

# Java<sup>TM</sup> magazine

By and for the Java community 

## UI Tools

17

SCRIPTING  
JAVAFX WITH  
FXML

24

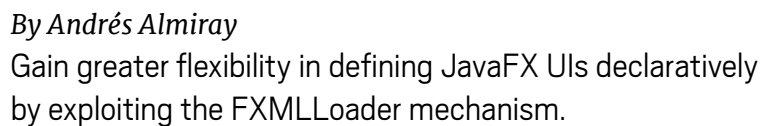
VISUAL DESIGN  
WITH SCENE  
BUILDER

30

MVC 1.0: BUILDING  
WITH THE MVC  
ARCHITECTURE

40

WEB UI  
CONSTRUCTION  
WITH ORACLE JET



## 04 From the Editor

The demanding economic model for JVM languages to be successful means that few will ever be widely adopted.

## Comments, questions, suggestions, and kudos

## 14 Review

Review of Java 8 refactoring video

*By Johan Vos*  
The interactive UI design tool originally developed by Oracle continues to advance in the open source community.

*By Josh Juneau*  
A look at a remarkably flexible  
framework that builds on JAX-RS

By John Brock  
Using Oracle's open source  
JavaScript toolkit

Use the principal IoT messaging protocol to asynchronously send and receive data from devices—in this case, from drones.

*By Ben Evans*  
Need smaller executables? Migrate to Java 9. Can't migrate? Then consider Java 8's Compact Profiles.

*By Peter Ledbrook*  
As builds become more complex—consider monolithic repos—library dependencies present a special challenge that build tools such as Gradle are working at solving.

## 66

### Fix This

By Simon Roberts

Our latest code quiz

## 65 Java Proposals of Interest

## **71** **Contact Us**

Have a comment? Suggestion?  
Want to submit an article proposal?  
Here's how.



Stephen Chin

Kathy Cygnarowicz

Jennifer Kurtz

Contact your sales representative.

+1.888.283.0591 (US)

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

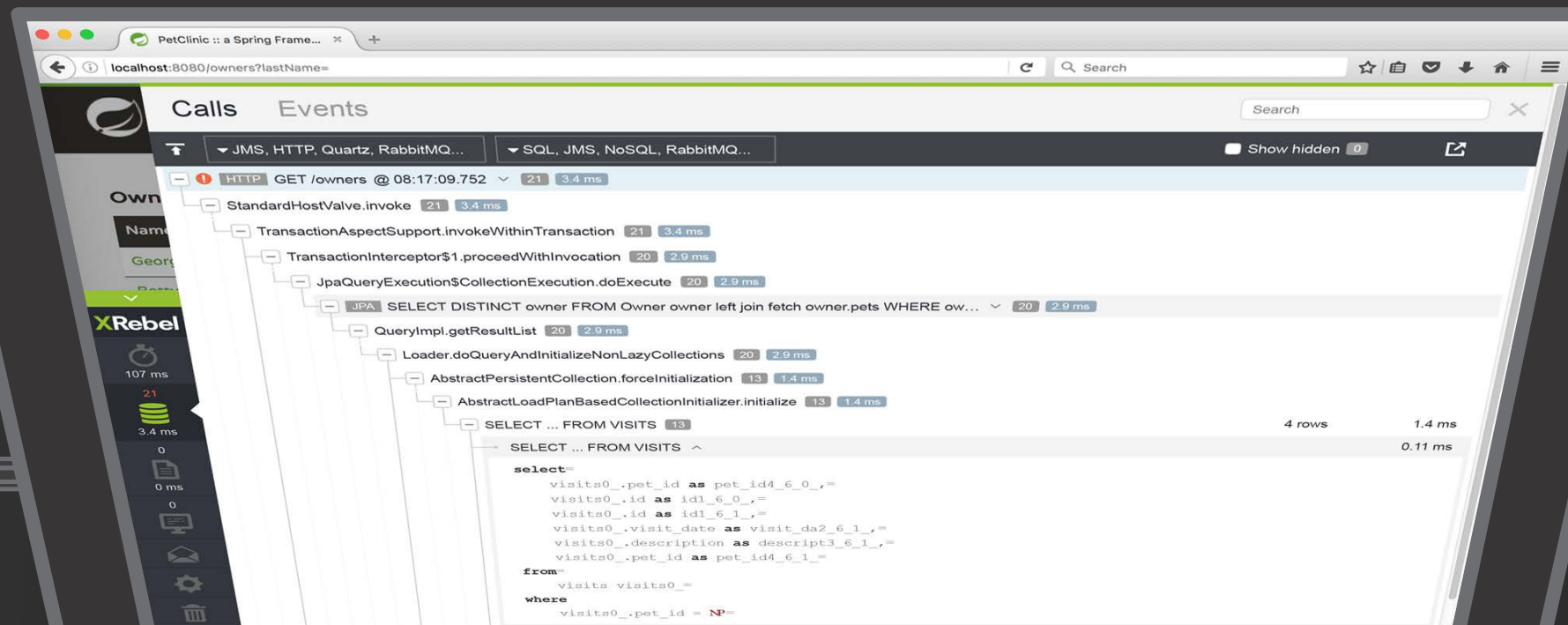
JAVA MAGAZINE READERS GET  
**20% OFF** WITH CODE **JAVA**

**Performance-Focused Developer Tool** lets you see request breakdowns, IO queries, logs and exceptions without leaving the window or your webapp.

**14 DAY FREE TRIAL!**

**XRebel**

Real time insight,  
faster apps





A middle-aged man with glasses, wearing a light blue button-down shirt and blue jeans, is walking towards the camera on a city street. He is holding a grey folder or tablet under his left arm. The background is a blurred city street with other pedestrians and buildings.

A viable business model is key to language adoption.

JVM languages generally fall into two major categories: those that are ports of existing lan-

Because of our long coverage of JVM lan-

PHOTOGRAPH BY BOB ADLER/VERBATIM

A vertical banner with a dark blue background featuring diagonal stripes. At the top is a red rectangle with the word "ORACLE" in white. Below it is a white icon of a cloud containing a code editor window with the symbols "</>". The main text is in large white font: "The Best Resource for Modern Cloud Dev". Below this is a paragraph in white: "The Oracle Developer Gateway is the best place to jump-start your modern cloud development skills with free trials, downloads, tutorials, documentation, and more." Further down is another paragraph in white: "Trials. Downloads. Tutorials. Start here: developer.oracle.com". At the bottom, there are two white boxes: the first contains "developer.oracle.com" in red, and the second contains "#developersrule" in dark blue.

guages, I am occasionally asked which of them will become popular enough to “cross the chasm.” This term, which originated in Geoffrey Moore’s book of the same name, refers to an increase in popularity that drives a technology from the exclusive domain of visionaries and early adopters into the wider embrace of pragmatists and especially of businesses. I believe there are only three languages that are capable of this crossing or have already done so: Groovy, Scala, and Kotlin.

Groovy found success as a quirky scripting language that has filled numerous niches where quick but expressive coding is needed. It is the scripting language for many testing frameworks and is used for writing build scripts in Gradle. It is also unique among the primary JVM languages (the three mentioned above plus Java) in that it did not require corporate sponsorship to become popular. (Even though Pivotal did support it for a few years, Groovy was popular long before Pivotal's acquisition and has continued to be since Pivotal stopped sponsorship.) This is testament to the community skills of the project's longtime leader, Guillaume Laforge.

Today, no language can hope

to cross the chasm as Groovy did—that is, without serious financial backing. Writing a language is a very expensive proposition, as is promoting it. While originally an academic creation, Scala was backed by the startup Typesafe until the company realized—as Pivotal did with Groovy—that there is no revenue to be made in selling a new language. As a result, Typesafe changed its name to Lightbend and refocused on its nonlanguage products. The break from being the “Scala company” was so clean that the [press release](#) announcing the name change did not even mention the language in the body of the announcement. As I said, there’s just no money in languages.

Kotlin relies on a rather different model. The language was devised in part for JetBrains' internal use. Its design is pragmatic and aimed at helping the company reduce costs in developing its extensive line of developer tools. The benefits of developing and promoting Kotlin outweigh its costs and, crucially, JetBrains derives its income from products other than Kotlin. The costs, however, are significant. According to Andrey Breslav at JetBrains, more than two dozen full-time equivalent

lents are developing and promoting Kotlin.

In the process, Kotlin has morphed into more than just an efficiency tool for JetBrains. Its intensely pragmatic orientation has strongly resonated with a significant and active community, which accelerates its movement across the chasm. Kotlin thereby enables JetBrains to bring new developers into its tool ecosystem. But the growing user base also presents the company with the challenge that successful languages often face: managing the demands of users versus the company's own desires for the language.

Because economics support Kotlin's evolution and JetBrains' longstanding knowledge of developers will help it work with the community, I expect that within the next few years Kotlin will fully cross the chasm and emerge as a—or possibly *the*—primary non-Java JVM language, so proving yet again the robustness of the JVM ecosystem.

**Andrew Binstock, Editor in Chief**  
[javamag\\_us@oracle.com](mailto:javamag_us@oracle.com)  
[@platypusguy](https://twitter.com/platypusguy)



# Level Up at Oracle Code

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

**Get on the list  
for event updates:**

**go.oracle.com/oracledoradshow**

**developer.oracle.com**

#developersrule







JANUARY/FEBRUARY 2017

## JavaScript in Java Magazine

We received more than 30 replies to our request in the January/February issue ([“The Polyglot Future”](#)) for comments about regular coverage of JavaScript. The replies boiled down to three principal points of view, as articulated in the following three notes.

## Lots of It!

Yes, please include a regular column on JavaScript. It can be arbitrarily large!

I have been a Java programmer for 16 years. Two years ago, I landed in a job that required client-side JavaScript coding. After a frustrating six months, I learned to respect JavaScript: it is a language and an ecosystem capable of producing quality software and fantastic tooling.

Despite seeing the elegance of libraries and performance of runtimes like Node, I am wondering how “large”-scale development is possible with a language that does not have interfaces, like Java has.

I feel really curious to learn how Java programmers with a similar mindset find their place in a world without interfaces and without a threading API (another topic that fascinates me).

—Csaba Koncz

## Some JavaScript, Please

My first thought after reading your editorial was, “I’d read it, but I don’t know how much other JVM developers want to hear about the JavaScript ecosystem.” But the more I think about it, the more I think it would be a disservice *not* to cover JavaScript. There’s a fine line between focusing on one thing and pretending everything else doesn’t exist.

That's especially true when JVM-focused lan-

guages are compiling to JavaScript. It has been exactly two years since Scala.js went from a science project to something the Scala community is actively promoting. And Kotlin is gunning to be both a Java replacement and a TypeScript replacement, which has meant making Kotlin's type system able to impersonate JavaScript.

I'm not an Android developer, but my wife is an iOS developer who works closely with Android and web developers to keep their code in sync. Naturally they use JavaScript on all three platforms, and I've suggested they look into React Native to help unify the codebase. React is something you could cover without even mentioning JavaScript.

If anything, the argument against covering JavaScript is that there's just too much to do it justice. It seems that every time you dip your toe in, some new framework has taken over. Blink and you miss it.

I'd be fascinated to see a deep dive comparing the tradeoffs between JITs from HotSpot, Dalvik, and the various JavaScript JITs. But maybe that's just me.

—David Leppik

## No JavaScript at All

I suggest “Absolutely None” for the introduction of other programming languages in *Java Magazine*. At the same time, I suggest another magazine that covers the polyglot issues discussed in the editorial, with topics involving basic and advanced programming integrated with some database, such as Oracle and MySQL, as well as other programming languages.

—Marcos André Pisching

Professor de Informática no Campus Lages do IFSC  
Brazil







# IntelliJ IDEA

Making development\*  
an enjoyable experience

Get it now

JET  
BRAINS



\*Java, Groovy, Kotlin, Scala, Android and much more





## **GREAT INDIAN DEVELOPER SUMMIT**

APRIL 25–28

BANGALORE, INDIA

The Great Indian Developer Summit (GIDS), now in its 10th year, offers four days of content grouped by theme. April 26 focuses on Java and JVM languages. Other days focus on web, mobile, DevOps, and big data. Register for each day separately.

## **JavaLand**

MARCH 28–30

BRÜHL, GERMANY

This annual conference features more than 100 lectures on subjects such as core Java and JVM languages, enterprise Java and cloud technologies, IoT, front-end and mobile computing, and much more. Scheduled presentations include “Multiplexing and Server Push: HTTP/2 in Java 9,” “The Dark and Light Side of JavaFX,” “JDK 8 Lambdas: Cool Code that Doesn’t Use Streams,” “Migrating to Java 9 Modules,” and “Java EE 8: Java EE Security API.”

## **O’Reilly Software**

### **Architecture Conference**

APRIL 2–3, TRAINING

APRIL 3–5, TUTORIALS

AND CONFERENCE

NEW YORK, NEW YORK

This event promises four days of in-depth professional training that covers software architecture fundamentals; real-world case studies; and the latest trends in technolo-

gies, frameworks, and techniques. Past presentations have included “Introduction to Reactive Applications, Reactive Streams, and Options for the JVM,” as well as “Microservice Standardization.”

## **JAX DevOps**

APRIL 3 AND 6, WORKSHOPS

APRIL 4 AND 5, CONFERENCE

LONDON, ENGLAND

This event for software experts features in-depth knowledge of the latest technologies and methodologies for lean businesses. The focus is on accelerated delivery cycles, faster changes in functionality, and increased quality in delivery. Conference tracks include agile and company culture, cloud platforms, container technologies, continuous delivery and automation, microservices, and real-world case studies. The conference is preceded and followed by a day of workshops. There’s also a two-in-one conference package that provides free access to a parallel conference, JAX Finance.





## Devoxx France

APRIL 5, WORKSHOPS  
APRIL 6–7, CONFERENCE  
PARIS, FRANCE

Devoxx France presents workshops, tutorials, and keynotes from prestigious speakers, followed by a cycle of eight mini conferences every 50 minutes. You can build your own calendar and follow the sessions as you wish. Founded by developers for developers, Devoxx France covers

topics ranging from web security to cloud computing. (No English page available.)

## IoT Tech Day 2017

APRIL 19  
UTRECHT, THE NETHERLANDS  
Machine learning and AI, security, wearables, and other smart technologies are among the topics investigated in Europe's biggest IoT-centered conference. One timely track, "Connected Living

and Office," examines all aspects of the smart home or office: control of lighting, temperature control, appliances, and security locks for gates and doors. The Things Network—a global community whose mission is to build a global IoT data network—is hosting its popular hackathon at the conference again this year.

## Java Day Istanbul 2017

MAY 6  
ISTANBUL, TURKEY  
With the slogan "By developers, for developers," this conference organized by the Istanbul Java User Group explores Java, web, mobile, big data, cloud, DevOps, and more. It also provides the opportunity for developers to network with tech companies and startups.

## JAX 2017

MAY 9–11, CONFERENCE  
MAY 8 AND 12, WORKSHOPS  
MAINZ, GERMANY  
More than 200 internationally renowned speakers give practical and performance-oriented lectures on topics such as Java, Scala, Android, web technologies, agile

development models, and DevOps. Workshops are offered on the day preceding and the day following the conference. (No English page available.)

## Devoxx UK

MAY 11–12  
LONDON, ENGLAND  
Devoxx returns to the UK with a focus on Java, web, mobile, JVM languages, architecture, big data, and security. Attracting more than 1,200 attendees, the conference includes more than 120 sessions, with 50-minute conference sessions, three-hour hands-on labs, and many quickie presentations.

## Riga Dev Days 2017

MAY 15–17  
RIGA, LATVIA  
The biggest tech conference in the Baltic States, this three-day event is a joint project of Google Developer Group Riga, Java User Group Latvia, and Oracle User Group Latvia. By and for software developers, Riga Dev Days focuses on the most-relevant topics and technologies for that audience with more than 50 sessions on Java, web, and cloud programming.





Since 2001, the No Fluff Just Stuff (NFJS) Software Symposium Tour has delivered more than 450 events with more than 70,000 attendees. This event in Boston covers the latest trends within the Java and JVM ecosystem, DevOps, and agile development environments.

REGISTER NOW | [WWW.DEVOXX.CO.UK](http://WWW.DEVOXX.CO.UK)



**DEVOXX**<sup>TM</sup>  
United Kingdom  
a developer community conference  
**11th & 12th MAY, 2017**



**1200  
DEVELOPERS**



**100+  
SPEAKERS**



**110  
SESSIONS**



**11  
TRACKS**





## REFACTORING TO MODERN JAVA: GETTING THE MOST FROM JAVA 8

(LiveLessons video)

By Trisha Gee

Pearson/InformIT LiveLessons

Although this column regularly focuses on books, in this issue we look at a training video sold commercially by the InformIT division of Pearson, the company behind Addison-Wesley and other respected imprints.

This video is a downloadable product (priced around US\$100) that consists of DRM-free MP4 files. All told, they represent several hours of high-quality instruction. I watched the videos on my desktop and opened the window to fill the screen so that I could read the code and see the changes easily.

The lecturer is Java Champion Trisha Gee, who is a frequent and well-respected speaker at conferences. Here she presents a variety of refactorings that are available because of the innovations introduced in Java 8. Gee focuses primarily on lambdas, streams, Optionals, and several lesser features. To follow along, you will need to understand these features. The video is squarely aimed at intermediate to advanced

developers, and if you don't know the syntax for lambdas or understand what functional interfaces are, you'll quickly find yourself stumbling as you try to follow along. In this sense, the video is tremendously satisfying precisely because it avoids introductory material. That is, you must know Java 8 going in.

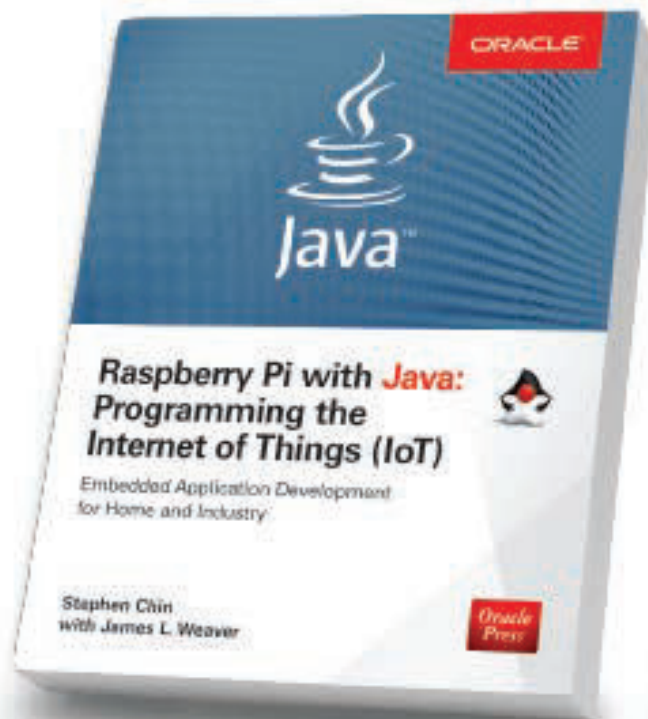
After presenting the refactorings, Gee cleverly spends a significant amount of time assessing the performance impact of the code changes. She does this using the Java Microbenchmarking Harness ([JMH](#)) and shows that some but not all refactorings improve performance. It's kind of a fascinating process to watch.

Despite the high production values of the video and Gee's practiced delivery (she never stutters or repeats and easily moves backward and forward through subjects to explain a given topic), the video has a few aspects that could bear improvement. At times Gee speaks very fast, which makes it hard to

fully understand her point because you're still trying to catch up with what she said 15 seconds ago. My other complaint is that Gee lets the IDE (IntelliJ) find refactorings and implement them, so they're too often done in the blink of an eye. She helpfully shows before and after code in some examples, but it would be easier to follow if we saw her make the changes manually rather than see code instantly transformed by a keystroke. In this regard, although Gee claims that all the refactorings are available in other IDEs, I found that some were not available in NetBeans. I did not verify Eclipse.

Overall, my reservations are about details, not about content, which is uniformly highly instructive and full of immediately applicable information. Gee has posted a less intense and shorter version of this video on [YouTube](#), so you can sample the goods before buying this. I expect you'll be as impressed as I am.—*Andrew Binstock*

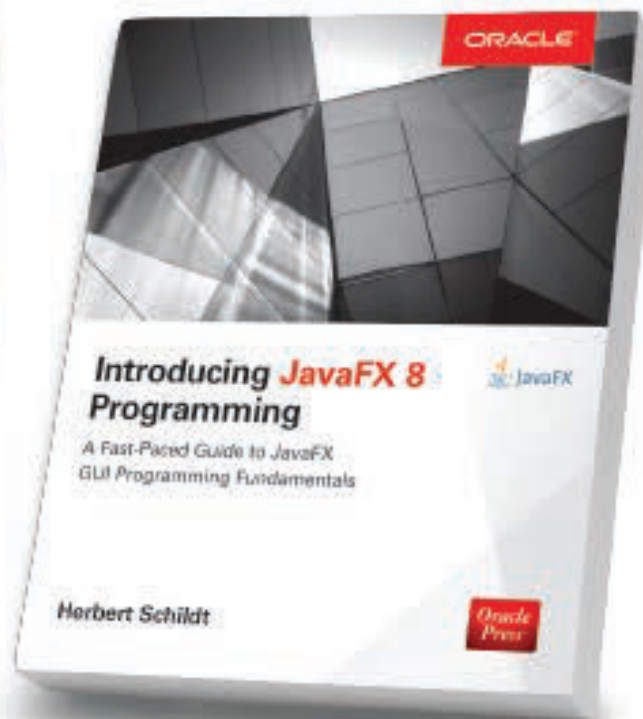
Written by leading Java experts, Oracle Press books offer the most definitive, complete, and up-to-date coverage of Java available.



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

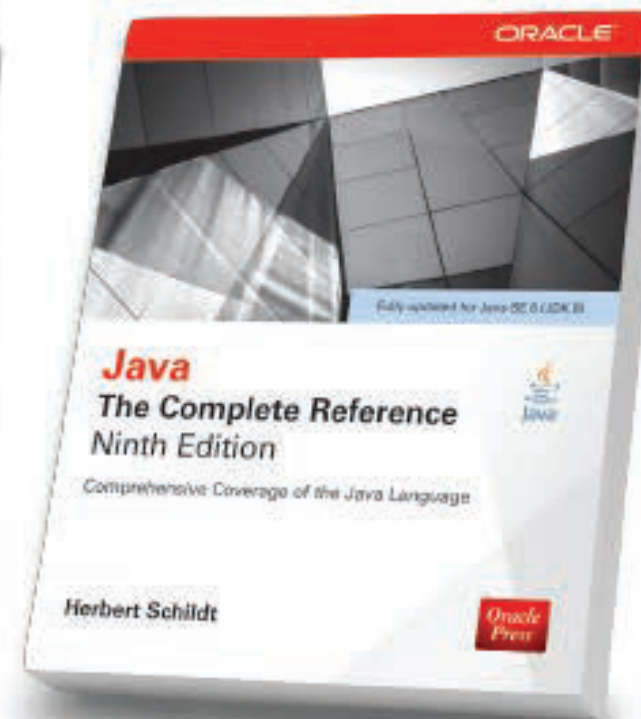
**Stephen Chin, James Weaver**

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



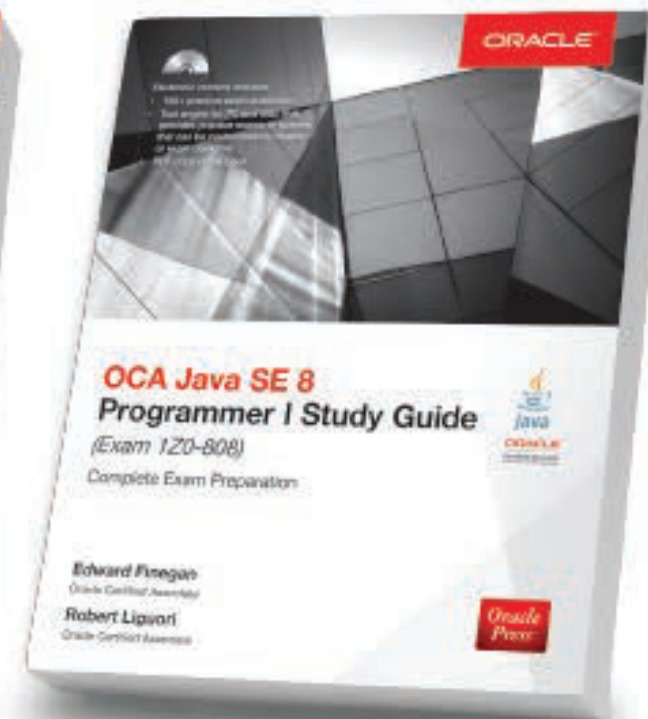
## **Introducing JavaFX 8 Programming** **Herbert Schildt**

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.





ANDRÉS **ALMIRAY**

The JavaFX UI toolkit delivers a refreshing and modern set of APIs that can be used to build desktop and mobile applications that target the JVM. Similar to its predecessor, Swing, JavaFX provides a standard widget set, as well as the means to extend this widget set with custom components and new behavior.

The following benefits are some of the advantages of choosing FXML over the programmatic API:

- You have probably seen FXML before, but in case you haven't, here's a quick sample. The following code snippet defines a grid in which six UI elements are placed in a two-column layout:

```
<?xml version="1.0" encoding="UTF-8"?>
<?import javafx.scene.control.Button?>
<?import javafx.scene.control.Label?>
<?import javafx.scene.control.PasswordField?>
<?import javafx.scene.control.TextField?>
<?import javafx.scene.layout.GridPane?>
<GridPane>
    <Label text="Username:"
        GridPane.columnIndex="0" GridPane.rowIndex="0"/>
    <TextField
        GridPane.columnIndex="1" GridPane.rowIndex="0"/>
    <Label text="Password:"
        GridPane.columnIndex="0" GridPane.rowIndex="1"/>
    <PasswordField
        GridPane.columnIndex="1" GridPane.rowIndex="1"/>
    <Button text="Cancel"
        GridPane.columnIndex="0" GridPane.rowIndex="2"/>
    <Button text="Login"
```



```
GridPane.columnIndex="1" GridPane.rowIndex="2"/>
</GridPane>
```

Figure 1 shows how the application looks when running.

Note that, as in Java code, you must specify import statements for the types that are used in the FXML file. These import statements serve as a hint to the FXML loading mechanism that's in charge of interpreting the declarative UI description and turning it into a proper SceneGraph with the UI elements in it. This loading mechanism is known as `FXMLLoader`. Using `FXMLLoader` is straightforward, as shown by the following code:

```
sample/App.java
package sample;

import javafx.application.Application;
import javafx.fxml.FXMLLoader;
import javafx.scene.Scene;
import javafx.stage.Stage;

import java.net.URL;

public class App extends Application{
    @Override
    public void start(Stage stage) throws Exception {
        URL fxml = getClass().getClassLoader()
            .getResource("sample/app.fxml");
        FXMLLoader fxmlLoader = new FXMLLoader(fxml);

        stage.setScene(new Scene(fxmlLoader.load()));
        stage.sizeToScene();
        stage.show();
    }
}
```

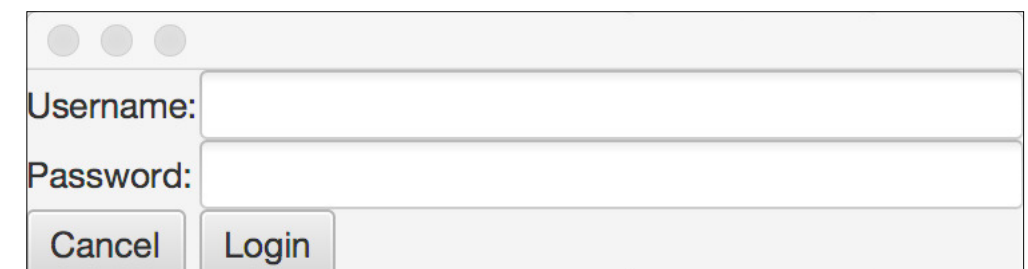
For now, you only need to tell FXMLLoader the location of the FXML resource that you want to load. It's important to remember that the action of creating UI elements and attaching them to the SceneGraph must happen inside the UI thread (known as the *FX application thread*). Bad things can happen when you do not follow this rule. Fortunately, the base type `Application` provides a basic lifecycle that ensures code can be called inside such a thread. In this particular case, the `start` method is guaranteed to be called inside the UI thread, which means everything is OK.

## Node Properties

If you look closely at the FXML snippet in the first example, you'll see that the FXML nodes `Label` and `Button` define a text attribute. This attribute is in turn mapped to a JavaBean property found in the matching type. Thus when `FXMLLoader` instantiates the first `Label` it encounters, it sets the label's text property to the exact value of the text attribute. In other words, it's as if `FXMLLoader` invoked the following code on your behalf:

```
Label label = new Label();
label.setText("Username:");
// insert label into SceneGraph
```

That's quite a short snippet, and this functionality doesn't seem to be much of an advantage right now; however, take into consideration that UI elements may have several prop-



**Figure 1. Login screen using FXML**





```

        .addListener((ov, oldValue, newValue) -> {
            if (control.getText().length() > maxLength) {
                String s = control.getText()
                    .substring(0, maxLength);
                control.setText(s);
            }
        });
    }

    public static int getMaxTextLimit(
        TextInputControl control) {
        Object value =
            control.getProperties().get(LIMIT);
        if (value instanceof Number) {
            return ((Number) value).intValue();
        }
        return -1;
    }
}

```

Note that this code does not take into account any invalid values (such as `maxLength` being equal or less than zero), nor does it provide the means to unregister the listener if the `setMaxTestLimit` method is called more than once with the same target node. This code is for illustration purposes and should not be used “as is” in production without additional tweaks.

Moving on, you only need to update the FXML definitions to look like this:

```
sample/app.fxml
<?xml version="1.0" encoding="UTF-8"?>
<?import javafx.scene.control.Button?>
<?import javafx.scene.control.Label?>
<?import javafx.scene.control.PasswordField?>
<?import javafx.scene.control.TextField?>
```

```
<?import javafx.scene.layout.GridPane?>
<?import sample.InputControlUtils?>
<GridPane>
    <Label text="Username:"
        GridPane.columnIndex="0" GridPane.rowIndex="0"/>
    <TextField
        GridPane.columnIndex="1" GridPane.rowIndex="0"
        InputControlUtils.maxTextLimit="10"/>
    <Label text="Password:"
        GridPane.columnIndex="0" GridPane.rowIndex="1"/>
    <PasswordField
        GridPane.columnIndex="1" GridPane.rowIndex="1"
        InputControlUtils.maxTextLimit="10"/>
    <Button text="Cancel"
        GridPane.columnIndex="0" GridPane.rowIndex="2"/>
    <Button text="Login"
        GridPane.columnIndex="1" GridPane.rowIndex="2"/>
</GridPane>
```

There are two other features to be mentioned when dealing with properties in FXML. The first is the ability to define the name of a default property, which allows you to omit the property name when defining the hierarchy. The name of the property must be defined in code by applying the `@javafx.beans.DefaultProperty` annotation to the target type. Many of the UI elements in the standard JavaFX widget set use this annotation. For example, the `javafx.scene.layout.Pane` type, which is the base type for many widget containers, is annotated with `@DefaultProperty("children")`. This allows you to write FXML as shown in the previous example instead of this:

```
sample/app.fxml
<?xml version="1.0" encoding="UTF-8"?>
<?import javafx.scene.control.Label?>
<?import javafx.scene.control.TextField?>
```



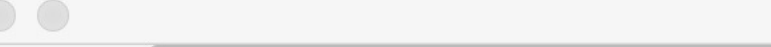


You also must change the FXML file to reflect the keys of the properties you just defined. The convention is to use % as a prefix to distinguish a key value from a literal value. Your IDE might even be able to suggest key completions.

```
<Label text="%username.label"
    GridPane.columnIndex="0" GridPane.rowIndex="0"/>
<TextField
    GridPane.columnIndex="1" GridPane.rowIndex="0"
    InputControlUtils.maxTextLimit="10"/>
<Label text="%password.label"
    GridPane.columnIndex="0" GridPane.rowIndex="1"/>
<PasswordField
    GridPane.columnIndex="1" GridPane.rowIndex="1"
    InputControlUtils.maxTextLimit="10"/>
<Button text="%action.cancel.label"
    GridPane.columnIndex="0" GridPane.rowIndex="2"/>
<Button text="%action.login.label"
    GridPane.columnIndex="1" GridPane.rowIndex="2"/>
```

If you run the application again, you should see the same visuals as in **Figure 1**. Now, change the code to load the German version of the properties file, and you'll notice the application looks like **Figure 2**.

Great! These are the first steps to get localization support in an application. As mentioned earlier, you can create the `ResourceBundle` in many ways that are commonly used in regular Java programming. A further step would be to make



Benutzername:

Passwort:

Abbrechen Anmeldung

**Figure 2. German login screen**

```
username.label=Benutzername:  
password.label=Passwort:  
action.cancel.label=Abbrechen  
action.login.label=Anmeldung
```

FXML provides a quick and easy way to define UIs declaratively. The tooling support (IDEs and SceneBuilder) is good enough to get you started writing FXML with ease. Many of the node types that you can use inside FXML are straightforward, because they follow the JavaBeans conventions. But when they don't, there's still a way to use them with FXML given that the format provides useful extensions and hints to its loading mechanism, `FXMLLoader`. You can also configure localization concerns on widgets via FXML, enabling a wider audience for your applications. In an upcoming article, I'll examine the mechanism exposed by `FXMLLoader` and FXML to further customize and bind the widgets defined in the FXML file—for example, features such as the `fx:controller` attribute and the `@FXML` annotation. [</article>](#)

**Andrés Almiray** is a Java and Groovy developer and a Java Champion with more than 17 years of experience in software design and development. He has been involved in web and desktop application development since the early days of Java. He is a true believer in open source and has participated in popular projects such as Groovy, JMatter, Asciidoctor, and others. He's a founding member and current project lead of the Griffon framework, and he is the specification lead for JSR 377.

## Oracle's tutorial on FXML



Practical training in the tools, techniques,  
and leadership skills you need for the evolving  
world of software architecture.

**April 2-5, 2017**  
[softwarearchitecturecon.com/ny](http://softwarearchitecturecon.com/ny)

JAVA MAGAZINE READERS GET  
**20% OFF** WITH CODE **JAVA**





The interactive UI design tool originally developed by Oracle continues to advance in the open source community.

## About Scene Builder

In practice, many applications contain both JavaFX code and FXML code. The JavaFX APIs and the FXML constructs are designed to work together.

On the JavaFX code side, the [FXMLLoader](#) class loads an FXML file from the JAR containing the application file or from the classpath:

The connection between your Java code and the UI elements that are declared in the FXML file is made by a controller class. This is a regular Java class that may contain annotations linking the UI elements to Java classes. This approach separates the UI declaration from the behavior of the application, while still allowing the application to access the UI elements directly.

Although the FXML file can be edited within any IDE as a regular XML file, this practice is not recommended, because the IDE provides only basic syntax checking and autocomple-

installing it on Windows. (This is a new feature available in Scene Builder 8.3.0.) Once you have installed Scene Builder, open your IDE so you can set its location and you can open any FXML file from your IDE:

- On NetBeans, select Tools -> Options (or Preferences on Mac) -> Java -> JavaFX, and click Browse to find the main Scene Builder folder.
- On IntelliJ, select File -> Settings (or Preferences on Mac)

- On NetBeans, select Tools -> Options (or Preferences on Mac) -> Java -> JavaFX, and click Browse to find the main Scene Builder folder.
- On IntelliJ, select File -> Settings (or Preferences on Mac)



25





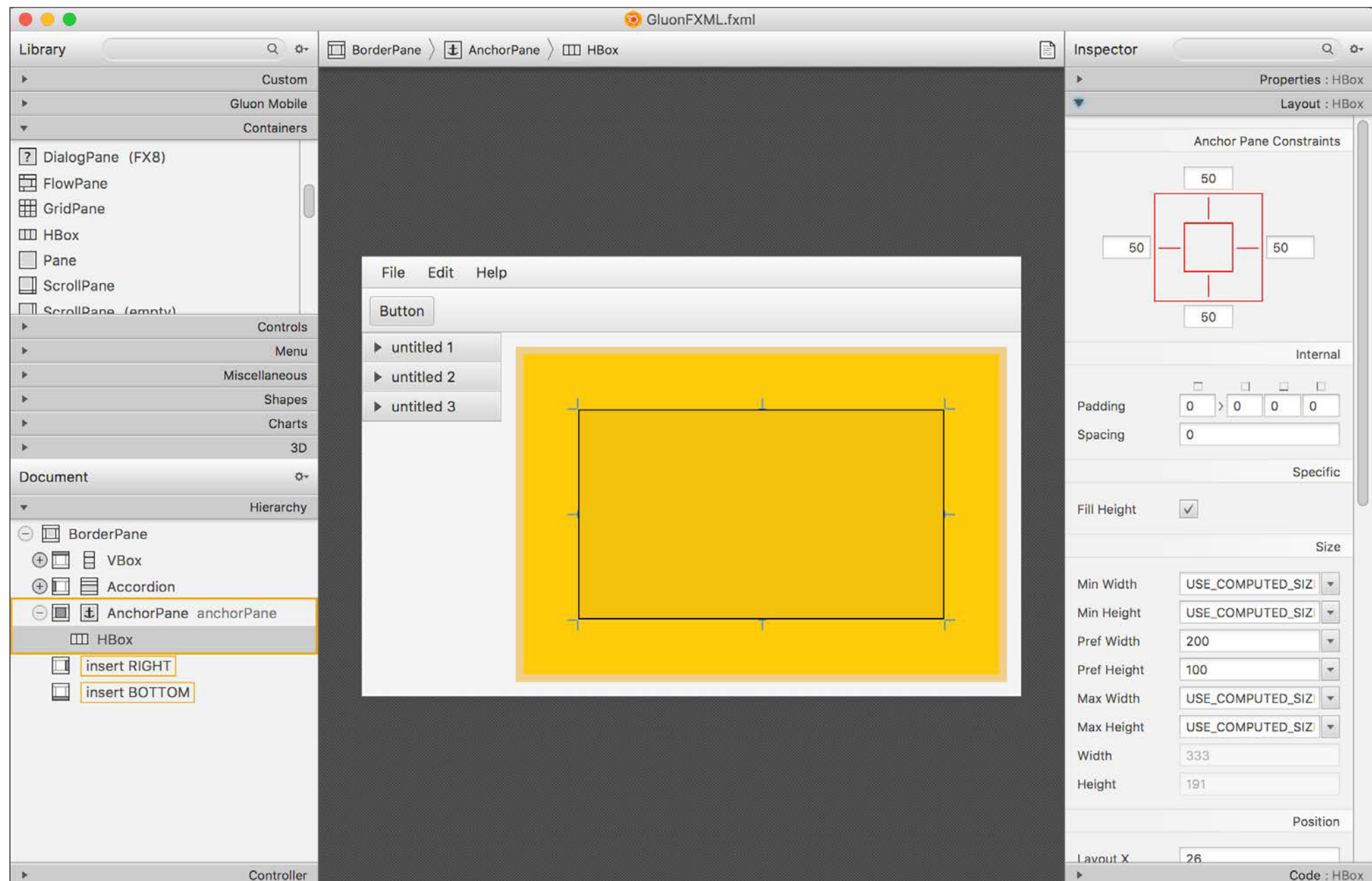
```
private ScrollPane scrollPane;

@FXML
private Avatar avatar;

}
```

You can easily add CSS to the scene by providing a CSS file that can be included using the Stylesheets option in the Properties panel on the right. You can add inline styling also, by providing style rules to any node on the scene. You can apply new or existing style classes to any node as well.

Features related to layout, such as position, dimensions, margin, padding, and transforms (translation, rotation,



**Figure 2.** Defining anchor pane constraints in the Layout panel



scaling, and so forth), can be set in the Layout right panel, as shown in Figure 2.

At any moment, you can preview the created scene by clicking Preview -> Show Preview in Window. A resizable dialog box with the designed scene will be shown. By resizing it, you can make sure every node behaves as expected.

## Integrate the Basic Interface in a Java App and Show It

Once the FXML is ready, you can integrate it into your Java application by calling `FXMLLoader` to load it. Here's how:

```
public class GluonSceneBuilder extends Application {

    @Override
    public void start(Stage stage) throws Exception {
        Parent root = FXMLLoader.load(getClass()
            .getResource("GluonFXML.fxml"));

        Scene scene = new Scene(root);
        stage.setScene(scene);
        stage.show();
    }
}
```

In the controller class, you can add the required action handlers and the response to the user interaction. You can create new controls as well, and combine them with those injected by the `FXMLLoader`. Notice that for the controls you add programmatically, you need to create new instances. This is not needed for controls that are declared in the FXML file, because the `FXMLLoader` already creates them for you. The code snippet below shows a piece of a controller class that works with two controls: an `HBox` control defined in the FXML file, and a `Label` that is not created in the FXML file. The `HBox` instance does not need to be created in the controller class, but the `Label` instance does.

```
@FXML
private HBox hBox;

private Label label;

public void initialize() {
    label = new Label();
    hBox.getChildren().add(label);
    titledPane1.expandedProperty()
        .addListener((obs, ov, nv) -> {
            if (nv) {
                label.setText("TitledPane1");
            }
        });
    . . .
}
```

In a controller class, you can annotate not only fields representing controls with the `@FXML` annotation, but also methods. As an example, I define the following event handler in the controller:

```
@FXML
void buttonClicked(ActionEvent event) {
    label.setText("Button");
}
```

This event handler simply sets the text of the `Label` to `"Button"`. Because the event handler annotated it with `@FXML`, Scene Builder will find it and can assign it to a corresponding action for a node.

## Integrate the Interface with Your Favorite IDE

Scene Builder is a standalone application. When FXML is being edited outside Scene Builder (by directly modifying it in your IDE, for example), Scene Builder reacts to the changes





## A look at a remarkably flexible framework that builds on JAX-RS

MVC 1.0 was considered for inclusion in Java EE 8 (as JSR 371). Since that time, Oracle has instead sought to move control of the project to the community. In this article, I examine this new framework and explain why I like it, even though there are many other great choices. I also develop an application from the ground up using MVC 1.0 in the NetBeans IDE, allowing you to follow along each step of the way.

MVC 1.0 does not contain any components, leaving the view completely up to the developer. However, this design does not mean options are not available, because MVC allows just about any view technology to be used to create a front end. The new framework does not contain request phases,

because requests are handled completely by the implementation of controller methods. Given that view options are abundant and the request/response is completely open to custom development, the MVC framework promises to deliver a very flexible web development option. Where JSF is component-based and handles much of the session management for the developer, MVC leaves the developer to create applications using just about any view technology, and it hands over full control of the session management.

## Building a Project with MVC 1.0

The MVC framework reference implementation, named Ozark, was defined in JSR 371. This framework is layered on top of JAX-RS. If you are familiar with JAX-RS, you will find it easy to use. If you are not familiar with JAX-RS, you can learn both MVC and JAX-RS reading this article, because they both function very much the same way.

In this article, I walk through the development of the Duke Issue Tracker application, illustrating the basics of the MVC framework along the way. To follow along, you need to have a solid understanding of how Java EE applications are built. I use NetBeans 8.2 here, which already provides support for MVC 1.0, but you can follow along with the IDE of your choice.

Duke Issue Tracker is used to create and view issues for Duke's application. The initial view of the application lists all of the open issues in the upper half of the UI and displays an issue-creation form on the lower half. An Apache Derby database is used to store the data, and this article demonstrates how to use JAX-RS web services and Enterprise JavaBeans (EJBs) to read, create, and update data.

MVC is not quite a production-ready framework. Even though it is fairly developed at this point, it has not yet been completed, nor has it been released as part of any production application server. This means you will need to obtain the Ozark reference implementation JAR file from the proj-

ect site and deploy it along with your WAR file to a valid container, such as GlassFish 4.1. For the purposes of this example, I deploy to the GlassFish 5 nightly build, and I include Ozark 1.0.0-m02 and MVC API 1.0-edr2 as dependencies in the POM file of the project.

## Server and Project

To get started, create a new NetBeans Maven web application project, and name it DukeIssueTracker. Once the project has been generated, add the required dependencies to the POM, as shown in **Listing 1**. Configure the application to run on JDK 8 and the GlassFish or Payara server of your choice.

### ■ Listing 1.

```
<dependency>
    <groupId>javax.mvc</groupId>
    <artifactId>javax.mvc-api</artifactId>
    <version>1.0-edr2</version>
    <scope>provided</scope>
</dependency>
<dependency>
    <groupId>org.glassfish.ozark</groupId>
    <artifactId>ozark</artifactId>
    <version>1.0.0-m02</version>
    <scope>provided</scope>
</dependency>
```

Next, I have created a small set of tables that needs to be installed into the local Apache Derby database or the database of your choice. To do so, run the `create.sql` file that is packaged with the source code against the database.

**An MVC 1.0 application is simply a JAX-RS application that consists of one or more resources annotated with @Controller.**



The application requires a `beans.xml` Contexts and Dependency Injection (CDI) configuration file to be placed within the `WEB-INF` folder. The `beans.xml` file should configure all beans to be automatically discoverable, as shown in **Listing 2**.

## ■ Listing 2.

```
<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"
        bean-discovery-mode="all">
</beans>
```

The views for the application are placed inside the WEB-INF folder because MVC 1.0 searches for view files there and because the contents of WEB-INF are protected by the container. Create a folder named views within WEB-INF in which to place the view files. For the purposes of this article, I use Facelets for the view engine, but you can opt for JSP or other view languages if you choose.

An MVC 1.0 application is simply a JAX-RS application that consists of one or more resources annotated with `@Controller`. Because the MVC framework is based upon JAX-RS, application configuration must provide the URL pattern that will be used to access the controller methods or JAX-RS resources. I create a new Java class in a package named `org.dukeissuetrackermvc.config`, and I name the class `ApplicationConfig`. The class should extend the `javax.ws.rs.core.Application` class. I annotate the class with `@ApplicationPath("tracker")` to indicate that “tracker” should be applied to the end of the application path URL to access RESTful or MVC resources. Listing 3 contains the complete code for this class.

■ **Listing 3.**

```
package org.mvc.dukeissue trackermvc;

import java.util.HashMap;
import java.util.Map;
import javax.mvc.security.Csrf;
import javax.ws.rs.ApplicationPath;
import javax.ws.rs.core.Application;
// Optionally this configuration can occur within
// the web.xml deployment descriptor
@ApplicationPath("tracker")
public class ApplicationConfig extends Application {
    // Additional configuration code, if needed
}
```

## Entity Classes and RESTful Client

This application uses entity classes that are mapped to the database tables that have been generated for the Apache Derby database. For the purposes of this application, I use the JAX-RS client API to invoke RESTful web services that will provide the data for the application. Use NetBeans wizards to easily generate entity classes from the database and the RESTful application infrastructure.

Once the entity classes and web services have been generated, create a service class that uses a JAX-RS client to read from the web services to obtain data. Create a class in a new package named `org.mvc.dukeissuetracker.service`, and name the class `DukeIssueTrackerService`. This class is a CDI `ApplicationScoped` bean, and it creates a new JAX-RS client in a method annotated with `@PostConstruct` so that the client is generated when the class is constructed. Once the client is created, it will be used to load the data from the RESTful web service into a locally defined list of `DukeIssues` objects. The full listing (due to its length, available from *Java Magazine's* [download area](#)) contains the complete source code for the `DukeIssueTrackerService` class, which also includes

**A controller class is instantiated and initialized for each request,**  
so there is no state saved in between requests.

The `@Path` annotation is used to determine which controller class and method is invoked when the URL including the indicated pattern is navigated to from within a browser. The return type of a controller method returns the String name of the view to which the browser will navigate upon return. The default return type of a controller is `text/html`,

Next, I create a method annotated with `@GET` (from `javax.ws.rs.GET`) that will be used for retrieving all active issues in the tracker. I name the method `displayIssues()` and provide a return type of `String`. The `@GET` annotation is





The `issues.xhtml` view will be displayed in the browser after the `IssuesBean` is loaded with data.

## Models

The *model* of the application refers to the data binding between the data store and the application logic. In the case of MVC 1.0, data can be made available to the application controller, and subsequently to the views, by the injection of a standard `javax.mvc.Models` map or via CDI beans. The preferred technique to make data available is via CDI-based models, which is the technique I use here. However, all MVC 1.0 implementations support the `Models` instance as well.

The `Models` map allows data objects to be made available to the views via a series of key/value pairs in the `Models<String, Object>` map. Much like a standard `HashMap`, the key can be used to access the object, which is the value of the map. In an MVC 1.0 view, the key can be used to obtain access to the data that has been placed into the `Models` object. In the following case, the data would be available via the view by referencing the `dukeIssues` key:

```
...
@Inject
private Models models;

...

public String displayIssues(){
    ...
    models.put("dukeIssues",data);
    return "issues.xhtml";
}
```

The default technique, CDI-based models, relies upon the generation and proper scoping of CDI beans to hold data that will be made available to views. `IssuesBean`, shown in Listing 5, is used by `DukeIssueTracker` to hold the `List<DukeIssues>` object and make it available to the views.

In the case of a CDI bean, the name of the bean or the value provided to the `@Named` annotation is used to reference the data within a view.

## Views

The *view* pertains to the presentation layer of the application or the web views. The MVC framework supports two standard view engines out of the box—JSP and Facelets—although others are available. You can also create a new view engine to support any view technology that is not already covered by generating a view engine that implements the `javax.mvc.engine.ViewEngine` interface. A view engine is used to locate and load views, prepare required models, and render the views back to the client. This application uses the Facelets view engine, along with PrimeFaces UI components.

**Building the view.** I create a folder within the WEB-INF folder and name it `views`. All the views for the application are placed within this folder, because this is the place where the framework searches for view files. Next, I create a new Facelet view named `issues.xhtml` within the newly generated folder.

Facelets is not enabled by default, so to enable it, a web.xml deployment descriptor or a faces-config.xml file must be created for the application. Once that is generated, add the configuration shown in **Listing 7** to enable the Facelets view engine.

### ■ Listing 7.

```
<servlet>
    <servlet-name>Faces Servlet</servlet-name>
    <servlet-class>
        javax.faces.webapp.FacesServlet
    </servlet-class>
    <load-on-startup>1</load-on-startup>
</servlet>
<servlet-mapping>
    <servlet-name>Faces Servlet</servlet-name>
```





```
...
//Accessor methods
...
}
```

To implement the creation, a `DukeIssues` object should be instantiated within the `createIssue()` method of `IssuesController`, and the fields should be populated with those fields from the `Issue` class. Corresponding `DukeUser`, `DukePriority`, and `DukeStatus` objects should also be instantiated and populated accordingly. Finally, a `DukeIssuesFacade` EJB `createItem()` method is called upon to persist the new record. Listing 10 shows the source code for the method.

■ **Listing 10.**  
@POST  
@Path("/create")  
@Controller

```
public String createItem(@BeanParam Issue form) {
    DukeIssues entity = new DukeIssues();
    entity.setId(new Long(
        dukeIssueTrackerService.getIssueList().size() + 1));

    entity.setDescription(form.getDescription());
    entity.setSubject(form.getSubject());
    DukeUser user = new DukeUser();
    user.setId(Long.valueOf(
        dukeUserFacade.count()+1));
    user.setEmail(form.getRequestorEmail());
    user.setFirstName(form.getRequestorFirstName());
    user.setLastName(form.getRequestorLastName());
    entity.setRequestor(user);
    DukePriority priority = new DukePriority();
    priority.setId(Long.valueOf(
        dukePriorityFacade.count()+1));
    priority.setPriority(form.getPriority());
    entity.setPriority(priority);
    DukeStatus status = new DukeStatus();
    status.setId(Long.valueOf(
        dukeStatusFacade.count()+1));
    status.setStatus("OPEN");
    entity.setStatus(status);
    // Create record
    dukeIssuesFacade.create(entity);
    // Initialize issue list
    dukeIssueTrackerService.setIssueList(null);
    issuesBean.setIssueList(
        dukeIssueTrackerService.getIssueList());

    return "issues.xhtml";
}
```

### Issue Listing

Issue ID	Subject	Requestor	Priority	Status	Assignee
1	Test	duke@java.com	0	OPEN	
2	Test2	duke@test.com	0	OPEN	

### Create Issue

**Issue Information**

Subject:

Priority:

**Requestor**

First:

Last:

Email:

Figure 1. Duke Issue Tracker main view

The final product should look like Figure 1.







setting the property `CsrfOptions.IMPLICIT`, or manual validation can be configured by setting the property `CsrfOptions.EXPLICIT` and annotating resource methods with `@CsrfValid`.

## Additional Items

There are several ways to pass parameters to resource methods, and this article covered only `@RequestParam`. Other techniques are the use of query and path parameters. A query parameter is passed to the method at the end of the URL using the following notation:

```
/controller?param1=value&param2=value
```

Path parameters are passed to the resource method as part of the URL using the following notation:

```
/controller/param1/param2
```

Query parameters can be indicated using the `@QueryParam` annotation, and path parameters can be indicated using the `@PathParam` annotation within the resource method signature. Listing 12 demonstrates how to use a path parameter in a resource method.

### ■ Listing 12.

```
@POST
@Path("/date/{year}/{month}")
public String pathParamDate(
    @PathParam("year") int year,
    @PathParam("month") int month) {
    models.put("specifiedDate",
        month + "/" + year);
    return "showDate.xhtml";
}
```

Observers can be used for logging or other work that must be done when a particular event occurs. An observer can be configured by using the `@Observes` annotation on a resource method parameter, followed by the CDI event to observe.

```
public void onBeforeProcessView
    (@Observes BeforeProcessEvent e) {
    // do some work
}
```

Note that CDI observers are fired in a synchronous manner, so long-running tasks should not be handled within an observer.

## Conclusion

The MVC 1.0 framework provides a very flexible solution, worthy of serious consideration for enterprise apps. There are numerous excellent posts on MVC 1.0, and I recommend performing web searches to read the bevy of content and examples already available on this new framework. </article>

**Josh Juneau** (@javajuneau) is a Java Champion and an application developer, system analyst, and database administrator. He is a technical writer for Oracle Technology Network and *Java Magazine*, and he has written several books on Java and Java EE for Apress. Juneau is also a JCP expert group member for JSRs 372 and 378.

learn more

[GitHub project source code](#)

## Ozark project site

## MVC specification

## Adding MVC to GlassFish



## A look inside Oracle's open source JavaScript toolkit

Oracle JET is designed to be a comfortable toolkit for the traditional JavaScript developer. Installation is provided through the Node.js npm and bower modules, while starter templates are provided by using a Yeoman generator. The steps that follow assume that you are familiar with JavaScript and with Node.js, and that you already have Node.js and Git

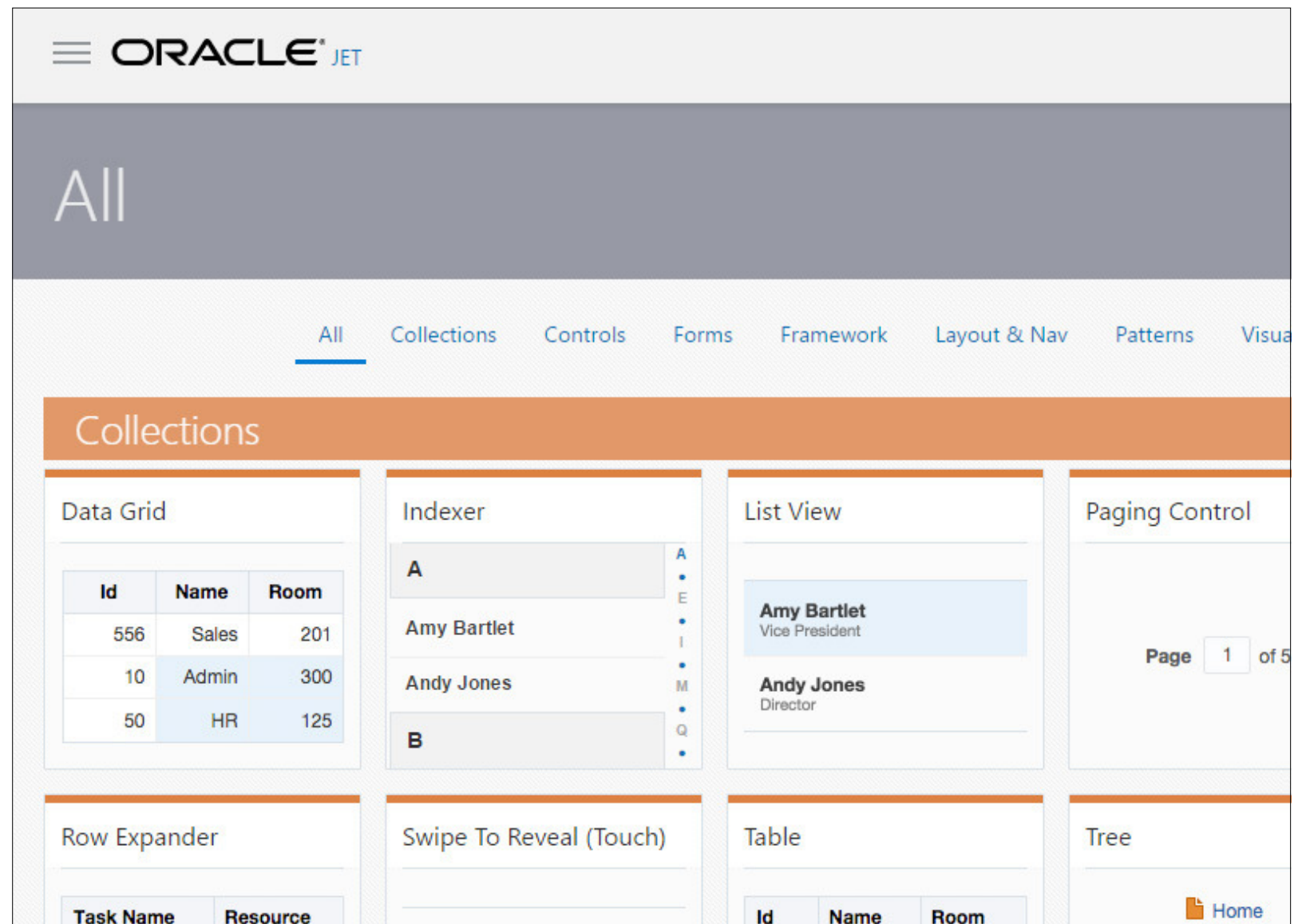
Oracle JET: Your app is ready! Change to your new app



The browser opens automatically (on localhost at port 8000). The application should show a mostly empty dashboard window when the browser opens.

If you resize the browser window or use the browser tools to render the content in mobile mode, you will see that the application shell is already configured as a responsive application and will change layout depending on the viewport size. **Figure 1** shows this same dashboard in a simulated device environment (here, the “device toolbar” in the Developer Tools menu of Google Chrome).

Tools menu of Google Chrome).



### Figure 2. Cookbook samples



Getting started is always the tricky part for anything new. With Oracle JET, you can use a helpful feature of the Oracle JET website called the Cookbook, which has recipes for all kinds of display components and widgets. **Figure 2** shows some samples. The code in the Cookbook is live, so you can make changes to HTML, JavaScript, or CSS and see the changes with a simple click of the Apply button. This is a great way to learn about the API and various options for different UI components.

Once you are ready to add to your own code, you can simply copy and paste the HTML and JavaScript to your own application and continue your development.

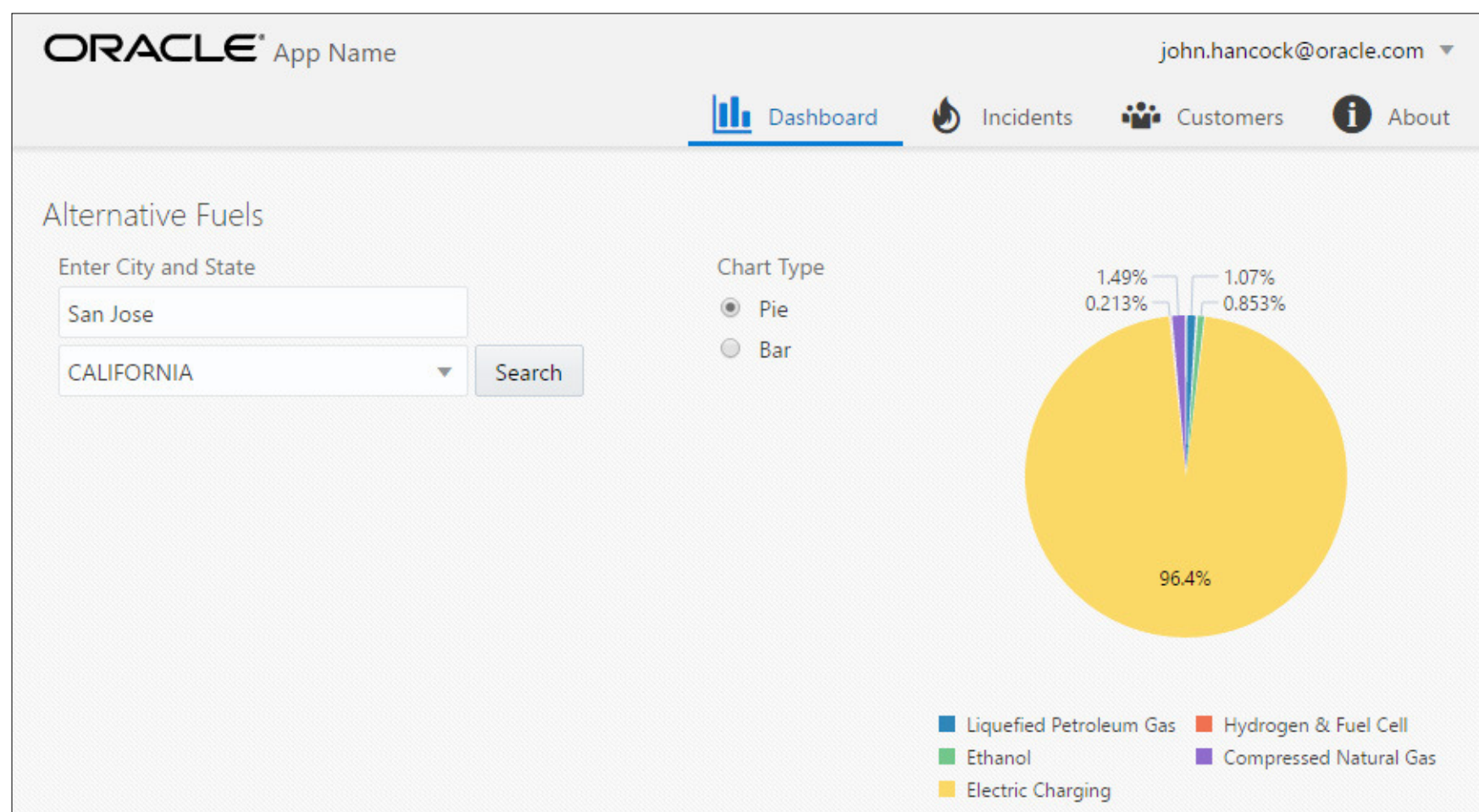
For this example, I am going to add one text input field, a selection menu, a button, a radio button set, and a chart.

The input field will take a city name; the selection menu will allow you to select a state; and the button will call a REST service to get all of the fueling stations offering alternative fuels in that city and state. The chart will display the returned data. Using radio buttons, you can select whether you want to see the results as a bar chart or a pie chart. You will also be able to click any of the items in the legend of the chart to show or hide that particular item. **Figure 3** is what the final application looks like.

Note that out of the box, all Oracle JET UI components meet the Web Content Accessibility Guidelines ([WCAG](#)) 2.0 at the AA level. While this might not mean much to some developers, it should be a goal of all developers to write software that can be used by persons with or without disabilities.

Beyond this just being the right thing to do, it's also a requirement for many government and public sector customers, as well as many enterprises around the world.

Let's take a look at the code behind the sample application. The architectural structure of an Oracle JET application follows the Model-View-View model (MVVM) pattern. MVVM is derived from the model-view-controller (MVC) pattern and is designed to separate development of the view layer (usually HTML) from the business logic and data layers (usually JavaScript and JSON, respectively). The toolkit is designed to be as



### Figure 3. What the sample application will look like











You will never connect one client to another client through a direct connection. **The dialogue is always between a client and the MQTT broker.**

broker. I provide many useful links to allow you to learn more about Mosquitto and MQTT at the end of this article.

## Using the Java Client for MQTT

```
<repositories>
  <repository>
    <id>Eclipse Paho Repo</id>
    <url>https://repo.eclipse.org/content/
      repositories/paho-releases/</url>
  </repository>
</repositories>
<dependencies>
  <dependency>
    <groupId>org.eclipse.paho</groupId>
    <artifactId>
      org.eclipse.paho.client.mqttv3
    </artifactId>
    <version>1.1.0</version>
  </dependency>
</dependencies>
```





I am going to create the following three classes. All these classes are included in the [code bundle](#) for this article.

- `MessageActionListener` implements the `org.eclipse.paho.client.mqttv3.IMqttActionListener` interface. I use instances of this class to specify the callback that will run code whenever a message has been successfully published to a topic.
- `Drone` represents a drone that has a name and can send and receive messages through the MQTT broker. This class not only encapsulates the data and logic related to drones as well as the messages and the commands included in messages, but it also implements the `MqttCallback` and `ImqttActionListener` interfaces defined in `org.eclipse.paho.client.mqttv3`. There is no type in these interface names—the naming convention for interfaces in the Java library is a bit confusing because some interfaces start with `I` while others don't. I use `Drone` instances as callbacks for specific events.
- `MqttSample01` creates three instances of the `Drone` class, makes them connect to the MQTT broker, and sends messages and commands. This class declares the `main` static method for the example application.

## Creating a Class That Is Notified When Asynchronous Actions Are Complete

The asynchronous API requires you to work with callbacks. In this example, I demonstrate many ways of working with the necessary callbacks. The following code shows the import statements and the code for the `Message ActionListener` class that implements the `IMqttActionListener` interface.

```
import
    org.eclipse.paho.client.mqttv3.IMqttActionListener;
import
    org.eclipse.paho.client.mqttv3.IMqttToken;
```

```
public class MessageActionListener
implements IMqttActionListener {
    protected final String messageText;
    protected final String topic;
    protected final String userContext;

    public MessageActionListener(
        String topic,
        String messageText,
        String userContext) {
        this.topic = topic;
        this.messageText = messageText;
        this.userContext = userContext;
    }

    @Override
    public void onSuccess(
        IMqttToken asyncActionToken) {
        if ((asyncActionToken != null) &&
            asyncActionToken.getUserContext()
                .equals(userContext))
        {
            System.out.println( String.format(
                "Message '%s' published to topic '%s'",
                messageText, topic));
        }
    }

    @Override
    public void onFailure(
        IMqttToken asyncActionToken,
        Throwable exception) {
        exception.printStackTrace();
    }
}
```

**What does it mean that a message was successfully delivered** by the MQTT broker? It depends on the quality of service (QoS) that you select.

that you select when you work with the MQTT protocol. The QoS level is the agreement between the publisher and the receiver of a message about the guarantees for delivering the message. Delivering a message involves publishing from the client to the broker and then from the broker to the subscribed client. MQTT supports three possible QoS values:

- In this example, I will work with QoS level 2, because I don't want the possibility of receiving a command twice. Messages are delivered even across network and client restarts. However, for that to occur, each message needs to be stored in a safe location until it has been successfully delivered. The Java client works with a pluggable persistence mechanism to store the messages. To keep things simple in this example, I will use memory-based persistence, which is not the best

What does it mean that a message was successfully delivered by the MQTT broker? It depends on the quality of service (QoS)





```
// options.setPassword(
//     "replace with your password"
//     .toCharArray());
// Replace with ssl:// and work with TLS/SSL
// best practices in a
// production environment
memoryPersistence =
    new MemoryPersistence();
String serverURI =
    "tcp://iot.eclipse.org:1883";
clientId = MqttAsyncClient.generateClientId();
client = new MqttAsyncClient(
    serverURI, clientId,
    memoryPersistence);
// I want to use this instance as the callback
client.setCallback(this);
connectToken = client.connect(
    options, null, this);
} catch (MqttException e) {
    e.printStackTrace();
}
}

public boolean isConnected() {
    return (client != null) &&
        (client.isConnected());
}

@Override
public void connectionLost(Throwable cause) {
    // The MQTT client lost the connection
    cause.printStackTrace();
}
```

```
        "%s subscribed to the %s topic",
        name, TOPIC));
    publishTextMessage( String.format(
        "%s is listening.", name));
    }
}

@Override
public void onFailure(IMqttToken asyncActionToken,
        Throwable exception)
{
    // The method will run if an operation failed
    exception.printStackTrace();
}
```

I implemented the same interface in the `MessageActionListener` class. However, in this case, I will implement the interface to run code when the success or failure is related to the connection, not with messages as happened in the `MessageActionListener` class.

The code saves the `IMqttToken` returned by the `client.connect` method to the `connectToken` protected field. This way, I am able to check whether the `onSuccess` method's execution is related to this token or not. The connection uses asynchronous execution, and the `onSuccess` method for the `Drone` class will be executed after the connection with the MQTT broker has been successfully established.

The `onSuccess` method displays a message indicating that the specific drone has been successfully connected. Then, the code calls the `client.subscribe` method with the topic to which I want to subscribe and the desired QoS level. The call to this method specifies `this` as the last argument because I will also use the actual instance as the callback that will execute specific methods when some asynchronous events related to the subscription occur. The fourth argument for the `subscribe` method requires an argument of the

`IMqttActionListener` type. So, after a successful subscription, the Java client will run the same `onSuccess` method, but the code will recognize that the event is related to the subscription because the received token won't match the connection token and instead will match the subscription token. It is not necessary to make it this way. It is possible to create an anonymous type to declare the methods that are necessary for the subscription callback for the asynchronous subscription. However, I wanted to demonstrate the usage of the tokens.

The `onSuccess` method displays a message indicating that the specific drone has been successfully subscribed to the specific topic. Then, the code calls the `publishTextMessage` method to publish a message to the topic indicating that the drone is listening.

```
public MessageActionListener publishTextMessage(
    String messageText)
{
    byte[] bytesMessage;
    try {
        bytesMessage =
            messageText.getBytes(ENCODING);
        MqttMessage message;
        message = new MqttMessage(bytesMessage);
        String userContext = "ListeningMessage";
        MessageActionListener actionListener =
            new MessageActionListener(
                TOPIC, messageText, userContext);
        client.publish(TOPIC, message,
            userContext, actionListener);
        return actionListener;
    } catch (UnsupportedEncodingException e) {
        e.printStackTrace();
        return null;
    } catch (MqttException e) {
        e.printStackTrace();
    }
}
```







```
*Master Drone* successfully connected
[Drone #2] successfully connected
[Drone #1] successfully connected
[Drone #2] subscribed to the java-magazine-mqtt/drones/altitude topic
[Drone #1] subscribed to the java-magazine-mqtt/drones/altitude topic
*Master Drone* subscribed to the java-magazine-mqtt/drones/altitude topic
Message '[Drone #1] is listening.' published to topic 'java-magazine-mqtt/drones/altitude'
Message '*Master Drone* is listening.' published to topic 'java-magazine-mqtt/drones/altitude'
Message '[Drone #2] is listening.' published to topic 'java-magazine-mqtt/drones/altitude'
[Drone #2] received java-magazine-mqtt/drones/altitude: [Drone #2] is listening.
[Drone #2] received java-magazine-mqtt/drones/altitude: *Master Drone* is listening.
[Drone #2] received java-magazine-mqtt/drones/altitude: [Drone #1] is listening.
*Master Drone* received java-magazine-mqtt/drones/altitude: [Drone #2] is listening.
[Drone #1] received java-magazine-mqtt/drones/altitude: [Drone #2] is listening.
*Master Drone* received java-magazine-mqtt/drones/altitude: *Master Drone* is listening.
[Drone #1] received java-magazine-mqtt/drones/altitude: *Master Drone* is listening.
*Master Drone* received java-magazine-mqtt/drones/altitude: [Drone #1] is listening.
[Drone #1] received java-magazine-mqtt/drones/altitude: [Drone #1] is listening.
[Drone #1] received java-magazine-mqtt/drones/altitude: COMMAND:GET_ALTITUDE:[Drone #1]
Message 'COMMAND:GET_ALTITUDE:[Drone #1]' published to topic 'java-magazine-mqtt/drones/altitude'
[Drone #2] received java-magazine-mqtt/drones/altitude: COMMAND:GET_ALTITUDE:[Drone #1]
[Drone #1] altitude: 3746 feet
*Master Drone* received java-magazine-mqtt/drones/altitude: COMMAND:GET_ALTITUDE:[Drone #1]
Message 'COMMAND:GET_ALTITUDE:[Drone #2]' published to topic 'java-magazine-mqtt/drones/altitude'
[Drone #2] received java-magazine-mqtt/drones/altitude: COMMAND:GET_ALTITUDE:[Drone #2]
[Drone #1] received java-magazine-mqtt/drones/altitude: COMMAND:GET_ALTITUDE:[Drone #2]
[Drone #2] altitude: 4224 feet
*Master Drone* received java-magazine-mqtt/drones/altitude: COMMAND:GET_ALTITUDE:[Drone #2]
Message 'COMMAND:GET_ALTITUDE:[Drone #1]' published to topic 'java-magazine-mqtt/drones/altitude'
[Drone #1] received java-magazine-mqtt/drones/altitude: COMMAND:GET_ALTITUDE:[Drone #1]
[Drone #2] received java-magazine-mqtt/drones/altitude: COMMAND:GET_ALTITUDE:[Drone #1]
[Drone #1] altitude: 433 feet
*Master Drone* received java-magazine-mqtt/drones/altitude: COMMAND:GET ALTITUDE:[Drone #1]
```

### Figure 1. Console messages

} }

Then, a forever-running loop generates a pseudorandom number every five seconds and, based on that number, it makes `masterDrone` publish a command to get the altitude for either `drone1` or `drone2`.

**Figure 1** shows an example of the output messages shown in the Console window of the IDE. Notice that both Drone #1 and Drone #2 receive the same messages with the `GET_ALTITUDE` command. However, only the drone that is the destination for the message processes the command and displays its pseudorandom altitude.

## Conclusion

This example demonstrates how you can use the Eclipse Paho Java Client and a Mosquitto MQTT broker to subscribe to a topic and publish messages to a topic. There is also a Java client library that can run on Android, in case you need to work with MQTT in Android. Whenever you need to exchange messages with an asynchronous, nonblocking API, you can consider using MQTT and the Java client. </article>

**Gastón Hillar** (@gastonhillar) has been working as a software architect with Java since its first release. He has 20 years of experience designing and developing software. He is the author of many books related to software development, hardware, electronics, and the Internet of Things, and he has been awarded the Intel Black Belt Software Developer Award eight times.

learn more

[The MQTT protocol](#)

[The `iot.eclipse.org` project](#)

# Exploring Compact Profiles

Java 8 introduced the concept of Compact Profiles, which are reduced versions of the Java runtime environment (JRE) that do not contain the usual full contents of `rt.jar`. In this article, I explore the advantages of using Compact Profiles and how they point the way toward a modular future for the JDK, which takes a big step forward in Java 9.

- Faster JVM startup times
- Reduced resource consumption
- Removal of packages that, in hindsight, shouldn't be in the core
- Improved security, because removing unused classes reduces the attack surface of the platform
- Convergence of the Java ME Connected Device Configuration (CDC) with Java SE

The initial approach to this effort was a full modularization of the platform, known as Project Jigsaw. This ambitious project also included other goals:

- Incorporate best practices for dependencies into the platform core, and apply lessons learned about dependency management from tools such as Maven, Apache Ivy, OSGi standard, and Linux distributions.
- Isolate dependencies—solve the library versioning problem.

The catch is that in order to achieve the full set of goals, major surgery on the Java platform core is required. In particular, a new approach to classloading—involving a “modular classloader”—is required. This has a lot of edge cases and is a complex undertaking, especially if we need to maintain backward compatibility.

- Is fully compliant with the JVM and Java language specifications
- Has a substantially reduced footprint
- Removes functionality that is not always needed (for example, CORBA)
- Works for many applications (especially server-side code)

Compact Profiles are based on packages; they contain a number of full packages, and no partial packages are currently allowed. They are also subject to two other restrictions:

- A profile must form a closed set; references to classes not contained in the profile are not allowed.
- If a profile contains some classes from another, smaller profile, it must contain all of them, so partially overlapping profiles are not allowed. Put another way, profiles are additive.





```
-> java.lang.Class
-> java.lang.InterruptedException
-> java.lang.Object
-> java.lang.String
-> java.lang.Thread
-> junit.extensions.ActiveTestSuite$1      junit.jar
-> junit.framework.Test                    junit.jar
-> junit.framework.TestCase                junit.jar
```

If we want a slightly more high-level view, we can use `-V package` to show dependencies between packages, as shown in **Listing 3** (some of the output was truncated because it was 1,297 lines long).

### ■ Listing 3.

```
jdeps -V package junit.jar
junit.jar -> /Library/Java/JavaVirtualMachines/openjdk8/
Contents/Home/jre/lib/rt.jar
  junit.extensions (junit.jar)
    -> java.lang
  junit.framework (junit.jar)
    -> java.io
    -> java.lang
    -> java.lang.annotation
    -> java.lang.reflect
    -> java.util
  junit.runner (junit.jar)
    -> java.io
    -> java.lang
    -> java.lang.reflect
    -> java.text
    -> java.util
```

`jdeps` is also very flexible about what it will accept as input: a JAR file, a directory, or a single `.class` file. It provides capabilities for recursive traversal and for specifying that only

packages with a name that matches a given regular expression should be considered. It can also warn that code uses an internal API and is not portable between Java environments (and might break if run against a future version of Java).

Finally, let's look at the NetBeans IDE. The current version already has support for a wide range of JDK 8 features, including Compact Profiles. When selecting which JDK or JRE to use in Project Properties, for JDK 8 and later, a developer can choose whether to compile against the full JRE or a profile. This makes it much easier to ensure that when you're targeting a particular profile, unwanted dependencies don't creep in. With luck, other IDEs will follow suit and also add support to allow developers to write code in the IDE that checks conformance with a specific profile at development time.

## A Word About Stripped Implementations

In addition to Compact Profiles, there was another technique that was proposed but ultimately not included in Java 8: Stripped Implementations. A Stripped Implementation was to be a reduced JRE, which was packaged with an application that had the exclusive use of it. Because the application was the only possible client for the Stripped Implementation, the runtime could be aggressively pruned, removing packages, classes, and even methods that were not used by the application.

This approach was advantageous in circumstances where resource limitations were severe. It relied on extensive testing to ensure that nothing that the application could rely on was removed by the stripping process. In general, it is extremely difficult to get a precise accounting of an application's dependencies. This is due in large part to the existence of techniques such as reflection and classloading, which can greatly complicate (or even render impossible) the task of ascertaining the true dependencies of a set of classes.

Compact Profiles are very different from Stripped Implementations—the former are Java runtimes designed

Stripped Implementations were targeted as a feature for Java SE 8, but due to some complications with licensing and the current testing kit, they had to be dropped very close to the release date.

[Java 9, which ships midyear, implements a whole new modularity system based on Project Jigsaw. In most cases, that modularity system does away with the need for Compact Profiles and Stripped Implementations. This article, which originally appeared in 2014 and has been lightly updated, shows how the drive to smaller executables has been a focus of the Java team for a long time. Going forward, Compact Profiles will remain a useful option only for projects that need the reduced executables but, for one reason or another, cannot migrate to Java 9. —Ed.] [</article>](#)

learn more

## [Hinkmond Wong's EclipseCon 2014 presentation on Compact Profiles](#)



world. From its humble beginnings with 35 members sitting on the floor of an abandoned office, the JUG has grown to more than a thousand members. JUG meetings typically see 70 to 100 developers attending on a regular basis. Over the years, the JUG has hosted many Java luminaries, such as Neal Ford, Brian Goetz, Cameron Purdy, Rod Johnson, Gavin King, Marc Fleury, Greg Luck, Yakov Fain, and Kito Mann. The group has a strong roster of regular local speakers. You can follow the JUG's activities via its [meetup group](#) and its [Twitter account](#).

The Philly JUG aims to further expand its reach and continue to serve its local community. Among these efforts, the JUG is an active adopter of critical Java SE and Java EE JSRs, including Java EE 8, Servlet 4, and Java SE 9. The JUG is an active partner member of the Java Community Process.





PETER LEDBROOK

# Gradle's Java Library Management

As builds become more complex, library dependencies present a special challenge that build tools are working to solve.

**B**efore the days of Apache Maven and Apache Ivy, making use of third-party libraries was a laborious affair. Every time you wanted to include a library, you would have to download manually not only its JAR but also any other JARs that it depended on. You would then go through a similar process whenever you wanted to upgrade a library. Tracking down the necessary transitive dependencies could be particularly painful, as could identifying any dependencies you no longer needed.

Both Maven and Ivy simplified the process by allowing you to declare which dependencies you needed via a set of standard coordinates: a group, a name, and a version. The tools took on the responsibility of locating and downloading the necessary JARs for you—a process known as *dependency resolution*. Make no mistake, automatic dependency management (in addition to a single public Java library repository—Maven Central) fundamentally changed the way Java developers worked.

This approach has been refined over the years, but what you are using now is still at heart the same. In fact, you might be forgiven for thinking that dependency management is a “solved” problem.

Automatic dependency management introduced its own set of issues that can bedevil developers. Build tools can and should do more to help, because they are best placed to improve the situation. If you're interested in learning what that help looks like, then read on as I investigate several common challenges of working with Java libraries—both third-party and your own.

## Consuming Third-Party Libraries

In an ideal world, you would declare your project's dependencies and everything would just work. But this rarely happens except in small projects. A more likely outcome is that you would encounter one or more of the following issues:

- An unexpected library version that breaks the build or your app/library
- Multiple versions of a library on a classpath
- Multiple JARs with the same class or classes
- Unwanted or unnecessary dependencies
- Dependency resolution that works for some developers but not others

The sources of these issues are varied, ranging from poorly defined metadata to changes in the names of dependencies. And don't underestimate the inherent challenge of keeping a system working that relies on perhaps hundreds of moving parts, with each library having its own release schedule, time constraints, priorities, and so on.

Given that you're likely to encounter problems with dependency resolution (unless you never change or upgrade any of your dependencies), it's crucial that you be able to identify the underlying issues quickly. That's why I think developers deserve better diagnostic tools.

## Diagnosing Dependency Issues

You might think that tracking down the source of a version conflict or some other dependency issue is simply a case of looking at the dependency graph. You can do this with most

build tools, but remember that dependency graphs can be large and complex. Trying to identify the source of a problem in such scenarios is often elusive and frustrating—even if you know what you’re looking for.

Imagine you have added a library to your Hibernate 4-based project that, without you realizing it, has pulled in Hibernate 5. If you're lucky, your project continues to work regardless of which version of Hibernate the build tool selects. Alternatively, the build or the running project might fail in a way that makes it obvious there's an issue with the version of the Hibernate library. In that case, you can run Gradle's `dependencyInsight` command (which is a built-in task) to find out which dependencies transitively depend on Hibernate.

What if your project falls into neither category? What if, instead, the build or running project fails in a distinctly unhelpful way? In that case, the obvious question to ask is: what changed? Unfortunately, build tools haven't made it easy to answer that question in the past.

One workaround is to disable automatic conflict resolution, resulting in the build tool reporting an error if there is more than one version of a library in the dependency graph. You can do this in Gradle by adding the following snippet to your build file:

```
configurations.all {
    resolutionStrategy {
        failOnVersionConflict()
    }
}
```

This approach unfortunately forces you to resolve even minor version conflicts yourself. It also fails to help if a library's name or group has changed.

Wouldn't it be great if you could just ask the build tool to show the differences between the dependency graphs of

the current build and the last working one? This feature is available in Gradle Enterprise, saving developers effort and time in diagnosing many dependency issues.

Identifying the source of a dependency issue is only part of the solution, because you still need to fix it somehow.

## Fixing Dependency Issues

Build masters and developers traditionally resolve dependency problems using exclusions, a solution that reminds me of the adage that when all you have is a hammer, everything looks like a nail. Exclusions may solve the problem in practice, but they're often fragile and time-consuming to maintain.

What you need is an expressive way to specify rules or constraints for the dependency resolution engine. Taking the earlier Hibernate example, what you probably want to say is “always use version 4.3.11.Final of Hibernate.” You can do this with Gradle using the following build script snippet:

```
configurations.all {
    resolutionStrategy {
        force
            'org.hibernate:hibernate-core:4.3.11.Final'
    }
}
```

This ensures that only the specified version of the hibernate-core dependency is included on your project's classpath.

You might also want to consider what happens if Hibernate 3 or 5 appears in the dependency graph. Do you simply want to force their versions as previously described? Or does it make more sense to add a warning or even fail the build

**Ideally, what you need** is some way to declare whether a dependency is exposed in your component's API or not.

# Managing library dependencies at the build level is deceptively hard.

in order to compile, which you would specify in Gradle as

This seems simple enough, so what's the problem? The flaw is that this simple dependency graph ignores the consumer's perspective. It doesn't answer the question: does `core` (the consumer of `date-utils`) need `joda-time` in order to compile, or is the `date-utils` module enough?

Ideally, what you need is some way to declare whether a dependency is exposed in your component's API or not. You will be able to do so from Gradle 3.4 onward. We are introducing two new configurations (or *scopes*, in Maven terminology) for Java projects: `api` and `implementation`. As part of the new Java library plugin, these configurations will allow you to express the effect of dependencies on consumers.

```
dependencies {
    api 'joda-time:joda-time:2.9.7'
}
```





By declaring `joda-time` as an API dependency, you ensure that it is included on the compilation classpath of `core`, because `core` requires `date-utils` to compile.

Now consider the relationship between `date-utils` and `core` from the perspective of `App`. Does `App` need to be aware of `date-utils` at compile time? In this case, let's assume that `date-utils` is used purely internally by `core`. You would express this in `core`'s build script this way:

```
dependencies {
    implementation project(':date-utils')
}
```

What these two dependency declarations mean for the overall project is

- core has both date-utils and joda-time on its compilation classpath
- App has date-utils and joda-time on its runtime classpath but not its compilation classpath
- App requires core at compile time

This approach has several significant advantages:

- JARs don't leak onto compilation classpaths when it's unnecessary for them to do so.
- Changing an implementation dependency doesn't force a recompilation of consumers.
- Published POMs match the requirements of consumers.
- The relationships between components are clearer.

With respect to the published POM, Gradle uses a simple mapping of `api` to `compile` and of `implementation` to `runtime`. This mapping makes sense because a transitive `api` dependency is required in order to compile the consumer, but that's not the case for a transitive `implementation` dependency.

I've discussed consuming libraries so far, but only in the context of dependency repositories and multimodule builds. There is one other scenario to consider: when you

want to work from the source code version of a library that is not part of your multimodule build.

## Improving Versioned-Library Workflows

There is an interesting debate going on at the moment regarding the relative merits of a *monolithic repository* (monorepo), which is a single source code repository for *all* your code, versus the more usual design, which relies on separate repositories for different projects. Of particular relevance to this discussion of libraries is one of the declared benefits of a monorepo: easier cross-project changes.

Consider this workflow:

- You discover that the project you're working on requires a fix to a library it uses.
- You ask that library's team to implement the fix.
- You incorporate the fixed library into your project.

In the case of a monorepo, you have access to the fixed version as soon as you update your local working copy or repository. And if you deploy off the trunk/master, you can deploy a new version of your project with the fix right away.

The process is less straightforward if the library is in a different source repository. The typical solution involves getting the library maintainer's team to publish a new version of the JAR, often as a snapshot. But snapshots are notoriously unreliable because the actual binary can change at any moment. So there is no guarantee that a given fix will appear in the next release version.

Another option is to check out or clone the source repository for the library and build it locally. Then you become less dependent on the library’s team to publish updated versions. On the other hand, you still end up relying on locally installed or published snapshots—unless you are able to make your project depend on the library’s source project, and then build them together. Adding a dependency on the source distribution (rather than just an evolving

binary) would give you the following advantages:

- You control exactly what code you're working against, via a commit ID, tag, or some other label.
- You can make fixes yourself and test them without any intermediate steps.
- You don't need to rely on snapshots that can easily break your project again.
- You can easily test in-progress work by updating your working copy/repository.

You can achieve this with Gradle's *composite builds* feature. It enables you to incorporate any other Gradle build into your own, dynamically replacing the corresponding declared dependencies.

To understand how it works, think back to **Figure 1** and imagine you need a fix for the Joda Time library. Also imagine that it has a Gradle build. The maintainer is too busy to implement the fix, so you get hold of the source code project and put it alongside or even inside your project's root directory. Then you can apply the necessary changes yourself.

Now, rather than building joda-time separately and installing or publishing its JAR, you can run

```
./gradlew --include-build ../joda-time-copy build
```

This will automatically build the project at `../joda-time-copy` and put its JAR on the compilation classpath of `date-utils` when it builds that, even though the latter declares a versioned dependency of `joda-time`. Even better, your tests will run against this development version of `joda-time` as well. This approach greatly improves the development cycle for cross-project changes and works particularly well with the IDE support in IntelliJ and Eclipse, which uses the [Buildship plugin](#).

One other consideration is how you share your work with other team members without having to publish intermediate versions of the library for them to use. It's an awkward

problem, but you can solve it by persisting the build inclusion definition in your project's `settings.gradle` file. This is explained on the [Gradle blog](#). Just be aware that you should use something like Git submodules or Subversion externals to ensure that your team members have a local copy of the library's source when they build your project. Alternatively, you can make the inclusion conditional on the library existing locally.

It's also worth thinking about how you manage the continuous integration of your own libraries and applications that reside in separate source repositories. With Gradle, you can create a composite build for your continuous integration servers that includes only the builds you need—it's otherwise empty—and runs all the integration tests. Both the application and the library teams then get immediate feedback on any breaking changes without having to publish intermediate binaries.

Composite builds smooth the development process and buy time for teams to coordinate releases where necessary. While this discussion involved separate repositories, you can also use this feature with a monorepo if you want to use that architecture while retaining independently versioned libraries.

## Conclusion

Managing library dependencies at the build level is deceptively hard, as anyone who has worked extensively with them will attest. In this article, I've presented several of the challenges that Java developers and build masters face. Many of you have probably encountered at least some of these in your own day-to-day work. The ultimate goal of the build tool developer is to make dependency management truly manageable. [.</article>](#)

**Peter Ledbrook** is an independent consultant who has maintained many builds over the years, including ones based on Make, Ant, and Maven. He now does training and technical writing for Gradle Inc.





## Intermediate and advanced test questions

**A**s promised in my last column, with this issue, I begin to include questions that simulate the level of difficulty of the Oracle Certified Associate exam, which contains questions for a more preliminary level of certification than the questions that regularly have appeared here.

Of course, I also include the more advanced questions that simulate those from the [1Z0-809 Programmer II exam](#), which is the certification test for developers who have been certified at a basic level of Java 8 programming knowledge and now are looking to demonstrate more-advanced expertise.

Because there is little value in asking easy questions, I provide only intermediate and advanced questions (and they are marked as such). These levels correlate with the two exams. However, don't let the "intermediate" tag fool you—the questions are never trivial.

**Question 1 (intermediate).** Given this code (with line numbers at left), which is a fragment of a larger method body:

```
14: StringBuilder[] sba = {
15:     new StringBuilder("Fred"),
16:     new StringBuilder("Jim"),
17:     new StringBuilder("Sheila")
18: };
19:
20: System.out.println("sba[2] is " + sba[2]);
21:
```

**Which two of the following are true?**

- a. The array referred to by `sba` might be eligible for garbage collection at line 19.
- b. The array referred to by `sba` might be eligible for garbage collection at line 21.
- c. Assigning `sba = null;` at line 21 would make the array referred to by `sba` and the three `StringBuilder` objects definitely eligible for garbage collection.
- d. The array referred to by `sba` and the three `StringBuilder` objects will definitely be eligible for garbage collection when the method returns to its caller.
- e. The array referred to by `sba` and the three `StringBuilder` objects might not be eligible for garbage collection even after the method returns to its caller.

**Question 2 (intermediate).** Given the following code:

```
// line n1
switch (x) {}
```

**Which two of the following lines of code can be added successfully at line n1? Assume that `x` has no declaration in scope at line n1 and assume that each line is added individually.**

- `boolean x = false;`
- `short x = 99;`
- `int x = 0;`
- `long x = 0;`
- `StringBuilder x = new StringBuilder("x");`

**Question 3 (advanced).** Given this:

```
public class Recorder {
    class Record {}
    class LongRecord extends Record {}

    public List<LongRecord> gatherRecords() {
        return Arrays.asList(
            new LongRecord(), new LongRecord()
        );
    }

    public void processRecords(
        Collection<Record> records) { // line n1
        records.forEach(System.out::println);
    }

    public void gatherAndProcess() {
        List<LongRecord> lr = gatherRecords(); // line n2
        processRecords(lr);
    }

    public static void main(String[] args) {
        new Recorder().gatherAndProcess();
    }
}
```

**Which is true?** Choose one.

- a.** The code produces output in the following form:

```
Recorder$LongRecord@1218025c
Recorder$LongRecord@816f27d
```

- b.** Changing the argument type declaration at line n1 to `List<Record>` produces output in the following form:

Recorder\$LongRecord@1218025c

Recorder\$LongRecord@816f27d

- c. Changing the argument type declaration at line n1 to `Collection<? super Record>` produces output in the following form:

```
Recorder$LongRecord@1218025c
Recorder$LongRecord@816f27d
```

- d. Changing the argument type declaration at line n1 to `Collection<? extends Record>` produces output in the following form:

Recorder\$LongRecord@1218025c  
Recorder\$LongRecord@816f27d

- e. Changing the variable type declaration at line n2 to `List<Record>` produces output in the following form:

Recorder\$LongRecord@1218025c  
Recorder\$LongRecord@816f27d

**Question 4 (advanced).** Given this code:

```
List<String> ls = Arrays.asList("Fred", "Jim", "Sheila",  
    "Fred");  
Set<String> s1 = new HashSet<>(ls); // line n1  
Set<String> s2 = new TreeSet<>(ls); // line n2  
System.out.println(s1.equals(s2));
```

**What is the result?** Choose one.

- a. Line n1 causes compilation to fail.
- b. Both line n1 and line n2 cause compilation to fail.
- c. An exception is thrown at line n1.
- d. Outputs true.
- e. Outputs false.

However, if line 21 executes `sba = null;`, the previous value of the reference is overwritten. Because the now-lost value in `sba` was the *only* reference to the array created in lines 14 through 18, and that value was never stored anywhere else (not even as an argument to a method call) between its creation and line 21, you know that there are now zero references to that object that are available to the program. As a result, the array is definitely unreachable and is eligible for garbage collection. Also, references to the three `StringBuilder` objects were written only into the array, and because the array is not reachable in the program, neither are the `StringBuilder` objects, and they, too, are eligible for garbage collection. Because of this, option C is correct.

In this case, because one or the other must be true, either you can definitely collect the objects at the return of the method or you cannot be sure. Which is it? Generally, local variables go out of scope and cease to exist when a method returns. This would suggest that the objects referred to by those variables become eligible for garbage collection. However, in this case, you don't see the whole method body. If a reference to the array "escapes" from the method, you can make no such assumption. At least three mechanisms exist by which this escape might occur, the most obvious of which is probably when the method itself returns the value of `sb` to its caller. A second possibility is that this method stores the value of `sb` in a variable that has greater visibility and longer life, such as a static variable or collection. In addition, passing `sb` as a method argument might allow the value to escape, because you cannot know if that method stores the value somewhere in a manner similar to the previous point. As a result, you cannot know if the array and, therefore, the three `StringBuilder` objects are eligible for garbage collection. Consequently, option E is true and option D is false.

**Question 2.** The correct answers are options B and C. The rules for a `switch` statement require that the argument be assignment-compatible with an `int` (allowing for auto-unboxing), an `enum`, or a `String`. *Java Language Specification* section 14.11 states that the type of the expression “must be



char, byte, short, int, Character, Byte, Short, Integer, String, or an enum type, or a compile-time error occurs.” String and enum types haven’t always been allowed; enum was new with Java 5. Therefore, it didn’t apply before that and, in fact, String was not legal until Java 7.

Although `String` is a legal type for a `switch` statement, `StringBuilder` is not; neither is it assignment-compatible with `String`.

Because of these rules, options A, D, and E are all incorrect, while both options B and C are correct.

**Question 3.** The correct answer is option D. This question delves into one of the most startling consequences of the type-erasure mechanism of Java’s generics system. When a variable of a collection type is declared with its generic specification—as in something like `List<Record>`—the “Record” part of the information exists only at compile time. The underlying object is still a `List` that actually accepts any object type. The power of generics is that they allow the compiler to do “consistency checking” that can ensure that type errors cannot happen in the code.

As part of the consistency checking, the compiler verifies that all assignments are safe from the perspective of the Liskov Substitution Principle. This is a principle in object orientation that requires that if you assign `b = a`, `a` must be capable of fully substituting for `b`. That is, all the behaviors expected of the type of `b` must be properly implemented, and work as expected, in the object referred to by `a`.

Now, the trick here is that, in fact, the compiler does not accept that assigning a `List<LongRecord>` to a `Collection<Record>` is a valid substitution. Because the assignment that occurs in the invocation of the method declared around line n1 is not valid, the code does not compile and option A is incorrect.

This question really hinges on why that assignment is not valid, and how you might correct that problem.

Option B suggests that by changing the argument type from `Collection` to `List`, you might fix it. However, a `List` is fully capable of substituting for a `Collection` (the `List` interface extends `Collection`, and all the methods are implemented). This part of the assignment is not the problem, and the change wouldn't alter the situation. Therefore, option B is incorrect.

Consider the following, because it's a little less confusing. The compiler thinks that a `List<LongRecord>` is not a valid substitute for a `List<Record>`. There's something that you can do with a `List<Record>` that isn't properly supported by a `List<LongRecord>`. That illegal operation is the addition of a `Record`. Think about that for a moment. A `List<LongRecord>` can properly contain anything that's assignment-compatible with a `LongRecord`: that would be `LongRecord` objects and any subclasses thereof. However, it should *not* contain any instances of the parent class `Record`. And it should not contain any sibling classes of `LongRecord`, if they existed. However, a `List<Record>` can legitimately contain such objects, and the compiler doesn't know that you don't add any in the method declared around line n1. Imagine the consequences if the method, prior to looping over the contents of `Collection`, invoked `records.add(new Record())`. That would be bad, right?

The code does not actually do any such terrible operations, but the compiler doesn't analyze that; it just knows that the code *could*. Fortunately, there's a syntax that lets you do what you want to do—that is, iterate over the contents of the collection, safely extracting things from it while knowing they're assignment-compatible with `Record`. In effect, you define that `Collection` can be a collection of “anything that's assignment-compatible with `Record`.” Therefore, if `X` is a type that's assignment-compatible with `Y`, `X` must be “further down” the inheritance hierarchy (regardless of whether it is an object or an interface) than `Y`. In a sense, `X` extends `Y`. And that's how the syntax comes about; the language lets you say,

in effect, “collection of something,” provided that something extends `Record`. (Note that in this sense, `Record` “extends” `Record` because it’s assignment-compatible.) That syntax, of course, is `Collection<? extends Record>`. On that basis, option D is correct.

Option C is also interesting. In this situation, the proposed change would say that whatever `Collection` is intended to contain, `Record` must be assignment-compatible with that. If you presented a collection of `Object`, then `Object` is a superclass of `Record`, and you'd be able to *insert* the `Record` safely into `Collection`. However, you have no idea what types you might pull out of `Collection`, which might be a problem. But regardless of this, you still cannot pass `List<LongRecord>` into the method, because `List<LongRecord>` could not properly accept assignment of a `Record` into `List` even if you wanted to do that. Therefore, this syntax doesn't solve the problem at hand, and option C is incorrect. However, this is an important syntax when you want to assign things to the generic type of whatever is passed in.

Option E fails for the same reasons the code fails in its unchanged form. The return type of the `gatherRecords` method is `List<LongRecord>`, and that's not assignment-compatible with `List<Record>`. Therefore, changing the variable type on line n2 would fail to compile, even though it would allow the invocation of the method around line n1 to compile. Substituting one problem for another doesn't result in any output, though, so option E is incorrect.

**Question 4.** The correct answer is option D. Options A and B suggest compilation failures while attempting to construct and initialize the sets. This might happen if there is no constructor available that accepts a `List` as an argument. However, both `Set` types provide such a constructor. The documentation for collections calls for collection implementations to provide constructors that accept an argument of type `Collection`, and as a result, a `List` is valid. Because of

this, both options A and B are false.

Option C suggests that an exception is thrown during construction of the `Set`. This might be plausible, given that the `List` has a duplicate entry, which is not permitted in a `Set`. However, the documentation notes that “all constructors must create a set that contains no duplicate elements.” The effect is that both `Set` objects contain only the three distinct elements, “Fred”, “Jim”, and “Sheila”. It’s also fair to observe that, in general, a `Set` recognizes and ignores attempts to add duplicates, so an exception in this initialization situation would be unhelpful and counterintuitive. Because no exception is thrown, option C is false.

As a side note, it's possible for a `TreeSet` to throw an exception during the addition of items, if the items being added to it are not `Comparable` and no suitable `Comparator` has been provided. In this case, however, the items being added are `String` objects, and `String` implements `Comparable<String>` as needed, so no such problem arises.

Having eliminated all the other options, options D and E are essentially alternatives. The interesting thing here is that the documentation for the `equals` method of `Set` requires that all implementations return a value that indicates only whether the contents of the set are the same, without regard to the implementing class. As the documentation notes further, “This definition ensures that the `equals` method works properly across different implementations of the set interface.” As a result, the output is the value `true`, and option D is correct, while option E is incorrect. `</article>`

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



## 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](mailto: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](mailto: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 our editor, **Andrew Binstock**, at [javamag\\_us@oracle.com](mailto:javamag_us@oracle.com).

While it will have no influence on our decision whether to publish your article or letter, cookies and edible treats will be gratefully accepted by our staff at *Java Magazine*, Oracle Corporation, 500 Oracle Parkway, MS OPL 3A-3133, Redwood Shores, CA 94065, USA.

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