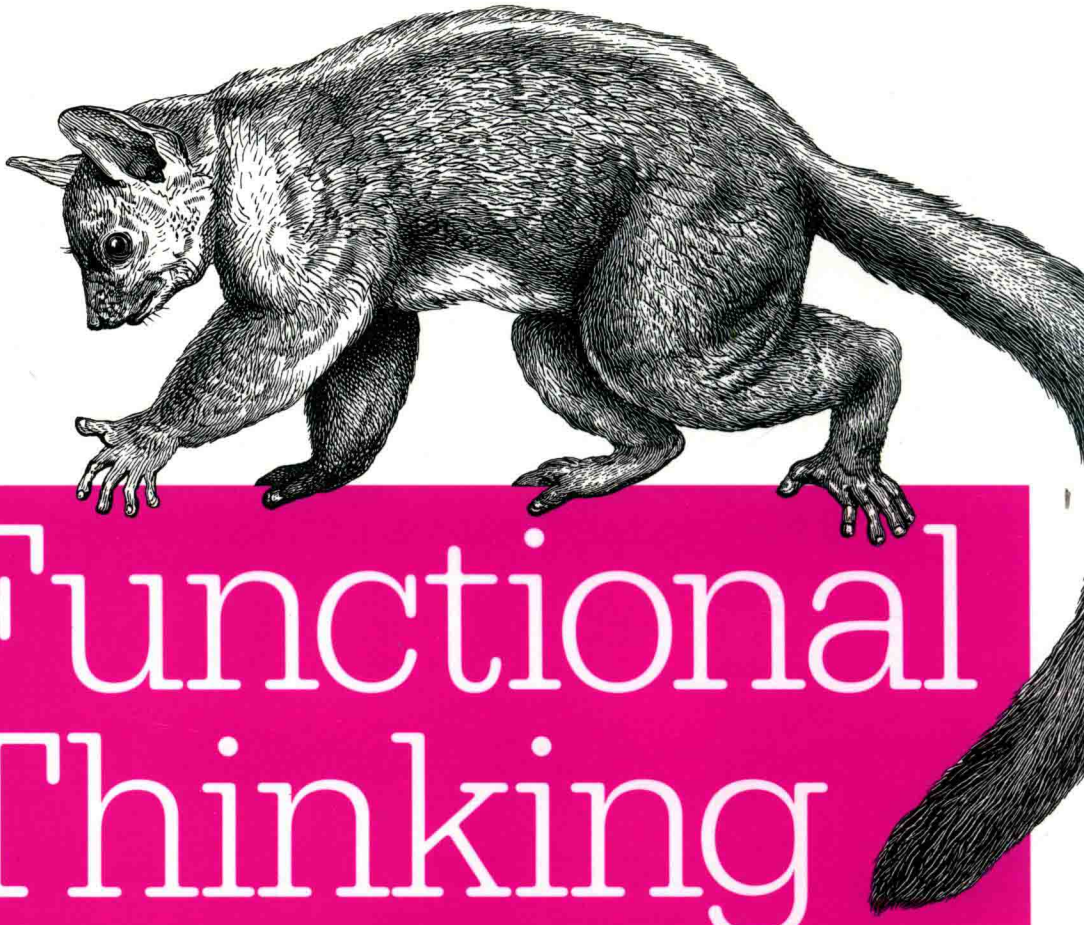
O'REILLY®



Functional Thinking

函数式编程思想（影印版）

東南大學出版社

Neal Ford 著

函数式编程思想（影印版）

Functional Thinking

Neal Ford 著

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

O'Reilly Media, Inc.授权东南大学出版社出版

南京 东南大学出版社

图书在版编目(CIP)数据

函数式编程思想：英文/(美)弗德(Ford, N.)著. —影印本. —南京：东南大学出版社，2015.2

书名原文：Functional Thinking

ISBN 978-7-5641-5388-5

Ⅰ. ①函… Ⅱ. ①弗… Ⅲ. ①函数—程序设计—英文 Ⅳ. ①TP311.1

中国版本图书馆 CIP 数据核字(2014)第 294381 号

江苏省版权局著作权合同登记

图字：10-2014-156 号

函数式编程思想(影印版)

出版发行：东南大学出版社
地　　址：南京四牌楼 2 号　　邮编：210096
出 版 人：江建中
网　　址：http://www.seupress.com
电子邮件：press@seupress.com
印　　刷：常州市武进第三印刷有限公司
开　　本：787 毫米×980 毫米　　16 开本
印　　张：11.25
字　　数：220 千字
版　　次：2015 年 2 月第 1 版
印　　次：2015 年 2 月第 1 次印刷
书　　号：ISBN 978-7-5641-5388-5
定　　价：39.00 元

本社图书若有印装质量问题，请直接与营销部联系。电话(传真)：025-83791830

Preface

The first time I seriously looked at functional programming was in 2004. I became intrigued by alternative languages on the .NET platform, and started playing with Haskell and a few pre-F#, ML-based languages. In 2005, I did a conference talk named "Functional Languages and .NET" at a few venues, but the languages at the time were more proof-of-concept and toys than anything else. The possibilities of thinking within a new paradigm fascinated me, however, and changed the way I approached some problems in more familiar languages.

I revisited this topic in 2010 because I observed the rise of languages such as Clojure and Scala in the Java ecosystem and remembered the cool stuff from five years before. I started one afternoon on Wikipedia, following link after link, and was mesmerized by the end of the day. That started an exploration of numerous branches of thought in the functional programming world. That research culminated in the "Functional Thinking" talk, debuting in 2011 at the *33rd Degree Conference* (*http://33degree.org*) in Poland and the IBM developerWorks article series of the same name (*http://bit.ly/dev-works-ft-series*). Over the course of the next two years, I wrote an article each month on functional programming, which was a great way to establish and maintain a research and exploration plan. I continued delivering (and refining, based on feedback) the presentation until the present day.

This book is the culmination of all the ideas from the "Functional Thinking" talk and article series. I've found that the best way to hone material is to present it to audiences over and over, because I learn something new about the material every time I present or write about it. Some relationships and commonalities appear only after deep research and forced thought (deadlines are great focusers!).

My last book, *Presentation Patterns* (*http://presentationpatterns.com*), described the importance of visual metaphors in conference presentations. For *Functional Thinking*, I chose a blackboard and chalk theme (to invoke the mathematical connection to functional programming concepts). At the end of the presentation, as I talk about practical

applications, I show a picture of a piece of chalk resting at the foot of a blackboard, metaphorically imploring viewers to pick it up and explore these ideas on their own.

My goal in the talk, the article series, and this book is to present the core ideas of functional programming in a way that is accessible to developers steeped in imperative, object-oriented languages. I hope you enjoy this distillation of ideas, and pick up the chalk and continue your own exploration.

—Neal Ford, Atlanta, June 2014

Chapter Overview

Each chapter in this book shows examples of functional thinking. Chapter 1, *Why*, provides a broad overview and shows some examples of the mental shift prevalent in the rest of the book. Chapter 2, *Shift*, describes the gradual process of shifting your perspective from that of an object-oriented, imperative programmer to that of a functional programmer. To illustrate the *shift* in thinking required, I solve a common problem in both imperative and functional styles. I then do an extensive case study, showing the way a functional perspective (and some helper syntax) can help *shift* you toward a functional mindset.

Chapter 3, *Cede*, shows examples of common chores you can now cede to your language or runtime. One of the "moving parts" described by Michael Feathers is *state*, which is typically managed explicitly in nonfunctional languages. Closures allow you to defer some state-handling to the runtime; I show examples of how that state handling mechanism works underneath. In this chapter, I show how functional thinking also allows you to cede details like accumulation to recursion, and impacts your granularity of code reuse.

Chapter 4, *Smarter, Not Harder*, focuses on two extended examples of *eliminating moving parts* by allowing the runtime to cache function results for you and implementing laziness. Many functional languages include *memoization* (either natively, via a library, or a trivial implementation), which handles a common performance optimization for you. I show an example, based on the *number classifier* example in Chapter 2, of several levels of optimization, both handwritten and via memoization. At the risk of giving away the ending, memoization wins. *Lazy* data structures, which defer calculation until necessary, allow you to think differently about data structures. I show how to implement lazy data structures (even in nonfunctional languages) and how to leverage laziness that already exists.

Chapter 5, *Evolve*, shows how languages are evolving to become more functional. I also talk about evolutionary language trends such as operator overloading and new dispatch options beyond just method calls, about bending your language toward your problem (not the other way around), and common functional data structures such as `Option`.

Chapter 6, *Advance*, shows examples of common approaches to problems. I show how design patterns change (or disappear) in the functional programming world. I also contrast code reuse via inheritance versus composition and discuss their explicit and implicit coupling points.

Chapter 7, *Practical Thinking*, shows specific long-anticipated functional features that recently appeared in the Java Developer Kit (JDK). I show how Java 8 fits in with the *functional thinking* from other languages, including the use of higher-order functions (i.e., lambda blocks). I also discuss some clever ways in which Java 8 maintains graceful backward compatibility, and I highlight the `Stream` API, which allows concise and descriptive workflows. And, finally, I show how Java 8 has added `Option` to eliminate potential `null` confusion. I also cover topics such as functional architecture and databases and how the functional perspective changes those designs.

Chapter 8, *Polyglot and Polyparadigm*, describes the impact of functional programming on the polyglot world we now live in; we increasingly encounter and incorporate numerous languages on projects. Many new languages are also *polyparadigm*, supporting several different programming models. For example, Scala supports object-oriented and functional programming. The last chapter discusses the pros and cons of living in a paradigmatically richer world.

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.

Using Code Examples

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

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: "*Functional Thinking* by Neal Ford (O'Reilly). Copyright 2014 Neal Ford, 978-1-449-36551-6."

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://bit.ly/functional-thinking*.

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

Thanks to my family at ThoughtWorks, the best place of employment around, and to all my fellow speakers on the conference circuit, especially the No Fluff, Just Stuff speakers, against whom I've bounced many ideas. Thanks to all the people who have attended my "Functional Thinking" talks at conferences over the years—your feedback helped me hone this material. Special thanks to the technical reviewers on this book, who made outstanding substantive suggestions, especially the early readers who took the time to submit errata, many of which exposed subtle opportunities for clarification. Thanks to friends and family too numerous to mention who act as my incredible support network, especially John Drescher, who looks after the cats when we're away. And, of course, my long-suffering wife Candy, who long ago lost hope that I would ever stop doing this.

Table of Contents

CHAPTER 1
Why

Let's say for a moment that you are a lumberjack. You have the best axe in the forest, which makes you the most productive lumberjack in the camp. Then one day someone shows up and extols the virtues of a new tree-cutting paradigm, the *chainsaw*. The sales guy is persuasive, so you buy a chainsaw, but you don't know how it works. Demonstrating your expertise with the previous tree-cutting paradigm, you swing it vigorously at a tree—without cranking it. You quickly conclude that this newfangled chainsaw is a fad, and you return to your axe. Then, someone appears and shows you how to *crank* the chainsaw.

The problem with a completely new programming paradigm isn't learning a new language. After all, everyone reading this has learned numerous computer languages—language syntax is merely details. The tricky part is learning to *think* in a different way.

This book explores the subject of functional programming but isn't really about functional programming languages. Make no mistake—I show lots of code, in numerous languages; this book is all about code. As I'll illustrate, writing code in a "functional" manner touches on design trade-offs, different reusable building blocks, and a host of other insights. Because I favor ideas over syntax, I start with Java, the most familiar baseline for the largest group of developers, and mix in both pre-Java 8 and Java 8 examples. As much as possible, I show functional programming concepts in Java (or close relatives) and move to other languages only to demonstrate unique capabilities.

Even if you don't care about Scala or Clojure, and are happy coding in your current language for the rest of your career, your language will change underneath you, looking more functional all the time. Now is the time to learn functional paradigms, so that you can leverage them when (not if) they appear in your everyday language. Let's take a look at the reasons why all languages are gradually becoming more functional.

Shifting Paradigms

Computer science often advances in fits and starts, with good ideas appearing decades before they suddenly become part of the mainstream. For example, Simula 67, created in 1967, is the first object-oriented language, yet object orientation didn't really become mainstream until after the popularization of C++, which first appeared in 1983. Often, good ideas await foundation technologies to catch up. In its early years, Java was regularly considered too slow and expansive in memory usage for high-performance applications, but shifts in the hardware market made it an attractive choice.

Functional programming follows the same conceptual trajectory as object orientation: developed in academia over the last few decades, it has slowly crept into all modern programming languages. Yet just adding new syntax to a language doesn't inform developers of the best way to leverage this new way of thinking.

I start with a contrast between the traditional programming style (imperative loops) and a more functional way of solving the same problem. For the problem to solve, I dip into a famous event in computer science history, a challenge issued from Jon Bentley, the writer of a regular column in *Communications of the ACM* called "Programming Pearls," to Donald Knuth, an early computer science pioneer. The challenge is common to anyone who has written text-manipulation code: *read a file of text, determine the most frequently used words, and print out a sorted list of those words along with their frequencies.* Just tackling the word-frequency portion, I write a solution in "traditional" Java, shown in Example 1-1.

Example 1-1. Word frequencies in Java

```
public class Words {
    private Set<String> NON_WORDS = new HashSet<String>() {{
        add("the"); add("and"); add("of"); add("to"); add("a");
        add("i"); add("it"); add("in"); add("or"); add("is");
        add("d"); add("s"); add("as"); add("so"); add("but");
        add("be"); }};

    public Map wordFreq(String words) {
        TreeMap<String, Integer> wordMap = new TreeMap<String, Integer>();
        Matcher m = Pattern.compile("\\w+").matcher(words);
        while (m.find()) {
            String word = m.group().toLowerCase();
            if (! NON_WORDS.contains(word)) {
                if (wordMap.get(word) == null) {
                    wordMap.put(word, 1);
                }
                else {
                    wordMap.put(word, wordMap.get(word) + 1);
                }
            }
        }
        return wordMap;
```

```
    }
}
```

In Example 1-1, I create a set of nonwords (articles and other "glue" words), then create the `wordFreq()` method. In it, I build a `Map` to hold the key/value pairs, then create a regular expression to allow me to determine words. The bulk of this listing iterates over the found words, ensuring that the actual word is either added the first time to the map or its occurrence is incremented. This style of coding is quite common in languages that encourage you to work through collections (such as regular expression matches) piecemeal.

Consider the updated version that takes advantage of the `Stream` API and the support for higher-order functions via lambda blocks in Java 8 (all discussed in more detail later), shown in Example 1-2.

Example 1-2. Word frequency in Java 8

```
private List<String> regexToList(String words, String regex) {
    List wordList = new ArrayList<>();
    Matcher m = Pattern.compile(regex).matcher(words);
    while (m.find())
    wordList.add(m.group());
    return wordList;
}

public Map wordFreq(String words) {
    TreeMap<String, Integer> wordMap = new TreeMap<>();
    regexToList(words, "\\w+").stream()
            .map(w -> w.toLowerCase())
            .filter(w -> !NON_WORDS.contains(w))
            .forEach(w -> wordMap.put(w, wordMap.getOrDefault(w, 0) + 1));
    return wordMap;
}
```

In Example 1-2, I convert the results of the regular expression match to a stream, which allows me to perform discrete operations: make all the entries lowercase, filter out the nonwords, and count the frequencies of the remaining words. By converting the iterator returned via `find()` to a stream in the `regexToList()` method, I can perform the required operations one after the other, in the same way that I think about the problem. Although I could do that in the imperative version in Example 1-1 by looping over the collection three times (once to make each word lowercase, once to filter nonwords, and once to count occurrences), I know not to do that because it would be terribly inefficient. By performing all three operations within the iterator block in Example 1-1, I'm trading *clarity* for *performance*. Although this is a common trade-off, it's one I'd rather not make.

In his "Simple Made Easy" keynote (*http://www.infoq.com/presentations/Simple-Made-Easy*) at the Strange Loop conference, Rich Hickey, the creator of Clojure (*http://clojure.org*), reintroduced an arcane word, *complect*: to join by weaving or twining

together; to interweave. Imperative programming often forces you to *complect* your tasks so that you can fit them all within a single loop, for efficiency. Functional programming via higher-order functions such as `map()` and `filter()` allows you to elevate your level of abstraction, seeing problems with better clarity. I show many examples of functional thinking as a powerful antidote to incidental complecting.

Aligning with Language Trends

If you look at the changes coming to all major languages, they all add functional extensions. Groovy has been adding functional capabilities for a while, including advanced features such as memoization (the ability for the runtime to cache function return values automatically). Even Java itself will finally grow more functional extensions, as lambda blocks (i.e., higher-order functions) finally appear in Java 8, and arguably the most widely used language, JavaScript, has numerous functional features. Even the venerable C++ added lambda blocks in the language's 2011 standard, and has generally shown more functional programming interest, including intriguing libraries such as the Boost.Phoenix library (*http://bit.ly/phoenix-library*).

Learning these paradigms now allows you to utilize the features as soon as they appear, either in your use of a new language such as Clojure or in the language you use every day. In Chapter 2, I cover how to *shift* your thinking to take advantage of these advanced facilities.

Ceding Control to the Language/Runtime

During the short history of computer science, the mainstream of technology sometimes spawns branches, either practical or academic. For example, in the 1990s, the move to personal computers saw an explosion in popularity of fourth-generation languages (4GL) such as dBASE, Clipper, FoxPro, Paradox, and a host of others. One of the selling points of these languages was a higher-level abstraction than a 3GL like C or Pascal. In other words, you could issue a single command in a 4GL that would take many commands in a 3GL because the 4GL already had more "prebaked" context. These languages were already equipped to read popular database formats from disk rather than force customized implementations.

Functional programming is a similar offshoot from academia, where computer scientists wanted to find ways of expressing new ideas and paradigms. Every so often, a branch will rejoin the mainstream, which is what is happening to functional programming now. Functional languages are sprouting not just on the Java Virtual Machine (JVM), where the two most interesting new languages are Scala and Clojure, but on the .NET platform as well, which includes F# as a first-class citizen. Why this embrace of functional programming by all the platforms?

Back in the early 1980s, when I was in university, we used a development environment called Pecan Pascal, whose unique feature was the ability to run the same Pascal code on either the Apple][or IBM PC. The Pecan engineers achieved this feat by using something mysterious called "bytecode." When the developer compiled his code, he compiled it to this "bytecode," which ran on a "virtual machine," written natively for each of the two platforms. And it was a hideous experience. The resulting code was achingly slow even for simple class assignments. The hardware at the time just wasn't up to the challenge.

Of course, we all recognize this architecture. A decade after Pecan Pascal, Sun released Java using the same techniques, straining but succeeding in mid-1990s hardware environments. It also added other developer-friendly features, such as automatic garbage collection. I never want to code in a non-garbage-collected language again. Been there, done that, got the T-shirt, and don't want to go back, because I'd rather spend my time at a higher level of abstraction, thinking about ways to solve complex business scenarios, not complicated plumbing problems. I rejoice in the fact that Java reduces the pain of explicit memory management, and I try to find that same level of convenience in other places.

Life's too short for `malloc`.

Over time, developers cede more control over tedious tasks to our languages and runtimes. I don't lament the lack of direct memory control for the types of applications I write, and ignoring that allows me to focus on more important problems. Java eased our interaction with memory management; functional programming languages allow us to replace other core building blocks with higher-order abstractions.

Examples of replacing detailed implementations with simpler ones relying on the runtime to handle mundane details abound in this book.

Concision

Michael Feathers, author of *Working with Legacy Code*, captured a key difference between functional and object-oriented abstractions in 140 lowly characters on Twitter (*https://twitter.com/mfeathers/status/29581296216*):

> OO makes code understandable by encapsulating moving parts. FP makes code understandable by minimizing moving parts.
>
> — Michael Feathers

Think about the things you know about object-oriented programming (OOP) constructs: encapsulation, scoping, visibility, and other mechanisms exist to exert fine-grained control over who can see and change state. More complications pile up when you deal with state plus threading. These mechanisms are what Feathers referred to as "moving parts." Rather than build mechanisms to *control* mutable state, most functional languages try to *remove* mutable state, a "moving part." The theory follows that if the language exposes fewer potentially error-prone features, it is less likely for developers to make errors. I will show numerous examples throughout of functional programming eliminating variables, abstractions, and other moving parts.

In object-oriented imperative programming languages, the units of reuse are classes and the messages they communicate with, captured in a class diagram. The seminal work in that space, *Design Patterns: Elements of Reusable Object-Oriented Software* (by Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides), includes at least one class diagram with each pattern. In the OOP world, developers are encouraged to create unique data structures, with specific operations attached in the form of methods. Functional programming languages don't try to achieve reuse in quite the same way. In functional programming languages, the preference is for a few key data structures (such as `list`, `set`, and `map`) with highly optimized operations on those data structures. To utilize this machinery, developers pass data structures plus higher-order functions to plug into the machinery, customizing it for a particular use.

Consider a simplified portion of the code from Example 1-2:

```
regexToList(words, "\\b\\w+\\b").stream()
                .filter(w -> !NON_WORDS.contains(w))
```

To retrieve a subset of a list, call the `filter()` method, passing the list as a stream of values and a higher-order function specifying the filter criteria (in this case, the syntactically sugared `(w → !NON_WORDS.contains(w))`). The machinery applies the filter criteria in an efficient way, returning the filtered list.

Encapsulation at the function level allows reuse at a more granular, fundamental level than building new class structures for every problem. Dozens of XML libraries exist in the Java world, each with its own internal data structures. One advantage of leveraging higher-level abstractions is already appearing in the Clojure space. Recent clever innovations in Clojure's libraries have managed to rewrite the `map` function to be automatically parallelizable, meaning that all map operations get a performance boost without developer intervention.

Functional programmers prefer a few core data structures, building optimized machinery to understand them. Object-oriented programmers tend to create new data structures and attendant operations constantly—building new classes and messages between them is the predominant object oriented paradigm. Encapsulating all data structures within classes discourages reuse at the method level, preferring larger framework-style