Java™ magazine

By and for the Java community

SEPTEMBER/OCTOBER 2015

Testing

HUNTING DOWN HARD-TO-FIND ERRORS

ORACLE.COM/JAVAMAGAZINE

ORACLE®

//table of contents /

COVER ART BY I-HUA CHEN

EDITORIAL

Editor in Chief
Andrew Binstock

Managing Editor
Claire Breen

Copy Editor
Karen Perkins

Section Development
Michelle Kovac

Technical Reviewers
Stephen Chin, Reza Rahman, Simon Ritter

DESIGN

Senior Creative Director
Francisco G Delgadillo

Design Director
Richard Merchán

Senior Designer
Arianna Pucherelli

Designer
Jaime Ferrand

Senior Production Manager
Sheila Brennan

Production Designer
Kathy Cygnarowicz

PUBLISHING

Publisher
Jennifer Hamilton +1.650.506.3794

Associate Publisher and Audience Development Director
Karin Kinnear +1.650.506.1985

Audience Development Manager
Jennifer Kurtz

ADVERTISING SALES

Josie Damian +1.626.396.9400 x 200

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)

ARTICLE SUBMISSION

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

SUBSCRIPTION INFORMATION

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

MAGAZINE CUSTOMER SERVICE

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

PRIVACY

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

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

Digital Publishing by GTxcel

BIO

The Unreasonable Effectiveness of Static Analysis

It's easy to run, provably effective, and greatly underused. Why?

There is perhaps no greater peculiarity in the profession of software development than the lack of interest in software engineering. This field, which is the empirical, quantitative analysis of software-development techniques, should be the cornerstone of the trade. And yet for most development organizations, it remains a closed domain into which they never venture. Even the emerging emphasis on metrics and dashboards has not led to curiosity about what thousands of projects tell us about those very numbers. The disregard extends even to the parlance we use: In most locales, a "software engineer" is a seasoned programmer. There is no implication of familiarity with software engineering.

Part of this neglect is the view that practitioners are mostly academics. Although those same academics study real-life projects, both open source and inside companies, the perception that they are disconnected from reality endures. Their utility, however, shows up when a project with unusual requirements confronts an organization. Suppose after several years of leading various teams at your company and delivering projects roughly on time and of the desired quality, you're charged with a green-field project that requires a much lower level of defects than your organization is accustomed to. How will you amp up the quality?

This is where software engineering becomes a crucial resource: You can see which techniques deliver lower defect rates and what their impact on productivity has been historically. Informed with this data, you'll be able to plan how to go about reaching the new goals for this project. (In

PHOTOGRAPH BY BOB ADLER/GETTY IMAGES

real life, of course, most organizations just call in consultants.)

Were you to perform this inquiry, it might lead to consideration of what practices you could add to existing projects to raise their overall quality without sacrificing productivity. In many—likely in most—cases, that additional practice would be static analysis of code. Relying on the work of Capers Jones, one of the most widely experienced analysts of software-engineering data, static analysis reduces defects in projects by 44 percent (from an average of 5.0 defects per unit of functionality to 2.8 defects). Among standard quality assurance techniques, only formal inspections have a higher effectiveness (60 percent). By comparison, test-driven development (TDD) comes in at 37 percent. These numbers are not additive, of course. Inspections plus static analysis don't deliver 100 percent defect-free code. By the same token, static analysis reveals problems that inspections and TDD cannot find. Let me explain.

Static-analysis tools today fall into roughly two tiers. There are the open source tools, which in Java are particularly good. These include FindBugs, PMD, walkmod, and to a lesser extent Checkstyle. Commercial tools—such as those from Coverity, Parasoft, Rogue Wave Software, and other vendors—are significantly more advanced. They can do magical things such as data-motion analysis, which reveals defects that are hard to see via other means. For example, this analysis can show that a given data item can never be null because all paths that touch it first pass through another module far afield that guarantees nonnullness. Or it can find the opposite, that a method that counts on being passed only prescreened objects can in fact be passed a null. Some of these tools excel at finding other very difficult bugs, such as unwanted interactions between two threads, or the rare case that a data item can be read incorrectly due to the design of the Java memory model. These are subtle items that can be hard to locate and costly to fix—and they're typically not revealed by formal inspections or unit testing.

The resistance to static analysis, I believe, stems from a reputation built up in its early days of delivering false positives—that is, claiming to find a defect where in fact none exists. Per Jones, false positives a decade ago averaged 10 percent but now are well below 3 percent. In commercial products, the rate is even lower. The other perceived drawback is that the tools are slow. This remains true. You typically run the high-end tools once a day for the whole project, but always on code about to be checked in. The tools can analyze new code by cross-checking with the database of artifacts from the most recent whole-project scan.

Static analysis has an additional upside: It's not disruptive to existing site practices. You can add it to a testing pipeline with ease. This aspect and its remarkable effectiveness make it one of the simplest, but undeservedly underused, ways to improve quality.

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

PS: Update on our evolution: In this issue, you'll see that we updated our fonts to improve legibility and we've begun what will be a long-running series of articles on JVM languages. Let me know if you have any suggestions.

Adobe ColdFusion Celebrates 20 Years of Making Complex Coding Tasks Easy

Originally developed by Jeremy Allaire and JJ Allaire in 1995 to make it easier for developers to connect simple HTML pages to a database, ColdFusion has been around almost as long as the web itself and predates many popular web development languages. A full scripting language—CFML—and an integrated development environment were added, and ColdFusion remains popular with coders for being a rapid web application development platform that helps them reduce development time by an order of magnitude.

Tridib Roy Chowdhury, General Manager and Senior Director of Products, Adobe

A history of innovation

Over two decades ColdFusion has been at the forefront of simplifying all tasks programmers find tedious and time consuming, such as charts and graphs, PDF generation and manipulation, WebSockets and many more. Many of the things that seem so obvious to us now were innovative solutions first offered by ColdFusion, allowing developers to create robust, scalable, high-performing, secure web- and mobile-based applications with minimal effort.

Each release has built on the previous ones, and ColdFusion has long since gone beyond its "tag-based scripting language" past. Today, ColdFusion can talk to pretty much everything, from legacy COM and CORBA to .NET assemblies and Java classes. And it can be used to develop pretty much every application required to meet dynamic business needs—which is why it is so prevalent in large enterprises and government setups.

ColdFusion has a fan following in smaller organizations, too. "When we started out, as [with] any startup, the budget was always short. So we needed a language that allowed us to develop the application in a short period of time. ColdFusion dramatically cut down the development cost," says Sumit Verma, co-owner at Ten24 Digital Solutions.

"ColdFusion has clearly stood the test of time," says Rakshith Naresh, product manager at ColdFusion. "One of the reasons why customers keep coming back to ColdFusion is the productivity benefit it offers."

Continuing to blaze the trail

"There really isn't a comparison at all between how Adobe reacts to our needs versus any of the other software companies that we work with. Basically, what happens is ColdFusion allows us to change our market," quotes Eric Kratz, president at VSR Systems.

Be it overhauling your company's HR operations, updating your firm's global intranet, or powering the world's busiest electronic storefronts, you can be sure this 20-year-old will rise to the occasion.

"ColdFusion continues to do well while staying at the forefront of technology. The unique integration with HTML5 along with the end-to-end mobile development platform and the increased focus on security in ColdFusion give enterprises the confidence and edge to easily adopt the latest technologies as they make their digital transitions. The future roadmap for improvement in the mobile development platform and the addition of social analytics while staying true to the ColdFusion credo of 'making hard things easy' indicate exciting times for ColdFusion in the coming years."

Tridib Roy Chowdhury, General Manager and Senior Director of Products, Adobe

Adobe ColdFusion Summit 2015
(November 9–10, Las Vegas, Nevada) brings together the web application community. Interact with ColdFusion experts, domain leaders, and peers, and learn about the latest technologies, techniques, and strategies to help you rapidly build and successfully deliver web applications to market. Explore how ColdFusion is driving change and how you can propel this momentum.

Adobe ColdFusion Summit is the largest summit for ColdFusion developers and has had three successful annual installments.

For more information, visit **adobe.com/coldfusion**

MAY/JUNE 2015

Casting for long?

In the article "What's New in JPA" in the May/June 2015 issue, I saw in Listing 17 that the author uses a cast to long. But in fact, if the code were

```
return em.createQuery(
      ..., Long.class)
      .getSingleResult();
```

it would remove the need for the cast.

—Josh Toepfer

Author Josh Juneau responds: "You are correct that we can get rid of the cast by passing Long.class as the second argument to createQuery() on line 6 of the listing. Thanks for pointing this out."

IoT Coverage

Now that we're in an IoT-based world, I wish to see articles covering the ecosystem of embedded Java for IoT. This includes the processor, programming, development kits, embedded Java, and so on.

—Sam Desd

Editor Andrew Binstock responds: "We had a long article on using Java for IoT on the Raspberry Pi in the May/June issue. We will have many more similar articles in the future."

Inside the CPU

Thank you so much for driving the magazine in a more technical direction. The article "Inside the CPU: The Unexpected Effects of Instruction Execution" (in the March/April and July/August issues) is absolutely awesome!

—Yury Pitsichin

A Paper Version of *Java Magazine*?

I was wondering if there is a possibility of getting a printed edition of the magazine, at least through a third-party provider? After a complete day of work in front of the computer, it's good to have a print magazine for a change of environment.

—Raj Thondepu

Publisher Jennifer Hamilton responds: "I totally understand wanting to disconnect and enjoy a printed publication. However, Java Magazine *is being published only in digital format. When I want to read a hard-copy version of an article, I download the PDF and print the pages in landscape mode using the Fit option in Adobe Acrobat Reader."*

How About a Kindle Version?

The magazine needs to be available in the Kindle format. I don't get why people would download a PDF, or read the same on their phones. It's too cumbersome.

Having a Kindle version will get you 10x the readers.

—Jude Pereira

Editor Andrew Binstock responds: "In Silicon Valley, at least, most people in tech read magazines on tablets (which, in my opinion, are better for reading articles with code). For users of tablets, we offer both iOS and Android apps. For all others, we offer PDF. We are actively looking at expanding our list of distribution targets, and the Kindle is at the top of that list. However, we don't expect to be able to provide that upgrade before the end of the year or early next year."

Contact Us

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

//events /


San Francisco, California

JavaOne 2015 *OCTOBER 25–29*
SAN FRANCISCO, CALIFORNIA
Join the single largest gathering of Java developers. From sessions, workshops, labs, and exhibits to keynotes and Birds-of-a-Feather sessions, learn about the latest language changes to improve coding efficiency. You'll also learn how to build modern enterprise and server-based applications, create rich and immersive client-side solutions, build next-generation apps targeting smart devices, and compose sophisticated Java web services and cloud solutions.

JavaDay Kharkiv
OCTOBER 1
KHARKIV, UKRAINE
Enjoy and learn in a full day of world-class talks. Topics include core JVM platform and Java SE (Java 8), JVM languages and new programming paradigms, web development, and Java enterprise technologies.

Silicon Valley Code Camp
OCTOBER 3–4
SAN JOSE, CALIFORNIA
Last year, 4,500 people attended this free community event where developers learn from fellow developers. In addition to technical topics, speakers present on software branding and legal issues.

JAX London
OCTOBER 12–14
LONDON, ENGLAND
JAX London brings Java, JVM, and enterprise professionals together for a technology- and methodology-packed event. Participants get full access to Big Data Con London, which features modern datastores, big data architectures based on Hadoop, and advanced data processing techniques.

JDD
OCTOBER 13–14
KRAKOW, POLAND
JDD is a two-day conference for all Java enthusiasts, who can participate in more than 30 lectures, workshops, interactive trainings, and networking opportunities. JDD attracts speakers from all over the world and offers lectures in English.

GeeCON
OCTOBER 23–24
PRAGUE, CZECH REPUBLIC
Join more than 2,000 participants at GeeCON, which is focused on JVM-based technologies with special attention to dynamic languages such as Groovy and Ruby. GeeCON participants share experiences about development methodologies and craftsmanship, enterprise architectures, design patterns, distributed computing, and more.

W-JAX 15
NOVEMBER 2–6
MUNICH, GERMANY
The W-JAX conference covers current and future aspects of technologies such as Java, Scala, and Android. Also addressed are web applications techniques, agile development models, and DevOps.

J-Fall 2015
NOVEMBER 5
EDE, NETHERLANDS
The annual Java conference organized by the Dutch Java User Group (NLJUG) typically sells out and has outgrown its usual venue. This year, J-Fall

PHOTOGRAPH BY WESTEND61/GETTY IMAGES

will take place in the CineMec in Ede.

Devoxx Belgium

NOVEMBER 9–13
ANTWERP, BELGIUM
By developers for developers, this event has 200 speakers and 3,500 attendees from 40 countries. Tracks this year include Java SE, JVM languages, and server-side Java, as well as cloud and big data, mobile, and architecture and security, among others.

Devoxx Morocco

NOVEMBER 16–18
CASABLANCA, MOROCCO
Formerly the JMaghreb conference, this event is a university day of training, workshops, and labs followed by conference days of sessions on software development, web, mobile, gaming, security, methodology, Internet of Things, and cloud. The Decision Makers evening includes discussion of issues related to the IT industry in Morocco.

QCon San Francisco 2015

NOVEMBER 16–20
SAN FRANCISCO, CALIFORNIA
A practitioner-driven software development conference, QCon is designed for technical team leads, architects, engineering directors, and project managers who influence innovation in their teams. Tracks this year include Taking Java to the Next Level and The Dark Side of Security. The last two days are devoted to workshops.

Codemotion Milan

NOVEMBER 18–21
MILAN, ITALY
This conference is open to users of all languages and platforms. It offers full-day workshops on the first two days, followed by keynotes and conference sessions.

Codemotion Spain

NOVEMBER 27–28
MADRID, SPAIN
This two-day event draws nearly 2,000 attendees, represents more than 30 communities, and features coding lectures and workshops. Activities for startups, recruiting, and networking are included.

Clojure eXchange 2015

DECEMBER 3–4
LONDON, ENGLAND
Meet with the world's leading experts, learn how to use Clojure with your team, and discuss war stories with your peers. Both days will feature a mixture of talks covering various aspects of Clojure development: from libraries to music, from ClojureScript to data.

Groovy and Grails eXchange 2015

DECEMBER 14–15
LONDON, ENGLAND
Stay ahead of the curve and hear the 2016 roadmap for Groovy and Grails from core committers and Groovy authorities Guillaume Laforge and Graeme Rocher. Engage with other leading experts and fellow enthusiasts and learn the latest innovations and practices.

Apache Hadoop Innovation Summit

FEBRUARY 11–12
SAN DIEGO, CALIFORNIA
With presentations from more than 25 hands-on industry speakers, topics covered will include MapReduce and Spark, building privacy-protected data systems, scalable data curation, best practices, and architectural considerations for Hadoop applications.

Embedded World 2016

FEBRUARY 23–25
NUREMBERG, GERMANY
The 14th annual gathering of embedded system developers will explore the latest developments, define trends, and once again present the key areas of focus for future developments. This is where hardware, software, and system development engineers come together to turn the next milestones of the Internet of Things into reality.

Have an upcoming conference you'd like to add to our listing? Send us a link and a description of your event at least four months in advance at javamag_us@oracle.com. We'll include as many as space permits.

PHOTOGRAPH BY HENRYK SADURA/GETTY IMAGES

AOT Compilation Is Coming to Java 8

Excelsior JET 11 will support Java SE 8 and JavaFX 8 on all desktop platforms.

Get Your Early Access Copy Now

Registration-Free Download

EXCELSIOR

Java™ LICENSEE

//java books /

JAVA EE 7: THE BIG PICTURE

By Danny Coward
Oracle Press (McGraw-Hill)

For Java SE developers, Java EE has long appeared to be a large tangle of technologies, primarily of interest to enterprise developers. Even though the Enterprise Edition has vastly improved since the days of J2EE, it continues to be trailed by its early reputation. In part, this is because there is a sense that any development on Java EE requires familiarity with many different services that need to be integrated just-so to create a proper app. These skills are in addition to the language skills, which are assumed. For many developers, this seeming complexity has been solved by just using a standard container (Tomcat, for instance) and JDBC on the back end for database needs. Such an approach works well enough until the requirements expand, at which point continuing to organically grow the app with other unintegrated technologies becomes a path of declining returns.

Using Java EE from the start can facilitate the addition of features, scalability, and even security for many projects. The trouble is how to go from Java SE to Java EE without getting overwhelmed by the seeming labyrinth of technologies? That question is precisely what this book addresses. It provides an overview of Java EE 7, systematically explaining the core technologies—how they work and how they fit together.

The author, a principal architect of Java EE, presents the material in the context of a hypothetical application that would use the core technologies: a web front end, a logic layer, and a persistence layer. He then amplifies this content by bringing in additional technologies, including WebSocket, dependency injection, security, and so forth.

The explanations start at a high level but quickly descend to code. This code is both clear and extensive. This book aims squarely at developers, rather than architects or executives, and the programmer reader will find a clear, very well written text.

My only reservation about the book is that it does not cover secondary Java EE technologies, such as Java Message Service or JavaMail. But these consider-

When we began doing critical book reviews in the May/June issue, *we stated that due to possible conflicts of interest, we would not review books from Oracle Press. However, after securing approval for independent reviews from both Oracle and McGraw-Hill (the publishers of Oracle Press books), we begin in this issue to examine two recent titles from that imprint: one we liked a lot and one we did not. —Ed.*

ations aside, this is an excellent programming introduction to Java EE 7. —*Andrew Binstock*

JAVAFX RICH CLIENT PROGRAMMING ON THE NETBEANS PLATFORM
Gail and Paul Anderson
Addison-Wesley

Of all the words in this book's title, *platform* most indicates its true content. This book is not a tutorial on JavaFX using NetBeans. Rather, it's a book about using JavaFX in conjunction with NetBeans as a software platform for developing desktop applications. In this sense, NetBeans as platform corresponds roughly to the Eclipse Rich Client Platform.

That is, it provides the software platform on which rich desktop apps can be built. NetBeans provides its own internal module system (not Eclipse's OSGI) and exposes an extensive UI metaphor—which includes a windows manager with multiwindow support, search capabilities, a wizard builder, and so forth.

Under this presentation goodness is a lot of technology that delivers useful services: for example, an entire file-management layer, which includes goodies such as a file-status monitor that issues a message whenever a watched set of files is modified; RESTful web services; application update services; and so on.

The explanations of the best ways to exploit the NetBeans platform are clear, approachable, and illustrated with useful examples. The integration with JavaFX, however, feels somewhat rougher. Even though JavaFX is presented from scratch (presumably aimed at someone new to the technology), a beginner would be unlikely to be capable of following along without getting lost. The ideal reader is proficient with Java and has familiarity with writing UIs either in Swing or JavaFX. For such a reader, the JavaFX chapters will serve as a review and an update that enables safe passage to the discussion of NetBeans features. Fortunately, almost anyone undertaking a project using the NetBeans platform will have that kind of background. For those readers, this book is a godsend.

The question I'm led to ask is, how many such readers could there be? I expect it's a small set, which makes me admire both the authors and the publisher for releasing a nearly 1,000-page volume on so narrow a topic. But for anyone in that group, this volume is *the* guide to have. —*AB*

INTRODUCING JAVAFX 8 PROGRAMMING
By Herbert Schildt
Oracle Press (McGraw-Hill)

As fun as JavaFX programming is, it is a world unto itself that requires a good guidebook to understand its ins and outs. But instead, this book is a most incomplete introduction to the technology.

First, let me touch on what it doesn't cover, because these are crucial gaps: FXML and CSS styling. These two technologies, essential elements of any serious introduction, are dismissed via this glib line: "[All the examples in this book] are in Java. Therefore, no understanding of CSS or FXML is needed." The use of FXML and CSS precisely to avoid coding details in Java is apparently lost on the author.

Many other items are not covered. The author mentions JavaFX technologies that he tells readers to learn by themselves. However, if you turn to the section entitled "For Further Study" to do this, you find only a list of books all written by this same author *not one* of which is about JavaFX.

As to the actual content, the author is unhelpful. He covers the easy things in great detail, and the hard things are glossed over. There is precious little here that cannot be found in freely available tutorials. For serious developers, Hendrik Ebbers' *Mastering JavaFX 8 Controls* provides a far better introduction to JavaFX; it covers both CSS and FXML and does not skirt difficult material. —*AB*

Testing:

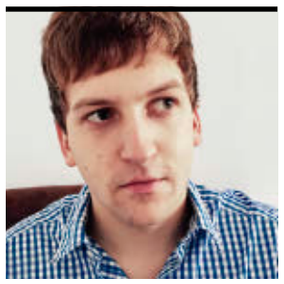
WE'RE ALL IN THIS TOGETHER

Between the pre- and post-agile generations of developers, there is perhaps no greater difference than the role of developers in testing their code. The concept of developers writing tests as they pushed out code was a radical idea that took root and became an essential part of the coding process. Today some companies, such as Atlassian (page 42), are radicalizing this notion even further by moving *all* the responsibilities for QA to developers and training newly hired programmers from their first day in QA principles and techniques.

While agile values set the course in this new direction, it was undoubtedly the advent of JUnit—a fast test framework with intuitive mechanics—that made developer testing a universal reality. Despite JUnit's ubiquity, most of us, I fear, use only a familiar subset of its features and re-create capabilities already available. Our article on useful but underused features of JUnit (page 20) should help.

User interfaces are not as amenable to JUnit and so require specialized tools. For JavaFX, that tool is increasingly TestFX (page 14). For web-based interfaces, unfortunately, it is clusters of virtual machines exercising hundreds of combinations of browser releases and operating systems (page 26). Whatever your project's test tools, it is clear that we will not soon return to the days when coding and testing were segregated activities performed by different teams. I'm good with that. —*Andrew Binstock*

ART BY I-HUA CHEN



BENNET SCHULZ

BIO

Test JavaFX Apps with TestFX

Simple JUnit-style testing of JavaFX UIs

TestFX is an API for testing user interfaces written in JavaFX. It automates tests for JavaFX applications by simulating user interactions such as button clicks, typing text into a field, and many other interactions done in JavaFX applications.

This article starts with a short background on the TestFX framework, so you can get an understanding of what it is based on and what its goals are. After that, the article shows how to get started with TestFX in Maven projects and how to write tests for a sample application. I also discuss limitations of TestFX that you should know about before using it.

Version 4.0.x of TestFX is currently in alpha state. Therefore, this article covers the latest stable version, 3.1.2. I presume that you've used JavaFX and have a good understanding of how it works, including having a familiarity with FXML.

Background

TestFX is based on the well-known unit testing framework JUnit. Like JUnit, it is very simple to learn and easy to use. In TestFX, tests can be written in a similar way to JUnit tests. There are just a few complements developers need to add. For example, TestFX uses Hamcrest matchers on top of JUnit for test assertions.

The benefit of Hamcrest matchers when compared with standard JUnit assertions is that you can use natural assertions and get more-helpful error messages when an assertion fails. This increases the code quality of tests, and it also reduces the complexity of assertions by offering additional options.

Getting Started with Maven

Adding TestFX to a Maven build is as simple as adding the JUnit dependency. You need to add only the following snippet to your pom.xml file:

```
<dependency>
    <groupId>org.loadui</groupId>
    <artifactId>testFx</artifactId>
    <version>3.1.2</version>
</dependency>
```

Initializing a Sample Test Class

If you want to write a TestFX test, your test class needs to extend `org.loadui.testfx.GuiTest`. After that, you have to override the `getRootNode` method. This method has to return the view that is to be tested. In the following snippet, the method returns an FXML view, but it is also possible to return a programmatically created Swing-style class as the view.

```
public class SampleTest extends GuiTest {

    @Override
    protected Parent getRootNode() {
        Parent parent = null;
        try {
            parent = FXMLLoader.load(getClass()
                .getResource("sample.fxml"));
            return parent;
        } catch (IOException ex) {
            // ...
```

```
        }
        return parent;
    }
    // ...
}
```

By extending `GuiTest`, you also get additional UI test methods that simulate user interactions. For example, you have access to methods for finding UI elements, such as buttons or fields, and methods for clicking buttons, scrolling scrollbars, and so forth.

The Sample Application

The sample application is a calculator based on the GNOME gcalctool (see **Figure 1**). It looks like gcalctool and behaves like gcalctool, but it is written in JavaFX (instead of gcalctool's original language, Vala).

Figure 1. The view of the sample application

In contrast to other implementations, this calculator shows all user input in a field, and when the equals (=) button is clicked, the result is calculated. Other calculators show only the last numeric input and after the = button is clicked, they show the result without showing intermediate steps.

The view of this application is an FXML file created with Gluon Scene Builder 8.0.0, and the buttons are styled with CSS using a linear gradient. The more interesting part is the corresponding controller. It contains three methods: `handleButtonAction`, `handleRemoveButtonAction`, and `handleCalculationAction`. The first two methods are simple. The last one is the method that calculates the result after the = button is clicked. I'll focus on testing it.

Listing 1 shows the method for calculating the result depending on the user's input.

Listing 1.

```
@FXML
private void
  handleCalculationAction(ActionEvent e) {
    String displayText = display.getText();
    int textLength = displayText.length();
    String result = "";
    if (displayText.contains("+")) {
        int plusIndex = displayText.indexOf("+");
        Double a = Double.valueOf(
            displayText.substring(0, plusIndex));
        Double b = Double.valueOf(
            displayText.substring(plusIndex + 1,
            textLength));
        result = String.valueOf((a + b));
    } else if (displayText.contains("x")) {
        int multiplyIndex =
            displayText.indexOf("x");
        Double a = Double.valueOf(
            displayText.substring(
                0, multiplyIndex));
        Double b = Double.valueOf(
```

```
            displayText.substring(
                multiplyIndex + 1, textLength));
        result = String.valueOf((a * b));
        display.setText(result);
    }
```

This method multiplies, divides, adds, and so on. At the end of this method, the calculated result will be set as the text for the field that is shown at the top of the UI. In the next section, this method is used to introduce how to write tests for this application using TestFX.

Some features (such as subtraction, square, curly brackets, floating points, and so on) were left out for simplicity's sake in this tutorial. If you want to have a version of this calculator that has all its features, it is available in the download area for this issue of *Java Magazine*.

Writing Tests for the Sample Application

Let's write a test that simulates a user who wants to calculate the sum of 1 + 2. The user does this by clicking 1, +, and 2. After the user clicks =, the calculator should display the result of 3 in the field at the top of the UI.

To test this scenario with automated tests, first we add the Maven dependency, as described earlier, and initialize the test class (see **Listing 2**).

Listing 2.

```
public class CalculatorControllerTest
             extends GuiTest {

    @Override
    protected Parent getRootNode() {
        Parent p = null;
        try {
            p = FXMLLoader.load(getClass()
                .getResource("gcalctoolFX.fxml"));
        } catch (IOException ex) {
            Logger.getLogger(
                CalculatorControllerTest
                .class.getName())
                .log(Level.SEVERE, null, ex);
        }
        return parent;
    }
```

The next step is about initializing the gcalctoolFX.fxml view (see **Listing 2**) that we want to test with this test class. This can be done by loading the respective view and returning it as the result of the `getRootNode` method of the test class.

After that initialization step, we can start writing tests for this test scenario. A TestFX test must follow standard JUnit 4 conventions, including using the `@Test` annotation, a public access modifier, and a return type of `void`. A sample TestFX test is shown in **Listing 3**.

Listing 3.

```
@Test
    public void testAddition() {
        Button one = find("#one");
        Button plus = find("#plus");
        Button two = find("#two");
        Button equalSign = find("#equal");

        click(one);
        click(plus);
        click(two);

        verifyThat("#display", hasText("1+2"));
        click(equalSign);
        verifyThat("#display", hasText("3.0"));
    }
```

As you can see in **Listing 3**, the test is as simple as a plain JUnit test. Before the test is run, the view will be constructed by the `getRootNode` method in **Listing 2**. You need only to

find the button you want to click via a literal by using the `find` method, which is provided by extending the `GuiTest` class. In case of an FXML view, the literal is the identifier (fx:id) of a UI element. This identifier has to be set to use the UI element in a controller class for bindings and event handling. This can be done either by setting it in Scene Builder or editing the FXML file itself. When using programmatically created views, this identifier has to be set with a `setId()` method call on the UI elements that are used in test classes.

After getting the buttons with `find()`calls, the buttons need to be clicked in this order: 1, +, 2, =. This can be done by using the `click()` method. Before clicking the = button, the calculator should display "1+2" in the field. After clicking the = button, the display should change and show "3.0" to indicate the result of the addition. The display text can be verified with `verifyThat()`, and the expected text should be an argument of the `hasText()` method call.

TestFX is a great framework for testing JavaFX applications. It is simple and intuitive, and beginners can learn it quickly. In addition, it offers a clear API that results in understandable tests, which facilitates diagnosing the errors that cause test failures.

Note that TestFX is compatible with the open source JaCoCo coverage tool and likely with other tools. It recognizes the lines of code that are covered by your TestFX tests, and you can track the code coverage as you can do with JUnit tests.

Tests That Fail

Another important part of unit tests is information about the ones that fail, especially when you are testing user interfaces. Failing tests should be interpretable, understandable, and easy to reconstruct. Therefore, it's important that the assertions be clear so it's evident what's wrong with the code when a failure occurs. In TestFX, Hamcrest matchers are helpful, because they provide more-readable assertions. **Listing 4** shows the simplicity of TestFX tests and the usage of Hamcrest matchers for verification.

Listing 4.

```
@Test
public void testMultiplication() {
    Button two = find("#two");
    Button times = find("#times");
    Button three = find("#three");
    Button equalSign = find("#equal");

    click(two);
    click(times);
    click(three);

    verifyThat("#display", hasText("2x3"));
    click(equalSign);
    verifyThat("#display", hasText("6.0"));
}
```

The test in **Listing 4** expects a value of 6. A bug is simulated by changing the multiply operation of the controller so that it multiplies two operands and spuriously adds "+1" at the end of the multiplication. This bug is introduced to show how TestFX deals with errors. Because of this bug, the test fails, and the corresponding stack trace looks like the following:

```
testMultiplication(...CalculatorControllerTest)
    Time elapsed: 2.434 sec  <<< FAILURE!
java.lang.AssertionError:
Expected: Node should have label "6.0"
     but: Label was "7.0" Screenshot saved as
          /home/…/screenshot1436949687849.png
 at ...testfx.Assertions
```

```
            .verifyThat(Assertions.java:38)
 at ...testfx.Assertions
            .verifyThat(Assertions.java:26)
 at . . .
 ...
Caused by: java.lang.AssertionError:
Expected: Node should have label "6.0"
     but: Label was "7.0"
 at org.hamcrest.MatcherAssert
       .assertThat(MatcherAssert.java:20)
 at org.loadui.testfx.Assertions
       .verifyThat(Assertions.java:33)
 ... 35 more
```

This (abbreviated) stack trace tells us in a very clear way which test failed, in which line, and with which condition, and it also tells us where the corresponding screenshot was saved.

Limitations

While TestFX is a very helpful project for testing JavaFX apps, it has some limitations you should be aware of.

One of the major limitations of TestFX is that debugging tests written in TestFX is not necessarily easy. TestFX doesn't allow mouse movements while running tests. If you are running tests in debug mode and you want to check the values of variables in the NetBeans view, for example, TestFX won't react on break points, and the test will also abort because of the mouse movement.

There is another major limitation when styling components with CSS, as I did with the calculator buttons for the sample calculator application. The styled component has two IDs: the controller ID and one in CSS. In this version of TestFX, the framework searches for the occurrence of `id` instead of `fx:id` and, therefore, it will use the CSS ID instead of the controller ID. This results in an error, because TestFX won't be able to find the respective button. The workaround for this issue is to use different CSS IDs for every object, which means that the same gradient will need to be applied to each button separately.

Another limitation is that screenshots are taken as full-screen screenshots. Instead of showing the UI window only, TestFX grabs the complete screen with all the other opened windows in the background. In most cases, the opened windows in the background are unimportant for the re-creation of failed tests. They offer too much information and could show private information about testers or developers, for example. In addition to that, there is currently no enable and disable mode for taking screenshots. Screenshots will be taken for every test that fails. Sometimes this is undesirable.

Also, there has been less development activity for TestFX lately. TestFX 4.0 has been in prerelease development since 2014. To ensure that there are future releases of this very useful software, it would be helpful if there were additional active developers.

Conclusion

TestFX is a great framework for testing JavaFX applications. It is simple and intuitive, and beginners can learn it quickly. In addition, it offers a clear API that results in understandable tests, which facilitates diagnosing the errors that cause test failures. Nevertheless, there are a few limitations, which you should keep in mind when using the current release of TestFX. `</article>`

LEARN MORE

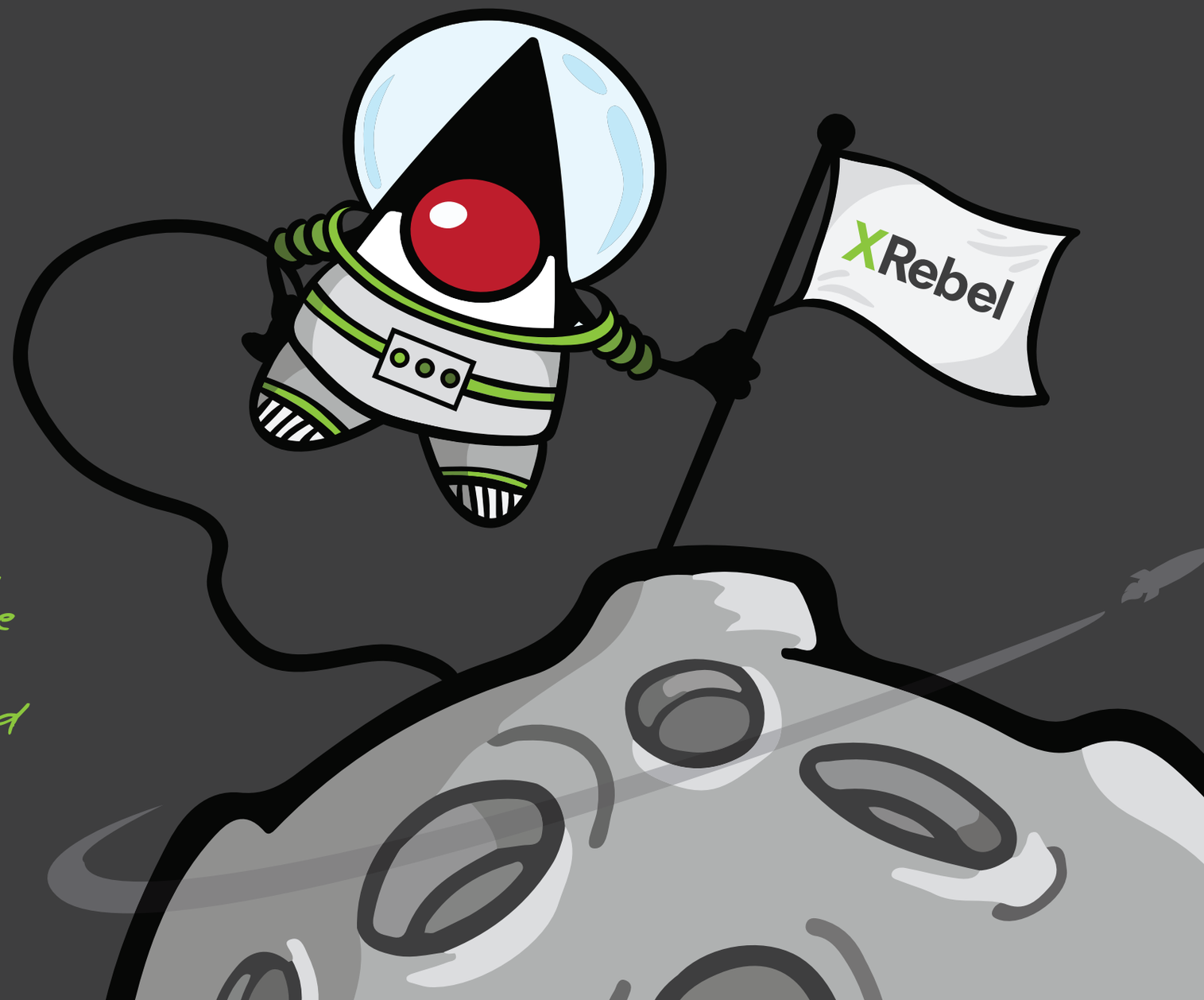
- Download the calculator
- Video lecture on TestFX, with additional usage details
- Gluon Scene Builder 8.0.0

FIND ISSUES AS YOU CODE

with XRebel The Lightweight Java Profiler

TRY IT FREE NOW!

One small step for Duke
one giant leap for Java-kind



ZEROTURNAROUND

MERT ÇALIŞKAN

BIO

Eight Greatly Underused Features of JUnit

Make your testing a whole lot easier.

When it comes to unit testing, we can safely say that JUnit is the most popular test framework among Java developers. Despite the headline on the official JUnit website stating that it's a *simple* framework, there is a good deal of extensive functionality bundled within it. Unfortunately, most of us use a familiar set of features, and so we duplicate available goodies by reinventing them. With this article, I dig into eight convenient but underused features of JUnit that should ease writing test code—including creating and automatically cleaning up temporary directories, running all assertions despite one failing, killing wayward tests, and the like.

All the examples in this article are based on JUnit version 4.12, which is the latest available version at the time of this writing. However, unless otherwise noted, all the features mentioned here have been available since JUnit 4.7. The complete source code for the article is available at the *Java Magazine* download site.

First Things First: @Rule

I will make use of JUnit's `@Rule` annotation, which offers a generic way to add extended features on a per-test-method basis. By employing rules, the various activities I want to enable can be stated in one place, and I won't need to define them over and over again for each test method.

The `@Rule` annotation applies to a public field of a test class, which should be an implementation of either the `TestRule` or `MethodRule` interface. Both interfaces contain the method signature: `Statement apply(…)`. Implementation of this `apply()` method can be executed before, after, or instead of a test method. This approach adds flexibility for applying new behaviors for each test method, à la interceptors.

JUnit 4.9 introduced the `@ClassRule` annotation that extends the test method rule approach to the class level so that it can be invoked before or after *all* test methods of that class. It's efficient to use this capability, if, for example, a server needs to be started before running all the test methods so that it won't be started and shut down for each test method inside a test class.

Creating Temporary Files and Folders Using @Rule

From time to time, you might need to create certain files and folders in your application, but how would you test that with code to see whether the files are created as expected? You can do all the work in a temporary folder with some extra coding, but most solutions present portability conflicts if the test environment is moved. In addition, any handmade solution can get messy if cleanup is not done carefully.

Since version 4.7, JUnit has offered the `TemporaryFolder` test rule, which provides a convenient way to create and manage a temporary directory. It guarantees that the new

directory's name does not conflict with existing files or directories. It also guarantees that content will be deleted after the test method finishes, whether the execution result is successful or not.

Listing 1 creates a file named `sample.txt` in a temporary folder that JUnit creates and then writes "JUnit Rocks!" The test then reads the first line of the newly created file and compares the text value that was written. For testing purposes, the Apache Commons IO framework is used for the simplicity that it provides in handling files.

Listing 1.

```
public class TemporaryFolderTests {

    @Rule
    public TemporaryFolder tempFolder =
        new TemporaryFolder();

    @Test
    public void fileCreatedAndWrittenSuccessfully()
            throws IOException {
        File file = tempFolder.newFile("sample.txt");
        FileUtils.writeStringToFile(
            file, "JUnit Rocks!");

        String line = FileUtils.readFileToString(file);
        assertThat(line, is("JUnit Rocks!"));
    }
}
```

JUnit deletes the file and the directory once the test is completed (regardless of whether the test passes or fails).

Getting the Name of the Currently Executing Test

If you'd like to get the name of a test while it's executing, simply instantiate a `TestName` rule as a field and annotate it with the `@Rule` annotation, as shown in **Listing 2**.

Listing 2.

```
public class TestNameTests {

    @Rule
    public TestName name = new TestName();

    @Test
    public void methodNameShouldBePrinted() {
        System.out.println("Test method name: " +
                           name.getMethodName());
    }
}
```

The name of the test method can be accessed via `getMethodName()`. The output of the test method will be

```
Test method name: methodNameShouldBePrinted
```

Collecting Errors When Multiple Assertions in One Test Fail

With the `ErrorCollector` class, JUnit offers a way to handle multiple test failures of a single test case. So, you can enable a test not to stop on an error by doing all assertions and listing the failed ones at the end. **Listing 3** shows a test method with three statements that are checked with the `ErrorCollector`.

Listing 3.

```
public class ErrorCollectorTests {

    @Rule
    public ErrorCollector collector =
        new ErrorCollector();

    @Test
    public void statementsCollectedSuccessfully() {
        String s = null;
        collector.checkThat
            ("Value should not be null", null, is(s));

        s = "";
        collector.checkThat(
            "Value should have the length of 1",
            s.length(), is(1));
```

```
        s = "Junit!";
        collector.checkThat(
            "Value should have the length of 10",
             s.length(), is(10));
    }
}
```

Note the use of the `ErrorCollector` instance in the tests themselves. If the first statement passes but the next two fail, the output is as follows (condensed here by removing the stack trace detail):

```
java.lang.AssertionError: Value should have the length of 1
Expected: is <1>
     but: was <0>
       at org.hamcrest.MatcherAssert...

java.lang.AssertionError: Value should have the length of 10
Expected: is <10>
     but: was <6>
       at org.hamcrest.MatcherAssert...
```

Running One Test Multiple Times with Different Parameters

It is fairly common to need to run the same method with varying inputs to see that the executed results are asserted successfully. Since version 4.0, JUnit provides the flexibility to do this with the `@Parameters` annotation, which enables you to run one test method again and again providing different input values each time.

To demonstrate the parameterized usage, I will test a method that calculates a Fibonacci number according to a given index value of it in the sequence. To get the `@Parameters` annotation picked up, I need to state that the test class should be handled with a custom runner, which is `Parameterized.class`, rather than with the default runner that JUnit uses.

To provide input values for the Fibonacci series, a public static method with a list of objects should be defined, and it should be marked with the `@Parameterized.Parameters` annotation. Each element of that list will be provided as a parameter to the constructor of the test class. Thus, the number of parameters passed to the constructor must match the item count of each element of the list. **Listing 4** shows how this is done.

Listing 4.

```
@RunWith(Parameterized.class)
public class FibonacciNumbersTests {

    @Parameterized.Parameters
    public static List data() {
        return Arrays.asList(new Object[][]{
                {0, 0}, {1, 1}, {2, 1},
                {3, 2}, {4, 3}, {5, 5},
                {6, 8}});
    }

    private int value;
    private int expected;

    public FibonacciNumbersTests(
            int input, int expected) {
                value = input;
                this.expected = expected;
    }

    @Test
    public void fibonacciNumberCalc () {
        assertEquals(expected, fib(value));
    }

    public static int fib(int n) {
        if (n < 2) {
            return n;
        } else {
            return fib(n - 1) + fib(n - 2);
        }
    }
}
```

Note the first line with the `@RunWith` annotation, which tells JUnit to use a specialized runner.

Here, with each item, we are providing the *input* value first and then the *expected* value to make the assertion. For purposes of illustration, I have included the `fib()` method that we are testing as a static method within the test class. Also note that the test class can have only one constructor defined. Multiple constructor declarations will lead to an illegal argument exception.

Adding a Timeout for Execution

It is frequently useful to be able to kill a test that is taking too long to execute. Unexpected stalls, for example, can turn quick runs of unit tests into long-lasting affairs. The `@Test` annotation offers the `timeout` attribute, which specifies a time that, if exceeded, causes a test method to fail. The `timeout` attribute uses milliseconds. A time limit applied to all tests in a test class is a common need, and JUnit offers the `Timeout` rule class to handle this. It can use units of seconds or milliseconds. **Listing 5** defines a global timeout rule with five seconds and a test method that never finishes.

Listing 5.

```
public class LongRunningTests {

    @Rule
    public Timeout globalTimeout = Timeout.seconds(5);

    @Test
    public void whatWeDoInATestMethodEchoesInEternity() {
        while(true);
    }
}
```

An excerpt of the output of this test after five seconds of execution will be as follows:

```
org.junit.runners.model.TestTimedOutException:
   test timed out after 5 seconds
      at tr.com.t2.labs.tdd.sample5.LongRunningTests…
```

Grouping Your Tests with @Category

In a sophisticated build process, it's frequently useful to group and run the fast-running tests first to get quick feedback from them. To achieve this, JUnit 4.8 first offered the `@Category` annotation to group tests with a marker interface or a class. **Listing 6** illustrates two test methods, one annotated with `@Category`.

Listing 6.

```
public class CategorizedTests {

    @Test
    @Category(SlowTests.class)
    public void thisTestRunsSlowly() {
        System.out.println("Slow test running");
    }

    @Test
    public void thisTestRunsFast() {
        System.out.println("Fast test running");
    }
}
```

`SlowTests.class` is just a marker interface that is used to determine the test methods that will run more slowly than normal:

```
public interface SlowTests {
}
```

One approach to run the categorized tests is to implement a test suite class. **Listing 7** gives an example of this by employing a custom runner, which is `Categories.class`. Without the custom runner, `@Category` annotations cannot be interpreted. In order to state which categories should run within this test suite, the `@IncludeCategory` annotation with the value of the `SlowTests` marker interface is used. All subclasses of the `SlowTests` interface will also be picked up with this category inclusion.

Listing 7.

```
@RunWith(Categories.class)
@Categories.IncludeCategory(SlowTests.class)
@Suite.SuiteClasses(CategorizedTests.class)
public class SlowTestsTestSuite {
}
```

Besides the test suite approach, JUnit categories are supported by popular build tools—such as Maven, Gradle, and SBT—to run specific groups of tests. You can exclude a category with the `@ExcludeCategory` annotation, which operates precisely as you would expect.

Creating Your Own Rules

Implementing your own rules is as simple as implementing the `TestRule` interface. This interface has only one method, `apply()`. You should implement the `apply()` method and return an instance of `Statement`, which is an abstract class that provides an `evaluate()` method. An outcome will be evaluated from that method when a rule is applied with the invocation of the `apply()` method. **Listing 8** gives an example implementation of a custom rule.

Listing 8.

```
public class MyCustomRule implements TestRule {

    private String label;

    public MyCustomRule(String label) {
        this.label = label;
    }

    @Override
    public Statement apply(
        final Statement base,
        Description description){
        return new Statement() {
            @Override
            public void evaluate() throws Throwable
            {
                System.out.println(label + " before");
                base.evaluate();
                System.out.println(label + " after");
            }
        };
    }
}
```

Using our custom rule is done in the same way as using the built-in rules provided by JUnit. **Listing 9** shows the usage.

Listing 9.

```
public class CustomRuleTests {

    @Rule
    public MyCustomRule myCustomRule =
        new MyCustomRule("custom");

    @Test
    public void myAwesomeMethodInvokedSuccessfully() {
        System.out.println("Test worked OK");
    }
}
```

The following is the execution output:

```
custom before
Test worked OK
custom after
```

Chaining Rules

Once you start working with rules, especially when writing your own, you might want to chain them so that they run in a specific order. With the `RuleChain` rule, it's possible to order multiple rules according to your needs. This feature, which has been available only as of JUnit 4.10, is effective if you need to do configuration in a specified order, such as configure your web server with one rule and then start it with another rule.

The `outerRule()` method defines the first rule to execute and the

`around()` method defines the subsequent rules. **Listing 10** gives an example implementation that employs my previously created custom rule definition, the `MyCustomRule` class from **Listing 9**.

Listing 10.

```
public class RuleChainTests {

    @Rule
    public RuleChain chain = RuleChain.outerRule(
        new MyCustomRule("outer"))
            .around(new MyCustomRule("inner"));

    @Test
    public void ruleChainWorkedOK() {
        System.out.println("Test worked OK");
    }
}
```

This is the execution output of the test class:

```
outer before
inner before
Test worked OK
inner after
outer after
```

Conclusion

JUnit contains many options that facilitate testing. JUnit's developers know that the framework is used for far more than unit testing. They've provided numerous capabilities for running large test suites in various ways, for customizing test runs, and for handling many of the minor "busy" tasks that testing requires. Pay attention to new releases of JUnit for small additions that make automated testing easier. `</article>`

LEARN MORE

- A good tutorial on JUnit
- JUnit Anti-Patterns

FEATURED JDK ENHANCEMENT PROPOSAL

JEP 259: Stack-Walking API

[JDK Enhancement Proposals (JEPs) are proposals that allow OpenJDK committers and other participants to come together on a project that might eventually evolve into a full-blown JSR. Having covered many JSRs in past issues of *Java Magazine*, we'll be looking at some of the more interesting JEPs. —*Ed.*]

JEP 259: Stack-Walking API proposes an API for walking the stack. The goal is that at any moment (although almost certainly during debugging), the API will enable a developer to walk up or down the entire calling stack. As currently proposed, the API, which is intended to have only three principal methods, can locate a specific stack frame (based on filter parameters) and locate a caller's stack frame. A possible additional method that would return a stream of stack frames has been proposed, although it's not currently part of the JEP. It should be noted that the entire stack trace can be accessed via existing APIs, such as `getStackTrace()` in `Throwable`. This JEP simply refines the access to specific stack frames. More information on the JEP can be found here.

NEIL MANVAR

BIO

Building and Automating a Functional Test Grid

How to assemble a grid for Selenium testing

On July 9, 2015, Apple released Mac OS X El Capitan beta, which offered significant new functionality. In this release, Apple introduced the mandate that all applications use SSL with a certificate that exceeds 1024-bit encryption. Google Chrome has also launched an update that now flags all SSL certificates that are from authorities not on the whitelist (that is, those that do not have public records) and certificates that use the (insecure) SHA-1 hash function.

What does all this have to do with functional testing? These seemingly innocuous changes will make users of Safari and the latest update of Chrome question security, and they could possibly raise some uncomfortable questions. The only way for IT organizations to spot these issues is by using a strong and current functional testing grid. In this article, I discuss how to set up your own testing grid for browser-based apps using Selenium, the widely used open source testing tool.

The Grid

The idea of running manual tests on a browser of a particular type and version is straightforward. But applications in the wild do not run on just one type or version of a browser. No matter what browser your customers use, they expect a quality experience.

A testing grid without automation provides only modest help to an organization.

You soon find that the matrix of browser types and versions gets complicated quickly. Add the OS versions and configurations, Selenium versions, and individual browsers' web drivers, and you suddenly have an unmanageably complex test environment. This calls for automation. Without automation, a testing grid provides only modest help to an organization.

The Balance

When it comes to testing grids, the entire test suite should be run against the top 80 percent of browsers used. This rule is important, because testing on less common browsers has an effort cost that is greater than the impact of the potential issues that are found. The remaining browsers should be available for testing when specific issues arise, usually from some support request. The idea is to create a system where those browsers could be available for testing on demand.

According to the online course site W3Schools, its traffic shows that 97 percent of browser usage is shared among Chrome, Firefox, Internet Explorer, and Safari, in that order. Sites with a more business-oriented audience see a greater ratio of Internet Explorer, especially older versions of the browser. This shows that it is hard to predict what combinations of browser, version number, and operating system releases a given application must be tested for

The caveat to the 80-percent rule is effort cost. Most shops find that they don't even have the ability to test the top 50

percent of browsers. The reason: Even the top 50 percent include a lot of variation when you add operating systems. (Most of the time, it ends up that only Chrome and Firefox are no more than two versions back.)

You can safely say that to have a testing grid with approximately 80 percent testing coverage, you would need to have instances of Chrome C42 through C45, Firefox 37 through Firefox 40, Internet Explorer 9 through 11, and Safari S7 and S8 on Mac OS X 10.8 through 10.10, Microsoft Windows 7 through 8.1 and XP, Debian, Ubuntu, CentOS, Red Hat, Gentoo, Fedora, and SUSE—for a total of 226 combinations. This really is not possible without a cloud-based solution. An on-premises combination is usually Chrome C43 through C45, Firefox 38 through Firefox 40, Internet Explorer 9 through 11, and Safari S7 and S8 running on Mac OS X 10.9, Windows 7 and 8, and Debian.

When considering your grid, the team needs to keep current on the ecosystem. There are periodically some large changes in the ecosystem. For example, some notable recent and upcoming events are the launch of Mac OS X 10.11 and Microsoft's Edge browser in Windows 10, and the end of Chrome support for Windows XP at the close of 2015.

The best environment is where every automated test automatically provisions a new VM, runs the test, and shares the results with the QA team.

When teams set up testing in the cloud, they can test even more-obscure browser versions and types on demand.

The lowest common denominator of what is possible is based on your organization's infrastructure capabilities, and your QA team's ability to use them. It is not one size fits all.

The Needed Infrastructure

Installing browsers on local machines prohibits flexible functional testing, and in the world of automation, it is not a sustainable approach. To implement your testing environment, you will need a virtual machine hypervisor or container. (From here on, I will refer to all types of containers and virtual machines as "VMs.") To be most effective, the QA team needs access to VMs and the ability to control the provisioning of VMs.

Before you start installing your browsers, you need to consider some very important questions, such as

- Should you put browser installations on the same VM?
- Should you use a new VM for each?
- Should you use a new VM per group of browsers?

As an example, putting all browsers on the same VM offers the benefit of bringing everything together in one place, thus making it easier to deploy and run automated scripts. However, it does not allow you to test against different versions of the same browser type in parallel, and it increases the risk of environmental variables that could affect functionality, especially over time when reused.

Automation helps with infrastructure as well. Automation helps not only to run tests but also to provision the ideal environment's infrastructure. Usually, developers and QA engineers are hands-off when it comes to automation. This approach is not always beneficial. It encourages an immutable infrastructure that is reused and contaminated, and it ultimately limits the test grid variation mix and throttles the number of tests that can be run at any given moment in time. With cloud solutions, automation infrastructure does not matter, because provisioning is the job of the QA-tailored testing solution.

There are three quality levels for environments: workable, good, and best. And it is not always possible to implement the ideal environment.

Workable environments. In these environments, you provision a hub VM once. It then acts as the Selenium Hub (formerly, the Selenium Remote Control), which can distribute tests across your test nodes. Test nodes are also VMs that have been provisioned once, and they house the actual browser

configurations. On the nodes, the client libraries run the tests provided by the hub. Repeated tests are done automatically by rerunning the test suite from the hub. The system is only as strong as the weakest node, which means that because you are repeatedly using the same infrastructure over time, the risk of something breaking increases.

Here is how you start the hub:

```
java -jar selenium-server-standalone-2.44.0.jar
     -role hub
```

Then start the node(s) connected to the hub:

```
java -jar selenium-server-standalone-2.44.0.jar
     -role node
     -hub http://localhost:4444/grid/register
```

If you are creating a Selenium grid with many nodes, then for each node, you have to specify the hub:

```
java -jar selenium-server-standalone-2.44.0.jar
     -role node
     -hub http://<IP_ADDR_OF_HUB>/grid/register
```

Good environments. These environments take *workable* one step further and go from immutable testing of VMs that are rarely reprovisioned to using a new VM with every major automated test run. In order to do this, you need to leverage orchestration tools such as Puppet, Vagrant, Chef, and others.

Best environment. The best environment could also be called continuous integration, where every automated test automatically provisions a new VM, runs the test, and collects and shares the results with the QA team. It also does this on a select number of commits or even on every commit by the developers, allowing the team, without any manual effort, to identify bugs very early in the development process.

Test Automation

Using automation tools such as Selenium breaks teams free from the risk and limitations associated with manual testing. Automation is far more scalable, can be run anytime without human intervention, and allows the expansion of test cases.

But automation requires infrastructure and setup. The first step is to decide what you will test on—that is, the combination of operating systems and browsers. Then you need to create a baseline infrastructure that the browsers will run on. And, finally, you need to wire up the automation processes with tools or direct API calls.

Setup of Browser Grids

I recommend the following steps:

1. Select your host operating system(s). Most of the time, the operating system does not have a large impact on browser functionality. The notable changes will be in performance, because operating systems optimize the running of processes—as well as security—with new releases, as seen with the operating system releases mentioned earlier. These releases typically don't have a large impact on functional testing, but they could affect application performance or the browser experience. At a minimum, if you are testing on a common Windows, Linux, or Mac OS X platform that is no older than two versions back, you have covered a substantial portion of your host operating systems.
2. Next, create a gold master VM for each of the test nodes, and create one gold master for the Selenium Hub. The gold master for the test nodes contains the OS configuration, the browsers already installed, and their associated web drivers. Once you have decided what browser mix will go on your VM, it is best to create a base instance of each VM with the browser mix that will make up your entire grid, and have a snapshot of the VMs for real-time provisioning based on test runs.

Updates to browsers and configurations will happen, and when they are completed, a new snapshot should be taken, creating an ever-expanding library of browser versions and types.

3. You now run the Selenium Hub and connect it to all the running test nodes. You can initiate a test directly from the Selenium Hub.

Always provision on a clean VM to avoid contamination and to create a history of snapshots, both of which are essential for the versioning of your grid. The snapshots will allow you to go back to previous browser versions, if necessary, for support reasons.

Wiring up Test Automation

Once you have the infrastructure, you need to wire up the automation capability. Selenium is used to drive the tests. Listing 1 shows a basic example that will navigate the browser to a Google page and submit a query. A simple test might look like this.

Listing 1.

```
import org.openqa.selenium.By;
import org.openqa.selenium.Platform;
import org.openqa.selenium.WebDriver;
import org.openqa.selenium.WebElement;
import org.openqa.selenium.remote.*;
import java.net.URL;

public class SampleSeleniumTest {
    public static final String USERNAME =
        System.getenv("USERNAME");
    public static final String ACCESS_KEY =
        System.getenv("ACCESS_KEY");
    public static final String URL =
       "http://<ADDR_OF_SELENIUM_HUB>:4444/wd/hub";

    // if localhost, then:
    // public static final String URL =
    //     "http://localhost:4444/wd/hub";

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

      DesiredCapabilities caps =
          new DesiredCapabilities();
      caps.setCapability(
          CapabilityType.BROWSER_NAME, "firefox");
      caps.setCapability(
          CapabilityType.VERSION, "37");
      caps.setCapability(
          CapabilityType.PLATFORM, "Windows 7");

      WebDriver driver =
          new RemoteWebDriver(new URL(URL), caps);
      driver.get("http://www.amazon.com");
      WebElement element =
          driver.findElement(
              By.name("field-keywords"));

      element.sendKeys("Sauce Labs");
      element.submit();

      System.out.println(driver.getTitle());
      driver.quit();

    }
}
```

The script is run on the same machine on which the browser resides. Selenium has a large set of functions for finding and executing web applications. However, running the test without results is useless. You also need to track data on how the test ran.

Test Results

The basic outcomes of any test are PASS, FAIL, or SKIP. If you

are creating the reporting element of your testing environment, you will need to programmatically collect this data and report it to an output of your choosing—commonly XML via JUnit, HTML, or CSV with Microsoft Excel. However, there are tools that help with the depth of test reporting, with custom web-based dashboards built for reporting. This approach is ideal, due to faster visuals, and it makes sharing data with the team easier.

Below is a sample JUnit output generated by TestNG that includes a failed test:

```
<testcase classname="com.yourcompany.SampleSauceTest"
    name="pandoraTitleTest" time="14.593"/>
<testcase classname="com.yourcompany.SampleSauceTest"
    name="welcomeScreenLaunchTest" time="14.664"/>
<testcase classname="com.yourcompany.SampleSauceTest"
    name="coldplayTest" time="16.538"/>
<testcase classname="com.yourcompany.SampleSauceTest"
    name="welcomeScreenLaunchTest" time="16.549"/>
<testcase classname="com.yourcompany.SampleSauceTest"
    name="welcomeScreenLaunchTest" time="19.71"/>
<testcase classname="com.yourcompany.SampleSauceTest"
    name="welcomeScreenLaunchTest" time="20.694"/>
<testcase classname="com.yourcompany.SampleSauceTest"
    name="welcomeScreenLaunchTest" time="25.121"/>
<testcase classname="com.yourcompany.SampleSauceTest"
    name="pandoraTitleTest" time="7.073"/>
<testcase classname="com.yourcompany.SampleSauceTest"
    name="pandoraTitleTest" time="8.906"/>
<testcase classname="com.yourcompany.SampleSauceTest"
    name="coldplayTest" time="12.206">
<failure message="stale element reference:
    element is not attached to the page document
 (Session info: chrome=41.0.2272.76)
 (Driver info: chromedriver=2.15.322448
              ConstructorAccessorImpl.java:45)">
</failure>
```

But PASS, FAIL, and SKIP do not tell you a great deal about how and when a test failed. That is why it is best to report other metadata such as timestamps, page elements, test case names, and even stack traces—anything that will support faster identification of the location of the error and assignment of a ticket item to a developer.

In addition to the various data elements you can collect, you should consider taking a screenshot of every exception and failure, as shown in **Listing 2**. For extra credit, you can capture video as well. This requires integration with video recording, which demands a lot of effort, but some cloud providers have this functionality built in.

Listing 2.

```
catch(Exception e)
{
    Assert.fail(); //To fail test in case of any
                   //element identification failure
}
```

and

```
public void takeScreenShotOnFailure(
    ITestResult testResult) throws IOException
{
    if (testResult.getStatus() ==
        ITestResult.FAILURE)
    {
        System.out.println(
            testResult.getStatus());
        File scrFile =
            ((TakesScreenshot)driver)
            .getScreenshotAs(OutputType.FILE);
        FileUtils.copyFile( scrFile,
             new File("D:\\testScreenShot.jpg"));
    }
}
```

By storing the screenshots in a location that is associated with a build, application version, test run ID, and so on, you can quickly focus only on the exceptions. It is also useful to tie these back to the elements being acted on at the time, so that you can quickly reference your Selenium script and know what test was being run.

Best Practices

Once you set up your test grid and begin using it, you'll quickly find that certain basic best practices will go a long way to giving you the results you need. These practices include the following:

- **Avoid contamination.** System-level bugs are the hardest to track down. And the likelihood of having system-level issues when reusing VMs is incrementally higher with each test run. (It cannot be overstated that you should use a new VM for each test run.)
- **Establish consistent naming conventions.** As the number of test cases increases, the management of the entire library becomes more complex. Using a consistent naming convention for tests (for example, tying them to the application name and release) will help substantially when it comes time to cull unneeded tests or rerun old tests for issues that resurface.
- **Ensure test results are not stored on VMs directly.** Reporting is key. In order to build in autonomy and support testing on a new VM with each test run, make sure all the processes store test results and screenshots in a centralized location.
- **Build a workflow for exceptions.** Beyond video and screenshots, building an entire workflow for exceptions is recommended. Some teams take automation too far and have all exceptions go directly into tickets. But the false positives waste a lot of time. Don't let the tools create the workflow for you.

> If you are not doing automated testing on a functional test grid today, **you will be soon.**

What to Be Aware Of

Bear in mind the following concerns:

- **On-premises testing.** Choosing to maintain the testing infrastructure in-house is naturally limiting. For small, manual testing, it works fine. But when automation picks up, it is usually necessary to turn to cloud infrastructure and automated tools, ranging from provisioning to running tests on new instances.
- **Testing on the Mac OS.** Testing on the Mac OS is not simple. The primary reason is licensing. In order to comply with Mac OS X licensing, you need to test on a physical Mac. You can virtualize, but when you do, the VM also has to be on a physical Mac.
- **Mobile testing.** Mobile testing and mobile applications are coming into play rapidly—faster than the established methods for testing. It can feel like the Wild West. However, many tools are already available, such as Appium, which will allow Selenium-style automated testing on emulators. (A lot of the elements are the same as with web application testing, but the testing grid size is multiplied nearly by four.)

Conclusion

If you are not doing automated testing on a functional test grid today, you will be soon—so it is important to think about how you will set up the process. On-demand setup of a testing grid most often results in something that is not sustainable and doesn't leverage all the possibilities of automation. Today, however, you do not always need to worry about infrastructure. QA teams can actually take over the infrastructure. Teams can elect to use cloud-based testing, and then spend most of their time writing test cases and building test strategies. `</article>`

Stress Testing Java EE Applications

Identify application server configuration problems, potential bottlenecks, synchronization bugs, and memory leaks in Java EE code.

ADAM BIEN

BIO

Unit and integration tests are helpful for the identification of business logic bugs, but in the context of Java EE 6 and later, they are meaningless. Both integration and unit tests access your application in a single-threaded way. After the deployment, however, your code will always be executed concurrently.

Stop Talking, Start Stressing

It is impossible to predict all nontrivial bottlenecks, deadlocks, and potential memory leaks by having theoretical discussions. It is also impossible to find memory leaks with unit and integration tests. Bottlenecks are caused by locks and I/O problems that are hard to identify in a single-threaded scenario. With lots of luck and patience, memory leaks can be identified with an integration test, but they can be far more easily spotted under massive load. The heavier the load, the greater is the probability of spotting potential concurrency and robustness problems. Cache behavior, the frequency of Java Persistence API (JPA) `OptimisticLockException`, and the amount of memory needed in production can also be evaluated easily with stress tests.

Even in the unlikely case of a perfect application without defects, your application server will typically be unable to handle the load using default factory settings. Stress tests are a perfect tool to learn the behavior of your application under load in the first iterations without any stress. Stress-test-driven development is a good choice for Java EE.

Instead of applying optimizations prematurely, you should concentrate on implementing pure business logic and verifying the expected behavior with automated tests and hard numbers.

Don't Be Realistic

Load tests are configured by taking into account the expected number of concurrent users and realistic user behavior. To meet the requirements, "think times" need to be kept realistic, which in turn reduces the amount of concurrency in the system. The heavier the load, the easier it is to find problems. Realistic load tests are usually performed too late in the development cycle and are useful only for ensuring that nonfunctional requirements are met. They are less valuable in verifying system correctness.

Instead of relying on realistic but lax load tests defined by domain experts, we should utilize as much load as possible using developer-driven stress tests. The goal is not to verify expected scalability or performance. Instead, it is to find defects, of course, and to learn about the behavior of the system under load.

Stressing the Oracle

My "oracle" application records predictions and returns them as a JavaScript Object Notation (JSON) string. (For more information on the "oracle" application, see "Unit Testing for Java EE".)

PHOTOGRAPH BY THOMAS EINBERGER/ GETTY IMAGES

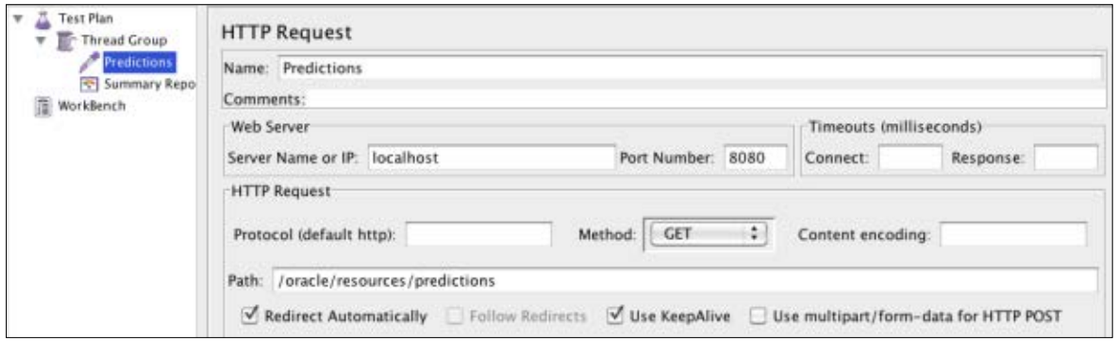


Figure 1. HTTP request configuration in JMeter

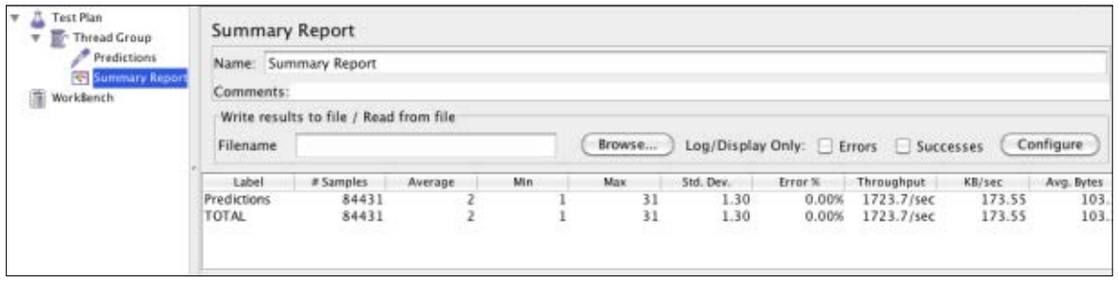


Figure 2. JMeter Summary Report

Figure 3. VisualVM CPU and memory monitoring overview

Sending a GET request to the URI http://localhost:8080/oracle/resources/predictions returns a `Prediction` entity serialized as:

```
{"prediction":
   {"result": "JAVA_IS_DEAD",
    "predictionDate":
      "1970-01-01T19:57:39+01:00",
    "success":"false"}
}.
```

Services provided by Java API for RESTful Web Services (JAX-RS) are easily stress-testable; you need only execute several HTTP requests concurrently.

The open source load-testing tool Apache JMeter comes with built-in HTTP support. After creating the `ThreadGroup` and setting the number of threads (and, thus, concurrent users), an HTTP request has to be configured to execute the GET requests (right-click, select Sampler, and then select HTTP Request). See **Figure 1**.

While the results can be visualized in various ways, the JMeter Summary Report is a good start (see **Figure 2**). It turns out that the sample application is able to handle 1,700 transactions per second for five concurrent users "out of the box."

Every request is a true transaction and is processed by an Enterprise JavaBeans 3.1 (EJB 3.1) JAX-RS `PredictionArchiveResource`, delegated to the `PredictionAudit` EJB 3.1 bean, which in turn accesses the database through `EntityManager` (with exactly one record).

At this point, we have learned only that with EJB 3.1, JPA 2, and JAX-RS, we can achieve 1,700 transactions per second without any optimization. But we still have no idea what is happening under the hood.

VisualVM Turns Night to Day

GlassFish Server Open Source Edition 3.1.x and Java DB (the open source version of Apache Derby) are Java processes that can be easily monitored with VisualVM. Although VisualVM is

shipped with the current JDK, you should check the VisualVM Website for updates.

VisualVM is able to connect locally or remotely to a Java process and monitor it. VisualVM provides an overview showing CPU load, memory consumption, number of loaded classes, and number of threads, as shown in **Figure 3**.

The overview is great for estimating resource consumption and monitoring the overall stability of the system. We learn from **Figure 3** that for 1,700 transactions per second, GlassFish Server Open Source Edition 3.1 needs 58 MB for the heap, 67 threads, and about 50 percent of the CPU. The other 50 percent was consumed by the load generator (JMeter).

Although this colocation is adequate for the purposes of this article, it blurs the results. The load generator should run on a dedicated machine or at least in an isolated (virtual) environment. Sometimes, you even have to run distributed JMeter load tests to generate enough load to stress the server. For internet applications, it might be necessary to deploy the load generators to the cloud.

In a stress-test scenario, the plain numbers are interesting but unimportant. Stress tests do not generate a realistic load but, rather, they try to break the system. To ensure stability, you should monitor the VisualVM Overview average values. All the lines should be, on average, flat.

An increasing number of loaded classes might indicate problems with class loading and can lead to an `OutOfMemoryError` due to a shortage of `PermGen` space. An increasing number of threads indicates slow, asynchronous methods. A `ThreadPool` configured with an unbounded number of threads will also lead to an `OutOfMemoryError`. And a steady increase in memory consumption can eventually lead to an `OutOfMemoryError` caused by memory leaks.

VisualVM comes with an interesting profiling tool called Sampler. You can attach and detach to a running Java process with a little overhead and measure the most expensive invocations or the size of objects (see **Figure 4**).

The sampling overhead is about 20 percent, so with an active sampler, you can still achieve 1400 transactions per second. As expected, the application spends the largest amount of time communicating with the database.

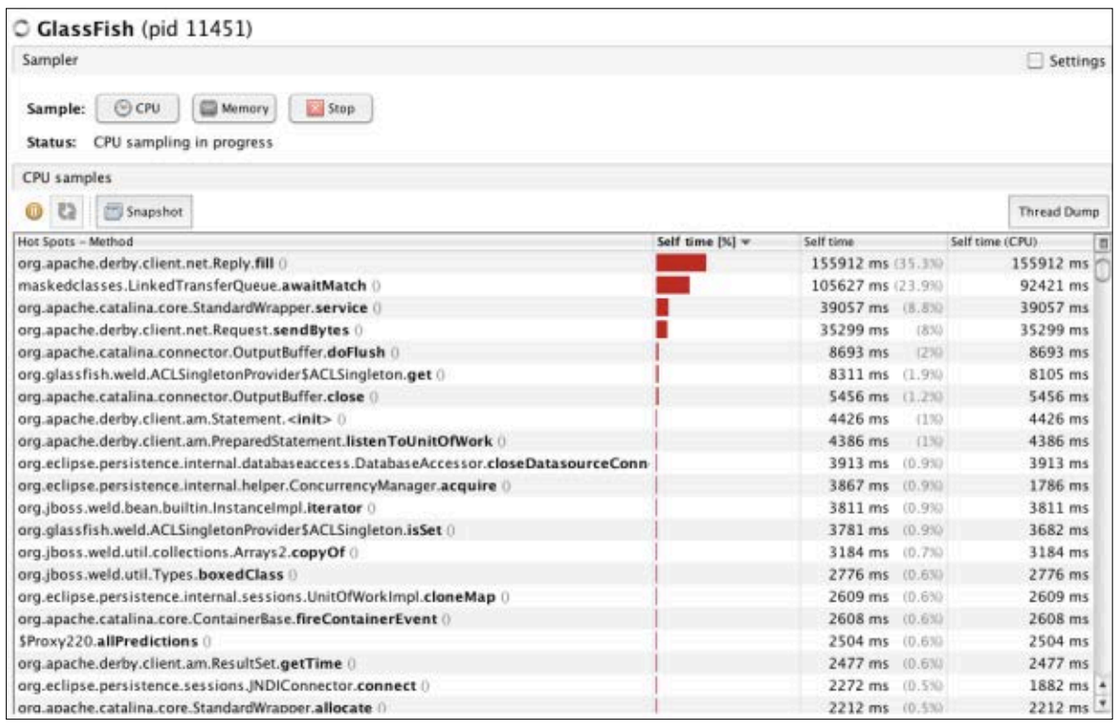


Figure 4. CPU monitoring with Sampler

How Expensive Is System.out.println?

A single `System.out.println` can lead to significant performance degradation. To measure the overhead, every invocation of the method `allPredictions` is logged with a `System.out.println` invocation, as shown in **Listing 1**.

Listing 1.

```
public List<Prediction> allPredictions(){
    System.out.println("-- returning predictions");
    return this.em
       .createNamedQuery(Prediction.findAll)
       .getResultList();
}
```

Write results to file / Read from file

Filename Browse... Log/Display Only: Errors Successes Configure

Label	# Samples	Average	Min	Max	Std. Dev.	Error %	Throughput	KB/sec	Avg. Bytes
Predictions	196380	5	1	226	7.42	0.00%	842.8/sec	84.82	103.1
TOTAL	196380	5	1	226	7.42	0.00%	842.8/sec	84.82	103.1

Figure 5. Performance degradation of 50 percent

CPU samples

Snapshot Thread Dump

Hot Spots – Method	Self time [%]	Self time	Self time (CPU)
org.apache.felix.gogo.runtime.threadio.ThreadPrintStream.println ()		199065 ms (36.7%)	63527 ms
org.apache.derby.client.net.Reply.fill ()		113520 ms (20.9%)	113520 ms
maskedclasses.LinkedTransferQueue.awaitMatch ()		113170 ms (20.9%)	66655 ms
org.apache.catalina.core.StandardWrapper.service ()		23189 ms (4.3%)	23189 ms
org.apache.derby.client.net.Request.sendBytes ()		23023 ms (4.2%)	23023 ms
org.glassfish.weld.ACLSingletonProvider$ACLSingleton.get ()		7089 ms (1.3%)	6995 ms
org.apache.catalina.connector.OutputBuffer.doFlush ()		4262 ms (0.8%)	4262 ms
org.apache.catalina.connector.OutputBuffer.close ()		4240 ms (0.8%)	4240 ms
org.eclipse.persistence.internal.helper.ConcurrencyManager.acquire ()		3651 ms (0.7%)	1280 ms
org.eclipse.persistence.internal.databaseaccess.DatabaseAccessor.closeDatasourceConn		2835 ms (0.5%)	2835 ms
org.glassfish.weld.ACLSingletonProvider$ACLSingleton.isSet ()		2754 ms (0.5%)	2754 ms
org.apache.derby.client.am.Statement.<init> ()		2210 ms (0.4%)	2210 ms
org.jboss.weld.bean.builtin.InstanceImpl.iterator ()		2083 ms (0.4%)	2083 ms
org.jboss.weld.util.Types.boxedClass ()		1943 ms (0.4%)	1943 ms
org.eclipse.persistence.queries.DatabaseQuery.clone ()		1918 ms (0.4%)	1918 ms
org.apache.catalina.core.StandardPipeline.getBasic ()		1871 ms (0.3%)	1871 ms
org.jboss.weld.util.collections.Arrays2.copyOf ()		1776 ms (0.3%)	1776 ms
org.eclipse.persistence.internal.sessions.UnitOfWorkImpl.cloneMap ()		1564 ms (0.3%)	1564 ms
org.apache.catalina.core.StandardWrapper.allocate ()		1509 ms (0.3%)	1509 ms
$Proxy285.allPredictions ()		1378 ms (0.3%)	1378 ms
org.apache.derby.client.am.ResultSet.getTime ()		1321 ms (0.2%)	1321 ms

actual EJB invocation...

Figure 6. CPU Sampler with System.out.println

Label	# Samples	Average	Min	Max	Std. Dev.	Error %	Throughput	KB/sec	Avg. Bytes
Predictions	290190	3	0	60028	258.55	0.00%	1398.2/sec	146.14	107.0
TOTAL	290190	3	0	60028	258.55	0.00%	1398.2/sec	146.14	107.0

Figure 7. Performance after reducing the pool size

CPU samples

Snapshot Thread Dump

Hot Spots – Method	Self time [%]	Self time	Self time (CPU)
org.eclipse.persistence.sessions.JNDIConnector.connect ()		244018 ms (36.4%)	4463 ms
org.apache.derby.client.net.Reply.fill ()		126834 ms (18.9%)	126834 ms
maskedclasses.LinkedTransferQueue.awaitMatch ()		77974 ms (11.6%)	64153 ms
org.apache.catalina.core.StandardWrapper.service ()		51539 ms (7.7%)	51539 ms
org.apache.derby.client.net.Request.sendBytes ()		37897 ms (5.6%)	37897 ms
org.apache.catalina.connector.OutputBuffer.doFlush ()		13184 ms (2%)	13184 ms
org.glassfish.weld.ACLSingletonProvider$ACLSingleton.get ()		10831 ms (1.6%)	10735 ms
org.apache.catalina.connector.OutputBuffer.close ()		8955 ms (1.3%)	8955 ms
org.eclipse.persistence.internal.databaseaccess.DatabaseAccessor.clos		8310 ms (1.2%)	6551 ms
org.glassfish.weld.ACLSingletonProvider$ACLSingleton.isSet ()		5961 ms (0.9%)	5961 ms
org.apache.derby.client.am.Statement.<init> ()		5186 ms (0.8%)	5186 ms
org.eclipse.persistence.internal.helper.ConcurrencyManager.acquire ()		4921 ms (0.7%)	1885 ms
org.apache.derby.client.am.PreparedStatement.listenToUnitOfWork ()		4623 ms (0.7%)	4623 ms
$Proxy153.name ()		4491 ms (0.7%)	4491 ms
org.jboss.weld.util.collections.Arrays2.copyOf ()		4474 ms (0.7%)	4474 ms
org.eclipse.persistence.queries.DatabaseQuery.clone ()		4087 ms (0.6%)	4087 ms
org.eclipse.persistence.internal.sessions.UnitOfWorkImpl.cloneMap ()		3917 ms (0.6%)	3917 ms
org.apache.catalina.core.StandardWrapper.allocate ()		3146 ms (0.5%)	3146 ms

Figure 8. Sampler output with two connections in the pool

Instead of 1,700 transactions per second, the insertion of this simple print command has reduced our performance to about 800 transactions per second, as shown in **Figure 5**.

Let's take a look at the VisualVM Sampler output shown in **Figure 6**. More time is spent in `ThreadPrintStream.println()` than in the most expensive database operation.

The actual EJB 3.1 overhead is negligible. The call to `$Proxy285.allPredictions()` is in the very last position and orders of magnitude faster than a single `System.out.println`.

Having a reference measurement makes identification of potential bottlenecks easy. You should perform stress tests as often as possible and compare the results. Performing nightly stress tests from the very first iteration is desirable. You will get fresh results each morning so you can start fixing potential bottlenecks.

Causing More Trouble

Misconfigured application servers are a common cause of bottlenecks. GlassFish Server Open Source Edition 3.1 comes with reasonable settings, so we can reduce the maximum number of connections from the Derby pool to two connections to simulate a bottleneck. With five concurrent threads (users) and only two database connections, there should be some contention (see **Figure 7**).

The performance is still surprisingly good. We get 1.400 transactions per second with two connections. The max response time went up to 60 seconds, which correlates surprisingly well with the "Max Wait Time: 60000 ms" connection pool setting in GlassFish Server Open Source Edition 3.1. A hint in the log files also points to the problem, as shown in **Listing 2**.

Listing 2.

```
WARNING: RAR5117 : Failed to obtain/create connec-
tion from connection pool [ SamplePool ]. Reason :
```

```
com.sun.appserv.connectors.internal.api.Pooling-
Exception: In-use connections equal max-pool-size
and expired max-wait-time. Cannot allocate more
connections.

WARNING: RAR5114 : Error allocating connection :
[Error in allocating a connection. Cause: In-use
connections equal max-pool-size and expired max-
wait-time. Cannot allocate more connections.]
```

Also interesting is the Sampler view in VisualVM (see Figure 8).

The method `JNDIConnector.connect()` became the most expensive method. It even displaced the `Reply.fill()` method from its first rank.

The package `org.eclipse.persistence` is the JPA provider for GlassFish Server Open Source Edition 3.1, so it should give us a hint about the bottleneck's location. There is nothing wrong with the persistence layer; it only has to wait for a free connection. This contention is caused by the artificial limitation of having only two connections available for five virtual users.

A look at the `JNDIConnector.connect` method confirms our suspicion (see **Listing 3**). In the method `JNDIConnect.connect`, a connection is acquired from a `DataSource`. In the case of an empty pool, the method will block until either an in-use connection becomes free or the Max Wait Time is reached. The method can block up to 60 seconds with the GlassFish Server Open Source Edition default settings. This rarely happens with the default settings, because the server ships with a Maximum Pool Size of 32 database connections.

Listing 3.

```
public Connection connect(Properties properties)
   throws DatabaseException, ValidationException {
String user = properties.getProperty("user");
      DataSource dataSource = getDataSource();
      try{
        ...
      return dataSource.
          getConnection(user, password);
      } catch (SQLException exception) {
          throw DatabaseException.sqlException(
              exception, true);
    }
}
```

How to Get the Interesting Stuff

The combination of JMeter and VisualVM is useful for ad hoc measurements. In real-world projects, stress tests should be not only repeatable but also comparable. A history of results with visualization makes the resultant comparison and identification of hotspots easier.

VisualVM provides a good overview, but the really interesting monitoring information can be obtained only from an application server in a proprietary way. All major application servers provide extensive monitoring information via Java Management Extensions (JMX). GlassFish Server Open Source Edition 3.1 exposes its monitoring and management data through an easily accessible REST interface.

You can easily persist monitoring data with a simple Java EE 6 application. JPA 2, EJB 3.1, Contexts and Dependency Injection, and JAX-RS reduce the task to only three classes.

To activate the monitoring, open the GlassFish Admin Console by specifying the Admin Console URL (http://localhost:4848). Then select Server, select Monitor, and then select Configure Monitoring. Then select the HIGH level for all components. Alternatively, you can activate monitoring by using the `asadmin` command from the com-

mand line or by using the REST management interface.

Now, all the monitoring information is accessible from the following root URI: http://localhost:4848/monitoring/domain/server. The interface is self-explanatory. You can navigate through the components from a browser or from the command line.

The command `curl -H "Accept: application/json" http://localhost:4848/monitoring/domain/server/jvm/memory/usedheapsize-count` returns the current heap size formatted as a JSON object (see **Listing 4**).

Listing 4.

```
{"message":"",
 "command":"Monitoring Data",
 "exit_code":"SUCCESS",
 "extraProperties":{
   "entity":{
     "usedheapsize-count":{
       "count":217666480,
       "lastsampletime":1308569037982,
       "description":
         "Amount of used memory in bytes",
       "unit":"bytes",
       "name":"UsedHeapSize",
       "starttime":1308504654922}
   },
 "childResources":{}}
}.
```

The most interesting key is `usedheapsize-count`. It contains the amount of used memory in bytes, as described by the description tag. The good news is that the entire monitoring API relies on the same structure and can be accessed in a generic way.

Monitoring Java EE 6 with Java EE 6

Executing HTTP GET requests from the command line still does not solve the challenge. In order to be comparable, the data has to be persistently stored. A periodic snapshot between 1 and 30 seconds is good enough for smoke tests and stress tests.

It turns out that you can easily persist monitoring data with a simple Java EE 6 application. JPA 2, EJB 3.1, Contexts and Dependency Injection (CDI), and JAX-RS reduce the task to only three classes. The JPA 2 entity `Snapshot` holds the relevant monitoring data (see **Listing 5**).

Listing 5.

```
@XmlAccessorType(XmlAccessType.FIELD)
@XmlRootElement
@Entity
public class Snapshot {
    @Id
    @GeneratedValue
    private long id;
    @Temporal(TemporalType.TIME)
    private Date monitoringTime;
   //...field declarations omitted

    public Snapshot(long usedHeapSize,
       int threadCount, int totalErrors,
       int currentThreadBusy, int committedTX,
       int rolledBackTX, int queuedConnections) {
           this();
           this.usedHeapSize = usedHeapSize;
     //...
}

    public Snapshot() {
        this.monitoringTime = new Date();
    }
}
```

The entity `Snapshot` represents the interesting data, such as the number of busy threads, `queuedConnections`, errors, committed and rolled-back transactions, heap size, and the time stamp. You can extract such information from any other

application server through different channels and APIs.

Listing 6 shows how to access REST services with Jersey. The managed bean `DataProvider` uses the Jersey client to access the GlassFish Server Open Source Edition's REST interface and convert the JSON result into Java primitives. The `fetchData` method is the core functionality of `DataProvider`, and it returns the populated `Snapshot` entity.

Listing 6.

```
public class DataProvider {
    public static final String BASE_URL =
        "http://localhost:4848" +
        "/monitoring/domain/server/";
    public static final String HEAP_SIZE =
        "jvm/memory/usedheapsize-count";
    private Client client;

    public Snapshot fetchData(){
        try {
            long usedHeapSize = usedHeapSize();
            //… other assignments omitted
            return new Snapshot(
                usedHeapSize, threadCount,
                totalErrors, currentThreadBusy,
                committedTX, rolledBackTX,
                queuedConnections);
        } catch (JSONException e) {
            throw new IllegalStateException(
               "Cannot fetch monitoring" +
               "data because of: " + e);
        }
    }

    long usedHeapSize() throws JSONException{
        final String uri = BASE_URL + HEAP_SIZE;
        return getLong(uri,"usedheapsize-count");
    }

    //other accessors omitted

    long getLong(String uri,String name)
    throws JSONException{
        ClientResponse result =
            client.resource(uri)
            .accept(MediaType.APPLICATION_JSON)
            .get(ClientResponse.class);
        return getJSONObject(result,name)
            .getLong("count");
    }

    //accessors omitted

    JSONObject getJSONObject(
        ClientResponse result,String name)
        throws JSONException {
            JSONObject response =
                result.getEntity(JSONObject.class);
            return response.getJSONObject(
                "extraProperties")
                .getJSONObject("entity")
                .getJSONObject(name);
        }
    }
```

Every five seconds, the `MonitoringController` `@Singleton` EJB 3.1 bean shown in Listing 7 asks the `DataProvider` for a `Snapshot` and persists it. In addition, the persisted data is exposed through REST. You can access all snapshots using http://localhost:8080/stm/resources/snapshots, and you will get `Snapshot` instances as a JSON object (see Listing 8).

Listing 7.

```
@Singleton
@Path("snapshots")
@Produces(MediaType.APPLICATION_JSON)
public class MonitoringController {
```

```
@Inject
DataProvider dataProvider;
@PersistenceContext
EntityManager em;
@Schedule(minute="*",second="*/5",hour="*")
public void gatherAndPersist(){
    Snapshot current =
        dataProvider.fetchData();
    em.persist(current);
}
@GET
public List<Snapshot> all(){
    CriteriaBuilder cb =
        this.em.getCriteriaBuilder();
    CriteriaQuery q =
        cb.createQuery();
    CriteriaQuery<Snapshot> select =
        q.select(q.from(Snapshot.class));
    return this.em.createQuery(select)
                  .getResultList();
    }
}
```

Interestingly, a Java EE 6 solution is significantly leaner than a comparable Plain Old Java Object (POJO) implementation. Periodic timer execution, transactions, and `EntityManager` bookkeeping are provided out of the box in Java EE 6 but must be implemented in Java SE.

Listing 8.

```
{"snapshot":
  [{"id":"1",
    "monitoringTime":"1970-01-01T10:50:30+01:00",
    "usedHeapSize":"158408536",
    "threadCount":"131",
    "totalErrors":"4",
    "currentThreadBusy":"-1",
    "committedTX":"23",
    "rolledBackTX":"2-",
    "queuedConnections":"0"},
   {"id":"2",
    "monitoringTime":"1970-01-01T10:50:35+01:00",
    "usedHeapSize":"160421672",
    "threadCount":"131",
    "totalErrors":"4",
    "currentThreadBusy":"-1",
    "committedTX":"23",
    "rolledBackTX":"2",
    "queuedConnections":"0"
   }]
}.
```

Automating the Stress Test

Using the StressTestMonitoring (STM) application, we can collect application server monitoring data systematically and persistently, and we can analyze and compare the stress test results after each run. This approach is not perfect, though, because both the stress test and the load generator must be started and stopped manually.

Java EE 6+ applications are always executed concurrently. Even with unrealistic stress tests, you will learn a lot about system behavior and identify some bottlenecks or concurrency bugs.

A continuous integration (CI) tool, such as Jenkins or Hudson, is capable of automating the whole lifecycle and can easily deploy STM and start the load generator automatically.

Hudson and Jenkins support a periodic build execution (for example, with a single configuration tag: `@midnight`). With both products, setting up nightly executed stress tests takes only minutes. You need to deploy the STM application first and launch the stress test generator afterward. For a GlassFish Server Open Source Edition deployment, this requires just a single line:

```
asadmin deploy --force […]/StressTestMonitor.war.
```

JMeter can also be started without the GUI in headless mode. It requires another line: `jmeter -n -t predictions.jmx -l predictions.jtl`. The parameter `-n` prevents the GUI from appearing, the parameter `-t` specifies the configuration file (created with JMeter in GUI mode), and `-l` specifies the log file, which can be analyzed with JMeter after the test.

A Hudson or Jenkins "Free Style Software Project" build with two build steps (Execute Windows Shell Command or Execute Shell) does the job: Deploy STM and Execute Stress Tests. You don't need to create a shell script or a batch file. The commands can be executed directly by Hudson or Jenkins.

VisualVM can still be used to monitor the application in real time or identify the hot spots. All the relevant monitoring data is persisted in a database table and can be analyzed easily after the test.

Because of JSR 77 (the management and monitoring API), all application servers also provide access to the statistics of all deployed Java EE components.

Nice-to-Haves

The solution described so far is good enough for getting started. The interesting parameters, such as JVM and application server monitoring data, are gathered and persisted automatically during a nightly job. Because of JSR 77 (the management and monitoring API), all application servers also provide access to the statistics of all deployed Java EE components. Application-specific EJB beans (usually the facade to your business logic) can be monitored using exactly the same mechanism.

You will get the number of concurrent requests, the current number of active instances (and, thus, the number of concurrent transactions accessing the bean), and business method runtime statistics, such as slowest and average execution times.

For example, `PredictionAudit` EJB bean pool data is accessible under http://localhost:4848/monitoring/domain/server/applications/com.abien_TestingEJBAndCDI.war_1.0-SNAPSHOT/PredictionAudit/bean-pool.

All the monitoring data is stored in a single table, which makes the data available to tools such as JasperReports and Eclipse BIRT. Charts and reports are only a few clicks away. The Snapshot entity is exposed through JSON, which makes it "consumable" by all JavaScript and JavaFX applications as well.

Furthermore, the `Snapshot` entity is a Java class and can contain additional validation or processing logic. It is trivial to escalate "suspicious" `Snapshot` instances with CDI events, as shown in **Listing 9**.

Listing 9.

```
@Inject
Event<Snapshot> escalationSink;
@Schedule(minute="*",second="*/5",hour="*")
public void gatherAndPersist(){
   Snapshot current = dataProvider.fetchData();
   em.persist(current);
   if(current.isSuspicious())
       escalationSink.fire(current);
}
```

A suspicious `Snapshot` listener will only have to implement a method with an annotated parameter to receive the event `public void onSuspiciousEvent(@Observes Snapshot snapshot){}`.

All suspicious events can be analyzed in real time, sent using e-mail, or just aggregated and exposed with JMX.

Conclusion: No Excuses

Java EE 6+ applications are always executed concurrently. Even with unrealistic stress tests, you will learn a lot about system behavior and identify some bottlenecks or concurrency bugs.

The earlier and more frequently stress tests are executed, the more you will learn about the application's runtime behavior. Application server configuration problems, potential bottlenecks, synchronization bugs, and memory leaks can be also identified during stress tests. `</article>`

[This article first appeared in this magazine in the November/December 2011 issue and has since been updated. —*Ed.*]

LEARN MORE

- *Real World Java EE Night Hacks: Dissecting the Business Tier*
- Sample Java EE testing code on Project Kenai web page
- StressTestMonitoring application
- "Unit Testing for Java EE"
- JSR 77

MARK HRYNCZAK

BIO

Think Like a Tester and Get Rid of QA

Atlassian's developers are required to formulate tests before they code and after. QA is there to help—but not test.

One of the best practices of modern software development is to actively involve developers in software testing. This practice mirrors the trend driven by DevOps of involving developers intimately with operations. However, some organizations, such as the one I work in, are making developers take on almost all the testing—a move that has proven successful and redefined the role of QA. I work at Atlassian, and here *QA* stands for quality *assistance* rather than quality *assurance*. The QA team is not able to test everything and does not aim to. Testing activities are mainly carried out by developers, while QA engineers focus on teaching developers the skills to effectively and efficiently test software.

In this article, I discuss the techniques we use to train developers, and how you can start thinking like a tester, too.

Testing from Day 1

New developers joining our company are put through "boot camp"—a series of classes that get them up to speed as quickly as possible. Members of the QA team run a class entitled "Exploratory Testing Workshop for Developers." The content is aimed at new hires, who might not have been expected to perform their own testing previously. More-experienced developers are always welcome to use the class as a refresher. The goal is to teach developers how to use manual testing effectively and deliver high-quality features.

We start the workshop by explaining that the whole team is responsible for quality. For developers, owning quality means that they should think about error cases, security vulnerabilities, and nonstandard user scenarios before they start coding. We explain why good automation is important but insufficient, and promote exploratory testing as a technique to find problems efficiently.

Some developers have a preconception that exploratory testing means manual scripted testing, which is tedious, low-value, and as a rule should never be done by a human (because it can be automated).

We also steer developers away from what we call "ad hoc testing"—that is, mindlessly clicking around the UI in the hope that bugs will just appear. The approach we recommend is anything but mindless. Exploratory testing involves thinking critically about the feature you are testing, and finding problems by identifying specific valid risks.

The second half of the workshop invites attendees to give exploratory testing a try on a real feature, with QA guidance to avoid the mindless-clicking-around trap. Generally, we need to give hints like these:

- Don't test the "happy path." Instead, think about what *might* not work and what *should* not work (such as users with and without expected permissions, or conflicting concurrent actions on a multiuser system).

- Use challenging data input, and avoid relying on provided defaults and demo content (such as overly long text, characters from non-English alphabets, or XSS strings).
- Think about the implementation choices that were made, and the risks they imply (such as inefficient queries on massive data sets, or Ajax requests on an expired session).
- Imagine ways users might ask the feature to do something unexpected (such as spoofing other users, or using REST API endpoints to access content they should not access).
- Remember that a UI isn't necessary for exploratory testing (you can find problems by theorizing without touching your computer—even before code has been written).

Initially, developers often do a poor job of testing due to inexperience. But as exploratory testing becomes part of their routine, they learn to think from a new perspective and identifying risks becomes second nature.

Story-Level Testing

Known internally as *Developer on Test* (DoT), we set up the team's workflow so that when one developer finishes implementing a story, another developer, the DoT, verifies that it meets the team's quality requirements. The DoT is the gatekeeper for that story—once the gatekeeper is satisfied, the code goes into production.

When starting off with a new team, the process of DoT validation is a good way to introduce developers to the idea that they can—and must—be able to test. We might start them on simple stories, leaving the more-complex ones to QA engineers. Or we might provide QA pairing sessions to get them up to speed. In both cases, the long-term goal is to get the team to a level where both skill and confidence in testing can enable the original developers to reliably test their own work. Use of a DoT is an interim step toward this end goal, because having two people performing testing on a single story is ultimately inefficient.

> We are careful not to simply produce checklists that can be mindlessly followed. **The aim is to get the developers to think like testers.**

When developers take on a DoT role for the first time, they need to shift away from a code-centric view of a story and take on an end-user mindset. So we give them some tips:

Feature exploration.

- Set a time limit to explore most of the functionality relevant to the story.
- Focus on exploration and familiarization.

Quick attacks.

- Use known problematic input wherever data input is required or data is displayed.
- Check the well-known problematic cases whenever there are commonly used elements such as lists, trees, sessions, or permissions.
- Establish a time limit for the activity to make sure you have enough time left for more-diligent attempts to test the feature.

20 percent use case.

- Think about what could go wrong and affect the feature's functionality.
- Consider user, product, and environment.
- Avoid the happy path.

Heuristics.

- Do some modeling and analysis to generate ideas that are not immediately obvious for what and how to test. Heuristics will help you.
- Identify any values that can change in a feature and the interaction between them.

QA can provide resources to help a DoT with the above by providing a catalog of known problematic input or a list of proven valuable heuristics, for example. But we are careful not to simply produce checklists that can be mindlessly followed. The aim is to get the developers to think like testers,

after all. This can be presented as a challenge: Assume that problems exist, and keep searching until you find them.

We measure the effectiveness of the team's quality process by two important metrics: the ratio of successful stories (those that did not result in a shipped bug) and the ratio of rejected stories (those where the DoT found an issue before shipping) to total stories.

As the team matures, developers no longer **depend on the QA engineer being available for every discussion.**

Obviously, we want the first metric to be trending toward 100 percent. If we are shipping problems to production, then the DoT has not done a good job. QA's focus is on improving that with more workshops, more pairing, and so on.

Once the team is reaching the quality bar on nearly all stories, the second metric is of particular interest. The DoT adds value by rejecting stories that aren't up to par, but rejections are an inefficiency in the process. We want the rejection rate to trend toward zero. Ultimately, we want to remove the DoT step and have the original developer deliver high quality on the first go, not after someone else has pointed out problems.

Precoding Test Thinking

As the team matures, we focus more on prevention rather than detection. This means removing the DoT step—often against the wishes of the team, who come to see it as a safety net. We replace the postcoding DoT step by a precoding step we call a *QA kickoff*: the developers talk a QA engineer through the story, explaining what they plan to implement. Together the developer and QA engineer brainstorm test ideas, possible risks, and anything else related to the story.

Typically, the QA engineer asks lots of questions, such as, "What happens when you do X? What if we move Y to Z and then back again?" This discussion adds huge value. When developers are aware of edge cases and potential problems, they can code and test to mitigate them rather than reworking to fix them.

The outcome of the kickoff is a set of testing notes, which developers can use to determine when the story is done. Once they are confident, the story can be shipped. *Voilà!* We have removed the need for an additional testing step.

The format of kickoffs and testing notes varies according to the context, but here are some common elements:

Specific risks. What are the critical areas that must work? Which are the parts most likely *not* to work? What needs to be tested thoroughly?

Test ideas. What user scenarios should be covered? What should be automated? What level of automation is appropriate? What needs to be checked manually?

Notes. A kickoff often raises a lot of questions and details that do not form coherent test ideas but still are good to jot down at this point—for example, in-product analytics, feature discoverability and accessibility, and so on.

Again, we use metrics to validate that the kickoff process is achieving the quality goals of the team. As the team matures, developers can take on both roles in a QA kickoff—they no longer depend on the QA engineer being available for every discussion on every story.

The Big Picture

As a team succeeds with the processes described previously, the developers take on more of the testing activities, while the QA engineer takes on less. You might wonder what QA can do with all that free time. A high-performing team of developers means that QA can focus on big-picture quality improvements. Here are some examples:

Scale. We can increase the number of developers for whom a single QA engineer can provide value. At Atlassian, for example, the ratio is often between 1:10 and 1:20.

Knowledge. We can look ahead to see how the team can

meet customer needs better, and teach that to the whole team through the kickoffs.

Innovation. We can experiment with process changes and measure their impact.

Tools. We can identify ways to make testing easier for developers. For example, the Atlassian QA team built an Amazon EC2 service that provides quick and easy access to any supported browser for testing and troubleshooting.

Results

We've had great results with these techniques. They have enabled the QA function to scale effectively with a rapidly growing development team; individual QA engineers have successfully innovated and multiplied their value; and we have rendered obsolete much of the tedious busy work that is accepted as a necessary evil of the traditional QA role.

But perhaps most importantly, the team of (often skeptical) developers has enthusiastically accepted these changes. We've given them confidence that they can test their own work, deliver it to a measurable high standard, and make efficiency savings that increase the team's velocity without sacrificing quality. Internal surveys about the process and the team consistently show this.

If you are a developer who currently relies on a traditional testing phase to catch the bugs that you produce, you should question whether this is a satisfactory state of affairs. Which statement matches your mindset? "I'm a good developer; I don't need to test" or "Developers are not good developers unless they can also test." `</article>`

LEARN MORE

- Further details on the Atlassian process (by the same author)
- "The Day the QA Died" (another view of the same transition)

THE VIRTUAL JUG

The Virtual JUG (vJUG) is an online-only Java user group with a global audience in more than 100 countries. It is now 18 months old and already has more than 3,500 members. To date, it has hosted 39 online sessions and runs at least two sessions every month, as well as a monthly podcast.

The most popular live sessions have included one by Java creator James Gosling on his new Wave Glider project, and a deep tech session on the Java memory model with Oracle's Aleksey Shipilëv. The most watched session in the vJUG's short history is a look at 55 new features in Java 8 by Simon Ritter, who heads Java Technology Evangelism at Oracle. Sessions can be watched live or in replay at http://virtualjug.com. All of our speakers are interviewed after their sessions to talk about what makes them tick and to answer many probing and sometimes uncomfortable questions.

The vJUG has created a couple of new initiatives in 2015, including a new podcast called the Java Council. Its goal is to fill the gap left by the Java Posse, which ended in 2014.

Another new initiative this year is the vJUG book club, which coordinates the worldwide reading of a chosen book, a review, and a discussion over several vJUG sessions. The subject of the book could be anything from a deep look at technology to time management. The first book will be *Effective Java, Second Edition* by Joshua Bloch. Which book will be next? The great thing is the community will decide! If you want to be part of a unique online JUG, visit us.

HADI HARIRI

BIO

Kotlin: A Low-Ceremony, High-Integration Language

Work with a statically typed, low-ceremony language that provides first-class functions, nullability protections, and complete integration with existing Java libraries.

It has been more than five years since Project Kotlin, an open source language targeting the JVM, was announced by JetBrains. Since then, much progress has been made, some language designs have changed, a new platform—namely JavaScript—is now supported, and even the project name has changed. It's now simply known as Kotlin. But if there's one thing that still remains, it is the initial goals and intent for why Kotlin was developed.

At JetBrains, we have been developing IDEs for many programming languages, and yet despite this, much of our code continues to be written in Java. In 2009, we were looking for an alternative to Java, something that could reduce the size of our codebase by being more concise and, at the same time, offer features that we felt could provide important benefits. We needed a language that was also syntactically similar enough to Java that ramp-up time would not be substantial. The main contender at the time was Scala, but the compiler performance wasn't that great and providing tooling for Scala was (and continues to be) quite challenging. Tooling and performance were important aspects in our decision, so we opted not to go with Scala. Thus was born Project Kotlin.

One of Kotlin's goals is to reduce the amount of somewhat pointless code yet maintain readability and functionality.

Shortly afterward, news about the JVM language Ceylon was announced; and at one point we considered joining efforts. However, at the time, Ceylon's focus on interoperability was somewhat of a low priority. Given that we have a large codebase in Java, being able to use and extend it was of considerable importance to us. Thus, we decided to continue down our own path with Kotlin.

Removing the Pain

Five years later and close to reaching the 1.0 milestone, Kotlin remains true to its initial goals of being concise, safe, interoperable, "toolable," and performant. Many, if not all, of these goals are for a single purpose: removing some of the pain and the errors we encounter when writing code.

The IntelliJ platform is an extremely large codebase on which all our IDEs, including IntelliJ IDEA, are built. The open source Community Edition alone, which is hosted on GitHub, has millions of lines of code. While there is a lot of functionality, much of the code is often boilerplate code that is necessary because, well, because it's Java. One of Kotlin's goals has been to reduce the amount of somewhat pointless code yet maintain readability and functionality. For instance, a typical Java bean with property getters, setters, `toString`,

and equality in Kotlin can be reduced to the following:

```
data class Customer(
    var name: String, var email: String)
```

Things such as smart-casting remove the need for verbosity by delegating the work to the compiler. For example, when checking an immutable value for a specific type, it's no longer necessary to cast to that type when operating on it:

```
fun convert(obj: Shape) {
    if (obj is Circle) {
        val radius = obj.radius()
          ...
    }
}
```

Kotlin also has support for named objects, which, in essence, means a singleton would simply be written as follows:

```
val MySingleton = object {
    val numberOfDays = 10
}
```

Another pain point Kotlin addresses is the need to use functional constructs—such as lambda expressions and higher-order functions—and to treat functions as first-class citizens. While Java 8 addresses some of these concerns, our goal was and continues to be to provide this functionality when using Java 6, 7, or 8—thus, even allowing support for these features on the Android platform. This is one reason that Kotlin has enjoyed considerable popularity in the Android development community.

Functions can be top-level in Kotlin, much like they are in JavaScript, meaning there's no need to attach a function to an object. As such, we could simply declare a function in a file like this:

```
fun toSentenceCase(input: String) {
    . . .
}
```

Much like C#, Kotlin also allows extension functions, meaning a type (either of Java or Kotlin) can be extended with new functionality simply by suffixing the type. Taking the previous example, if I want the `String` type to have `toSentenceCase()`, I can simply write the following:

```
fun String.toSentenceCase() {
   ... // 'this' would hold an
       //    instance of the object
}
```

To work efficiently with functions as primitives, there needs to be support for higher-order functions—that is, functions that take functions as parameters or return functions. With Kotlin, this is possible, for example:

```
fun operate(x: Int, y: Int,
            operation: (Int, Int) -> Int) {
   ...
}
```

This code declares a function that takes three parameters: two integers and a third parameter that is a function that, in turn, takes two integers and returns an integer. We can then invoke functions as follows:

```
fun sum(x: Int, y: Int) {
   . . .
}

operate(2, 3, ::sum)
```

This code shows a function, `sum`, being defined and passed as a parameter to `operate`. It shows that Kotlin supports ref-

erencing functions by name. Of course, a lambda expression can be passed in as well:

```
operate(2, 3, { x, y -> x +y })
```

These capabilities deliver an elegant way of doing function pipelining:

```
val numbers = 1..100

numbers.filter { it % 2 == 0 }
       .map { it + 5 }
       .forEach {
            println(it)
       }
```

One more issue we attack with Kotlin is null pointer exceptions. In Kotlin, by default, things cannot be null, meaning that potentially the only way we'd get a null reference exception would be if we explicitly force it.

```
var city = "London"
```

In the code above, `city` could never be assigned a null value. If we want it to be null, we need to go out of our way to be explicit:

```
var city : String? = null
```

where `?` indicates that a type can be nullable. When interoperating with Java, we provide certain mechanisms to warn of possible null references, as well as providing some operators to make the code more concise, such as the *safe call operator*:

```
var file = File("...")
file?.length()
```

Because of the `?.`, this code would invoke `length()` on `file` only if `file` were not null. The standard library, a small runtime that ships with Kotlin, also provides additional functions in this area, such as the `let` function, which when combined with the safe call operator allows for succinct code, such as the following:

```
obj?.let {
     ... // execute code here
}
```

This results in the code block executing if the object is not null.

One last thing worth mentioning about Kotlin is its ability to easily enable the creation of DSLs—without the overhead that necessarily comes with maintaining them or the language knowledge required to implement them. Top-level functions, higher-order functions, extension functions, and a few conventions, such as not having to use brackets when the last parameter to a function is another function—these features allow for creating rich DSLs that are strongly typed. The quintessential example is that of type-safe Groovy-style builders. The following function generates the expected HTML output:

Kotlin has the ability to easily create DSLs—without the overhead that necessarily comes with maintaining them or the language knowledge required to implement them.

```
html {
    head {
      title {+"XML encoding with Kotlin"}
    }
    body {
      h1 {+"XML encoding with Kotlin"}
```

```
      p  {+"this is an alternative markup to XML"}
    }
}
```

Growth and the Road Ahead

The previous code snippets are just a few examples of what Kotlin provides. It wouldn't be possible to cover all the little details and conveniences Kotlin offers in a single article.

Over the past year, and despite not having released version 1.0, we've noticed a substantial growth and interest in Kotlin. There has been an increase in downloads and visits to the Kotlin site, as well as an increase in technical questions in both our forums and public venues such as StackOverflow.

While a lot of this interest is due to the Android community and our contributions in terms of additional tooling and frameworks for this space, there's also interest in other more generic areas such as web development, both client-side (given Kotlin's ability to compile to JavaScript) and server-side.

We're close to reaching the first major release, and we've made significant steps toward that. Over the past few milestone releases, we've been removing and adjusting some things in the language to make sure that once we release, we'll be fairly certain that what we ship is there to stay. As any language designer or developer knows, whatever goes in a language stays as baggage pretty much forever.

Some of our newer tools are being written in Kotlin and our existing tools, such as IntelliJ IDEA and YouTrack, are adopting Kotlin.

As a company that provides tooling for developers, we've tried to make language release transitions as smooth as possible. To this end, a new release usually comes with a compiler warning about a potential upcoming change or deprecation. The IDE also provides a quick fix to easily migrate code to newer syntax. We believe that this way, we create a smooth experience for developers that are already using Kotlin in production.

Beyond these quick fixes, we're also focusing on improving other aspects of tooling. For Kotlin to be successful, the entry barrier should be low in all aspects. That is why we not only provide tooling for IntelliJ IDEA, in both Ultimate and the open source Community Edition, but also for build tools such as Gradle, Ant, and Maven, as well as a simple command-line compiler. We've also released a preliminary version of Kotlin for Eclipse, and we're hoping that much like there are contributions to Kotlin in other areas, the community will contribute to Eclipse support as well.

Conclusion

In conclusion, we developed Kotlin for our own use primarily and are heavily invested in it. For us, it's a tool that we're using to drive our own business, which is developer tools. We already have several internal and public-facing web applications written in Kotlin. Some of our newer tools are being written in Kotlin and our existing tools, such as IntelliJ IDEA and YouTrack, are adopting Kotlin. `</article>`

[This article is the inaugural installment of a new series on JVM languages that will appear in *Java Magazine*. We will examine the full range of languages, from large commercial efforts to projects driven by determined groups of hackers. In the next issue, we'll cover Jython. —*Ed.*]

LEARN MORE

- Kotlin home
- Kotlin on StackOverflow
- Kotlin on Reddit

VENKAT
SUBRAMANIAM

BIO

Part 2

Functional Programming in Java: Using Collections

Removing code smells from lambda-based functional routines

In the first article in this two-part series, I demonstrated how lambda expressions harness the power of the functional style of programming in Java. Lambdas create more-expressive and concise code with less mutability and fewer errors. In this final part, I explore this further and consider a cautionary warning. As we'll see, lambda expressions are deceivingly concise, and it's easy to carelessly duplicate them in code. Duplicate code leads to poor-quality code that's hard to maintain; if we needed to make a change, we'd have to find and touch the relevant code in several places.

Avoiding duplication can also help improve performance. By keeping the code related to a piece of knowledge concentrated in one place, we can easily study its performance profile and make changes in one place to get better performance.

Reusing Lambda Expressions

Now, let's see how easy it is to fall into the duplication trap when using lambda expressions, and consider ways to avoid it. Suppose we have a few collections of names: `friends`, `editors`, and `comrades`:

```
final List<String> friends =
   Arrays.asList("Brian", "Nate", "Neal",
                 "Raju", "Sara", "Scott");

final List<String> editors =
    Arrays.asList("Brian", "Jackie",
                  "John", "Mike");

final List<String> comrades =
    Arrays.asList("Kate", "Ken", "Nick",
                  "Paula", "Zach");
```

Suppose we want to filter out names that start with a certain letter. We will first take a naive approach to this using the `filter()` method:

```
final long countFriendsStartN =
   friends.stream()
       .filter(name -> name.startsWith("N"))
       .count();

final long countEditorsStartN =
    editors.stream()
        .filter(name -> name.startsWith("N"))
        .count();

final long countComradesStartN =
    comrades.stream()
        .filter(name -> name.startsWith("N"))
        .count();
```

The lambda expressions made the code concise, but it quietly led to duplicate code. In the previous example, one

change to the lambda expression requires changes in more than one place—that's a no-no. Fortunately, we can assign lambda expressions to variables and reuse them, just like with objects.

The `filter()` method, the receiver of the lambda expression in the previous example, takes a reference to a `java.util.function.Predicate` functional interface. Here, the Java compiler works its magic to synthesize an implementation of the `Predicate`'s `test()` method from the given lambda expression. Rather than asking Java to synthesize the method at the argument-definition location, we can be more explicit. In this example, it's possible to store the lambda expression in an explicit reference of type `Predicate` and then pass it to the function; this is an easy way to remove the duplication.

Let's refactor the previous code to make it adhere to the DRY (Don't Repeat Yourself) best practice.

```
final Predicate<String> startsWithN =
    name -> name.startsWith("N");

final long countFriendsStartN =
    friends.stream()
        .filter(startsWithN)
        .count();
final long countEditorsStartN =
    editors.stream()
        .filter(startsWithN)
        .count();
final long countComradesStartN =
    comrades.stream()
        .filter(startsWithN)
        .count();
```

Rather than duplicate the lambda expression several times, we created it once and stored it in a reference named `startsWithN` of type `Predicate`. In the three calls to the `filter` method, the Java compiler happily took the lambda expression stored in the variable under the guise of the `Predicate` instance.

Our examples illustrate how to pass functions to functions, create functions within functions, and return functions from within functions.

The new variable gently removed the duplication that sneaked in. Unfortunately, it's about to sneak back in with a vengeance, as we'll see next, and we need something a bit more powerful to thwart it.

Using Lexical Scoping and Closures

There's a misconception among some developers that using lambda expressions might introduce duplication and lower code quality. Contrary to that belief, even when the code gets more complicated, we still don't need to compromise code quality to enjoy the conciseness that lambda expressions give, as we'll see in this section.

We managed to reuse the lambda expression in the previous example; however, duplication will sneak in quickly when we bring in another letter to match. Let's explore the problem further and then solve it using lexical scoping and closures.

Duplication in lambda expressions. Let's pick the names that start with *N* or *B* from the `friends` collection of names. Continuing with the previous example, we might be tempted to write something like the following:

```
final Predicate<String> startsWithN =
    name -> name.startsWith("N");
final Predicate<String> startsWithB =
    name -> name.startsWith("B");

final long countFriendsStartN =
   friends.stream()
          .filter(startsWithN)
          .count();
final long countFriendsStartB =
   friends.stream()
          .filter(startsWithB)
          .count();
```

The first predicate tests whether the name starts with an *N* and the second tests for a *B*. We pass these two instances to the two calls to the `filter()` method, respectively. Seems reasonable, but the two predi-

cates are mere duplicates, with only the letter they use being different. Let's figure out a way to eliminate this duplication.

Removing duplication using lexical scoping. As a first option, we could extract the letter as a parameter to a function and pass the function as an argument to the `filter()` method. That's a reasonable idea, but the `filter()` method will not accept some arbitrary function. It insists on receiving a function that accepts one parameter representing the context element in the collection, and returning a `boolean` result. It's expecting a `Predicate`.

For comparison purposes, we need a variable that will cache the letter for later use and hold onto it until the parameter, `name` in this example, is received. Let's create a function for that:

```
public static Predicate<String>
    checkIfStartsWith(final String letter) {
    return name -> name.startsWith(letter);
}
```

We defined `checkIfStartsWith()` as a `static` function that takes a `letter` of type `String` as a parameter. It then returns a `Predicate` that can be passed to the `filter()` method for later evaluation. `checkIfStartsWith()` returns a function as a result.

The `Predicate` that `checkIfStartsWith()` returned is different from the lambda expressions we've seen so far. In `return name -> name.startsWith(letter)`, it's clear what `name` is: it's the parameter passed to this lambda expression. But what's the variable `letter` bound to? Because that's not in the scope of this anonymous function, Java reaches over to the scope of the definition of this lambda expression and finds the variable `letter` in that scope. This is called *lexical scoping*. Lexical scoping is a powerful technique that lets us cache values provided in one context for use later in another context. Since this lambda expression *closes over* the scope of its definition, it's also referred to as a *closure*.

It's worth noting here that there are a few restrictions to lexical scoping. For one thing, from within a lambda expression, we can access only local variables that are `final` or effectively `final` in the enclosing scope. A lambda expression may be invoked right away, or it may be invoked lazily or from multiple threads. To avoid race conditions, the local variables we access in the enclosing scope are not allowed to change once they are initialized. Any attempt to change them will result in a compilation error. Variables marked `final` directly fit this bill, but Java does not insist that we mark them as such. Instead, Java looks for two things. First, the accessed variables have to be initialized within the enclosing methods before the lambda expression is defined. Second, the values of these variables don't change anywhere else—that is, they're effectively `final` although they are not marked as such.

When using lambda expressions that capture local state, we should also be aware that stateless lambda expressions are runtime constants, but those that capture local state have an additional evaluation cost.

With these restrictions in mind, let's see how to use the lambda expression returned by `checkIfStartsWith()` in the calls to the `filter()` method.

```
final long countFriendsStartN =
    friends.stream()
           .filter(checkIfStartsWith("N"))
           .count();
final long countFriendsStartB =
    friends.stream()
           .filter(checkIfStartsWith("B"))
           .count();
```

In the calls to the `filter()` method, we first invoke the `checkIfStartsWith()` method, passing in a desired letter. This call immediately returns a lambda expression that is then passed on to the `filter()` method.

By creating a higher-order function, `checkIfStartsWith()` in this example, and by using lexical scoping, we managed to remove the duplication in code. We did not have to repeat the comparison to check whether the name starts with different letters.

Refactoring to narrow the scope. In the preceding (smelly) example, we used a `static` method, but we don't want to pollute the class with `static` methods to cache each variable in the future. It would be nice to narrow the function's scope to where it's needed. We can accomplish

that by using a `Function` interface.

```
final Function<String, Predicate<String>>
    startsWithLetter = (String letter) -> {
        Predicate<String> checkStarts =
            (String name) ->
                name.startsWith(letter);
    return checkStarts;
};
```

This lambda expression replaces the `static` method `checkIfStartsWith()` and can appear within a function, just before it's needed. The `checkIfStartsWith` variable refers to a `Function` that takes in a `String` and returns a `Predicate`.

This version is verbose compared to the `static` method we saw earlier, but we'll refactor that soon to make it concise. For all practical purposes, this function is equivalent to the `static` method; it takes a `String` and returns a `Predicate`. Instead of explicitly creating the instance of the `Predicate` and returning it, we can replace it with a lambda expression:

```
final Function<String, Predicate<String>>
     startsWithLetter =
         (String letter) -> (String name) ->
         name.startsWith(letter);
```

We reduced clutter, but we can take the conciseness up another notch by removing the types and letting the Java compiler infer the types based on the context. Let's look at the concise version.

```
final Function<String, Predicate<String>>
   startsWithLetter =
       letter -> name -> name.startsWith(letter);
```

It takes a bit of effort to get used to this concise syntax. Feel free to look away for a moment if this makes you cross-eyed. Now that we've refactored that version, we can use it in place of `checkIfStartsWith()`, like so:

```
final long countFriendsStartN =
   friends.stream()
          .filter(startsWithLetter.apply("N"))
          .count();
final long countFriendsStartB =
   friends.stream()
          .filter(startsWithLetter.apply("B"))
          .count();
```

We've come full circle with higher-order functions in this section. Our examples illustrate how to pass functions to functions, create functions within functions, and return functions from within functions. They also demonstrate the conciseness and reusability that lambda expressions facilitate.

We made good use of both `Function` and `Predicate` in this section, but let's discuss how they're different. A `Predicate<T>` takes in one parameter of type `T` and returns a `boolean` result to indicate a decision for whatever check it represents. We can use it anytime we want to make a go or no-go decision for a candidate we pass to the predicate. Methods such as `filter()` that evaluate candidate elements take in a `Predicate` as their parameter.

On the other hand, a `Function <T, R>` represents a function that takes a parameter of type `T` and returns a result of type `R`. This is more general than a `Predicate` that always returns a `boolean`. We can employ a `Function` anywhere we want to transform an input to another value, so it's quite logical that the `map()` method uses `Function` as its parameter.

Selecting elements from a collection was easy. Next, we'll cover how to pick just one element out of a collection.

Picking an Element

It's reasonable to expect that picking one element from a collection would be simpler than picking multiple elements. But there are a few complications. Let's look at the complexity introduced by the habitual approach and then bring in lambda expressions to solve it.

Let's create a method that looks for an element that starts with a given letter and prints it.

```
public static void pickName(
    final List<String> names,
    final String startingLetter) {
        String foundName = null;
        for(String name : names) {
            if(name.startsWith(startingLetter)) {
                foundName = name;
                break;
            }
         }
        System.out.print(
            String.format(
               "A name starting with %s: ",
               startingLetter));

        if(foundName != null) {
            System.out.println(foundName);
        } else {
            System.out.println("No name found");
        }
}
```

This method's smell can easily compete with passing garbage trucks. We first created a `foundName` variable and initialized it to `null`—that's the source of our first bad smell. This will force a `null` check, and if we forget to deal with it, the result could be a `NullPointerException` or an unpleasant response. We then used an external iterator to loop through the elements, but had to break out of the loop if we found an element—here are other sources of rancid smells: primitive obsession, imperative style, and mutability. Once out of the loop, we had to check the response and print the appropriate result. That's quite a bit of code for a simple task.

Let's rethink the problem. We simply want to pick the first matching element and safely deal with the absence of such an element. Let's rewrite the `pickName()` method, this time using lambda expressions.

```
public static void pickName(
    final List<String> names,
    final String startingLetter) {
        final Optional<String> foundName =
        names.stream()
             .filter(name ->
                  name.startsWith(startingLetter))
             .findFirst();
        System.out.println(
            String.format(
                "A name starting with %s: %s",
                startingLetter,
                foundName.orElse("No name found")));
}
```

Some powerful features in the JDK library came together to help achieve this conciseness. First we used the `filter()` method to grab all the elements matching the desired pattern. Then the `findFirst()` method of the `Stream` class helped pick the first value from that collection. This method returns a special `Optional` object, which is the state-appointed `null` deodorizer in Java.

> Collections are common in programming and, thanks to lambda expressions, using them is now easier and simpler in Java. **We can trade the long-winded old methods for elegant, concise code** to perform common operations on collections.

The `Optional` class is useful whenever the result may be absent. It protects us from getting a `NullPointerException` by accident, and makes it quite explicit to the reader that "no result found" is a possible outcome. We can inquire whether an object is present by using the `isPresent()` method, and we can obtain the current value using its `get()` method. Alternatively, we could suggest a substitute value for the missing instance, using the method (with the most threatening name) `orElse()`, as in the previous code.

Let's exercise the `pickName()` function with the sample `friends` collection we've used in the examples so far:

```
pickName(friends, "N");
pickName(friends, "Z");
```

The code picks out the first matching element, if found, and prints an appropriate message otherwise.

```
A name starting with N: Nate
A name starting with Z: No name found
```

The combination of the `findFirst()` method and the `Optional` class reduced our code and its smell quite a bit. We're not limited to the preceding options when working with `Optional`, though. For example, rather than providing an alternate value for the absent instance, we can ask `Optional` to run a block of code or a lambda expression only if a value is present, like so:

```
foundName.ifPresent( name ->
     System.out.println("Hello " + name));
```

When compared to using the imperative version to pick the first matching name, the nice, flowing functional style looks better. But are we doing more work in the fluent version than we did in the imperative version? The answer is no—these methods have the smarts to perform only as much work as necessary.

The search for the first matching element demonstrated a few more neat capabilities in the JDK. Next, we'll look at how lambda expressions help compute a single result from a collection.

We could lose all the gains we made from concise and elegant code if not for a newly added join() function. This simple method is so useful that it's poised to become one of the most used functions in the JDK.

Reducing a Collection to a Single Value

We've gone over quite a few techniques to manipulate collections so far: picking matching elements, selecting a particular element, and transforming a collection. All these operations have one thing in common: they all worked independently on individual elements in the collection. None required comparing elements against each other or carrying over computations from one element to the next. In this section, we look at how to compare elements and carry over a computational state across a collection.

Now let's begin with some basic operations and then build up to something a bit more sophisticated. As the first example, we are going to read over the values in the `friends` collection of names and determine the total number of characters.

```
System.out.println(
    "Total number of characters in all names: " +
    friends.stream()
        .mapToInt(name -> name.length())
        .sum());
```

To find the total of the characters, we need the length of each name. We can easily compute that using the `mapToInt()` method. Once we transform the names to their lengths, the final step is to total them. We perform this step using the built-in `sum()` method. Here's the output for this operation:

```
Total number of characters in all names: 26
```

We leveraged the `mapToInt()` method, a variation of the *map* operation (variations such as `mapToInt()`, `mapToDouble()`, and so on create type-specialized streams such as `IntStream` and `DoubleStream`) and then *reduced* the resulting length to the sum value.

Instead of using the `sum()` method, we could use a variety of methods, such as `max()` to find the longest length, `min()` to find the shortest length, `sorted()` to sort the lengths, `average()` to find the average of the length, and so on.

The hidden charm in the preceding example is the increasingly popular MapReduce pattern, with the `map()` method being the spread operation and the `sum()` method being the special case of the more general reduce operation. In fact, the implementation of the `sum()` method in

the JDK uses a `reduce()` method. Let's look at the more general form of the reduce operation.

As an example, let's read over the given collection of names and display the longest one. If there is more than one name with the same longest length, we'll display the first one we find. One way we could do that is to figure out the longest length, and then pick the first element of that length. But that would require going over the list twice, which is not efficient. This is where a `reduce()` method comes into play.

We can use the `reduce()` method to compare two elements against each other and pass along the result for further comparison with the remaining elements in the collection. Much like the other higher-order functions on collections we've seen so far, the `reduce()` method iterates over the collection. In addition, it carries forward the result of the computation that the lambda expression returns. An example will help clarify this:

```
final Optional<String> aLongName =
    friends.stream()
           .reduce((name1, name2) ->
                name1.length() >= name2.length() ?
                    name1 : name2);
    aLongName.ifPresent(name ->
        System.out.println(
            String.format("A longest name: %s", name)));
```

The lambda expression we are passing to the `reduce()` method takes two parameters, `name1` and `name2`, and returns one of them based on the length. The `reduce()` method has no clue about our specific intent. That concern is separated from this method into the lambda expression that we pass to it—this is a lightweight application of the *strategy pattern*.

This lambda expression conforms to the interface of an `apply()` method of a JDK functional interface named `BinaryOperator`. This is the type of the parameter the `reduce()` method receives. Let's run the `reduce()` method and see whether it picks the first of the two longest names from the `friends` list.

```
A longest name: Brian
```

As the `reduce()` method iterated through the collection, it called the lambda expression first, with the first two elements in the list. The result from the lambda expression is used for the subsequent call. In the second call, `name1` is bound to the result from the previous call to the lambda expression, and `name2` is bound to the third element in the collection. The calls to the lambda expression continue for the rest of the elements in the collection. The result from the final call is returned as the result of the `reduce()` method call.

The result of the `reduce()` method is an `Optional` because the list on which `reduce()` is called might be empty. In that case, there would be no longest name. If the list had only one element, `reduce()` would return that element and the lambda expression would not be invoked.

From the example, we can infer that the `reduce()` method's result is at most one element from the collection. If we want to set a default or a base value, we can pass that value as an extra parameter to an overloaded variation of the `reduce()` method. For example, if the shortest name we want to pick is "Steve," we can pass that to the `reduce()` method, like so:

```
final String steveOrLonger =
   friends.stream()
          .reduce("Steve", (name1, name2) ->
               name1.length() >= name2.length() ?
               name1 : name2);
```

If any name is longer than the given base, it is picked up; otherwise, the function returns the base value, which is "Steve" in this example. This version of `reduce()` does not return an `Optional` because if the collection is empty, the default will be returned; there's no concern about an absent or nonexistent value.

Before I wrap up, let's visit a fundamental, yet seemingly difficult, operation on collections: joining elements.

Joining Elements

We've explored how to select elements, iterate, and transform collections. Yet in a trivial operation—concatenating a collection—we could lose all the gains we made from concise and elegant code if not for a

newly added `join()` function. This simple method is so useful that it's poised to become one of the most used functions in the JDK. Let's see how to use it to print the values in a comma-separated list.

Let's work with our `friends` list. What does it take to print the list of names, separated by commas, using only the old JDK libraries?

We have to iterate through the list and print each element. Because the enhanced Java 5 `for` construct is better than the archaic `for` loop, let's start with that.

```
for(String name : friends) {
   System.out.print(name + ", ");
}
System.out.println();
```

That was simple code. It yielded this:

```
Brian, Nate, Neal, Raju, Sara, Scott,
```

Darn it! There's a stinking comma at the end (shall we blame it on Scott?). How do we tell Java not to place a comma there? Unfortunately, the loop will run its course and there's no easy way to tell the last element apart from the rest. To fix this, we can fall back on the habitual loop.

```
for(int i = 0; i < friends.size() - 1; i++) {
   System.out.print(friends.get(i) + ", ");
}

if(friends.size() > 0)
  System.out.println(
      friends.get(friends.size() - 1));
```

Let's see if the output of this version was decent.

```
Brian, Nate, Neal, Raju, Sara, Scott
```

The result looks good, but the code to produce the output does not. Beam us up, modern Java.

We no longer have to endure that pain. A `StringJoiner` class cleans up all that mess in Java 8, and the `String` class has an added convenience method, `join()`, to turn that smelly code into a simple one-liner.

```
System.out.println(String.join(", ", friends));
```

Let's quickly verify that the output is as charming as the code that produced it.

```
Brian, Nate, Neal, Raju, Sara, Scott
```

Under the hood, the `String`'s `join()` method calls upon the `StringJoiner` to concatenate the values in the second argument, a `varargs`, into a larger string separated by the first argument. We're not limited to concatenating only with a comma using this feature. We could, for example, take a bunch of paths and concatenate them to form a classpath easily, thanks to the new methods and classes.

We saw how to join a list of elements; we can also transform the elements before joining them. We already know how to transform elements using the `map()` method. We can also be selective about which elements we want to keep by using methods such as `filter()`. The final step of joining the elements, separated by commas or something else, is simply a `reduce` operation.

We could use the `reduce()` method to concatenate elements into a string, but that would require some effort on our part. The JDK has a convenience method named `collect()`, which is another form of `reduce` that can help us collect values into a target destination.

The `collect()` method does the reduction but delegates the actual implementation or target to a collector. We could drop the transformed elements into an `ArrayList`, for instance. Or, to continue with the current example, we could collect the transformed elements into a string concatenated with commas.

```
System.out.println(
    friends.stream()
        .map(String::toUpperCase)
        .collect(joining(", ")));
```

We invoked the `collect()` method on the transformed list and provided it a collector returned by the `joining()` method, which is a static method on a `Collectors` utility class. A collector acts as a sink object to receive elements passed by the `collect()` method and stores them in a desired format: `ArrayList`, `String`, and so on.

Here are the names, now in uppercase and comma-separated.

```
BRIAN, NATE, NEAL, RAJU, SARA, SCOTT
```

The `StringJoiner` gives a lot more control over the format of concatenation; we can specify a prefix, a suffix, and infix character sequences, if we desire.

Conclusion

As we've seen in this two-part series, lambda expressions and the newly added classes and methods make programming in Java so much easier and more fun, too.

Collections are common in programming and, thanks to lambda expressions, using them is now much easier and simpler in Java. We can trade the long-winded old methods for elegant, concise code to perform common operations on collections. Internal iterators make it convenient to traverse collections, transform collections without enduring mutability, and select elements from collections without much effort. Using these functions means less code to write. That can lead to more maintainable code, more code that does useful domain- or application-related logic, and less code to handle the basics of coding. `</article>`

This article was adapted from Functional Programming in Java: Harnessing the Power of Java 8 Lambda Expressions *with kind permission from the publisher, The Pragmatic Bookshelf.*

LEARN MORE

- Overview of functional programming
- Cay Horstmann's explanation of using lambdas in Java 8

ANTONIO GONCALVES

BIO

Part 3

Contexts and Dependency Injection: The New Java EE Toolbox

More loose coupling with observers, interceptors, and decorators

This series of four articles attempts to demystify Contexts and Dependency Injection (CDI). In the previous two articles, I discussed what strong typing really means in dependency injection and how to use CDI to integrate third-party frameworks. In this article, I focus on how to get loose coupling with interceptors, decorators, and events. The final article will cover the integration of CDI within Java EE.

If you have read the previous articles, you should know by now that CDI is all about loose coupling. But CDI can go even further by using interceptors, decorators, and events. *Interceptors* are a way to solve the problem of cross-cutting technical concerns by intercepting method invocation. *Decoration* is similar to interception but is applied to business concerns. CDI *events* bring about even more loose coupling by implementing the observer/observable pattern in a very easy way.

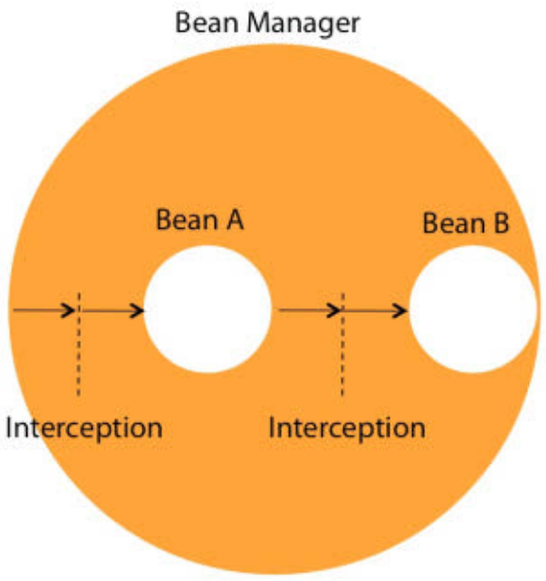


Figure 1. Intercepting method calls

What Is Interception?

Let's start with an explanation of interception, which is used to interpose on method invocations. It is a programming paradigm that separates cross-cutting concerns from our business code. Most applications have common code that is repeated across components—the cross-cutting concerns. These could be technical concerns, such as logging the entry and exit from each method or logging the duration of a method invocation. Or they could be business concerns, such as to perform additional checks if a customer buys more than US$10,000 of items or send a refill order when the inventory level is too low. Both technical concerns and business concerns rely on the container to do much of the legwork of implementation.

The container does the interception. We need to remember that CDI beans live in a managed environment known as a *container* or *bean manager.* This bean manager provides many services, one of which is the ability to intercept method invocation. As can be seen in **Figure 1**, Bean A and Bean B are both managed by the CDI container. When we invoke a method on Bean A or Bean B, we can ask the container to intercept the calls and process some business logic. This can happen before or after the bean method is invoked, so we can also add business logic when the invocation returns.

Interceptors

Interceptors allow adding cross-cutting technical concerns to our beans. For technical concerns, think of transaction management, authentication, authorization, or even a logging mechanism that logs every method invocation. Once we know how to log a method entry, we isolate the code into an interceptor, and enable this interceptor for several beans.

An interceptor binding is just an annotation. You can think of it as a qualifier but for interceptors.

Take the `BookService` class defined in **Listing 1**. Let's say we want to log an entry each time the `createBook` method is invoked. A simple way of doing this is by injecting a logger and tracing the entry. It appears that this technical concern can also be used by other methods and other beans. Instead of copying and pasting code around, we can isolate this code in an interceptor, give it a name (here, it is `@Loggable`), and apply it on whatever method we want. This tells the container, "intercept this method, and do whatever you need to do to log an entry." This behavior can be extended to other technical concerns. In **Listing 1**, we tell the container that every method of the bean needs to be secured and transactional. `@Transactional`, `@Secured`, and `@Loggable` are called *interceptor binding*.

Listing 1.

```
@Transactional
@Secured
public class BookService {

    @Inject
    private IsbnGenerator generator;

    @Loggable
    public Book createBook(String title,
                           Float price) {
        return new Book(
            title, price,
            generator.generateNumber());
    }
    @Loggable
    public Book raisePrice(Book book) {
        book.setPrice(book.getPrice() * 2.5F);
        return book;
    }
}
```

Interceptor binding. An interceptor binding is just an annotation. You can think of it as a qualifier but for interceptors. We give it a meaningful name—here, `Loggable`—and annotate it with `@InterceptorBinding`. Now that we have an interceptor binding, we need to attach it to the interceptor itself, which will provide the logging mechanism.

```
@InterceptorBinding
@Retention(RUNTIME)
@Target({METHOD, TYPE})
@Documented
public @interface Loggable {
}
```

Interceptor implementation. The `LoggingInterceptor` (see **Listing 2**) is a separate class, annotated with our `@Loggable` interceptor binding, but it also needs the special `@javax.interceptor.Interceptor` annotation. This will tell CDI that `LoggingInterceptor` is the interceptor called `@Loggable`. This class uses a logger to log method entries. Notice here that interceptors are CDI beans and can take advantage of dependency injection, for example. Despite being annotated with `@AroundInvoke`, the intercept method—which could be called whatever we want, by the way—must follow certain rules: The method must have an `InvocationContext` parameter and must return `Object`. But what is an *invocation context*?

Listing 2.

```
@Loggable
@Interceptor
public class LoggingInterceptor {

    @Inject
    private Logger logger;

    @AroundInvoke
    private Object intercept(InvocationContext ic)
                    throws Exception {
        logger.info("> {}", ic.getMethod());
        try {
            return ic.proceed();
        } finally {
            logger.info("< {}", ic.g13/16ethod());
        }
    }
}
```

Invocation context. The invocation context allows interceptors to control the behavior of the invocation chain. If several interceptors are chained, the same invocation context is passed to each interceptor. For example, when we invoke the method `createBook` on the `BookService` bean (**Listing 1**), the call is intercepted by the `LoggingInterceptor` (**Listing 2**). As I'll demonstrate later, the chain of interceptors can be longer, and the call can actually be intercepted by interceptor 2, interceptor 3, and so on. All these interceptors can share the same invocation context. With the `InvocationContext` API, each interceptor can get information about the target class (here, `BookService`), the target method, or the method parameters, or it can add contextual data to be processed by other interceptors.

In `LoggingInterceptor` (see **Listing 2**), we can see that the `intercept` method has an `InvocationContext` parameter and is then used to get the method of the bean class for which the interceptor was invoked (`createBook` or `raisePrice`).

Notice that the interceptor needs to invoke the `proceed` method. Calling `InvocationContext.proceed()` is important, because it tells the container that it should proceed to the next interceptor or call the bean's business method. Not calling `proceed` would stop the interceptors chain and would avoid calling the business method.

Intercepting returned invocations. An interceptor can intercept method invocations but also the method returns. Just add a `finally` block to the code (see **Listing 2**). When the `createBook` method is called, the container intercepts the call and first logs a message with the method name. The interceptor proceeds to call the `BookService`, invokes the `createBook` method, returns, and only then logs a message after exiting the method.

Enabling interceptors. By default all interceptors are disabled, so we need to enable them. We can do that using the beans .xml descriptor (see **Listing 3**) by specifying the class name of the interceptor implementation Without this declaration, the interceptor is not taken into account.

Listing 3.

```
<beans xmlns="http://xmlns.jcp.org/xml/ns/javaee"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xsi:schemaLocation="http://xmlns.jcp.org/xml/ns/javaee
           http://xmlns.jcp.org/xml/ns/javaee/beans_1_1.xsd"
       version="1.1" bean-discovery-mode="all">

    <decorators>
       <class>org.foo.bar.ChristmasDiscountDecorator</class>
    </decorators>
</beans>
```

Dealing with Several Interceptors

Interceptor bindings can be seen as qualifiers for interceptors. So, like qualifiers, interceptor bindings can have some advanced usage. For now we have a `LoggingInterceptor` that logs each method invocation. Its interceptor binding is called `@Loggable`. But in certain cases we might want to log

more debug information and use another interceptor—let's say `LoggingDebugInterceptor` (see **Listing 4**, which I'll explain shortly). How would we call its interceptor binding—`@LoggableWithDebug`? And if we have other logging interceptors, we would have as many interceptor bindings. Like qualifiers, we could create as many different interceptor bindings as we have implementations, or we could add members to our interceptor bindings, or we could aggregate them.

Listing 4.

```
@Loggable(debug = true)
@Interceptor
public class LoggingDebugInterceptor {

    @Inject
    private Logger logger;

    @AroundInvoke
    private Object intercept(InvocationContext ic)
            throws Exception {
logger.info("> {}", ic.getMethod());
logger.info("> Parameters            : {}",
    ic.getParameters());
final Class<? extends Object> runtimeClass =
    ic.getTarget().getClass();
logger.info("> Runtime class         : {}",
    runtimeClass.getName());
logger.info("> Extended classes      : {}",
    new Object[]{runtimeClass.getClasses()});
logger.info("> Implemented interfaces: {}",
    new Object[]{runtimeClass
       .getInterfaces()});
logger.info("> Annotations ({})       : {}",
      runtimeClass.getAnnotations().length,
      runtimeClass.getAnnotations());
final Class<?> declaringClass =
   ic.getMethod().getDeclaringClass();
logger.info("> Declaring class       : {}",
    declaringClass);
    logger.info("> Extended classes      : {}",
        new Object[]{declaringClass
            .getClasses()});
    logger.info("> Annotations ({})       : {}",
        declaringClass.getAnnotations().length,
        declaringClass.getAnnotations());
    try {
          return ic.proceed();
    } finally {
        logger.info("< {}", ic.getMethod());
    }
   }
}
```

Interceptor binding with members. An interceptor binding is an annotation, so it can have as many members of any type as needed. Here, to differentiate between logging and debug logging, we could use a Boolean. Any other members of any data type are, of course, allowed.

```
@InterceptorBinding
@Retention(RUNTIME)
@Target({METHOD, TYPE})
@Documented
public @interface Loggable {
    boolean debug();
}
```

So now we have two interceptor implementations: `LoggingInterceptor` (see **Listing 5**) for normal logging and `LoggingDebugInterceptor` (see **Listing 4**) for extra debug logging. To differentiate them, we use our interceptor binding and set different values on the Boolean member: `debug = false` for `LoggingInterceptor` and `debug`

CDI events implement the observer/observable design pattern and are perfect for decoupling components that have no compile time dependency.

= true for LoggingDebugInterceptor. Each interceptor implementation is now uniquely defined.

Listing 5.

```
@Loggable(debug = false)
@Interceptor
public class LoggingInterceptor {

    @Inject
    private Logger logger;

    @AroundInvoke
    private Object intercept(InvocationContext ic)
            throws Exception {
        logger.info("> {}", ic.getMethod());
        try {
            return ic.proceed();
        } finally {
            logger.info("< {}", ic.getMethod());
        }
    }
}
```

The way we use interceptor bindings with members on our beans is the same as qualifiers. In **Listing 6**, the createBook method will be intercepted and will trace a log, while the raisePrice method will log debug information.

Listing 6.

```
public class BookService {

    @Loggable(debug = true)
    public Book createBook(
        String title, Float price) {
        // ...
    }

    @Loggable(debug = false)
    public Book raisePrice(Book book) {
        // ...
    }
}
```

Aggregating interceptor bindings. Another way of qualifying an interceptor and a bean is to specify multiple interceptor bindings. Like qualifiers, interceptor bindings can be aggregated. So we could keep @Loggable for the createBook method, and add a @Debug interceptor binding (see **Listing 7**).

Listing 7.

```
public class BookService {

    @Loggable @Debug
    public Book createBook(
        String title, Float price) {
        // ...
    }

    @Loggable
    public Book raisePrice(Book book) {
        // ...
    }
}
```

Ordering Interceptors

Interceptors are very handy, and it's quite common to have several of them intercepting methods. As an example, when we invoke the createBook method, we might want an interceptor to log the call, another one to see how many milliseconds the call takes, and an additional one to track the thread. The order in which interceptors are called is very important. Do we want @Loggable to execute first? Would it make sense to track the thread at the very end of the chain or at the beginning? It depends on what we want to do, but ordering can definitely make a difference.

Multiple interceptors. The `BookService` in Listing 8 shows how to pack a list of interceptor bindings on top of the `createBook` method. The Java language doesn't take annotations ordering into account. It's not because `@ThreadTrackable` is at the bottom, in the middle, or at the top that the ordering will change. Ordering comes in two flavors: with the beans.xml deployment descriptor, or with the `@Priority` annotation.

Listing 8.

```
public class BookService {

    @Inject
    private IsbnGenerator generator;

    @Loggable
    @Auditable
    @ThreadTrackable
    public Book createBook(
          String title, Float price)  {
          return new Book(
             title, price,
             generator.generateNumber());
    }
}
```

Sequencing interceptors with beans.xml. Remember that interceptors are disabled by default, and they need to be enabled in the beans.xml file. The order in which they are listed in the file will be the order in which they will be executed. For example, in Listing 9, `LoggingInterceptor` will be executed first, `AuditInterceptor` second, and `ThreadTrackerInterceptor` last. If I change the order, the invocation chain will change. However, this activation and ordering applies only to the beans in that archive, not to the beans in the entire application. Since CDI 1.1, interceptors can be enabled and ordered for the whole application using the `@Priority` annotation.

Listing 9.

```
<beans xmlns="http://xmlns.jcp.org/xml/ns/javaee"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://xmlns.jcp.org/xml/ns/javaee
          http://xmlns.jcp.org/xml/ns/javaee/beans_1_1.xsd"
    version="1.1" bean-discovery-mode="all">

 <interceptors>
    <class>org.foo.bar.LoggingInterceptor</class>
    <class>org.foo.bar.AuditInterceptor</class>
    <class>org.foo.bar.ThreadTrackerInterceptor</class>
 </interceptors>
</beans>
```

Sequencing interceptors with @Priority. `@Priority` takes an integer that can have any value. The rule is that interceptors with smaller priority values are called first. While any value can be used, keep in mind that Java EE 7 defines several platform-level priorities and so interceptors can be called before or after a specific action. `PLATFORM_BEFORE` has the value of zero, which is the starting value for early interceptors. `LIBRARY_BEFORE` equals 1000; it is for early interceptors defined by extension libraries. `APPLICATION` equals 2000 and is for interceptors defined by applications; this is usually when our interceptors will be executed. Then comes `LIBRARY_AFTER` for late interceptors defined by extension libraries and `PLATFORM_AFTER` for late interceptors. So if we want our interceptor to be executed before any application interceptor but after any early platform interceptor, we can use the value `LIBRARY_BEFORE + 10`.

Ordering interceptors with @Priority. Instead of ordering with beans.xml we can order our interceptors with the `@Priority` annotation (see Listing 10). `AuditInterceptor` will be executed first because it has the lowest priority value, 2010 (or `APPLICATION + 10`); then `LoggingInterceptor`, 2020; and last, `ThreadTrackerInterceptor`. If we change the numbers, the ordering will change as well. Coming back to the

interceptors of the `createBook` method defined in **Listing 8**, `@Auditable` will be executed first, then `@Loggable`, and finally `@ThreadTrackable`.

Listing 10.

```
@Auditable
@Interceptor
@Priority(APPLICATION + 10)
public class AuditInterceptor {
      // ...
}
@Loggable
@Interceptor
@Priority(APPLICATION + 20)
public class LoggingInterceptor {
    // ...
}
@ThreadTrackable
@Interceptor
@Priority(APPLICATION + 30)
public class ThreadTrackerInterceptor {
 // ...
}
```

Decorators

Interceptors perform cross-cutting tasks and are perfect for solving technical concerns. By their nature, interceptors are unaware of the actual semantics of the actions they intercept, and therefore are not appropriate for separating business-related concerns. The reverse is true for decorators. Decorators are a common design pattern, initially described by the the Gang of Four. The idea is to take a class and wrap another class around it. This way, when you call a decorated class you always pass through the surrounding decorator before you reach the target class, also known as the `@Delegate`. Decorators are meant to facilitate adding additional logic to a business method. Interceptors and decorators, though similar in many ways, are complementary. Just remember that interceptors are called before decorators when applied to the same method.

Figure 2 illustrates decoration. Here, the CDI bean `PurchaseOrderService` has one compute method that computes the total amount for a purchase order. An interceptor could intercept the call to this method and perform some cross-cutting technical logic. The decorator is slightly different because it is aware of the target's logic. For example, we could need a decorator that adds a discount to the purchase order for Christmas. When we invoke the business method `compute`, a `ChristmasDiscountDecorator` intercepts the call, gets the total amount of the purchase order, and then applies a certain discount percentage. The `PurchaseOrderService` is called the target, and the `ChristmasDiscountDecorator` is the decorator.

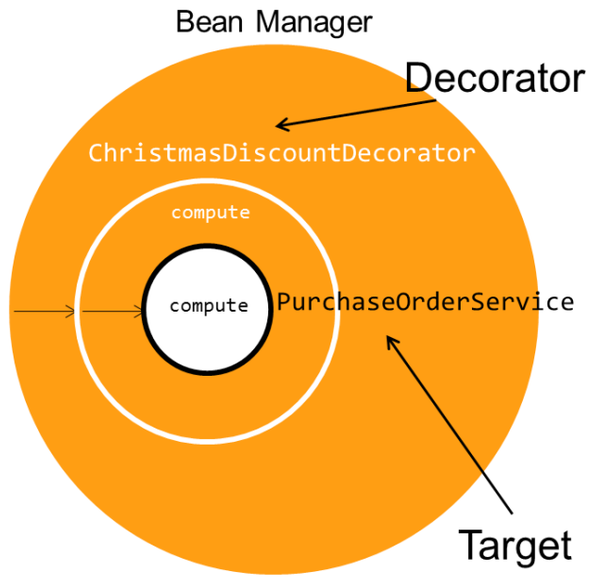


Figure 2. Inside a decorator

Implementing the target. Now let's illustrate this business case with code. For decorators to be aware of the target's business logic, both the target and the decorator need to implement the same interface. In our example, we use the `Computable` interface (see **Listing 11**), which has a `compute` method that takes a list of items (let's say books), computes the price, and returns a purchase order.

Listing 11.

```
public interface Computable {

    PurchaseOrder compute(List<Item> items);
}
```

The `PurchaseOrderService` (in Listing 12) is the concrete class responsible for computing the value of a purchase order, so it implements the `Computable` interface, and therefore, the `compute` method. This method loops through the items, sums up the price, does not apply any discount, and returns a `PurchaseOrder` object.

Listing 12.

```
public class PurchaseOrderService
       implements Computable {

    @Inject
    @Vat
    private Float vatRate;

    @Override
    public PurchaseOrder compute(List<Item> items) {
        PurchaseOrder po = new PurchaseOrder();
        Float subtotal = 0f;

        // Sum up the quantities
        for (Item cartItem : items) {
            subtotal += (cartItem.getSubTotal());
        }

        Float vat = subtotal * (vatRate / 100);
        Float total = subtotal + vat;

        po.setSubtotal(subtotal);
        po.setTotal(total);
        po.setTotalAfterDiscount(total);
        return po;
    }
}
```

Implementing the decorator. The `ChristmasDiscount Decorator` in Listing 13 is supposed to apply a discount to a purchase order that has been computed. Like the previous `PurchaseOrderService`, the `ChristmasDiscountDecorator` needs to implement the same `Computable` interface. This is because the decorator conforms to the interface of the component it decorates so that its presence is transparent to the clients. The class also needs to be annotated with `@Decorator`. Then, the decorator implements the methods of the decorated type that it wants to intercept. In our case, we implement only `compute`. But we could have implemented other methods, or declared the decorator abstract so that it does not have to implement all the business methods of the interface. The `compute` method invokes the target bean, adds some business logic, and returns the purchase order. The question is, what do we need to inject to get the `PurchaseOrderService`?

Decorators have a special injection point, called the *delegate injection point*, with the same type as the beans they decorate—here, the `Computable` interface and the annotation `@Delegate`. In this code, the decorator forwards the request to the target class `PurchaseOrderService` and performs additional actions. A delegate injection point may specify any number of qualifiers if needed. And because a decorator is a CDI bean, it can inject any needed object—in this case, the `discountRate`, which has been produced somewhere else.

Listing 13.

```
@Decorator
public abstract class ChristmasDiscountDecorator
       implements Computable {
```

```
    @Inject
    @Discount
    private Float discountRate;

    @Inject
    @Delegate
    private Computable purchaseOrderService;

    @Override
    public PurchaseOrder compute(List<Item> items){
        PurchaseOrder po =
            purchaseOrderService.compute(items);

        po.setTotalAfterDiscount(po.getTotal() -
            po.getTotal() * discountRate);

        return po;
    }
}
```

Enabling decorators. By default, all decorators are disabled like alternatives and interceptors, so we need to enable them using the beans.xml descriptor just by specifying the class name of the decorator implementation (see **Listing 14**). If we have multiple decorators, we can order them. And like interceptors, decorators can alternatively be enabled and ordered using the `@Priority` annotation.

Listing 14.

```
<beans xmlns="http://xmlns.jcp.org/xml/ns/javaee"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xsi:schemaLocation="http://xmlns.jcp.org/xml/ns/javaee
           http://xmlns.jcp.org/xml/ns/javaee/beans_1_1.xsd"
       version="1.1" bean-discovery-mode="all">

    <decorators>
        <class>org.foo.bar.ChristmasDiscountDecorator</class>
    </decorators>
</beans>
```

Events

Dependency injection, alternatives, interceptors, and decorators enable loose coupling by allowing additional behavior to vary, either at deployment time or at runtime. Events go one step further, allowing beans to interact with no compile time dependency at all. One bean can fire an event, and another bean can observe the event. The beans can be in separate packages and even in separate tiers of the application. This basic schema follows the observer/observable design pattern from the Gang of Four. Event notifications decouple event producers (the ones firing events) from event consumers (the ones consuming the events).

Figure 3. Loose coupling CDI-style

To illustrate event management, let's look at an example. In **Figure 3**, we have a `PurchaseOrderService` with a `create` method. When we invoke the `create` method, it creates a purchase order and then calls the `InventoryService` to update the item stock. As we've seen, CDI is the perfect framework to deal with dependencies. So here, `PurchaseOrderService` depends on `InventoryService`, and we could easily use `@Inject`. Then we realize that we need the `ShippingService` to ship items to the right location. Again, another `@Inject` could do. And what if we need to update some statistics of items sold, or invoke other services? Do we need to change the code of the `PurchaseOrderService` each time? Dependency injection enables loose coupling by allowing the implementation of the injected bean type to vary, either at deployment

time or runtime, but with a compile time dependency.

In our case, we want to decouple the dependencies, so we can have as many services as needed, without changing the `PurchaseOrderService`. Events go one step further, allowing beans to interact with no compile time dependency at all. Instead of depending on all these services, `PurchaseOrderService` fires an event to inform potential observers that a purchase order has been created. `PurchaseOrderService` is the event producer and is not aware of what will observe this event. Somewhere in the application, the `InventoryService` decides to observe this event and performs some business logic. `ShippingService` and `StatisticService` can also observe the same event and perform their own business logic. All this happens in a very loosely coupled way, because no service is aware of any other. Event producers fire events that are delivered to event observers by the container. Not only are event producers decoupled from observers, but observers are completely decoupled from producers.

It is important to understand that the event mechanism is synchronous. Once the event is fired, the container pauses the code execution and passes the event to any registered observers.

Event producer. Let's have a look at the event produce. The `PurchaseOrderService` is defined in **Listing 15**. The `create` method takes a list of items, does some business logic, and returns a `PurchaseOrder` object. Now, to fire an event is quite easy in terms of code. Event producers fire events using an instance of the `Event` interface. An instance of this interface is obtained by injection and is typed the same as the producer's object. In our example, we want to fire the `PurchaseOrder`, so `Event` is typed with `PurchaseOrder`.

Next, the producer fires the event by calling the `fire` method of the `Event` interface, passing the `PurchaseOrder` object. The `PurchaseOrder` object will be carried from the producer to the consumers. As we can see, this code doesn't have any reference to any other service. Producers and observers are totally decoupled.

Listing 15.

```
public class PurchaseOrderService {

    @Inject
    private Event<PurchaseOrder>
        purchaseOrderEvent;

    public PurchaseOrder create(
        List<Item> items) {
        PurchaseOrder po = new PurchaseOrder();
        // Sum up the quantities
        // Set total and subtotal

        purchaseOrderEvent.fire(po);
        return po;
    }
}
```

Event observer. Now the `InventoryService` in **Listing 16** needs to catch this event and do some processing. For that, it needs to declare an `observer` method. This method takes the `PurchaseOrder` as a parameter and is annotated with `@Observes`. That's it. The `addItems` method is notified of the event by the container and can then keep the inventory up to date. It is important to understand that the event mechanism is synchronous. Once the event is fired from the `PurchaseOrderService`, the CDI container pauses the execution and passes the event to any registered observers. In our case, the `addItems` method will be invoked, the inventory will be updated, and the container will then continue the code execution where it paused in

the `PurchaseOrderService`. Events in CDI are not treated asynchronously.

Listing 16.

```
public class InventoryService {

    @Inject
    private Logger logger;

    public void addItems(
      @Observes PurchaseOrder po) {
         logger.info("Purchase Order of ${}",
                     po.getTotal());
    }
}
```

Multiple event observers. Now it's just a matter of having as many observers as needed. The `ShippingService` observes the same event and, once the event is received, it takes all the needed information from the `PurchaseOrder` to ship the items to a specific destination. The `StatisticService` also observes the event to update its statistics of items sold, just by having a method annotated with `Observes` and by specifying the type of the event: `PurchaseOrder`.

Being synchronous, the method firing the event has to wait until the end of all the observer's invocations before executing the instructions after event firing. If any `observer` method throws an exception, the container stops calling `observer` methods, and the exception is rethrown by the `fire` method. One thing to keep in mind with observers is that there is no ordering in CDI 1.1 and the invocation is synchronous. The `@Priority` annotation doesn't work with observers, so there is no solution to guarantee the order of the observers' execution. Ordering observers, however, is on the roadmap of CDI 2.0.

One thing to keep in mind with observers is that there is no ordering in CDI 1.1 and the invocation is synchronous. There is no solution to guarantee the order of the observers' execution. Ordering observers, however, is on the roadmap of CDI 2.0.

Conclusion

In this article, we've seen that CDI delivers loose coupling by implementing several design patterns in a very easy way. Interceptors interpose in method invocations and perform technical cross-cutting tasks, such as logging or auditing. Decorators intercept invocations for a specified business interface, and therefore are aware of all the semantics attached to that interface. Because decorators directly implement operations with business semantics, they are the perfect tool for intercepting business concerns. Interceptors and decorators can be ordered if needed. CDI events implement the observer/observable design pattern and are perfect for decoupling components that have no compile time dependency. `</article>`

LEARN MORE

- CDI specification
- *Beginning Java EE 7* (book)
- PluralSight course on CDI 1.1
- Weld CDI reference implementation

A First Look at Microservices

The latest trend in enterprise computing is microservices. What exactly are they?

ARUN GUPTA

BIO

Customers expect a solution to be available on a variety of devices ranging from mobile clients and wearables to the Internet of Things. They expect a personalized experience and more-frequent updates. Businesses are trying to gain advantage over competitors with shortened time to market and a faster release cadence. Microservices are quickly emerging as part of the solution of this quest for speed of delivery.

Microservices Defined

Microservices are the product of an architectural approach that emphasizes the functional decomposition of applications into single-purpose, loosely coupled services managed by cross-functional teams for delivering and maintaining complex software systems quickly.

That's quite a lot of terms in this definition. Let's break them down and understand each one.

Architectural approach. Unlike Java EE, Spring, and .NET, the microservice architecture is a language-, platform-, and operating system–agnostic style that provides recommended guidelines on how applications need to be built.

Functional decomposition. A typical monolithic application in Java EE, for example, would consist of an EAR or WAR file. The entire functionality expected from the application is packaged in that one archive. There are several advantages of this approach, but the application tends to break down as it grows. Such an application needs to be decomposed into multiple WAR files where each WAR consists of a single function. This is where we get the concept of a microservice.

Each service can be independently deployed, and redeployed again, without affecting the overall system. This approach allows a service to be easily upgraded, for example, to add more features.

Single-purpose. Each decomposed part of an application is responsible for a single purpose only, and does it well. This smallness of scope is what the *micro* in *microservices* refers to.

Loosely coupled services. Service consumers do not have any knowledge about implementation of the producer service. Producers and consumers communicate only through a predefined contract. This makes the service loosely coupled, which leads to more manageable, robust, and scalable applications. This also means that updating producer services doesn't require changing the consumer.

Decomposed applications are loosely coupled, which leads to more manageable, robust, and scalable applications. This means that updating one service doesn't require changing another service.

Cross-functional teams. Each microservice is created by a team that has a mix of different skills such as persistence, API design, and UI. This gives each team the competence to control the technology stack.

Complex software systems. The benefits of such an architecture style are evident only when building a nontrivial software system. This is so because

building a microservice simplifies the application code, but pushes the problem of plumbing to an integration layer and thus requires significant support from the infrastructure. Return on investment for a simple application might not be evident to begin with.

Characteristics

With this basic definition, let's examine the key characteristics of a microservices-based application.

- **Domain-driven design.** Functional decomposition of an application can be achieved using the well-defined principles of domain-driven design (DDD), by Eric Evans. Decomposition by domain is not the only way to break down the application but certainly a very common way. Each team is responsible for building the entire functionality around that domain or function of the business. Each team that builds a service includes the full range of developer skills, thus enabling a full-stack development methodology.
- **Single responsibility.** Each microservice should have responsibility for a single part of the functionality, and it should do that well.
- **Explicitly published interface.** Each service publishes an explicitly defined interface and honors that at all times. The *consuming service* cares only about that interface and should not have any runtime dependency on the *consumed service*. The services agree upon the domain models, API, payload, or some other contract; they communicate using only that. A newer version of the interface may be introduced, but either the previous versions continue to exist or the newer services are backward-compatible. Microservices may not break compatibility by changing contracts.
- **Independent DURS (Deploy, Upgrade, Replace, Scale).** Each service can be independently deployed, and redeployed again, without affecting the overall system. This approach allows a service to be easily upgraded, for example, to add more features. Each service can also scale independently using horizontal duplication or sharding. Implementation of the service, or even the underlying technology stack, can change as long as the exact same contract is published. This is possible because of the loose coupling.
- **Potential heterogeneity.** The implementation details of one service should not affect another service. This enables services to be decoupled from each other, and allows the team building a service to pick the language, persistence store, tools, and methodology that are most appropriate for them. A service that needs to store data in a relational database can choose MySQL, and another service that needs to store documents can choose MongoDB. Different teams can choose whatever technology is most efficient for them.
- **Lightweight communication.** Services communicate with each other using a lightweight communication, such as REST over HTTP. An alternative mechanism is to use a publish-subscribe mechanism that supports asynchronous messaging. Any of the messaging protocols, such as AMQP, STOMP, MQTT, or WebSocket, that meets the needs can be used here. Simple messaging implementations, such as ActiveMQ, that provide a reliable asynchronous fabric are quite appropriate for such usages. The choice of synchronous and asynchronous messaging is very specific to each service; even a combination of the two approaches can be used. Similarly, the choice of protocol is very specific to each service, but there is enough choice and independence for each team building a service.

> The scope of each microservice is much smaller than a monolith, and this leads to a smaller executable. **As a result, the deployment and the startup are much faster.**

Benefits

What are the benefits of a microservices-based application?

- **Easy to develop, understand, and maintain.** Code in a microservice is restricted to one function of the business and is thus easier to understand than a full application might be.
- **Starts faster than a monolith.** The scope of each microservice is much smaller than a monolith, and this leads to a smaller executable. As a result, the deployment and the startup are much faster.
- **Easy to deploy local change.** Each service can be deployed independently of other services. Developers can easily make any change local to the service without requiring coordination with other teams.
- **Scales independently.** Each service can scale independently based upon need. In contrast, monolithic applications might have different requirements and yet must be deployed and scaled together.
- **Improves fault isolation.** A misbehaving service, such as one with a memory leak or unclosed database connections, will affect only itself as opposed to affecting the entire monolithic application. This improves fault isolation.
- **Eliminates long-term commitment to any stack.** Developers are free to pick the language and the stack that are best suited for their service. Although the organization may restrict the choice of technology, you are not penalized because of past decisions. It also enables you to rewrite the service using better languages and technologies. This gives you freedom of choice when selecting the technology, tools, and frameworks.

What Is the Relationship with SOA?

A common misconception is that there is nothing new about microservices—that they are similar to service-oriented architecture (SOA). Some of the common pithy names for microservices are "fine-grained SOA," "SOA for hipsters," and "SOA done right." In this sense, microservices do capture the original essence of SOA, where multiple services are composed together to achieve business functionality. Microservices also bear a resemblance to service location transparency, service discovery, loose coupling, and stateless services. But there are some fundamental differences.

- **Services tied to a common platform.** SOA has been hijacked by the enterprise service bus (ESB) vendors. All services are tied to an SOA product. Vendors pitch their proprietary products, with minimal to no common standards for interoperability across products. This limits the ability to innovate within each product.
- **Overly complex SOAP-based web services.** Web-service specifications are overly complex. All the XML and SOAP processing makes it time consuming to generate and consume messages. Implementations of these specifications require significant investment that is primarily done for Java and .NET only. These implementations serve a specific purpose—in some enterprises today, rather limited—but most of the recent projects have been adopting the REST style of communication.
- **Centralized routing and transformation.** All messages are routed through a central ESB, which can transform and route the messages based upon business rules. This is opposed to the "smart endpoint, dumb pipe" philosophy for microservices, where the connection performs no transformation and is simply a conduit.
- **Centralized governance.** SOA has a major emphasis on centralized governance that defines how services will be defined, developed, and maintained. All services are registered with a central registry and are invoked by an ESB to monitor and meter usage. Service registry and discov-

You are free to choose any stack of your choice. No one framework is ideal for building your microservices; pick the one that you are most comfortable with.

ery in the microservices world are done using a third-party tool such as ZooKeeper, Consul, or etcd. The services invoke each other directly, as opposed to going through a central bus.

Refactor Monolith into Microservices

While this task is worthy of a separate article, here's a quick overview of what is entailed.

Consider a trivial shopping cart web application consisting of catalog, shopping cart, and shipping functionality. A typical monolithic way to package such an application would be to have all the web pages, classes, and configuration files, for the entire functionality, together in a single WAR or EAR file, which is deployed to a Java EE application server. There is a single database for the entire application, and it is shared by all the classes.

Functional decomposition of this application would separate the web pages, classes, and configuration files for one domain, such as catalog, in a separate WAR file. One WAR file is split up into three WAR files, each containing all the artifacts required to serve its functionality. The database shared by the monolithic application is also refactored into three separate databases. Each WAR file publishes a well-defined interface, likely using REST. And a WAR file can be deployed in a separate application server and scaled independently.

Is Java EE Well Suited for Microservices?

Microservices enable heterogeneity and polyglot stacks. This is possible because the only requirement is a well-defined interface, and the implementation is a detail left to each team to accomplish as it sees fit. This gives teams freedom from a long-term commitment to any particular stack, and it allows them to innovate. There are some recent technologies in Java EE that make it particularly well suited to microservices, especially if team members have experience with server-side Java. These capabilities include the following:

- Java Naming and Directory Interface provides location transparency and service discovery.
- Contexts and Dependency Injection (CDI) gives loose coupling and strong cohesion.
- JAX-RS provides full support for REST.
- Java Message Service and CDI events enable the publish-subscribe model for asynchronous messaging.
- Native support is available for JSON data representation. Transactions within a microservice are easily supported by Enterprise JavaBeans and `@Transactional`. Many platform-as-a-service vendors already offer Java EE support.
- A modular application server, such as JBoss Enterprise Application Platform, allows you to bundle only the required components, so you can keep the footprint size minimal.
- Each microservice can be deployed as a WAR file.
- Extensive tooling simplifies Java EE development, which lowers the bar for building such applications.

If there is a missing capability, it can be implemented as a CDI extension.

As mentioned earlier, a team is not limited to Java EE, and you can choose any stack of your choice. For example, Vert.x is a simple, polyglot, and JVM-based framework that facilitates the creation of lightweight applications and fat JARs easily. No one framework is ideal for building your microservices; pick the one that you are most comfortable with.

Ready or not, microservices are coming your way. Start thinking about them for your next project. Start with a pilot project, benefit from that first tryout, and grow from there.

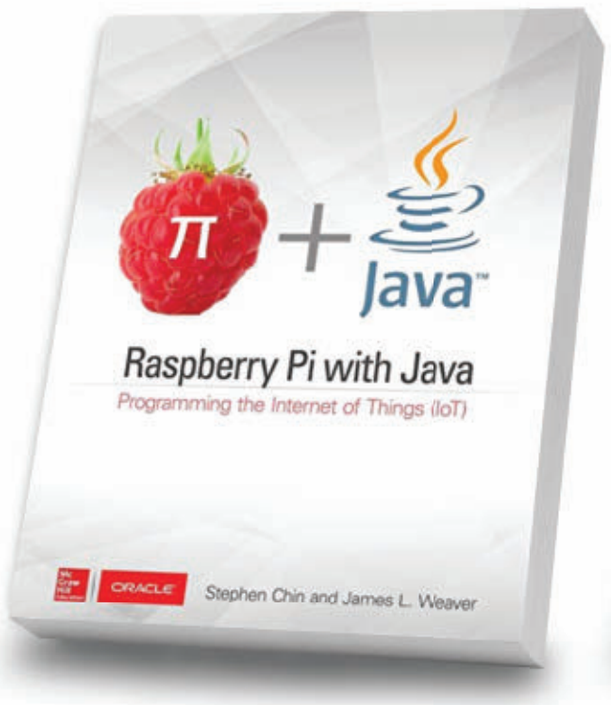
Happy microservicing! `</article>`

Ready or not, microservices are coming your way. Start thinking about them for your next project. Start with a pilot project, benefit from that first tryout, and then grow from there.

Oracle Press™

Your Destination for Java Expertise

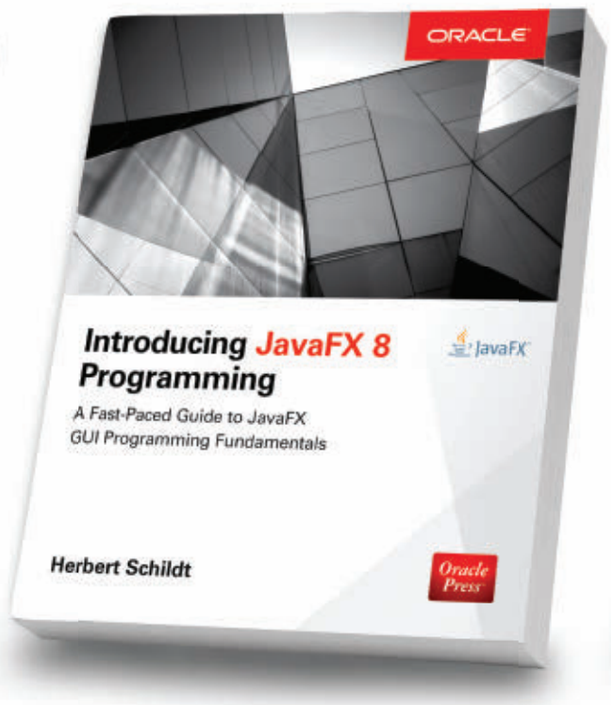
Written by leading Java experts, Oracle Press books offer the most definitive, complete, and up-to-date coverage of Java available.



Raspberry Pi with Java: Programming the Internet of Things (IoT)

Stephen Chin, James Weaver

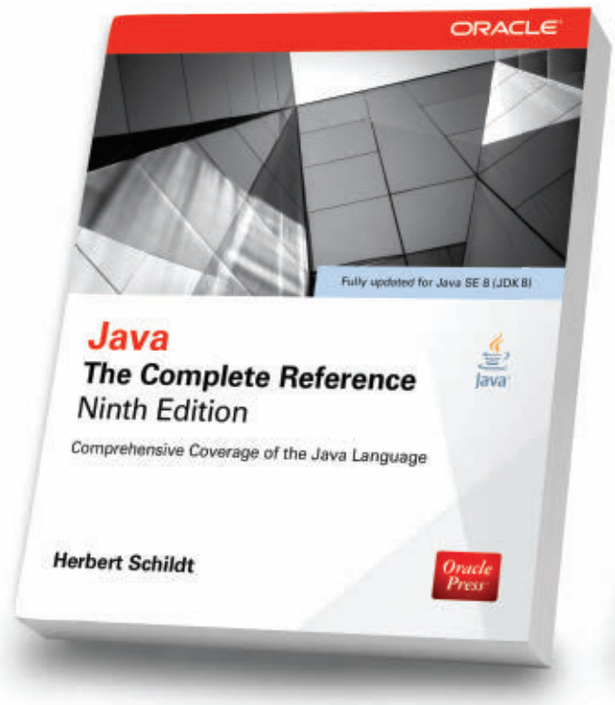
Use Raspberry Pi with Java to create innovative devices that power the internet of things.



Introducing JavaFX 8 Programming

Herbert Schildt

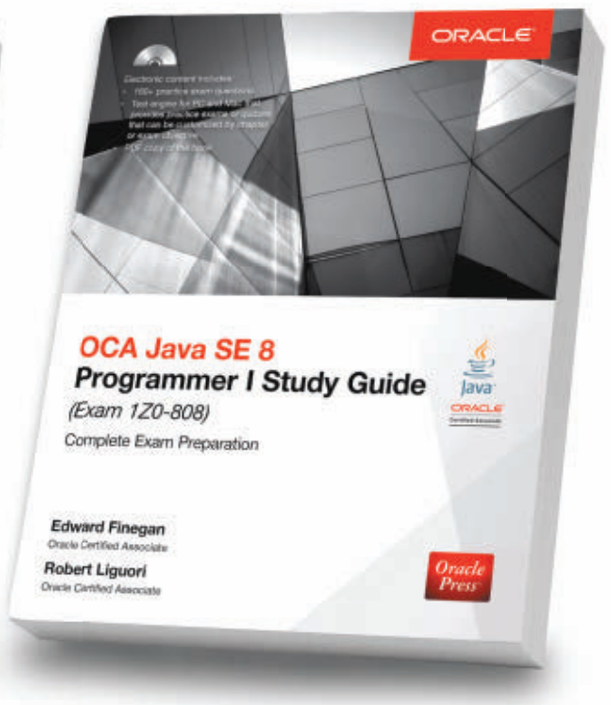
Learn how to develop dynamic JavaFX GUI applications quickly and easily.



Java: The Complete Reference, Ninth Edition

Herbert Schildt

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



OCA Java SE 8 Programmer I Study Guide (Exam 1Z0-808)

Edward Finegan, Robert Liguori

Get complete coverage of all objectives for Exam 1Z0-808. Electronic practice exam questions are included.

Available in print and as eBooks

Oracle Press™

www.OraclePressBooks.com • @OraclePress

Quiz Yourself

We all write code from within a subset of the full Java language. How well do we know that subset? Let's see.

The questions in this quiz are taken from certification test 1Z0-809: Oracle Certified Associate, Java SE 8 Programmer II, Oracle Certified Java Programmer. The purpose of this certification is to enable beginners to demonstrate their knowledge of Java 8 concepts, somewhat above the fundamental level.

Naturally, these questions can have small unexpected traps that can snag even the attentive coder. But in this way, they reflect situations that occur in real life in which we wonder why code that looks right does not behave as expected. Ready? (Answers appear in the "Answers" section immediately after the questions.)

Question 1. Given this code fragment:

```
Map<Integer, Integer> codes = new TreeMap<>();
codes.put(1, 30);
codes.put(3, 20);
codes.put(2, 10);
codes.put(1, 40);
System.out.println(codes);
```

What is the result?

a. {1=40, 2=10, 3=20}
b. {1=40, 1=30, 2=10, 3=20}
c. {2=10, 3=20, 1=40}
d. {1=40, 3=20, 2=10}

Question 2. Given this code fragment:

```
public abstract class Product { // line n1
    String name;
    Product(String name) {      // line n2
        this.name = name;
    }
    public final void printProduct() { // line n3
        System.out.println(name);
    }
    public void printLabel();    // line n4
}
```

Which line causes a compile-time error?

a. line n1
b. line n2
c. line n3
d. line n4

Question 3. Let's have a look at using the `Path` interface to operate on file and directory paths. Given this code fragment:

```
Path path = Paths.get("/home/user/./info.txt");
path.normalize();
System.out.println(path.getNameCount());
```

What is the result if the /home/users/./info.txt file does not exist?

a. 3
b. 4

c. 1
d. A `NoSuchFileException` is thrown at runtime.

Question 4. Now, let's filter a collection using lambda expressions. Given this code fragment:

```
Stream<Integer> nS = Stream.of(5, 6, 8);
// line n1
```

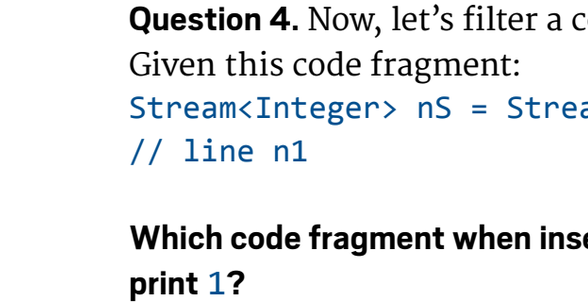
Which code fragment when inserted at `line n1` enables the code to print `1`?

a.
```
List<Integer> oS =
        nS.filter(n -> n%2==1).toList();
    System.out.println(oS.size());
```

b.
```
Stream<Integer> oS =
        nS.filter(n -> n%2==1);
    System.out.println(oS.count());
```

c.
```
List<Integer> oS =
        nS.filter(n -> n%2==1).collect();
    System.out.println(oS.size());
```

d.
```
Stream<Integer> oS =
        nS.allMatch(n -> n%2==1);
    System.out.println(oS.count());
```



Question 1. Option A is correct. The `TreeMap` instance is sorted according to the natural ordering of its keys. The keys are unique in all maps. The `put()` method replaces the previous value associated with the given key in the map and, therefore, 30 is replaced with 40.

Option B is incorrect. The keys are unique in a map object. 30 is replaced with 40. Option C is incorrect. The values in the `TreeMap` instance are sorted according to the natural order of their keys. Option D is incorrect. The elements of the `codes` map are sorted in ascending order of its keys.

Question 2. Option D is correct: line n4 is invalid. A method that does not have its definition must be declared with `abstract`. To fix the compilation error at line n4, replace it with `public abstract void printLabel();`.

Option A is incorrect: line n1 is valid. A class declared with `abstract` can contain both abstract methods and concrete methods. Option B is incorrect: line n2 is valid. An abstract class can have a constructor. Option C is incorrect: line n3 is valid. An abstract class can have final methods.

Question 3. Option B is correct. The program prints the number of elements in the path string.

Options A and C are incorrect. On the second line, the path is normalized. The `Path` instance is immutable. Therefore, the `path.getNameCount()` method returns the number of elements in the path string. The program does *not* print the count of path elements to be redundant. Option D is incorrect. The `Path` instance is the string representation of the given path. The `/home/users/.info.txt` file need not be available in the file system. Only when `path.toRealPath()` is used and the file is not available in the file system is a `NoSuchFileException` thrown.

Question 4. Option B is correct. The `filter()` method returns a stream. Option A is incorrect. The `filter()` method returns a stream. A stream can be converted to a `List` type using the `collect()` method with the `Collectors.toList()` parameter. The `toList()` method cannot be used to convert a stream into a list. Option C is incorrect. The `collect()` method performs a mutable reduction operation on the elements in this stream using a `Collector`. It requires the `Collectors.toList()` parameter to convert the stream into a `List` instance. Option D is incorrect. The `allMatch()` method returns a `boolean` based on the provided predicate test.

BRUNO SOUZA AND
EDSON YANAGA

BIO

Part 3

More Ideas to Boost Your Developer Career

Skills to develop, activities to explore

Think about the best developers you know—people you admire for the work they do. Maybe you follow them on Twitter, watch their presentations, or read some of their articles or blog posts. Have some of them made an impact on your life and career? Why—because they write amazing code? Chances are that this is not the only thing they do to inspire you to follow and admire their work. The truth is that a flourishing developer career involves much more than just writing great code.

This series of articles focuses on three areas that we think have equal weight in a developer's career: code, community, and an open mind. Focusing on one while ignoring the others is a bad practice that limits personal and professional growth.

The good news is that changing your mindset does not require a huge effort. It is something you can do no matter what project you are working on—or what your employer thinks about it. It doesn't take a lot of time, and it is really a lot simpler than most things you deal with in your daily job. The best way to steer something in the right direction is to get it moving first. By taking small daily steps in the right areas, you can continuously push your career forward and achieve your goals.

Code: Use the Command-Line Interface

We've already seen that to code better every day you need to practice: code, code, and code. And what's the purpose of code? To solve problems and create solutions that can benefit people. Many developers don't even think about it, but we have a development environment available literally at our fingertips: the command-line interface (CLI). The CLI gives you the ability to control your whole environment. Menus and icons of tools, the IDEs, and the operating systems give you easy access only to what is visible. The CLI gives you access to everything.

Do you think of the CLI as a development environment? We do. You have to deal with the syntax and know the statements to execute. There is data and state, and you can process information. You even get unexpected results, and interpreter errors. That sounds exactly like coding!

The CLI provides a way to code and create automated software solutions. We can use it to create even more code: it boosts our productivity through automation. After all, this is how software enables our lives.

To be effective, the CLI requires you to learn, experiment, and remember what is available and how to use it. Get proficient with it. Experiment every day with new things you can do on the command line. And try to automate actions you need to perform repeatedly.

ACTION ITEM: Explore how many steps of your daily routine you can do through the CLI.

ACTION ITEM: Join some of those steps into scripts that automate parts of your work.

BRUNO SOUZA PHOTOGRAPH
BY BOB ADLER/GETTY IMAGES

ACTION ITEM: Play with tools such as sed, awk, and grep to unleash some of the CLI's power.
BONUS ITEM: Find a tool you already use that has a good CLI (such as Jenkins and AWS). Experiment to see how far can you go in using the CLI.

Community: Speak at an Event

We all learn from content that other developers created—videos, webinars, blogs, talks. Usually these are the developers we look up to. The ability to communicate clearly is a terrific asset for your career, as is the wisdom to focus on and deliver pertinent and interesting information.

Presenting ideas using slides, video, or text brings immediate results. You'll be astonished at how much you learn just by preparing your presentation. Research shows that teaching something is the most effective way to learn. You'll need to study and research your topic to become a confident public speaker. And by doing so, you become known as an expert on that particular subject. You'll have the satisfaction of helping others, and also the pride of professional accomplishment.

Even with these benefits, presenting a talk is still intimidating. To conquer your fear, remind yourself that presenting is a big challenge for most people. Even seasoned speakers get butterflies before walking on stage. Don't get discouraged.

You can reduce the fear by reducing the challenge. Start small. Present something you learned last week to your coworkers during a quick lunchtime meetup. Or, present something you know well, such as a technology or a specific tool, to students taking a computer science course. Later, maybe you'll decide to submit your talk to your local Java user group.

A neat trick to remember: When people attend your talk, it is because they want to learn about your topic. You probably know more about it than they do; otherwise, they would be doing something else. Remember that the value you bring to your audience is your experience.

It isn't difficult to steer your career in the right direction. Keep your mind open to new possibilities, experiment with new ideas, and engage with the community.

Presenting talks and writing blog posts and articles are acquired skills that you need to develop. By starting with small, easy steps, you will boost your confidence and benefit your career. This is an attitude change that will last your whole life.
ACTION ITEM: Present the slides of a presentation you watched to your coworkers.
ACTION ITEM: Prepare a short talk that will be useful to your friends.
ACTION ITEM: Turn those slides into a blog post or propose to present them to a local user group, university, or technical school. Maybe you'll decide to record your talk and post it on YouTube.
BONUS ITEM: Experiment with getting this talk approved. Submit it to a conference or tradeshow, even if the event is one that you don't expect to accept it—maybe an event in another country. Add your blog post or your recorded presentation as supporting material.

DevOps: Start Now

Software development is all about creating something new. It is about looking to the future and shaping new functionalities out of thin air. That's why great developers are open-minded. They experiment. And some ideas or technologies really help: they push the boundaries of our thinking. DevOps is one of those attitude-changing ideas.

As developers, we need to take responsibility for the software we build, and review our definition of "done." Software is not truly done until the end user benefits from it. Software development should be all about delivering value to end users. What good is code sitting idle in a repository somewhere? There are too many reasons why we allow this: It isn't ready. It has bugs. It needs improving. It needs to perform better. These are recurring fears that can serve as excuses on every project.

DevOps is the idea that Developers and Operations (every-

one, in fact) work together to deliver software. DevOps gets us to think differently. It makes delivering value to the users our top priority. When the whole company works together toward the goal of people using the software, we focus more on the benefits to our users and less on the fears that prevent us from doing it. We become better developers.

DevOps is an all-encompassing skill. It requires us to improve everywhere—from our tools and collaboration to our code and tests. And it is easy to start. Think big, but start small. Change your attitude. Focus on automation and delivery. Others will follow.

ACTION ITEM: Automate something that you do manually, such as your build, your deployment, or one of your high-level tests and test suites.

ACTION ITEM: Define something useful to measure, such as the number of bugs in the project, or how long it takes to deploy a new version. Start tracking this for the next few weeks. Add new metrics later.

BONUS ITEM: Learn one automation tool that you currently don't use, such as Jenkins, Ansible, Chef, or Selenium. Go beyond the tutorial: Run a small but real test to automate one thing in your project.

Pulling It All Together

The skills described previously are easily combined. Using the CLI to automate tasks is a useful DevOps skill. Spending time automating your deployment will not only get you moving going in the right direction, but it will also make your life easier. Consider delivering a brief presentation to your coworkers demonstrating what you did. You can then pass along this important skill by presenting the same talk to students in a nearby university.

It isn't difficult to steer your career in the right direction. Keep your mind open to new possibilities, and experiment with new ideas. Engage with the community in ways that are positive for you and others. And keep coding! `</article>`

Comments

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

Article Proposals

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

Customer Service

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

Where?

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

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