Java™ magazine

By and for the Java community

INTERFACES IN DEPTH 90 | BOOKS ON JAVA 9 07

JANUARY/FEBRUARY 2018

REACTIVE PROGRAMMING

Handling large data streams efficiently

ORACLE®

IntelliJ IDEA

Level up your code
with a Pro Java IDE

jetbrains.com/idea

//table of contents /

REACTIVE PROGRAMMING WITH JAX-RS

By Mert Çalışkan

Using an asynchronous approach and staging to develop responsive reactive apps

COVER FEATURES

OTHER FEATURES

DEPARTMENTS

COVER ART BY PEDRO MURTEIRA

EDITORIAL

Editor in Chief
Andrew Binstock

Managing Editor
Claire Breen

Copy Editors
Karen Perkins, Leslie Steere

Technical Reviewer
Stephen Chin

DESIGN

Senior Creative Director
Francisco G Delgadillo

Design Director
Richard Merchán

Senior Designer
Arianna Pucherelli

Designer
Jaime Ferrand

Senior Publication Designer
Sheila Brennan

Production Designer
Kathy Cygnarowicz

PUBLISHING

Publisher and Audience Development Director
Karin Kinnear

Audience Development Manager
Jennifer Kurtz

ADVERTISING SALES

Sales Director
Tom Cometa

Account Manager
Mark Makinney

Mailing-List Rentals
Contact your sales representative.

RESOURCES

Oracle Products
+1.800.367.8674 (US/Canada)

Oracle Services
+1.888.283.0591 (US)

ARTICLE SUBMISSION

If you are interested in submitting an article, please email the editors.

SUBSCRIPTION INFORMATION

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

MAGAZINE CUSTOMER SERVICE

java@omeda.com

PRIVACY

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

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.

ORACLE CODE

Register Now

Oracle Code is BACK! | 1-Day, Free Event

Explore the Latest Developer Trends:

- DevOps, Containers, Microservices, and APIs
- MySQL, NoSQL, Oracle, and Open Source Databases
- Development Tools and Low Code Platforms
- Open Source Technologies
- Machine Learning, AI, and Chatbots

Live for the Code

Coming to a city near you:

developer.oracle.com/code

ORACLE®

The Decline of Dynamic Typing

A feature once viewed as a convenience has become more troublesome than it's worth.

If you follow the rise and fall of programming languages—either from the comfort of an armchair, ensconced with your preferred tools but interested in other people's choices, or from a keyboard, happy to hyperkinetically try out all kinds of new idioms—you will have noticed an unmistakable trend in modern language design: a preference for static typing.

Look at the major languages that have emerged in the past decade—Go, Swift, Kotlin, and Rust—they're all statically typed. Moreover, languages that were once dynamic have added static typing. The most conspicuous example is the recent set of updates to JavaScript (or more accurately, ECMAScript). Apple's choice to replace dynamically typed Objective-C with Swift also follows this trend.

As a quick refresher, static typing refers to a type system that makes it possible to know the type of every data item and expression at compile time. Specifically, this means that the language does not allow the use of types that are resolved at runtime. For example, in JavaScript (a dynamically typed language) a variable is declared by using `var`, rather than a specific type. A variable can hold a string, a number, or a boolean at various times in the same program. In contrast, static types, such as those found in Java, force you to declare the type when you define the variable.

Static typing provides several important advantages. The first advantage is that the compiler can perform significant program verification. Because the compiler knows that `i`, for example, has been declared an integer, it can

PHOTOGRAPH BY BOB ADLER/THE VERBATIM AGENCY

check that all places where `i` is used do expect or at least can support an integer. Likewise, static source code checkers are much more capable when they have complete type information.

A second key benefit is performance. A runtime that must determine the type of every variable and the methods that are available to it while the program is running has significant overhead that statically typed languages don't require. In part, this is why many of the traditional dynamic languages—Python, Perl, and Ruby—run much more slowly than statically typed options. An exception to this might be JavaScript, which runs much faster than many dynamic languages. But this speed is a comparatively recent advance driven by massive investments by Google and Microsoft into their respective JavaScript engines.

A final benefit, which in my view is the one that has turned the tide against dynamic languages, is maintainability. First, for readability, it is much easier to understand code if types are declared statically because it is then possible to tell exactly what you're looking at. For debugging, this aspect is invaluable. Stepping through code in which the contents of a variable can change type is not anybody's idea of fun. The dynamic aspect also introduces a kind of uncertainty when a bug is discovered. Say the variable `i`, which previously held an integer, now holds a string; was that a spelling error by the developer, who meant to store it in `u`, or was it an intentional reuse of a variable? And if the latter, how should you understand other instances of `i` in the codebase? These problems are bearable in the small but extremely troublesome in the large. This problem—precisely as it appears in large projects—was the primary motivation for Microsoft to create TypeScript, its superset of JavaScript that added one principal feature: types.

Dynamic typing flourished in popular languages in the mid-1990s (Python, Ruby, JavaScript, and PHP all appeared within a four-year window), when PC hardware had become powerful enough to run languages that needed runtime support. At the time, tools were primitive and compile times were long, so dynamic typing, which facilitated quick and easy development, was a welcome step forward.

But while dynamic languages have retained considerable popularity, some 15 years later the cost of dynamic typing is more apparent as codebases grow larger, performance becomes more important, and the cost of maintenance rises steadily. While those dynamically typed languages will surely be with us for a long time, it is unlikely that many new languages will embrace the model.

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

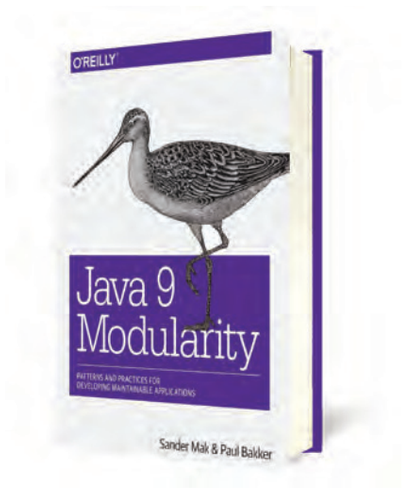
Java 9 Books

The wave of books for the new release is now arriving.

As with all previous major releases, the arrival of Java 9 has unleashed a wave of books examining and explaining its new features. The next few book columns will review important titles that you'll want to be aware of. In this installment, I look at two books, one specifically on Java 9 modules and one on the larger language.

JAVA 9 MODULARITY

By Sander Mak and Paul Bakker



The introduction of modules in Java 9 significantly changed how Java applications are built and delivered. These changes are particularly important for developers of Java libraries, who need to work out their strategy for delivering modular JAR files while continuing to provide the traditional bits that run on JVMs prior to this new release. Although you certainly can run apps on the Java 9 runtime without using modules, it is expected that most sites will switch over to module-based binaries during the next few years. Some sites, especially those wrestling with so-called "classpath hell," will likely find incentive to move to modules as quickly as possible. Those sites will discover a trove of useful information in Mak and Bakker's new work.

The book opens with a detailed explanation of what modules are and how they work. The first four chapters cover the anatomy and use of modules in detail, with plenty of examples. It's a very readable guide. The remaining 180 pages are where the value is really apparent. These pages start out covering modularity patterns, which are ways of architecting modules so that they work together ideally. The goal is to find the balance between devising modules that naturally (that is, conceptually) fit together while creating the minimum number of dependencies on external modules. This tension is familiar to Java architects designing JARs. However, the impetus to get the design right in traditional JARs historically has been more of a desirable goal than an imperative. With modules, it becomes a much more serious proposition. The whole point of

modularity is to manage dependencies intelligently to get rid of classpath conflicts and enable delivery of modestly sized binaries.

The authors tackle many aspects of the problem: splitting modules, aggregating modules, and creating new modules as facades. They then move into technical problems such as encapsulating resources, after which they explore building modules specifically for use in containers.

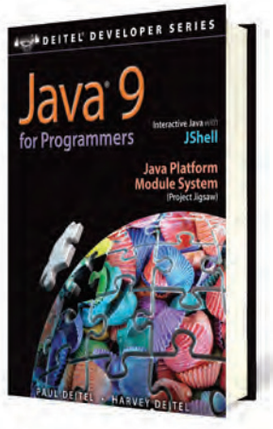
The final sections have pointers for library designers and a handy section on configuring tools, such as Maven and Gradle, for modules.

Wedged into all this goodness is a lengthy discussion of how to run Java 9 without migrating to modules. This will be particularly useful to sites that are planning to adopt modules at some future point and want to understand the full scenario, starting with limited migration to the new runtime and then slowly implementing the considerations presented in the rest of the book.

Taken together, all these topics represent a comprehensive overview of Java 9 modularity. The writing is clear and easy to understand, and the authors do not expect the reader to know much more than how to program in Java; low-level details (such as how classloaders work) are explained on the fly. This book was explicitly recommended at Devoxx in November by members of the core Java team, and I fully agree with their assessment. —*Andrew Binstock*

JAVA 9 FOR PROGRAMMERS (4TH EDITION)

By Paul Deitel and Harvey Deitel



This book is the first of the comprehensive language tutorials to come to market that includes extensive coverage of Java 9. In this context, it competes with other 1,000-page volumes that present the entire language and its principal APIs. For example, it competes with Cay Horstmann's excellent *Core Java*, which I've reviewed previously in this column. Both entrants are fine works, and choosing one or the other depends in large part on your personal preferences. (Note: *Core Java* has not been released for Java 9, although an abridged version is available.)

The Deitels' book is notable for its hands-on orientation: it is code-intensive with *numerous* examples. It even includes a full project (comprising 77 pages) that goes from initial design of an ATM machine all the way through to completion. The design portion includes introduction to the basic Unified Modeling Language (UML) diagrams, putting together the object-oriented design, and incrementally developing the code. Working through this project is an excellent education quite apart from the use of Java.

Java 9's most important features receive rich coverage. For example, the section on modules is a full 52 pages that explore the need for modules, how modules work, and how to use them in your own code. To get a sense of the hands-on nature of the explanations, see the lightly edited excerpt from this section that ran in this magazine. It was one of our most popular articles in 2017.

This is the first book I've seen on Java 9 that has in-depth coverage of JShell, the new REPL introduced in Java 9. Its pedagogical benefits are not lost on the authors, who drill into how to make best use of it both as a programming aid and as a teaching tool.

In addition to the language proper, the book covers JavaFX, JDBC, and JPA. Each chapter includes self-review exercises, with accompanying solutions; the explanations are sprinkled with caveats for dangers, reminders about good programming practices, and tips on writing idiomatic Java. In other words, this is a complete presentation.

I have only one gripe with this volume, and that is the excessive use of color highlighting in the code. Even if you're a fan of brightly colored code, your eyes will quickly tire of reading pale blue text or squinting at bright green comments on a canary-yellow background. But if you can handle that, you'll have a very fine book that does an excellent job of presenting Java 9. —*AB*

//events /

DevNexus

FEBRUARY 21–23
ATLANTA, GEORGIA

DevNexus is an international open source developer conference. Scheduled sessions this year include "Java Microservices Patterns & Practices with Kubernetes/OpenShift and Istio," "Pragmatic Microservices with Java EE and WildFly Swarm," and "Practical JVM Performance Tuning with jPDM."

PHOTOGRAPH BY NDINHTRAN, ESQ./FLICKR

SnowCamp

JANUARY 24, WORKSHOPS
JANUARY 25–26, CONFERENCE
JANUARY 27, SOCIAL EVENT
GRENOBLE, FRANCE

SnowCamp is a developer conference held in the French Alps that focuses on Java, web, cloud, DevOps, and software architecture, with a mix of sessions in French (the majority) and English. The last day, dubbed "unconference," offers a unique opportunity to socialize with peers and speakers on the ski slopes.

AgentConf

JANUARY 25–26, SPEAKER SESSIONS
JANUARY 27–28, SKIING/NETWORKING
DORNBIRN AND LECH, AUSTRIA

AgentConf is two days of talks and two days of skiing. It is a conference dedicated to software engineering, focusing on JavaScript, ReactJS, ReactNative, Node, and similar technologies. The event connects industry experts from around the world who use these technologies, and whose teams build projects with them. Speaker sessions are hosted at Spielboden in Dornbirn, while skiing and networking take place in Lech.

DevConf.cz

JANUARY 26–28
BRNO, THE CZECH REPUBLIC

DevConf.cz is a free, three-day, open source developer and DevOps conference. All talks, presentations, and workshops will be conducted in English. Several tracks are usually devoted specifically to Java EE, and the conference can be attended online.

DeveloperWeek

FEBRUARY 3–4, HACKATHON
FEBRUARY 5, WORKSHOPS
FEBRUARY 5–7, CONFERENCE
FEBRUARY 6–7, EXPO
OAKLAND, CALIFORNIA

DeveloperWeek promises the world's largest developer expo and conference series, gathering 8,000 participants for a week-long, technology-neutral programming conference and associated events. The theme for 2018 is "Industrial Revolution of Code," and tracks include artificial intelligence, serverless development, blockchain, APIs and microservices, and JavaScript.

PHOTOGRAPH BY BRIYYZ/FLICKR

Domain-Driven Design Europe

JANUARY 30–31, WORKSHOPS
FEBRUARY 1–2, CONFERENCE
AMSTERDAM, THE NETHERLANDS

This software development and engineering event spans analysis, modeling and design, systems thinking and complexity theory, architecture, testing and refactoring, visualization, and collaboration. Scheduled workshops include "Event-Driven Microservices with Axon Framework" (Java experience required) and "Techniques for Complex Domains."

Jfokus

FEBRUARY 5–7
STOCKHOLM, SWEDEN

The annual Scandinavian Java developer conference encompasses Java SE and Java EE, front-end and web development, mobile, cloud, IoT, and JVM languages such as Scala and Clojure.

O'Reilly Software Architecture Conference

FEBRUARY 25–26, TRAINING
FEBRUARY 26–28, 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 technologies, frameworks, and techniques. Scheduled sessions include "High-performance JavaScript Web App Architecture," "Pragmatic Event-driven Microservices," and "Evolving Database Architecture."

Embedded World

FEBRUARY 27–MARCH 1
NUREMBERG, GERMANY

The theme for the 16th annual gathering of embedded system developers is "Embedded Goes Autonomous." Topics include IoT, autonomous systems, software engineering, and safety and security.

JSConf Iceland

MARCH 1–2
REYKJAVIK, ICELAND

JSConf will take place at Harpa, one of Reykjavik's most distinguished landmarks, and feature two tracks of educational JavaScript talks by more than 30 speakers from around the world, followed by evening parties and socializing.

QCon London

MARCH 5–7, CONFERENCE
MARCH 8–9, WORKSHOPS
LONDON, ENGLAND

QCon conferences feature tracks related to web development, DevOps, cloud computing, and more. Confirmed speakers this year include Java Champion Trisha Gee, Docker engineer Anil Madhavapeddy, and Netflix cloud platform engineer Allen Wang.

Voxxed Days Zürich

MARCH 8
ZÜRICH, SWITZERLAND

Voxxed Days Zürich shares the Devoxx philosophy that content comes first, and draws internationally renowned and local speakers. Sessions include "The Power and Practicality of Immutability" and "A Hitchhiker's Guide to the Functional Exception Handling in Java."

JavaLand

MARCH 13–15
BRÜHL, GERMANY

This conference features lectures on subjects such as core Java and JVM languages, microservices architecture, front-end development, and much more. Scheduled presentations include

"The Java 9 Module System Beyond the Basics," "Securing JAX-RS," and "Next-Generation Web Components with Java Vaadin Flow."

JAX DevOps

APRIL 9 AND 12, WORKSHOPS
APRIL 10–11, CONFERENCE
LONDON, ENGLAND

This event for software experts highlights the latest technologies and methodologies for accelerated delivery cycles, faster changes in functionality, and increased quality in delivery. More than 60 workshops, sessions, and keynotes will be led by international speakers and industry experts. There's also a two-in-one conference package that provides free access to a parallel conference, JAX Finance.

Voxxed Days Melbourne

MAY 2–3
MELBOURNE, AUSTRALIA

Voxxed Days is heading down under to Melbourne, Australia. The event will feature insights into cloud, containers and infrastructure, real-world architectures, data and machine learning, the modern web, and programming languages.

Java Day Istanbul

MAY 5
ISTANBUL, TURKEY

Java Day Istanbul is one of the most effective international community-driven software conferences in Turkey, organized by the Istanbul Java User Group. The conference helps developers network and learn the newest technologies, including Java, web, mobile, big data, cloud, DevOps, and agile.

WeAreDevelopers World Congress

MAY 16–18
VIENNA, AUSTRIA

Billed as the largest developer congress in Europe, WeAreDevelopers expects more than 8,000 participants and more than 150 speakers for keynotes, panel discussions, workshops, hackathons, contests, and exhibitions. The program includes talks and sessions on front-end and back-end development, artificial intelligence, robotics, blockchain, security, and more.

JEEConf

MAY 18–19
KIEV, UKRAINE

JEEConf, the largest Java conference in Eastern Europe, focuses on practical experience and development. Topics include modern approaches in development of distributed, highly loaded, scalable, enterprise systems with Java and innovations and new directions in application development using Java.

Oracle Code Events

Oracle Code is a free event for developers to learn about the latest programming technologies, practices, and trends. Learn from technical experts, industry leaders, and other developers in keynotes, sessions, and hands-on labs. Experience cloud development technology in the Code Lounge with workshops and other live, interactive experiences and demos.

FEBRUARY 27, Los Angeles, California
MARCH 8, New York, New York
APRIL 4, Hyderabad, India
APRIL 10, Bangalore, India
APRIL 17, Boston, Massachusetts
MAY 17, Singapore

J On The Beach

MAY 23–25
MALAGA, SPAIN

J On The Beach (JOTB) is an international workshop and conference event for developers interested in big data, JVM and .NET technologies, embedded and IoT development, functional programming, and data visualization.

jPrime

MAY 29–30
SOFIA, BULGARIA

jPrime will feature two days of talks on Java, JVM languages, mobile and web programming, and best practices. The event is run by the Bulgarian Java User Group and provides opportunities

for hacking and networking.

Riga Dev Days
MAY 29–31
RIGA, LATVIA
The biggest tech conference in the Baltic States covers Java, .NET, DevOps, cloud, software architecture, and emerging technologies. This year, Java Champion Simon Ritter is scheduled to speak.

O'Reilly Fluent
JUNE 11–12, TRAINING
JUNE 12–14, TUTORIALS AND CONFERENCE
SAN JOSE, CALIFORNIA
The O'Reilly Fluent conference is devoted to practical training for building sites and apps for the modern web. This event is designed to appeal to application, web, mobile, and interactive developers, as well as engineers, architects, and UI/UX designers. The conference will be collocated with O'Reilly's Velocity conference for system engineers, application developers, and DevOps professionals.

EclipseCon France
JUNE 13–14
TOULOUSE, FRANCE
EclipseCon France is the Eclipse Foundation's event for the entire European Eclipse community. The conference program includes technical sessions on current topics pertinent to developer communities, such as modeling, embedded systems, data analytics and data science, IoT, DevOps, and more. Attendance at EclipseCon France qualifies for French training credits.

JavaOne
OCTOBER 28–NOVEMBER 1
SAN FRANCISCO, CALIFORNIA
Whether you are a seasoned coder or a new Java programmer, JavaOne is the ultimate source of technical information and learning about Java. For five days, the world's largest collection of Java developers gather to talk about all aspects of Java and JVM languages, development tools, and trends in programming. Tutorials on numerous related Java and JVM topics are offered.

Are you hosting an upcoming Java conference that you would like to see included in this calendar? Please send us a link and a description of your event at least 90 days in advance at javamag_us@oracle.com. Other ways to reach us appear on the last page of this issue.

THE DENVER JUG

The first Denver Java User Group (DJUG) meeting was held in November 1995 as an opportunity for technical discussion of the Java language, APIs, applets, and applications. Since then, the DJUG has grown to more than 2,500 members.

Its goal is to promote the use of Java, educate users of Java technology, provide a venue for the exchange of ideas, and create a community for Java developers in the Denver, Colorado, area.

Membership in the DJUG is free, and all Denver Java enthusiasts are encouraged to join. DJUG members have access to conference discounts for events such as the No Fluff Just Stuff Software Symposium, UberConf, and Devoxx. Meeting attendees also have the opportunity to win discounts on software-related products.

DJUG meetings are held on the second Wednesday of every month, and the typical meeting has between 70 and 120 attendees. Presentation topics from the past year include machine learning, microservices, Project Jigsaw, hack-proof security, and lightning talks.

Organized and run by volunteers, the meetings follow a typical format: networking time, speaker presentation, door prizes, and then more networking at a local restaurant. Door prizes and food and beverages for the networking sessions are provided with the generous help of sponsors.

Follow the DJUG's activities by joining its meetup group or visiting its website. Contact the DJUG on Twitter with proposals for talks.

DEVELOPER COMMUNITY EVENTS IN EARLY 2018
FROM THE DEVOXX FAMILY

DEVOXX™

DEVOXX.COM

FRANCE 18-20 APRIL

UNITED KINGDOM 9-11 MAY

POLAND 20-22 JUNE

ZURICH 8 MARCH

VIENNA 12 & 13 MARCH

BRISTOL 15 MARCH

BUCHAREST 22–23 MARCH

MELBOURNE 2–3 MAY

MINSK 26 MAY

SINGAPORE 1 JUNE

LUXEMBOURG 22 JUNE

VOXXEDDAYS

VOXXEDDAYS.COM

What Is Reactive Programming?

Reactive programming is a term that means slightly different things to different people. Central to the concept, though, is a model of computing that is alerted to certain kinds of events, can process or ignore those events, and works with the event source to manage the number of events to be processed.

In practice, this model rests on several technologies: a message-passing framework, a subscription-based notification system, and an asynchronous execution of the event-driven tasks. The benefit is a loosely coupled implementation that is scalable and tends to isolate failures. The scalability here refers to the ability to scale horizontally quickly, and it anticipates handling the number of events associated with big data—millions to billions of incoming events. This aspect in particular is what makes the reactive model different from its familiar forebear, the event loop in GUI development.

ART BY PEDRO MURTEIRA

In this issue, we provide an overview of reactive development (page 16) and then do a deep dive into RxJava (page 32), one of the leading libraries for developing reactive applications on the JVM. We follow that up by looking at the reactive capabilities built into the most recent release of Spring 5.0 (page 61). Finally, we examine a slightly different model for developing CRUD applications, called Command Query Responsibility Segregation, or CQRS (page 69), which while not reactive per se implements an approach that overlaps with reactive programming.

It might seem that reactive programming is a design that would lead naturally to microservice implementation. And indeed it is.

MERT ÇALIŞKAN

Reactive Programming with JAX-RS

Using an async approach and staging to develop responsive reactive apps

Reactive programming sounds like the name of an emerging programming paradigm at first, but it refers to a programming technique that offers an event-driven approach for handling asynchronous streams of data. Based on data that flows continuously, reactive systems react to the data by executing a series of events.

Reactive programming follows the Observer design pattern, which can be defined as follows: when there is a change of state in one object, the other objects are notified and updated accordingly. Therefore, instead of polling events for the changes, events are pushed asynchronously so the observers can process them. In this example, observers are functions that are executed when an event is emitted. And the data stream that I mentioned is the actual observable that will be observed.

Nearly all languages and frameworks have adopted this programming approach in their ecosystems, and Java has kept the pace up in its latest releases. In this article, I explain how reactive programming can be applied by using the latest version of JAX-RS from Java EE 8 and by using Java 8 features under the hood.

The Reactive Manifesto

The Reactive Manifesto lists four fundamental aspects an application must have in order to be more flexible, loosely coupled, and easily scalable—and, therefore, capable of being reactive. It says an application should be responsive, elastic (that is, scalable), resilient, and message-driven.

Having an application that is truly responsive is the foundational goal. Suppose you have an application that heavily depends on one big thread to handle user requests, and this thread

typically sends responses back to its originating requesters after doing its work. When the application gets more requests than it can handle, this thread will start to be a bottleneck and the application itself will not be able to be as responsive as it was before. To have the application be responsive, you need to make it scalable and resilient, because responsiveness is possible only with both scalability and resilience. Resilience occurs when an application exhibits features such as auto-recovery and self-healing. In most developers' experience, only a message-driven architecture can enable a scalable, resilient, and responsive application.

The reactive implementation might look more complicated at first glance, but after closer examination you will see that it's fairly straightforward.

Reactive programming has started to be baked into the bits of the Java 8 and Java EE 8 releases. The Java language introduced concepts such as `CompletionStage` and its implementation, `CompletableFuture`, and Java EE started to employ these features in specifications such as the Reactive Client API of JAX-RS.

JAX-RS 2.1 Reactive Client API

Let's look at how reactive programming can be used in Java EE 8 applications. To follow along, you'll need familiarity with the basic Java EE APIs.

JAX-RS 2.1 introduced a new way of creating a REST client with support for reactive programming. The default invoker implementation provided by JAX-RS is synchronous, which means the client that is created will make a blocking call to the server endpoint. An example for this implementation is shown in **Listing 1**.

Listing 1.

```
Response response =
    ClientBuilder.newClient()
                 .target("http://localhost:8080/service-url")
                 .request()
```

```
                .get();
```

As of version 2.0, JAX-RS provides support for creating an asynchronous invoker on the client API by just invoking the `async()` method, as shown in **Listing 2**.

Listing 2.

```
Future<Response> response =
    ClientBuilder.newClient()
                .target("http://localhost:8080/service-url")
                .request()
                .async()
                .get();
```

Using an asynchronous invoker on the client returns an instance of `Future` with type `javax.ws.rs.core.Response`. This would either result in polling the response, with a call to `future.get()`, or registering a callback that would be invoked when the HTTP response is available. Both of these implementation approaches are suitable for asynchronous programming, but things usually get complicated when you want to nest callbacks or you want to add conditional cases in those asynchronous execution flows.

JAX-RS 2.1 offers a reactive way to overcome these problems with the new JAX-RS Reactive Client API for building the client. It's as simple as invoking the `rx()` method while building the client. In **Listing 3**, the `rx()` method returns the reactive invoker that exists on the client's run-time and the client returns a response of type `CompletionStage.rx()`, which enables the switch from sync to async invoker by this simple invocation.

Listing 3.

```
CompletionStage<Response> response =
    ClientBuilder.newClient()
                .target("http://localhost:8080/service-url")
                .request()
```

```
                .rx()
                .get();
```

`CompletionStage<T>` is a new interface introduced in Java 8, and it represents a computation that can be a stage within a larger computation, as its name implies. It's the only reactive portion of Java 8 that made it into the JAX-RS.

After getting a response instance, I can just invoke `thenAcceptAsync()`, where I can provide the code snippet that would be executed asynchronously when the response becomes available, such as shown in **Listing 4**.

Listing 4.

```
response.thenAcceptAsync(res -> {
    Temperature t = res.readEntity(Temperature.class);
    //do stuff with t
});
```

Adding Reactive Goodness to a REST Endpoint

The reactive approach is not limited to the client side in JAX-RS; it's also possible to leverage it on the server side. To demonstrate this, I will first create a simple scenario where I can query a list of locations from one endpoint. For each location, I will make another call to another endpoint with that location data to get a temperature value. The interaction of the endpoints would be as shown in **Figure 1**.

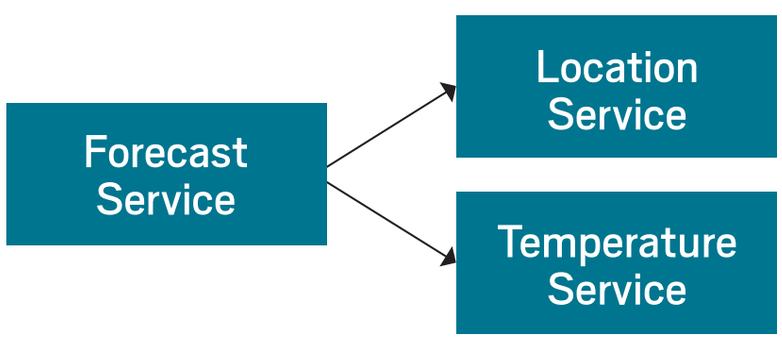


Figure 1. Interaction between endpoints

First, I simply define the domain model and then I define the services for each domain model. **Listing 5** defines the `Forecast` class, which wraps the `Temperature` and `Location` classes.

Listing 5.

```
public class Temperature {

    private Double temperature;
    private String scale;

    // getters & setters
}

public class Location {

    String name;

    public Location() {}

    public Location(String name) {
        this.name = name;
    }

    // getters & setters
}

public class Forecast {

    private Location location;
    private Temperature temperature;

    public Forecast(Location location) {
```

```
        this.location = location;
    }

public Forecast setTemperature(
                final Temperature temperature) {
        this.temperature = temperature;
        return this;
    }

    // getters
}
```

For wrapping a list of forecasts, the `ServiceResponse` class is implemented in **Listing 6**.

Listing 6.

```
public class ServiceResponse {

    private long processingTime;
    private List<Forecast> forecasts = new ArrayList<>();

    public void setProcessingTime(long processingTime) {
        this.processingTime = processingTime;
    }

    public ServiceResponse forecasts(
              List<Forecast> forecasts) {
        this.forecasts = forecasts;
        return this;
    }

    // getters
```

```
}
```

`LocationResource`, which is shown in **Listing 7**, defines three sample locations returned with the path `/location`.

Listing 7.

```
@Path("/location")
public class LocationResource {

    @GET
    @Produces(MediaType.APPLICATION_JSON)
    public Response getLocations() {

        List<Location> locations = new ArrayList<>();
        locations.add(new Location("London"));
        locations.add(new Location("Istanbul"));
        locations.add(new Location("Prague"));

        return Response.ok(
            new GenericEntity<List<Location>>(locations){})
            .build();
    }
}
```

`TemperatureResource`, shown in **Listing 8**, returns a randomly generated temperature value between 30 and 50 for a given location. A delay of 500 ms is added within the implementation to simulate the sensor reading.

Listing 8.

```
@Path("/temperature")
public class TemperatureResource {
```

```
@GET
@Path("/{city}")
@Produces(MediaType.APPLICATION_JSON)
public Response getAverageTemperature(
       @PathParam("city") String cityName) {

    Temperature temperature = new Temperature();
    temperature.setTemperature(
        (double) (new Random().nextInt(20)+30));
    temperature.setScale("Celsius");

    try {
        Thread.sleep(500);
    } catch (InterruptedException ignored) {}

        return Response.ok(temperature).build();
    }
}
```

I will first show the implementation for the synchronous `ForecastResource` (shown in **Listing 9**), which first fetches all locations. Then, for each location, it invokes the temperature service to retrieve the Celsius value.

Listing 9.

```
@Path("/forecast")
public class ForecastResource {

    @Uri("location")
    private WebTarget locationTarget;

    @Uri("temperature/{city}")
```

```
private WebTarget temperatureTarget;

@GET
@Produces(MediaType.APPLICATION_JSON)
public Response getLocationsWithTemperature() {
    long startTime = System.currentTimeMillis();
    ServiceResponse response = new ServiceResponse();

    List<Location> locations = locationTarget.request()
            .get(new GenericType<List<Location>>() {});

    locations.forEach(location -> {
        Temperature temperature = temperatureTarget
            .resolveTemplate("city", location.getName())
            .request()
            .get(Temperature.class);

        response.getForecasts().add(
            new Forecast(location)
                .setTemperature(temperature));
    });
    long endTime = System.currentTimeMillis();
    response.setProcessingTime(endTime - startTime);

    return Response.ok(response).build();
  }
}
```

When the forecast endpoint is requested as `/forecast`, you should see output similar to **Listing 10**. Notice that the processing time of the request took 1,533 ms, which makes sense because requesting temperature values for three different locations synchronously would add up to 1,500 ms.

Listing 10.

```
{
   "forecasts": [
      {
         "location": {
            "name": "London"
         },
         "temperature": {
            "scale": "Celsius",
            "temperature": 33
         }
      },
      {
         "location": {
            "name": "Istanbul"
         },
         "temperature": {
            "scale": "Celsius",
            "temperature": 38
         }
      },
      {
         "location": {
            "name": "Prague"
         },
         "temperature": {
            "scale": "Celsius",
            "temperature": 46
         }
      }
   ],
```

```
   "processingTime": 1533
}
```

So far, so good. Now it's time to introduce reactive programming on the server side, where a call for each location could be done in parallel after getting all the locations. This can definitely enhance the synchronous flow shown earlier. This is done in **Listing 11**, which defines a reactive version of this forecast service.

Listing 11.

```
@Path("/reactiveForecast")
public class ForecastReactiveResource {

    @Uri("location")
    private WebTarget locationTarget;

@Uri("temperature/{city}")
private WebTarget temperatureTarget;

@GET
@Produces(MediaType.APPLICATION_JSON)
public void getLocationsWithTemperature(
      @Suspended final AsyncResponse async) {

    long startTime = System.currentTimeMillis();

    // Create a stage on retrieving locations
    CompletionStage<List<Location>> locationCS =
        locationTarget.request()
            .rx()
            .get(new GenericType<List<Location>>() {});
```

```
// By composing another stage on the location stage
// created above, collect the list of forecasts
// as in one big completion stage
final CompletionStage<List<Forecast>> forecastCS =
locationCS.thenCompose(locations -> {

   // Create a stage for retrieving forecasts
   // as a list of completion stages
   List<CompletionStage<Forecast>> forecastList =

      // Stream locations and process each
      // location individually
      locations.stream().map(location -> {

         // Create a stage for fetching the
// temperature value just for one city
// given by its name
final CompletionStage<Temperature> tempCS =
    temperatureTarget
    .resolveTemplate("city",
                     location.getName())
    .request()
    .rx()
    .get(Temperature.class);

// Then create a completable future that
// contains an instance of forecast
// with location and temperature values
return CompletableFuture.completedFuture(
   new Forecast(location))
       .thenCombine(tempCS,
```

```
                        Forecast::setTemperature);
            }).collect(Collectors.toList());

            // Return a final completable future instance
            // when all provided completable futures are
            // completed
            return CompletableFuture.allOf(
               forecastList.toArray(
                  new CompletableFuture[forecastList.size()]))
                    .thenApply(v -> forecastList.stream()
                    .map(CompletionStage::toCompletableFuture)
                            .map(CompletableFuture::join)
                            .collect(Collectors.toList()));
        });

                    // Create an instance of ServiceResponse,
                    // which contains the whole list of forecasts
                    // along with the processing time.
                    // Create a completed future of it and combine to
                    // forecastCS in order to retrieve the forecasts
                    // and set into service response
                    CompletableFuture.completedFuture(
                         new ServiceResponse())
                            .thenCombine(forecastCS,
                            ServiceResponse::forecasts)
                             .whenCompleteAsync((response, throwable) -> {
                                response.setProcessingTime(
                                  System.currentTimeMillis() - startTime);
                                async.resume(response);
                             });
                }
            }
```

Reactive programming is more than enhancing the implementation from synchronous to asynchronous; it also eases development with concepts such as nesting stages.

The reactive implementation might look more complicated at first glance, but after closer examination you will see that it's fairly straightforward. Within the `ForecastReactiveResource` implementation, I first create a client invocation on the location services with the help of the JAX-RS Reactive Client API. As I mentioned previously, this is an addition to Java EE 8, and it helps to create a reactive invoker simply by use of the `rx()` method.

Now I compose another stage based on location to collect the list of forecasts. They will be stored in one big completion stage, named `forecastCS`, as a list of forecasts. I will ultimately create the response of the service call by using only `forecastCS`.

Let's continue by collecting the forecasts as a list of completion stages as defined in the `forecastList` variable. To create the completion stages for each forecast, I stream on the locations and then create the `tempCS` variable by again using the JAX-RS Reactive Client API, which will invoke the temperature service with city name. I use the `resolveTemplate()` method here to build a client, and that enables me to pass the name of the city to the builder as a parameter.

As a last step of streaming on locations, I do a call to `CompletableFuture.completedFuture()` by providing a newly created instance of `Forecast` as the parameter. I combine this future with the `tempCS` stage so that I have the temperature value for the iterated locations.

The `CompletableFuture.allOf()` method in **Listing 11** transforms the list of completion stages to `forecastCS`. Execution of this step returns the big completable future instance when all provided completable futures are completed.

The response from the service is an instance of the `ServiceResponse` class, so I create a completed future for that as well, and then I combine the `forecastCS` completion stage with the list of forecasts and calculate the response time of the service.

Of course, this reactive programming makes only the server side execute asynchronously; the client side will be blocked until the server sends the response back to the requester. In order to overcome this problem, Server Sent Events (SSEs) can also be used to partially send

the response once it's available so that for each location, the temperature values can be pushed to the client one by one. The output of `ForecastReactiveResource` will be something similar to **Listing 12**. As shown in the output, the processing time is 515 ms, which is the ideal execution time for retrieving a temperature value for one location.

Listing 12.

```
{
   "forecasts": [
      {
         "location": {
            "name": "London"
         },
         "temperature": {
            "scale": "Celsius",
            "temperature": 49
         }
      },
      {
         "location": {
            "name": "Istanbul"
         },
         "temperature": {
            "scale": "Celsius",
            "temperature": 32
         }
      },
      {
         "location": {
            "name": "Prague"
         },
         "temperature": {
```

```
            "scale": "Celsius",
            "temperature": 45
        }
      }
   ],
   "processingTime": 515
}
```

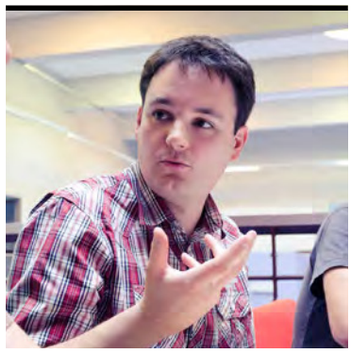
Conclusion

Throughout the examples in this article, I first showed the synchronous way to retrieve the forecast information by choreographing location and temperature services. Then I moved on to the reactive approach in order to have the asynchronous processing occur between service calls. When you leverage the use of the JAX-RS Reactive Client API of Java EE 8 and classes such as `CompletionStage` and `CompletableFuture` shipping with Java 8, the power of asynchronous processing is unleashed with the help of reactive-style programming.

Reactive programming is more than enhancing the implementation from a synchronous to an asynchronous model; it also eases development with concepts such as nesting stages. The more it is adopted, the easier it will be to handle complex scenarios in parallel programming. `</article>`

Mert Çalişkan (@mertcal) is a Java Champion and a coauthor of *PrimeFaces Cookbook* (Packt Publishing, 2013) and *Beginning Spring* (Wiley Publications, 2015). He currently is working on his latest book, *Java EE 8 Microservices*, and he works as a developer on the Payara Server inside the Payara Foundation.

CLEMENT ESCOFFIER



JULIEN PONGE

JULIEN PONGE PHOTOGRAPH BY MATT BOSTOCK/GETTY IMAGES

Going Reactive with Eclipse Vert.x and RxJava

Building responsive, scalable apps with one of the most popular reactive libraries

Eclipse Vert.x is a toolkit for implementing reactive and distributed systems on top of the JVM. It was designed from the start with a *reactive* design and *asynchrony* in mind. Vert.x is also about freedom. It does not tell you how to shape your system; you are in charge. Its extensive ecosystem provides everything you need to build responsive, distributed, and interactive applications. This article describes how Vert.x combines an asynchronous execution model and a reactive implementation to let you build applications that can handle uncertain and ever-evolving development needs.

What Does It Mean to Be Reactive?

Let's start from the beginning: what does *reactive* actually mean? The *Oxford English Dictionary* defines *reactive* as "showing a response to a stimulus." So, by extension, reactive software can be defined as *software that reacts to stimuli*. But using that definition, software has been reactive since the early age of computers. Software is designed to react to user demands such as input, clicks, commands, and so on.

However, with the rise of distributed systems, applications started reacting to messages sent by peers and by failure events. The recent reactive renaissance is mainly due to the difficulties of building robust distributed systems. As developers painfully learned, distributed systems are difficult, and they fail for many reasons such as capacity issues, network outages, hardware problems, and bugs. In response, a few years ago, the Reactive Manifesto defined *reactive systems* as distributed systems with the following characteristics:

- Message-driven: They use asynchronous message passing to communicate.

- Elastic: They stay responsive under varying workloads.
- Resilient: They stay responsive in the face of failure.
- Responsive: They respond in a timely manner.

This architectural style promotes a new way to build distributed systems, infusing asynchrony into the core of these systems. While reactive systems are described as "distributed systems done right," they can be difficult to build. Taming the asynchronous beast is particularly difficult from the developer standpoint. In addition, the traditional threading model (one thread per request) tends to create memory and CPU hogs, and, when dealing with asynchronous code, this approach is particularly inefficient.

Several development models have emerged to make the development of asynchronous applications easier, including actors, fibers, coroutines, and reactive programming. This article focuses on the latter.

Reactive programming (and its main derivative, Reactive eXtensions, or RX) is an asynchronous programming paradigm focused on the manipulation of data streams. It provides an API to compose asynchronous and event-driven applications. When using reactive programming, you are handling streams of data in which data flows. You observe these streams and react when new data is available.

But data streams have an inherent flaw. What happens if you receive too many messages and you can't process them in time? You could put a buffer between the source and the handler, but it would help only with handling small bumps. Dropping incoming data is also a solution, but that is not always acceptable. Ultimately, you need a way to control the pace. This is what the reactive streams specification proposes. It defines an asynchronous and nonblocking back-pressure protocol. In this flow of control, the consumer notifies the producer of its current capacity. So, the producer does not send too much data on the stream, and your system auto-adapts to its capacity without burning.

Why Do Reactive Systems Matter?

Why did reactive programming become so prevalent in the past few years? For a very long time, most applications have been developed using a synchronous execution model and

most APIs have been designed to follow this approach.

However, computer systems and distribution systems are asynchronous. Synchronous processing is a simplification made to provide ease of comprehension. For years, the asynchronous nature of systems has been ignored, and now it's time to catch up. Many modern applications are relying on I/O operations, such as remote invocations or access to the file system. Because of the synchronous nature of application code, however, these I/O operations are designed to be blocking, so the application waits for a response before it can continue its execution. To enable concurrency, the application relies on multithreading and increases the number of threads. But, threads are expensive. First, the code has to protect itself from concurrent access to its state. Second, threads are expensive in terms of memory and—often overlooked—in CPU time, because switching between threads requires CPU cycles.

Implementing reactive systems requires two shifts: an execution shift to use an asynchronous execution model and a development shift to write asynchronous APIs and applications.

Therefore, a more efficient model is needed. The asynchronous execution model promotes a task-based concurrency in which a task releases the thread when it cannot make progress anymore (for instance, it invokes a remote service using nonblocking I/O and will be notified when the result is available). Thus, the same thread can switch to another task. As a result, a single thread can handle several interleaved tasks.

Traditional development and execution paradigms are not able to exploit this new model. However, in a world of cloud and containers, where applications are massively distributed and interconnected and they must handle continuously growing traffic, the promise made by reactive systems is a perfect match. But, implementing reactive systems requires two shifts: an execution shift to use an asynchronous execution model and a development shift to write asynchronous APIs and applications. This is what Eclipse Vert.x offers. In the rest of this article, we present how Vert.x combines both to give you superpowers.

RxJava: The Reactive Programming Toolbox for Java

Let's focus on reactive programming—a development model for writing asynchronous code. When using reactive programming, the code manipulates streams of data. The data is generated by publishers. The data flows between a publisher and consumers, which process the data. Consumers observing a data stream are notified when a new item is available, when the stream completes, and when an error is caught. To avoid overloading consumers, a back-pressure protocol is required to control the amount of data flowing in the stream. This is generally handled transparently by the reactive framework.

There are several implementations of the reactive programming paradigm. RxJava is a straightforward implementation of reactive extensions (RX) for the Java programming language. It is a popular library for reactive programming that can be used to develop applications in networked data processing, graphical user interfaces with JavaFX, and Android apps. RxJava is the principal toolkit for reactive libraries in Java, and it provides five data types to describe data publishers depending on the types of data streams, as shown in **Table 1**.

These types represent data publishers and convey data processed by consumers observing them. Depending on the number of items flowing in the stream, the type is different. For streams with a bounded or unbounded sequence of items, the types `Observable` and `Flowable` are used.

The difference between `Observable` and `Flowable` is that `Flowable` handles back-pressure (that is, it implements a reactive streams protocol) while `Observable` does not. `Flowable` is better

USE CASE	NUMBER OF EXPECTED ITEMS IN THE STREAM	RXJAVA TYPES
NOTIFICATION, DATA FLOW	0..N	`Observable, Flowable`
ASYNCHRONOUS OPERATION PRODUCING (MAYBE) A RESULT	1..1 0..1	`Single` `Maybe`
ASYNCHRONOUS OPERATION THAT DOES NOT PRODUCE A RESULT	0	`Completable`

Table 1. RxJava reactive publisher types

suited for large streams of data coming from a source that supports back-pressure (for example, a TCP connection), while `Observable` is better suited at handling so-called "hot" observables for which back-pressure cannot be applied (such as GUI events and other user actions). It is important to note that not all streams can support back-pressure. In fact, most of the streams conveying data captured in the physical world are not capable of this. Reactive programming libraries propose strategies such as buffers and acceptable data loss for handling these cases.

Getting started with RxJava. It's time to see some code and make reactive clearer. The complete project source code is available online. Clone or download this project and check the content of the rxjava-samples subproject. It uses RxJava 2.x and the logback-classic logging library. You will see later how it helps you understand threading with RxJava.

In the previous section, we briefly examined the different reactive types proposed by RxJava. The following class creates instances of these types and applies some basic operations:

```
package samples;

import io.reactivex.Completable;
import io.reactivex.Flowable;
import io.reactivex.Maybe;
import io.reactivex.Single;
import io.reactivex.functions.Consumer;
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;

public class RxHello {

private static final Logger logger =
    LoggerFactory.getLogger(RxHello.class);

public static void main(String[] args) {
```

```
        Single.just(1)
            .map(i -> i * 10)
            .map(Object::toString)
            .subscribe((Consumer<String>) logger::info);

        Maybe.just("Something")
            .subscribe(logger::info);

        Maybe.never()
            .subscribe(o -> logger.info("Something is here..."));

        Completable.complete()
            .subscribe(() -> logger.info("Completed"));

        Flowable.just("foo", "bar", "baz")
                .filter(s -> s.startsWith("b"))
                .map(String::toUpperCase)
                .subscribe(logger::info);
        }
    }
```

Running this example yields output similar to this:

```
11:24:28.638 [main] INFO samples.RxHello - 10
11:24:28.661 [main] INFO samples.RxHello - Something
11:24:28.672 [main] INFO samples.RxHello - Completed
11:24:28.716 [main] INFO samples.RxHello - BAR
11:24:28.716 [main] INFO samples.RxHello - BAZ
```

It is important to note that as with Java collection streams, no processing happens until an end event takes place. In RxJava, that event is a *subscription*. In this example, we used `subscribe()`

with a single parameter, which is a lambda called to receive each event. The following are other forms of `Subscribe` depending on the events the consumer wants to receive:

- No arguments, which just triggers the processing
- Two arguments to process events and errors
- Three arguments to process events, to process errors, and to provide notification when the processing is complete

Creating publishers and recovering from errors. Of course, RxJava would be quite limited if creating data streams such as `Observable`s were limited to calling the `just()` factory method as we did in the previous example. All types of publishers support a `create()` method to define the code to deal with new subscribers:

```
List<String> data =
    Arrays.asList("foo", "bar", "baz");
Random random = new Random();

Observable<String> source =
    Observable.create(subscriber -> {
        for (String s : data) {
            if (random.nextInt(6) == 0) {
                subscriber.onError(
                    new RuntimeException("Bad luck for you..."));
            }
            subscriber.onNext(s);
        }
    subscriber.onComplete();
});
```

The example above creates an `Observable` of `String` values (in other words, a stream of `String` values), where the values are being picked from a predefined list. We also introduced random failures. The following three methods can be used to notify subscribers:

- `onNext`, when a new value is sent to the subscriber, possibly passing through intermediate operators before it reaches the subscriber
- `onComplete` to indicate that no more values will be sent
- `onError` to indicate that an error happened and that no further value will be sent; any `Throwable` can be used as an error value

Note that `create()` is not the only way to define custom publishers, but presenting all options would be outside the scope of this article.

Because there is a good probability that errors will happen, we can test this `Observable` 10 times:

```
for (int i = 0; i < 10; i++) {
    logger.info("====================================");
    source.subscribe(
        next -> logger.info("Next: {}", next),
        error -> logger.error("Whoops"),
        () -> logger.info("Done"));
}
```

We can observe successful completions as well as errors in the execution traces:

```
11:51:47.469 [main] INFO samples.RxCreateObservable -
====================================
11:51:47.469 [main] INFO samples.RxCreateObservable - Next: foo
11:51:47.469 [main] INFO samples.RxCreateObservable - Next: bar
11:51:47.469 [main] INFO samples.RxCreateObservable - Next: baz
11:51:47.469 [main] INFO samples.RxCreateObservable - Done
11:51:47.469 [main] INFO samples.RxCreateObservable -
====================================
11:51:47.469 [main] INFO samples.RxCreateObservable - Next: foo
11:51:47.469 [main] INFO samples.RxCreateObservable - Next: bar
```

```
11:51:47.469 [main] ERROR samples.RxCreateObservable - Whoops
11:51:47.469 [main] INFO samples.RxCreateObservable -
=====================================
11:51:47.469 [main] INFO samples.RxCreateObservable - Next: foo
11:51:47.469 [main] ERROR samples.RxCreateObservable - Whoops
```

RxJava supports various ways to recover from errors, such as switching to another stream or providing a default value. Another option is to use `retry()`:

```
source
    .retry(5)
    .subscribe(next -> logger.info("Next: {}", next),
        error -> logger.error("Whoops"),
        () -> logger.info("Done"));
```

Above, we specified that in case of error, we should retry at most five times with new subscriptions. Note that retries might use another thread for execution. Because errors are random, your exact output trace will vary across executions, but the following output shows an example of retries:

```
11:51:47.472 [main] INFO samples.RxCreateObservable - Next: foo
11:51:47.472 [main] INFO samples.RxCreateObservable - Next: bar
11:51:47.472 [main] INFO samples.RxCreateObservable - Next: foo
11:51:47.472 [main] INFO samples.RxCreateObservable - Next: bar
11:51:47.472 [main] INFO samples.RxCreateObservable - Next: baz
11:51:47.472 [main] INFO samples.RxCreateObservable - Done
```

RxJava and threads. So far, we haven't cared much about multithreading. Let's take another example and run it:

```
Flowable.range(1, 5)
```

```
        .map(i -> i * 10)
        .map(i -> {
            logger.info("map({})", i);
            return i.toString();
        })
        .subscribe(logger::info);

Thread.sleep(1000);
```

You can see from the logs that all processing happens on the `main` thread:

```
12:01:01.097 [main] INFO samples.RxThreading - map(10)
12:01:01.100 [main] INFO samples.RxThreading - 10
12:01:01.100 [main] INFO samples.RxThreading - map(20)
12:01:01.100 [main] INFO samples.RxThreading - 20
12:01:01.100 [main] INFO samples.RxThreading - map(30)
12:01:01.100 [main] INFO samples.RxThreading - 30
12:01:01.100 [main] INFO samples.RxThreading - map(40)
12:01:01.100 [main] INFO samples.RxThreading - 40
12:01:01.100 [main] INFO samples.RxThreading - map(50)
12:01:01.100 [main] INFO samples.RxThreading - 50
```

In fact, both the operator processing and the subscriber notifications happen from that `main` thread. By default, a publisher (and the chain of operators that you apply to it) will do its work, and will notify its consumers, on the same thread on which its `subscribe` method is called. RxJava offers `Schedulers` to offload work to specialized threads and executors. `Schedulers` are responsible for notifying the subscribers on the correct thread even if it's not the thread used to call `subscribe`.

The `io.reactivex.schedulers.Schedulers` class offers several schedulers, with the most interesting being these:

- `computation()` for CPU-intensive work with no blocking I/O operations
- `io()` for all blocking I/O operations
- `single()`, which is a shared thread for operations to execute in order
- `from(executor)` to offload all scheduled work to a custom executor

Now, back to our previous example, we can specify how the subscription and observation will be scheduled:

```
Flowable.range(1, 5)
    .map(i -> i * 10)
    .map(i -> {
        logger.info("map({})", i);
        return i.toString();
     })
    .observeOn(Schedulers.single())
    .subscribeOn(Schedulers.computation())
    .subscribe(logger::info);

Thread.sleep(1000);
logger.info("================================");
```

The `subscribeOn` method specifies the scheduling for the subscription and operator processing, while the `observeOn` method specifies the scheduling for observing the events. In this example, the `map` operations are invoked on the computation thread pool while the `subscribe` callback (`logger::info`) is invoked by a different thread (which does not change). Running the example gives an execution trace where you clearly see different threads being involved:

```
12:01:03.127 [RxComputationThreadPool-1] INFO
samples.RxThreading - map(10)
12:01:03.128 [RxComputationThreadPool-1] INFO
samples.RxThreading - map(20)
```

```
12:01:03.128 [RxSingleScheduler-1] INFO
samples.RxThreading - 10
12:01:03.128 [RxComputationThreadPool-1] INFO
samples.RxThreading - map(30)
12:01:03.128 [RxSingleScheduler-1] INFO
samples.RxThreading - 20
12:01:03.128 [RxComputationThreadPool-1] INFO
samples.RxThreading - map(40)
12:01:03.128 [RxSingleScheduler-1] INFO
samples.RxThreading - 30
12:01:03.128 [RxSingleScheduler-1] INFO
samples.RxThreading - 40
12:01:03.128 [RxComputationThreadPool-1] INFO
samples.RxThreading - map(50)
12:01:03.128 [RxSingleScheduler-1] INFO
samples.RxThreading - 50
12:01:04.127 [main] INFO
samples.RxThreading
===================================
```

Combining observables. RxJava offers many ways to combine streams. We'll illustrate that with the `merge` and `zip` operations. Merging streams provides a single stream that mixes elements from the various sources, as the following example shows:

```
package samples;

import io.reactivex.Flowable;
import io.reactivex.schedulers.Schedulers;
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
```

```
import java.util.UUID;
import java.util.concurrent.TimeUnit;

public class RxMerge {

    private static final Logger logger =
        LoggerFactory.getLogger(RxMerge.class);

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

        Flowable<String> intervals = Flowable
            .interval(100, TimeUnit.MILLISECONDS,
                      Schedulers.computation())
            .limit(10)
            .map(tick -> "Tick #" + tick)
            .subscribeOn(Schedulers.computation());

Flowable<String> strings = Flowable.just(
        "abc", "def", "ghi", "jkl")
    .subscribeOn(Schedulers.computation());

Flowable<Object> uuids = Flowable
   .generate(emitter -> emitter.onNext(UUID.randomUUID()))
   .limit(10)
   .subscribeOn(Schedulers.computation());

Flowable.merge(strings, intervals, uuids)
    .subscribe(obj -> logger.info("Received: {}", obj));

Thread.sleep(3000);
```

```
  }
}
```

Running this example gives a trace in which elements from the various sources may be interleaved. Another useful option is `zip()`, which takes elements from various sources and assembles them:

```
Flowable.zip(intervals, uuids, strings,
    (i, u, s) -> String.format("%s {%s} -> %s", i, u, s))
        .subscribe(obj -> logger.info("Received: {}", obj));
```

It produces a trace similar to this:

```
14:32:40.127 [RxComputationThreadPool-7] INFO
samples.RxMerge - Received: Tick #0
{67e7cde0-3f29-49cb-b569-e01474676d98} -> abc
14:32:40.224 [RxComputationThreadPool-7] INFO
samples.RxMerge - Received: Tick #1
{a0a0cc83-4bed-4793-9ee0-11baa7707610} -> def
14:32:40.324 [RxComputationThreadPool-7] INFO
samples.RxMerge - Received: Tick #2
{7b7d81b6-cc39-4ec0-a174-fbd61b1d5c71} -> ghi
14:32:40.424 [RxComputationThreadPool-7] INFO
samples.RxMerge - Received: Tick #3
{ae88eb02-52a5-4af7-b9cf-54b29b9cdb85} -> jkl
```

In real-world scenarios, `zip()` is useful for gathering data from other parties, such as services, and then producing a result based on what was received.

Implementing Reactive Systems with Reactive Programming

While reactive programming lets you compose asynchronous and event-driven applications, don't lose sight of the overall goal. To successfully build responsive distributed systems in a world of cloud and containers, embracing the asynchronous execution model is essential. Reactive programming addresses the asynchronous development model, but you still need a task-based concurrency model and nonblocking I/O. Eclipse Vert.x provides these two missing pieces as well as RxJava-friendly APIs.

The Vert.x execution model is based on the concept of an *event loop*. An event loop is a thread consuming events from a queue. For each event, it looks for a handler interested in the event and calls it. Handlers are methods that receive an event as a parameter. In this model, your code can be single-threaded while handling lots of concurrent and entangled tasks. However, this approach comes with some drawbacks. The executed handlers must *never* block the event loop: if they do, the system loses its responsiveness and the number of unprocessed events in the queue rises.

Fortunately, Vert.x comes with a large ecosystem for implementing almost anything in an asynchronous and nonblocking way. For instance, Vert.x provides building blocks for building modern web applications, accessing databases, and interacting with legacy systems. Let's look at a few examples. The Vert.x "hello world" application (code available online) is the following:

```
package samples;

import io.vertx.core.Vertx;

public class HttpApplication {

    public static void main(String[] args) {
        // 1 - Create a Vert.x instance
        Vertx vertx = Vertx.vertx();

        // 2 - Create the HTTP server
```

```
        vertx.createHttpServer()
            // 3 - Attach a request handler processing the requests
            .requestHandler(req -> req.response()
                    .end("Hello, request handled from "
                        + Thread.currentThread().getName()))
            // 4 - Start the server on the port 8080
            .listen(8080);
    }
}
```

For each incoming HTTP request (event), the request handler is called. Notice that the handler is always called by the same thread: the event loop thread. Now, if you want to call another service (using HTTP) in the request handler, you would do something like this:

```
package samples;

import io.vertx.core.Vertx;
import io.vertx.ext.web.client.WebClient;

public class TwitterFeedApplication {

    public static void main(String[] args) {
        Vertx vertx = Vertx.vertx();
        // 1 - Create a Web client
        WebClient client = WebClient.create(vertx);
        vertx.createHttpServer()
            .requestHandler(req -> {
                // 2 - In the request handler, retrieve a Twitter feed
                client
                    .getAbs("https://twitter.com/vertx_project")
                    .send(res -> {
```

```
                                    // 3 - Write the response based on the result
                                    if (res.failed()) {
                                        req.response().end("Cannot access "
                                            + "the twitter feed: "
                                            + res.cause().getMessage());
                                    } else {
                                        req.response().end(res.result()
                                            .bodyAsString());
                                    }
                                });
                        })
                        .listen(8080);
            }
        }
```

This example relies on the Vert.x nonblocking I/O, so the entire code runs on the Vert.x event loop (in a single-thread manner). This does not prevent handling concurrent requests. It's actually the opposite; a single thread handles all the requests. However, you can quickly see the issue: the code becomes difficult to understand because of the nested callbacks. This is where RxJava comes into play. The previous code can be rewritten as follows:

```
package samples;

import io.vertx.reactivex.core.Vertx;
import io.vertx.reactivex.core.http.HttpServer;
import io.vertx.reactivex.ext.web.client.HttpResponse;
import io.vertx.reactivex.ext.web.client.WebClient;

public class RXTwitterFeedApplication {

    public static void main(String[] args) {
```

```
Vertx vertx = Vertx.vertx();
WebClient client = WebClient.create(vertx);
HttpServer server = vertx.createHttpServer();
server
    // 1 - Transform the sequence of request into a stream
    .requestStream().toFlowable()
    // 2 - For each request, call the twitter API
    .flatMapCompletable(req ->
        client.getAbs("https://twitter.com/vertx_project")
            .rxSend()
            // 3 - Extract the body as string
            .map(HttpResponse::bodyAsString)
            // 4 - In case of a failure
            .onErrorReturn(t -> "Cannot access the twitter " +
                "feed: " + t.getMessage())
            // 5 - Write the response
            .doOnSuccess(res -> req.response().end(res))
            // 6 - Just transform the restul into a completable
            .toCompletable()
        )
        // 7 - Never forget to subscribe to a reactive type,
        // or nothing happens
        .subscribe();

    server.listen(8080);
  }
}
```

By restructuring the code around the RxJava reactive types, you benefit from the RxJava operators.

Implementing a Reactive Edge Service

Let's look at another simple yet effective example. Suppose that you have three services offering bids, and you want to offer an edge service to select the best offer at a point in time. Let these services offer simple HTTP/JSON endpoints. Obviously in real-world scenarios, these services might fail temporarily, and their response times might greatly vary.

We will simulate such a system by developing the following:

- A bidding service, with artificial delays and random errors
- An edge service to query services through HTTP

By using RxJava, we can show how to combine request streams, deal with failures, and provide time-bound guarantees for returning the best offer. All verticles will be deployed within the same application as we are prototyping, but this does not result in any loss of generality. The complete code is available in the vertx-samples subproject.

Instead of starting the application using a `main` method, we are going to use *verticles*. A verticle is a chunk of code, generally a Java class, that is deployed and run by Vert.x. Verticles are simple and scalable, and they use an actor-like deployment and concurrency model. They let you organize your code into a set of loosely coupled components. By default, verticles are executed by the event loop and observe different types of events (HTTP requests, TCP frames, messages, and so on). When the application starts, it instructs Vert.x to deploy a set of verticles.

Bidding service verticle. The verticle is designed with the HTTP port being configurable, as follows:

```
public class BiddingServiceVerticle extends AbstractVerticle {

    private final Logger logger =
        LoggerFactory.getLogger(BiddingServiceVerticle.class);

    @Override
    public void start(Future<Void> verticleStartFuture) throws Exception {
        Random random = new Random();
```

```
        String myId = UUID.randomUUID().toString();
        int portNumber = config().getInteger("port", 3000);

       // (...)
    }
}
```

The `config()` method provides access to a verticle configuration, and accessor methods such as `getInteger` support a default value as a second argument. So here, the default HTTP port is 3000. The service has a random UUID to identify its endpoint in responses, and it makes use of a random number generator.

The next step is to use the Vert.x web router to accept HTTP GET requests on path `/offer`:

```
Router router = Router.router(vertx);
router.get("/offer").handler(context -> {
String clientIdHeader = context.request()
    .getHeader("Client-Request-Id");
String clientId =
    (clientIdHeader != null) ? clientIdHeader : "N/A";
int myBid = 10 + random.nextInt(20);
JsonObject payload = new JsonObject()
    .put("origin", myId)
    .put("bid", myBid);
if (clientIdHeader != null) {
    payload.put("clientRequestId", clientId);
}
long artificialDelay = random.nextInt(1000);
vertx.setTimer(artificialDelay, id -> {
    if (random.nextInt(20) == 1) {
        context.response()
            .setStatusCode(500)
```

```
                        .end();
                    logger.error("{} injects an error (client-id={}, "
                        + "artificialDelay={})",
                        myId, myBid, clientId, artificialDelay);
                } else {
                    context.response()
                        .putHeader("Content-Type",
                            "application/json")
                        .end(payload.encode());
                    logger.info("{} offers {} (client-id={}, " +
                        "artificialDelay={})",
                        myId, myBid, clientId, artificialDelay);
                }
            });
        });
```

Note that to simulate failures, we built in a 5 percent chance of failure (in which case, the service issues an HTTP 500 response) and the final HTTP response is delayed by using a random timer between 0 and 1,000 milliseconds.

Finally, the HTTP server is started as usual:

```
vertx.createHttpServer()
    .requestHandler(router::accept)
    .listen(portNumber, ar -> {
        if (ar.succeeded()) {
            logger.info("Bidding service listening on HTTP " +
                "port {}", portNumber);
            verticleStartFuture.complete();
        } else {
            logger.error("Bidding service failed to start",
                ar.cause());
```

```
            verticleStartFuture.fail(ar.cause());
        }
    });
```

Edge service: selecting the best offer. This service is implemented using the RxJava API provided by Vert.x. Here are the preamble and the `start` method of the verticle class:

```
public class BestOfferServiceVerticle extends AbstractVerticle {

    private static final JsonArray DEFAULT_TARGETS = new JsonArray()
        .add(new JsonObject()
            .put("host", "localhost")
            .put("port", 3000)
            .put("path", "/offer"))
        .add(new JsonObject()
            .put("host", "localhost")
            .put("port", 3001)
            .put("path", "/offer"))
        .add(new JsonObject()
            .put("host", "localhost")
            .put("port", 3002)
            .put("path", "/offer"));
    private final Logger logger = LoggerFactory
        .getLogger(BestOfferServiceVerticle.class);
    private List<JsonObject> targets;
    private WebClient webClient;

    @Override
    public void start(Future<Void> startFuture) throws Exception {
        webClient = WebClient.create(vertx);
```

```
targets = config().getJsonArray("targets",
    DEFAULT_TARGETS)
    .stream()
    .map(JsonObject.class::cast)
    .collect(Collectors.toList());

vertx.createHttpServer()
    .requestHandler(this::findBestOffer)
    .rxListen(8080)
    .subscribe((server, error) -> {
        if (error != null) {
            logger.error("Could not start the best offer " +
                "service", error);
            startFuture.fail(error);
        } else {
            logger.info("The best offer service is running " +
                "on port 8080");
            startFuture.complete();
        }
    });
}
```

There are several interesting points in this code:

- To access the RxJava API offered by Vert.x, we import and extend the `io.vertx.reactivex.core.AbstractVerticle` class.
- It is possible to specify the target services, with the defaults being on the local host and ports 3000, 3001, and 3002. Such configuration can be passed as a JSON array containing JSON objects with `host`, `port`, and `path` keys.
- Variants of the Vert.x APIs that return RxJava objects are prefixed with "rx": here `rxListen` returns a `Single<HttpServer>`. The server is not actually started until we subscribe.

We can now focus on the implementation of the `findBestOffer` method. It first issues HTTP requests to each service, obtaining a list of `Single<JsonObject>` responses, and then it reduces them to the single, best response and eventually ends the HTTP response:

```
private final AtomicLong requestIds = new AtomicLong();
private static final JsonObject EMPTY_RESPONSE = new JsonObject()
    .put("empty", true)
    .put("bid", Integer.MAX_VALUE);

private void findBestOffer(HttpServerRequest request) {
    String requestId = String.valueOf(requestIds.getAndIncrement());

    List<Single<JsonObject>> responses = targets.stream()
        .map(t -> webClient
            .get(t.getInteger("port"),
                t.getString("host"),
                t.getString("path"))
        .putHeader("Client-Request-Id",
            String.valueOf(requestId))
        .as(BodyCodec.jsonObject())
        .rxSend()
        .retry(1)
        .timeout(500, TimeUnit.MILLISECONDS,
            RxHelper.scheduler(vertx))
        .map(HttpResponse::body)
        .map(body -> {
            logger.info("#{} received offer {}", requestId,
                body.encodePrettily());
            return body;
        })
        .onErrorReturnItem(EMPTY_RESPONSE))
```

```
            .collect(Collectors.toList());

        Single.merge(responses)
            .reduce((acc, next) -> {
                if (next.containsKey("bid") && isHigher(acc, next)) {
                    return next;
                }
                return acc;
            })
            .flatMapSingle(best -> {
                if (!best.containsKey("empty")) {
                    return Single.just(best);
                } else {
                    return Single.error(new Exception("No offer " +
                        "could be found for requestId=" + requestId));
                }
            })
            .subscribe(best -> {
                logger.info("#{} best offer: {}", requestId,
                    best.encodePrettily());
                request.response()
                    .putHeader("Content-Type",
                        "application/json")
                    .end(best.encode());
            }, error -> {
                logger.error("#{} ends in error", requestId, error);
                request.response()
                    .setStatusCode(502)
                    .end();
            });
    }
```

It is interesting to note the following for each HTTP request:

- The response is converted to a `JsonObject` using the `as()` method.
- A retry is attempted if the service issued an error.
- The processing times out after 500 milliseconds before returning an empty response, which is how we avoid waiting for all responses and errors to arrive.

Note that all RxJava operations that expect a scheduler can use `RxHelper::scheduler` to ensure that all events remain processed on Vert.x event loops.

The whole processing is just a matter of composing functional idioms such as `map`, `flatMap`, and `reduce` and handling errors with a default value. If no service can deliver a bid within 500 milliseconds, no offer is being made, resulting in an HTTP 502 error. Otherwise, the best offer is selected among the responses received.

Deploying verticles and interacting with the services. The main verticle code is as follows:

```
public class MainVerticle extends AbstractVerticle {

  @Override
  public void start() {
    vertx.deployVerticle(new BiddingServiceVerticle());

    vertx.deployVerticle(new BiddingServiceVerticle(),
                         new DeploymentOptions().setConfig(
                             new JsonObject().put("port", 3001)));

    vertx.deployVerticle(new BiddingServiceVerticle(),
                         new DeploymentOptions().setConfig(
                             new JsonObject().put("port", 3002)));

    vertx.deployVerticle("samples.BestOfferServiceVerticle",
                         new DeploymentOptions().setInstances(2));
  }
}
```

We deploy the bidding service three times on different ports to simulate three services, passing the HTTP port those services should listen on in the JSON configuration. We also deploy the edge service verticle with two instances to process the incoming traffic on two CPU cores rather than one. The two instances will listen on the same HTTP port, but note that there will be no conflict because Vert.x distributes the traffic in a round-robin fashion.

We can now interact with the HTTP services, for instance, by using the HTTPie command-line tool. Let's talk to the service on port 3000:

```
$ http GET localhost:3000/offer 'Client-Request-Id:1234' --verbose
GET /offer HTTP/1.1
Accept: */*
Accept-Encoding: gzip, deflate
Client-Request-Id: 1234
Connection: keep-alive
Host: localhost:3000
User-Agent: HTTPie/0.9.9

HTTP/1.1 200 OK
Content-Length: 83
Content-Type: application/json

{
        "bid": 21,
        "clientRequestId": "1234",
        "origin": "fe299565-34be-4a7b-ac09-d88fcc1e42e2"
}
```

The logs reveal both artificial delays and errors:

```
[INFO] 16:08:03.443 [vert.x-eventloop-thread-1] ERROR
samples.BiddingServiceVerticle -
```

```
6358300b-3f2d-40be-93db-789f0f1cde17 injects an error (
client-id=1234, artificialDelay=N/A)

[INFO] 16:11:10.644 [vert.x-eventloop-thread-1]
INFO  samples.BiddingServiceVerticle -
6358300b-3f2d-40be-93db-789f0f1cde17 offers 10 (
client-id=1234, artificialDelay=934)
```

Similarly, you can play with the edge service, observe responses, and check the logs to see how a response is being assembled. Sometimes you will get an error:

```
$ http GET localhost:8080 'Client-Request-Id:1234'
HTTP/1.1 502 Bad Gateway
Content-Length: 0
```

This is because all responses took longer than 500 milliseconds to arrive and some services injected an error:

```
[INFO] 16:12:51.869 [vert.x-eventloop-thread-2]
INFO  samples.BiddingServiceVerticle -
d803c4dd-1e9e-4f76-9029-770366e82615 offers 16 (
client-id=0, artificialDelay=656)
[INFO] 16:12:51.935 [vert.x-eventloop-thread-1]
INFO  samples.BiddingServiceVerticle -
6358300b-3f2d-40be-93db-789f0f1cde17 offers 17 (
client-id=0, artificialDelay=724)
[INFO] 16:12:52.006 [vert.x-eventloop-thread-3]
INFO  samples.BiddingServiceVerticle -
966e8334-4543-463e-8348-c6ead441c7da offers 14 (
client-id=0, artificialDelay=792)
```

Sometimes you will observe that only one or two responses have been taken into account.

The key point in this sample is that the combination of Vert.x and RxJava offers a declarative and functional model for describing how to perform and process a flexible number of network requests while remaining purely driven by asynchronous events.

Conclusion

In this article, you have seen how Eclipse Vert.x combines reactive programming and the asynchronous execution model to build reactive systems. Reactive programming lets you compose asynchronous and event-driven applications by manipulating and combining data streams. Modern reactive programming libraries such as RxJava implement reactive streams to handle back-pressure. However, a reactive approach is not limited to reactive programming. Don't lose sight that you want to build better systems that are responsive, robust, and interactive. By using the execution model and nonblocking I/O capabilities promoted by Vert.x, you are on the path to becoming truly reactive.

This article just scratched the surface. Vert.x gives you significant power and agility to create compelling, scalable, twenty-first-century applications the way you want to. Whether it's simple network utilities, sophisticated modern web applications, HTTP/REST microservices, high-volume event processing, or a full-blown back-end message-bus application, Vert.x is a great fit. `</article>`

Clement Escoffier (@clementplop) is a principal software engineer at Red Hat, where he is working as a Vert.x core developer. He has been involved in projects and products touching many domains and technologies such as OSGi, mobile app development, continuous delivery, and DevOps. Escoffier is an active contributor to many open source projects, including Apache Felix, iPOJO, Wisdom Framework, and Eclipse Vert.x.

Julien Ponge (@jponge) is an associate professor at INSA Lyon and a researcher at the CITI-INRIA laboratory. He is a longtime open source developer, having created IzPack and the Golo programming language, and is now a member of the Eclipse Vert.x team. Ponge is currently on leave from INSA and working as a delegated consultant to Red Hat on the Vert.x project.

JOSH LONG

Reactive Spring

Proceeding from fundamentals, use the Spring Framework to quickly build a reactive application.

Reactive programming is an approach to writing software that embraces asynchronous I/O. Asynchronous I/O is a small idea that portends big changes for software. The idea is simple: alleviate inefficient resource utilization by using resources that would otherwise sit idle as they waited for I/O activity. Asynchronous I/O inverts the normal design of I/O processing: clients are notified of new data instead of asking for it. This approach frees the client to do other things while waiting for new notifications.

There is, of course, always the risk that too many notifications will overwhelm a client; so, a client must be able to push back, rejecting work it can't handle. This is a fundamental aspect of flow control in distributed systems. In reactive programming, the ability of the client to signal how much work it can manage is called *back-pressure*.

Many projects, such as Akka Streams, Vert.x, and RxJava, support reactive programming. [Vert.x and RxJava are examined in detail in the accompanying article, "Going Reactive with Eclipse Vert.x and RxJava," on page 32. —*Ed.*] The Spring team has a project called Reactor, which provides reactive capabilities for the Spring Framework. There's common ground across these different approaches, which has been summarized in the Reactive Streams initiative—an informal standard of sorts.

The Fundamental Data Types

The Reactive Streams initiative defines four data types. `Publisher` is a producer of values that might eventually arrive. A `Publisher` produces values of type `T`, as shown in **Listing 1**.

Listing 1: The Reactive Streams `Publisher<T>`

```
package org.reactivestreams;

public interface Publisher<T> {
    void subscribe(Subscriber<? Super T> s);
}
```

A `Subscriber` subscribes to a `Publisher`, receiving notifications on any new values of type `T`, as shown in **Listing 2**.

Listing 2: The Reactive Streams `Subscriber`

```
package org.reactivestreams;

    public interface Subscriber<T> {
    public void onSubscribe(Subscription s);
    public void onNext(T t);
    public void onError(Throwable t);
    public void onComplete();
}
```

When a `Subscriber` subscribes to a `Publisher`, it results in a `Subscription`, as shown in **Listing 3**.

Listing 3: The Reactive Streams `Subscription`

```
package org.reactivestreams;

public interface Subscription {
    public void request(long n);
    public void cancel();
}
```

A `Publisher` that is also a `Subscriber` is called a `Processor`, which is shown in **Listing 4**.

Listing 4: The Reactive Streams `Processor`

```
package org.reactivestreams;

public interface Processor<T, R> extends Subscriber<T>, Publisher<R> {
}
```

The specification is not meant to be a prescription for the implementations; instead, it defines types for interoperability. The Reactive Streams types eventually found their way into Java 9 as one-to-one semantically equivalent interfaces in the `java.util.concurrent.Flow` class.

Reactor

The Reactive Streams types are not enough; you'll need higher-order implementations to support operators such as filtering and transformation. Pivotal's Reactor project is a good choice here; it builds on top of the Reactive Streams specification. It provides two specializations of `Publisher<T>`. The first, `Flux`, is a `Publisher` that produces zero or more values. It's unbounded. The second, `Mono<T>`, is a `Publisher` that produces one or zero values. They're both publishers and you can treat them that way, but they go much further than the Reactive Streams specification. They both provide ways to process a stream of values. Reactor types compose nicely: the output of one thing can be the input to another.

Reactive Spring

As useful as project Reactor is, it's only a foundation. Applications need to talk to data sources. They need to produce and consume HTTP, Server-Sent Events (SSE), or WebSocket endpoints. They support authentication and authorization. Spring Framework 5.0 provides these things. It was released in September 2017 and builds on Reactor and the Reactive Streams specification. It includes a new reactive runtime and component model called Spring WebFlux. Spring WebFlux does not depend on or require the Servlet APIs to work. It ships with adapters that allow it to

work on top of a servlet engine, if need be, but that is not required. It also provides a Netty-based web server. Spring Framework 5, which works with a baseline of Java 8 and Java EE 7, is the foundation for changes in much of the Spring ecosystem. Let's look at an example.

Example Application

Let's build a simple Spring Boot 2.0 application that represents a service to manage books. You could call the project Library or something like that. Go to the Spring Initializr. Make sure that some version of Spring Boot 2.0 (or later) is selected in the version drop-down menu. You're writing a service to manage access to books in the library, so give this project the artifact ID `library-service`. Select the elements you'll need: Reactive Web, Actuator, Reactive MongoDB, Reactive Security, and Lombok.

I often use the Kotlin language, even if most of the project I am building is in Java. I keep Java artifacts collocated in a Kotlin project. Click Generate and it'll download an archive. Unzip it and open it in your favorite IDE that supports Java 8 (or later), Kotlin (optionally), and Maven. While you could have chosen Gradle in the Spring Initializr, I chose Maven for the purposes of this article. The stock standard Spring Boot application has an entry class that looks like Listing 5.

Listing 5: The empty husk of a new Spring Boot project

```
package com.example.libraryservice;

import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;

@SpringBootApplication public class LibraryServiceApplication {
    public static void main(String[] args) {
        System.setProperty("spring.profiles.active",
                           "security,authorization,frpjava");
        SpringApplication.run(LibraryServiceApplication.class, args);
```

```
    }
}
```

Data Access with Reactive Spring Data Modules

The most recent release of Spring Data debuts support for reactive data access when that is supported in the underlying datastores (such as MongoDB, Cassandra, Redis, and Couchbase). The release also introduces new reactive repository and template implementations. Because you have the reactive MongoDB driver and Spring Data module on the classpath, let's use them to manage some data. Create a new entity called `Book`, as shown in **Listing 6**.

Listing 6: A MongoDB `@Document` **entity,** `Book`

```
package com.example.libraryservice;

import lombok.AllArgsConstructor;
import lombok.Data;
import lombok.NoArgsConstructor;
import org.springframework.data.annotation.Id;
import org.springframework.data.mongodb.core.mapping.Document;

@Document
@Data
@AllArgsConstructor
@NoArgsConstructor
public class Book {
    @Id
    private String id;
    private String title;
    private String author;
}
```

Next, create a Spring Data repository to support the data management lifecycle of the entity. This should look very familiar to anyone who has ever used Spring Data, except that the repository supports reactive interactions: methods return `Publisher` types, and input can be given as `Publisher` instances. See **Listing 7**.

Listing 7: A reactive Spring Data MongoDB repository

```
package com.example.libraryservice;
import org.springframework.data.mongodb.repository.ReactiveMongoRepository;
import reactor.core.publisher.Flux;

public interface BookRepository extends ReactiveMongoRepository {
    Flux findByAuthor(String author);
}
```

Install Some Sample Data

With that, you now have enough to install some sample data (just for your demo). Spring Boot invokes the `#run(ApplicationArguments)` method when the application has started, passing wrappers for the arguments (`String[] args`) into the application. Let's create an `ApplicationRunner` that deletes all the data in the data source, then emits a few book titles, then maps them to `Book` entities, and then persists those books. Finally, it queries all the records in the data source and then prints out everything. **Listing 8** shows all this.

Listing 8: An `ApplicationRunner` to write data

```
package com.example.libraryservice;

import lombok.extern.slf4j.Slf4j;
import org.springframework.boot.ApplicationArguments;
import org.springframework.boot.ApplicationRunner;
```

```
import org.springframework.stereotype.Component;
import reactor.core.publisher.Flux;

@Slf4j
@Component

class SampleBookInitializer implements ApplicationRunner {
    private final BookRepository bookRepository;
    SampleBookInitializer(BookRepository bookRepository) {
         this.bookRepository = bookRepository;
    }

    @Override
    public void run(ApplicationArguments args) throws Exception {
        this.bookRepository
                .deleteAll()
                .thenMany(
                    Flux.just(
                        "Cloud Native Java|jlong",
                       "Spring Security 3.1|rwinch",
                       "Spring in Action|cwalls"))
                .map(t -> t.split("\\|"))
                .map(tuple -> new Book(null, tuple[0], tuple[1]))
                .flatMap(this.bookRepository::save)
                .thenMany(this.bookRepository.findAll())
                .subscribe(book -> log.info(book.toString()));
    }
}
```

The example looks at the titles of various books and one of the (possibly numerous) books' authors, and then it writes them to the database. First the strings are split by the | delimiter.

Then the title and book author are used to create a `Book`. Then the records are saved to the data source, MongoDB. The result of the save operation is a `Mono<Book>`. Something needs to subscribe to each of those resulting `Publisher<T>` instances, so I use the `flatMap` operator. Then, I turn my focus to the results of finding all records and then to logging them for inspection.

This code defines a pipeline; each operator defines a stage in a pipeline. The pipeline is not eager; that is, it won't be executed until it is activated. You activate the pipeline by subscribing to it (the last step in the code in **Listing 8**). `Publisher` defines only one type of subscription, but Reactor provides hooks to process each emitted value, as well as any exceptions thrown, among other things.

Were you to put a breakpoint in any of the lambdas in **Listing 8** and then inspect `Thread.currentThread().getName()`, you'd see that the thread on which processing is running is different than the main thread (which is named `main`). Reactor defers to a `Scheduler` implementation for its processing. You can specify the default global `Scheduler` you'd like to use by calling `Schedulers.setFactory(Factory)`. You can specify on which thread a particular `Publisher` should run when it subscribes by specifying `Mono::subscribeOn(Scheduler)` or `Flux::subscribeOn(Scheduler)`.

Conclusion

You have now used Spring Boot and Spring Initializr to quickly create and run a reactive data application that hews closely to the requirements of reactive development. In the second (and final) part of this article, I'll use Spring Framework 5.0 to stand up a REST API and to implement secure access to this data. Meanwhile, if you want to look at the complete application, the source code is all online. `</article>`

Josh Long (@starbuxman) is a Java Champion and a Spring developer advocate at Pivotal. He is the author of several books on Spring programming, and he speaks frequently at developer conferences.

SEBASTIAN **DASCHNER**

Command Query Responsibility Segregation with Java

Combining event sourcing and event-driven architectures to build scalable, eventually consistent systems

Most of today's enterprise applications are based on a CRUD data model that is simple and straightforward to implement. Event sourcing, event-driven architectures, and Command Query Responsibility Segregation (CQRS) offer another way to model applications that enables interesting solutions and use cases, especially with the rising demands of scalability. Before getting into CQRS, I'll quickly describe some of the limitations of the CRUD model.

Shortcomings of CRUD-Based Applications

A CRUD-based application always contains the current state of the system. The domain entities are stored in the database or in an in-memory representation with their properties as they are at any given moment. That aspect comes in handy when users read the current state, but it falls short in other aspects.

For example, a model that is solely CRUD-based has no information about the history or the context—why the system, including all domain objects, is in its current state and how it got there. Once an update is performed, the objects are then in a new state and their old state is forgotten. This can make it tricky to reproduce and debug situations in production. It's harder to comprehend the current state and find potential bugs if the whole history is not available.

Another challenge of CRUD-based models is that due to storing only the current state, all transactions and interactions need to modify the system in a consistent way. This sounds normal to enterprise developers but can become quite complex when you are dealing with competing transactions—for example, when users want to update their contact information and at the

same time some other use case updates their account balance. If this information affects the same database entries, the two activities lead to a locking situation. Usually, this optimistic locking results in one transaction winning over the other. However, strictly speaking, there should be no need to mutually exclude either transaction.

Because the real world is all about distributed collaboration—often in an asynchronous way—it makes sense to model applications in an event-driven way.

A similar problem occurs when a use case updates business objects whose new states require verification to keep the system in a consistent state. Verifying and maintaining these consistent states can become both redundant and complex.

Because CRUD-based applications need to store the status quo and keep a consistent state within their data model, they cannot scale horizontally. To maintain consistency, such applications need to lock the data (as in good old atomicity, consistency, isolation, and durability [ACID] transactions) until the update has taken place. If several distributed systems are involved, the synchronization will become a bottleneck.

Event Sourcing

In contrast to a CRUD data model, event-sourced systems store all modifications that happen to a system as atomic entities. The application does not necessarily contain the current state, but it can be calculated by applying all events that have happened in the past. These events are the single source of truth in the system.

The most prominent example for this model is bank accounts. You can calculate your current balance by starting at zero and adding or subtracting the amounts of all transactions accordingly. The example in **Figure 1** shows a simple set of customer-related events that can be used to arrive at a customer representation.

The events are atomic and immutable, because they happened in the past and cannot be undone. This implies that, for example, a deletion action also changes the current state by just adding a `CustomerDeleted` event to the log—no entry is actually deleted.

While the current state could be calculated on demand using all events that have happened in the past, enterprise systems use so-called *snapshots* that represent the state as of a certain moment in time. Events that arose after that moment are then applied to the snapshot in order to form a new state, which again can be persisted. This is, however, an optimization technique to deal with a growing number of events—the atomic events remain the golden source of truth.

One of the benefits of this architecture is that the full history of what has happened enables developers to reproduce complex use-case scenarios and debug the system with ease. Another advantage of event-sourced systems is the possibility of calculating statistics and implementing future use cases later. Because all atomic information that ever was applied to the system is available, you can use this information and simply redeploy the application with updated behavior and recalculate the status from the events. That makes it possible to implement future use cases on events that happened in the past—as if that new functionality was always there. For example, answering the question, "How many users signed up on a Tuesday?" is possible using the information contained in the events even if this functionality wasn't considered previously.

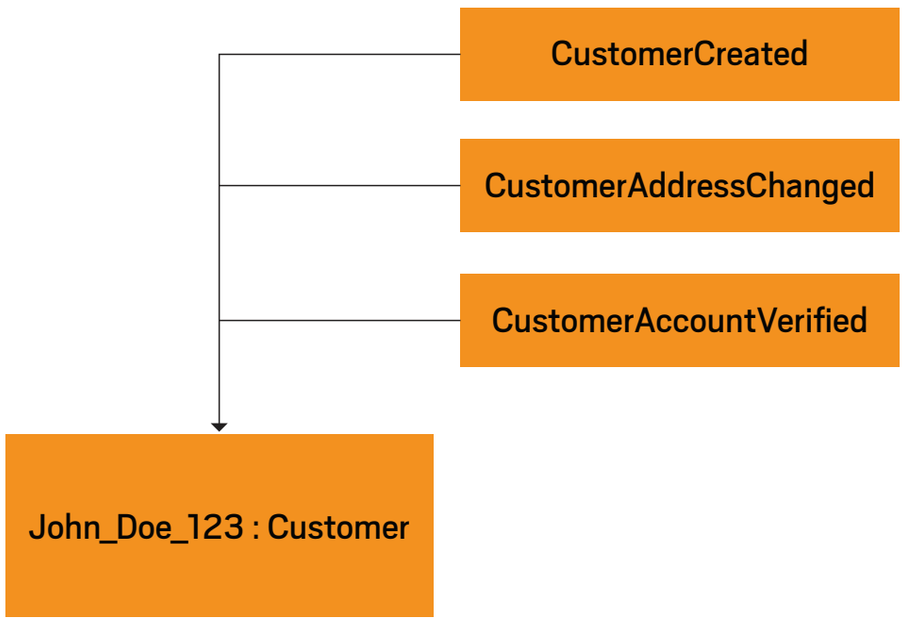


Figure 1. Events that determine the current state of a customer entry

Event sourcing alone doesn't imply that the application has to be implemented using an event-driven or CQRS approach. However, in order to apply CQRS, you need to model applications with event sourcing.

Event-Driven Applications

In contrast to the benefits of an event-sourced system, the motivations behind event-driven applications differ. If you want to model distributed systems—such as microservices—that aim to maintain a consistent state throughout several systems, you need to take transactions into account. Because distributed transactions don't scale well, you split up a transaction into several transactions that still maintain consistency—at least in an eventually consistent way.

An event-driven architecture (see **Figure 2**) realizes use cases that involve multiple systems by collaborating via commands and events. For ordering a cup of coffee at a café, for example, you would first attempt to place an order, which results in an `OrderPlaced` event—or an error.

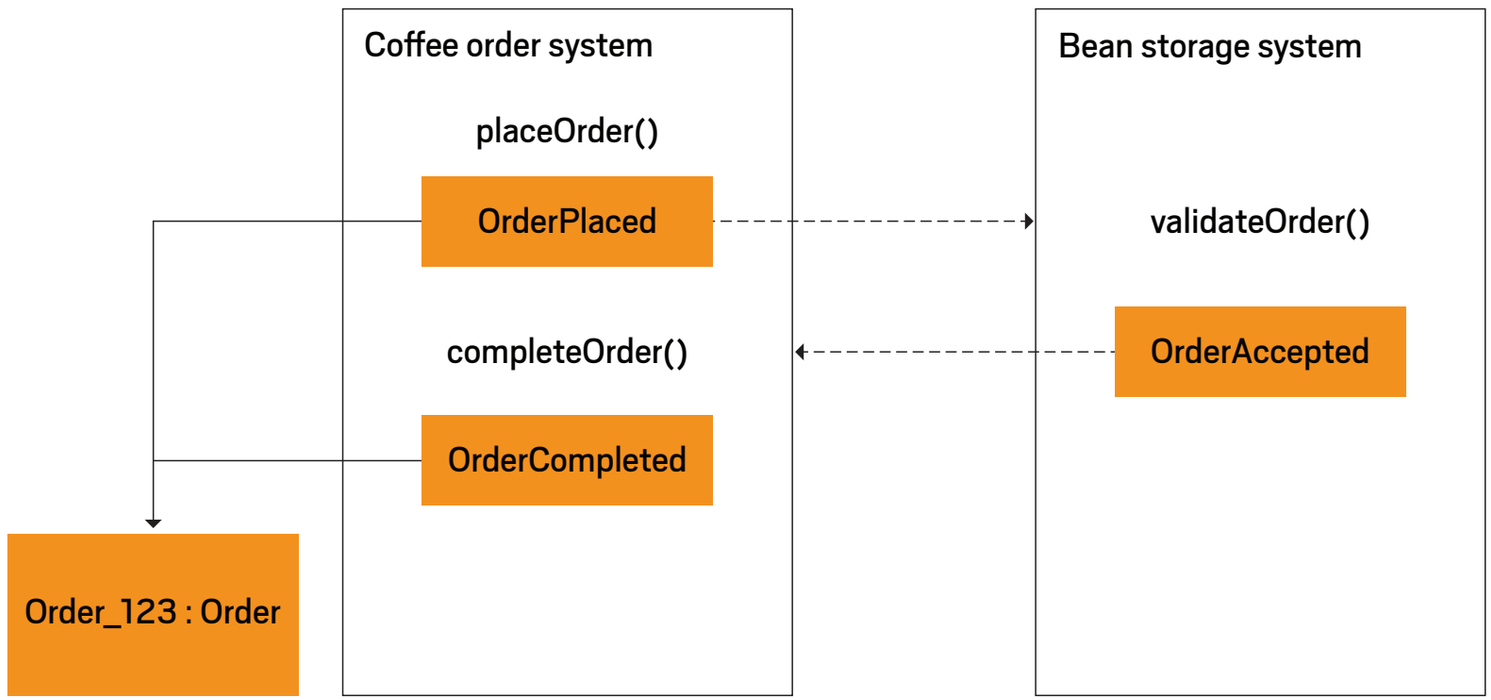


Figure 2. Example event-driven architecture

This `OrderPlaced` event then causes the coffee bean storage to check whether there are beans available and to publish either an `OrderAccepted` event or an `OrderFailedInsufficientBeans` event. The current state of the order is calculated by applying all events related to that order as in an event-sourced system.

This way of modeling causes the process to be eventually consistent, and because the application ensures that all events are published in a reliable way, the final outcome of the use cases will be consistent.

If you compare this way of modeling to the real world, you can see that these methods of collaboration are common. When you order a cup of coffee, the waiter accepts your order—even though it's possible that for some reason the coffee will never make it to you. In that case, the waiter will come back later and apologize for not being able to deliver the coffee and will offer a compensating transaction—even though the order was accepted in the first place. Eventually, you will end up with your coffee, another drink, or your money back.

One of the benefits of separating the responsibilities of reads and writes in the CQRS model is the fact that the query and command sides can scale independently.

Because the real world is all about distributed collaboration—often in an asynchronous way—it makes sense to model applications in an event-driven way.

Enter CQRS

Now that I've summarized implementing event-driven and event-sourced applications, I will introduce the CQRS principle, which prescribes separating the responsibilities of reads and writes. CQRS causes methods to either modify the state of the system without returning any value or to return values without any side effect. The commands (that is, the writes) are not supposed to return values; they either return successfully or throw an error. The queries (that is, the reads) only return data (see **Figure 3**).

This principle is simple in theory but has important implications. Once you split up a system following this approach, the applications collaborate only by events that are published to

an event store. The command and query components maintain their own domain object representations by consuming the events from the hub and updating the state of their internal model. The storage representations of each side can differ and be optimized according to their best fit.

When an update command, `placeOrder(Order)`, reaches the command side, the service performs the action using the domain object representations in its internal storage and publishes events (`OrderPlaced`). When the client reads at the query side, this service returns the current state from its internal storage. The services are coupled only by the event store and can operate and be deployed independently from each other.

The events that are published from the event store are consumed by all subscribed consum-

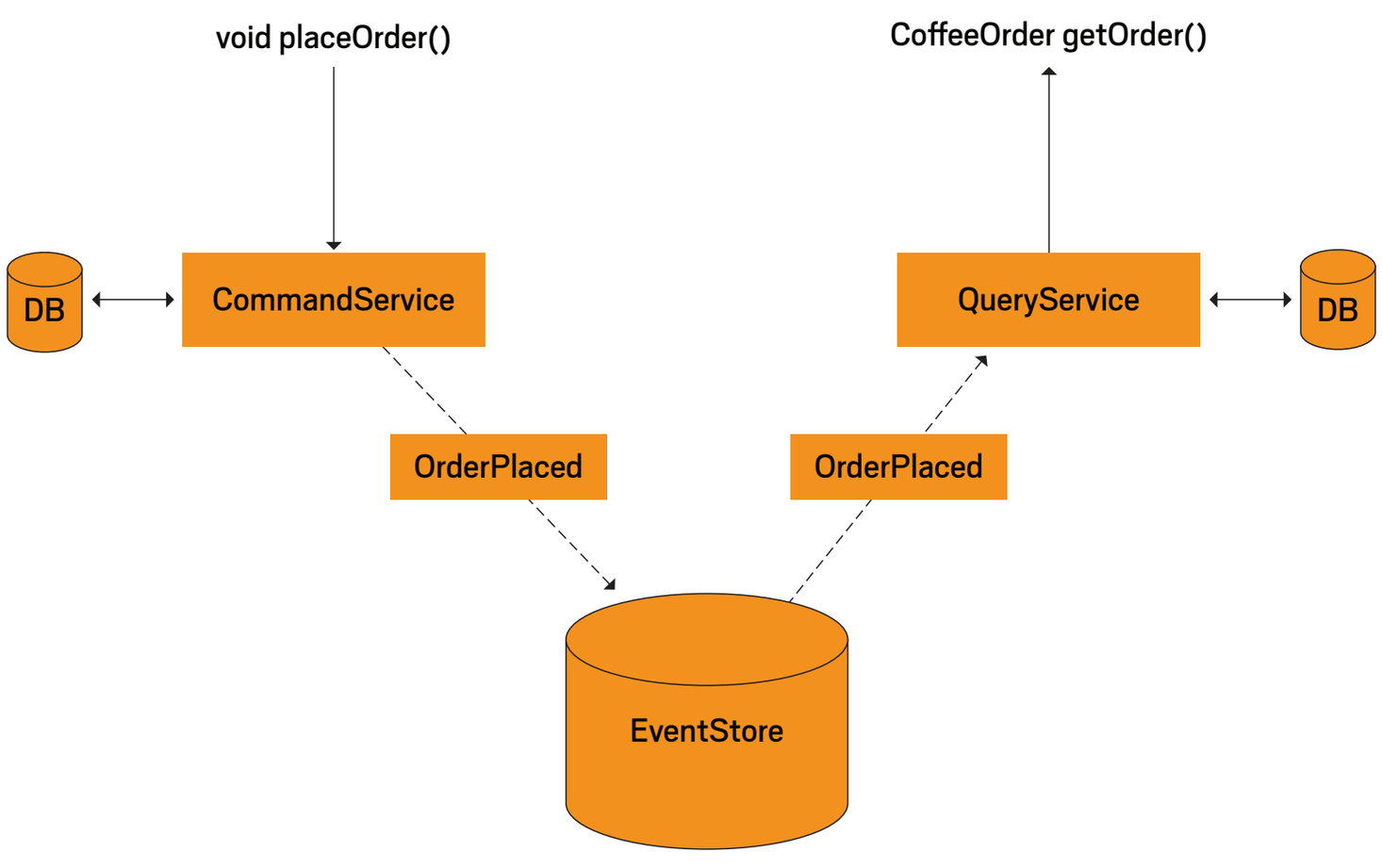


Figure 3. Example of a CQRS implementation

ers to update their internal model—but only one subscriber, `EventHandler`, is supposed to trigger further commands from these events. Publishing the events has to happen in a reliable way to keep the system in a consistent state in the long run.

Benefits of CQRS

One of the benefits of separating the responsibilities of reads and writes in the CQRS model is the fact that the query and command sides can scale independently. In typical enterprise applications, the read operations outnumber the write operations. Because being eventually consistent on the read side is, in most cases, not a big problem, returning replicated data has a positive impact on the overall performance. Using CQRS enables you to deploy, for example, a greater number of query service instances to scale out just the read side.

The domain model representations of each of the services solve the problem of the rising numbers of events in an event-sourced system. Because more and more events are stored in the system over time, the overall performance of operations would decrease if the application state were solely calculated on demand by applying all events each time. Updating the representation continuously and using these models in the commands and queries maintains a constant level of performance. This corresponds to the concept of snapshots.

Another benefit of this separation is the given failover capacity—at least for the read side. Because all instances maintain an eventually consistent representation of the system's state, this cached state is still available if the event store goes down. Even though no new events can be written, the clients can still access the last state.

Applications that implement CQRS also have the capability to implement further use cases that operate on events from the past, because they implement event sourcing as well.

Now, I will show an actual CQRS implementation in a Java EE application.

Example CQRS Application

As an example, I'm using a scalable coffee shop that consists of three services, responsible for order management (`orders`), bean storage (`beans`), and coffee brewing (`barista`). Each service is free to choose its internal domain object representation, and the collaboration is done using

Apache Kafka as the event hub. Once events are published to Kafka, the services handle the events accordingly and update their representation.

The business use cases for ordering a cup of coffee are shown in **Figure 4**.

When a client creates an order, the command service publishes an event (`OrderPlaced`) and returns the request successfully—even though the system can't tell yet whether the order will be finished successfully. The client can request the status of the order from the query service

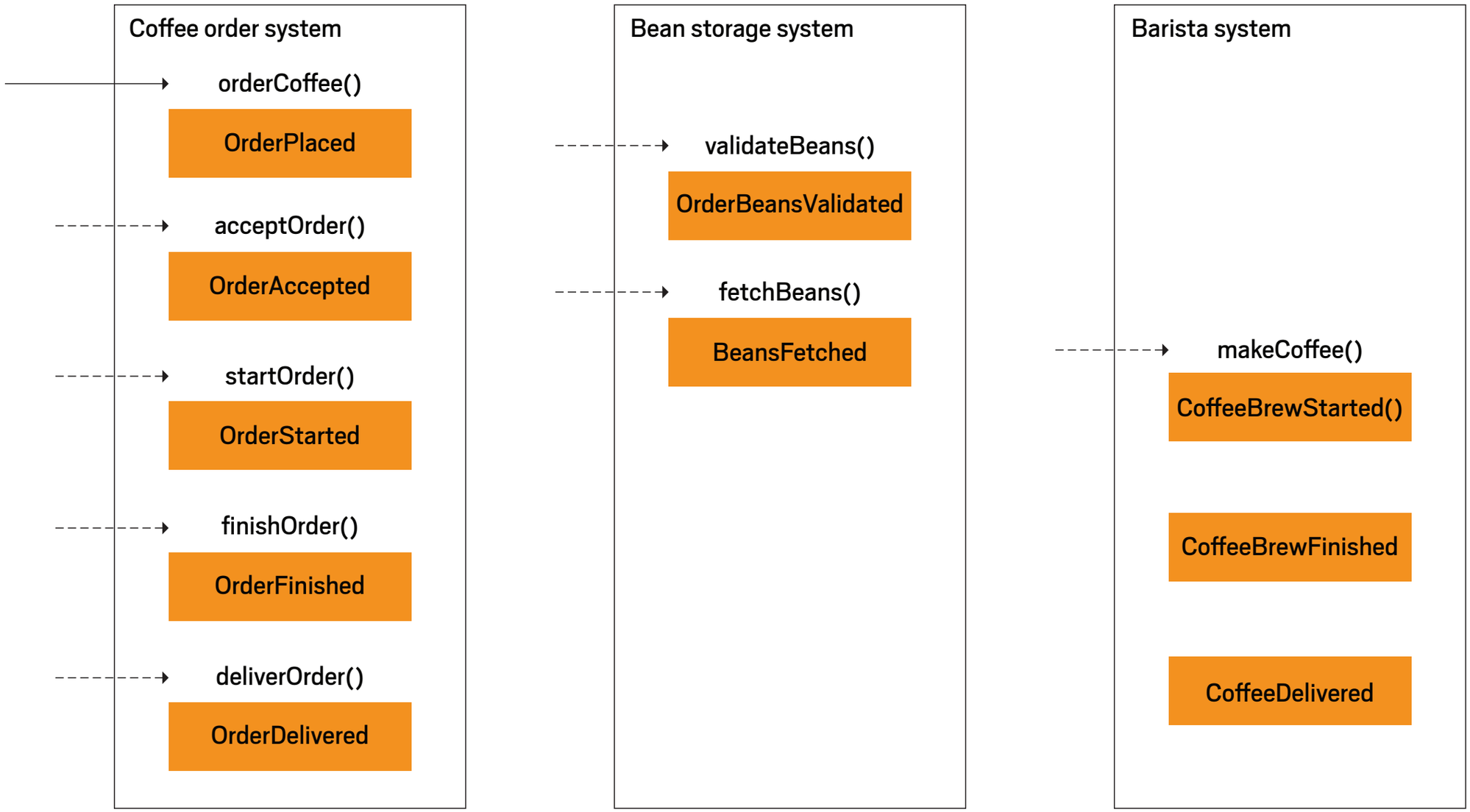


Figure 4. Use cases for ordering a cup of coffee

anytime, with the state being updated on incoming events.

Application Architecture

The Java EE application is organized with the Entity Control Boundary (ECB) pattern. The application boundary contains the external REST interface, a `*CommandService` and `*QueryService`, and the event handling functionality that will call subsequent commands. The control packages contain the storage representations that contain the current domain object representations, as well as functionality to access Kafka. The entity packages consist of the event and domain object definitions.

The command service contains the business methods and publishes events at the event hub. The query service accesses the storage only to return data.

The following code shows examples for the order command service, which processes the commands by publishing the events to the event hub. This service is the use-case entry point from both the application boundary and the event handler.

```
public class OrderCommandService {

    @Inject
    EventProducer eventProducer;

    @Inject
    CoffeeOrders coffeeOrders;

    public void placeOrder(OrderInfo orderInfo) {
        eventProducer.publish(new OrderPlaced(orderInfo));
    }

    void acceptOrder(UUID orderId) {
        OrderInfo orderInfo = coffeeOrders.get(orderId)
            .getOrderInfo();
```

```
        eventProducer.publish(new OrderAccepted(orderInfo));
    }

    void cancelOrder(UUID orderId, String reason) {
        eventProducer.publish(
            new OrderCancelled(orderId, reason));
    }

    void startOrder(UUID orderId) {
        eventProducer.publish(new OrderStarted(orderId));
    }

    void finishOrder(UUID orderId) {
        eventProducer.publish(new OrderFinished(orderId));
    }

    void deliverOrder(UUID orderId) {
        eventProducer.publish(new OrderDelivered(orderId));
    }
}
```

The order query service, shown in the following code, is used to retrieve the coffee order representations. It uses the coffee order store, which keeps track of the orders.

```
public class OrderQueryService {

    @Inject
    CoffeeOrders coffeeOrders;

    public CoffeeOrder getOrder(UUID orderId) {
```

```
        return coffeeOrders.get(orderId);
    }
}
```

Incoming events are delivered as Contexts and Dependency Injection (CDI) events within the application. The store itself observes these CDI events and updates and stores the domain object representations. For simplicity, in the following code, I'm using solely in-memory storage with the Kafka events being redelivered and reapplied at application startup. In a production environment, this functionality would likely be integrated with a persistent database that stores the last calculated state.

```
@Singleton
@Startup
@ConcurrencyManagement(ConcurrencyManagementType.BEAN)
public class CoffeeOrders {

    private final Map<UUID, CoffeeOrder> coffeeOrders =
        new ConcurrentHashMap<>();

    public CoffeeOrder get(UUID orderId) {
        return coffeeOrders.get(orderId);
    }

    public void apply(@Observes OrderPlaced event) {
        coffeeOrders.putIfAbsent(event.getOrderInfo()
            .getOrderId(), new CoffeeOrder());

        applyFor(event.getOrderInfo().getOrderId(),
            o -> o.place(event.getOrderInfo()));
    }
```

```
    public void apply(@Observes OrderCancelled event) {
        applyFor(event.getOrderId(), CoffeeOrder::cancel);
    }

    public void apply(@Observes OrderAccepted event) {
        applyFor(event.getOrderInfo().getOrderId(),
            CoffeeOrder::accept);
    }

    public void apply(@Observes OrderStarted event) {
        applyFor(event.getOrderId(), CoffeeOrder::start);
    }

    public void apply(@Observes OrderFinished event) {
        applyFor(event.getOrderId(), CoffeeOrder::finish);
    }

    public void apply(@Observes OrderDelivered event) {
        applyFor(event.getOrderId(), CoffeeOrder::deliver);
    }

    private void applyFor(UUID orderId,
            Consumer<CoffeeOrder> consumer) {

        CoffeeOrder coffeeOrder = coffeeOrders.get(orderId);
        if (coffeeOrder != null)
            consumer.accept(coffeeOrder);
    }
}
```

For simplicity, both the query and command services are using the same `CoffeeOrders` instance. However, this could be split into several components or systems and further optimized for each side accordingly. For my purpose—to show an example implementation—this model is sufficient.

The connection for incoming events that trigger subsequent commands is done in the event handler. This handler calls the command service for further processing of orders. It both consumes Kafka messages and fires the corresponding CDI events.

```
@Singleton
@Startup
public class OrderEventHandler {

    private EventConsumer eventConsumer;

    @Resource
    ManagedExecutorService mes;

    @Inject
    Properties kafkaProperties;

    @Inject
    Event<CoffeeEvent> events;

    @Inject
    OrderCommandService orderService;

    @Inject
    Logger logger;

    public void handle(@Observes OrderBeansValidated event) {
        orderService.acceptOrder(event.getOrderId());
```

```
}

public void handle(@Observes
        OrderFailedBeansNotAvailable event) {

    orderService.cancelOrder(event.getOrderId(),
        "No beans of the origin were available");
}

public void handle(@Observes CoffeeBrewStarted event) {
    orderService.startOrder(event.getOrderInfo().getOrderId());
}

public void handle(@Observes CoffeeBrewFinished event) {
    orderService.finishOrder(event.getOrderId());
}

public void handle(@Observes CoffeeDelivered event) {
    orderService.deliverOrder(event.getOrderId());
}

@PostConstruct
private void initConsumer() {
    kafkaProperties.put("group.id", "order-handler");

    eventConsumer = new EventConsumer(kafkaProperties, ev -> {
        logger.info("firing = " + ev);
        events.fire(ev);
    }, "barista", "beans");

    mes.execute(eventConsumer);
```

```
    }

    @PreDestroy
    public void closeConsumer() {
        eventConsumer.stop();
    }
}
```

Integrating Apache Kafka

Apache Kafka serves as a reliable, persistent, and scalable event hub that delivers events to the services involved. I make use of event topics that are consumed in so-called consumer groups. In this case, I configure Kafka to deliver the events reliably once in every consumer group. By configuring the same group for all event handlers, I ensure that only one instance processes an event.

The event producer, shown in the following code, publishes the events to Kafka:

```
@ApplicationScoped
public class EventProducer {

    private Producer<String, CoffeeEvent> producer;

    @Inject
    Properties kafkaProperties;

    @Inject
    Logger logger;

    @PostConstruct
    private void init() {
```

```
        kafkaProperties.put("transactional.id",
            UUID.randomUUID().toString());
        producer = new KafkaProducer<>(kafkaProperties);
        producer.initTransactions();
    }

    public void publish(CoffeeEvent event) {
        ProducerRecord<String, CoffeeEvent> record =
            new ProducerRecord<>("order", event);
        try {
            producer.beginTransaction();
            logger.info("publishing = " + record);
            producer.send(record);
            producer.commitTransaction();
        } catch (ProducerFencedException e) {
            producer.close();
        } catch (KafkaException e) {
            producer.abortTransaction();
        }
    }

    @PreDestroy
    public void close() {
        producer.close();
    }
}
```

The following code uses transactional producers that were introduced in Kafka version 0.11. They ensure that an event has been sent reliably before the client call returns. The event consumer infinitely consumes new Kafka events and passes them to a functional `Consumer`.

```
public class EventConsumer implements Runnable {

    private KafkaConsumer<String, CoffeeEvent> consumer;
    private final Consumer<CoffeeEvent> eventConsumer;
    private final AtomicBoolean closed = new AtomicBoolean();

    public EventConsumer(Properties kafkaProperties,
            Consumer<CoffeeEvent> eventConsumer,
            String... topics) {
        this.eventConsumer = eventConsumer;
        consumer = new KafkaConsumer<>(kafkaProperties);
        consumer.subscribe(asList(topics));
    }

    @Override
    public void run() {
        try {
            while (!closed.get()) {
                consume();
            }
        } catch (WakeupException e) {
            // will wake up for closing
        } finally {
            consumer.close();
        }
    }

    private void consume() {
        ConsumerRecords<String, CoffeeEvent> records =
            consumer.poll(Long.MAX_VALUE);
```

```
            for (ConsumerRecord<String, CoffeeEvent> record : records) {
                eventConsumer.accept(record.value());
            }
            consumer.commitSync();
        }

        public void stop() {
            closed.set(true);
            consumer.wakeup();
        }
    }
```

After an event has been processed, I commit to the consumption by calling `commitSync`. This event consumer is started from both the event handler and the updating consumer. Both are then responsible for firing the CDI events. See the `OrderEventHandler` definition shown earlier and the following `OrderUpdateConsumer`:

```
@Startup
@Singleton
public class OrderUpdateConsumer {

    private EventConsumer eventConsumer;

    @Resource
    ManagedExecutorService mes;

    @Inject
    Properties kafkaProperties;

    @Inject
```

```
    Event<CoffeeEvent> events;

    @Inject
    Logger logger;

    @PostConstruct
    private void init() {
        kafkaProperties.put("group.id", "order-consumer-" +
            UUID.randomUUID());

        eventConsumer = new EventConsumer(kafkaProperties, ev -> {
            logger.info("firing = " + ev);
            events.fire(ev);
        }, "order");

        mes.execute(eventConsumer);
    }

    @PreDestroy
    public void close() {
        eventConsumer.stop();
    }
}
```

To ensure that the consumers are managed correctly, I use Java EE's managed executor service to run the consumers in threads managed by the application server. For the updating consumers, unique group IDs are generated to ensure that every service gets all events.

When these services start, they connect to their corresponding Kafka topics and ask for all the undelivered events in their consumer group. To update the domain object representations to the latest state, the updating consumer group that has the matching ID applies the events—for

example, in the `CoffeeOrders`—that occurred since the very beginning. As I mentioned before, I'm using only in-memory storage without persistent snapshots. For the full example application, see the scalable-coffee-shop project on GitHub.

Conclusion

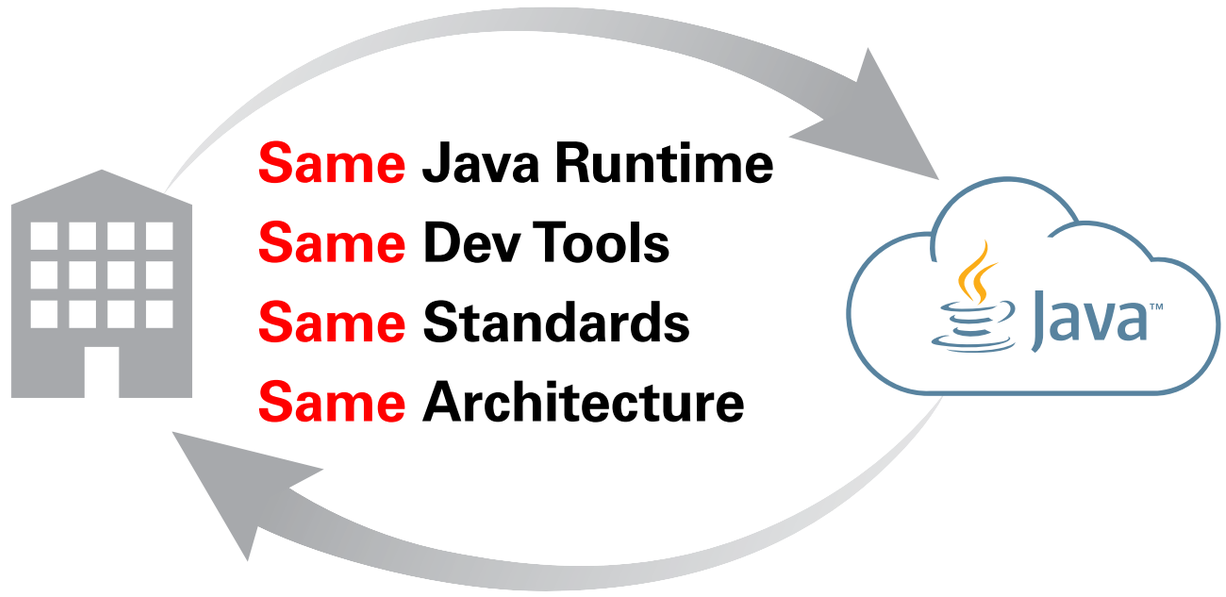
CQRS provides a useful alternative to the traditional CRUD-based way of building enterprise applications by combining the benefits of event sourcing and event-driven architectures to build scalable, eventually consistent systems.

Of course, this approach is no silver bullet. If the situation does not require the scalability of event-driven architectures, it's advisable to go with monolithic, consistent applications instead. CQRS introduces some overhead, which certainly is avoidable in most enterprise applications. An application that solely requires the benefits of event sourcing can be based on this approach while still using a relational database and consistent use cases. `</article>`

Sebastian Daschner (@DaschnerS) is a Java Champion who works as a consultant and trainer. He participates in the Java Community Process (JCP), serving in the JSR 370 and JSR 374 Expert Groups. Daschner is also a heavy user of Linux and container technologies such as Docker. When not working with Java, he loves to travel.

Push a Button

Move Your Java Apps to the Oracle Cloud



…or Back to Your Data Center

ORACLE®

MICHAEL KÖLLING

The Evolving Nature of Java Interfaces

Understanding multiple inheritance in Java

In the occasional "New to Java" series, I try to pick topics that invite a deeper understanding of the conceptual background of a language construct. Often, novice programmers have a working knowledge of a concept—that is, they can use it in many situations, but they lack a deeper understanding of the underlying principles that would lead to writing better code, creating better structures, and making better decisions about when to use a given construct. Java *interfaces* are often just such a topic.

In this article, I assume that you have a basic understanding of inheritance. Java interfaces are closely related to inheritance, as are the `extends` and `implements` keywords. So, I will discuss why Java has two different inheritance mechanisms (indicated by these keywords), how abstract classes fit in, and what various tasks interfaces can be used for.

As is so often the case, the story of these features starts with some quite simple and elegant ideas that lead to the definition of concepts in early Java versions, and the story gets more complicated as Java advances to tackle more-intricate, real-world problems. This challenge led to the introduction of default methods in Java 8, which muddied the waters a bit.

A Little Background on Inheritance

Inheritance is straightforward to understand in principle: a class can be specified as an extension of another class. In such a case, the present class is called a *subclass*, and the class it's extending is called the *superclass.* Objects of the subclass have all the properties of both the superclass and the subclass. They have all fields defined in either subclass or superclass and also all methods from both. So far, so good.

PHOTOGRAPH BY JOHN BLYTHE

Inheritance is, however, the equivalent of the Swiss Army knife in programming: it can be used to achieve some very diverse goals. I can use inheritance to reuse some code I have written before, I can use it for subtyping and dynamic dispatch, I can use it to separate specification from implementation, I can use it to specify a contract between different parts of a system, and I can use it for a variety of other tasks. These are all important, but very different, ideas. It is necessary to understand these differences to get a good feel for inheritance and interfaces.

Type Inheritance Versus Code Inheritance

Two main capabilities that inheritance provides are the ability to inherit code and the ability to inherit a type. It is useful to separate these two ideas conceptually, especially because standard Java inheritance mixes them together. In Java, every class I define also defines a type: as soon as I have a class, I can create variables of that type, for example.

When I create a subclass (using the `extends` keyword), the subclass inherits both the code and the type of the superclass. Inherited methods are available to be called (I'll refer to this as "the code"), and objects of the subclass can be used in places where objects of the superclass are expected (thus, the subclass creates a subtype).

Let's look at an example. If `Student` is a subclass of `Person`, then objects of class `Student` have the type `Student`, but they also have the type `Person`. A student is a person. Both the code and the type are inherited.

The decision to link type inheritance and code inheritance in Java is a language design choice: it was done because it is often useful, but it is not the only way a language can be designed. Other programming languages allow inheriting the code without inheriting the type (such as C++ *private inheritance*) or inheriting the type without inheriting the code (which Java also supports, as I explain shortly).

Multiple Inheritance

The next idea entering the mix is *multiple inheritance*: a class may have more than one superclass. Let me give you an example: PhD students at my university also work as instructors. In that sense, they are like faculty (they are instructors for a class, have a room number, a payroll

number, and so on). But they are also students: they are enrolled in a course, have a student ID number, and so on. I can model this as multiple inheritance (see **Figure 1**).

`PhDStudent` is a subclass of both `Faculty` and `Student`. This way, a PhD student will have the attributes of both students and faculty. Conceptually this is straightforward. In practice, however, the language becomes more complicated if it allows multiple inheritance, because that introduces new problems: What if both superclasses have fields with the same name? What if they have methods with the same signature but different implementations? For these cases, I need language constructs that specify some solution to the problem of ambiguity and name overloading. However, it gets worse.

Diamond Inheritance

A more complicated scenario is known as *diamond inheritance* (see **Figure 2**). This is where a class (`PhDStudent`) has two superclasses (`Faculty` and `Student`), which in turn have a common superclass (`Person`). The inheritance graph forms a diamond shape.

Now, consider this question: If there is a field in the top-level superclass (`Person`, in this case), should the class at the bottom (`PhDStudent`) have one copy of this field or two? It inherits

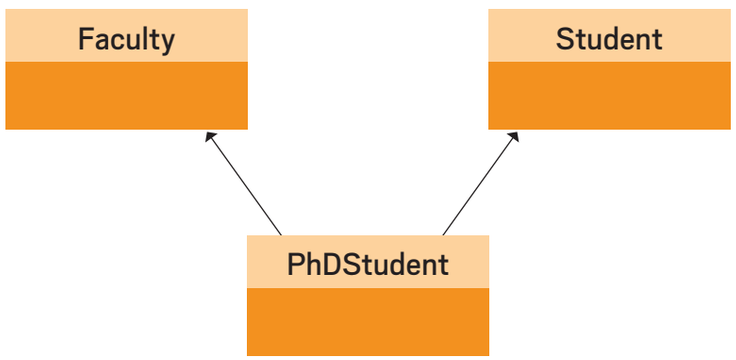


Figure 1. An example of multiple inheritance

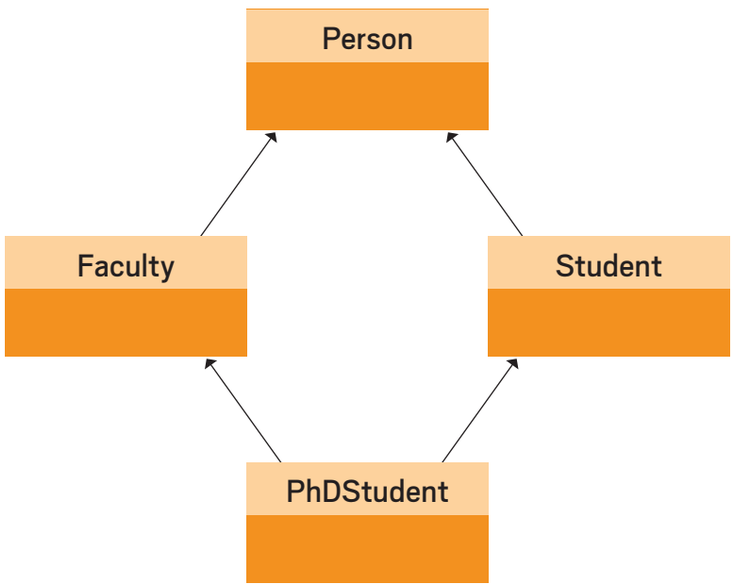


Figure 2. An example of diamond inheritance

this field twice, after all, once via each of its inheritance branches.

The answer is: it depends. If the field in question is, say, an ID number, maybe a PhD student should have two: a student ID and a faculty/payroll ID that might be a different number. If the field is, however, the person's family name, then you want only one (the PhD student has only one family name, even though it is inherited from both superclasses).

The Java designers arrived at a pragmatic solution: allow only single inheritance for code, but allow multiple inheritance for types.

In short, things can become very messy. Languages that allow full multiple inheritance need to have rules and constructs to deal with all these situations, and these rules are complicated.

Type Inheritance to the Rescue

When you think about these problems carefully, you realize that all the problems with multiple inheritance are related to inheriting code: method implementations and fields. Multiple code inheritance is messy, but multiple type inheritance causes no problems. This fact is coupled with another observation: multiple code inheritance is not terribly important, because you can use *delegation* (using a reference to another object) instead, but multiple subtyping is often very useful and not easily replaced in an elegant way.

That is why the Java designers arrived at a pragmatic solution: allow only single inheritance for code, but allow multiple inheritance for types.

Interfaces

To make it possible to have different rules for types and code, Java needs to be able to specify types without specifying code. That is what a Java interface does.

Interfaces specify a Java type (the type name and the signatures of its methods) without specifying any implementation. No fields and no method bodies are specified. Interfaces can contain constants. You can leave out the modifiers (`public static final` for constants and `public` for methods)—they are implicitly assumed.

This arrangement provides me with two types of inheritance in Java: I can inherit a class (using `extends`), in which I inherit both the type and the code, or I can inherit a type only (using `implements`) by inheriting from an interface. And I can now have different rules concerning multiple inheritance: Java permits multiple inheritance for types (interfaces) but only single inheritance for classes (which contain code).

Benefits of Multiple Inheritance for Types

The benefits of allowing the inheritance of multiple types—essentially of being able to declare that one object can be viewed as having a different type at different times—are quite easy to see. Suppose you are writing a traffic simulation, and in it you have objects of class `Car`. Apart from cars, there are other kinds of active objects in your simulation, such as pedestrians, trucks, traffic lights, and so on. You may then have a central collection in your program—say, a `List`—that holds all the actors:

```
private List<Actor> actors;
```

`Actor`, in this case, could be an interface with an `act` method:

```
public interface Actor
{
    void act();
}
```

Your `Car` class can then implement this interface:

```
class Car implements Actor
{
    public void act()
    {
        ...
```

```
    }
}
```

Note that, because `Car` inherits only the type, including the signature of the `act` method, but no code, it must itself supply the code to implement the type (the implementation of the `act` method) before you can create objects from it.

So far, this is just single inheritance and could have been achieved by inheriting a class. But imagine now that there is also a list of all objects to be drawn on screen (which is not the same as the list of actors, because some actors are not drawn, and some drawn objects are not actors):

```
private List<Drawable> drawables;
```

You might also want to save a simulation to permanent storage at some point, and the objects to be saved might, again, be a different list. To be saved, they need to be of type `Serializable`:

```
private List<Serializable> objectsToSave;
```

In this case, if the `Car` objects are part of all three lists (they act, they are drawn, and they should be saved), the class `Car` can be defined to implement all three interfaces:

```
class Car implements Actor, Drawable, Serializable ...
```

Situations like this are common, and allowing multiple supertypes enables you to view a single object (the car, in this case) from different perspectives, focusing on different aspects to group them with other similar objects or to treat them according to a certain subset of their possible behaviors.

Java's GUI event-processing model is built around the same idea: event handling is achieved via event listeners—interfaces (such as `ActionListener`) that often just implement a single method—so that objects that implement it can be viewed as being of a listener type when necessary.

Abstract Classes

I should say a few words about *abstract classes*, because it is common to wonder how they relate to interfaces. Abstract classes sit halfway between classes and interfaces: they define a type and can contain code (as classes do), but they can also have *abstract methods*—methods that are specified only, but not implemented. You can think of them as partially implemented classes with some gaps in them (code that is missing and needs to be filled in by subclasses).

In my example above, the `Actor` interface could be an abstract class instead. The `act` method itself might be abstract (because it is different in each specific actor and there is no reasonable default), but maybe it contains some other code that is common to all actors.

In this case, I can write `Actor` as an abstract class, and the inheritance declaration of my `Car` class would look like this:

```
class Car extends Actor implements Drawable, Serializable ...
```

If I want several of my interfaces to contain code, turning them all into abstract classes does not work. As I stated before, Java allows only single inheritance for classes (that means only one class can be listed after the `extends` keyword). Multiple inheritance is for interfaces only.

There is a way out, though: *default methods*, which were introduced in Java 8. I'll get to them shortly.

Empty Interfaces

Sometimes you come across interfaces that are empty—they define only the interface name and no methods. `Serializable`, mentioned previously, is such an interface. `Cloneable` is another. These interfaces are known as *marker interfaces*. They mark certain classes as possessing a specific property, and their purpose is more closely related to providing metadata than to implementing a type or defining a contract between parts of a program. Java, since version 5, has had annotations, which are a better way of providing metadata. There is little reason today to use marker interfaces in Java. If you are tempted, look instead at using annotations.

A New Dawn with Java 8

So far, I have purposely ignored some new features that were introduced with Java 8. This is because Java 8 adds functionality that contradicts some of the earlier design decisions of the language (such as "only single inheritance for code"), which makes explaining the relationship of some constructs quite difficult. Arguing the difference between and justification for the existence of interfaces and abstract classes, for instance, becomes quite tricky. As I will show in a moment, interfaces in Java 8 have been extended so that they become more similar to abstract classes, but with some subtle differences.

In my explanation of the issues, I have taken you down the historical path—explaining the pre-Java 8 situation first and now adding the newer Java 8 features. I did this on purpose, because understanding the justification for the combination of features as they are today is possible only in light of this history.

If the Java team were to design Java from scratch now, and if breaking backward compatibility were not a problem, they would not design it in the same way. The Java language is, however, not foremost a theoretical exercise, but a system for practical use. And in the real world, you must find ways to evolve and extend your language without breaking everything that has been done before. Default methods and static methods in interfaces are two mechanisms that made progress possible in Java 8.

Evolving Interfaces

One problem in developing Java 8 was how to evolve interfaces. Java 8 added lambdas and several other features to the Java language that made it desirable to adapt some of the existing interfaces in the Java library. But how do you evolve an interface without breaking all the existing code that uses this interface?

Imagine you have an interface `MagicWand` in your existing library:

```
public interface MagicWand
{
```

```
    void doMagic();
}
```

This interface has already been used and implemented by many classes in many projects. But you now come up with some really great new functionality, and you would like to add a really useful new method:

```
public interface MagicWand
{
    void doMagic();
    void doAdvancedMagic();
}
```

If you do that, then all classes that previously implemented this interface break, because they are required to provide an implementation for this new method. So, at first glance, it seems you are stuck: either you break existing user code (which you don't want to do) or you're doomed to stick with your old libraries without a chance to improve them easily. (In reality, there are some other approaches that you could try, such as extending interfaces in subinterfaces, but these have their own problems, which I do not discuss here.) Java 8 came up with a clever trick to get the best of both worlds: the ability to add to existing interfaces without breaking existing code. This is done using *default methods* and *static methods*, which I discuss now.

Default Methods

Default methods are methods in interfaces that have a method body—the default implementation. They are defined by using the `default` modifier at the beginning of the method signature, and they have a full method body:

```
public interface MagicWand
{
```

```
    void doMagic();
    default void doAdvancedMagic()
    {
        ... // some code here
    }
}
```

Classes that implement this interface now have the chance to provide their own implementation for this method (by overriding it), or they can completely ignore this method, in which case they receive the default implementation from the interface. Old code continues to work, while new code can use this new functionality.

Static Methods

Interfaces can now also contain static methods with implementations. These are defined by using the usual `static` modifier at the beginning of the method signature. As always, when writing interfaces, the `public` modifier may be left out, because all methods and all constants in interfaces are always public.

So, What About the Diamond Problem?

As you can see, abstract classes and interfaces have become quite similar now. Both can contain abstract methods and methods with implementations, although the syntax is different. There still are some differences (for instance, abstract classes can have instance fields, whereas interfaces cannot), but these differences support a central point: since the release of Java 8, you have multiple inheritance (via interfaces) that can contain code!

At the beginning of this article I pointed out how the Java designers treaded very carefully to avoid multiple code inheritance because of possible problems, mostly related to inheriting multiple times and to name clashes. So what is the situation now?

As usual, the Java designers devised the following sensible and practical rules to deal with these problems:

- Inheriting multiple abstract methods with the same name is not a problem—they are viewed as the same method.
- Diamond inheritance of fields—one of the difficult problems—is avoided, because interfaces are not allowed to contain fields that are not constants.
- Inheriting static methods and constants (which are also static by definition) is not a problem, because they are prefixed by the interface name when they are used, so their names do not clash.
- Inheriting from different interfaces multiple default methods with the same signature and different implementations is a problem. But here Java chooses a much more pragmatic solution than some other languages: instead of defining a new language construct to deal with this, the compiler just reports an error. In other words, it's your problem. Java just tells you, "Don't do this."

Conclusion

Interfaces are a powerful feature in Java. They are useful in many situations, including for defining contracts between different parts of the program, defining types for dynamic dispatch, separating the definition of a type from its implementation, and allowing for multiple inheritance in Java. They are very often useful in your code; you should make sure you understand their behavior well.

The new interface features in Java 8, such as default methods, are most useful when you write libraries; they are less likely to be used in application code. However, the Java libraries now make extensive use of them, so make sure you know what they do. Careful use of interfaces can significantly improve the quality of your code. </article>

[An earlier version of this article ran in the September/October 2016 issue of *Java Magazine*. —*Ed.*]

Michael Kölling is a Java Champion and a professor at the University of Kent, England. He has published two Java textbooks and numerous papers on object orientation and computing education topics, and he is the lead developer of BlueJ and Greenfoot, two educational programming environments. Kölling is also a Distinguished Educator of the ACM.

SIMON ROBERTS

MIKALAI ZAIKIN

Quiz Yourself

More intermediate and advanced test questions

If you're a regular reader of this quiz, you know that these questions simulate the level of difficulty of two different certification tests. Those marked "intermediate" correspond to questions from the Oracle Certified Associate exam, which contains questions for a preliminary level of certification. Questions marked "advanced" come 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.

These questions rely on Java 8. I'll begin covering Java 9 in future columns, of course, and I will make that transition quite clear when it occurs.

I'd also like to welcome Mikalai Zaikin to this column as a coauthor. He's been working on these questions with me for some time now, so you've already been seeing the benefit of his work.

Question 1 (advanced). Given this code:

```
public class OneValue {
    private final int x;
}
```

Consider these possible changes:

Change 1. Change the declaration of `x` as follows:

```
    private final int x = 99;
```

Change 2. Add to the class as follows:

```
    public OneValue() {
```

```
        x = 100;
    }
```

Change 3. Add to the class as follows:

```
    private void setX(int x) {
        this.x = x;
    }
    public OneValue() {
        setX(100);
    }
```

Which are true? Choose two.

A. The code compiles as it is.

B. The code compiles if change 1 is done.

C. The code compiles if change 2 is done.

D. The code compiles if change 3 is done.

E. The code compiles if change 1 and change 2 are both done.

Question 2 (advanced). Which of the following classes produce immutable objects? Choose two.

A.

```
public class Immut1 {
   final int[] data = { 1, 1, 2, 3, 5, 8, 13 };
   String name;
   public Immut1(String n) {  this.name = n; }
}
```

B.

```
public class Immut2 {
   final int[] data = { 1, 1, 2, 3, 5, 8, 13 };
   final String name;
```

//fix this /

```
   public Immut2(String n) { this.name = n; }
}
```

C.

```
public class Immut3 {
   private int x;
   public Immut3(int x) { this.x = x; }
}
```

D.

```
public class Immut4 {
   private List<String> ls;
   public Immut4() {
      ls = Arrays.asList("Fred", "Jim", "Sheila");
   }
   public String get(int idx) {
       return ls.get(idx);
   }
}
```

E.

```
public class Immut5 {
    private List<String> ls;
    public Immut5(String... strings) {
        ls = Collections.unmodifiableList(Arrays.asList(strings));
    }
    public String get(int idx) {
        return ls.get(idx);
    }
}
```

Question 3 (intermediate). Given this method:

```
public final void doStuff(int val) /* point A */ {
    if (val < 0) throw new NullPointerException();
    if (val < 1) throw new IOException();
    if (val < 2) throw new OutOfMemoryError();
}
```

Which of the following is best?

A. Insert the following at point A:
```
throws Exception
```

B. Insert the following at point A:
```
throws NullPointerException, OutOfMemoryError
```

C. Insert the following at point A:
```
throws IOException, OutOfMemoryError, SQLException
```

D. Insert the following at point A:
```
throws IOException
```

E. Insert the following at point A:
```
throws NullPointerException
```

Question 4 (intermediate). Given this:

```
String s = "Hello";
StringBuilder sb = new StringBuilder("Hello");
StringBuilder sb2 = new StringBuilder("Hello");
// line n1
```

Which is true?

A. Placed at line n1, the following fragment:
```
System.out.println(sb + sb);
```

prints this:
```
HelloHello
```

B. Placed at line n1, the following fragment:
```
System.out.println(sb.equals(sb2));
```

prints this:
```
true
```

C. Placed at line n1, the following fragment compiles successfully:
```
String val = sb.equals(s)?sb:"Differ";
System.out.println(val);
```

D. Placed at line n1, the following fragment:
```
CharSequence val = sb.equals(s)?sb:"Differ";
System.out.println(val);
```

prints this:
```
Hello
```

E. Placed at line n1, the following fragment:
```
System.out.println(sb.equals(s)?sb:"Differ");
```

prints this:
```
Differ
```

Answer 1. The correct answers are options B and C. *Java Language Specification* section 8.3.1.2 says this about final fields: "A blank final instance variable must be definitely assigned at the end of every constructor of the class in which it is declared, or a compile-time error occurs."

This means that the final field `x` must receive exactly one explicit assignment, which must happen before the constructor is complete. This tells you immediately that option A must be incorrect, because in the original code presented in the question there is no assignment to the field. Note that the field as declared is termed a "blank final" (the terminology used in the Java specification paragraph above) and as such, the default assignment to zero that is implicit for all object members does not satisfy the requirement.

Change 1 assigns a value to `x` as part of its declaration and, therefore, `x` is definitely assigned even before any constructor runs. Therefore, option B is correct.

Change 2 adds a simple constructor that initializes the value of `x`. This change, made in isolation, would result in exactly one constructor and causes that constructor to unconditionally assign a value to `x`. Because the blank final is definitely assigned, exactly once, before the end of the only constructor, this change works, and option C is correct.

Change 3 suggests adding a constructor that might seem functionally equivalent to the one proposed in change 2. However, in this case, the change fails. The private method that attempts to assign the value to `x` will not compile, because it's possible for it to be invoked after the object has been initialized. Because this fails to compile, option D must be incorrect.

Performing both change 1 and change 2 also fails, because this would result in an attempt to perform two assignments to the variable `x`, and the Java specification demands exactly one assignment. Therefore, option E is also incorrect.

Answer 2. The correct answers are options C and D. An object is immutable if no syntactically permissible interaction with it by external code can change its state after construction, and no code within the class ever makes any such change either. Literally, once it is created, the value remains the same.

Now, before analyzing this question, be aware that if you decide to create classes that yield immutable objects (which is a design style that can reap significant rewards in terms of correctness, particularly in concurrent systems), you should do a better job than the examples shown here. In particular, although you'll see that the `final` keyword is *not* sufficient to render everything it touches unalterable, it should almost certainly be used anyway. In particular, it has some value in concurrency that is not part of this discussion. Also keep in mind that it's possible to break many forms of immutability through reflection, which might have unexpected consequences.

This question investigates the rules and purpose of Java's exception mechanism, and it also dares to stray into that troublesome territory of asking what's "best" rather than merely what's "correct."

In option A, the fields have default accessibility, rather than being private, so it's a simple matter for any other class that has access (that is, any other class in the same package) to mutate the value of the `String name` to point to a different string. Therefore, option A is incorrect.

In option B, the `String name` field has been marked final, so even though it's accessible to other members of the package, it cannot be mutated; it must refer to the string object that's passed into the constructor. Strings themselves are immutable, so that field's value can never be changed. However, the contents of `final int [] data` can, in fact, be changed (and actually, it could be changed in option A, too, although you already know option A is incorrect based on the string field). This is because the `final` keyword prevents the value of `data`—which is a pointer—from being modified. So, `data` can never refer to any array other than the one with which it's initialized. Of course, the length of arrays can never change after they are created, but their contents *can* be changed. Therefore, the values in the `data` array are actually mutable by any code in other classes in the same package. Hence, option B is incorrect.

In option C, there is a single field: `int x`. The field is private but not final. The value of the field is initialized with a *copy* of the value passed to the constructor. (All arguments in non-remote Java method invocations are passed by value, and with primitive types, the "value" really is the value being represented, not the "value of the reference.") Because of this, changes to the original variable that was passed as an argument to the constructor do not affect the value of `x`. Also, no code in the class ever changes the value of `x` after the object is constructed. So, even though the field is not marked final, instances of this class are immutable, and option C is correct.

In option D, you again see a private, nonfinal field. This time, it's `List<String> ls`. Because it's private, and nothing outside the class ever has a copy of the reference value in `ls`, nothing will ever change the contents of the list or point the variable at a different list. Therefore, option D is correct.

It's prohibited for an overriding method to declare checked exceptions that were not already permitted in the context of the overridden method.

Option E is a little more subtle. You have a variable, `ls`, which is identical to the one described in option D. Therefore, you know that nothing changes the value of `ls` to make it refer to a different list object. If you can be sure that the list that `ls` refers to cannot be altered in any way, you would know the object is immutable.

The variable `ls` is initialized to refer to a list created by the `Arrays.asList` method, which is a utility that describes itself as creating a "structurally immutable list"—which sounds promising; the list will not allow the addition or removal of elements. However, the list created by `Arrays.asList` actually *honors* attempts to reassign any given element of the list. But to counter that, this list is wrapped in `Collections.unmodifiableList`, which puts a proxy wrapper around the object, so that any attempt to modify the list will throw an exception. Surely this must be immutable, right? Well, it turns out that the list that's created uses the *provided* array as its backing storage. Therefore, if the caller of the constructor chooses to provide an explicit `String []` as its argument, any changes made to that array will be reflected in the list. Because of this, the objects are not reliably immutable and option E is incorrect.

If you want to examine this effect, try running this code:

```
String [] names = {"Tony", "Jane"};
Immut5 i5 = new Immut5(names);
System.out.println("i5.get(0) " + i5.get(0));
names[0] = "Anthony";
System.out.println("i5.get(0) " + i5.get(0));
```

Answer 3. The correct answer is option D. This question investigates the rules and purpose of Java's exception mechanism, and it also dares to stray into that troublesome territory of asking what's "best" rather than merely what's "correct." However, we hope to make a good case for that value judgment, and while we are happy to include this question because it creates a useful discussion—both about Java's exception mechanism and about how to evaluate a judgment like this—we doubt that this question would survive unchanged in the real exam.

The first point is that Java distinguishes checked exceptions from unchecked exceptions and errors. In particular, a method that might throw a checked exception must announce this in that method's signature. In this question, the appropriate point for the syntax that declares such information is marked `/* point A */`. Therefore, the question is really asking what exception declaration would best suit this method. It's pretty clear that any situation that doesn't even compile cannot be considered satisfactory, so as long as some of the options would compile they must be "better than" any that do not. Consider the issue of compilation first.

If the method might throw any checked exceptions, it must carry a declaration that announces that. In this case, the only checked exception that is potentially thrown is the `IOException`; so at a minimum, the method must declare something that encompasses that exception. Options B and E fail on that point, because they declare unchecked exceptions. (Note that `OutOfMemoryError` falls into the category of "unchecked," although it's a subclass of `Throwable`, not of `Exception`.) For convenience, we'll simply use the term *unchecked exceptions* to include errors that are not parents of `IOException`. Therefore, options B and E are incorrect.

However, the remaining options all declare either `IOException` or a parent class of `IOException`. It's important to note that a `throws` clause that mentions a parent exception class is sufficient to encompass any child classes. Because of this, options A, C, and D all allow the code to compile correctly, so how can you choose the "best" option among these?

When you declare an exception in a `throws` clause, you impose an obligation on the caller of the method; the caller must do something about the exception. Also, the `throws` clause is a form of description of the type of problems that can arise when calling the method. These points both suggest that a `throws` clause should be as specific as possible. To be more general or to mention irrelevant exceptions might place an additional burden on the caller by creating a perceived requirement to handle situations that don't in fact arise. Further, additional generality will likely have the effect of hiding the real problem that might arise, making it harder for the caller to know how to respond if an exception is reported. On this basis, it's clear that option C, which reports one unchecked exception (`OutOfMemoryError`) with a checked exception that *cannot* arise, is unlikely to qualify as "best." Therefore, option C is incorrect.

Testing equality between different types almost always returns false regardless of the contents.

By the same arguments, you can also see that option A, which simply (and vaguely) reports that an `Exception` might arise, is also less helpful than option D, which gets directly to the point of reporting the single checked exception that could arise from the method. As a result, you can conclude that option A is incorrect, and option D is the correct answer.

There's another small point to consider as part of this discussion. The question mentions that the method is final. Why would that make any difference? It's certainly a tenuous point in this case, but it helps justify the "best" value judgment. Often, an abstract method in an interface declares a fairly general exception (consider the `close()` method in the `AutoCloseable` interface, which throws `Exception`). Given that such a method cannot possibly throw any exceptions because it doesn't have any implementation, why would this be? The answer is that it's prohib-

ited for an overriding method (which, of course, includes the methods that implement interface abstract methods) to declare checked exceptions that were not already permitted in the context of the overridden method. Without this restriction, you could have a reference of a parent type, and the compiler would let you call a method on it without handling a particular checked exception, but if the reference turns out to refer—at runtime—to an implementation that *does throw* that checked exception, you would have effectively cheated the checked exception mechanism. Generally, allowing an overriding method to do something not permitted for the overridden method would break the Liskov Substitution Principle, and in this specific case, it would break the protections provided by checked exceptions.

This means that if a method is expected to be overridden, it's not unreasonable to declare it as throwing some checked exceptions that simply don't arise in its current form. Had the method not been final, it would have been much harder to make a convincing case that option C was not the "best" choice (because it allows additional flexibility). But as it is, declaring throws `SQLException` is just a source of confusion, because the method does not throw `SQLException` nor is it possible that any overriding method might do so.

Answer 4. The correct answer is option E. This question investigates several aspects of `StringBuilder` and its relationship with `String`.

In option A, the code uses the `+` operator with two operands that are `StringBuilder` objects. One of Java's "special case" rules is that the only allowed operator overloading is the language-defined ability to concatenate `String` types using the `+` operator. Another fundamental rule is that when a `+` operator has a `String` type as an operand, if the other operand is not a `String`, it will be converted into one—and that, of course, brings up a third fundamental rule: in Java *any* data type can be converted into a `String`. (Admittedly, the conversion isn't always very helpful, but it's definitely legal.) However, in this case, although both operands represent "text" in the general sense—indeed, they're both instances of the interface `CharSequence`—neither operand is a `String`. Therefore, the code fails to compile, because it attempts to use the `+` operator with illegal arguments. Because of this, option A is incorrect.

//fix this /

Option B considers equality testing. This turns out to be a pretty simple rule, too. Testing equality between different types almost always returns false regardless of the contents (with some exceptions—take a look at the API-documented requirements for the `equals` methods in the `List` and `Set` implementations). However, in this question, you have two `StringBuilder`s that contain the same text. In this situation, it's easy to assume that the two objects will test as equal. However, that's not at all the case; indeed, relatively few of the core Java API classes implement a useful `equals` method, and `StringBuilder` is *not* one of them. The way you can determine this is by looking at the documentation of the class. Look at `String`'s `equals` method, and you'll see the API docs define how it tests for identical character sequences. But, look at `StringBuilder`; the only mention of the `equals` method is that it's "inherited from `java.lang.Object`." Of course, the default `equals` method defined by `Object` tests to see if two references refer to the same object in memory. As a result, the fragment in option B actually prints `false`, and option B is incorrect.

The final three options all hinge on related points. `String` and `StringBuilder` are different types on independent branches of the inheritance tree. As such, they are not assignment-compatible with one another. However, they also have elements of a shared type hierarchy; they're both subclasses of `Object`, and they both implement the `CharSequence` interface.

In these options, a ternary expression has `String` and `StringBuilder` as the two option values. The type of such an expression cannot be `String`; it must be some common parent of both arguments. Therefore, the attempt to assign the result of the ternary expression to the `String` variable `val` in option C will cause a compilation failure. Therefore, option C is incorrect.

In option D, the type of the variable `val` has been changed to `CharSequence`. This now forms a legal, compilable fragment. However, the test in the ternary expression `sb.equals(s)` will evaluate to false, because the arguments are of differing types, and `StringBuilder` does not handle that. Given that the test evaluates to false, the ternary expression as a whole evaluates to the third operand, and the fragment prints `Differ`. Because of this, option D is incorrect.

In option E, the intermediate variable `val` has been removed and the ternary expression is the argument to the `println` call. In this case, it's up to the compiler to find a suitable type

for the expression, and it doesn't really matter if it chooses `Object` or `CharSequence`. Either is a legitimate argument to the `println` method and, consequently, the code compiles successfully. Of course, the expression `sb.equals(s)` still evaluates to false and the output that is printed is `Differ`—as it was in option D. Therefore, option E is correct.

As a side note, the `CharSequence` interface isn't explicitly mentioned in the exam objectives. However, both `String` and `StringBuilder` are, and this interface is an aspect of both. We doubt you'll come across it in the real exam, but our excuse is that by using it here, we were able to make the example a little more interesting and, perhaps, teach something useful. We hope you'll forgive the indulgence! `</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 publishes recorded and live video training through Pearson InformIT (available direct and through the O'Reilly Safari Books Online service). He remains involved with Oracle's Java certification projects.

Mikalai Zaikin is a lead Java developer at IBA IT Park in Minsk, Belarus. During his career, he has helped Oracle with development of the Java certification exams, and he has been a technical reviewer of several Java certification books, including three editions of the famous *Sun Certified Programmer for Java* study guides by Kathy Sierra and Bert Bates.

//contact us /

Comments

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

Article Proposals

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

Customer Service

If you're having trouble with your subscription, please contact the folks at java@omeda.com, 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.

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.

- World's shortest subscription form
- Download area for code and other items
- *Java Magazine* in Japanese