitigating them.

JMH is popular for writing *microbenchmarks*, that is, benchmarks that stress a very specific piece of code. JMH also excels at concurrent benchmarks. That being said, JMH is a general-purpose benchmarking harness. It is useful for larger benchmarks, too.

Creating and running a JMH project. While JMH releases are being regularly published to Maven Central Repository, JMH development is very active and it is a great idea to make builds of JMH yourself. To do so, you need to clone the JMH Mercurial repository, and then build it with Apache Maven, as shown in **Listing 6**. Once this is done, you can bootstrap a new Maven-based JMH project, as shown in **Listing 7**.

This creates a project in the benchmarks folder. A sample benchmark can be found in src/main/java/MyBenchmark.java. While we will dissect the sample benchmark in a minute, we can already build the project with Apache Maven:

```
$ cd benchmarks/
$ mvn package
 (...)
$ java \
  -jar target/microbenchmarks.jar
  (...)
```

When you run the self-contained microbenchmarks.jar executable JAR file, JMH launches all the benchmarks of the project with default settings. In this case, it runs MyBenchmark with the default JDK and no specific JVM tuning. Each benchmark is run with 20 warm-up rounds of 1 second each and then with 20 measurement rounds of 1 second each. Also, JMH launches a new JVM 10 times for running each benchmark.

As we will see later, this behavior can be customized in the benchmark source code, and it can be overridden using command-line flags. Running java -jar target/microbenchmarks.jar -help allows us to see the available flags.

Let's instead run the benchmark with the parameters shown in **Listing 8**. These parameters specify the following:

- We use only one fork (-f 1).
- We run five warm-up iterations (-wi 5).
- We run five iterations of 3 seconds each (-i 5 -r 3s).
- We tune the JVM configuration with jvmArgs.
- We run all benchmarks whose class name matches the .*Benchmark.* regular expression.

The execution gives a recap of the configuration, the information for each iteration and, finally, a summary of the results that includes confidence intervals, as shown in **Listing 9**.

Anatomy of a JMH benchmark. The sample benchmark that was generated looks like **Listing 10**. A JMH benchmark is simply a class in which each @GenerateMicroBenchmark annotated method is a benchmark. Let's transform the benchmark to measure the cost of adding two integers (see **Listing 11**).

We have a baseline benchmark that gives us a reference on returning an int value. JMH takes care of reusing return values so as to defeat dead-code elimination. We also return the value of field x; because the value can be changed from a large number of sources, the virtual machine is unlikely to attempt constant folding optimizations. The code of sum is very similar.

The benchmark has more configuration annotations present. The @State annotation is useful in the context of concurrent benchmarks. In our case, we simply hint to JMH that x and y are thread-scoped. The other annotations are self-explanatory. Note that these values can be overridden from the command line. By running the benchmark on a sample machine, we get the results shown in **Listing 12**.

Lifecycle and parameter injection. In simple cases, class fields can hold the benchmark state values.

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

```
$ java -jar target/microbenchmarks.jar \
  -f 1 -wi 5 -i 5 -r 3s \
  -jvmArgs '-server -XX:+AggressiveOpts' \
  .*Benchmark.*
```

Download all listings in this issue as text

In more-elaborate contexts, it is better to extract those into separate @State annotated classes. Benchmark methods can then have parameters of the type of these state classes, and JMH arranges instances injection. A state class can also have its own lifecycle with a setup and a tear-down method. We can also specify whether a state holds for the whole benchmark, for one trial, or for one invocation.

We can also require JMH to inject a Blackhole object. A Blackhole is used when it is not convenient to return a single object from a benchmark method. This happens when the benchmark produces several values, and we want to make sure that the virtual machine will not speculate based on the observation that the benchmark code does not make use of these. The Blackhole class provides several consume(...) methods.

The class shown in **Listings 13a** and **13b** is an elaborated version of the previous benchmark with a state class, a lifecycle for the state class, and a Blackhole.

A BIT OF HISTORY

JMH was designed with concurrent benchmarks in mind. These kinds of benchmarks are very difficult to measure correctly, because they involve several threads and inherently nondeterministic behaviors.

When a benchmark method returns a value, JMH takes it and consumes it into a Blackhole. Returning a value and using a Blackhole object are equivalent, as shown by the benchmark results in **Listing 14**.

The @TearDown annotation was illustrated for the sake of completeness, but we could clearly have omitted the shutdown() method for this simple benchmark. It is mostly useful for cleaning up resources such as files.

Our "wrong" benchmark, JMH-style. We can now revisit with JMH the benchmark we did in the beginning of the article. The enclosing class looks like **Listing 15**.

We will be measuring throughput in terms of operations per ms. Data is enclosed within an @State-annotated static inner class whose mutable fields will prevent Java HotSpot VM from doing certain optimizations that we discussed earlier.

We are using two baselines. The first is a void empty method, and the second simply returns a con-

LISTING 13a / LISTING 13b / LISTING 14 / LISTING 15

```
package com.mycompany;

import org.openjdk.jmh.annotations.*;
import org.openjdk.jmh.logic.BlackHole;

import java.util.Random;
import java.util.concurrent.TimeUnit;

@BenchmarkMode(Mode.Throughput)
@OutputTimeUnit(TimeUnit.MILLISECONDS)
@Fork(value = 3,
  jvmArgsAppend = {"-server", "-disablesystemassertions"})
public class MyBenchmark {

 @State(Scope.Thread)
 static public class AdditionState {

  int x;
  int y;

  @Setup(Level.Iteration)
  public void prepare() {
   Random random = new Random();
   x = random.nextInt();
   y = random.nextInt();
  }

  @TearDown(Level.Iteration)
  public void shutdown() {
   x = y = 0; // useless in this benchmark...
  }
 }
```

Download all listings in this issue as text

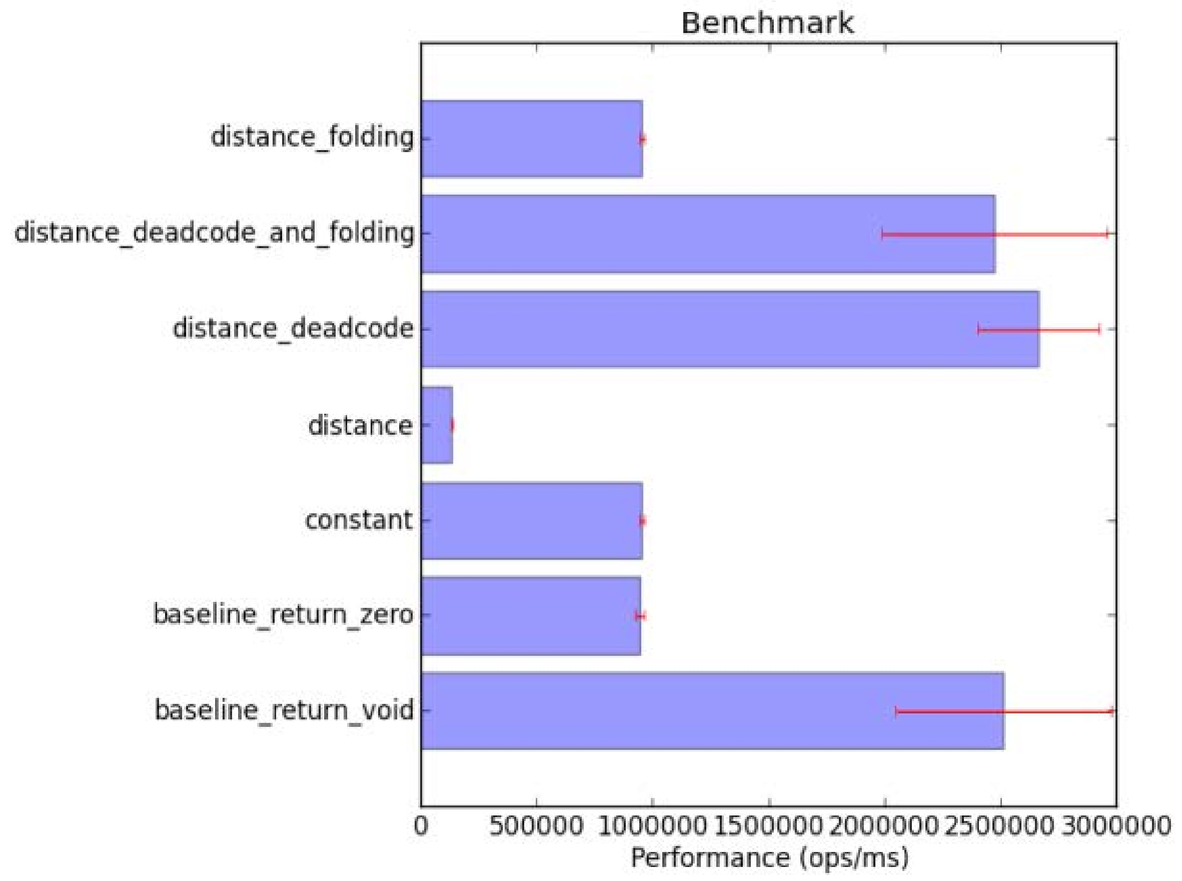


Figure 1

LISTING 16 / LISTING 17 / LISTING 18 / LISTING 19

```
@GenerateMicroBenchmark
public void baseline_return_void() {

}

@GenerateMicroBenchmark
public double baseline_return_zero() {
  return 0.0;
}
```

Download all listings in this issue as text

stant double value, as shown in **Listing 16**. Benchmarking constant() and distance() is as simple as **Listing 17**.

To put things into perspective, we also include flawed measurements subject to dead-code elimination and constant folding optimizations (see **Listing 18**).

Finally, we can also provide a main method to this benchmark using the JMH builder API, which mimics the command-line arguments that can be given to the self-contained JAR executable. See **Listing 19**.

Figure 1 shows the results as a bar chart with the mean error included for each benchmark.

Given the two baselines, we clearly see the effects of dead-code elimination and constant folding. The only meaningful measurement of distance() is when the value is being consumed by JMH and parameters are passed through field values. All other cases converge to either the performance of returning a constant double or an empty void-returning method.

Devising Concurrent Benchmarks

JMH was designed with concurrent benchmarks in mind. These kinds of benchmarks are very difficult to measure correctly, because they involve several threads and inherently nondeterministic behaviors.

Next, let's examine concurrent benchmarking with JMH for the comparison of readers and writers over an incrementing long value. To do so, we use a pessimistic implementation based on a long value for which every access is protected by a synchronized block, and an optimistic implementation based on java.util.concurrent.atomic.AtomicLong. We want to compare the performance of each implementation depending on the proportion of readers and writers that we have.

JMH has the ability to execute a group of threads with different benchmark code. We can specify how many threads will be allocated to a certain benchmark method. In our case, we will have cases with more readers than writers and, conversely, cases with more writers than readers.

Benchmarking the pessimistic implementation. We start with the following benchmark class code:

```
@BenchmarkMode(
Mode.AverageTime)
```

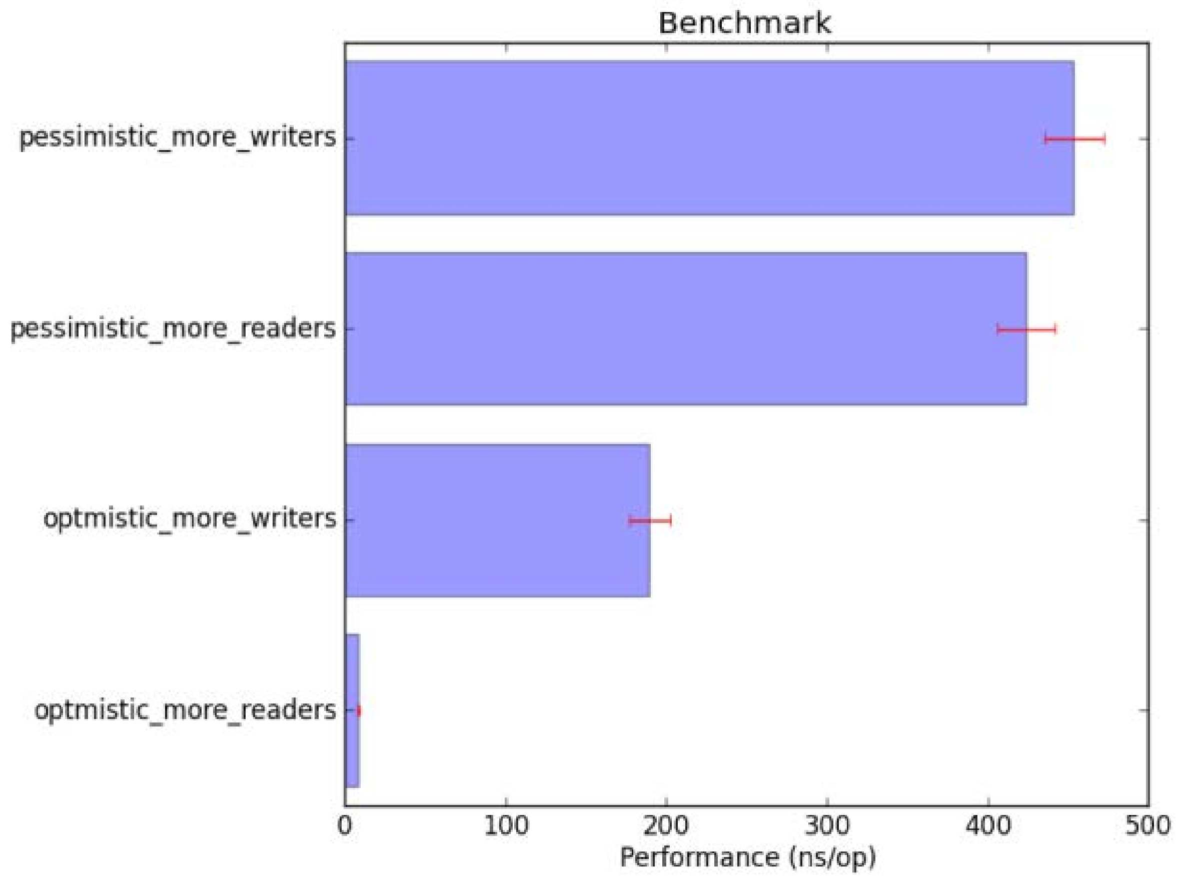


Figure 2

LISTING 20 / LISTING 21 / LISTING 22 / LISTING 23

```
State(Scope.Group)
@Threads(8)
public static class Pessimistic {

  long value = OL;
  final Object lock = new Object();

  @Setup(Level.Iteration)
  public void prepare() {
    value = OL;
  }

  public long get() {
    synchronized (lock) {
      return value;
    }
  }

  public long incrementAndGet() {
    synchronized (lock) {
      value = value + 1L;
      return value;
    }
  }
}
```

Download all listings in this issue as text

```
@OutputTimeUnit(
TimeUnit.NANOSECONDS)
public class ConcurrentBench {
  // (...)
}
```

The pessimistic case is implemented using an inner class of ConcurrentBench, as shown in **Listing 20**.

The @State annotation specifies that there should be a shared instance per group of threads while running benchmarks. The @Threads annotation specifies that eight threads should be allocated to run the benchmarks (the default value is 4).

Benchmarking the pessimistic case is done through the methods shown in **Listing 21**. The @Group annotation gives a group name, while the @GroupThreads annotation specifies how many threads from the group should be allocated to a certain benchmark. We, hence, have two groups: one with seven readers and one writer, and one with one reader and seven writers.

Benchmarking the optimistic implementation. This case is quite symmetrical, albeit with a different implementation (see **Listing 22**). The benchmark methods are also split in two groups, as shown in **Listing 23**.

Execution and plotting. JMH offers a variety of output formats beyond plain-text console output, including JSON and CSV output. The JMH configuration shown in **Listing 24** allows us to obtain results in a .csv file.

The console output provides detailed results with metrics for each benchmarked method. In our case, we can distinguish the performance of reads and writes. There is also a consolidated performance result for the whole benchmark.

The resulting .csv file can be processed with a variety of tools, including spreadsheet software and plotting tools. For concurrent benchmarks, it contains only the consolidated results. **Listing 25** is a processing example using the Python matplotlib library. The result is shown in **Figure 2**.

As we could expect, we see that the pessimistic implementation is very predictable: reads and writes share a single intrinsic lock, which is consistent, albeit slow. The optimistic case takes advantage of compare and swap, and reads are very fast when there is low write contention. As a warning, we could further increase the contention with more writers, and then the performance would be worse than that of the pessimistic case.

Conclusion

This article introduced JMH, a benchmark harness for the JVM. We started with our own benchmarking code and quickly realized that the JVM was doing optimizations that rendered the results meaningless. By contrast, JMH provides a coherent framework to write benchmark code and avoid common pitfalls.

As usual, benchmarks should always be taken with a grain of salt. Microbenchmarks are very peculiar, since stressing a small portion of code does not preclude what actually happens to that code when it is part of a larger application. Nevertheless, such benchmarks are great quality assets for performance-critical code, and JMH provides a reliable foundation for writing them correctly. </article>

LISTING 24 LISTING 25

```
public static void main(String... args) throws RunnerException {
 Options opts = new OptionsBuilder()
  .include(".*.ConcurrentBench.*")
  .warmupIterations(5)
  .measurementIterations(5)
  .measurementTime(TimeValue.milliseconds(5000))
  .forks(3)
  .result("results.csv")
  .resultFormat(ResultFormatType.CSV)
  .build();
 new Runner(opts).run();
}
```

Download all listings in this issue as text

MORE ON TOPIC:
Java Virtual Machine

LEARN MORE

- OpenJDK Code Tools: JMH
- JMH samples

CREATE THE FUTURE
oracle.com/java

BEN EVANS

BIO

Part 1

Understanding Java JIT Compilation with JITWatch

A primer on JIT compilation in Java HotSpot VM

Oracle's Java HotSpot VM is equipped with a highly advanced just-in-time (JIT) compiler. This means that the class files (which are compiled from Java source code) are further compiled at runtime, and they can be turned into very highly optimized machine code. This optimized code runs extremely fast—usually as fast as (and, in certain cases, faster than) compiled C/C++ code.

The JIT compiler is, therefore, one of the most important parts of Java HotSpot VM, and yet many Java developers do not know much about it or how to check that their applications work well with the JIT compiler.

Fortunately, a new open source tool called JITWatch is being developed to give developers much better insight into how the JIT compiler treats their code. For most effective use, the JITWatch tool relies on developers already understanding the basic mechanisms and terminology of JIT compilation.

CODE INSIGHT

A new open source tool called JITWatch is being developed to give developers much better insight into how the JIT compiler treats their code.

This article provides a basic primer on JIT compilation as it happens in Java HotSpot VM. We'll discuss how to switch on simple logging for the JIT compiler and some of the most common (and important) JIT compilation techniques that modern Java HotSpot VM versions use. Then we'll talk about the more-detailed logging options available (these are the options that JITWatch makes use of). This will pave the way for a full introduction to JITWatch in Part 2 of this series.

Let's kick off with a few fundamentals about JIT compilation as it is done in Java HotSpot VM.

Basic JIT Compilation

Java HotSpot VM automatically monitors which methods are being executed. Once a method has become eligible (by meeting some criteria, such as being called often), it is scheduled for compilation into machine code, and it is then known as a *hot method*. The compilation into machine code happens on a separate JVM thread and will not interrupt the execution of the program. In fact, even while the compiler thread is compiling a hot method, the Java Virtual Machine (JVM) will keep on using the original, interpreted version of the method until the compiled version is ready.

To learn more about the JIT compilation process, see "Understanding the Java HotSpot VM Code Cache," and "Introduction to JIT Compilation in Java HotSpot VM."

The first step to understanding how JIT compilation in Java HotSpot VM is affecting your code is to see which of your methods are getting compiled. Fortunately this is very easy to do, and only requires you to add the -XX:+PrintCompilation flag to the script you use to start your Java processes.

Java Virtual Machine

PHOTOGRAPH BY JOHN BLYTHE

Note: The resulting log of compilation events will end up in the standard log (that is, the standard output), and there is currently no way to redirect the entries to another file.

The exact format of the PrintCompilation flag's log entries varies between different Java versions. Here are some examples of log formats from different Java versions. **Listing 1** shows an example log from JDK 6.

In the JDK 6 form of the PrintCompilation flag's log entries, the first number corresponds to the compilation ID. This ID essentially tracks an individual method as it is compiled, optimized, and possibly deoptimized again.

Thereafter follow some flags that indicate properties of the method; for example, the s indicates the method is synchronized, and the ! indicates the method has exception handlers.

Next comes the name of the method—in fully qualified form—followed by the number of bytes of bytecode contained in the method being compiled. There is a minor annoyance: the method signatures are not printed out in the output.

Listing 2 shows an example log from JDK 7 onward. The big change in the JDK 7 form of the logs is that the first column is now the time—in milliseconds since the JVM started—at which the compilation occurred. Otherwise, the other fields are essentially the same as with JDK 6.

There are a number of excellent posts on the subject of reading PrintCompilation output, for example, those by Stephen Colebourne and Chris Vest, which are both highly recommended.

If you're using a different JVM language, such as Scala or Groovy, then you should be aware that those languages' compilers might alter (mangle) the names of methods and add or remove methods as part of their process for producing class files.

GET TO KNOW IT

The JIT compiler is one of the most important parts of Java HotSpot VM, and yet many Java developers do not know much about it or how to check that their applications work well with the JIT compiler.

LISTING 1 LISTING 2

```
JDK 6:

22     java.util.HashMap::getEntry (79 bytes)
23 s!   sun.misc.URLClassPath::getLoader (136 bytes)
```

Download all listings in this issue as text

Some JIT Compilation Techniques

One of the most common JIT compilation techniques used by Java HotSpot VM is *inlining*, which is the practice of substituting the body of a method into the places where that method is called. Inlining saves the cost of calling the method; no new stack frames need to be created. By default, Java HotSpot VM will try to inline methods that contain less than 35 bytes of JVM bytecode.

Another common optimization that Java HotSpot VM makes is *monomorphic dispatch*, which relies on the observed fact that, usually, there aren't paths through a method that cause an object reference to be of one type most of the time but of another type at other times.

You might think that having different types via different code paths would be ruled out by Java's static typing, but remember that an instance of a subtype is always a valid instance of a supertype (this principle is known as the *Liskov substitution principle*, after Barbara Liskov). This situation means that there could be two paths into a method—for example, one that passes an instance of a supertype and one that passes an instance of a subtype—which would be legal by the rules of Java's static typing (and does occur in practice).

In the usual case (the monomorphic case), however, having different, path-dependent types does not happen. So we know the exact method definitions that will be called when methods are called on the passed object, because we don't need to check which override is actually being used. This means we can eliminate the overhead of doing virtual method lookup, so the JIT compiler can emit optimized machine code that is often faster than an equivalent C++ call (because in the C++ case, the virtual lookup cannot easily be eliminated).

Java HotSpot VM uses many other techniques to optimize the code that JIT compilation produces. *Loop optimization*, *type sharpening*, *dead-code elimination*, and *intrinsics* are just some of the other ways that Java HotSpot VM tries to optimize code as much as it can. Techniques are frequently layered one on top of another, so that once one optimization has been applied, the compiler might be able to see more optimizations that can be performed.

Compilation Modes

Inside Java HotSpot VM, there are actually two separate JIT compiler modes, which are known as *C1* and *C2*. C1 is used for applications where quick startup and rock-solid optimization are required; GUI applications are often good candidates for this compiler. C2, on the other hand, was originally intended for long-running, predominantly server-side applications. Prior to some of the later Java SE 7 releases, these two modes were available using the -client and -server switches, respectively.

The two compiler modes use different techniques for JIT compilation, and they can output very different machine code for the same Java method. Modern Java applications, however, can usually make use of both compilation modes. To take advantage of this fact, starting with some of the later Java SE 7 releases, a new feature called *tiered compilation* became available. This feature uses the C1 compiler mode at the start to provide better startup performance. Once the application is properly warmed up, the C2 compiler mode takes over to provide more-aggressive optimizations and, usually, better performance. With the arrival of Java SE 8, tiered compilation is now the default behavior.

CHOOSE YOUR MODE

The two Java HotSpot VM compiler modes use different techniques for JIT compilation, and they can output very different machine code for the same Java method. **Modern Java applications, however, can usually make use of both compilation modes.**

Java HotSpot VM has the ability to produce a more detailed log of compilation events. Let's move on to see how to enable the production of such a log.

Full Logging of JIT Compilation

The switch for enabling full logging is -XX:+LogCompilation, and it must be preceded by the option -XX:+UnlockDiagnosticVMOptions. Using the -XX:+LogCompilation switch produces a separate log file, hotspot_pid<PID>.log, in the startup directory. To change the location of the file, use -XX:LogFile=<path to file>.

LISTING 3

```
<nmethod compile_id='2' compiler='C1' level='3'
entry='0x00000001023fe240' size='1224'
address='0x00000001023fe0d0' relocation_offset='288'
insts_offset='368' stub_offset='880' scopes_data_offset='1032'
scopes_pcs_offset='1104' dependencies_offset='1200'
nul_chk_table_offset='1208'
method='java/lang/String hashCode ()I' bytes='55' count='512'
backedge_count='8218' iicount='512' stamp='0.350'/>
```

Download all listings in this issue as text

The output log is a large XML file (often comprising dozens or hundreds of megabytes), containing a high level of detail about the decisions the Java HotSpot VM compilers made. This log contains a lot more information than the simple format we discussed above.

Listing 3 is a sample entry from the detailed compilation log, and it contains a lot of detail about the compilation decisions that Java HotSpot VM made when compiling the method—in this case, the method String::hashCode(). However, the format is complex and difficult to work with. This presents a barrier for many developers, which means that they can't use the detailed logs to understand their applications. Fortunately, help is at hand.

In Part 2 of this series, we will introduce JITWatch, a new open source tool that can consume the detailed compilation logs and provide simple, graphical visualizations of many aspects of JIT compilation. You can download JITWatch from GitHub, which is where continued development of the tool takes place.

Conclusion

In this article, we have introduced some of the basic concepts of JIT compilation as deployed in Java HotSpot VM. We have illustrated the flags needed to produce compilation log output—both the compact format and the more extensive XML output. In doing so, we have paved the way to discuss a new visualization tool in Part 2 of this series. </article>

MORE ON TOPIC:

Java Virtual Machine

LEARN MORE

• GitHub repository for JITWatch

Production-Time Profiling with Oracle Java Mission Control

Low-overhead profiling and diagnostics for Java applications running on the JVM

MARCUS HIRT

BIO

There's a new kid on the JDK tooling block: Oracle Java Mission Control, a production-time profiling and diagnostics tool suite. Starting with the release of Java SE Development Kit 7, Update 40 (JDK 7u40), Oracle Java Mission Control is bundled with Oracle's Java HotSpot VM. This article will explain why it is worthwhile to take a closer look at this technology, as well as provide pointers on how to get started.

History

JRockit Mission Control originated as part of the development effort for JRockit—a proprietary Java Virtual Machine (JVM)—to provide tooling for analyzing runtime performance. The JRockit team needed information about how real production systems were using JRockit. Requests to get customers to lend us their latest top-secret trading applications for in-house evaluation were usually, understandably, met with blank stares calling our sanity into question. Their applications probably wouldn't have done us much good anyway, because we wanted production data from systems under real loads. Thus, we decided to build a tool (the JRockit Runtime Analyzer, which later evolved into the JRockit Flight Recorder) with low enough overhead that we could convince customers to actually use it to collect production data for us.

Eventually, we accidentally solved some customer problems using the tool, and customers started asking if they could license the tool. Thus, the idea arose to spend some more resources on the tool to create a commercial tool that would pay for its own development costs, and JRockit Mission Control was born.

After Oracle acquired Sun Microsystems, Oracle suddenly had two of the top three most commonly used general-purpose JVMs on the market. One (the HotSpot JVM) was the open source reference JVM, for which many people knew the codebase and on which licensees based their own versions of the JVM. The other (the JRockit JVM), while being a quick and pretty little thing, was proprietary and only a small number of people knew the codebase. Instead of having to support two JVMs, Oracle wanted to pool the available resources to build a best-of-breed JVM. It was decided that the base would be Java HotSpot VM and that the most-useful features in JRockit would be ported over—one of them being JRockit Mission Control.

ONE PLACE

When profiling capabilities or better diagnostic information is needed, the one-stop shop is really the Java Flight Recorder.

In JDK 7u40, the functionality available in Java HotSpot VM reached critical mass, and the first version of Oracle Java Mission Control was released. It mainly contains the equivalents of two of the JRockit Mission Control tools: the Java Management Extensions (JMX) Console and the Java

Java Virtual Machine

Flight Recorder. There is no online heap analyzer yet. There is, however, a set of quite useful (experimental) plugins that extend Oracle Java Mission Control to do heap dump analysis, do targeted analysis for various Oracle products, or simply extend existing Java Mission Control functionality in more-useful ways.

Getting Started with Oracle Java Mission Control

Starting Oracle Java Mission Control is quite easy. Download and install a recent-enough Java SE JDK (7u40 or later), and then simply run %JDK_HOME%/bin/jmc. The alien thing that starts is not, as I am sometimes asked, a native application. It's Java, but it's built upon Eclipse RCP technology. If you would rather run Oracle Java Mission Control inside the Eclipse IDE, you can install it into your Eclipse from the Oracle Java Mission Control site.

The JMX Console. The console in Oracle Java Mission Control can be thought of as a JConsole on steroids. As shown in **Figure 1**, it allows you to monitor JMX data in various ways, to take action when attributes attain certain values, and to persist the data and later look at what was recorded. There are various experimental plugins for the console, such as an Oracle Coherence plugin, a plugin for running JConsole plugins, and a plugin for tweeting messages when an action is triggered.

To connect the console to a JVM, simply choose the JVM process you want to connect to in the JVM browser tree and select **Start JMX Console**. JVM processes will appear automatically in the JVM browser tree if the JVM is started locally or with the Java Discovery Protocol (JDP). If you have a remote JVM without JDP running, just enable the built-in jmxrmi agent as you normally would to be able to connect with JMX clients such as JConsole.

The console is typically used to monitor a small set of critical attributes, such as the CPU load and Java heap usage, that are sampled at a relatively low frequency. The console can be configured to take action when attributes reach an undesirable value, and one of those actions can be to dump Java Flight Recorder data. The console also contains special tabs for looking at thread information, such as deadlocked threads, per-thread allocation information, and per-thread profiling information. That said, the

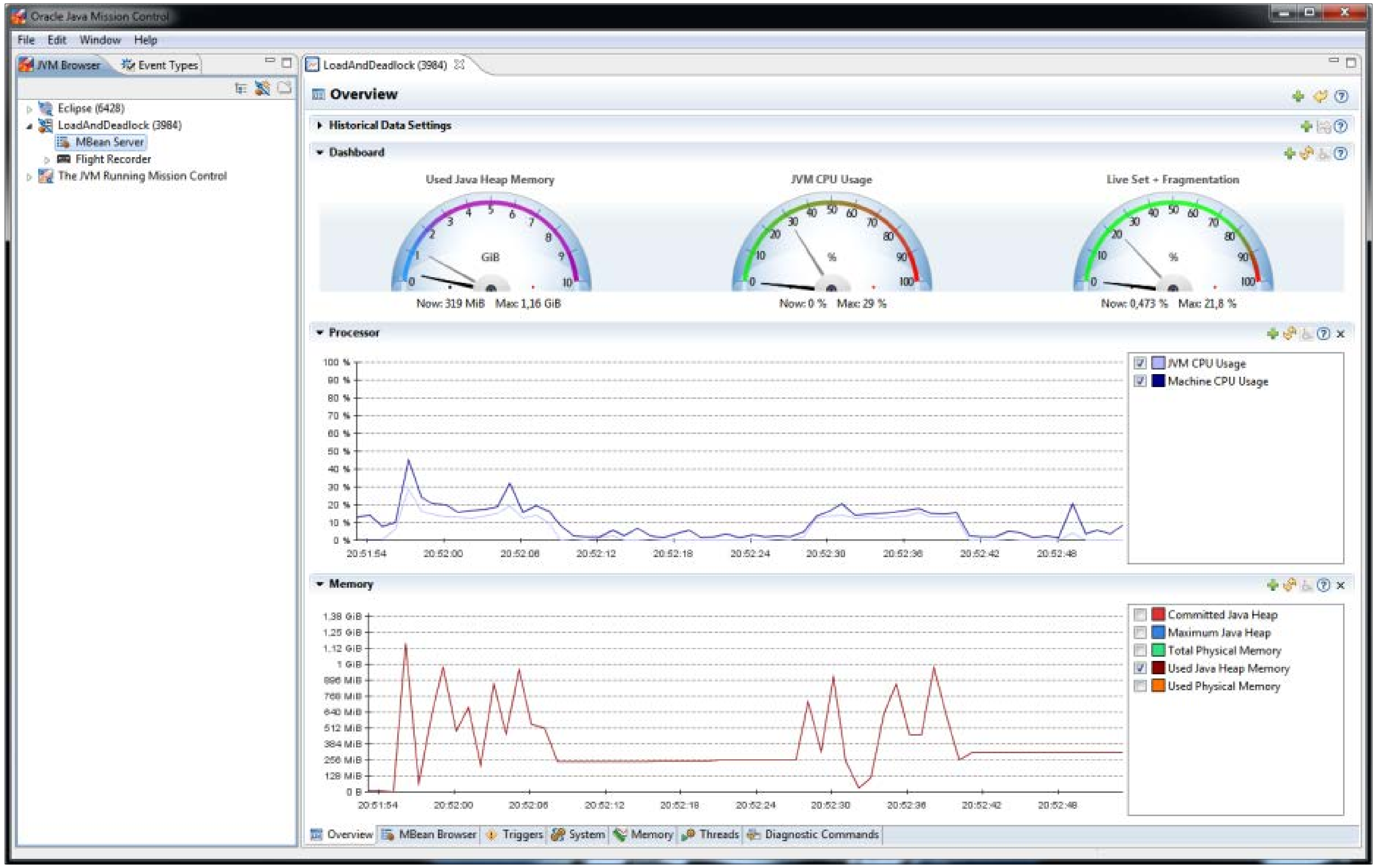


Figure 1

console is used for monitoring the runtime. When profiling capabilities or better diagnostic information is needed, the one-stop shop is really the Java Flight Recorder.

The Java Flight Recorder. The Java Flight Recorder can be thought of as the equivalent of an airplane's flight data recorder for the Java runtime. While it is running, it records information about the JVM and its environment. When something "interesting" happens, the data in the Java Flight Recorder can be dumped, and the information can be analyzed offline to gain an understanding of why things suddenly went from good to "interesting." Running the Java Flight Recorder has an almost unnoticeable impact on the performance of a Java application running in the JVM. The overhead is usually well below one percent. This is achieved by a high-performance recording engine built directly into the runtime, which collects data that is already being tracked by the runtime or is already being generated by another activity (as opposed to actively having to do additional work to get the data). There are a lot of interesting things that can be said about the recording engine implementation, but because this is an overview article, I'll move on to how to use it.

PUMPED-UP

The console in Oracle Java Mission Control can be thought of as a **JConsole on steroids.**

Creating Flight Recordings

The most important difference between how the recorder worked in JRockit and how it works in Oracle Java Mission Control is that in Java HotSpot VM, two JVM startup flags must be enabled on the JVM for which you want to do flight recordings: -XX:+UnlockCommercialFeatures and -XX:+FlightRecorder.

That was probably the most important line in this article.

There are two different types of recordings, and you can have multiple recordings (of different types) running simultaneously:

- Timed recordings. These recordings run for a preconfigured duration. They are automatically stopped when the time is up. If they are initiated from Oracle Java Mission Control, they are automatically downloaded and opened in the Java Flight Recorder user interface when they are done.
- Continuous recordings. These recordings have no explicit end time and must be dumped by the end user.

There are three different ways you can do actual recordings, once the parameters are in place:

- From Oracle Java Mission Control. This is probably the easiest way. Just point and click.
- From jcmd. This is a way to control the Java Flight Recorder from the command line, which is quite useful when you can't access the machine that is running the JVM of interest from Oracle Java Mission Control and you only have access to a shell.
- Using command-line flags. This is handy when you want to always run with a continuous recording or when you want to record the behavior of the JVM right from the very start.

Figure 2 shows some example recordings.

Analyzing Flight Recordings

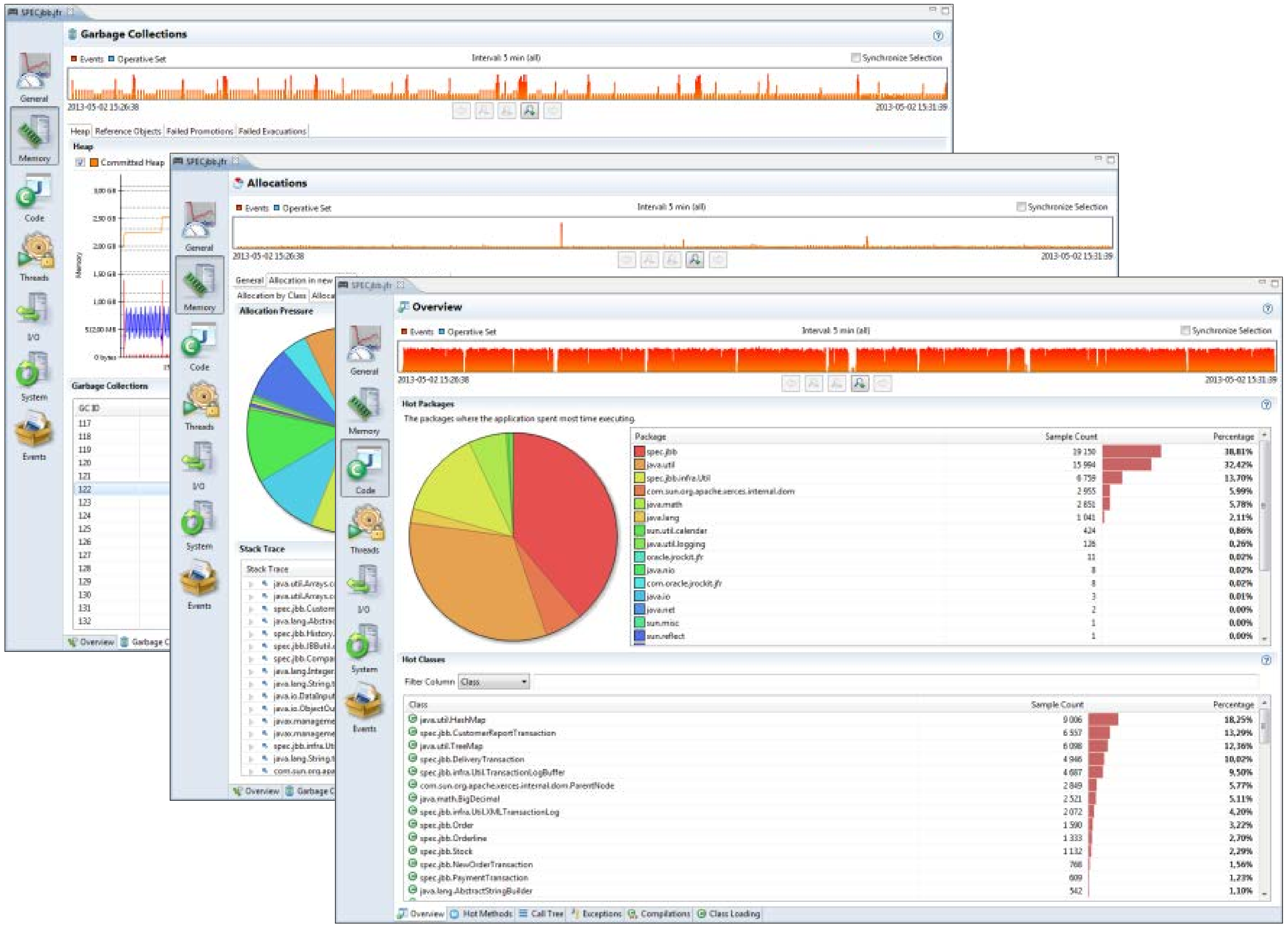
There is a lot of useful information in the flight recordings, and there are a lot of different things the information can be used for, for example:

- Method profiling. The Java Flight Recorder will quite happily do method profiling on production systems while causing very low overhead. As a matter of fact, it's even enabled in the continuous template, so go ahead and use it. It will tell you where the hotspots are in your application. In other words, if you have a CPU-bound problem, the method profiling information will tell you where to optimize to get things to go faster.
- Garbage collection (GC) profiling. The GC implementations emit useful events about GC-related activity, such as information that can be used to check on the live set, semireferences, GC pauses (and their individual phases), and so on. This is quite useful for GC tuning, finding out if you're overusing finalizers, and more.
- Allocation profiling. If you notice a lot of garbage collection, but you don't notice anything strange about the individual GC phases, you might want to reduce the allocation a bit. Allocation profiling will help you see where all that allocation activity is putting a toll on the memory system.
- Oracle WebLogic Server analysis. Oracle WebLogic Server produces its own set of events for the Java Flight Recorder. They are quite useful in their own right, but they can also be good for putting all the other recorded information into a context, for example, to see what was really happening during a transaction. This article on the Operative Set feature of the Java Flight Recorder shows some of the capabilities.
- Latency profiling. The Java Flight Recorder has many different

events for various thread-stalling activities that can occur, such as blocking on entering a monitor, parking, waiting, and so on. Latency profiling is usually the first place to look if you do not have a CPU-bound problem, but you still have performance issues.

- OS information. There is a lot of operating system information, for example, information about CPU load, JVM CPU load, environment variables, and running processes. If you still can't find what you're looking for, Oracle Java Mission Control has a DTrace plugin for retrieving everything you ever wanted to know but were too afraid to ask. Note that the overhead for using DTrace, even with very few probes, is usually more than an order of magnitude higher than the overhead for using the Java Flight Recorder, so use with caution.

Much more information is available from the event providers built into the JVM, such as class loading and compiler events. One way to learn more about what is available is to take a closer look at the metadata from a recording.



Package	Sample Count	Percentage
spec.jbb	19 150	38,81%
java.util	15 994	32,42%
spec.jbb.infra.Util	6 759	13,70%
com.sun.org.apache.xerces.internal.dom	2 955	5,99%
java.math	2 851	5,78%
java.lang	1 041	2,11%
sun.util.calendar	424	0,86%
java.util.logging	126	0,26%
oracle.jrockit.jfr	11	0,02%
java.nio	8	0,02%
com.oracle.jrockit.jfr	8	0,02%
java.io	3	0,01%
java.net	2	0,00%
sun.misc	1	0,00%
sun.reflect	1	0,00%

Class	Sample Count	Percentage
java.util.HashMap	9 006	18,25%
spec.jbb.CustomerReportTransaction	6 557	13,29%
java.util.TreeMap	6 098	12,36%
spec.jbb.DeliveryTransaction	4 946	10,02%
spec.jbb.infra.Util.TransactionLogBuffer	4 687	9,50%
com.sun.org.apache.xerces.internal.dom.ParentNode	2 849	5,77%
java.math.BigDecimal	2 521	5,11%
spec.jbb.infra.Util.XMLTransactionLog	2 072	4,20%
spec.jbb.Order	1 590	3,22%
spec.jbb.Orderline	1 333	2,70%
spec.jbb.Stock	1 132	2,29%
spec.jbb.NewOrderTransaction	768	1,56%
spec.jbb.PaymentTransaction	609	1,23%
java.lang.AbstractStringBuilder	542	1,10%

Figure 2

Conclusion

As of JDK 7u40, a new tool suite is bundled with the JDK: Oracle Java Mission Control. The main focus of the suite is on production-time profiling and diagnostics. This focus means that the gathered data is quite true to the dynamics of the application being profiled, because the observer effect is kept quite low. In other words, instead of profiling the profiler itself, most of the time is actually spent profiling the application and the runtime.

While the main focus of Oracle Java Mission Control is production systems, it can be quite useful during development, too. It is also free for use during development, per the standard Oracle Binary Code License (BCL). </article>

MORE ON TOPIC:

Java Virtual Machine

LEARN MORE

- Oracle Java Mission Control home page

CASIMIR SATERNOS

BIO

Server-Side Deployment on the JVM

JVM deployments with the server outside, inside, and alongside a web app

Over time, the success of a web application can influence technology choices for ongoing support and customization. For example, Twitter was initially a Ruby on Rails application but the back-end Ruby services were replaced with applications running on the Java Virtual Machine (JVM) and written in Scala. With this in mind, it is important to recognize that Java development practices and the use of the JVM have expanded greatly in recent years. These changes can ease the transition from starter projects to large-scale, mature applications.

This article will demonstrate how a project can initially be developed very quickly with a view toward a variety of deployment options. More specifically, a small web application will be written in Ruby but packaged in a web application archive (WAR) file. Options for deploying the WAR file with the server outside, inside, or alongside allow such an application the type of scalability and availability demanded in modern high-traffic, cloud-based deployments. By using a standard WAR file for deployment, the application is packaged in a form that can readily be replaced with a similar package written in Java or other JVM language.

Note: For the example described in this article, you can use any Ruby implementation of your choice (either JRuby or a C-based Ruby).

Java and Alternative JVM Languages

The JVM has emerged as the preeminent language-independent virtual machine target and is optimized to run on a wide range of platforms. It has matured over the years and now supports a growing number of alternative languages that compile to JVM bytecode. Each release of the JVM includes new features that enhance the JVM's ability to support and optimize the performance of additional languages. These qualities have resulted in the popularity of the JVM and make it a serious consideration for many types of software projects.

Java is a fine general-purpose software development language, but it is not necessarily the best option for every project. JVM languages include JRuby, Jython, Groovy, JavaScript, Scala, and Clojure. These languages are impressive implementations in their own right, and all share the desirable quality of running on the JVM. During the initial phases of a project, it might be preferable to use scripting languages specifically geared for programmer productivity. By using JVM implementations, a project written in one language can be ported more easily to another, if needed at a later point.

CREATIVE TYPES

Programmers are creative people and typically **delight in contriving clever ways to solve problems.**

JRuby is one of a number of popular JVM languages in use. It is less "strict" than Java in several ways, which promotes quick initial development. But some of the features that make Ruby a popular choice for programmer productivity can be the source of problems as a project grows. Java provides type safety and the high performance required by more-mature projects. One way the process of transitioning a project from Ruby to Java can be simplified is by packaging Ruby code in a Java

WAR file. This allows the target deployment hardware and the foundational software that is installed to remain constant when the project transitions from JRuby to Java.

Deployment Options

Although much time and attention are spent during the process of software development, the end goal is deployment to a production environment for end users. This goal is best achieved by up-front planning that, at a minimum, does not result in a choice that is untenable for production usage. Deployment practices are driven by each application's unique functional and technical requirements. Target server capabilities, expected web traffic patterns, and available hardware also contribute to the decision of where an application will be deployed. After one or more deployment destinations have been selected, the manner of deployment needs to be considered.

EASY TO PORT

During the initial phases of a project, it might be preferable to use scripting languages specifically geared for programmer productivity. By using JVM implementations, a project written in one language can be ported more easily to another, if needed at a later point.

Programmers are creative people and typically delight in contriving clever ways to solve problems. This quality is commendable when addressing end users' requirements, as long as standards and conventions provide the structure for the work. Build and deployment tasks are not the place for creativity. These tasks do not directly meet end user needs and should be completed as efficiently and consistently as possible. Creativity in build and deployment practices results in highly customized unique projects that are difficult to support and modify. In the end, this type of "snowflake" project automation hampers the development efforts it is meant to promote.

Server outside. Traditionally, web application servers were installed independently and configured by system administrators. WAR files or enterprise archive (EAR) files were then deployed to these servers after all installation and configuration was completed. The server was *outside* the web application. This method is still a great way to deploy applications and conforms with Java EE standards that are well established and understood among Java developers.

However, one challenge with this method is that each developer needs an application server for local testing. Unless an application server is readily available on a preconfigured virtual machine, individual developers are required to configure their own workstations. This process might be scripted and straightforward, but in some cases, developers are forced to cobble together a configuration whose description is buried in e-mail messages and then debug error-log messages to resolve configuration issues.

Another challenge is that developers might use different processes to deploy an application. One developer might use a web interface, another might copy the file into a directory that is "watched" by the application server, and another might run a utility or homegrown script to deploy the application. Such inconsistencies in the deployment process can frustrate attempts to have clear communication among developers.

These problems can be overcome by incorporating deployment functionality within the project itself. For instance, Maven-based projects can use standard plugins (for example, for Oracle WebLogic Server) that reduce the deployment process to a single command. This is an improvement that provides consistency and control among developers.

Server within a development project. There are also build plugins that embed a server, such as Jetty or Apache Tomcat, within a plugin used by the project. This approach eliminates the need to install an independent web application server. The server (and any associated configuration) can be maintained within version control. If the server is updated, all developers have the updated version the next time they retrieve the latest code.

This solution is specifically geared toward developers, however, not the actual deployment. It is unusual to deploy an application that runs on a server provided through a Maven plugin. In general, teams that use such plugins still deploy to an external server for centralized testing efforts and final production. A server should not be coupled to a project build tool when deployed to a production environment.

Server inside and alongside. This leads to two more-recent solutions: embed the server *inside* the application archive itself or deploy the server *alongside* the application. A server can be included inside the application code and controlled within this context. This method reduces the deployment requirements to a single, standalone archive that can be invoked using the java -jar command. An alternative is to keep the application server code separate, and include it alongside the actual web application. This method is typically performed by passing the application server as the first argument to the java -jar command and passing the web application as the second argument.

Example Project

The following example project will demonstrate the three methods of deployment (outside, inside, and alongside) using a minimal example written in Ruby. This project is intentionally minimal to focus on the specific concerns at hand. It shows how to create a non-Java-based web application that can be packaged in a WAR file and deployed using a server running outside, inside, or alongside the application.

Ruby packages (analogous to Java archive [JAR] files) are called *gems*. Two gems are required by the web application created in this project. The web application is written using the Sinatra gem, which is a domain-specific language (DSL) for web development. The JSON gem is used to illustrate the ease of creating a web API. The project will demonstrate that both Ruby code and gems can be packaged together for distribution in a single WAR file.

Write the application. Create a new directory whose name will be used as the name of the WAR file:

```
mkdir testwebapp
cd testwebapp
```

Next, include the bundler gem to provide dependency management similar to what Java developers know from tools such as Maven or Ivy:

```
gem install bundler
```

Note: The implementation of Ruby in use at this point does not need to be JRuby. Another gem called Warbler includes JRuby and will be used when the WAR file is created.

With bundler installed, run the following command to create a configuration file called a *Gemfile*.

```
bundle init
```

The contents of this file can then be populated to reference the two dependent gems:

```
source "https://rubygems.org"
gem "json","1.8.0"
gem "sinatra","1.4.3"
```

LISTING 1

```
require 'sinatra'
require 'json'

class Hiwar < Sinatra::Application
  get "/" do
    "<h1>Hello World</h1>"
  end

  get "/json" do
    content_type :json
    {:greeting => 'Hello', :audience => 'World'}.to_json
  end
end
```

Download all listings in this issue as text

Download and install the gems locally by running the bundle command:

```
bundle
```

Next, create a Ruby script named webapp.rb in a directory named bin:

```
mkdir bin
touch bin/webapp.rb
```

Edit this file and add the Ruby code to create the web application, as shown in **Listing 1**. The code includes the two gems referenced earlier, and it creates a web application class that responds to two URL paths.

Next create a config.ru file that will be used as the entry point for running a Sinatra web application:

```
touch config.ru
```

Add the required libraries along with a command to kick off the web application server:

```
require 'rubygems'
require 'bin/webapp'

run Hiwar.new
```

Package the application. Use Warbler to package the Ruby code in a JAR file:

```
gem install warbler
```

Create a configuration file to list gem dependencies that are intended to be packaged:

```
mkdir config
warble config
```

Replace the contents of config/warble.rb to indicate which directories and gems should be packaged in the WAR file:

```
Warbler::Config.new do |config|
  config.dirs = %w(config bin)
  config.gems += ["json",
 "sinatra"]
end
```

Run the warble command to create the actual WAR file:

```
warble executable war
```

Use the standard Java SDK jar utility to view the contents of the newly created WAR file:

```
jar tf testwebapp.war
```

Deploy and run the application. The newly created WAR file can be deployed to an existing, previously installed web application server. The server is outside the web application at the time of development, so initial deployment involves accessing a web page, running a utility, or simply copying the file. Tomcat, for instance, can be installed at an arbitrary location. In the example below, the CATALINA_BASE environment variable can point to a directory such as /opt/local/apache-tomcat-6.0.18.

The server can be started by running a script, for example:

```
$CATALINA_BASE/bin/startup.sh
```

The archive can be deployed by copying it to the required directory:

```
cp testwebapp.war \
$CATALINA_BASE/webapps/
```

With the application running, you can use a browser or a utility such as curl to execute requests at the two configured URLs:

```
curl http://localhost:8080/json
curl http://localhost:8080/
```

The WAR file is packaged in such a way that it can be run using an embedded server. The server is inside the web application during development and deployment.

```
java -jar testwebapp.war
```

There are limitations to including a server inside the application. For example, any change in configuration will require an entirely new build and deployment cycle. Instead, the server can be deployed alongside the application using Jetty Runner, as shown in **Listing 2**.

Jetty Runner is an independent JAR file that can be upgraded or configured without the need to change and repackage application code, as shown in **Listing 3**.

LISTING 2 LISTING 3

Note: The following command has been broken into multiple lines for readability. It must be executed in a single line.

```
curl -O http://repo2.maven.org/maven2/org/mortbay/jetty/
jetty-runner/8.1.9.v20130131/jetty-runner-8.1.9.v20130131.jar
```

Download all listings in this issue as text

Conclusion

It is difficult to predict the future in any realm of life. A software project can span years of time, be developed by teams of developers, and be used by a broad audience of users. The specific purpose of an application can vary significantly as a business pivots or changes focus. The amount of web traffic an application gets or changes in the development team can affect technical and deployment needs.

The JVM is an effective piece of technology that provides a wide range of options for subsequent growth due to its cross-platform, multilanguage support. This flexibility, along with the established standard of using WAR files for web application deployment, reduces the need to attempt to predict how a web application will fare in the future, freeing you to focus on more-pressing business and technical concerns at hand. </article>

MORE ON TOPIC:

Java Virtual Machine

LEARN MORE

- Ruby programming language
- Warbler: A gem to make a Java JAR or WAR file out of any Ruby, Rails, or Rack application

Introduction to the Java Temporary Caching API

Use a caching strategy without worrying about implementation details.

JOHAN VOS

BIO

Most enterprise and web applications use some method of caching. If an application receives a large number of requests, it is often beneficial to store some of the responses in a cache. Then, under certain conditions, the same response can be sent to answer similar requests. Rather than redoing all the processing and computations, returning the previously cached response can save lots of CPU resources.

A number of commercial solutions provide cache implementations to developers. Different solutions often come with different topologies, concepts, and features. Fortunately, the Java Temporary Caching API allows Java developers to use a common approach for working with caches, without having to worry about implementation details. This article provides an introduction on how to leverage the Java Temporary Caching API in Java applications.

Java specifications are defined by the Java Community Process (JCP) organization. For each specification, a Java Specification Request (JSR) is submitted. The development of Java specifications follows a number of well-defined steps, starting with the formation of an Expert Group and ultimately resulting in the availability of a specification, a Reference Implementation, and a test suite.

Sometimes, the transition between the formation of the Expert Group and the final approval of the specification happens very quickly. In other cases, it takes a bit longer. The Java Temporary Caching API, JSR 107, was considered an important area for standardization in 2001. However, it wasn't until March 18, 2014, that the specification was finally released.

The good thing about the long time between the start of the standardization effort and the final release is that all the experience gathered by the Expert Group members and the community was taken into account in the specification. Although the terminology slightly varies among the different producers of cache software, there are enough common concepts that are now captured in JSR 107.

While caching is often linked to database calls—that is, the result of a database call is often stored in a cache—it should be stressed that caching concepts are much broader than database applications. Indeed, a cache can also be used to store images, responses from web services, and so on. The most generic representation of an entry that can be cached, as defined by JSR 107, is a Java object.

In this article, we will see how to use the Java Temporary Caching API with the Amazon DynamoDB service. However, the same concepts apply to most relational and nonrelational databases, as well as to any services that result in a Java object being produced.

Note: The source code for the sample application developed in this article can be downloaded here.

Amazon DynamoDB

Amazon DynamoDB is a cloud-based NoSQL data store provided by Amazon.

PHOTOGRAPH BY TON HENDRIKS

An overview of the DynamoDB features is outside the scope of this article, and the Java Temporary Caching API can be demonstrated with other NoSQL or SQL data storage systems as well.

One of the nice things about DynamoDB is the availability of a local version that can easily be used by developers during testing. We will use this local version in our examples. DynamoDB Local can be downloaded by following these instructions.

The downloaded Java archive (JAR) file can easily be started using the command shown in **Listing 1**.

This command will start the local version of DynamoDB and cause it to run in memory instead of using a database file. As a consequence, all data supplied to DynamoDB Local will be kept in memory and will be lost when the server is stopped. This is by no means how the cloud version of DynamoDB works. However, the APIs required for accessing the local version and the cloud version are the same. The cloud offering requires more configuration and, unlike the local version, is not a free service. Therefore, DynamoDB Local is very popular with software engineers during the code development process.

The Amazon Web Services SDK (AWS SDK) contains a number of APIs that allow developers to communicate with DynamoDB. In our examples, we will use the high-level API, which provides a direct mapping between Java objects and entries in DynamoDB. In order to do so, we will use a few annotations on the Java objects we want to store.

The Sample Application

While most of the applications that will leverage the Java Temporary Caching API are probably enterprise applications, the API itself is designed to work in a Java SE environment as well. For simplicity, we will write our examples on top of the Java SE 8 platform.

Our sample application will create a number of Person instances. The code for the Person class is shown in **Listings 2a** and **2b**. The Person class is a typical Java class containing fields for the first name, the last name, and the age of a person. Additionally, we have a myKey field that will be used for storing a primary key.

The annotation @DynamoDBTable(tableName = "Person") is used to associate the Person class with the "Person" table that we will create in DynamoDB. Further, the @DynamoDBHashKey annotation indicates that the annotated method (getMyKey) returns the hash key for the object in DynamoDB.

LISTING 1 LISTING 2a LISTING 2b

```
java -Djava.library.path=./DynamoDBLocal_lib
-jar DynamoDBLocal.jar -inMemory
```

Download all listings in this issue as text

The examples we will create will perform three steps:

1. Create a datastore.
2. Populate the datastore.
3. Query the datastore.

The main method for our sample application is very simple (see **Listing 3**).

In our first example, we will not use any caching. We will write all data directly to DynamoDB, and when querying data, we will directly query DynamoDB as well.

The createDatabase function will create the DynamoDB datastore. The code in **Listing 4** demonstrates how a DynamoDB datastore is created. We won't go into DynamoDB-specific details, but on a high level, the createDatabase call does the following:

- Creates credentials (key and secret) for communicating with Amazon DynamoDB. For the DynamoDB Local version, these credentials don't matter, although they have to be supplied.
- Creates an AmazonDynamoDBClient and a DynamoDBMapper instance. We will use the DynamoDBMapper instance later.
- Defines the key that will be used as the primary key for indexing the data.
- Specifies the requested throughput (for read and write operations). Again, these values are irrelevant for the DynamoDB Local version, but they have to be provided.
- Deletes the table "Person," if it exists.
- Creates the table "Person."

Now that a table has been created, and a DynamoDBMapper object has been constructed, we can populate the table.

We will create 1000 instances of Person with random content. An index ranging from 0 to 999 will be used to set the myKey field for these instances. The code in **Listing 5** populates the table with these instances. Using the DynamoDB high-level API, storing an object is as simple as calling the save method on a DynamoDBMapper instance and providing the object that we want to store.

We will now query DynamoDB to ask for Person instances with a specific key, as shown in **Listing 6**. We will make 1000 requests to the DynamoDBMapper. Each request asks the DynamoDBMapper to find the Person instance corresponding to a specific key. The key is a random value between 0 and 1099, whereas the keys of the stored Person instances range between 0 and 999. As a consequence, we expect about 10 percent of the requests to result in an empty answer.

LISTING 3 / LISTING 4 / LISTING 5 / LISTING 6

```
public static void main(String[] args) {
   createDatabase();
   populateDatabase();
   queryDatabase();
}
```

Download all listings in this issue as text

Introducing the Java Temporary Caching API

Until now, all requests for storing and retrieving data have gone directly to the DynamoDB system. At this point, we will introduce the Java Temporary Caching API. Before we look at the code, here are the main classes defined by JSR 107:

- CacheProvider, which contains an implementation of the Java Temporary Caching API
- CacheManager
- Cache
- Entry
- Expiry

We will now modify the code used to populate the datastore and introduce the caching features. The modified code is shown in **Listing 7**. Before we populate the datastore, we create a Cache instance. First, we have to obtain a reference to the CachingProvider. This is done by the code shown in **Listing 8**.

The Java Temporary Caching API is defined by JSR 107, but the API itself does not contain a concrete implementation. At runtime, the API will use the Java ServiceLoader to check whether a CachingProvider implementation is available on the classpath.

In our example, we use the Reference Implementation of JSR 107, which is available in Maven Central. The JAR file containing the Reference Implementation contains a META-INF/services directory containing a reference to the CachingProvider implementation.

If another JSR 107–compliant implementation is available on the classpath, that implementation will be used. Our code doesn't have to change, though, because we are using only the APIs provided by the JSR 107 specification. We are not using implementation-specific APIs or features.

COMMON APPROACH

The Java Temporary Caching API allows Java developers to use a common approach for working with caches, **without having to worry about implementation details.**

Once we have a CachingProvider, we need to get a CacheManager. This is done using the code shown in **Listing 9**.

Now that we have a CacheManager instance, we can create a Cache instance. Our cache needs to hold instances of Person. A Cache contains entries that have a key and a value. In our case, it makes sense to use the myKey field of the Person as the key and to use the Person itself as the value. The code in **Listing 10** creates our cache and provides the required configuration.

The configuration of the cache is defined in a MutableConfiguration instance. The MutableConfiguration class allows you to set a number of configuration policies by using the fluent API . The fluent API implies that the result of a set method is the instance itself.

In our example, we set the following:

- The setTypes statement in **Listing 10** specifies that the key for our cache is of type Long, and the value is of type Person.
- The setExpiryPolicyFactory statement defines how long cache entries are valid.

In order to create a Cache instance, we call the createCache method on the CacheManager instance and supply the name of the cache (personCache) and the configuration object.

Apart from creating the cache, we add only one line of code. In the for loop where the Person instances are created, we add the following line:

```
cache.put(person.getMyKey(),
person);
```

LISTING 7 / LISTING 8 / LISTING 9 / LISTING 10

```
  private static void populateDatabaseCache() {

     CachingProvider cachingProvider =
Caching.getCachingProvider();
     CacheManager cacheManager =
cachingProvider.getCacheManager();

     MutableConfiguration<Long, Person> config
         = new MutableConfiguration<Long, Person>()
         .setTypes(Long.class, Person.class)
         .setExpiryPolicyFactory(
AccessedExpiryPolicy.factoryOf(ONE_HOUR))
         .setStatisticsEnabled(true);

     cache = cacheManager.createCache("personCache", config);

     Random age = new Random();
     DynamoDBMapper mapper = new DynamoDBMapper(client);
     for (long i = 0; i < 1000; i++) {
       String f1 = UUID.randomUUID().toString();
       String f2 = UUID.randomUUID().toString();
       Person person = new Person(
i, f1, f2, age.nextInt(100));
       mapper.save(person);
       cache.put(person.getMyKey(), person);
     }
  }
```

Download all listings in this issue as text

This line actually stores the Person instance in the cache. However, a typical issue with caches is that you never know for sure whether the data you added to the cache is still in the cache. In our configuration, we explicitly stated that the cache entries should be in the cache for at most one hour. However, the cache implementation might decide to remove entries from the cache sooner than that, for example, when the cache reaches a critical size. The maximum cache size can be very complex, depending on the implementation architecture. Explicitly setting the maximum cache size often doesn't make sense, because there are many internal and external parameters that define the optimal cache size. Hence, JSR 107 does not provide a method for setting the maximum cache size.

When retrieving data from the cache, we have to take into account that the cache might not contain the data anymore. In **Listing 6**, we queried DynamoDB and asked whether it has a Person object corresponding to a specific key. That key is a Long value between 0 and 1099. If the key is 1000 or more, DynamoDB will tell us it can't find a corresponding value, and then we know that there is no Person associated with this key. However, we can't apply the same logic when using the cache. It is very possible that the cache doesn't contain a Person instance belonging to a specific key, but DynamoDB might have this Person instance. This will be the case for Person instances that are evicted from the cache.

We ask the cache for a specific Person instance using the following method:

```
Person p = cache.get(KEY)
```

If this method returns a Person instance, we're done. If this method returns null, however, the DynamoDB datastore might still have the Person instance. In that case, we apply the same code as in **Listing 6**.

Using Read-Through and Write-Through Caches

So far, we used the cache and DynamoDB independent from each other. We stored the Person instances in the cache as well as storing them manually in DynamoDB. And when retrieving data, we first asked the cache, and only when that failed did we query DynamoDB.

Typically, caches also allow read-through and write-through operations. Simply stated, in a write-through operation, data that is written to a cache is also made persistent (for example, via DynamoDB), and in a read-through operation, data read from the cache is retrieved from persistent storage if it isn't in the cache.

An advantage of this approach is that the application code does not need to query the persistent storage itself each time the cache fails to return the requested object.

JSR 107 allows developers to leverage this read-through and write-through functionality. We will modify the previous example and show how we can achieve this.

Enabling write-through and read-through functionality is part of the configuration process. The MutableConfiguration object that we used before to specify the configuration of the cache can also be used to specify the write-through and read-through behavior. Specifying write-through is done as shown in **Listings 11a** and **11b**.

As you can see, we use the MutableConfiguration.setCacheWriterFactory() method to provide a Factory for a CacheWriter. This Factory provides a CacheWriter class. We create our own extension of the CacheWriter class,

LISTING 11a LISTING 11b

```
config.WriteThrough(true);
config.setCacheWriterFactory(new Factory<CacheWriter< Long, Per-
son>>() {
  @Override
  public CacheWriter<Long, Person> create() {
    CacheWriter<Long, Person> writer = new CacheWriter<Long, Person>()
{

      @Override
      public void write(Cache.Entry<? extends Long, ? extends Person> en-
try) throws CacheWriterException {
        mapper.save(entry.getValue());
      }

      @Override
      public void writeAll(Collection<Cache.Entry<? extends Long, ? extends
Person>> clctn) throws CacheWriterException {
        clctn.forEach((Cache.Entry ce) -> mapper.save(ce.getValue()));
      }
```

Download all listings in this issue as text

and we map the methods in this CacheWriter to API calls to DynamoDB. Each time an entry is added to, updated in, or deleted from the cache, the CacheWriter methods are called. As a consequence, we can execute all storage tasks on the cache only. Thanks to the CacheWriterFactory, changes in the cache will be written to DynamoDB as well.

Similar to a CacheWriterFactory, we can also specify a CacheLoaderFactory. The CacheLoader that is returned by the CacheLoaderFactory is responsible for querying DynamoDB if an entry is requested from the cache but is not found there. In this case, the overridden implementations from the CacheLoader will send queries to DynamoDB.

The code in **Listing 12** adds the read-through behavior on the MutableConfiguration instance we used before. As you can see, the load method in the CacheLoader does exactly what we did in **Listing 6**. It creates a query for DynamoDB and tries to retrieve the Person using the provided key.

Using the CacheLoader—and, hence, leveraging the read-through behavior—often makes it easier for developers to focus on the application logic rather than on the cache logic. Without read-through, a developer has to query the persistent storage every time a query on a cache returns no result. By using a CacheLoader, that behavior is delegated to the CacheLoader, and it needs to be coded only once.

NOT JUST JAVA EE

While most of the applications that will leverage the Java Temporary Caching API are probably enterprise applications, **the API itself is designed to work in a Java SE environment as well.**

Conclusion

In this article, we only scratched the surface of JSR 107. The specification provides a common approach to functionality offered by most caching providers. We covered only simple get and store operations, and we briefly touched on the concepts of write-through and read-through. There is much more to discover, and the interested reader is referred to the Javadoc for more information.

The most important achievement of JSR 107 is that application developers can now use a caching strategy in their applications, independent of a specific implementation. Some providers of cache software already have a JSR 107–compliant implementation, and it is expected that more will follow. During development, developers are encouraged to use only the functionality provided by JSR 107. At runtime, a concrete implementation (free or commercial) is added to the classpath, and it will automatically be used. </article>

LISTING 12

```
config.setReadThrough(true)
config.setCacheLoaderFactory(new Factory<CacheLoader<Long, Per-
son>>() {
  public CacheLoader<Long, Person> create() {
    return new CacheLoader<Long, Person>() {
      public Person load(Long k) throws CacheLoaderException {
        Person hashKeyValues = new Person();
        hashKeyValues.setMyKey(k);
        DynamoDBQueryExpression<Person> queryExpression = new Dynam
oDBQueryExpression<Person>().withHashKeyValues(hashKeyValues);
        List<Person> itemList = mapper.query(Person.class, queryExpression);
        if (itemList.size() == 0) {
          return null;
        } else {
          return itemList.get(0);
        }
      }
      public Map<Long, Person> loadAll(Iterable<? extends Long> itrbl)
throws CacheLoaderException {
        Map<Long, Person> answer = new HashMap<>();
        itrbl.forEach((Long k) -> answer.put(k, load(k)));
        return answer;
      }
    };
  }
})
```

Download all listings in this issue as text

LEARN MORE

- Amazon DynamoDB
- JSR 107
- Specification and Reference Implementation

JOSH JUNEAU

BIO

JavaFX with Alternative Languages

Alternative languages and custom APIs enable rapid JavaFX application development.

In this day and age, the more tools you have in your toolbox, the better. That is, it makes sense to know more than one way to implement a solution, because in some cases one implementation might have advantages over others. Although the JavaFX 8 API is very nice, there are advantages to knowing more than one way to develop a JavaFX application.

This article demonstrates how to develop JavaFX applications using alternative languages, so that you can add some additional tools to your toolbox. In the first section of this article, we'll take a look at a JavaFX drawing application, and we will compare its implementation across a couple of different languages. In the second section, we will take a look at GroovyFX, a JavaFX API for the Groovy language, which is focused on rapid JavaFX application development.

Note: The source code for this article can be found on GitHub.

Bare-Bones JavaFX Development with Alternative Languages

In this section, we will not use any special APIs, but rather, we will implement a simple JavaFX application using only the standard features of a couple of different languages.

The application in this section is written using JavaFX 8 without FXML, and it is a drawing application. It allows the user to select from a handful of different colors, change the size of the pen, and reset the canvas to do it all over again. The application contains two ChoiceBox nodes for choosing the color and size of the pen, a Canvas to draw upon, and a Button to reset the canvas. **Figure 1** shows what the application looks like when it is executed (and a great artist has spent some time sketching).

Take a look at **Listing 1**, which contains the complete code for the application, written using the JavaFX 8 API. As with all JavaFX applications, it is launched via the execution of a main() method, and the start() method contains the primary stage construction and much of the application implementation.

To note a few pieces of code in particular, let's first take a look at the **Reset** button event handler, because it is implemented using a lambda expression. When the user clicks the **Reset** button, the canvas is cleared.

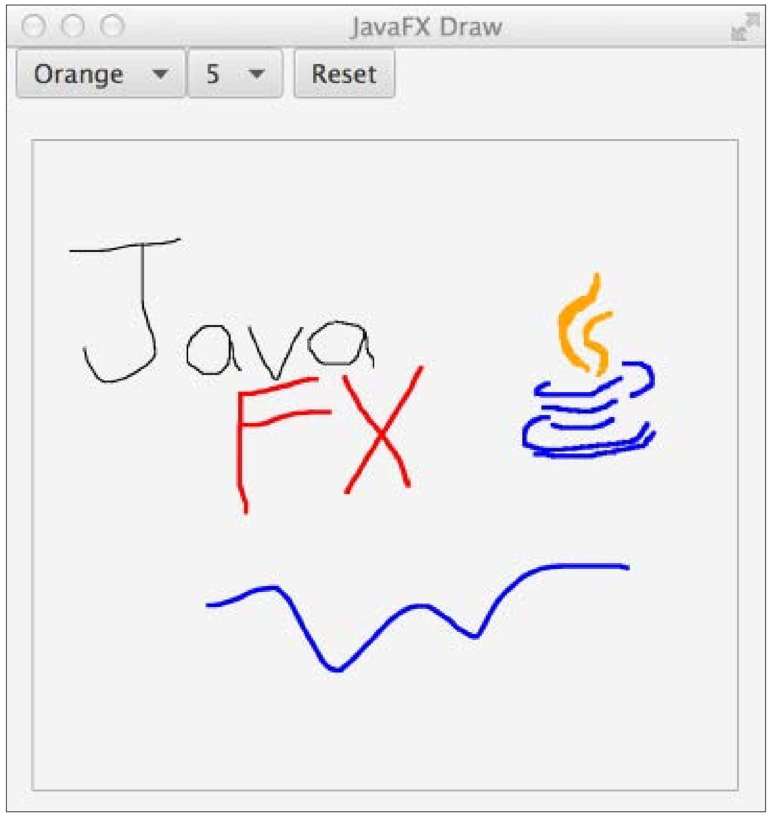


Figure 1

Next, let's look at one of the canvas action listeners, which in JavaFX 8 can also be written as a lambda expression. When the user presses the mouse button while on the canvas, a stroke marks the canvas until the mouse button is released:

```
canvas.addEventHandler(
  MouseEvent.MOUSE_PRESSED,
   (MouseEvent event) -> {
 graphicsContext.beginPath();
 graphicsContext.moveTo(
     event.getX(),
     event.getY());
  graphicsContext.stroke();
});
```

The rest of the code is fairly straightforward. The two ChoiceBox controls are constructed containing observable array lists of values. The Button and ChoiceBox controls are grouped together within an HBox, which is then added to the top of a BorderPane, and the Canvas is added to the center of the BorderPane. Finally, the BorderPane is added to a StackPane to create the stage. Before the stage is fully constructed and made visible, the Canvas is initialized by calling a separate initDraw() method. Now let's take a look at how this same application differs when implemented in Groovy and Jython.

JavaFX 8 and Groovy. In this section, we'll implement the JavaFXDraw application via the Groovy language. If you are new to Groovy, it is a very easy alternative language to start with because it allows you to code using either the standard Java syntax or the Groovy syntax, providing an environment that is easy to begin working in for any experienced Java developer. This example takes advantage of the ability to mix both Java and Groovy syntax together.

Listing 2 contains code for the entire Groovy implementation, which is a few lines shorter than the JavaFX 8 implementation. The code could have used more of the Groovy syntax, but much of the Java syntax was retained to demonstrate the benefits of mixing both Java and Groovy.

For starters, note that the Groovy application still launches within the main method, and much of the code remains the same. However, the Groovy code is a bit more concise because it does not contain semicolons, since they are optional in Groovy. The Groovy code also uses closures—as opposed to Java SE 8 lambdas—which have a slightly different syntax but provide the same overall benefit.

Let's take a look at the **Reset** button in the Groovy implementation, and compare it to the JavaFX 8 API code shown earlier. The **Reset** button code uses a closure to

LISTING 1 LISTING 2

Note: The following listing has been excerpted for space, as noted by the ... symbol. The full code listing is available by downloading the code listings for this issue.

```
public class JavaFXDraw extends Application {

  public static void main(String[] args) {
    Application.launch(JavaFXDraw.class, args);
  }
  @Override
  public void start(Stage primaryStage) {

    StackPane root = new StackPane();
    Screen screen = Screen.getPrimary();
    Rectangle2D rect = screen.getVisualBounds();
    Canvas canvas = new Canvas(rect.getWidth()/2 + 50,
        rect.getHeight()/2 + 300 );
    final GraphicsContext graphicsContext =
        canvas.getGraphicsContext2D();

    final Button resetButton = new Button("Reset");
    resetButton.setOnAction(actionEvent-> {
      graphicsContext.clearRect(1, 1,
      graphicsContext.getCanvas().getWidth()-2,
      graphicsContext.getCanvas().getHeight()-2);
    });
    resetButton.setTranslateX(10);

    // Set up the pen color chooser
    ChoiceBox colorChooser = new ChoiceBox(
      FXCollections.observableArrayList(
    "Black", "Blue", "Red", "Green", "Brown", "Orange"
    ));
...
```

Download all listings in this issue as text

encapsulate the event handling implementation. Similarly, the canvas "mouse button pressed" event listener is implemented using a Groovy closure:

```
canvas.addEventHandler(
  MouseEvent.MOUSE_PRESSED, {
 MouseEvent event->
 graphicsContext.beginPath()
 graphicsContext.moveTo(
    event.getX(),
    event.getY())
 graphicsContext.stroke()
 } as EventHandler)
```

A few more notes about Groovy: Although the code in **Listing 2** contains switch statements that compare int values, it is possible to compare any type of switch value within a Groovy switch statement. Groovy switch statements can also perform different types of matching. Groovy contains many builders, making it easier to construct objects than with standard Java. There are many more features available in Groovy, and to learn more about them, take a look at the *Groovy User Guide*.

It is easy (and fun) to implement JavaFX 8 applications using Groovy. However, later in this article, we'll take a look at an even groovier way to write JavaFX using Groovy.

Note: You should use Groovy 2.3.x to ensure full compatibility with Java SE 8.

JavaFX 8 and Jython. It is said that Python allows developers to get things done without getting in the way. Python code is concise, easy to follow, and very productive. Jython is the Python language implemented for the Java Virtual Machine (JVM). Implementing JavaFX applications in Jython can be quite effective, and it is easy to translate Java to Python syntax.

Note: The code for this example was written with the beta versions of Jython 2.7 and JavaFX 8.

Listing 3 contains the complete code for our drawing application written in Jython. Note that the Jython implementation is even fewer lines than JavaFX 8 or Groovy. For starters, the application is launched a bit differently from Jython, although the same concept as the main method applies. The Jython main method looks a bit different than in the other two languages:

```
if __name__ == "__main__":
  ...
```

In **Listing 3**, the JythonFXDraw class implements Application, per the JavaFX standard. The start method contains the same logic as with the other implementations, but it is written in the Python language syntax.

LISTING 3

Note: The following listing has been excerpted for space, as noted by the ... symbol. The full code listing is available by downloading the code listings for this issue.

```
class JythonFXDraw(Application):

  def start(self, primaryStage):
    primaryStage.setTitle("JythonFX Draw")
    root = StackPane()
    screen = javafx.stage.Screen.getPrimary()
    rect = screen.visualBounds
    canvas = Canvas(rect.width/2 + 50,
                rect.height/2 + 300)
    graphics_context = canvas.graphicsContext2D

    def resetAction(event):
      graphics_context.clearRect(1,1,
      graphics_context.canvas.width-2,
      graphics_context.canvas.height-2)
...
```

Download all listings in this issue as text

Again, let's take a look at the **Reset** button implementation to see how it differs from the others. The event handler is written by assigning the resetAction function to the onAction property. The resetAction function contains the same logic as the others for clearing the canvas:

```
Button("Reset",
    onAction=resetAction)
```

Similarly, the canvas action listeners are implemented as functions, and then assigned to the canvas. The following code shows the "mouse pressed" listener implementation:

```
def mouse_pressed(event):
  graphics_context.
    beginPath()
  graphics_context.
    moveTo(event.x,
        event.y)
  graphics_context.
    stroke()
```

Jython does not contain a switch statement, so the implementation for the ChoiceBox selection uses an if statement instead:

```
if idx == 0:
  new_color = Color.BLACK
elif idx == 1:
  new_color = Color.BLUE
elif idx == 2:
  new_color = Color.RED
...
```

If you are new to Python, you'll see that there is a standard indentation structure to which you must adhere. That said, the code is concise and easy to follow. Another note about Jython is that, just as in Groovy, there is no need to declare types, so function parameters and variables are typeless.

Using Custom APIs to Develop JavaFX Applications

We've already seen that implementing JavaFX applications with alternative languages can have advantages. While there are some advantages to coding with alternative languages, there are even more advantages to using an API that is tailored specifically for use with JavaFX. In this section, we will take a look at the GroovyFX API, which completely changes the way in which a developer can construct a JavaFX application. The developers of GroovyFX have made it easy to visually see the scene graph as it is being built, making code easier to read, understand, and visualize.

GroovyFX at a glance. GroovyFX is an API that allows you to construct a JavaFX application using the Groovy builder syntax. The resulting code is very clean and easy to read, making the construction of JavaFX applications very easy. GroovyFX also eliminates lots of boilerplate code, allowing you to concentrate on what you'd like to get done, rather than on worrying about specific API calls. This is possible because GroovyFX takes advantage of Groovy's powerful domain-specific language (DSL) features and abstract syntax tree (AST) transformation.

OTHER OPTIONS

Although the JavaFX 8 API is very nice, there are advantages to knowing more than one way to develop a JavaFX application.

Let's begin by transforming the application that was written in the first section of this article using Groovy and Jython so it adheres to the GroovyFX API. **Listing 4** contains the code for the GroovyFXDraw application. The code is significantly shorter than any of the other code we've looked at thus far. It is

LISTING 4

Note: The following listing has been excerpted for space, as noted by the ... symbol. The full code listing is available by downloading the code listings for this issue.

```
import static groovyx.javafx.GroovyFX.start
import groovyx.javafx.beans.FXBindable
import javafx.stage.Screen
import javafx.scene.input.MouseEvent;
import javafx.scene.paint.Color
import javafx.collections.FXCollections
import javafx.scene.canvas.Canvas
import javafx.scene.canvas.GraphicsContext
import javafx.stage.Screen

start {
  stage(title: 'GroovyFX Draw', visible: true) {
    scene = scene(id: "sc", fill: WHITE,
      width: Screen.getPrimary()
        .getVisualBounds().getWidth(),
      height: Screen.getPrimary()
       .getVisualBounds().getHeight()) {

      canvas = canvas(id: "drawcanvas",
        width: bind(sc.width()),
        height: bind(sc.height()))
      GraphicsContext graphicsContext =
       canvas.graphicsContext2D
      canvas.onMousePressed {
        MouseEvent event->
        graphicsContext.beginPath()
        graphicsContext.moveTo(event.getX(),
                    event.getY())
        graphicsContext.stroke()
      }
...
```

Download all listings in this issue as text

also easy to read because it uses the Groovy builder pattern.

Another significant difference is the way in which the application is launched, because no main method is defined. Instead, a Groovy closure that contains the application constructs is simply passed to the static groovyx.javafx .GroovyFX.start method to initiate the application:

```
start {
  // Declare JavaFX
  // scene graph nodes
}
```

An additional important difference is the way in which the code is constructed. Each node embedded into the scene graph is constructed as a Groovy closure, and in some cases nodes are embedded within each other. This visual syntax provides an easy way to see how the user interface will be laid out. The properties of each node are also specified differently with GroovyFX, because they are treated as key/value pairs. Each node can have its own set of properties specified by placing them within a comma-separated key/value format, as seen here:

```
hbox(spacing:10, padding:10) {
...
```

Getting started with GroovyFX. There are a couple of easy steps you need to take before you can use GroovyFX. To use GroovyFX in any environment (without the use of an IDE), first, download GroovyFX using the latest distribution from the GroovyFX site. Alternatively, you can build the GroovyFX Project from the source code by cloning the project from GitHub.

To clone the project, ensure that you have Git installed on your system, and then clone the project by executing the following command:

```
git clone git://github.com/
groovyfx-project/groovyfx.git.
```

Note: Please ensure that you are using GroovyFX 0.4.0 for Java 8 and JavaFX 8 compatibility. The examples for this article were written with GroovyFX 0.3.1 and Java SE 7, as GroovyFX 0.4.0 was not available at the time.

Once you've cloned the project, you can build it from source code using Gradle. The current release, 0.4.0, includes support for JavaFX 2.2 on Java 7 or JavaFX 8 on Java 8.

```
git clone -b 0_4_SNAPSHOT git://
github.com/groovyfx-project/
groovyfx.git

cd <<groovyfx_location>>
gradlew build
```

You can then run any of the packaged examples using the gradlew command, followed by the name of the example. For example, to run HelloWorldDemo, use the following command:

```
gradlew HelloWorldDemo
```

If you prefer to use an IDE such as NetBeans, simply clone the GroovyFX project and open it within NetBeans. By default, the GroovyFX project distribution is a NetBeans IDE project. You can run any of the demos within the NetBeans IDE by expanding the **Demo Programs** folder, right-clicking the demo, and choosing **Run**. There are some excellent GroovyFX examples contained in the distribution. If you wish to use an IDE other than NetBeans, you can refer to the GroovyFX Guide for details.

Building a basic GroovyFX application. As mentioned previously, all GroovyFX applications are launched using the static groovyx .javafx.GroovyFX.start method. The scene graph is passed to the start method by enclosing it within brackets. We're able to use the Groovy builder syntax, but behind the scenes, GroovyFX' SceneGraphBuilder is used to construct the scene graph.

Each of the JavaFX controls is available to GroovyFX, but the syntax is a bit different than usual. For instance, each of the controls when used in Java code begins with a capital letter. However, its GroovyFX counterpart must begin with a lowercase letter.

Another difference is that instead of using Java "setters" to configure different control properties, GroovyFX allows you to pass a map (key/value pairs) of different properties to be set. Take, for instance, the JavaFX Button control. In JavaFX, the control must be instantiated and then set to different properties, and setter methods must be called upon the Button instance. In GroovyFX, a Button control is instantiated by placing it within the Scene closure, and different properties are set by enclosing them as key/value pairs within parentheses, as follows:

```
button("Save",
 style:"-fx-font: 14 arial",
  onAction: {
  textEntry =
    text.getText()
    println textEntry
})
```

Note that the event handler is written inline as a Groovy closure, and it is assigned to the onAction property. Styles are set inline as well using the style property. You can also access the javafx.scene.text.Font directly as a node property. For example, the style property from the previous example could be changed to font: '14pt arial', as follows, instead of listing out the full style string:

```
button("Save",
  font: '14pt arial',
  onAction: {
  textEntry =
    text.getText()
    println textEntry
})
```

The code in **Listing 5** includes this Button control, along with a TextField and Label. The basic application allows you to enter some text into the TextField, click the button, and have the text printed to the server log file. It is a very simple application, but you can see that it demonstrates basic layout with GroovyFX, and it uses an HBox. In this example, the HBox contains the label and the TextField, with a spacing of 5.

```
hbox(spacing: 5){
  ...
}
```

In addition, colors and paints can be set using either the JavaFX Paint or Color objects, or via pseudo color variables (RED, ORANGE, GREEN, and so on). Colors can even be defined as a hex string or a JavaFX cascading style sheet (CSS) style, making GroovyFX very flexible. GroovyFX includes a multitude of handy shortcuts; to learn about them, I recommend taking a look at the GroovyFX documentation.

EASY DOES IT

JavaFX 8 makes it easy to develop animations, and GroovyFX makes it even easier!

Abstracting inline code. Additional code can be placed outside the scene graph, and it will still reside within scope for use via scene graph nodes. **Listing 6** demonstrates an example of placing code outside the scene graph in an effort to organize. The code in **Listing 6** is for the GroovyFXDraw application, but the Button control for resetting the Canvas has had its event handler code abstracted and placed within a closure that has been defined outside the scene graph.

This style of coding for GroovyFX applications can come in very handy when working with large scene graphs that contain many events. It also can be handy if you want to make a class accessible within your scene graph, but still place it within the same Groovy file.

LISTING 5 LISTING 6

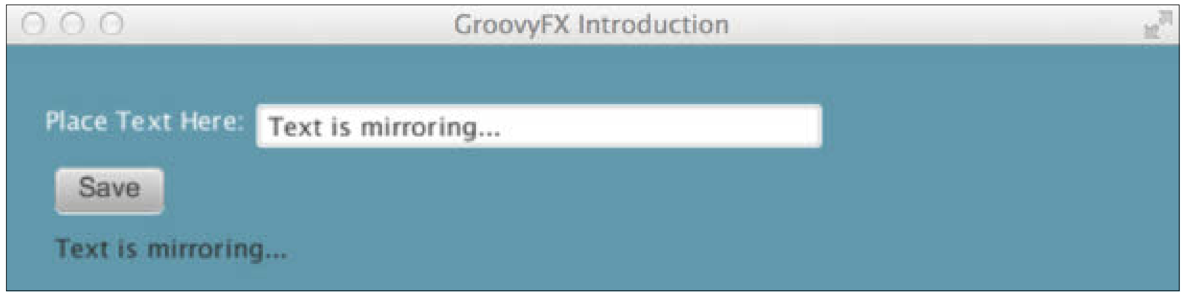
```
import static groovyx.javafx.GroovyFX.start

start{
  def textEntry;
  stage(title: "GroovyFX Introduction", width: 600,
     height: 300, visible: true){
    scene(fill: groovyblue){
      hbox(spacing: 5){
        label("Place Text Here:",
           textFill: white,
           halignment: center,
           valignment: center)
        text = textField(promptText: "Type here",
           prefColumnCount: 25)
      }
        button("Save", style:"-fx-font: 14 arial",
           onAction: {
           textEntry = text.getText()
           println textEntry
        })

    }
  }
}
```

Download all listings in this issue as text

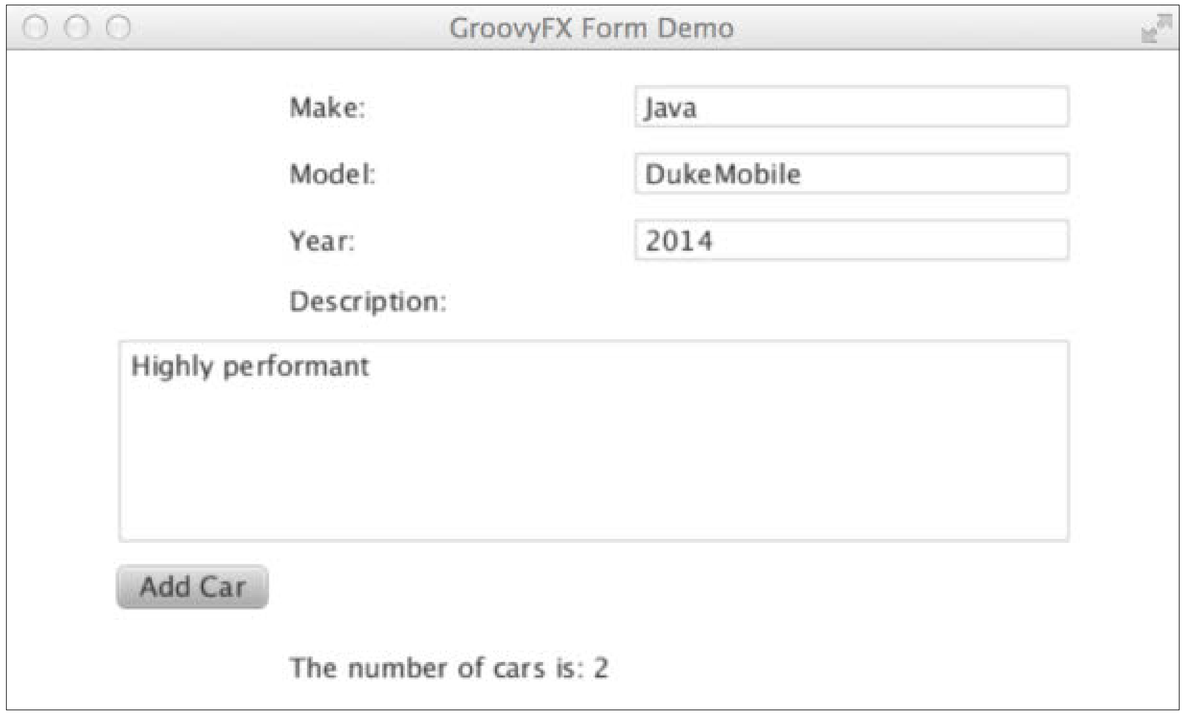
Binding properties with GroovyFX. One of the most powerful features of JavaFX is its binding ability. Binding allows nodes within a scene graph to exchange data and react together in real time. For instance, if the text property of a TextField control is bound to a Label, then the Label text will reflect any changes that are made to the TextField contents as a user is typing. JavaFX binding can contain lots of boilerplate code, and GroovyFX reduces the boilerplate by using its bind utility. The following Label written in GroovyFX demonstrates



Place Text Here: Text is mirroring...

Text is mirroring...

Figure 2



Make: Java

Model: DukeMobile

Year: 2014

Description:

Highly performant

The number of cars is: 2

Figure 3

binding to a TextField named text.

Label(text:bind(text, 'text'))

The code in **Listing 7** demonstrates simple binding using this technique. When the code is executed, the Label will mirror any text typed into the TextField (see **Figure 2**).

GroovyFX also provides a convenience annotation, @FXBindable, to automatically mark a property or entire class as bindable. By applying the @FXBindable annotation to a property or class, the binding logic will be automatically added to the code at runtime, alleviating the need to write redundant boilerplate code.

In the next example, a simple form has been generated for entering car information (see **Figure 3**). The form fields are each bound to

LISTING 7

```
String textEntry
start{

  stage(title: "GroovyFX Introduction", width: 600,
    height: 300, visible: true){
    scene(fill: groovyblue){
      gridPane(hgap: 5, vgap: 10,
          padding: 20){
        hbox(spacing: 5, row:1,
          columnSpan:2 ){
          label("Place Text Here:",
              textFill: white,
              halignment: center,
              valignment: center)
          text = textField(
                promptText: "Type here",
                prefColumnCount: 25)
        }

        button("Save", font: "14pt arial",
            row: 2, column: 1,
          onAction: {
            textEntry = text.getText()
            println textEntry
          })
        label(text: bind(text, 'text'),
             row: 3, column: 1)
      }
    }
  }
}
```

Download all listings in this issue as text

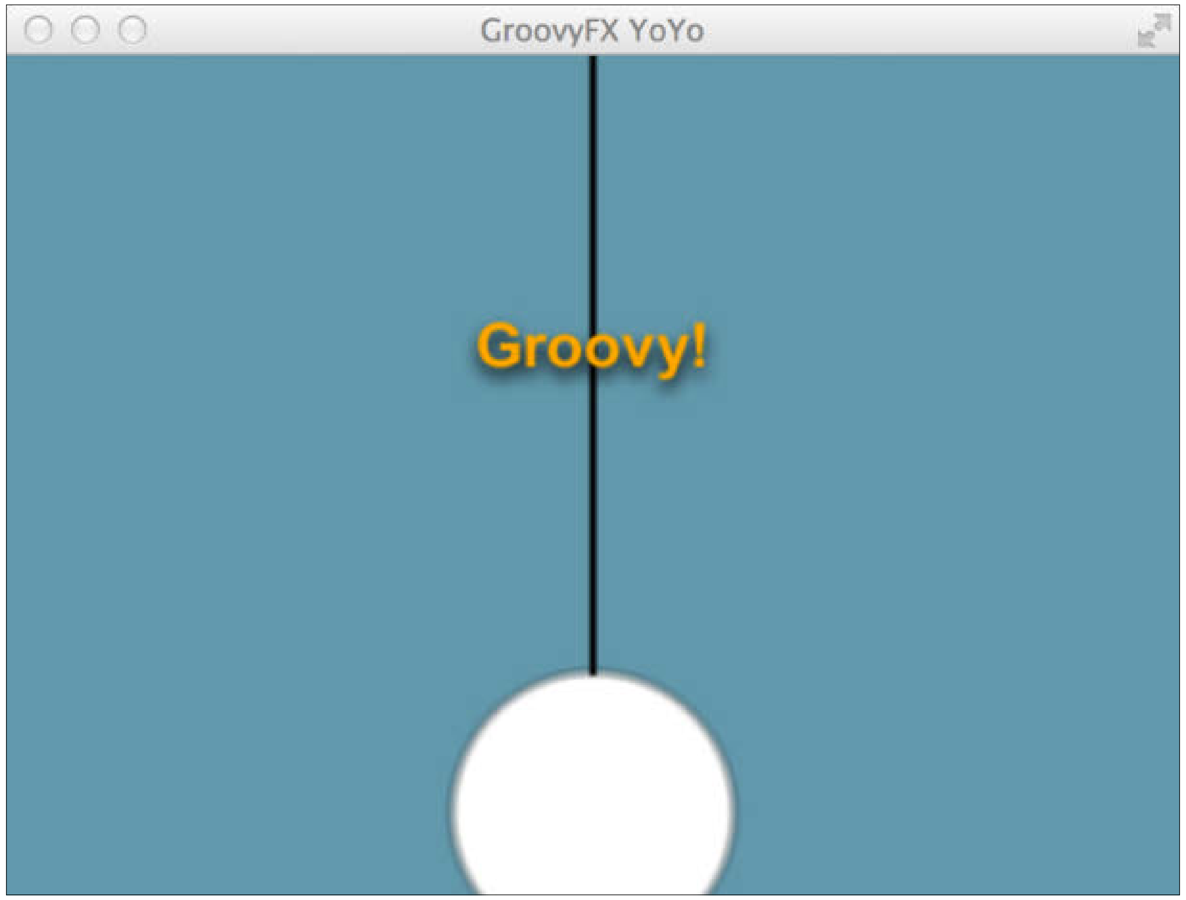


Figure 4

fields within a Car object. The **form** button adds the current Car object to a List of Car objects, and the current List size is automatically incremented and displayed as a Label on the form. Take a look at **Listing 8** to see the code for this example.

Animation with GroovyFX. Now, I am primarily an enterprise developer . . . so animation is not my forte. However, JavaFX 8 makes it easy to develop animations, and GroovyFX makes it even easier! Animations occur on a timeline, and different activities can occur within the span of the timeline.

In this example, the timeline contains two simple change effects that occur indefinitely, beginning at 1,000 milliseconds. This example animates a circle and a line to create a yo-yo effect (see **Figure 4**). Therefore, the circle is changed every 1,000 milliseconds to move from its starting centerY value of 60 to a centerY value of 340. Similarly, the line is changed every 1,000 milliseconds to expand from its starting Y value of 0 to an ending Y value of 280, and then back again. To make the example even more fun, I added some fading text, which

LISTING 8

Note: The following listing has been excerpted for space, as noted by the ... symbol. The full code listing is available by downloading the code listings for this issue.

```
@FXBindable
class Car{
   String make
   String model
   String year
   String description
}

@FXBindable
class CarHotel{

 List<Car> carList = new ArrayList()
 Car car = new Car()
 String carCount

 def addCar(car){
    print "Adding car: ${car.make}"
    carList.add(car)
    carCount = "The number of cars is: ${carList.size()}"
    car = new Car()
  }

}
...
```

Download all listings in this issue as text

fades from 0 opacity to 1 over a specified time span.

To see the code for the animation example, see **Listing 9**.

FXML support. Placing view code into FXML files is a recommended approach for separating view code from business logic for JavaFX applications. GroovyFX supports the use of FXML by referencing the files via an fxml node within the scene graph:

```
scene {
fxml(new File('./myfxml.fxml')
 .text)
...
```

TAILOR-MADE

Implementing JavaFX applications with alternative languages can have advantages. There are even more advantages to using an API that is **tailored specifically for use with JavaFX.**

Other Alternative APIs

Other alternative languages, such as JRuby and Scala, can be used to develop JavaFX applications as well. In fact, these two languages, in particular, have their own specialized APIs for JavaFX development: JRubyFX and ScalaFX, respectively. These APIs are similar to GroovyFX in that they provide an easy way to work with the JavaFX API in the alternative language syntax. For more information, check out their project websites.

Conclusion

JavaFX 8 contains a bevy of new features, and the ability to make use of Java SE 8 constructs, such as lambdas and streams, makes JavaFX development even easier. Alternative languages can provide benefits over—and in some cases work hand in hand with—the JavaFX API. In some cases, alternative languages have their own DSLs for developing JavaFX, which can be even more productive and easy to use.

Add a new tool to your JavaFX toolbox today by learning an alternative approach to developing JavaFX applications. </article>

MORE ON TOPIC:

Java Virtual Machine

LEARN MORE

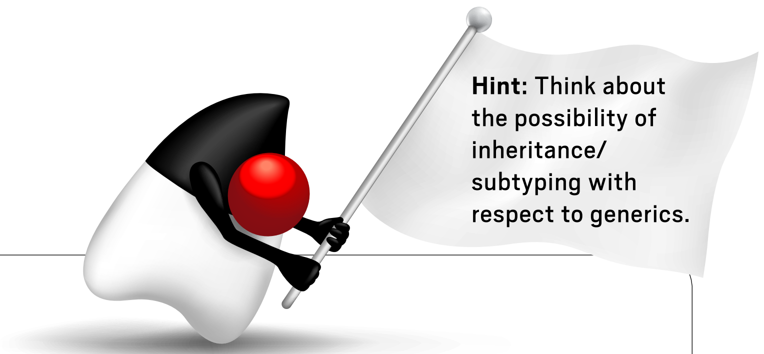
- "Introducing GroovyFX: It's About Time"
- Jython website

LISTING 9

Note: The following listing has been excerpted for space, as noted by the ... symbol. The full code listing is available by downloading the code listings for this issue.

```
import static groovyx.javafx.GroovyFX.start
start {
  stage(title: "GroovyFX YoYo", width: 500,
    height: 400, visible: true) {
    scene(fill: GROOVYBLUE) {
      circle = circle(centerX: 250,
        centerY: 60,
        radius:60,
        fill: WHITE,
        stroke: BLACK){
        effect: boxBlur(10, 10, 3)
      }
      line   = line (startX: 250,
        endX: 250,
        startY: 0,
        endY: 0,
        strokeWidth: 3)
      myTxt = text("Groovy!",
        x:  200, y: 150, fill: ORANGE,
        font: "bold 26pt Arial"){
         fade = fadeTransition(
               duration: 5000.ms,
               fromValue: 0.0,
               toValue: 1.0)
         effect dropShadow(offsetY: 4)
        }
    }
  }
...
```

Download all listings in this issue as text


Hint: Think about the possibility of inheritance/subtyping with respect to generics.

In the May/June 2014 issue, Simon Ritter, Java evangelist at Oracle, presented a Java SE 8 Stream API challenge. He gave us code that uses streams to determine the length of the longest line in a text file and asked how we could convert it to return the actual line, not its length. The correct answer is #2. We need to perform a reduction on the stream to generate a single result—in this case, the longest string in the file. The reduce() method is a form of a functional fold, which will apply a function to each element in a stream. Answer #2 applies a function that compares the length of two strings and returns the longest.

Answer #1 will not work because there is no longest() method in String to use as a method reference. Answer #3 will not compile. The reduce() method has a form that takes two parameters, but the parameter and return types of the lambda expression must all be of type String. Answers #4 and #5 will both work, but return the shortest line of the file because the subtraction of lengths is reversed.

This issue's challenge comes from Abhishek Gupta, a senior identity and access engineer, who gives us a generics problem.

1 THE PROBLEM

In spite of the introduction of generics in Java SE 5, raw types are supported in order to provide backward compatibility for code written using JDK 1.4 and below. But what happens when we mix generic and raw types?

2 THE CODE

Assume that this code is compiled and executed with JDK 1.5 or above, which has support for generics.

```
import java.util.ArrayList;
import java.util.List;
class Test{
private static void add(Object obj, List target){
target.add(obj);
}
public static void main(String[] args) throws Exception {
List<String> strings = new ArrayList<String>();
add(1, strings);
System.out.println("First element in the list:: "+ strings
   .get(0));
}
}
```

3 WHAT'S THE FIX?

1) The class compilation fails because of an attempt to add a primitive int type to a List.
2) The compilation fails and is resolved by changing the signature of the add method to use a generic type (List<Object>) instead of a raw List.
3) Compilation is successful but there is a java.lang.ClassCastException at runtime. It is fixed by changing the signature of the add method to use generics and invoking the add method using the appropriate argument type.
4) Compilation is successful but there is a java.lang.ClassCastException at runtime. It is fixed by changing the signature of the add method to use a generic type (List<Object>) instead of a raw List.

ART BY I-HUA CHEN

GOT THE ANSWER?
Look for the answer in the next issue. Or submit your own code challenge!