O'REILLY®

Becoming Functional

STEPS FOR TRANSFORMING INTO A FUNCTIONAL PROGRAMMER

Joshua Backfield

O'REILLY®

Becoming Functional

If you have an imperative (and probably object-oriented) programming background, this hands-on book will guide you through the alien world of functional programming. Author Joshua Backfield begins slowly by showing you how to apply the most useful implementation concepts before taking you further into functional-style concepts and practices.

In each chapter, you'll learn a functional concept and then use it to refactor the fictional XXY company's imperative-style legacy code, writing and testing the functional code yourself. As you progress through the book, you'll migrate from Java 7 to Groovy and finally to Scala as the need for better functional language support gradually increases.

- Learn why today's finely tuned applications work better with functional code
- Transform imperative-style patterns into functional code, following basic steps
- Get up to speed with Groovy and Scala through examples
- Understand how first-class functions are passed and returned from other functions
- Convert existing methods into pure functions, and loops into recursive methods
- Change mutable variables into immutable variables
- Get hands-on experience with statements and nonstrict evaluations
- Use functional programming alongside object-oriented design

"Scala is a bridging language that helps programmers make the transition from OO to functional. Future languages, while they might not be called functional, will be heavily influenced by functional paradigms. So regardless of whether you plan on using Scala or just want to better appreciate the concepts behind other emerging languages, this is the book for you."

—**Francesco Cesarini**
Founder and Technical Director at Erlang Solutions

Joshua F. Backfield is a Senior Software Development Engineer at Dell SecureWorks, Inc., responsible for the design and development of many internal UI tools and multiple backend processes. He has worked in a variety of languages, including C, C++, Perl, Java, JavaScript, and Scala.

PROGRAMMING/FUNCTIONAL

US $29.99 CAN $31.99

ISBN: 978-1-449-36817-3

Twitter: @oreillymedia
facebook.com/oreilly

Becoming Functional

Joshua Backfield

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

Becoming Functional
by Joshua Backfield

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*.

Editors: Meghan Blanchette and Brian Anderson
Production Editor: Kristen Brown
Copyeditor: Rachel Monaghan
Proofreader: Becca Freed
Indexer: Ellen Troutman
Cover Designer: Karen Montgomery
Interior Designer: David Futato
Illustrator: Rebecca Demarest

July 2014: First Edition

Revision History for the First Edition:

2014-06-30: First release

See *http://oreilly.com/catalog/errata.csp?isbn=9781449368173* for release details.

ISBN: 978-1-449-36817-3

[LSI]

Table of Contents

Preface

Although not a new concept, functional programming has started to take a larger hold in the programming community. Features such as immutable variables and pure functions have proven helpful when we have to debug code, and higher-order functions make it possible for us to extract the inner workings of functions and write less code over time. All of this leads to more expressive code.

Who Is This Book For?

I wrote this book for anyone who is interested in functional programming or is looking to transition from an imperative style to a functional one. If you've been programming in an imperative or object-oriented style, my hope is that you'll be able to pick up this book and start learning how to code in a functional one instead.

This book will teach you how to recognize patterns in an imperative style and then walk you through how to transition into a more functional one. We will approach this by looking at a fictional company called XXY and look at their legacy code. We'll then refactor its legacy code from an imperative style into a functional one.

We're going to use a few different languages throughout this book:

Java
: I assume that you are familiar with the Java syntax. The version used in this book is 1.7.0.

Groovy
: Using this language, we can keep most of our existing Java syntax; this helps us begin our transition into a fully functional language. I'll explain the main parts of the Groovy syntax as they are needed. The version used in this book is 2.0.4.

Scala

This is a fully functional language into which we will slowly transition. As with Groovy, I will explain the syntax as it is introduced. The version used in this book is 2.10.0.

Why No Java 8?

Some people might wonder why I'm not including any Java 8 right now. As of this writing, Java 7 is the currently stable and widely used version. Because I want everyone, not just early adopters, to be able to take something from this book, I thought starting from Java 7 would be most accessible.

Those using Java 8 will be able to use some of the Groovy concepts, such as higher-order functions, without actually transitioning into Groovy.

Math Notation Review

Because functional programming is so closely tied to mathematics, let's go over some basic mathematical notation.

Functions in mathematics are represented with a *name*(*parameters*) = *body* style. The example in Equation P-1 shows a very simple function. The *name* is *f*, the *parameter list* is *x*, the *body* is *x* + *1*, and the *return* is the numeric result of x + 1.

Equation P-1. A simple math function

$$f(x) = x + 1$$

`if` statements in math are represented by the array notation. We will have a list of operations in which one will be evaluated when the corresponding `if` statement is true. The simple example in Equation P-2 shows a set of statements to be evaluated. The function `abs(x)` will return `x * -1` if our `x` is less than `0`; otherwise, it will return `x`.

Equation P-2. A simple math if statement

$$abs(x) = \begin{cases} x * -1 & \text{if } x < 0 \\ x & \text{else} \end{cases}$$

We also use a *summation*, the sigma operator, in our notation. The example in Equation P-3 shows a simple summation. The notation says to have a variable `n` starting at `0` (defined by the `n=0` below the sigma) and continuing to `x` (as defined by the `x` above

the sigma). Then, for each `n` we add it to our sum (defined by the body, `n` in our case, to the right of the sigma).

Equation P-3. A simple math summation

$$f(x) = \sum_{n=0}^{x} n$$

Why Functional over Imperative?

There are quite a few paradigms, each with its own pros and cons. Imperative, functional, event-driven—all of these paradigms represent another way of programming. Most people are familiar with the imperative style because it is the most common style of programming. Languages such as Java and C languages are all imperative by design. Java incorporates object-oriented programming (OOP) into its language, but it still primarily uses an imperative paradigm.

One of the most common questions I've heard during my time in software is "why should I bother learning functional programming?" Because most of my new projects have been in languages like Scala, the easiest response I can give is "that is what the project is written in." But let's take a step back and actually answer the question in depth.

I've seen quite a bit of imperative code that requires cryptographers to fully understand what it does. Generally, with the imperative style, you can write code and make it up as you go. You can write classes upon classes without fully understanding what the implementation will be. This usually results in a very large, unsustainable code base filled with an overuse of classes and spaghetti code.

Functional programming, on the other hand, forces us to better understand our implementation before and while we're coding. We can then use that to identify where abstractions should go and reduce the lines of code we have written to execute the same functionality.

Why Functional Alongside OOP?

When we think of OOP, we normally think of a paradigm that is in a class of its own. But if we look at how we write OOP, the OOP is really used for encapsulation of variables into objects. Our code is actually in the imperative style—that is, it is executed "top to bottom." As we transition to functional programming, we'll see many more instances in which we just pass function returns into other functions.

Some people see functional programming as a replacement for OOP, but in fact we'll still use OOP so that we can continue using objects that can maintain methods. These methods, hoever, will usually call static versions that allow us to have purer and more

testable functions. So, we're not replacing OOP; rather, we're using object-oriented design in a functional construct.

Why Functional Programming Is Important

Concepts such as design patterns in Java are so integral to our daily programming that it's almost impossible to imagine life without them. So it is very interesting that, by contrast, the functional style has been around for many years but remains in the background as a main programming paradigm.

Why, then, is functional programming becoming so much more important today if it's been around so long? Well, think back to the dot-com era, a time when any web presence was better than none. And what about general applications? As long as the application worked, nobody cared about the language or paradigm in which it was written.

Requirements and expectations today are difficult, so being able to closely mirror mathematical functions allows engineers to design strong algorithms in advance and rely on developers to implement those algorithms within the time frame required. The closer we bind ourselves to a mathematical underpinning, the better understood our algorithms will be. Functional programming also allows us to apply mathematics on those functions. Using concepts such as derivatives, limits, and integrals on functions can be useful when we are trying to identify where functions might fail.

Large functions are not very testable and also not very readable. Often, as software developers, we find ourselves presented with large chunks of functionality thrown into one function. But, if we extract the inner workings of these large, cumbersome functions into multiple, smaller, more understandable functions, we allow for more code reuse as well as higher levels of testing.

Code reuse and higher levels of testing are two of the most important benefits of moving to a functional language. Being able to extract entire chunks of functionality from a function makes it possible for us to change the functionality later without using a copy-and-paste methodology.

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 icon signifies a tip, suggestion, or general note.

This icon indicates a warning or caution.

Math Warning
Every now and again, I'll introduce some mathematics; I'll try to warn you beforehand. Check out the section "Math Notation Review" on page viii if you are rusty on reading mathematical notations.

Using Code Examples

Supplemental material (code examples, exercises, etc.) is available for download at *https://github.com/jbackfield/BecomingFunctional*.

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: "*Becoming Functional* by Joshua Backfield (O'Reilly). Copyright 2014 Joshua Backfield, 978-1-449-36817-3."

If you feel your use of code examples falls outside fair use or the aforementioned permission, 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://bit.ly/becoming-functional*.

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

I'd like to thank my wife, Teri, and my daughter, Alyssa, for putting up with me during the writing of this book. I'd also like to thank Kevin Schmidt for introducing me to Simon St.Laurent, who made this book a reality, and my bosses, Gary Herndon and Alan Honeycutt, for allowing me to push the boundaries at work and try new things. I'd especially like to thank Meghan Blanchette, who kept moving me along and made sure that I was continuing to make progress along the way. Finally, I want to thank my parents, Sue and Fred Backfield, for believing in me and pushing me to continue learning and growing when I was a kid. If it weren't for all of you making a difference in my life, I wouldn't be here sharing my knowledge with so many other aspiring developers today.

There are lots of other people I've met along the way who have helped me become a better developer; I know I'm going to leave people out (and I'm sorry if I do), but here is a good attempt at a list: Nick Angus, Johnny Calhoun, Jason Pinkey, Ryan Karetas, Isaac Henry, Jim Williams, Mike Wisener, Yatin Kanetkar, Sean McNealy, Christopher Heath, and Dave Slusher.