Java 8 Lambdas

Richard Warburton

Beijing • Cambridge • Farnham • Köln • Sebastopol • Tokyo O'REILLY®

Java 8 Lambdas

by Richard Warburton

Printed in the United States of America.

Published by O'Reilly Media, Inc., 1005 Gravenstein Highway North, Sebastopol, CA 95472.

O'Reilly books may be purchased for educational, business, or sales promotional use. Online editions are also available for most titles (*http://my.safaribooksonline.com*). For more information, contact our corporate/institutional sales department: 800-998-9938 or *corporate@oreilly.com*.

Editor: Meghan Blanchette
Production Editor: Melanie Yarbrough
Copyeditor: Nancy Kotary
Proofreader: Rachel Head
Indexer: WordCo Indexing Services
Cover Designer: Karen Montgomery
Interior Designer: David Futato
Illustrator: Rebecca Demarest

March 2014: First Edition

Revision History for the First Edition:

2014-03-13: First release

See *http://oreilly.com/catalog/errata.csp?isbn=9781449370770* for release details.

ISBN: 978-1-449-37077-0

[LSI]

Table of Contents

Preface

For years, functional programming has been considered the realm of a small band of specialists who consistently claimed superiority to the masses while being unable to spread the wisdom of their approach. The main reason I've written this book is to challenge both the idea that there's an innate superiority in the functional style and the belief that its approach should be relegated to a small band of specialists!

For the last two years in the London Java Community, I've been getting developers to try out Java 8 in some form or another. I've found that many of our members enjoy the new idioms and libraries that it makes available to them. They may reel at the terminology and elitism, but they love the benefits that a bit of simple functional programming provides to them. A common thread is how much easier it is to read code using the new Streams API to manipulate objects and collections, such as filtering out albums that were made in the UK from a `List` of all albums.

What I've learned when running these kinds of events is that examples matter. People learn by repeatedly digesting simple examples and developing an understanding of patterns out of them. I've also noticed that terminology can be very off-putting, so anytime there's a hard-sounding concept, I give an easy-to-read explanation.

For many people, what Java 8 offers by way of functional programming is incredibly limited: no monads,[1] no language-level lazy evaluation, no additional support for immutability. As pragmatic programmers, this is fine; what we want is the ability to write library-level abstractions so we can write simple, clean code that solves business problems. We're even happier if someone else has written these libraries for us and we can just focus on doing our daily jobs.

1. This is the only mention of this word in this book.

Why Should I Read This Book?

In this book we'll explore:

- How to write simpler, cleaner, and easier-to-read code—especially around collections
- How to easily use parallelism to improve performance
- How to model your domain more accurately and build better DSLs
- How to write less error-prone and simpler concurrent code
- How to test and debug your lambda expressions

Developer productivity isn't the only reason why lambda expressions have been added to Java; there are fundamental forces in our industry at work here as well.

Who Should Read This Book?

This book is aimed squarely at Java developers who already have core Java SE skills and want to get up to speed on the big changes in Java 8.

If you're interested in reading about lambda expressions and how they can improve your lot as a professional developer, read on! I don't assume you know about lambda expressions themselves, or any of the core library changes; instead, I introduce concepts, libraries, and techniques from scratch.

Although I would love for every developer who has ever lived to go and buy this book, realistically, it's not appropriate for everyone. If you don't know any Java at all, this isn't the book for you. At the same time, though lambda expressions in Java are very well covered here, I don't explain how they are used in any other languages.

I don't provide a basic introduction to the use of several facets of the Java SE, such as collections, anonymous inner classes, or the event handling mechanism in Swing. I assume that you already know about all of these elements.

How to Read This Book

This book is written in an example-driven style: very soon after a concept is introduced, you'll see some code. Occasionally you might see something in the code that you're not 100% familar with. Don't worry—it'll be explained very soon afterward, frequently in the next paragraph.

This approach also lets you try out the ideas as you go along. In fact, at the end of most chapters there are further examples for you to practice on your own. I highly recommend that you try doing these katas as you get to the end of the chapter. Practice makes perfect,

and—as every pragmatic programmer knows—it's really easy to fool yourself into thinking that you understand some code when in reality you've missed a detail.

Because the use of lambda expressions is all about abstracting complexity away into libraries, I introduce a bunch of common library niceties as I go along. Chapters 2 through 6 cover the core language changes and also the improved libraries that JDK 8 brings.

The final three chapters are about applying functional programming in the wild. I'll talk about a few tricks that make testing and debugging code a bit easier in Chapter 7. Chapter 8 explains how existing principles of good software design also apply to lambda expressions. Then I talk about concurrency and how to use lambda expressions to write concurrent code that's easy to understand and maintain in Chapter 9. These chapters also introduce third-party libraries, where relevant.

It's probably worth thinking of the opening four chapters as the introductory material —things that everyone will need to know to use Java 8 properly. The latter chapters are more complex, but they also teach you how to be a more complete programmer who can confidently use lambda expressions in your own designs. There are also exercises as you go along, and the answers to these can be found on GitHub. If you practice the exercises as you go along, you'll soon master lambda expressions.

Conventions Used in This Book

The following typographical conventions are used in this book:

Italic
: Indicates new terms, URLs, email addresses, filenames, and file extensions.

`Constant width`
: Used for program listings, as well as within paragraphs to refer to program elements such as variable or function names, databases, data types, environment variables, statements, and keywords.

`Constant width bold`
: Shows commands or other text that should be typed literally by the user.

`Constant width italic`
: Shows text that should be replaced with user-supplied values or by values determined by context.

This element signifies a tip or suggestion.

This element signifies a general note.

This element indicates a warning or caution.

Using Code Examples

Supplemental material (code examples, exercises, etc.) is available for download at *https://github.com/RichardWarburton/java-8-lambdas-exercises*.

This book is here to help you get your job done. In general, if example code is offered with this book, you may use it in your programs and documentation. You do not need to contact us for permission unless you're reproducing a significant portion of the code. For example, writing a program that uses several chunks of code from this book does not require permission. Selling or distributing a CD-ROM of examples from O'Reilly books does require permission. Answering a question by citing this book and quoting example code does not require permission. Incorporating a significant amount of example code from this book into your product's documentation does require permission.

We appreciate, but do not require, attribution. An attribution usually includes the title, author, publisher, and ISBN. For example: "*Java 8 Lambdas* by Richard Warburton (O'Reilly). Copyright 2014 Richard Warburton, 978-1-449-37077-0."

If you feel your use of code examples falls outside fair use or the permission given above, feel free to contact us at *permissions@oreilly.com*.

Safari® Books Online

Safari Books Online is an on-demand digital library that delivers expert content in both book and video form from the world's leading authors in technology and business.

Technology professionals, software developers, web designers, and business and creative professionals use Safari Books Online as their primary resource for research, problem solving, learning, and certification training.

Safari Books Online offers a range of product mixes and pricing programs for organizations, government agencies, and individuals. Subscribers have access to thousands of books, training videos, and prepublication manuscripts in one fully searchable database

from publishers like O'Reilly Media, Prentice Hall Professional, Addison-Wesley Professional, Microsoft Press, Sams, Que, Peachpit Press, Focal Press, Cisco Press, John Wiley & Sons, Syngress, Morgan Kaufmann, IBM Redbooks, Packt, Adobe Press, FT Press, Apress, Manning, New Riders, McGraw-Hill, Jones & Bartlett, Course Technology, and dozens more. For more information about Safari Books Online, please visit us online.

How to Contact Us

Please address comments and questions concerning this book to the publisher:

O'Reilly Media, Inc.
1005 Gravenstein Highway North
Sebastopol, CA 95472
800-998-9938 (in the United States or Canada)
707-829-0515 (international or local)
707-829-0104 (fax)

We have a web page for this book, where we list errata, examples, and any additional information. You can access this page at *http://oreil.ly/java_8_lambdas*.

To comment or ask technical questions about this book, send email to *bookquestions@oreilly.com*.

For more information about our books, courses, conferences, and news, see our website at *http://www.oreilly.com*.

Find us on Facebook: *http://facebook.com/oreilly*

Follow us on Twitter: *http://twitter.com/oreillymedia*

Watch us on YouTube: *http://www.youtube.com/oreillymedia*

Acknowledgments

While the name on the cover of this book is mine, many other people have been influential and helpful in its publication.

Thanks should go firstly to my editor, Meghan, and the team at O'Reilly for making this process a pleasurable experience and accelerating their deadlines where appropriate. It was great to be introduced to Meghan by Martijn and Ben to begin with; this book would never have happened without that meeting.

The review process was a huge step in improving the overall quality of the book, and my heartfelt appreciation goes out to those who have helped as part of the formal and informal review process, including Martijn Verburg, Jim Gough, John Oliver, Edward

Wong, Brian Goetz, Daniel Bryant, Fred Rosenberger, Jaikiran Pai, and Mani Sarkar. Martijn in particular has been hugely helpful with his battle-won advice on writing a technical book.

It would also be remiss of me to ignore the Project Lambda development team at Oracle. Updating an established language is a big challenge, and they've done a great job in Java 8 of giving me something fun to write about and support. The London Java Community also deserves its share of praise for being so actively involved and supportive when helping to test out the early Java release and making it so easy to see what kinds of mistakes developers make and what can be fixed.

A lot of people have been incredibly supportive and helpful while I was going through the effort of writing a book. I'd like to specifically call out my parents, who have always been there whenever they were needed. It has also been great to have encouragement and positive comments from friends such as old compsoc members, especially Sadiq Jaffer and the Boys Brigade.

CHAPTER 1
Introduction

Before we begin our exploration of what lambda expressions are and how we can use them, you should at least understand why they exist to begin with. In this chapter, I'll cover that and also explain the structure and motivation of this book.

Why Did They Need to Change Java Again?

Java 1.0 was released in January 1996, and the world of programming has changed quite a bit since then. Businesses are requiring ever more complex applications, and most programs are executed on machines with powerful multicore CPUs. The rise of Java Virtual Machines (JVM), with efficient runtime compilers has meant that programmers can focus more on writing clean, maintainable code, rather than on code that's efficiently using every CPU clock cycle and every byte of memory.

The elephant in the room is the rise of multicore CPUs. Programming algorithms involving locks is error-prone and time-consuming. The `java.util.concurrent` package and the wealth of external libraries have developed a variety of concurrency abstractions that begin to help programmers write code that performs well on multicore CPUs. Unfortunately, we haven't gone far enough—until now.

There are limits to the level of abstractions that library writers can use in Java today. A good example of this is the lack of efficient parallel operations over large collections of data. Java 8 allows you to write complex collection-processing algorithms, and simply by changing a single method call you can efficiently execute this code on multicore CPUs. In order to enable writing of these kinds of bulk data parallel libraries, however, Java needed a new language change: lambda expressions.

Of course there's a cost, in that you must learn to write and read lambda-enabled code, but it's a good trade-off. It's easier for programmers to learn a small amount of new syntax and a few new idioms than to have to handwrite a large quantity of complex thread-safe code. Good libraries and frameworks have significantly reduced the cost

and time associated with developing enterprise business applications, and any barrier to developing easy-to-use and efficient libraries should be removed.

Abstraction is a concept that is familiar to us all from object-oriented programming. The difference is that object-oriented programming is mostly about abstracting over data, while functional programming is mostly about abstracting over behavior. The real world has both of these things, and so do our programs, so we can and should learn from both influences.

There are other benefits to this new abstraction as well. For many of us who aren't writing performance-critical code all the time, these are more important wins. You can write easier-to-read code—code that spends time expressing the intent of its business logic rather than the mechanics of how it's achieved. Easier-to-read code is also easier to maintain, more reliable, and less error-prone.

You don't need to deal with the verbosity and readbility issues surrounding anonymous inner classes when writing callbacks and event handlers. This approach allows programmers to work on event processing systems more easily. Being able to pass functions around easily also makes it easier to write lazy code that initializes values only when necessary.

In addition, the language changes that enable the additional collection methods, `default` methods, can be used by everyday programmers who are maintaining their own libraries.

It's not your grandfather's Java any longer, and that's a good thing.

What Is Functional Programming?

Functional programming is a term that means different things to different people. At the heart of functional programming is thinking about your problem domain in terms of immutable values and functions that translate between them.

The communities that have developed around different programming languages each tend to think that the set of features that have been incorporated into their language are the key ones. At this stage, it's a bit too early to tell how Java programmers will define functional programming. In a sense, it's unimportant; what we really care about is writing *good* code rather than functional code.

In this book, I focus on pragmatic functional programming, including techniques that can be used and understood by most developers and that help them write programs that are easier to read and maintain.

Example Domain

Throughout the book, examples are structured around a common problem domain: music. Specifically, the examples represent the kind of information you might see on albums. Here's a brief summary of the terms:

Artist
: An individual or group who creates music

  - *name*: The name of the artist (e.g., "The Beatles")
  - *members*: A set of other artists who comprise this group (e.g., "John Lennon"); this field might be empty
  - *origin*: The primary location of origin of the group (e.g., "Liverpool").

Track
: A single piece of music

  - *name*: The name of the track (e.g., "Yellow Submarine")

Album
: A single release of music, comprising several tracks

  - *name*: The name of the album (e.g., "Revolver")
  - *tracks*: A list of tracks
  - *musicians*: A list of artists who helped create the music on this album

This domain is used to illustrate how to use functional programming techniques within a normal business domain or Java application. You may not consider it the perfect example subject, but it's simple, and many of the code examples in this book will bear similarity to those that you may see in your business domain.

CHAPTER 2

Lambda Expressions

The biggest language change in Java 8 is the introduction of lambda expressions—a compact way of passing around behavior. They are also a pretty fundamental building block that the rest of this book depends upon, so let's get into what they're all about.

Your First Lambda Expression

Swing is a platform-agnostic Java library for writing graphical user interfaces (GUIs). It has a fairly common idiom in which, in order to find out what your user did, you register an *event listener*. The event listener can then perform some action in response to the user input (see Example 2-1).

Example 2-1. Using an anonymous inner class to associate behavior with a button click

```
button.addActionListener(new ActionListener() {
    public void actionPerformed(ActionEvent event) {
        System.out.println("button clicked");
    }
});
```

In this example, we're creating a new object that provides an implementation of the `ActionListener` class. This interface has a single method, `actionPerformed`, which is called by the `button` instance when a user actually clicks the on-screen button. The anonymous inner class provides the implementation of this method. In Example 2-1, all it does is print out a message to say that the button has been clicked.

This is actually an example of using *code as data*—we're giving the button an object that represents an action.

Anonymous inner classes were designed to make it easier for Java programmers to pass around code as data. Unfortunately, they don't make it easy enough. There are still four lines of boilerplate code required in order to call the single line of important logic. Look how much gray we get if we color out the boilerplate:

```
button.addActionListener(new ActionListener() {
    public void actionPerformed(ActionEvent event) {
        System.out.println("button clicked");
    }
});
```

Boilerplate isn't the only issue, though: this code is fairly hard to read because it obscures the programmer's intent. We don't want to pass in an object; what we really want to do is pass in some behavior. In Java 8, we would write this code example as a lambda expression, as shown in Example 2-2.

Example 2-2. Using a lambda expression to associate behavior with a button click

```
button.addActionListener(event -> System.out.println("button clicked"));
```

Instead of passing in an object that implements an interface, we're passing in a block of code—a function without a name. `event` is the name of a parameter, the same parameter as in the anonymous inner class example. `->` separates the parameter from the body of the lambda expression, which is just some code that is run when a user clicks our button.

Another difference between this example and the anonymous inner class is how we declare the variable `event`. Previously, we needed to explicitly provide its type—`ActionEvent event`. In this example, we haven't provided the type at all, yet this example still compiles. What is happening under the hood is that `javac` is inferring the type of the variable `event` from its context—here, from the signature of `addActionListener`. What this means is that you don't need to explicitly write out the type when it's obvious. We'll cover this inference in more detail soon, but first let's take a look at the different ways we can write lambda expressions.

Although lambda method parameters require less boilerplate code than was needed previously, they are still statically typed. For the sake of readability and familiarity, you have the option to include the type declarations, and sometimes the compiler just can't work it out!

How to Spot a Lambda in a Haystack

There are a number of variations of the basic format for writing lambda expressions, which are listed in Example 2-3.

Example 2-3. Some different ways of writing lambda expressions

```
Runnable noArguments = () -> System.out.println("Hello World");  ❶

ActionListener oneArgument = event -> System.out.println("button clicked");  ❷

Runnable multiStatement = () -> {  ❸
   System.out.print("Hello");
   System.out.println(" World");
};

BinaryOperator<Long> add = (x, y) -> x + y;  ❹

BinaryOperator<Long> addExplicit = (Long x, Long y) -> x + y;  ❺
```

❶ shows how it's possible to have a lambda expression with no arguments at all. You can use an empty pair of parentheses, `()`, to signify that there are no arguments. This is a lambda expression implementing `Runnable`, whose only method, `run`, takes no arguments and is a `void` return type.

❷ we have only one argument to the lambda expression, which lets us leave out the parentheses around the arguments. This is actually the same form that we used in Example 2-2.

Instead of the body of the lambda expression being just an expression, in ❸ it's a full block of code, bookended by curly braces (`{}`). These code blocks follow the usual rules that you would expect from a method. For example, you can return or throw exceptions to exit them. It's also possible to use braces with a single-line lambda, for example to clarify where it begins and ends.

Lambda expressions can also be used to represent methods that take more than one argument, as in ❹. At this juncture, it's worth reflecting on how to *read* this lambda expression. This line of code doesn't add up two numbers; it creates a function that adds together two numbers. The variable called `add` that's a `BinaryOperator<Long>` isn't the result of adding up two numbers; it is code that adds together two numbers.

So far, all the types for lambda expression parameters have been inferred for us by the compiler. This is great, but it's sometimes good to have the option of explicitly writing the type, and when you do that you need to surround the arguments to the lambda expression with parentheses. The parentheses are also necessary if you've got multiple arguments. This approach is demonstrated in ❺.

The *target type* of a lambda expression is the type of the context in which the lambda expression appears—for example, a local variable that it's assigned to or a method parameter that it gets passed into.