

Mastering JavaScript Functional Programming

Second Edition

Write clean, robust, and maintainable web and server code using functional JavaScript



Packt>

www.packt.com

Federico Kereki

Mastering JavaScript Functional Programming *Second Edition*

Write clean, robust, and maintainable web and server code
using functional JavaScript

Federico Kereki

Packt

BIRMINGHAM - MUMBAI

Mastering JavaScript Functional Programming

Second Edition

Copyright © 2020 Packt Publishing

All rights reserved. No part of this book may be reproduced, stored in a retrieval system, or transmitted in any form or by any means, without the prior written permission of the publisher, except in the case of brief quotations embedded in critical articles or reviews.

Every effort has been made in the preparation of this book to ensure the accuracy of the information presented. However, the information contained in this book is sold without warranty, either express or implied. Neither the author, nor Packt Publishing or its dealers and distributors, will be held liable for any damages caused or alleged to have been caused directly or indirectly by this book.

Packt Publishing has endeavored to provide trademark information about all of the companies and products mentioned in this book by the appropriate use of capitals. However, Packt Publishing cannot guarantee the accuracy of this information.

Commissioning Editor: Wilson D'souza
Acquisition Editor: Shweta Bairoliya
Content Development Editor: Aamir Ahmed
Senior Editor: Hayden Edwards
Technical Editor: Jane Dsouza
Copy Editor: Safis Editing
Project Coordinator: Manthan Patel
Proofreader: Safis Editing
Indexer: Manju Arasan
Production Designer: Joshua Misquitta

First published: November 2017
Second edition: January 2020

Production reference: 1240120

Published by Packt Publishing Ltd.
Livery Place
35 Livery Street
Birmingham
B3 2PB, UK.

ISBN 978-1-83921-306-9

www.packt.com

Writing a book involves many people, and even if I cannot mention and name all of them, there are some who really deserve to be highlighted.

At Packt Publishing, I want to thank Larissa Pinto, Senior Acquisition Editor, for proposing the theme for this book and helping me get started with it. Thanks must also go to Mohammed Yusuf Imaratwale, Content Development Editor, and Ralph Rosario, Technical Editor, for their help in giving shape to the book and making it clearer and better structured. I also want to send my appreciation to the reviewers, Gerónimo García Sgritta and Steve Perkins, who went through the initial draft, enhancing it with their comments.

There are some other people who deserve extra consideration. This book was written under unusual circumstances, around 10,000 miles away from home! I had gone from Uruguay, where I live, to work on a project in India, and that's where I wrote every single page of the text. This would not have been possible if I hadn't had complete support from my family, who stayed in Montevideo, but who were constantly nearby, thanks to the internet and modern communications. In particular, I must single out my wife, Sylvia Tosar, not only for supporting and aiding me both with the project and the book, but also for dealing with everything, and the rest of the family on her own in Uruguay—this book wouldn't have been possible otherwise, and she is the greatest reason the book could be written!

For the second edition

Revisiting and expanding a book for a second edition is a challenging, interesting task. I had great support from Packt, and I must now thank Aamir Ahmed, Content Development Editor; Jane D'souza, Technical Editor; and Crystian Bietti and (again, for double merit!) Steve Perkins, reviewers—both of whom helped produce a much better text.

– Federico Kereki



Packt.com

Subscribe to our online digital library for full access to over 7,000 books and videos, as well as industry leading tools to help you plan your personal development and advance your career. For more information, please visit our website.

Why subscribe?

- Spend less time learning and more time coding with practical eBooks and Videos from over 4,000 industry professionals
- Improve your learning with Skill Plans built especially for you
- Get a free eBook or video every month
- Fully searchable for easy access to vital information
- Copy and paste, print, and bookmark content

Did you know that Packt offers eBook versions of every book published, with PDF and ePub files available? You can upgrade to the eBook version at www.packt.com and as a print book customer, you are entitled to a discount on the eBook copy. Get in touch with us at customercare@packtpub.com for more details.

At www.packt.com, you can also read a collection of free technical articles, sign up for a range of free newsletters, and receive exclusive discounts and offers on Packt books and eBooks.

Contributors

About the author

Federico Kereki is an Uruguayan systems engineer, with a master's degree in education, and more than 30 years of experience as a consultant, system developer, university professor, and writer.

He is currently a subject matter expert at Globant, where he gets to use a good mixture of development frameworks, programming tools, and operating systems, such as JavaScript, Node.js, React and Redux, SOA, Containers, and PHP, with both Windows and Linux.

He has taught several computer science courses at Universidad de la República, Universidad ORT Uruguay, and Universidad de la Empresa. He has also written texts for these courses.

He has written several articles—on JavaScript, web development, and open source topics—for magazines such as *Linux Journal* and *LinuxPro Magazine* in the United States, *Linux+* and *Mundo Linux* in Europe, and for websites such as Linux.com and IBM Developer Works. He has also written booklets on computer security (*Linux in the Time of Malware* and *SSH: A Modern Lock for Your Server*), a book on GWT programming (*Essential GWT: Building for the Web with Google Web Toolkit*), and another one on JavaScript development (*Modern JavaScript Web Development Cookbook*).

Federico has given talks on functional programming with JavaScript at public conferences (such as JSCONF 2016 and Development Week Santiago 2019) and has used these techniques to develop internet systems for businesses in Uruguay and abroad.

His current interests tend toward software quality and software engineering—with agile methodologies topmost—while on the practical side, he works with diverse languages, tools, and frameworks, and open source software (FLOSS) wherever possible!

He usually resides, works, and teaches in Uruguay, but this book was fully written while on a project in India, and the revisions for the second edition were finished during a sojourn in Mexico.

About the reviewers

Steve Perkins is the author of *Hibernate Search by Example*. He has been working with Java and JavaScript since the late-1990's, with forays into Scala, Groovy, and Go. Steve lives in Atlanta, GA, with his wife and two children, and is currently a software architect at Banyan Hills Technologies, where he works on a platform for IoT device management and analytics.

When he is not writing code or spending time with family, Steve plays guitar and loses games at bridge and backgammon. You can visit his technical blog at steveperkins.com, and follow him on Twitter at [@stevedperkins](https://twitter.com/stevedperkins).

Cristian "Pusher" Bietti is an entrepreneur who is proactive and has a creative attitude to facing challenges in new technologies, with a great hunger to learn more! A senior developer with more than 18 years of experience in software development and software design and trained in a wide variety of technologies, he has participated in big banking projects and small applications for mobile and social networks, including video games.

He has focused on frontend and user experience (UI/UX). He is a subject matter expert at Globant, and he works in the Fintech industry as a technical leader and developer because he loves coding.

Packt is searching for authors like you

If you're interested in becoming an author for Packt, please visit authors.packtpub.com and apply today. We have worked with thousands of developers and tech professionals, just like you, to help them share their insight with the global tech community. You can make a general application, apply for a specific hot topic that we are recruiting an author for, or submit your own idea.

Table of Contents

Preface	1
Technical Requirements	7
Chapter 1: Becoming Functional - Several Questions	8
What is functional programming?	8
Theory versus practice	9
A different way of thinking	10
What FP is not	10
Why use FP?	11
What we need	11
What we get	12
Not all is gold	13
Is JavaScript functional?	13
JavaScript as a tool	14
Going functional with JavaScript	15
Key features of JavaScript	16
Functions as first-class objects	16
Recursion	17
Closures	18
Arrow functions	19
Spread	20
How do we work with JavaScript?	22
Using transpilers	23
Working online	25
Testing	26
Summary	27
Questions	27
Chapter 2: Thinking Functionally - A First Example	29
Our problem – doing something only once	29
Solution 1 – hoping for the best!	30
Solution 2 – using a global flag	31
Solution 3 – removing the handler	32
Solution 4 – changing the handler	33
Solution 5 – disabling the button	33
Solution 6 – redefining the handler	34
Solution 7 – using a local flag	34
A functional solution to our problem	35
A higher-order solution	36
Testing the solution manually	37

Testing the solution automatically	39
Producing an even better solution	41
Summary	42
Questions	43
Chapter 3: Starting Out with Functions - A Core Concept	44
All about functions	44
Of lambdas and functions	45
Arrow functions – the modern way	48
Returning values	49
Handling the this value	49
Working with arguments	51
One argument or many?	53
Functions as objects	54
A React-Redux reducer	55
An unnecessary mistake	57
Working with methods	58
Using functions in FP ways	59
Injection – sorting it out	59
Callbacks, promises, and continuations	62
Continuation passing style	62
Polyfills	64
Detecting Ajax	64
Adding missing functions	66
Stubbing	67
Immediate invocation	68
Summary	71
Questions	71
Chapter 4: Behaving Properly - Pure Functions	73
Pure functions	73
Referential transparency	74
Side effects	76
Usual side effects	76
Global state	77
Inner state	78
Argument mutation	80
Troublesome functions	81
Advantages of pure functions	83
Order of execution	83
Memoization	84
Self-documentation	88
Testing	88
Impure functions	89
Avoiding impure functions	89
Avoiding the usage of state	89
Injecting impure functions	91
Is your function pure?	93

Testing – pure versus impure	94
Testing pure functions	95
Testing purified functions	96
Testing impure functions	99
Summary	102
Questions	102
Chapter 5: Programming Declaratively - A Better Style	104
Transformations	105
Reducing an array to a value	105
Summing an array	107
Calculating an average	108
Calculating several values at once	110
Folding left and right	111
Applying an operation – map	113
Extracting data from objects	115
Parsing numbers tacitly	116
Working with ranges	117
Emulating map() with reduce()	119
Dealing with arrays of arrays	120
Flattening an array	120
Mapping and flattening – flatMap()	123
Emulating flat() and flatMap()	125
More general looping	127
Logical higher-order functions	129
Filtering an array	130
A reduce() example	131
Emulating filter() with reduce()	132
Searching an array	133
A special search case	134
Emulating find() and findIndex() with reduce()	134
Higher-level predicates – some, every	135
Checking negatives – none	136
Working with async functions	137
Some strange behaviors	138
Async-ready looping	140
Looping over async calls	140
Mapping async calls	141
Filtering with async calls	142
Reducing async calls	143
Summary	144
Questions	144
Chapter 6: Producing Functions - Higher-Order Functions	147
Wrapping functions – keeping behavior	148
Logging	149
Logging in a functional way	149
Taking exceptions into account	151

Working in a purer way	152
Timing functions	155
Memoizing functions	157
Simple memoization	158
More complex memoization	160
Memoization testing	162
Altering a function's behavior	165
Doing things once, revisited	165
Logically negating a function	168
Inverting the results	169
Arity changing	171
Changing functions in other ways	172
Turning operations into functions	172
Implementing operations	173
A handier implementation	174
Turning functions into promises	175
Getting a property from an object	176
Demethodizing – turning methods into functions	178
Finding the optimum	180
Summary	182
Questions	182
Chapter 7: Transforming Functions - Currying and Partial Application	184
A bit of theory	185
Currying	186
Dealing with many parameters	186
Currying by hand	189
Currying with bind()	191
Currying with eval()	194
Partial application	196
Partial application with arrow functions	197
Partial application with eval()	198
Partial application with closures	202
Partial currying	205
Partial currying with bind()	206
Partial currying with closures	209
Final thoughts	210
Parameter order	210
Being functional	212
Summary	213
Questions	214
Chapter 8: Connecting Functions - Pipelining and Composition	216
Pipelining	217
Piping in Unix/Linux	217
Revisiting an example	219

Creating pipelines	220
Building pipelines by hand	220
Using other constructs	222
Debugging pipelines	224
Using tee	224
Tapping into a flow	226
Using a logging wrapper	227
Pointfree style	228
Defining pointfree functions	228
Converting to pointfree style	229
Chaining and fluent interfaces	231
An example of fluent APIs	231
Chaining method calls	232
Composing	235
Some examples of composition	235
Unary operators	236
Counting files	237
Finding unique words	237
Composing with higher-order functions	239
Testing composed functions	243
Transducing	248
Composing reducers	251
Generalizing for all reducers	252
Summary	253
Questions	254
Chapter 9: Designing Functions - Recursion	256
Using recursion	257
Thinking recursively	258
Decrease and conquer – searching	259
Decrease and conquer – doing powers	260
Divide and conquer – the Towers of Hanoi	261
Divide and conquer – sorting	264
Dynamic programming – making change	265
Higher-order functions revisited	267
Mapping and filtering	268
Other higher-order functions	271
Searching and backtracking	274
The eight queens puzzle	274
Traversing a tree structure	278
Recursion techniques	281
Tail call optimization	282
Continuation passing style	285
Trampolines and thunks	289
Recursion elimination	292
Summary	292
Questions	293

Chapter 10: Ensuring Purity - Immutability	296
Going the straightforward JavaScript way	297
Mutator functions	297
Constants	298
Freezing	299
Cloning and mutating	301
Getters and setters	305
Getting a property	305
Setting a property by path	306
Lenses	308
Working with lenses	308
Implementing lenses with objects	311
Implementing lenses with functions	314
Prisms	317
Working with prisms	317
Implementing prisms	320
Creating persistent data structures	320
Working with lists	321
Updating objects	323
A final caveat	328
Summary	328
Questions	329
Chapter 11: Implementing Design Patterns - The Functional Way	331
Understanding design patterns	332
Design pattern categories	333
Do we need design patterns?	334
Object-oriented design patterns	335
Facade and adapter	336
Decorator or wrapper	338
Strategy, Template, and Command	344
Observer and reactive programming	346
Basic concepts and terms	347
Operators for observables	349
Detecting multi-clicks	352
Providing typeahead searches	354
Other patterns	358
Functional design patterns	359
Summary	361
Questions	361
Chapter 12: Building Better Containers - Functional Data Types	364
Specifying data types	364
Signatures for functions	365
Other data type options	367
Building containers	369

Extending current data types	370
Containers and functors	372
Wrapping a value – a basic container	372
Enhancing our container – functors	374
Dealing with missing values with Maybe	376
Dealing with varying API results	378
Implementing Prisms	382
Monads	384
Adding operations	385
Handling alternatives – the Either monad	388
Calling a function – the Try monad	391
Unexpected monads – promises	392
Functions as data structures	393
Binary trees in Haskell	393
Functions as binary trees	395
Summary	401
Questions	401
Bibliography	403
Answers to Questions	405
Other Books You May Enjoy	433
Index	436

Preface

In computer programming, paradigms abound. Some examples include imperative programming, structured (*goto-less*) programming, **object-oriented programming (OOP)**, aspect-oriented programming, and declarative programming. Lately, there has been renewed interest in a particular paradigm that can arguably be considered to be older than most (if not all) of the cited ones—**Functional Programming (FP)**. FP emphasizes writing functions and connecting them in simple ways to produce a more understandable and more easily tested code. Thus, given the increased complexity of today's web applications, it's logical that a safer, cleaner way of programming would be of interest.

This interest in FP comes hand in hand with the evolution of JavaScript. Despite its somewhat hasty creation (reportedly managed in only 10 days, in 1995, by Brendan Eich at Netscape), today it's a standardized and quickly growing language, with features more advanced than most other similarly popular languages. The ubiquity of the language, which can now be found in browsers, servers, mobile phones, and whatnot, has also impelled interest in better development strategies. Also, even if JavaScript wasn't conceived as a functional language by itself, the fact is that it provides all the features you'd require to work in that fashion, which is another plus.

It must also be said that FP hasn't been generally applied in industry, possibly because it has a certain aura of difficulty, and it is thought to be *theoretical* rather than *practical*, even *mathematical*, and possibly uses vocabulary and concepts that are foreign to developers—for example, functors? Monads? Folding? Category theory? While learning all this theory will certainly be of help, it can also be argued that even with zero knowledge of the previous terms, you can understand the tenets of FP, and see how to apply it in your own programming.

FP is not something that you have to do on your own, without any help. There are many libraries and frameworks that incorporate, in greater or lesser degrees, the concepts of FP. Starting with jQuery (which does include some FP concepts), passing through Underscore and its close relative, Lodash, or other libraries such as Ramda, and getting to more complete web development tools such as React and Redux, Angular, or Elm (a 100% functional language, which compiles into JavaScript), the list of functional aids for your coding is ever growing.

Learning how to use FP can be a worthwhile investment, and even though you may not get to use all of its methods and techniques, just starting to apply some of them will pay dividends in better code. You need not try to apply all of FP from the start, and you need not try to abandon every non-functional feature in the language either. JavaScript assuredly has some bad features, but it also has several very good and powerful ones. The idea is not to throw away everything you've learned and use and adopt a 100% functional way; rather, the guiding idea is *evolution, not revolution*. In that sense, it can be said that what we'll be doing is not FP, but rather **Sorta Functional Programming (SFP)**, aiming for a fusion of paradigms.

A final comment about the style of the code in this book—it is quite true that there are several very good libraries that provide you with FP tools: Underscore, Lodash, Ramda, and more are counted among them. However, I preferred to eschew their usage, because I wanted to show how things really work. It's easy to apply a given function from some package or other, but by coding everything out (a *vanilla FP*, if you wish), it's my belief that you get to understand things more deeply. Also, as I will comment in some places, because of the power and clarity of arrow functions and other features, the *pure JavaScript* versions can be even simpler to understand!

Who this book is for

This book is geared toward programmers with a good working knowledge of JavaScript, working either on the client side (browsers) or the server side (Node.js), who are interested in applying techniques to be able to write better, testable, understandable, and maintainable code. Some background in computer science (including, for example, data structures) and good programming practices will also come in handy.

What this book covers

In this book, we'll cover FP in a practical way; though, at times, we will mention some theoretical points:

Chapter 1, *Becoming Functional – Several Questions*, discusses FP, gives reasons for its usage, and lists the tools that you'll need to take advantage of the rest of the book.

Chapter 2, *Thinking Functionally – A First Example*, will provide the first example of FP by considering a common web-related problem and going over several solutions, to finally center on a functional solution.

Chapter 3, *Starting Out with Functions – A Core Concept*, will go over the central concept of FP, that is, functions, and the different options available in JavaScript.

Chapter 4, *Behaving Properly – Pure Functions*, will consider the concept of purity and pure functions, and demonstrate how it leads to simpler coding and easier testing.

Chapter 5, *Programming Declaratively – A Better Style*, will use simple data structures to show how to produce results that work not in an imperative way, but in a declarative fashion.

Chapter 6, *Producing Functions – Higher-Order Functions*, will deal with higher-order functions, which receive other functions as parameters and produce new functions as results.

Chapter 7, *Transforming Functions – Currying and Partial Application*, will explore some methods for producing new and specialized functions from earlier ones.

Chapter 8, *Connecting Functions – Pipelining and Composition*, will show the key concepts regarding how to build new functions by joining previously defined ones.

Chapter 9, *Designing Functions – Recursion*, will look at how a key concept in FP, recursion, can be applied to designing algorithms and functions.

Chapter 10, *Ensuring Purity – Immutability*, will present some tools that can help you to work in a pure fashion by providing immutable objects and data structures.

Chapter 11, *Implementing Design Patterns – The Functional Way*, will show how several popular OOP design patterns are implemented (or not needed!) when you program in FP ways.

Chapter 12, *Building Better Containers – Functional Data Types*, will explore some more high-level functional patterns, introducing types, containers, functors, monads, and several other more advanced FP concepts.

I have tried to keep the examples in this book simple and down to earth because I want to focus on the functional aspects and not on the intricacies of this or that problem. Some programming texts are geared toward learning, say, a given framework, and then work on a given problem, showing how to fully work it out with the chosen tools. (In fact, at the very beginning of planning for this book, I entertained the idea of developing an application that would use all the FP things I had in mind, but there was no way to fit all of that within a single project. Exaggerating a bit, I felt like an MD trying to find a patient on whom to apply all of his medical knowledge and treatments!) So, I have opted to show plenty of individual techniques, which can be used in multiple situations. Rather than building a house, I want to show you how to put the bricks together, how to wire things up, and so on, so that you will be able to apply whatever you need, as you see fit.

To get the most out of this book

To understand the concepts and code in this book, you don't need much more than a JavaScript environment and a text editor. To be honest, I even developed some of the examples working fully online, with tools such as JSFiddle (at <https://jsfiddle.net/>) and the like, and absolutely nothing else.

In this book, we'll be using ES2019, Node 13, and the code will run on any OS such as Linux, Mac OSX, or Windows; please do check the *Technical Requirements* section, for some other tools we'll also work with.

Finally, you will need some experience with the latest version of JavaScript, because it includes several features that can help you write more concise and compact code. We will frequently include pointers to online documentation, such as the documentation available on the **Mozilla Development Network (MDM)** at <https://developer.mozilla.org/>, to help you get more in-depth knowledge.

Download the example code files

You can download the example code files for this book from your account at www.packt.com. If you purchased this book elsewhere, you can visit www.packtpub.com/support and register to have the files emailed directly to you.

You can download the code files by following these steps:

1. Log in or register at www.packt.com.
2. Select the **Support** tab.
3. Click on **Code Downloads**.
4. Enter the name of the book in the **Search** box and follow the onscreen instructions.

Once the file is downloaded, please make sure that you unzip or extract the folder using the latest version of:

- WinRAR/7-Zip for Windows
- Zipeg/iZip/UnRarX for Mac
- 7-Zip/PeaZip for Linux

The code bundle for the book is also hosted on GitHub at <https://github.com/PacktPublishing/Mastering-JavaScript-Functional-Programming-2nd-Edition->. In case there's an update to the code, it will be updated on the existing GitHub repository.

We also have other code bundles from our rich catalog of books and videos available at <https://github.com/PacktPublishing/>. Check them out!

Conventions used

There are a number of text conventions used throughout this book.

CodeInText: Indicates code words in text, database table names, folder names, filenames, file extensions, pathnames, dummy URLs, user input, and Twitter handles. Here is an example: "Let's review our `once()` function."

A block of code is set as follows:

```
function newCounter() {
  let count = 0;
  return function() {
    count++;
    return count;
  };
}

const nc = newCounter();
console.log(nc()); // 1
console.log(nc()); // 2
console.log(nc()); // 3
```

When we wish to draw your attention to a particular part of a code block, the relevant lines or items are set in bold:

```
function fact (n) {
  if (n === 0) {
    return 1;

  } else {
    return n * fact (n - 1);
  }
}

console.log(fact(5)); // 120
```

Bold: Indicates a new term, an important word, or words that you see on screen. For example, words in menus or dialog boxes appear in the text like this. Here is an example: "Select the **EXPERIMENTAL** option to fully enable ES10 support."



Warnings or important notes appear like this.



Tips and tricks appear like this.

Get in touch

Feedback from our readers is always welcome.

General feedback: If you have questions about any aspect of this book, mention the book title in the subject of your message and email us at customer@packtpub.com.

Errata: Although we have taken every care to ensure the accuracy of our content, mistakes do happen. If you have found a mistake in this book, we would be grateful if you would report this to us. Please visit www.packtpub.com/support/errata, selecting your book, clicking on the Errata Submission Form link, and entering the details.

Piracy: If you come across any illegal copies of our works in any form on the internet, we would be grateful if you would provide us with the location address or website name. Please contact us at copyright@packt.com with a link to the material.

If you are interested in becoming an author: If there is a topic that you have expertise in and you are interested in either writing or contributing to a book, please visit authors.packtpub.com.

Reviews

Please leave a review. Once you have read and used this book, why not leave a review on the site that you purchased it from? Potential readers can then see and use your unbiased opinion to make purchase decisions, we at Packt can understand what you think about our products, and our authors can see your feedback on their book. Thank you!

For more information about Packt, please visit packt.com.

Technical Requirements

To develop and test the code in this book, I used several versions of commonly available software, including browsers and Node.js, as well as some other packages.

For this second edition, my main machine runs the *Tumbleweed* rolling release of OpenSUSE Linux, from <https://www.opensuse.org/#Tumbleweed>, currently including kernel 5.3.5. (The *rolling* term implies that the software is updated on a continuous basis, to keep getting the latest versions of all packages.) I've also tested portions of the code of this book on different Windows 7 and Windows 10 machines.

As to browsers, I usually work with Chrome, from <https://www.google.com/chrome/browser/>, and at the current time, I'm up to version 78. I also use Firefox, from <https://www.mozilla.org/en-US/firefox/>, and I got version 72 in my machine. I have also run code using the online JSFiddle environment, at <https://jsfiddle.net/>.

On the server side, I use Node.js, from <https://nodejs.org/>, currently at version 13.6.

For transpilation, I used Babel, from <https://babeljs.io/>: the current version of the `babel-cli` package is 7.7.7.

For testing, I went with Jasmine, from <https://jasmine.github.io/>, and the latest version in my machine is 3.5.0.

Finally, for code formatting, I used Prettier, from <https://prettier.io/>. You can either install it locally, or run it online at <https://prettier.io/playground/>; the version I have is 1.19.1.

The JavaScript world is quite dynamic, and it's a safe bet that by the time you get to read this book, all the software listed above will have been updated several times. Every single piece of software I used when I wrote the 1st edition of this book, received several updates over time. However, given the standardization of JavaScript, and the high importance of back compatibility, you shouldn't have problems with other versions.

1

Becoming Functional - Several Questions

Functional programming (or **FP**) has been around since the earliest days of computing, and is going through a sort of revival because of its increased use with several frameworks and libraries, most particularly in **JavaScript (JS)**. In this chapter, we shall do the following:

- Introduce some concepts of FP to give a small taste of what it means.
- Show the benefits (and problems) implied by the usage of FP and why we should use it.
- Start thinking about why JavaScript can be considered an appropriate language for FP.
- Go over the language features and tools that you should be aware of in order to fully take advantage of everything in this book.

By the end of this chapter, you'll have the basic tools that we'll be using in the rest of the book, so let's get started by learning about functional programming.

What is functional programming?

If you go back in computer history, you'll find that the second oldest programming language still in use, Lisp, is based on FP. Since then, there have been many more functional languages, and FP has been applied more widely. But even so, if you ask people what FP is, you'll probably get two widely dissimilar answers.



For trivia or history buffs, the oldest language still in use is Fortran, which appeared in 1957, a year before Lisp. Quite shortly after Lisp came another long-lived language, COBOL, for business-oriented programming.

Depending on whom you ask, you'll either learn that it's a modern, advanced, enlightened approach to programming that leaves every other paradigm behind or that it's mainly a theoretical thing, with more complications than benefits, practically impossible to implement in the real world. And, as usual, the real answer is not in the extremes, but somewhere in between. Let's start by looking at the theory versus practice and see how we plan to use FP.

Theory versus practice

In this book, we won't be going about FP in a theoretical way. Instead, our point is to show you how some of its techniques and tenets can be successfully applied for common, everyday JavaScript programming. But—and this is important—we won't be going about this in a dogmatic fashion, but in a very practical way. We won't dismiss useful JavaScript constructs simply because they don't happen to fulfill the academic expectations of FP. Similarly, we won't avoid practical JavaScript features just to fit the FP paradigm. In fact, we could almost say that we'll be doing **Sorta Functional Programming (SFP)** because our code will be a mixture of FP features, more classical imperative ones, and **object-oriented programming (OOP)**.

Be careful, though: what we just said doesn't mean that we'll be leaving all the theory by the side. We'll be picky, and just touch the main theoretical points, learn some vocabulary and definitions, and explain core FP concepts, but we'll always be keeping in sight the idea of producing actual, useful JavaScript code, rather than trying to meet some mystical, dogmatic FP criteria.

OOP has been a way to solve the inherent complexity of writing large programs and systems, and developing clean, extensible, scalable application architectures; however, because of the scale of today's web applications, the complexity of all codebases is continuously growing. Also, the newer features of JavaScript make it possible to develop applications that wouldn't even have been possible just a few years ago; think of mobile (hybrid) apps that are made with Ionic, Apache Cordova, or React Native or desktop apps that are made with Electron or NW.js, for example. JavaScript has also migrated to the backend with Node.js, so today, the scope of usage for the language has grown in a serious way that deals with all the added complexity of modern designs.

A different way of thinking

FP is a different way of writing programs, and can sometimes be difficult to learn. In most languages, programming is done in an imperative fashion: a program is a sequence of statements, executed in a prescribed fashion, and the desired result is achieved by creating objects and manipulating them, which usually means modifying the objects themselves. FP is based on producing the desired result by evaluating expressions built out of functions that are composed together. In FP, it's common to pass functions around (such as passing parameters to other functions or returning functions as the result of a calculation), to not use loops (opting for recursion instead), and to skip side effects (such as modifying objects or global variables).

In other words, FP focuses on *what* should be done, rather than on *how*. Instead of worrying about loops or arrays, you work at a higher level, considering what you need to be done. After becoming accustomed to this style, you'll find that your code becomes simpler, shorter, and more elegant, and can be easily tested and debugged. However, don't fall into the trap of considering FP as the goal! Think of FP only as a means to an end, as with all software tools. Functional code isn't good just for being functional, and writing bad code is just as possible with FP as with any other technique!

What FP is not

Since we have been saying some things about what FP is, let's also clear up some common misconceptions, and look at what FP is *not*:

- **FP isn't just an academic ivory tower thing:** It is true that the **lambda calculus** upon which it is based was developed by Alonzo Church in 1936 as a tool to prove an important result in theoretical computer science (which preceded modern computer languages by more than 20 years!); however, FP languages are being used today for all kinds of systems.
- **FP isn't the opposite of object-oriented programming (OOP):** It isn't a case of choosing declarative or imperative ways of programming. You can mix and match as best suits you, and we'll be doing this throughout this book, bringing together the best of all worlds.
- **FP isn't overly complex to learn:** Some of the FP languages are rather different from JavaScript, but the differences are mostly syntactic. Once you learn the basic concepts, you'll see that you can get the same results in JavaScript as with FP languages.

It may also be relevant to mention that several modern frameworks, such as the React and Redux combination, include FP ideas.

For example, in React, it's said that the **view** (whatever the user gets to see at a given moment) is a function of the current **state**. You use a function to compute what HTML and CSS must be produced at each moment, thinking in a **black-box** fashion.

Similarly, in Redux you have the concept of **actions** that are processed by **reducers**. An action provides some data, and a reducer is a function that produces the new state for the application in a functional way out of the current state and the provided data.

So, both because of the theoretical advantages (we'll be getting to those in the following section) and the practical ones (such as getting to use the latest frameworks and libraries), it makes sense to consider FP coding. Let's get on with it.

Why use FP?

Throughout the years, there have been many programming styles and fads. However, FP has proven quite resilient and is of great interest today. Why would you want to use FP? The question should rather first be, *what do you want to get?* and only then, *does FP get you that?* Let's answer these important questions in the following sections.

What we need

We can certainly agree that the following list of concerns is universal. Our code should have the following qualities:

- **Modular:** The functionality of your program should be divided into independent modules, each of which contains what it needs to perform one aspect of the program's functionality. Changes in a module or function shouldn't affect the rest of the code.
- **Understandable:** A reader of your program should be able to discern its components, their functions, and their relationships without undue effort. This is closely linked with the **maintainability** of the code; your code will have to be maintained at some time in the future, whether to be changed or to have new functionality added.
- **Testable:** **Unit tests** try out small parts of your program, verifying their behavior independently of the rest of the code. Your programming style should favor writing code that simplifies the job of writing unit tests. Unit tests are also like documentation in that they can help readers understand what the code is supposed to do.

- **Extensible:** It's a fact that your program will someday require maintenance, possibly to add new functionality. Those changes should impact the structure and data flow of the original code only minimally (if at all). Small changes shouldn't imply large, serious refactoring of your code.
- **Reusable:** **Code reuse** has the goal of saving resources, time, and money, and reducing redundancy by taking advantage of previously written code. There are some characteristics that help this goal, such as **modularity** (which we already mentioned), **high cohesion** (all the pieces in a module belong together), **low coupling** (modules are independent of each other), **separation of concerns** (the parts of a program should overlap in functionality as little as possible), and **information hiding** (internal changes in a module shouldn't affect the rest of the system).

What we get

So does FP give you the five characteristics we just listed in the previous section?

- In FP, the goal is to write separate independent functions that are joined together to produce the final results.
- Programs that are written in a functional style usually tend to be cleaner, shorter, and easier to understand.
- Functions can be tested on their own, and FP code has advantages in achieving this.
- You can reuse functions in other programs because they stand on their own, not depending on the rest of the system. Most functional programs share common functions, several of which we'll be considering in this book.
- Functional code is free from side effects, which means you can understand the objective of a function by studying it without having to consider the rest of the program.

Finally, once you get used to the FP style of programming, code becomes more understandable and easier to extend. So it seems that all five characteristics can be achieved with FP!



For a well-balanced look at the reasons to use FP, I'd suggest reading *Why Functional Programming Matters*, by John Hughes; it's available online at www.cs.kent.ac.uk/people/staff/dat/miranda/whyfp90.pdf. It's not geared towards JavaScript, but the arguments are easily understandable, anyway.

Not all is gold

However, let's strive for a bit of balance. Using FP isn't a silver bullet that will automatically make your code better. Some FP solutions are actually tricky, and there are developers who greatly enjoy writing code and then asking, *what does this do?* If you aren't careful, your code may become *write-only* and practically impossible to maintain; there goes understandable, extensible, and reusable out the door!

Another disadvantage is that you may find it harder to find FP-savvy developers. (Quick question: how many *functional programmers sought* job ads have you ever seen?) The vast majority of today's web code is written in imperative, non-functional ways, and most coders are used to that way of working. For some, having to switch gears and start writing programs in a different way may prove an unpassable barrier.

Finally, if you try to go fully functional, you may find yourself at odds with JavaScript, and simple tasks may become hard to do. As we said at the beginning, we'll opt for *sorta FP*, so we won't be drastically rejecting any language features that aren't 100% functional. After all, we want to use FP to simplify our coding, not to make it more complex!

So, while I'll strive to show you the advantages of going functional in your code, as with any change, there will always be some difficulties. However, I'm fully convinced that you'll be able to surmount them and that your organization will develop better code by applying FP. Dare to change! So, given that you accept that FP may apply to your own problems, let's now consider the other question, can we use JavaScript in a functional way and is it appropriate?

Is JavaScript functional?

At about this time, there is another important question that you should be asking: *Is JavaScript a functional language?* Usually, when thinking about FP, the list of languages that are mentioned does not include JavaScript, but does include less common options, such as Clojure, Erlang, Haskell, and Scala; however, there is no precise definition for FP languages or a precise set of features that such languages should include. The main point is that you can consider a language to be functional if it supports the common programming style associated with FP. Let's start by learning about why we would want to use JavaScript at all and how the language has evolved to its current version, and then see some of the key features that we'll be using to work in a functional way.

JavaScript as a tool

What is JavaScript? If you consider **popularity indices**, such as the ones at www.tiobe.com/tiobe-index/ or <http://pypl.github.io/PYPL.html>, you'll find that JavaScript is consistently in the top ten most popular languages. From a more academic point of view, the language is sort of a mixture, borrowing features from several different languages. Several libraries helped the growth of the language by providing features that weren't so easily available, such as classes and inheritance (today's version of the language does support classes, but that was not the case not too long ago), that otherwise had to be achieved by doing some **prototype** tricks.



The name *JavaScript* was chosen to take advantage of the popularity of Java—just as a marketing ploy! Its first name was *Mocha*, then *LiveScript*, and only then, *JavaScript*.

JavaScript has grown to be incredibly powerful. But, as with all power tools, it gives you a way to not only produce great solutions, but also to do great harm. FP could be considered as a way to reduce or leave aside some of the worst parts of the language and focus on working in a safer, better way; however, due to the immense amount of existing JavaScript code, you cannot expect it to facilitate large reworkings of the language that would cause most sites to fail. You must learn to live with the good and the bad, and simply avoid the latter parts.

In addition, the language has a broad variety of available libraries that complete or extend the language in many ways. In this book, we'll be focusing on using JavaScript on its own, but we will make references to existing, available code.

If we ask whether JavaScript is actually functional, the answer will be, once again, *sorta*. It can be seen as functional because of several features, such as first-class functions, anonymous functions, recursion, and closures—we'll get back to this later. On the other hand, it also has plenty of *non-FP* aspects, such as side effects (**impurity**), mutable objects, and practical limits to recursion. So, when programming in a functional way, we'll be taking advantage of all the relevant, appropriate language features, and we'll try to minimize the problems caused by the more conventional parts of the language. In this sense, JavaScript will or won't be functional, depending on *your* programming style!

If you want to use FP, you should decide which language to use; however, opting for fully functional languages may not be so wise. Today, developing code isn't as simple as just using a language; you will surely require frameworks, libraries, and other sundry tools. If we can take advantage of all the provided tools but at the same time introduce FP ways of working in our code, we'll be getting the best of both worlds, never mind whether JavaScript is functional!

Going functional with JavaScript

JavaScript has evolved through the years, and the version we'll be using is (informally) called JS10, and (formally) ECMAScript 2019, usually shortened to ES2019 or ES10; this version was finalized in June 2019. The previous versions were as follows:

- ECMAScript 1, June 1997
- ECMAScript 2, June 1998, which was basically the same as the previous version
- ECMAScript 3, December 1999, with several new functionalities
- ECMAScript 5, December 2009 (and no, there never was an ECMAScript 4, because it was abandoned)
- ECMAScript 5.1, June 2011
- ECMAScript 6 (or ES6; later renamed ES2015), June 2015
- ECMAScript 7 (also ES7, or ES2016), June 2016
- ECMAScript 8 (ES8 or ES2017), June 2017
- ECMAScript 9 (ES9 or ES2018), June 2018



ECMA originally stood for **European Computer Manufacturers Association**, but nowadays the name isn't considered an acronym anymore. The organization is responsible for more standards other than JavaScript, including JSON, C#, Dart, and others. For more details, go to its site at www.ecma-international.org/.

You can read the standard language specification at www.ecma-international.org/ecma-262/7.0/. Whenever we refer to JavaScript in the text without further specification, ES10 (ES2019) is what is being referred to; however, in terms of the language features that are used in the book, if you were just to use ES2015, then you'd mostly have no problems with this book.

No browsers fully implement ES10; most provide an older version, JavaScript 5 (from 2009), with an (always growing) smattering of features from ES6 up to ES10. This will prove to be a problem, but fortunately, a solvable one; we'll get to this shortly. We'll be using ES10 throughout the book.



In fact, there are only a few differences between ES2016 and ES2015, such as the `Array.prototype.includes` method and the exponentiation operator, `**`. There are more differences between ES2017 and ES2016—such as `async` and `await`, some string padding functions, and more—but they won't impact our code. We will also be looking at alternatives for even more modern additions, such as `flatMap()`, in later chapters.

As we are going to work with JavaScript, let's start by considering its most important features that pertain to our FP goals.

Key features of JavaScript

JavaScript isn't a purely functional language, but it has all the features that we need for it to work as if it were. The main features of the language that we will be using are as follows:

- Functions as first-class objects
- Recursion
- Arrow functions
- Closures
- Spread

Let's see some examples of each one and find out why they will be useful to us. Keep in mind, though, that there are more features of JavaScript that we will be using; the upcoming sections just highlight the most important features in terms of what we will be using for FP.

Functions as first-class objects

Saying that functions are **first-class objects** (also called **first-class citizens**) means that you can do everything with functions that you can do with other objects. For example, you can store a function in a variable, you can pass it to a function, you can print it out, and so on. This is really the key to doing FP; we will often be passing functions as parameters (to other functions) or returning a function as the result of a function call.

If you have been doing async Ajax calls, then you have already been using this feature: a **callback** is a function that will be called after the Ajax call finishes and is passed as a parameter. Using jQuery, you could write something like the following:

```
$.get("some/url", someData, function(result, status) {  
    // check status, and do something  
    // with the result  
});
```

The `$.get()` function receives a callback function as a parameter and calls it after the result is obtained.



This is better solved, in a more modern way, by using promises or `async/await`, but for the sake of our example, the older way is enough. We'll be getting back to promises, though, in the section called *Building better containers* in Chapter 12, *Building Better Containers – Functional Data Types*, when we discuss monads; in particular, see the section called *Unexpected Monads - Promises*.

Since functions can be stored in variables, you could also write something like the following. Pay attention to how we use the `doSomething` variable in the `$.get(...)` call:

```
var doSomething = function(result, status) {  
    // check status, and do something  
    // with the result  
};  
  
$.get("some/url", someData, doSomething);
```

We'll be seeing more examples of this in Chapter 6, *Producing Functions – Higher-Order Functions*.

Recursion

Recursion is the most potent tool for developing algorithms and a great aid for solving large classes of problems. The idea is that a function can at a certain point call itself, and when *that* call is done, continue working with whatever result it has received. This is usually quite helpful for certain classes of problems or definitions. The most often quoted example is the factorial function (the factorial of n is written as $n!$) as defined for nonnegative integer values:

- If n is 0, then $n!=1$
- If n is greater than 0, then $n! = n * (n-1)!$



The value of $n!$ is the number of ways that you can order n different elements in a row. For example, if you want to place five books in line, you can pick any of the five for the first place, and then order the other four in every possible way, so $5! = 5 \cdot 4!$. If you continue to work this example, you'll get $5! = 5 \cdot 4 \cdot 3 \cdot 2 \cdot 1 = 120$, so $n!$ is the product of all numbers up to n .

This can be immediately turned into code:

```
function fact(n) {
  if (n === 0) {
    return 1;

    } else {
    return n * fact(n - 1);
    }
}

console.log(fact(5)); // 120
```

Recursion will be a great aid for the design of algorithms. By using recursion, you could do without any `while` or `for` loops—not that we *want* to do that, but it's interesting that we *can*! We'll be devoting the entirety of [Chapter 9, Designing Functions – Recursion](#), to designing algorithms and writing functions recursively.

Closures

Closures are a way to implement data hiding (with private variables), which leads to modules and other nice features. The key concept of closures is that when you define a function, it can refer to not only its own local variables but also to everything outside of the context of the function. We can write a counting function that will keep its own count by means of a closure:

```
function newCounter() {
  let count = 0;
  return function() {
    count++;
    return count;
  };
}

const nc = newCounter();
console.log(nc()); // 1
console.log(nc()); // 2
console.log(nc()); // 3
```


Even after `newCounter()` exits, the inner function still has access to `count`, but that variable is not accessible to any other parts of your code.



This isn't a very good example of FP— a function (`nc()`, in this case) isn't expected to return different results when called with the same parameters!

We'll find several uses for closures, such as **memoization** (see Chapter 4, *Behaving Properly – Pure Functions*, and Chapter 6, *Producing Functions – Higher-Order Functions*) and the **module** pattern (see Chapter 3, *Starting out with Functions – A Core Concept*, and Chapter 11, *Implementing Design Patterns – The Functional Way*), among others.

Arrow functions

Arrow functions are just a shorter, more succinct way of creating an (unnamed) function. Arrow functions can be used almost everywhere a classical function can be used, except that they cannot be used as constructors. The syntax is either `(parameter, anotherparameter, ...etc) => { statements }` or `(parameter, anotherparameter, ...etc) => expression`. The first allows you to write as much code as you want, and the second is short for `{ return expression }`. We could rewrite our earlier Ajax example as follows:

```
$.get("some/url", data, (result, status) => {
  // check status, and do something
  // with the result
});
```

A new version of the factorial code could be like the following code:

```
const fact2 = n => {
  if (n === 0) {
    return 1;
  } else {
    return n * fact2(n - 1);
  }
};
console.log(fact2(5)); // also 120
```



Arrow functions are usually called **anonymous** functions because of their lack of a name. If you need to refer to an arrow function, you'll have to assign it to a variable or object attribute, as we did here; otherwise, you won't be able to use it. We'll learn more about this in the section called *Arrow functions* in Chapter 3, *Starting out with Functions - A Core Concept*.

You would probably write the latter as a one-liner—can you see the equivalence to our earlier code? Using a ternary operator in lieu of an `if` is quite common:

```
const fact3 = n => (n === 0 ? 1 : n * fact3(n - 1));  
  
console.log(fact3(5)); // again 120
```

With this shorter form, you don't have to write `return`—it's implied.



In lambda calculus, a function such as `x => 2*x` would be represented as $\lambda x.2*x$. Although there are syntactical differences, the definitions are analogous. Functions with more parameters are a bit more complicated; `(x,y)=>x+y` would be expressed as $\lambda x.\lambda y.x+y$. We'll learn more about this in the section called *Lambdas and functions*, in Chapter 3, *Starting out with Functions - A Core Concept*, and in the section called *Currying*, in Chapter 7, *Transforming Functions - Currying and Partial Application*.

There's one other small thing to bear in mind: when the arrow function has a single parameter, you can omit the parentheses around it. I usually prefer leaving them, but I've applied a JS beautifier, *Prettier*, to the code, which removes them. It's really up to you whether to include them or not! (For more on this tool, check out <https://github.com/prettier/prettier>.) By the way, my options for formatting were `--print-width 75 --tab-width 2 --no-bracket-spacing`.

Spread

The spread operator (see https://developer.mozilla.org/en/docs/Web/JavaScript/Reference/Operators/Spread_operator) lets you expand an expression in places where you would otherwise require multiple arguments, elements, or variables. For example, you can replace arguments in a function call, as shown in the following code:

```
const x = [1, 2, 3];  
  
function sum3(a, b, c) {  
  return a + b + c;  
}
```

```
}  
  
const y = sum3(...x); // equivalent to sum3(1,2,3)  
console.log(y); // 6
```

You can also create or join arrays, as shown in the following code:

```
const f = [1, 2, 3];  
  
const g = [4, ...f, 5]; // [4,1,2,3,5]  
  
const h = [...f, ...g]; // [1,2,3,4,1,2,3,5]
```

It works with objects too:

```
const p = { some: 3, data: 5 };  
  
const q = { more: 8, ...p }; // { more:8, some:3, data:5 }
```

You can also use it to work with functions that expect separate parameters instead of an array. Common examples of this would be `Math.min()` and `Math.max()`:

```
const numbers = [2, 2, 9, 6, 0, 1, 2, 4, 5, 6];  
const minA = Math.min(...numbers); // 0  
  
const maxArray = arr => Math.max(...arr);  
const maxA = maxArray(numbers); // 9
```

You can also write the following equality since the `.apply()` method requires an array of arguments, but `.call()` expects individual arguments:

```
someFn.apply(thisArg, someArray) === someFn.call(thisArg, ...someArray);
```



If you have problems remembering what arguments are required by `.apply()` and `.call()`, this mnemonic may help: *A is for an array, and C is for a comma*. See https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Function/apply and https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Function/call for more information.

Using the spread operator helps write a shorter, more concise code, and we will be taking advantage of it. We have seen all of the most important JavaScript features that we will be using. Let's round off the chapter by looking at some tools that we'll be working with.

How do we work with JavaScript?

This is all well and good, but as we mentioned before, it so happens that the JavaScript version available almost everywhere isn't ES10, but rather the earlier JS5. An exception to this is Node.js. It is based on Chrome's v8 high-performance JavaScript engine, which already has several ES10 features available. Nonetheless, as of today, ES10 coverage isn't 100% complete, and there are features that you will miss. (Check out <https://nodejs.org/en/docs/es6/> for more on Node.js and v8.) This will surely change in the future, as Internet Explorer will fade away, and the newest Microsoft's browser will share Chrome's engine, but for the time being, we must still deal with older, less powerful engines.

So what can you do if you want to code using the latest version, but the available one is an earlier, poorer one? Or what happens if most of your users are using older browsers, which don't support the fancy features you're keen on using? Let's see some solutions for this.



If you want to be sure of your choices before using any given new feature, check out the compatibility table at <https://kangax.github.io/compat-table/es6/> (see Figure 1.1). For Node.js specifically, check out <http://node.green/>.

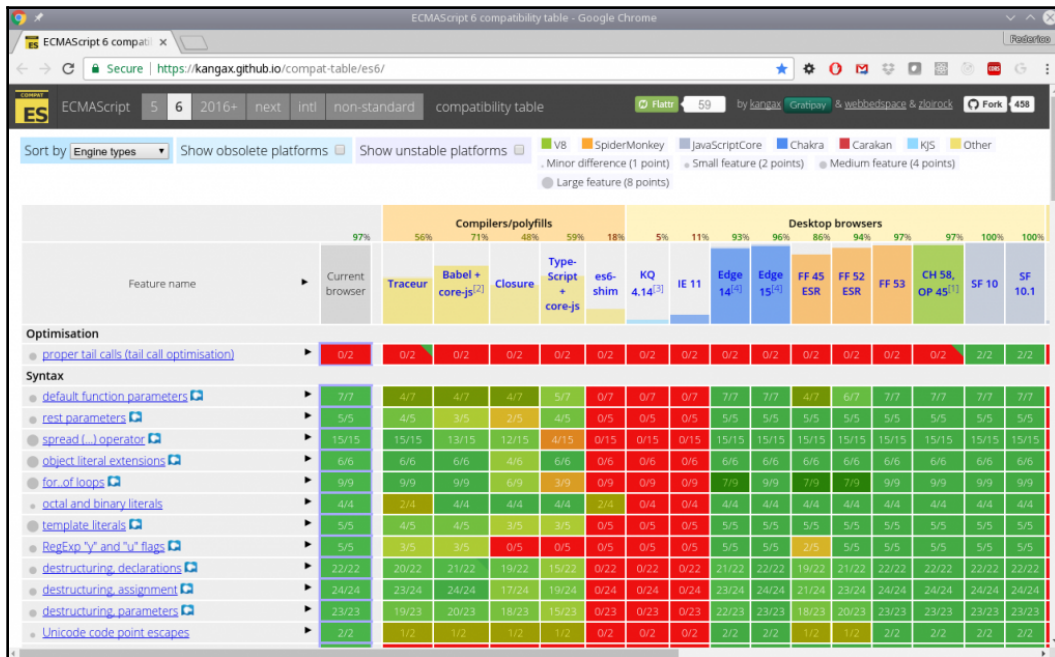


Figure 1.1. - The latest JavaScript features may not be widely and fully supported, so you'll have to check before using them.

Using transpilers

In order to get out of this availability and compatibility problem, there are a couple of **transpilers** that you can use. Transpilers take your original ES10 code, which might use the most modern JavaScript features, and transforms it into equivalent JS5 code. It's a source-to-source transformation, instead of a source-to-object code that would be used in compilation. You can code using advanced ES10 features, but the user's browsers will receive JS5 code. A transpiler will also let you keep up with upcoming versions of the language, despite the time needed by browsers to adopt new standards across desktop and mobile devices.



If you wonder where the word transpiler came from, it is a portmanteau of *translate* and *compiler*. There are many such combinations in technological speak: *email* (*electronic* and *mail*), *emoticon* (*emotion* and *icon*), *malware* (*malicious* and *software*), or *alphanumeric* (*alphabetic* and *numeric*), and many more.

The most common transpilers for JavaScript are Babel (at <https://babeljs.io/>) and Traceur (at <https://github.com/google/traceur-compiler>). With tools such as npm or webpack, it's fairly easy to configure things so that your code will get automatically transpiled and provided to end-users. You can also carry out transpilation online; see *Figure 1.2* for an example of this using Babel's online environment:

```
1 const fib = n => {
2   if (n < 2) {
3     return n;
4   } else {
5     return fib(n - 2) + fib(n - 1);
6   }
7 };
8
9 console.log(fib(10));
10
```

```
1 "use strict";
2
3 var fib = function fib(n) {
4   if (n < 2) {
5     return n;
6   } else {
7     return fib(n - 2) + fib(n - 1);
8   }
9 };
10
11 console.log(fib(10));
```

Figure 1.2 - The Babel transpiler converts ES10 code into compatible JS5 code

If you prefer Traceur, you can use its tool at <https://google.github.io/traceur-compiler/demo/repl.html#> instead, but you'll have to open a developer console to see the results of your running code (see *Figure 1.3* for an example of transpiled code). Select the **EXPERIMENTAL** option to fully enable ES10 support:

The screenshot shows the Traceur Transcoding Demo interface. The 'Source' tab is active, displaying the following code:

```

1 const fib = (n) => {
2   if (n<=1) {
3     return n;
4   } else {
5     return fib(n-1)+fib(n-2);
6   }
7 }
8
9 console.log(fib(7));
10 console.log(fib(10));
11
12

```

The transpiled code on the right is:

```

1 $traceurRuntime.ModuleStore.getAnonymousModule(
2   "use strict";
3   var fib = function(n) {
4     if (n <= 1) {
5       return n;
6     } else {
7       return fib(n - 1) + fib(n - 2);
8     }
9   };
10  console.log(fib(7));
11  console.log(fib(10));
12  return {};
13 });
14 // # sourceMappingURL=traceured.js
15

```

The Chrome DevTools console shows the following output:

```

13
55

```

The console also shows the source URLs for the log entries: `traceured.js:10` and `traceured.js:11`.

Figure 1.3 - The Traceur transpiler is an equally valid alternative for ES10-to-JS5 translation



Using transpilers is also a great way to learn new language features. Just type in some code on the left and see the equivalent code on the right. Alternatively, you can use the **command-line interface (CLI)** tools to transpile a source file and then inspect the produced output.

There's a final possibility that you may want to consider: instead of JavaScript, opt for Microsoft's TypeScript (at <http://www.typescriptlang.org/>), a superset of the language that is itself compiled to JS5. The main advantage of TypeScript is the ability to add (optional) static type checks to JavaScript, which helps detect certain programming errors at compile time. But beware: as with Babel or Traceur, not all of ES10 will be available.



You can also perform type checks without using TypeScript by using Facebook's Flow (see <https://flow.org/>).

If you opt to go with TypeScript, you can also test it online at their **playground** (see <http://www.typescriptlang.org/play/>). You can set options to be more or less strict with data type checks, and you can also run your code on the spot (see *Figure 1.4* for more details):

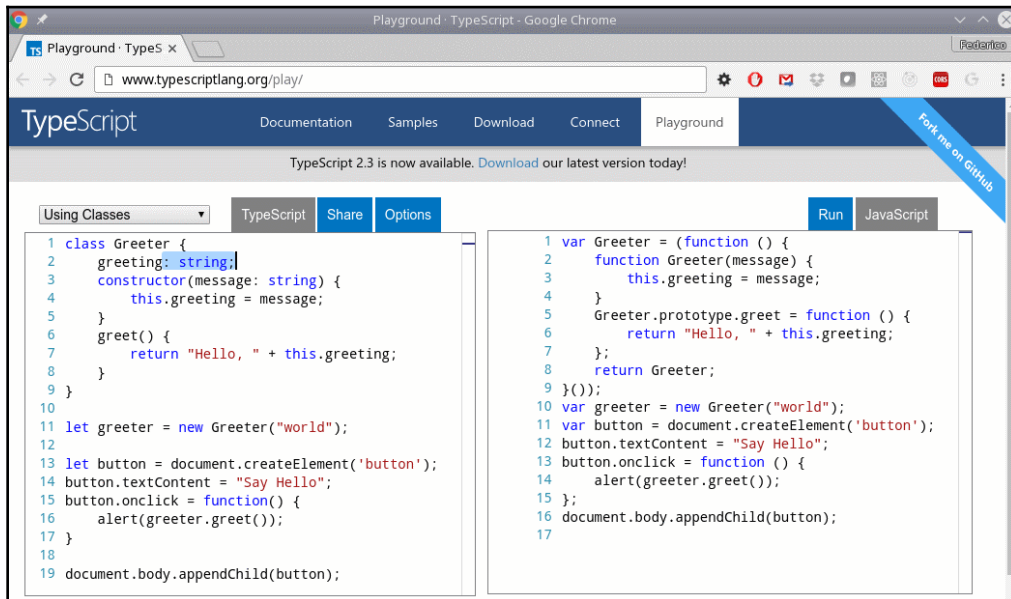


Figure 1.4 - TypeScript adds type-checking features for safer programming

By using TypeScript, you will be able to avoid common type-related mistakes. A positive trend is that most tools (frameworks, libraries, and so on) are slowly going in this direction, so work will be easier.

Working online

There are some more online tools that you can use to test out your JavaScript code. Check out JSFiddle (at <https://jsfiddle.net/>), CodePen (at <https://codepen.io/>), and JSBin (at <http://jsbin.com/>), among others. You may have to specify whether to use Babel or Traceur; otherwise, newer language features will be rejected. You can see an example of JSFiddle in *Figure 1.5*:

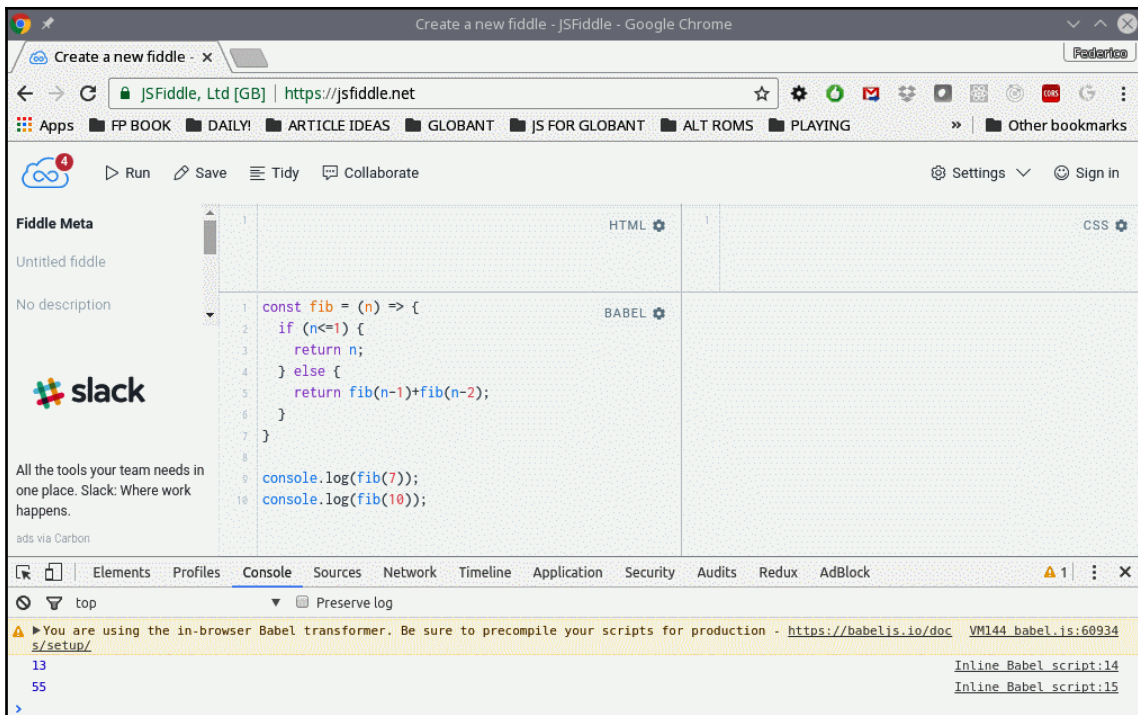


Figure 1.5 - JSFiddle lets you try out modern JavaScript code (plus HTML and CSS) without requiring any other tools

Using these tools provides a very quick way to try out code or do small experiments—and I can truly vouch for this since I've tested much of the code in the book in this way!

Testing

We will also touch on testing, which is, after all, one of FP's main advantages. For this, we will be using Jasmine (<https://jasmine.github.io/>), though we could also opt for Mocha (<http://mochajs.org/>).

You can run Jasmine test suites with a runner, such as Karma (<https://karma-runner.github.io>), but I opted for standalone tests; see <https://github.com/jasmine/jasmine#installation> for details.

Summary

In this chapter, we have seen the basics of FP, a bit of its history, its advantages (and also some possible disadvantages, to be fair), why we can apply it in JavaScript (which isn't usually considered a functional language), and what tools we'll need in order to go through the rest of this book.

In [Chapter 2, Thinking Functionally - A First Example](#), we'll go over an example of a simple problem, look at it in *common* ways, and end by solving it in a functional manner and analyzing the advantages of our method.

Questions

1.1. Classes as first-class objects: We learned that functions are first-class objects, but did you know that *classes* also are? (Though, of course, speaking of classes as *objects* does sound weird.) Look at the following example and see what makes it tick! Be careful: there's some purposefully weird code in it:

```
const makeSaluteClass = term =>
  class {
    constructor(x) {
      this.x = x;
    }

    salute(y) {
      console.log(`${this.x} says "${term}" to ${y}`);
    }
  };

const Spanish = makeSaluteClass("HOLA");
new Spanish("ALFA").salute("BETA");
// ALFA says "HOLA" to BETA

new (makeSaluteClass("HELLO"))("GAMMA").salute("DELTA");
// GAMMA says "HELLO" to DELTA

const fullSalute = (c, x, y) => new c(x).salute(y);
const French = makeSaluteClass("BON JOUR");
fullSalute(French, "EPSILON", "ZETA");
// EPSILON says "BON JOUR" to ZETA
```

1.2. **Factorial errors:** Factorials, as we defined them, should only be calculated for non-negative integers; however, the function that we wrote in the *Recursion* section doesn't verify whether its argument is valid. Can you add the necessary checks? Try to avoid repeated, redundant tests!

1.3. **Climbing factorial:** Our implementation of a factorial starts by multiplying by n , then by $n-1$, then $n-2$, and so on in what we could call a *downward fashion*. Can you write a new version of the factorial function that will loop *upwards*?

1.4. **Code squeezing:** Not that it's a goal in itself, but by using arrow functions and some other JavaScript features, you can shorten `newCounter()` to half its length. Can you see how?

2

Thinking Functionally - A First Example

In Chapter 1, *Becoming Functional – Several Questions*, we went over what FP is, mentioned some advantages of applying it, and listed some tools we'd be needing in JavaScript, but let's now leave theory behind, and start by considering a simple problem and how to solve it in a functional way.

In this chapter, we will do the following:

- Look at a simple, common, e-commerce related problem
- Consider several usual ways to solve it, with their associated defects
- Find a way to solve the problem by looking at it functionally
- Devise a higher-order solution that can be applied to other problems
- Work out how to carry out unit testing for functional solutions

In future chapters, we'll be coming back to some of the topics listed here, so we won't be going into too much detail. We'll just show how FP can give a different outlook for our problem and leave further details for later. After working through this chapter, you will have had a first look at a common problem and at a way of solving it by thinking functionally, as a prelude for the rest of this book.

Our problem – doing something only once

Let's consider a simple but common situation. You have developed an e-commerce site; the user can fill their shopping cart, and in the end, they must click on a **Bill me** button so their credit card will be charged. However, the user shouldn't click twice (or more) or they will be billed several times.

The HTML part of your application might have something like this somewhere:

```
<button id="billButton" onclick="billTheUser(some, sales, data)">Bill  
me</button>
```

And, among the scripts, you'd have something similar to the following code:

```
function billTheUser(some, sales, data) {  
  window.alert("Billing the user...");  
  // actually bill the user  
}
```



Assigning the events handler directly in HTML, the way I did it, isn't recommended. Rather, unobtrusively, you should assign the handler through code. So, *do as I say, not as I do!*

This is a very bare-bones explanation of the problem and your web page, but it's enough for our purposes. Let's now get to thinking about ways of avoiding repeated clicks on that button. *How can we manage to avoid the user clicking more than once?* That's an interesting problem, with several possible solutions—let's get started by looking at bad ones!

How many ways can you think of to solve our problem? Let's go over several solutions and analyze their quality.

Solution 1 – hoping for the best!

How can we solve the problem? The first *solution* may seem like a joke: do nothing, tell the user *not* to click twice, and hope for the best! Your page might look like *Figure 2.1*:

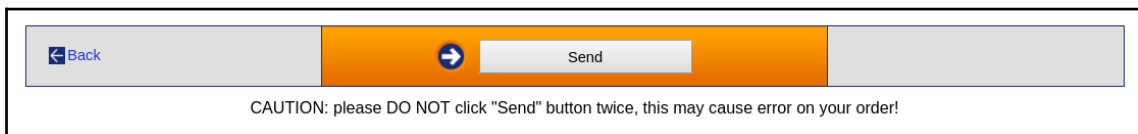


Figure 2.1: An actual screenshot of a page, just warning you against clicking more than once

This is a way to weasel out of the problem; I've seen several websites that just warn the user about the risks of clicking more than once (see *Figure 2.1*) and actually do nothing to prevent the situation: *The user got billed twice? We warned them... it's their fault!*

Your solution might simply look like the following code:

```
<button id="billButton" onclick="billTheUser(some, sales, data)">Bill  
me</button>  
<b>WARNING: PRESS ONLY ONCE, DO NOT PRESS AGAIN!!</b>
```

Okay, so this isn't actually a solution; let's move on to more serious proposals.

Solution 2 – using a global flag

The solution most people would probably think of first is using some global variable to record whether the user has already clicked on the button. You'd define a flag named something like `clicked`, initialized with `false`. When the user clicks on the button, if `clicked` was `false`, you'd change it to `true` and execute the function; otherwise, you wouldn't do anything at all. See all of this in the following code:

```
let clicked = false;  
.br/>.br/>function billTheUser(some, sales, data) {  
  if (!clicked) {  
    clicked = true;  
    window.alert("Billing the user...");  
    // actually bill the user  
  }  
}
```



For more good reasons *not* to use global variables, read <http://wiki.c2.com/?GlobalVariablesAreBad>.

This obviously works, but it has several problems that must be addressed:

- You are using a global variable, and you could change its value by accident. Global variables aren't a good idea, neither in JavaScript nor in other languages.
- You must also remember to re-initialize it to `false` when the user starts buying again. If you don't, the user won't be able to make a second purchase because paying will have become impossible.
- You will have difficulties testing this code because it depends on external things (that is, the `clicked` variable).

So, this isn't a very good solution. Let's keep thinking!

Solution 3 – removing the handler

We may go for a lateral kind of solution, and instead of having the function avoid repeated clicks, we might just remove the possibility of clicking altogether. The following code does just that; the first thing that `billTheUser()` does is remove the `onclick` handler from the button, so no further calls will be possible:

```
function billTheUser(some, sales, data) {
  document.getElementById("billButton").onclick = null;
  window.alert("Billing the user...");
  // actually bill the user
}
```

This solution also has some problems:

- The code is tightly coupled to the button, so you won't be able to reuse it elsewhere.
- You must remember to reset the handler, otherwise, the user won't be able to make a second buy.
- Testing will also be harder because you'll have to provide some DOM elements.

We can enhance this solution a bit and avoid coupling the function to the button by providing the latter's ID as an extra argument in the call. (This idea can also be applied to some of the following solutions.) The HTML part would be as follows, and note the extra argument to `billTheUser()`:

```
<button
  id="billButton"
  onclick="billTheUser('billButton', some, sales, data)"
>
  Bill me
</button>;
```

We also have to change the called function, so it will use the received `buttonId` value to access the corresponding button:

```
function billTheUser(buttonId, some, sales, data) {
  document.getElementById(buttonId).onclick = null;
  window.alert("Billing the user...");
  // actually bill the user
}
```

This solution is somewhat better. But, in essence, we are still using a global element—not a variable, but the `onclick` value. So, despite the enhancement, this isn't a very good solution either. Let's move on.

Solution 4 – changing the handler

A variant to the previous solution would be not to remove the click function, but rather assign a new one instead. We are using functions as first-class objects here when we assign the `alreadyBilled()` function to the click event. The function warning the user that they have already clicked could be something as follows:

```
function alreadyBilled() {
  window.alert("Your billing process is running; don't click, please.");
}
```

Our `billTheUser()` function would then be like the following code—and note how instead of assigning `null` to the `onclick` handler as in the previous section, now the `alreadyBilled()` function is assigned:

```
function billTheUser(some, sales, data) {
  document.getElementById("billButton").onclick = alreadyBilled;
  window.alert("Billing the user...");
  // actually bill the user
}
```

There's a good point to this solution; if the user clicks a second time, they'll get a warning not to do that, but they won't be billed again. (From the point of view of the user experience, it's better.) However, this solution still has the very same objections as the previous one (code coupled to the button, needing to reset the handler, and harder testing), so we won't consider it quite good anyway.

Solution 5 – disabling the button

A similar idea here is instead of removing the event handler, we can disable the button so the user won't be able to click. You might have a function like the following code, which does exactly that by setting the `disabled` attribute of the button:

```
function billTheUser(some, sales, data) {
  document.getElementById("billButton").setAttribute("disabled", "true");
  window.alert("Billing the user...");
  // actually bill the user
}
```

This also works, but we still have objections as with the previous solutions (coupling the code to the button, needing to re-enable the button, and harder testing), so we don't like this solution either.

Solution 6 – redefining the handler

Another idea: instead of changing anything in the button, let's have the event handler change itself. The trick is in the second line; by assigning a new value to the `billTheUser` variable, we are actually dynamically changing what the function does! The first time you call the function, it will do its thing, but it will also change itself out of existence, by giving its name to a new function:

```
function billTheUser(some, sales, data) {
  billTheUser = function() {};
  window.alert("Billing the user...");
  // actually bill the user
}
```

There's a special trick in the solution. Functions are global, so the `billTheUser=...` line actually changes the function's inner workings. From that point on, `billTheUser` will be the new (null) function. This solution is still hard to test. Even worse, how would you restore the functionality of `billTheUser`, setting it back to its original objective?

Solution 7 – using a local flag

We can go back to the idea of using a flag, but instead of making it global (which was our main objection), we can use an **Immediately Invoked Function Expression (IIFE)**, which we'll see more on in Chapter 3, *Starting Out with Functions – A Core Concept*, and Chapter 11, *Implementing Design Patterns – The Functional Way*. With this, we can use a closure, so `clicked` will be local to the function, and not visible anywhere else:

```
var billTheUser = (clicked => {
  return (some, sales, data) => {
    if (!clicked) {
      clicked = true;
      window.alert("Billing the user...");
      // actually bill the user
    }
  };
})(false);
```



See how `clicked` gets its initial `false` value from the call at the end.

This solution is along the lines of the global variable solution but using a private, local variable is an enhancement. About the only drawback we could find is that you'll have to rework every function that needs to be called only once to work in this fashion (and, as we'll see in the following section, our FP solution is similar to it in some ways). Okay, it's not too hard to do, but don't forget the **Don't Repeat Yourself (DRY)** advice!

We have now gone through multiple ways of solving our *do something only once* problem—but as we've seen, they were not very good! Let's think about the problem in a functional way, and we'll get a more general solution.

A functional solution to our problem

Let's try to be more general; after all, requiring that some function or other be executed only once isn't that outlandish, and may be required elsewhere! Let's lay down some principles:

- The original function (the one that may be called only once) should do whatever it is expected to do and nothing else.
- We don't want to modify the original function in any way.
- We need to have a new function that will call the original one only once.
- We want a general solution that we can apply to any number of original functions.



The first principle listed previously is the *single responsibility principle* (the S in S.O.L.I.D.), which states that every function should be responsible for a single functionality. For more on S.O.L.I.D., check the article by *Uncle Bob* (Robert C. Martin, who wrote the five principles) at <http://butunclebob.com/ArticleS.UncleBob.PrinciplesOfOod>.

Can we do it? Yes, and we'll write a *higher-order function*, which we'll be able to apply to any function, to produce a new function that will work only once. Let's see how! We will introduce higher-order functions (to which we'll later dedicate [Chapter 6, Producing Functions – Higher-Order Functions](#)) and then we'll go about testing our functional solution, as well as providing some enhancements to it.

A higher-order solution

If we don't want to modify the original function, we'll create a higher-order function, which we'll (inspiredly!) name `once()`. This function will receive a function as a parameter and will return a new function, which will work only a single time. (As we mentioned before, we'll be seeing more of higher-order functions in Chapter 6, *Producing Functions – Higher-Order Functions*; in particular, see the *Doing things once, revisited* section.)



Underscore and Lodash already have a similar function, invoked as `_.once()`. Ramda also provides `R.once()`, and most FP libraries include similar functionality, so you wouldn't have to program it on your own.

Our `once()` function may seem imposing at first, but as you get accustomed to working in FP fashion, you'll get used to this sort of code and find it to be quite understandable:

```
const once = fn => {
  let done = false;
  return (...args) => {
    if (!done) {
      done = true;
      fn(...args);
    }
  };
};
```

Let's go over some of the finer points of this function:

- The first line shows that `once()` receives a function (`fn`) as its parameter.
- We are defining an internal, private `done` variable, by taking advantage of closure, as in *Solution 7*, previously. We opted *not* to call it `clicked`, as previously, because you don't necessarily need to click on a button to call the function, so we went for a more general term. Each time you apply `once()` to some function, a new, distinct `done` variable will be created and will be accessible only from the returned function.
- The `return (...args) => ...` line says that `once()` will return a function, with some (one or more, or possibly zero) parameters. Note that we are using the spread syntax we saw in Chapter 1, *Becoming Functional – Several Questions*. With older versions of JavaScript, you'd have to work with the `arguments` object; see <https://developer.mozilla.org/en/docs/Web/JavaScript/Reference/Functions/arguments> for more on that. The modern JavaScript way is simpler and shorter!

- We assign `done = true` before calling `fn()`, just in case that function throws an exception. Of course, if you don't want to disable the function unless it has successfully ended, then you could move the assignment just below the `fn()` call.
- After the setting is done, we finally call the original function. Note the use of the spread operator to pass along whatever parameters the original `fn()` had.

So, how would we use it? We don't even need to store the newly generated function in any place. We can simply write the `onClick` method, shown as follows:

```
<button id="billButton" onclick="once(billTheUser)(some, sales, data)">  
  Bill me  
</button>;
```

Pay close attention to the syntax! When the user clicks on the button, the function that gets called with the `(some, sales, data)` argument isn't `billTheUser()`, but rather the result of having called `once()` with `billTheUser` as a parameter. That result is the one that can be called only a single time.



Note that our `once()` function uses functions as first-class objects, arrow functions, closures, and the spread operator; back in [Chapter 1, *Becoming Functional – Several Questions*](#), we said we'd be needing those, so we're keeping our word! All we are missing here from that chapter is recursion, but as the Rolling Stones sang, *You Can't Always Get What You Want!*

We now have a functional way of getting a function to do its thing only once; how would we test it? Let's get into that topic now.

Testing the solution manually

We can run a simple test. Let's write a `squeak()` function that will, appropriately, squeak when called! The code is simple:

```
const squeak = a => console.log(a, " squeak!!");  
  
squeak("original"); // "original squeak!!"  
squeak("original"); // "original squeak!!"  
squeak("original"); // "original squeak!!"
```

If we apply `once()` to it, we get a new function that will squeak only once. See the highlighted line in the following code:

```
const squeakOnce = once(squeak);

squeakOnce("only once"); // "only once squeak!!"
squeakOnce("only once"); // no output
squeakOnce("only once"); // no output
```

Check out the results at CodePen or see *Figure 2.2*:

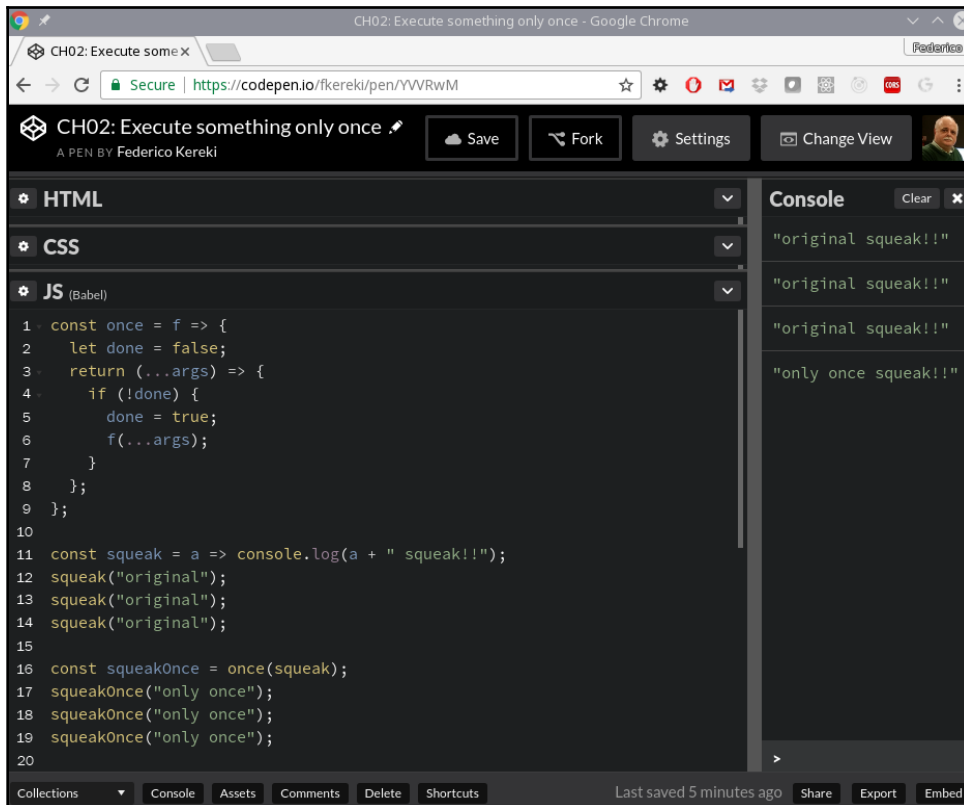


Figure 2.2: Testing our `once()` higher-order function

The previous steps showed us how we could test our `once()` function by hand, but the method we used is not exactly ideal. Let's see why, and how to do better, in the next section.

Testing the solution automatically

Running tests by hand is no good: it gets tiresome and boring and that leads, after a while, to not running the tests any longer. Let's do better and write some automatic tests with Jasmine. Following the instructions over at https://jasmine.github.io/pages/getting_started.html, I set up a standalone runner; the required HTML code, using Jasmine Spec Runner 2.6.1, is as follows:

```
<!DOCTYPE html>
<html>
<head>
  <meta charset="utf-8">
  <title>Jasmine Spec Runner v2.6.1</title>

  <link rel="shortcut icon" type="image/png"
        href="lib/jasmine-2.6.1/jasmine_favicon.png">
  <link rel="stylesheet" href="lib/jasmine-2.6.1/jasmine.css">

  <script src="lib/jasmine-2.6.1/jasmine.js"></script>
  <script src="lib/jasmine-2.6.1/jasmine-html.js"></script>
  <script src="lib/jasmine-2.6.1/boot.js"></script>

  <script src="src/once.js"></script>
  <script src="tests/once.test.1.js"></script>
</head>
<body>
</body>
</html>
```

The `src/once.js` file has the `once()` definition that we just saw, and `tests/once.test.js` has the actual suite of tests. The code for our tests is the following:

```
describe("once", () => {
  beforeEach(() => {
    window.myFn = () => {};
    spyOn(window, "myFn");
  });

  it("without 'once', a function always runs", () => {
    myFn();
    myFn();
    myFn();
    expect(myFn).toHaveBeenCalledTimes(3);
  });

  it("with 'once', a function runs one time", () => {
    window.onceFn = once(window.myFn);
```

```
spyOn(window, "onceFn").and.callThrough();
onceFn();
onceFn();
onceFn();
expect(onceFn).toHaveBeenCalledTimes(3);
expect(myFn).toHaveBeenCalledTimes(1);
});
});
```

There are several points to note here:

- To spy on a function, it must be associated with an object. (Alternatively, you can also directly create a spy using Jasmine's `createSpy()` method.) Global functions are associated with the window object, so `window.fn` is a way of saying that `fn` is actually global.
- When you spy on a function, Jasmine intercepts your calls and registers that the function was called, with which arguments, and how many times it was called. So, for all we care, `window.fn` could simply be `null` because it will never be executed.
- The first test only checks that if we call the function several times, it gets called that number of times. This is trivial, but if that didn't happen, we'd be doing something really wrong!
- In the second group of tests, we want to see that the `once()` function (`window.onceFn()`) gets called, but only once. So, we tell Jasmine to spy on `onceFn` but let calls pass through. Any calls to `fn()` will also get counted. In our case, as expected, despite calling `onceFn()` three times, `fn()` gets called only once.

We can see the results in *Figure 2.3*:

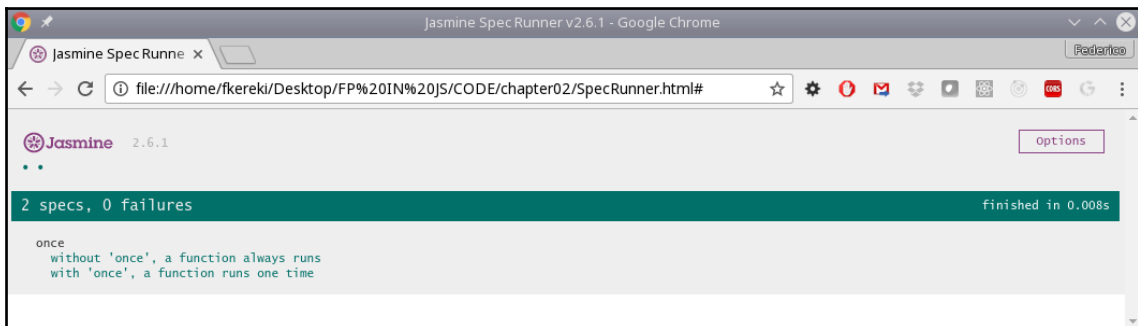


Figure 2.3: Running automatic tests on our function with Jasmine

Now we have seen not only how to test our functional solution by hand but also in an automatic way, so we are done with testing. Let's just finish by considering an even better solution, also achieved in a functional way.

Producing an even better solution

In one of the previous solutions, we mentioned that it would be a good idea to do something every time after the first click, and not silently ignore the user's clicks. We'll write a new higher-order function that takes a second parameter—a function to be called every time from the second call onward. Our new function will be called `onceAndAfter()` and can be written as follows:

```
const onceAndAfter = (f, g) => {
  let done = false;
  return (...args) => {
    if (!done) {
      done = true;
      f(...args);
    } else {
      g(...args);
    }
  }
};
```

We have ventured further in higher-order functions; `onceAndAfter()` takes *two* functions as parameters and produces a third one, which includes the other two within.



You could make `onceAndAfter()` more powerful by giving a default value for `g`, along the lines of `const onceAndAfter = (f, g = () => {})`, so if you didn't want to specify the second function, it would still work fine because it would call a *do-nothing* function, instead of causing an error.

We can do a quick-and-dirty test, along the same lines as we did earlier. Let's add a `creak()` creaking function to our previous `squeak()` one, and check out what happens if we apply `onceAndAfter()` to them. We can then get a `makeSound()` function that should `squeak()` once and `creak()` afterward:

```
const squeak = (x) => console.log(x, "squeak!!");
const creak = (x) => console.log(x, "creak!!");
const makeSound = onceAndAfter(squeak, creak);

makeSound("door"); // "door squeak!!"
makeSound("door"); // "door creak!!"
```

```
makeSound("door"); // "door creak!!"  
makeSound("door"); // "door creak!!"
```

Writing a test for this new function isn't hard, only a bit longer. We have to check which function was called and how many times:

```
describe("onceAndAfter", () => {  
  it("should call the first function once, and the other after", () => {  
    func1 = () => {};  
    spyOn(window, "func1");  
    func2 = () => {};  
    spyOn(window, "func2");  
    onceFn = onceAndAfter(func1, func2);  
  
    onceFn();  
    expect(func1).toHaveBeenCalledTimes(1);  
    expect(func2).toHaveBeenCalledTimes(0);  
  
    onceFn();  
    expect(func1).toHaveBeenCalledTimes(1);  
    expect(func2).toHaveBeenCalledTimes(1);  
  
    onceFn();  
    expect(func1).toHaveBeenCalledTimes(1);  
    expect(func2).toHaveBeenCalledTimes(2);  
  
    onceFn();  
    expect(func1).toHaveBeenCalledTimes(1);  
    expect(func2).toHaveBeenCalledTimes(3);  
  });  
});
```

Notice that we always check that `func1()` is called only once. Similarly, we check `func2()`; the count of calls starts at zero (the time that `func1()` is called), and from then on, it goes up by one on each call.

Summary

In this chapter, we've seen a common, simple problem, based on a real-life situation, and after analyzing several typical ways of solving that, we went for a *functional thinking* solution. We saw how to apply FP to our problem, and we found a more general higher-order solution that we could apply to similar problems with no further code changes. We saw how to write unit tests for our code to round out the development job.

Finally, we produced an even better solution (from the point of view of the user experience) and saw how to code it and how to unit test it. Now, you've started to get a grip on how to solve a problem functionally; next, in [Chapter 3, Starting Out with Functions – a Core Concept](#), we'll be delving more deeply into functions, which are at the core of all FP.

Questions

2.1. No extra variables: Our functional implementation required using an extra variable, `done`, to mark whether the function had already been called. Not that it matters, but could you make do without using any extra variables? Note that we aren't telling you *not* to use any variables, it's just a matter of not adding any new ones, such as `done`, and only as an exercise!

2.2. Alternating functions: In the spirit of our `onceAndAfter()` function, could you write an `alternator()` higher-order function that gets two functions as arguments and on each call, alternatively calls one and another? The expected behavior should be as in the following example:

```
let sayA = () => console.log("A");
let sayB = () => console.log("B");
let alt = alternator(sayA, sayB);

alt(); // A
alt(); // B
alt(); // A
alt(); // B
alt(); // A
alt(); // B
```

2.3. Everything has a limit! As an extension of `once()`, could you write a higher-order function, `thisManyTimes(fn, n)`, that would let you call the `fn()` function up to `n` times, but would afterward do nothing? To give an example, `once(fn)` and `thisManyTimes(fn, 1)` would produce functions that behave in exactly the same way.

3

Starting Out with Functions - A Core Concept

In [Chapter 2, *Thinking Functionally – A First Example*](#), we went over an example of **Functional Programming (FP)** thinking, but let's now look at the basics and review functions. In [Chapter 1, *Becoming Functional – Several Questions*](#), we mentioned that two important JavaScript features were functions: first-class objects and closures.

In this chapter, we'll cover several important topics:

- Functions in JavaScript, including how to define them, with a particular focus on arrow functions
- Currying and functions as first-class objects
- Several ways of using functions in an FP way

After all this content, you'll be up to date as to the generic and specific concepts relating to functions, which are, after all, at the core of FP!

All about functions

Let's get started with a short review of functions in JavaScript and their relationship to FP concepts. We can start something that we mainly mentioned in the *Functions as first-class objects* section in [Chapter 1, *Becoming Functional - Several Questions*](#), and in a couple of places in [Chapter 2, *Thinking Functionally - A First Example*](#), about functions as first-class objects, and then go on to several considerations about their usage in actual coding.

In particular, we'll be looking at the following:

- Some basic and very important concepts about lambda calculus, which is the theoretical basis for FP
- Arrow functions, which are the most direct translation of lambda calculus into JavaScript
- Using functions as first-class objects, a key concept in FP

Of lambdas and functions

In lambda calculus terms, a function can look like $\lambda x. 2 * x$. The understanding is that the variable after the λ character is the parameter for the function, and the expression after the dot is where you would replace whatever value is passed as an argument. Later in this chapter, we will see that this particular example could be written as `x => 2 * x` in JavaScript in arrow function form, which, as you can see, is very similar in form.



If you sometimes wonder about the difference between arguments and parameters, a mnemonic with some alliteration may help: *Parameters are Potential, Arguments are Actual*. Parameters are placeholders for potential values that will be passed, and arguments are the actual values passed to the function. In other words, when you define the function, you list its parameters, and when you call it, you provide arguments.

Applying a function means that you provide an actual argument to it, which is written in the usual way, by using parentheses. For example, $(\lambda x. 2 * x) (3)$ would be calculated as 6. What's the equivalent of these lambda functions in JavaScript? That's an interesting question! There are several ways of defining functions, and not all have the same meaning.



A good article that shows the many ways of defining functions, methods, and more is *The Many Faces of Functions in JavaScript* by Leo Balter and Rick Waldron, at <https://bocoup.com/blog/the-many-faces-of-functions-in-javascript>—give it a look!

In how many ways can you define a function in JavaScript? The answer is probably in more ways than you thought! At the very least, you could write the following:

- A named function declaration: `function first(...) {...};`
- An anonymous function expression: `var second = function(...) {...};`
- A named function expression: `var third = function someName(...) {...};`

- An immediately-invoked expression: `var fourth = (function() { ...; return function(...) {...}; })();`
- A function constructor: `var fifth = new Function(...);`
- An arrow function: `var sixth = (...) => {...};`

And, if you wanted, you could add object method declarations, since they actually imply functions as well, but the preceding list should be enough.



JavaScript also allows us to define generator functions (as in `function*(...) {...}`) that actually return a `Generator` object, and `async` functions that are really a mix of generators and promises. We won't be using these kinds of functions, but you can read more about them at https://developer.mozilla.org/en/docs/Web/JavaScript/Reference/Statements/function* and https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Statements/async_function—they can be useful in other contexts.

What's the difference between all these ways of defining functions, and why should we care? Let's go over them, one by one:

- The first definition, a standalone declaration starting with the `function` keyword, is probably the most used definition in JavaScript, and defines a function named `first` (that is, `first.name=="first"`). Because of *hoisting*, this function will be accessible everywhere in the scope where it's defined.



You can read more about hoisting at <https://developer.mozilla.org/en-US/docs/Glossary/Hoisting>. Keep in mind that it applies only to declarations and not to initializations.

- The second definition, which assigns a function to a variable, also produces a function, but an *anonymous* (that is, not named) one; however, many JavaScript engines are capable of deducing what the name should be, and will then set `second.name === "second"`. (Look at the following code, which shows a case where the anonymous function has no name assigned.) Since the assignment isn't hoisted, the function will only be accessible after the assignment has been executed. Also, you'd probably prefer defining the variable with `const` rather than `var`, because you wouldn't (*shouldn't*) be changing the function:

```
var second = function() {};  
console.log(second.name);  
// "second"
```

```
var myArray = new Array(3);
myArray[1] = function() {};
console.log(myArray[1].name);
// ""
```

- The third definition is the same as the second, except that the function now has its own name: `third.name === "someName"`.



The name of a function is relevant when you want to call it, and also if you plan to perform recursive calls; we'll come back to this in Chapter 9, *Designing Functions – Recursion*. If you just want a function for, say, a callback, you can use one without a name; however, note that named functions are more easily recognized in an error traceback, the kind of listing you get to use when you are trying to understand what happened, and which function called what.

- The fourth definition, with an immediately invoked expression, lets you use a closure. An inner function can use variables or other functions, defined in its outer function, in a totally private, encapsulated, way. Going back to the counter-making function that we saw in the *Closures* section of Chapter 1, *Becoming Functional – Several Questions*, we could write something like the following:

```
var myCounter = (function(initialValue = 0) {
    let count = initialValue;
    return function() {
        count++;
        return count;
    };
})(77);

myCounter(); // 78
myCounter(); // 79
myCounter(); // 80
```

Study the code carefully: the outer function receives an argument (77, in this case) that is used as the initial value of `count` (if no initial value is provided, we start at 0). The inner function can access `count` (because of the closure), but the variable cannot be accessed anywhere else. In all aspects, the returned function is a common function—the only difference is its access to private elements. This is also the basis of the *module* pattern.

- The fifth definition isn't safe, and you shouldn't use it! You pass the names of the arguments first, then the actual function body as a string, and the equivalent of `eval()` is used to create the function, which could allow many dangerous hacks, so don't do this! Just to whet your curiosity, let's look at an example of rewriting the very simple `sum3()` function we saw back in the *Spread* section of Chapter 1, *Becoming Functional - Several Questions*:

```
var sum3 = new Function("x", "y", "z", "var t = x+y+z; return t;");
sum3(4, 6, 7); // 17
```



This sort of definition is not only unsafe, but has some other quirks—they don't create closures with their creation contexts, and so they are always global. See https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Function for more on this, but remember that using this way of creating functions isn't a good idea!

- Finally, the last definition, which uses an arrow `=>` definition, is the most compact way to define a function, and the one we'll try to use whenever possible.

At this point, we have seen several ways of defining a function, but let's now focus on arrow functions, a style we'll be favoring in our coding for this book.

Arrow functions – the modern way

Even if the arrow functions work pretty much in the same way as the other functions, there are some important differences between them and the usual functions. Arrow functions can implicitly return a value even with no `return` statement present, the value of `this` is not bound, and there is no `arguments` object. Let's go over these three points.



There are some extra differences: arrow functions cannot be used as constructors, they do not have a `prototype` property, and they cannot be used as generators because they don't allow the `yield` keyword. For more details on these points, see https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Functions/Arrow_functions#No_binding_of_this.

In this section, we'll go into several JavaScript function-related topics, including:

- How to return different values
- How to handle problems with the value of `this`

- How to work with varying numbers of arguments
- An important concept, *currying*, for which we'll find many usages in the rest of the book

Returning values

In the lambda coding style, functions only consist of a result. For the sake of brevity, the new arrow functions provide a syntax for this. When you write something like `(x, y, z) =>` followed by an expression, a `return` is implied. For instance, the following two functions actually do the same as the `sum3()` function that we showed previously:

```
const f1 = (x, y, z) => x + y + z;

const f2 = (x, y, z) => {
  return x + y + z;
};
```

If you want to `return` an object, then you must use parentheses; otherwise, JavaScript will assume that code follows.



A matter of style: when you define an arrow function with only one parameter, you can omit the parentheses around it. For consistency, I prefer to always include them. However, *Prettier*, the formatting tool I use (we mentioned it in [Chapter 1, *Becoming Functional - Several Questions*](#)) doesn't approve. Feel free to choose your style!

And a final note: lest you think this is a wildly improbable case, check out the *Questions* section later in this chapter for a very common scenario!

Handling the `this` value

A classic problem with JavaScript is the handling of `this`, whose value isn't always what you expect it to be. ES2015 solved this with arrow functions, which inherit the proper `this` value so that problems are avoided. Look at the following code for an example of the possible problems: by the time the timeout function is called, `this` will point to the global (window) variable instead of the new object, so you'll get `undefined` in the console:

```
function ShowItself1(identity) {
  this.identity = identity;
  setTimeout(function() {
    console.log(this.identity);
  }, 1000);
}
```

```
}  
  
var x = new ShowItself1("Functional");  
// after one second, undefined is displayed
```

There are two classic ways of solving this with old-fashioned JavaScript and the arrow way of working:

- One solution uses a closure and defines a local variable (usually named `that` or sometimes `self`), which will get the original value of `this`, so it won't be undefined.
- The second way uses `bind()`, so the timeout function will be bound to the correct value of `this`.
- The third, more modern way just uses an arrow function, so `this` gets the correct value (pointing to the object) without further ado.



We will also be using `bind()`. See the *Of lambdas and functions* section.

Let's see the three solutions in actual code. We use a closure for the first timeout, binding for the second, and an arrow function for the third:

```
function ShowItself2(identity) {  
  this.identity = identity;  
  let that = this;  
  setTimeout(function() {  
    console.log(that.identity);  
  }, 1000);  
  
  setTimeout(  
    function() {  
      console.log(this.identity);  
    }.bind(this),  
    2000  
  );  
  
  setTimeout(() => {  
    console.log(this.identity);  
  }, 3000);  
}  
  
var x = new ShowItself2("JavaScript");  
// after one second, "JavaScript"
```



```
// after another second, the same
// after yet another second, once again
```

If you run this code, you'll get `JavaScript` after one second, then again after another second, and yet a third time after another second; all three methods worked correctly, so whichever you pick really depends on which you like better.

Working with arguments

In Chapter 1, *Becoming Functional - Several Questions*, and Chapter 2, *Thinking Functionally - A First Example*, we saw some uses of the spread (`...`) operator. However, the most practical usage we'll be making of it has to do with working with arguments; we'll see some cases of this in Chapter 6, *Producing Functions - Higher-Order Functions*. Let's review our `once()` function:

```
const once = func => {
  let done = false;
  return (...args) => {
    if (!done) {
      done = true;
      func(...args);
    }
  };
};
```

Why are we writing `return (...args) =>` and then afterwards `func(...args)`? The answer has to do with the more modern way of handling a variable number (possibly zero) of arguments. How did you manage such kinds of code in older versions of JavaScript? The answer has to do with the `arguments` object (*not* an array!) that lets you access the actual arguments passed to the function.



For more on this, read <https://developer.mozilla.org/en/docs/Web/JavaScript/Reference/Functions/arguments>.

In JavaScript 5 and earlier, if we wanted a function to be able to process any number of arguments, we had to write code as follows:

```
function somethingElse() {
  // get arguments and do something
}

function listArguments() {
```

```
    console.log(arguments);
    var myArray = Array.prototype.slice.call(arguments);
    console.log(myArray);
    somethingElse.apply(null, myArray);
}

listArguments(22, 9, 60);
// (3) [22, 9, 60, callee: function, Symbol(Symbol.iterator): function]
// (3) [22, 9, 60]
```

The first log shows that `arguments` is actually an object; the second log corresponds to a simple array. Also, note the complicated way that is needed to call `somethingElse()`, which requires using `apply()`.

What would be the equivalent code in the latest JavaScript version? It is much shorter, and that's why we'll be seeing several examples of the usage of the spread operator in the book:

```
function listArguments2(...args) {
  console.log(args);
  somethingElse(...args);
}

listArguments2(12, 4, 56);
// (3) [12, 4, 56]
```

You should bear in mind the following points when looking at this code:

- By writing `listArguments2(...args)`, we immediately and clearly express that our new function receives several (possibly zero) arguments.
- You need not do anything to get an array. The console log shows that `args` is really an array.
- Writing `somethingElse(...args)` is much clearer than the alternative way that we had to use earlier (using `apply()`).

By the way, the `arguments` object is still available in the current version of JavaScript. If you want to create an array from it, you have two alternative ways of doing so, without having to resort to the `Array.prototype.slice.call` trick:

- Use the `from()` method and write `var myArray=Array.from(arguments)`.
- Write `let myArray=[...arguments]`, which shows yet another type of usage of the spread operator.

When we get to the topic of higher-order functions, writing functions that deal with other functions, with a possibly unknown number of parameters, will be commonplace.

JavaScript provides a much shorter way of doing this, and that's why you'll have to get accustomed to this usage. It's worth it!

One argument or many?

It's also possible to write functions that return functions, and in [Chapter 6, *Producing Functions - Higher-Order Functions*](#), we will be seeing more of this. For instance, in lambda calculus, you don't write functions with several parameters, but only one; you do this by using a technique called **currying** (why would you do this? Hold that thought; we'll come to this later).



Currying gets its name from Haskell Curry, who developed the concept. Note that there is an FP language that is named after him—*Haskell*; double recognition!

For instance, the function that we saw previously that sums three numbers would be written as follows:

```
const altSum3 = x => y => z => x + y + z;
```

Why did I change the function's name? Simply put, because this is *not* the same function as the one we saw previously. As is, though, it can be used to produce the very same results as our earlier function. That said, it differs in an important way. Let's look at how you would use it, say, to sum the numbers 1, 2, and 3:

```
altSum3(1)(2)(3); // 6
```



Test yourself before reading on, and think on this: what would have been returned if you had written `altSum3(1, 2, 3)` instead? Tip: it would not be a number! For the full answer, keep reading.

How does this work? Separating it into many calls can help; this would be the way the previous expression is actually calculated by the JavaScript interpreter:

```
let fn1 = altSum3(1);  
let fn2 = fn1(2);  
let fn3 = fn2(3);
```

Think functionally! The result of calling `altSum3(1)` is, according to the definition, a function, which, in virtue of a closure, resolves to be equivalent to the following:

```
let fn1 = y => z => 1 + y + z;
```

Our `altSum3()` function is meant to receive a single argument, not three! The result of this call, `fn1`, is also a single-argument function. When you use `fn1(2)`, the result is again a function, also with a single parameter, which is equivalent to the following:

```
let fn2 = z => 1 + 2 + z;
```

And when you calculate `fn2(3)`, a value is finally returned—great! As we said, the function performs the same kind of calculations as we saw earlier, but in an intrinsically different way.

You might think that currying is just a peculiar trick: who would want to only use single-argument functions? You'll see the reasons for this when we consider how to join functions together in Chapter 8, *Connecting Functions – Pipelining and Composition*, and Chapter 12, *Building Better Containers – Functional Data Types*, where it won't be feasible to pass more than one parameter from one step to the next.

Functions as objects

The concept of first-class objects means that functions can be created, assigned, changed, passed as parameters, and returned as a result of yet other functions in the very same way that you can with, say, numbers or strings. Let's start with its definition. Let's look at how you usually define a function:

```
function xyzzy(...) { ... }
```

This is (almost) equivalent to writing the following:

```
var xyzzy = function(...) { ... }
```

However, this is not true for **hoisting**. In this case, JavaScript moves all definitions to the top of the current scope, but does not move the assignments; so, with the first definition you can invoke `xyzzy(...)` from any place in your code, but with the second, you cannot invoke the function until the assignment has been executed.



See the parallel with the Colossal Cave Adventure game? Invoking `xyzzy(...)` anywhere won't always work! And, if you have never played that famous interactive fiction game, try it online—for example, at <http://www.web-adventures.org/cgi-bin/webfrotz?s=Adventure> OR <http://www.amc.com/shows/halt-and-catch-fire/colossal-cave-adventure/landing>.

The point that we want to make is that a function can be assigned to a variable and can also be reassigned if desired. In a similar vein, we can define functions on the spot, when they are needed. We can even do this without naming them: as with common expressions, if they are used only once, then you don't need to name them or store them in a variable.

A React-Redux reducer

We can see another example that involves assigning functions. As we mentioned earlier in this chapter, React-Redux works by dispatching actions that are processed by a reducer. Usually, the reducer includes code with a switch:

```
function doAction(state = initialState, action) {
  let newState = {};
  switch (action.type) {
    case "CREATE":
      // update state, generating newState,
      // depending on the action data
      // to create a new item
      return newState;

    case "DELETE":
      // update state, generating newState,
      // after deleting an item
      return newState;

    case "UPDATE":
      // update an item,
      // and generate an updated state
      return newState;

    default:
      return state;
  }
}
```



Providing `initialState` as a default value for `state` is a simple way of initializing the global state the first time around. Pay no attention to that default; it's not relevant for our example, but I included it just for the sake of completeness.

By taking advantage of the possibility of storing functions, we can build a **dispatch table** and simplify the preceding code. First, we initialize an object with the code for the functions for each action type.

Basically, we are just taking the preceding code and creating separate functions:

```
const dispatchTable = {
  CREATE: (state, action) => {
    // update state, generating newState,
    // depending on the action data
    // to create a new item
    return newState;
  },

  DELETE: (state, action) => {
    // update state, generating newState,
    // after deleting an item
    return newState;
  },

  UPDATE: (state, action) => {
    // update an item,
    // and generate an updated state
    return newState;
  }
};
```

We store the different functions that process each type of action as attributes in an object that will work as a dispatcher table. This object is created only once and is constant during the execution of the application. With it, we can now rewrite the action-processing code in a single line of code:

```
function doAction2(state = initialState, action) {
  return dispatchTable[action.type]
    ? dispatchTable[action.type](state, action)
    : state;
}
```

Let's analyze it: given the action, if `action.type` matches an attribute in the dispatching object, we execute the corresponding function taken from the object where it was stored. If there isn't a match, we just return the current state as Redux requires. This kind of code wouldn't be possible if we couldn't handle functions (storing and recalling them) as first-class objects.

An unnecessary mistake

There is, however, a common (though in fact, harmless) mistake that is usually made. You often see code like this:

```
fetch("some/remote/url").then(function(data) {
  processResult(data);
});
```

What does this code do? The idea is that a remote URL is fetched, and when the data arrives, a function is called—and this function itself calls `processResult` with `data` as an argument. That is to say, in the `then()` part, we want a function that, given `data`, calculates `processResult(data)`. But don't we already have such a function?



A small bit of theory: in lambda calculus terms, we are replacing $\lambda x.func\ x$ with simply `func`—this is called an **eta conversion**, or more specifically, an **eta reduction**. (If you were to do it the other way round, it would be an **eta abstraction**.) In our case, it could be considered a (very, very small!) optimization, but its main advantage is shorter, more compact code.

Basically, there is a rule that you can apply whenever you see something like the following:

```
function someFunction(someData) {
  return someOtherFunction(someData);
}
```

This rule states that you can replace code resembling the preceding code with just `someOtherFunction`. So, in our example, we can directly write what follows:

```
fetch("some/remote/url").then(processResult);
```

This code is exactly equivalent to the previous method that we looked at (although it is infinitesimally quicker, since you avoid one function call), but it is simpler to understand?

This programming style is called **pointfree** style or **tacit** style, and its main characteristic is that you never specify the arguments for each function application. An advantage of this way of coding is that it helps the writer (and the future readers of the code) think about the functions themselves and their meanings instead of working at a low level, passing data around and working with it. In the shorter version of the code, there are no extraneous or irrelevant details: if you understand what the called function does, then you understand the meaning of the complete piece of code. In our text, we'll often (but not necessarily always) work in this way.



Unix/Linux users may already be accustomed to this style, because they work in a similar way when they use pipes to pass the result of a command as an input to another. When you write something as `ls|grep doc|sort`, the output of `ls` is the input to `grep`, and the latter's output is the input to `sort`—but input arguments aren't written out anywhere; they are implied. We'll come back to this in the *Pointfree style* section of Chapter 8, *Connecting Functions - Pipelining and Composition*.

Working with methods

There is, however, a case that you should be aware of: what happens if you are calling an object's method? Look at the following code:

```
fetch("some/remote/url").then(function(data) {
  myObject.store(data);
});
```

If your original code had been something along the lines of the preceding code, then the seemingly obvious transformed code would fail:

```
fetch("some/remote/url").then(myObject.store);
```

Why? The reason is that in the original code, the called method is bound to an object (`myObject`), but in the modified code, it isn't bound and is just a free function. We can then fix it in a simple way by using `bind()`, as follows:

```
fetch("some/remote/url").then(myObject.store.bind(myObject));
```

This is a general solution. When dealing with a method, you cannot just assign it; you must use `bind()` so that the correct context will be available. Look at the following code:

```
function doSomeMethod(someData) {
  return someObject.someMethod(someData);
}
```

Following this rule, code like the preceding code should be converted to the following:

```
const doSomeMethod = someObject.someMethod.bind(someObject);
```



Read more on `bind()` at https://developer.mozilla.org/en/docs/Web/JavaScript/Reference/Global_objects/Function/bind.

This looks rather awkward, and not too elegant, but it's required so that the method will be associated with the correct object. We will see one application of this when we *promisify* functions in [Chapter 6, Producing Functions - Higher-Order Functions](#). Even if this code isn't so nice to look at, whenever you have to work with objects (and remember, we didn't say that we would be trying to aim for fully FP code, and did say that we would accept other constructs if they made things easier), you'll have to remember to bind methods before passing them as first-class objects in pointfree style.

Using functions in FP ways

There are several common coding patterns that actually take advantage of FP style, even if you weren't aware of it. In this section, we will go through them and look at the functional aspects of the code so that you can get more accustomed to this coding style.

Then, we'll look in detail at using functions in an FP way by considering several FP techniques, such as the following:

- Injection, which is needed for sorting different strategies, as well as other uses
- Callbacks and promises, introducing the continuation-passing style
- Polyfilling and stubbing
- Immediate invocation schemes

Injection – sorting it out

The first example of passing functions as parameters is provided by the `Array.prototype.sort()` method. If you have an array of strings and you want to sort it, you can just use the `sort()` method. For example, to alphabetically sort an array with the colors of the rainbow, we would write something like the following:

```
var colors = [
  "violet",
  "indigo",
  "blue",
  "green",
  "yellow",
  "orange",
  "red"
];
colors.sort();
console.log(colors);
// ["blue", "green", "indigo", "orange", "red", "violet", "yellow"]
```

Note that we didn't have to provide any parameters to the `sort()` call, but the array got sorted perfectly well. By default, this method sorts strings according to their ASCII internal representation. So, if you use this method to sort an array of numbers, it will fail, since it will decide that 20 must be between 100 and 3, because 100 precedes 20 (taken as strings!) and the latter precedes 3, so this needs fixing! The following code shows the problem:

```
var someNumbers = [3, 20, 100];
someNumbers.sort();

console.log(someNumbers);
// [100, 20, 3]
```

But let's forget numbers for a while and stick to sorting strings. We want to ask ourselves what would happen if we wanted to sort some Spanish words (`palabras`) but following the appropriate locale rules? We would be sorting strings, but the results wouldn't be correct:

```
var palabras = ["ñandú", "oasis", "mano", "natural", "mítico", "musical"];
palabras.sort();

console.log(palabras);
// ["mano", "musical", "mítico", "natural", "oasis", "ñandú"] -- wrong result!
```



For language or biology buffs, "ñandú" in English is *rhea*, a running bird somewhat similar to ostriches. There aren't many Spanish words beginning with ñ, and we happen to have these birds in my country, Uruguay, so that's the reason for the odd word!

Oops! In Spanish, ñ comes between *n* and *o*, but "ñandú" got sorted at the end. Also, "mítico" (in English, *mythical*; note the accented *í*) should appear between "mano" and "musical" because the tilde should be ignored. The appropriate way of solving this is by providing a comparison function to `sort()`. In this case, we can use the `localeCompare()` method as follows:

```
palabras.sort((a, b) => a.localeCompare(b, "es"));

console.log(palabras);
// ["mano", "mítico", "musical", "natural", "ñandú", "oasis"]
```

The `a.localeCompare(b, "es")` call compares the `a` and `b` strings and returns a negative value should `a` precede `b`, a positive value should `a` follow `b`, and 0 if `a` and `b` are the same—but according to Spanish ("es") ordering rules.

Now things are right! And the code could be made clearer by introducing a new function, `spanishComparison()`, to perform the required strings comparison:

```
const spanishComparison = (a, b) => a.localeCompare(b, "es");

palabras.sort(spanishComparison);
// sorts the palabras array according to Spanish rules:
// ["mano", "mítico", "musical", "natural", "ñandú", "oasis"]
```

In upcoming chapters, we will be discussing how FP lets you write code in a more declarative fashion, producing more understandable code, and this sort of small change helps: when readers of the code get to the `sort`, they will immediately deduce what is being done, even if the comment wasn't present.



This way of changing the way that the `sort()` function works by injecting different comparison functions is actually a case of the strategy design pattern. We'll be learning more about this in Chapter 11, *Implementing Design Patterns – the Functional Way*.

Providing a `sort` function as a parameter (in a very FP way!) can also help with several other problems, such as the following:

- `sort()` only works with strings. If you want to sort numbers (as we tried to do previously), you have to provide a function that will compare numerically. For example, you would write something like `myNumbers.sort((a,b) => a-b)`.
- If you want to sort objects by a given attribute, you will use a function that compares to it. For example, you could sort people by age with something along the lines of `myPeople.sort((a,b) => a.age - b.age)`.



For more on the `localeCompare()` possibilities, see https://developer.mozilla.org/en/docs/Web/JavaScript/Reference/Global_Objects/String/localeCompare. You can specify which locale rules to apply, in which order to place upper/lowercase letters, whether to ignore punctuation, and much more. But be careful: not all browsers may support the required extra parameters.

This is a simple example that you have probably used before, but it's an FP pattern, after all. Let's move on to an even more common usage of functions as parameters when you perform Ajax calls.

Callbacks, promises, and continuations

Probably the most used example of functions passed as first-class objects has to do with callbacks and promises. In Node, reading a file is accomplished asynchronously with something like the following code:

```
const fs = require("fs");

fs.readFile("someFile.txt", (err, data) => {
  if (err) {
    console.error(err); // or throw an error, or otherwise handle the
    problem
  } else {
    console.log(data.toString()); // do something with the data
  }
});
```

The `readFile()` function requires a callback—in this example an anonymous function—that will get called when the file reading operation is finished.

A better way is using promises; read more at https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Promise. With this, when performing an Ajax web service call using the more modern `fetch()` function, you could write something along the lines of the following code:

```
fetch("some/remote/url")
  .then(data => {
    // Do some work with the returned data
  })
  .catch(error => {
    // Process all errors here
  });
```



Note that if you had defined appropriate `processData(data)` and `processError(error)` functions, the code could have been shortened to `fetch("some/remote/url").then(processData).catch(processError)` along the lines that we saw previously.

Finally, you should also look into using `async/await`; read more about it at https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Statements/async_function and <https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/await>.

Continuation passing style

The preceding code, in which you call a function but also pass another function that is to be executed when the input/output operation is finished, can be considered a case of **continuation passing style (CPS)**. What is this technique of coding? One way of looking at it is by thinking about the question: *how would you program if using the `return` statement was forbidden?*

At first glance, this may appear to be an impossible situation. We can get out of our fix, however, by passing a callback to the called function, so that when that procedure is ready to return to the caller, instead of actually returning, it invokes the passed callback. By doing this, the callback provides the called function with the way to continue the process, hence the word *continuation*. We won't get into this now, but in Chapter 9, *Designing Functions - Recursion*, we will study it in depth. In particular, CPS will help us to avoid an important recursion restriction, as we'll see.

Working out how to use continuations is sometimes challenging, but always possible. An interesting advantage of this way of coding is that by specifying yourself how the process is going to continue, you can go beyond all the usual structures (`if`, `while`, `return`, and so on) and implement whatever mechanisms you want. This can be very useful in some kinds of problems where the process isn't necessarily linear. Of course, this can also lead to you inventing a kind of control structure that is far worse than the possible usage of `GOTO` statements that you might imagine! Figure 3.1 shows the dangers of that practice!

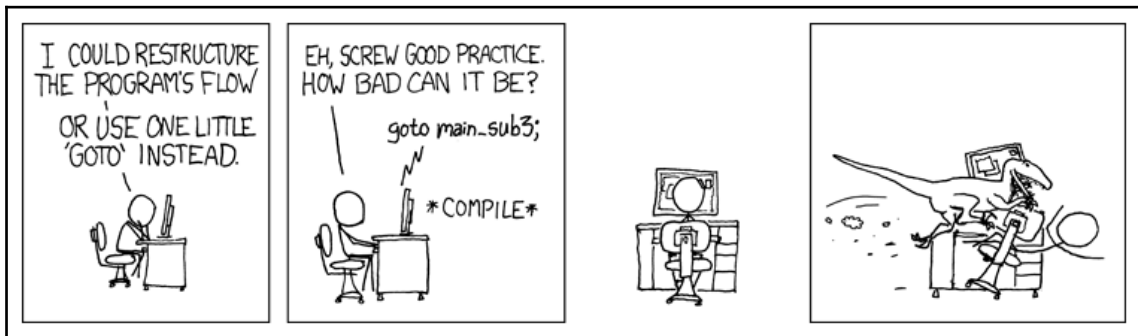


Figure 3.1: What's the worse that could happen if you start messing with the program flow?



This XKCD comic is available online at <https://xkcd.com/292/>

You are not limited to passing a single continuation. As with promises, you can provide two or more alternative callbacks. And this, by the way, can provide a solution to the problem of how you would work with exceptions. If we simply allowed a function to throw an error, it would be an implied return to the caller, and we don't want this. The way out of this is to provide an alternative callback (that is, a different continuation) to be used whenever an exception is thrown (in Chapter 12, *Building Better Containers - Functional Data Types*, we'll find another solution using monads):

```
function doSomething(a, b, c, normalContinuation, errorContinuation) {
  let r = 0;
  // ... do some calculations involving a, b, and c,
  // and store the result in r
  // if an error happens, invoke:
  // errorContinuation("description of the error")
  // otherwise, invoke:
  // normalContinuation(r)
}
```

Using CPS can even allow you to go beyond the control structures that JavaScript provides, but that would be beyond the objectives of this book, so I'll let you research that on your own!

Polyfills

Being able to assign functions dynamically (in the same way that you can assign different values to a variable) also allows you to work more efficiently when defining polyfills.

Detecting Ajax

Let's go back a bit in time to when Ajax started to appear. Given that different browsers implemented Ajax calls in distinct fashions, you would always have to code around these differences. The following code shows how you would go about implementing an Ajax call by testing several different conditions:

```
function getAjax() {
  let ajax = null;
  if (window.XMLHttpRequest) {
    // modern browser? use XMLHttpRequest
    ajax = new XMLHttpRequest();
  } else if (window.ActiveXObject) {
    // otherwise, use ActiveX for IE5 and IE6
    ajax = new ActiveXObject("Microsoft.XMLHTTP");
  } else {
```

```
        throw new Error("No Ajax support!");
    }
    return ajax;
}
```

This worked, but implied that you would redo the Ajax check for each and every call, even though the results of the test wouldn't ever change. There's a more efficient way to do this, and it has to do with using functions as first-class objects. We could define *two* different functions, test for the condition only once, and then assign the correct function to be used later; study the following code for such an alternative:

```
(function initializeGetAjax() {
    let myAjax = null;

    if (window.XMLHttpRequest) {
        // modern browsers? use XMLHttpRequest
        myAjax = function() {
            return new XMLHttpRequest();
        };
    } else if (window.ActiveXObject) {
        // it's ActiveX for IE5 and IE6
        myAjax = function() {
            new ActiveXObject("Microsoft.XMLHTTP");
        };
    } else {
        myAjax = function() {
            throw new Error("No Ajax support!");
        };
    }
    window.getAjax = myAjax;
})();
```

This piece of code shows two important concepts. First, we can dynamically assign a function: when this code runs, `window.getAjax` (that is, the global `getAjax` variable) will get one of three possible values according to the current browser. When you later call `getAjax()` in your code, the right function will execute without you needing to do any further browser-detection tests.

The second interesting idea is that we define the `initializeGetAjax` function, and immediately run it—this pattern is called the **immediately invoked function expression (IIFE)**. The function runs, but *cleans up after itself*, because all its variables are local and won't even exist after the function runs. We'll learn more about this later.

Adding missing functions

This idea of defining a function on the run also allows us to write *polyfills* that provide otherwise missing functions. For example, let's say that we had some code such as the following:

```
if (currentName.indexOf("Mr.") !== -1) {  
    // it's a man  
    ...  
}
```

Instead of this, you might very much prefer using the newer, clearer way of, and just write the following:

```
if (currentName.includes("Mr. ")) {  
    // it's a man  
    ...  
}
```

What happens if your browser doesn't provide `.includes()`? Once again, we can define the appropriate function on the fly, but only if needed. If `.includes()` is available, you need to do nothing, but if it is missing, you need to define a polyfill that will provide the very same workings. The following code shows an example of such a polyfill:



You can find polyfills for many modern JavaScript features at Mozilla's developer site. For example, the polyfill we used for `includes` was taken directly from https://developer.mozilla.org/en/docs/Web/JavaScript/Reference/Global_Objects/String/includes.

```
if (!String.prototype.includes) {  
    String.prototype.includes = function(search, start) {  
        "use strict";  
        if (typeof start !== "number") {  
            start = 0;  
        }  
  
        if (start + search.length > this.length) {  
            return false;  
        } else {  
            return this.indexOf(search, start) !== -1;  
        }  
    };  
}
```


When this code runs, it checks whether the `String` prototype already has the `includes` method. If not, it assigns a function to it that does the same job, so from that point onward, you'll be able to use `.includes()` without further worries. By the way, there are other ways of defining a polyfill: check the answer to question 3.5 for an alternative.



Directly modifying a standard type's prototype object is usually frowned upon, because in essence, it's equivalent to using a global variable, and thus it's prone to errors; however, this case (writing a polyfill for a well established and known function) is quite unlikely to provoke any conflicts.

Finally, if you happened to think that the Ajax example shown previously was old hat, consider this: if you want to use the more modern `fetch()` way of calling services, you will also find that not all modern browsers support it (check <http://caniuse.com/#search=fetch> to verify this), and so you'll have to use a polyfill, such as the one at <https://github.com/github/fetch>. Study the code and you'll see that it basically uses the same method as described previously to see whether a polyfill is needed and to create it.

Stubbing

Here, we will look at a use case that is similar in some aspects to using a polyfill: having a function do different work depending on the environment. The idea is to perform **stubbing**, an idea that comes from testing that involves replacing a function with another that does a simpler job, instead of doing the actual work.

Stubbing is commonly used with logging functions. You may want the application to perform detailed logging when in development, but not to say a peep when in production. A common solution would be to write something along the lines of the following:

```
let myLog = someText => {
  if (DEVELOPMENT) {
    console.log(someText); // or some other way of logging
  } else {
    // do nothing
  }
}
```

This works, but as in the example of Ajax detection, it does more work than it needs to because it checks whether the application is in development every time.

We could simplify the code (and get a really, really tiny performance gain!) if we stub out the logging function so that it won't actually log anything; an easy implementation is as follows:

```
let myLog;
if (DEVELOPMENT) {
  myLog = someText => console.log(someText);
} else {
  myLog = someText => {};
}
```

We can even do better with the ternary operator:

```
const myLog = DEVELOPMENT
  ? someText => console.log(someText)
  : someText => {};
```

This is a bit more cryptic, but I prefer it because it uses a `const`, which cannot be modified.



Given that JavaScript allows us to call functions with more parameters than arguments, and given that we aren't doing anything in `myLog()` when we are not in development, we could also have written `() => {}` and it would have worked fine. However, I do prefer keeping the same signature, and that's why I specified the `someText` argument, even if it wouldn't be used. It's your call!

You'll notice that we are using the concept of functions as first-class objects over and over again; look through all the code samples and you'll see!

Immediate invocation

There's yet another common usage of functions, usually seen in popular libraries and frameworks, that lets you bring some modularity advantages from other languages into JavaScript (even the older versions!). The usual way of writing this is something like the following:

```
(function() {
  // do something...
})();
```



Another equivalent style is `(function(){ ... })()`—note the different placement of the parentheses for the function call. Both styles have their fans; pick whichever suits you, but just follow it consistently.

You can also have the same style, but pass some arguments to the function that will be used as the initial values for its parameters:

```
(function(a, b) {  
    // do something, using the  
    // received arguments for a and b...  
})(some, values);
```

Finally, you could also return something from the function:

```
let x = (function(a, b) {  
    // ...return an object or function  
})(some, values);
```

As we mentioned previously, the pattern itself is called the IIFE (pronounced *iffy*). The name is easy to understand: you are defining a function and calling it right away, so it gets executed on the spot. Why would you do this, instead of simply writing the code inline? The reason has to do with scopes.



Note the parentheses around the function. These help the parser understand that we are writing an expression. If you were to omit the first set of parentheses, JavaScript would think you were writing a function declaration instead of an invocation. The parentheses also serve as a visual note, so readers of your code will immediately recognize the IIFE.

If you define any variables or functions within the IIFE, then because of how JavaScript defines the scope of functions, those definitions will be internal, and no other part of your code will be able to access them. Imagine that you wanted to write some complicated initialization, like the following:

```
function ready() { ... }  
  
function set() { ... }  
  
function go() { ... }  
  
// initialize things calling ready(),  
// set() and go() appropriately
```

What could go wrong? The problem hinges on the fact that you could (by accident) have a function with the same name of any of the three here, and hoisting would imply that the *last* function would be called:

```
function ready() {  
    console.log("ready");  
}
```

```
function set() {
  console.log("set");
}

function go() {
  console.log("go");
}

ready();
set();
go();

function set() {
  console.log("UNEXPECTED...");
}
// "ready"
// UNEXPECTED
// "go"
```

Oops! If you had used an IIFE, the problem wouldn't have happened. Also, the three inner functions wouldn't even be visible to the rest of the code, which helps to keep the global namespace less polluted. The following code shows a very common pattern for this:

```
(function() {
  function ready() {
    console.log("ready");
  }

  function set() {
    console.log("set");
  }

  function go() {
    console.log("go");
  }

  ready();
  set();
  go();
})();

function set() {
  console.log("UNEXPECTED...");
}
// "ready"
// set
// "go"
```

To see an example involving returned values, we could revisit the example from Chapter 1, *Becoming Functional - Several Questions*, and write the following, which would create a single counter:

```
const myCounter = (function() {
  let count = 0;
  return function() {
    count++;
    return count;
  };
})();
```

Then, every call to `myCounter()` would return an incremented count, but there is no chance that any other part of your code will overwrite the inner `count` variable because it's only accessible within the returned function.

Summary

In this chapter, we went over several ways of defining functions in JavaScript, focusing mainly on arrow functions, which have several advantages over standard functions, including being terser. We learned about the concept of currying (which we'll be revisiting later), considered some aspects of functions as first-class objects, and lastly, we considered several techniques that happen to be fully FP in concept. Rest assured that we'll be using everything in this chapter as the building blocks for more advanced techniques in the rest of the book; just wait and see!

In Chapter 4, *Behaving Properly - Pure Functions*, we will delve even more deeply into functions and learn about the concept of *pure functions*, which will lead us to an even better style of programming.

Questions

3.1 Uninitialized object? React-Redux programmers usually code **action creators** to simplify the creation of actions that will later be processed by a reducer. Actions are objects, which must include a `type` attribute that is used to determine what kind of action you are dispatching. The following code supposedly does this, but can you explain the unexpected results?

```
const simpleAction = t => {
  type: t;
};
```

```
console.log(simpleAction("INITIALIZE"));  
// undefined
```

3.2. **Are arrows allowed?** Would everything be the same if you defined `listArguments()` and `listArguments2()` from the *Working with arguments* section by using arrow functions instead of the way we did, with the `function` keyword?

3.3. **One liner:** Some programmer, particularly thrifty with lines of code, suggested rewriting `doAction2()` as a one-liner, even though you can't tell this from the formatting! What do you think: is it correct or isn't it?

```
const doAction3 = (state = initialState, action) =>  
  (dispatchTable[action.type] &&  
    dispatchTable[action.type](state, action)) ||  
  state;
```

3.4. **Spot the bug!** A programmer, working with a global store for state (similar in concept to those of Redux, Mobx, Vuex, and others used by different web frameworks) wanted to log (for debugging purposes) all calls to the store's `set()` method. After creating the new store object, he wrote the following so that the arguments to `store.set()` would be logged before actually being processed. Unfortunately, the code didn't work as expected. What's the problem? Can you spot the mistake?

```
window.store = new Store();  
const oldSet = window.store.set;  
window.store.set = (...data) => (console.log(...data), oldSet(...data));
```

3.5. **Bindless binding:** Suppose that `bind()` was not available; how could you do a polyfill for it?

4

Behaving Properly - Pure Functions

In Chapter 3, *Starting Out with Functions – A Core Concept*, we considered functions as the key elements in **Functional Programming (FP)**, went into detail about arrow functions, and introduced some concepts, such as injection, callbacks, polyfilling, and stubbing. In this chapter, we'll have the opportunity to revisit or apply some of those ideas. We will also do the following:

- Consider the notion of *purity*, and why we should care about *pure functions*—and *impure* functions as well!
- Examine the concept of *referential transparency*.
- Recognize the problems implied by side effects.
- Show some advantages of pure functions.
- Describe the main reasons behind impure functions.
- Find ways to minimize the number of impure functions.
- Focus on ways of testing both pure and impure functions.

Pure functions

Pure functions behave in the same way as mathematical functions and provide diverse benefits. A function is pure if it satisfies two conditions:

- **Given the same arguments, the function always calculates and returns the same result:** This should be true no matter how many times it's invoked or under which conditions you call it. This result cannot depend on any *outside* information or state, which could change during the program execution and cause it to return a different value. Nor can the function result depend on I/O results, random numbers, some other external variable, or a value that is not directly controllable.

- **When calculating its result, the function doesn't cause any observable *side effects*:** This includes output to I/O devices, the mutation of objects, changes to a program's state outside of the function, and so on.

If you want, you can simply say that pure functions don't depend on (and don't modify) anything outside their scope and always return the same result for the same input arguments.

Another word used in this context is **idempotency**, but it's not exactly the same. An idempotent function can be called as many times as desired, and will always produce the same result; however, this doesn't imply that the function is free from side effects. Idempotency is usually mentioned in the context of RESTful services. Let's look at a simple example showing the difference between purity and idempotency. A `PUT` call would cause a database record to be updated (a side effect), but if you repeat the call, the element will not be further modified, so the global state of the database won't change any further.

We might also invoke a software design principle and remind ourselves that a function should *do one thing, only one thing, and nothing but that thing*. If a function does anything else and has some hidden functionality, then that dependency on the state will mean that we won't be able to predict the function's output and make things harder for us as developers.

Let's look into these conditions in more detail.

Referential transparency

In mathematics, **referential transparency** is the property that lets you replace an expression with its value without changing the results of whatever you were doing.



The counterpart of referential transparency is, appropriately enough, **referential opacity**. A referentially opaque function cannot guarantee that it will always produce the same result, even when called with the same arguments.

To give a simple example, let's consider what happens with an optimizing compiler that performs *constant folding*. Suppose you have a sentence like this:

```
const x = 1 + 2 * 3;
```

The compiler might optimize the code to the following by noting that $2 * 3$ is a constant value:

```
const x = 1 + 6;
```


Even better, a new round of optimization could avoid the sum altogether:

```
const x = 7;
```

To save execution time, the compiler is taking advantage of the fact that all mathematical expressions and functions are (by definition) referentially transparent. On the other hand, if the compiler cannot predict the output of a given expression, it won't be able to optimize the code in any fashion, and the calculation will have to be done at runtime.



In lambda calculus, if you replace the value of an expression involving a function with the calculated value for the function, then that operation is called a **β (beta) reduction**. Note that you can only do this safely with referentially transparent functions.

All arithmetical expressions (involving both mathematical operators and functions) are referentially transparent: $22 * 9$ can always be replaced by 198 . Expressions involving I/O are not transparent, given that their results cannot be known until they are executed. For the same reason, expressions involving date- and time-related functions or random numbers are also not transparent.

With regard to JavaScript functions that you can produce yourself, it's quite easy to write some that won't fulfill the *referential transparency* condition. In fact, a function is not even required to return a value, though the JavaScript interpreter will return an undefined value in that situation.



Some languages distinguish between functions, which are expected to return a value, and procedures, which do not return anything, but that's not the case with JavaScript. There are also some languages that provide the means to ensure that functions are referentially transparent.

If you wanted to, you could classify functions as the following:

- **Pure functions:** These return a value that depends only on its arguments and have no side effects whatsoever.
- **Side effects:** These don't return anything (actually, in JavaScript, these functions return an `undefined` value, but that's not relevant here), but do produce some kind of side effects.
- **Functions with side effects:** This means that they return a value that may not only depend on the function arguments, but also involve side effects.

In FP, much emphasis is put on the first group, referentially transparent pure functions. Not only can a compiler reason about the program behavior (and thus be able to optimize the generated code), but also the programmer can more easily reason about the program and the relationship between its components. This in turn can help prove the correctness of an algorithm or optimize the code by replacing a function with an equivalent one.

Side effects

What are **side effects**? We can define these as a change in state or an interaction with outside elements (the user, a web service, another computer, whatever) that occurs during the execution of some calculations or a process.

There's a possible misunderstanding as to the scope of this meaning. In common daily speech, when you speak of side effects, it's a bit like talking about *collateral damage*—some unintended consequences for a given action; however, in computing, we include every possible effect or change outside the function. If you write a function that is meant to perform a `console.log()` call to display a result, then that would be considered a side effect, even if it's exactly what you intended the function to do in the first place!

In this section, we will look at the following:

- Common side effects in JavaScript programming
- The problems that global and inner states cause
- The possibility of functions mutating their arguments
- Some functions that are always troublesome

Usual side effects

In programming, there are (too many!) things that are considered side effects. In JavaScript programming, including both frontend and backend coding, the more common ones you may find include the following:

- Changing global variables.
- Mutating objects received as arguments.
- Performing any kind of I/O, such as showing an alert message or logging some text.
- Working with, and changing, the filesystem.
- Updating a database.
- Calling a web service.

- Querying or modifying the DOM.
- Triggering any external process.
- Just calling another function that happens to produce a side effect of its own. You could say that impurity is contagious: a function that calls an impure function automatically becomes impure on its own!

With this definition, let's start looking at what can cause functional impurity (or *referential opaqueness*).

Global state

Of all the preceding points, the most common reason for side effects is the usage of nonlocal variables that share a global state with other parts of the program. Since pure functions, by definition, always return the same output value given the same input arguments, if a function refers to anything outside its internal state, it automatically becomes impure. Furthermore, and this is a hindrance to debugging, to understand what a function has done, you must understand how the state got its current values, and that means understanding all of the past history from your program: not easy!

Let's write a function to detect whether a person is a legal adult by checking whether they were born at least 18 years ago. (Okay, that's not precise enough, because we are not considering the day and month of birth, but bear with me; the problem is elsewhere.) A version of an `isOldEnough()` function could be as follows:

```
let limitYear = 1999;

const isOldEnough = birthYear => birthYear <= limitYear;

console.log(isOldEnough(1960)); // true
console.log(isOldEnough(2001)); // false
```

The `isOldEnough()` function correctly detects whether a person is at least 18 years old, but it depends on an external variable for that (the variable is good for 2017 only). You cannot tell what the function does unless you know about the external variable and how it got its value. Testing would also be hard; you'd have to remember creating the global `limitYear` variable or all your tests would fail to run. Even though the function works, the implementation isn't the best that it could possibly be.

There is an exception to this rule. Check out the following case: is the following `circleArea()` function, which calculates the area of a circle given its radius, pure or not?

```
const PI = 3.14159265358979;  
const circleArea = r => PI * Math.pow(r, 2); // or PI * r ** 2
```

Even though the function is accessing an external state, the fact that `PI` is a constant (and thus cannot be modified) would allow us to substitute it inside `circleArea` with no functional change, and so we should accept that the function is pure. The function will always return the same value for the same argument, and thus fulfills our purity requirements.



Even if you were to use `Math.PI` instead of a constant as we defined in the code (a better idea, by the way), the argument would still be the same; the constant cannot be changed, so the function remains pure.

Here, we have dealt with problems caused by the global state; let's now move on to the inner state.

Inner state

The notion is also extended to internal variables, in which a local state is stored and then used for future calls. In this case, the external state is unchanged, but there are side effects that imply future differences as to the returned values from the function. Let's imagine a `roundFix()` rounding function that takes into account whether it has been rounding up or down too much so that the next time, it will round the other way, bringing the accumulated difference closer to zero. Our function will have to accumulate the effects of previous roundings to decide how to proceed next. The implementation could be as follows:

```
const roundFix = (function() {  
  let accum = 0;  
  return n => {  
    // reals get rounded up or down  
    // depending on the sign of accum  
    let nRounded = accum > 0 ? Math.ceil(n) : Math.floor(n);  
    console.log("accum", accum.toFixed(5), " result", nRounded);  
    accum += n - nRounded;  
    return nRounded;  
  };  
})();
```

Some comments regarding this function:

- The `console.log()` line is just for the sake of this example; it wouldn't be included in the real-world function. It lists the accumulated difference up to the point and the result it will return: the given number rounded up or down.
- We are using the IIFE pattern that we saw in the `myCounter()` example in the *Immediate Invocation* section of Chapter 3, *Starting Out with Functions – A Core Concept*, in order to get a hidden internal variable.
- The `nRounded` calculation could also be written as `Math[accum > 0 ? "ceil": "floor"](n)`—we test `accum` to see what method to invoke ("ceil" or "floor") and then use the `Object["method"]` notation to indirectly invoke `Object.method()`. The way we used it, I think, is more clear, but I just wanted to give you a heads up in case you happen to find this other coding style.

Running this function with just two values (recognize them?) shows that results are not always the same for a given input. The `result` part of the console log shows how the value got rounded, up or down:

```
roundFix(3.14159); // accum 0.00000    result 3
roundFix(2.71828); // accum 0.14159    result 3
roundFix(2.71828); // accum -0.14013   result 2
roundFix(3.14159); // accum 0.57815    result 4
roundFix(2.71828); // accum -0.28026   result 2
roundFix(2.71828); // accum 0.43802    result 3
roundFix(2.71828); // accum 0.15630    result 3
```

The first time around, `accum` is zero, so `3.14159` gets rounded down and `accum` becomes `0.14159` in our favor. The second time, since `accum` is positive (meaning that we have been rounding in our favor), then `2.71828` gets rounded up to 3, and now `accum` becomes negative. The third time, the same `2.71828` value gets rounded down to 2, because then the accumulated difference was negative—we got different values for the same input! The rest of the example is similar; you can get the same value rounded up or down, depending on the accumulated differences, because the function's result depends on its inner state.



This usage of the internal state is the reason why many FP programmers think that using objects is potentially bad. In OOP, we developers are used to storing information (attributes) and using them for future calculations; however, this usage is considered impure, insofar as repeated method calls may return different values, despite the fact that the same arguments are being passed.

We have now dealt with the problems caused by both global and inner states, but there are still more possible side effects. For example, what happens if a function changes the values of its arguments? Let's consider this next.

Argument mutation

You also need to be aware of the possibility that an impure function will modify its arguments. In JavaScript, arguments are passed by value, except in the case of arrays and objects, which are passed by reference. This implies that any modification to the parameters of the function will affect an actual modification of the original object or array. This can be further obscured by the fact that there are several **mutator** methods, that change the underlying objects by definition. For example, say you wanted a function that would find the maximum element of an array of strings (of course, if it were an array of numbers, you could simply use `Math.max()` with no further ado). A short implementation could be as follows:

```
const maxStrings = a => a.sort().pop();

let countries = ["Argentina", "Uruguay", "Brasil", "Paraguay"];
console.log(maxStrings(countries)); // "Uruguay"
```

The function does provide the correct result (and if you worry about foreign languages, we already saw a way around that in the *Injection - sorting it out* section of Chapter 3, *Starting Out with Functions – A Core Concept*), but it has a defect. Let's see what happened with the original array:

```
console.log(countries); // ["Argentina", "Brasil", "Paraguay"]
```

Oops—the original array was modified; this is a side effect by definition! If you were to call `maxStrings(countries)` again, then instead of returning the same result as before, it would produce another value; clearly, this is not a pure function. In this case, a quick solution is to work on a copy of the array (and we can use the spread operator to help), but we'll be dealing with more ways of avoiding these sorts of problems in Chapter 10, *Ensuring Purity – Immutability*:

```
const maxStrings2 = a => [...a].sort().pop();

let countries = ["Argentina", "Uruguay", "Brasil", "Paraguay"];
console.log(maxStrings2(countries)); // "Uruguay"
console.log(countries); // ["Argentina", "Uruguay", "Brasil", "Paraguay"]
```

So now we have found yet another cause for side effects: functions that modify their own arguments. There's a final case to consider: functions that just have to be impure!

Troublesome functions

Finally, some functions also cause problems. For instance, `Math.random()` is impure: it doesn't always return the same value, and it would certainly defeat its purpose if it did! Furthermore, each call to the function modifies a global *seed* value, from which the next *random* value will be calculated.



The fact that random numbers are actually calculated by an internal function, and are therefore not random at all (if you know the formula that's used and the initial value of the seed), implies that *pseudorandom* would be a better name for them.

For instance, consider the following function that generates random letters ("A" to "Z"):

```
const getRandomLetter = () => {
  const min = "A".charCodeAt();
  const max = "Z".charCodeAt();
  return String.fromCharCode(
    Math.floor(Math.random() * (1 + max - min)) + min
  );
};
```

The fact that it receives no arguments, but is expected to produce *different* results upon each call, clearly points out that this function is impure.



Go to https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Math/random for the explanation for the `getRandomLetter()` function I wrote and https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/String for the `.charCodeAt()` method.

Impurity can be inherited by calling functions. If a function uses an impure function, it immediately becomes impure itself. We might want to use `getRandomLetter()` in order to generate random filenames, with an optional given extension; our `getRandomFileName()` function could then be as follows:

```
const getRandomFileName = (fileExtension = "") => {
  const NAME_LENGTH = 12;
  let namePart = new Array(NAME_LENGTH);
  for (let i = 0; i < NAME_LENGTH; i++) {
    namePart[i] = getRandomLetter();
  }
  return namePart.join("") + fileExtension;
};
```



In Chapter 5, *Programming Declaratively – A Better Style*, we will see a more functional way of initializing a `namePart` array, by using `map()`.

Because of its usage of `getRandomLetter()`, `getRandomFileName()` is also impure, though it performs as expected, correctly producing totally random file names:

```
console.log(getRandomFileName(".pdf")); // "SVHSSKHXPQKG.pdf"
console.log(getRandomFileName(".pdf")); // "DCHKTMNWFHYZ.pdf"
console.log(getRandomFileName(".pdf")); // "GBTEFTVVHADO.pdf"
console.log(getRandomFileName(".pdf")); // "ATCBVUOSXLXW.pdf"
console.log(getRandomFileName(".pdf")); // "OIFADZKKNVAH.pdf"
```

Keep this function in mind; we'll see some ways around the unit testing problem later in this chapter, and we'll rewrite it a bit to help out with that.

The concern about impurity also extends to functions that access the current time or date, because their results will depend on an outside condition (namely the time of day) that is part of the *global state* of the application. We could rewrite our `isOldEnough()` function to remove the dependency upon a global variable, but it wouldn't help much. An attempt could be as follows:

```
const isOldEnough2 = birthYear =>
  birthYear <= new Date().getFullYear() - 18;

console.log(isOldEnough2(1960)); // true
console.log(isOldEnough2(2001)); // false
```

A problem has been removed—the new `isOldEnough2()` function is now *safer*. Also, as long as you don't use it near midnight just before new year's day, it will consistently return the same results, so you could say, paraphrasing the Ivory Soap slogan from the nineteenth century, that it's *about 99.44% pure*; however, an inconvenience remains: how would you test it? If you were to write some tests that worked fine today, then next year they'd start to fail. We'll have to work a bit to solve this, and we'll see how later on.

Several other functions that are also impure are those that cause I/O. If a function gets input from a source (a web service, the user himself, a file, or some other source), then obviously the returned result may vary. You should also consider the possibility of an I/O error, so the very same function, calling the same service or reading the same file, might at some point fail for reasons outside its control (you should assume that your filesystem, database, socket, and so on could be unavailable, and thus a given function call might produce an error instead of the expected constant, unvarying, answer).

Even a pure output and a generally safe statement (such as a `console.log()`) that doesn't change anything internally (at least in a visible way) causes some side effects because the user does see a change: the produced output.

Does this imply that we won't ever be able to write a program that requires random numbers, handles dates, or performs I/O, and also uses pure functions? Not at all—but it does mean that some functions won't be pure, and they will have some disadvantages that we will have to consider; we'll return to this in a bit.

Advantages of pure functions

The main advantage of using pure functions comes from the fact that they don't have any side effects. When you call a pure function, you don't need to worry about anything other than which arguments you are passing to it. Also, more to the point, you can be sure that you cannot cause any problems or break anything else because the function will only work with whatever you give it, and not with outside sources. But this is not their only advantage. Let's learn more in the following sections.

Order of execution

Another way of looking at what we have been saying in this chapter is to see pure functions as **robust**. You know that their execution—in whichever order—won't ever have any sort of impact on the system. This idea can be extended further: you could evaluate pure functions in parallel, resting assured that results wouldn't vary from what you would get in a single-threaded execution.



Unfortunately, JavaScript greatly restricts us in our parallel programming. We can make do, in very restricted ways, with web workers, but that's about as far as it goes. For Node developers, the `cluster` module may help out, though it isn't actually an alternative to threads, and only lets you spawn multiple processes, letting you use all available CPU cores. To sum it up, you don't get facilities such as Java's threads, for example, so parallelization isn't really an FP advantage in JavaScript terms.

When you work with pure functions, another consideration to keep in mind is that there's no explicit need to specify the order in which they should be called. If you work with mathematics, an expression such as $f(2) + f(5)$ is always the same as $f(5) + f(2)$; this is called the **commutative property**.

However, when you deal with impure functions, that can be `false`, as shown in the following purposefully written tricky function:

```
var mult = 1;
const f = x => {
  mult = -mult;
  return x * mult;
};

console.log(f(2) + f(5)); // 3
console.log(f(5) + f(2)); // -3
```



With impure functions such as the previous one, you cannot assume that calculating $f(3) + f(3)$ would produce the same result as $2 * f(3)$, or that $f(4) - f(4)$ would actually be 0; check it out for yourself! More common mathematical properties down the drain.

Why should you care? When you are writing code, willingly or not, you are always keeping in mind those properties you learned about, such as the commutative property. So while you might think that both expressions should produce the same result and code accordingly, you may be in for a surprise when using impure functions, with hard-to-find bugs that are difficult to fix.

Memoization

Since the output of a pure function for a given input is always the same, you can cache the function results and avoid a possibly costly recalculation. This process, which implies evaluating an expression only the first time and caching the result for later calls, is called **memoization**.

We will come back to this idea in [Chapter 6, Producing Functions – Higher-Order Functions](#), but let's look at an example done by hand. The Fibonacci sequence is always used for this example because of its simplicity and its hidden calculation costs. This sequence is defined as follows:

- For $n=0$, $fib(n)=0$
- For $n=1$, $fib(n)=1$
- For $n>1$, $fib(n)=fib(n-2)+fib(n-1)$



Fibonacci's name actually comes from *filius Bonacci*, or *son of Bonacci*. He is best known for having introduced the usage of digits 0-9 as we know them today, instead of the cumbersome Roman numbers. He derived the sequence named after him as the answer to a puzzle involving rabbits!



You can read more about it, and Fibonacci's life in general, at https://en.wikipedia.org/wiki/Fibonacci_number#History or <https://plus.maths.org/content/life-and-numbers-fibonacci>.

If you run the numbers, the sequence starts with 0, then 1, and from that point onwards, each term is the sum of the two previous ones: 1 again, then 2, 3, 5, 8, 13, 21, and so on. Programming this series by using recursion is simple; we'll revisit this example in Chapter 9, *Designing Functions – Recursion*. The following code, a direct translation of the definition, will do:

```
const fib = (n) => {
  if (n == 0) {
    return 0;
  } else if (n == 1) {
    return 1;
  } else {
    return fib(n - 2) + fib(n - 1);
  }
}
//
console.log(fib(10)); // 55, a bit slowly
```



If you really go for one-liners, you could also write `const fib = (n) => (n <= 1) ? n : fib(n-2)+fib(n-1)`—do you see why? And more importantly, is it worth the loss of clarity?

If you try out this function for growing values of n , you'll soon realize that there is a problem, and computation starts taking too much time. For example, on my machine, I took some timings, measured in milliseconds and plotted them on the following graph (of course, your mileage may vary). Since the function is quite speedy, I had to run calculations 100 times for values of n between 0 and 40. Even then, the times for small values of n were really tiny; it was only from 25 onwards that I got interesting numbers.

The chart (see *Figure 4.1*) shows an exponential growth, which bodes ill:

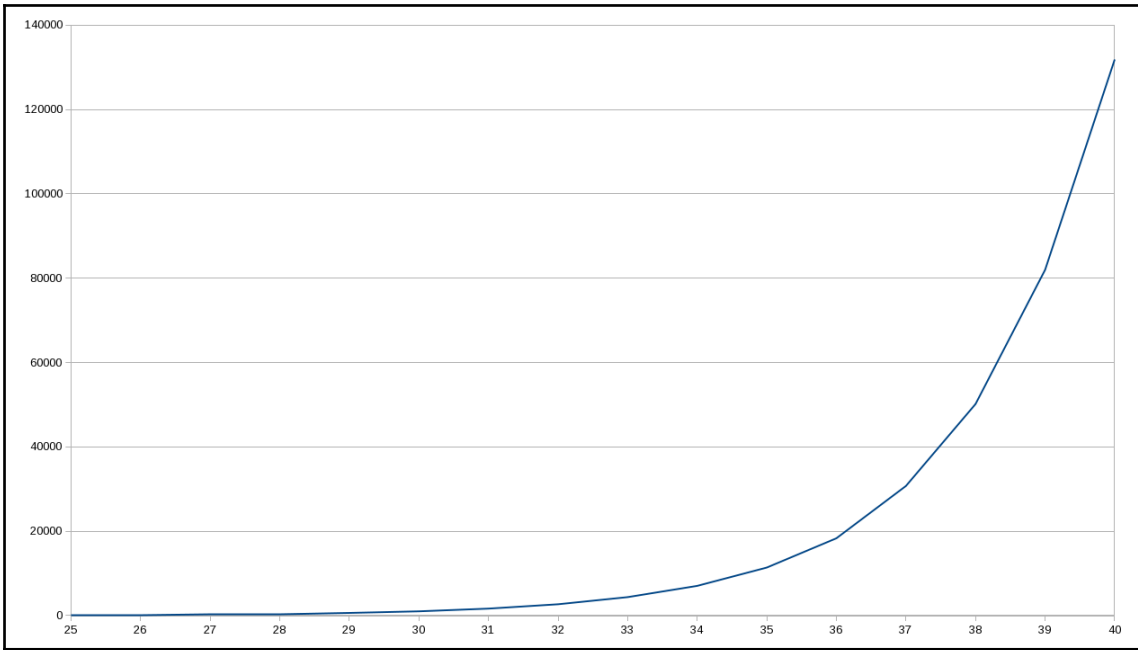


Figure 4.1: Calculation times for the fib() recursive function go up exponentially

If we draw a diagram of all the calls required to calculate `fib(6)`, you'll notice the problem. Each node represents a call to calculate `fib(n)`: we just note the value of `n` in the node. Every call, except those for `n=0` or `1`, requires further calls, as you can see in *Figure 4.2*:

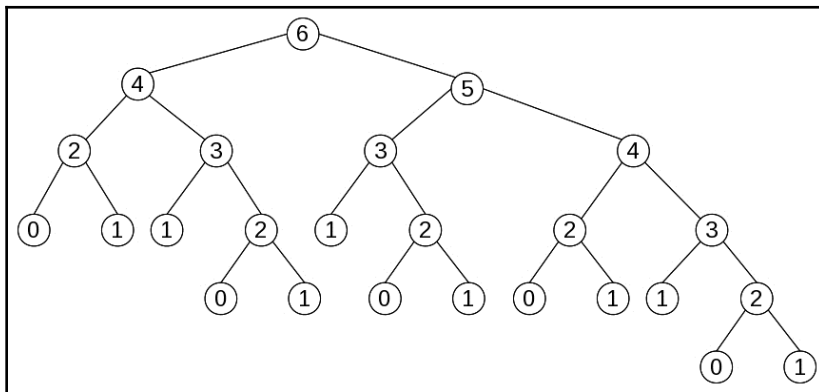


Figure 4.2: All the required calculations for fib(6) show lots of duplication

The reason for the increasing delays becomes obvious: for example, the calculation for `fib(2)` was repeated on four different occasions and `fib(3)` was itself calculated three times. Given that our function is pure, we could have stored the calculated values to avoid running the numbers over and over again. A possible version, using a `cache` array for previously calculated values, would be as follows:

```
let cache = [];  
const fib2 = (n) => {  
  if (cache[n] === undefined) {  
    if (n === 0) {  
      cache[0] = 0;  
  
    } else if (n === 1) {  
      cache[1] = 1;  
  
    } else {  
      cache[n] = fib2(n - 2) + fib2(n - 1);  
    }  
  }  
  
  return cache[n];  
}  
  
console.log(fib2(10)); // 55, as before, but more quickly!
```

Initially, the `cache` array is empty. Whenever we need to calculate the value of `fib2(n)`, we check whether it was already calculated beforehand. If that's not true, we do the calculation, but with a twist: instead of immediately returning the value, first we store it in the cache and then we return it. This means that no calculation will be done twice: after we have calculated `fib2(n)` for a certain `n`, future calls will not repeat the procedure, and will simply return the value that was already evaluated before.

A few short notes:

- We memoized the function by hand, but we can do it with a higher-order function. We'll see this later in Chapter 6, *Producing Functions – Higher-Order Functions*. It is perfectly possible to memoize a function without having to change or rewrite it.
- Using a global variable for the cache isn't a very good practice; we could have used an IIFE and a closure to hide the cache from sight—do you see how? See the `myCounter()` example in the *Immediate invocation* section of Chapter 3, *Starting Out with Functions – A Core Concept*, to review how we'd do this.

- Of course, you will be constrained by the available cache space, and it's possible you could eventually crash your application by eating up all available RAM. Resorting to external memory (a database, a file, or a cloud solution) would probably eat up all the performance advantages of caching. There are some standard solutions (involving eventually deleting items from the cache) but they are beyond the scope of this book.

Of course, you don't need to do this for every pure function in your program. You'd do this sort of optimization only for frequently called functions that take a certain important amount of time—if it were otherwise, then the added cache management time would end up costing more than whatever you expected to save!

Self-documentation

Pure functions have another advantage. Since everything the function needs to work with is given to it through its parameters, with no kind of hidden dependency whatsoever, when you read its source code, you have all you need to understand the function's objective.

An extra advantage: knowing that a function doesn't access anything beyond its parameters makes you more confident in using it, since you won't be accidentally producing a side effect; the only thing the function will accomplish is what you already learned through its documentation.

Unit tests (which we'll be covering in the next section) also work as documentation, because they provide examples of what the function returns when given certain arguments. Most programmers will agree that the best kind of documentation is full of examples, and each unit test can be considered such a sample case.

Testing

Yet another advantage of pure functions—and one of the most important ones—has to do with unit testing. Pure functions have a single responsibility: producing their output in terms of their input. So when you write tests for pure functions, your work is greatly simplified because there is no context to consider and no state to simulate.

You can simply focus on providing inputs and checking outputs because all function calls can be reproduced in isolation, independently from the *rest of the world*.

We have seen several aspects of pure functions. Let's move on and learn about impure functions a bit, and finish by testing both pure and impure functions.

Impure functions

If you decided to completely forego all kinds of side effects, then your programs would only be able to work with hardcoded inputs, and wouldn't be able to show you the calculated results! Similarly, most web pages would be useless: you wouldn't be able to make any web services calls or update the DOM; you'd have static pages only. And your Node code would be really useless for server-side work, as it wouldn't be able to perform any I/O.

Reducing side effects is a good goal in FP, but we shouldn't go overboard with it! So let's think of how to avoid using impure functions, if possible, and how to deal with them if not, looking for the best possible way to contain or limit their scope.

Avoiding impure functions

Earlier in this chapter, we saw the more common reasons for using impure functions. Let's now consider how we can reduce the number of impure functions, even if doing away with all of them isn't really feasible. Basically, we'll have two methods for this:

- Avoiding the usage of state
- Using a common pattern, *injection*, to have impurity in a controlled fashion

Avoiding the usage of state

With regard to the usage of the global state—both getting and setting it—the solution is well known. The key points to this are as follows:

- Provide whatever is needed of the global state to the function as arguments.
- If the function needs to update the state, it shouldn't do it directly, but rather produce a new version of the state and return it.
- It should be the responsibility of the caller to take the returned state, if any, and update the global state.

This is the technique that Redux uses for its reducers. The signature for a reducer is `(previousState, action) => newState`, meaning that it takes a state and an action as parameters and returns a new state as the result. Most specifically, the reducer is not supposed to simply change the `previousState` argument, which must remain untouched (we'll learn more about this in Chapter 10, *Ensuring Purity – Immutability*).

With regard to our first version of the `isOldEnough()` function, which used a global `limitYear` variable, the change is simple enough: we just have to provide `limitYear` as a parameter for the function. With this change, it will become pure, since it will produce its result by only using its parameters. Even better, we should provide the current year and let the function do the math instead of forcing the caller to do so. Our newer version of the adult age test could then be as follows:

```
const isOldEnough3 = (currentYear, birthYear) => birthYear <=
  currentYear-18;
```

Obviously, we'll have to change all the calls to provide the required `limitYear` argument (we could also use currying, as we will see in Chapter 7, *Transforming Functions – Currying and Partial Application*). The responsibility of initializing the value of `limitYear` still remains outside of the function, as before, but we have managed to avoid a defect.

We can also apply this solution to our peculiar `roundFix()` function. As you will recall, the function worked by accumulating the differences caused by rounding, and deciding whether to round up or down depending on the sign of that accumulator. We cannot avoid using that state, but we can split off the rounding part from the accumulating part. Our original code (with fewer comments and logging) looked as follows:

```
const roundFix1 = (function() {
  let accum = 0;
  return n => {
    let nRounded = accum > 0 ? Math.ceil(n) : Math.floor(n);
    accum += n - nRounded;
    return nRounded;
  };
})();
```

The newer version would have two parameters:

```
const roundFix2 = (a, n) => {
  let r = a > 0 ? Math.ceil(n) : Math.floor(n);
  a += n - r;
  return {a, r};
};
```

How would you use this function? Initializing the accumulator, passing it to the function, and updating it afterward are now the responsibility of the caller code. You would have something like the following:

```
let accum = 0;

// ...some other code...
```



```
let {a, r} = roundFix2(accum, 3.1415);
accum = a;
console.log(accum, r); // 0.1415 3
```

Note the following:

- The `accum` phrase is now part of the global state of the application.
- Since `roundFix2()` needs it, the current accumulator value is provided in each call.
- The caller is responsible for updating the global state, not `roundFix2()`.



Note the usage of the destructuring assignment in order to allow a function to return more than a value and easily store each one in a different variable. For more on this, go to https://developer.mozilla.org/en/docs/Web/JavaScript/Reference/Operators/Destructuring_assignment.

This new `roundFix2()` function is totally pure and can be easily tested. If you want to hide the accumulator from the rest of the application, you could still use a closure, as we have seen in other examples, but that would again introduce impurity in your code—your call!

Injecting impure functions

If a function becomes impure because it needs to call another function that is itself impure, a way around this problem is to inject the required function in the call. This technique actually provides more flexibility in your code and allows for easier future changes, as well as less complex unit testing.

Let's consider the random filename generator function that we saw earlier. The problematic part of this function is its usage of `getRandomLetter()` to produce the filename:

```
const getRandomFileName = (fileExtension = "") => {
  ...
  for (let i = 0; i < NAME_LENGTH; i++) {
    namePart[i] = getRandomLetter();
  }
  ...
};
```

A way to solve this is to replace the impure function with an injected external one; we must now provide a `randomLetterFunc()` argument for our random filename function to use:

```
const getRandomFileName2 = (fileExtension = "", randomLetterFunc) => {
  const NAME_LENGTH = 12;
```

```
let namePart = new Array(NAME_LENGTH);
for (let i = 0; i < NAME_LENGTH; i++) {
  namePart[i] = randomLetterFunc();
}
return namePart.join("") + fileExtension;
};
```

Now, we have removed the inherent impurity from this function. If we want to provide a predefined pseudorandom function that actually returns fixed, known, values, we will be able to easily unit test this function; we'll be seeing how to do this in the following examples. The usage of the function will change, and we would have to write the following:

```
let fn = getRandomFileName2(".pdf", getRandomLetter);
```

If this bothers you, you may want to provide a default value for the `randomLetterFunc` parameter, as follows:

```
const getRandomFileName2 = (
  fileExtension = "",
  randomLetterFunc = getRandomLetter
) => {
  ...
};
```

You can also solve this using partial application, as we'll be seeing in [Chapter 7, Transforming Functions – Currying and Partial Application](#).

This hasn't actually avoided the usage of impure functions. Normally, you'll call `getRandomFileName()` by providing it with the random letter generator we wrote, so it will behave as an impure function; however, for testing purposes, if you provide a function that returns predefined (that is, not random) letters, you'll be able to test it as if it was pure much more easily.

But what about the original problem function, `getRandomLetter()`? We can apply the same trick and write a new version, like the following, which will have an argument that will produce random numbers:

```
const getRandomLetter = (getRandomInt = Math.random) => {
  const min = "A".charCodeAt();
  const max = "Z".charCodeAt();
  return String.fromCharCode(
    Math.floor(getRandomInt() * (1 + max - min)) + min
  );
};
```

For normal usage, `getRandomFileName()` would call `getRandomLetter()` without providing any parameters, which would imply that the called function would behave in its expected random ways. But if we want to test whether the function does what we wanted, we can run it with an injected function that will return whatever we decide, letting us test it thoroughly.

This idea is actually very important and has a wide spectrum of applications to other problems. For example, instead of having a function directly access the DOM, we can provide it with injected functions that would do this. For testing purposes, it would be simple to verify that the tested function actually does what it needs to do without really interacting with the DOM (of course, we'd have to find some other way to test those DOM-related functions). This can also apply to functions that need to update the DOM, generate new elements, and do all sorts of manipulations—you just use some intermediary functions.

Is your function pure?

Let's end this section by considering an important question: can you ensure that a function is actually pure? To show the difficulties of this task, we'll go back to the simple `sum3()` function that we saw in the *Spread* section of Chapter 1, *Becoming Functional – Several Questions*, just rewritten to use arrow functions for brevity. Would you say that this function is pure? It certainly looks like it!

```
const sum3 = (x, y, z) => x + y + z;
```

Let's see: the function doesn't access anything but its parameters, doesn't even try to modify them (not that it could (or could it?)), doesn't perform any I/O, or work with any of the impure functions or methods that we mentioned earlier. What could go wrong?

The answer has to do with checking your assumptions. For example, who says the arguments for this function should be numbers? You might say to yourself *Okay, they could be strings, but the function would still be pure, wouldn't it?*, but for an (assuredly evil!) answer to that, see the following code:

```
let x = {};  
x.valueOf = Math.random;  
  
let y = 1;  
let z = 2;  
  
console.log(sum3(x, y, z)); // 3.2034400919849431  
console.log(sum3(x, y, z)); // 3.8537045249277906  
console.log(sum3(x, y, z)); // 3.0833258308458734
```



Note the way that we assigned a new function to the `x.valueOf` method. We are taking full advantage of the fact that functions are first-class objects. See the *An unnecessary mistake* section in [Chapter 3, Starting Out with Functions – A Core Concept](#), for more on this.

Well, `sum3()` ought to be pure, but it actually depends on whatever parameters you pass to it; in JavaScript, you can make a pure function behave in an impure way! You might console yourself by thinking that surely no one would pass such arguments, but edge cases are usually where bugs reside. But you need not resign yourself to abandoning the idea of pure functions. By adding some type checking (TypeScript might come in handy, as mentioned in the *Using transpilers* section of [Chapter 1, Becoming Functional – Several Questions](#)), you could at least prevent some cases, though JavaScript won't ever let you be totally sure that your code is *always* pure!

Over the course of these sections, we have gone through the characteristics of both pure and impure functions. Let's finish the chapter by looking at how we can test all these sorts of functions.

Testing – pure versus impure

We have seen how pure functions are conceptually better than impure ones, but we cannot set out on a crusade to vanquish all impurity from our code. First, no one can deny that side effects can be useful, or at least unavoidable: you will need to interact with the DOM or call a web service, and there are no ways to do this in a pure way. So, rather than bemoaning the fact that you have to allow for impurity, try to structure your code so that you can isolate the impure functions and let the rest of your code be the best it can possibly be.

With this in mind, you'll have to be able to write unit tests for all kinds of functions, pure or impure. Writing unit tests for functions is different, in terms of both their difficulty and complexity than dealing with pure or impure functions. While coding tests for the former is usually quite simple and follows a basic pattern, the latter usually requires scaffolding and complex setups. So let's finish this chapter by seeing how to go about testing both types of function.

Testing pure functions

Given the characteristics of pure functions that we have already described, most of your unit tests could simply be the following:

- Call the function with a given set of arguments.
- Verify that the results match what you expected.

Let's start with a couple of simple examples. Testing the `isOldEnough()` function would have been more complex than we needed for the version that required access to a global variable. On the other hand, the last version, `isOldEnough3()`, which didn't require anything because it received two parameters, is simple to test:

```
describe("isOldEnough", function() {
  it("is false for people younger than 18", () => {
    expect(isOldEnough3(1978, 1963)).toBe(false);
  });
  it("is true for people older than 18", () => {
    expect(isOldEnough3(1988, 1965)).toBe(true);
  });
  it("is true for people exactly 18", () => {
    expect(isOldEnough3(1998, 1980)).toBe(true);
  });
});
```

Testing another of the pure functions that we wrote is equally simple, but we must be careful because of precision considerations. If we test the `circleArea` function, we must use the Jasmine `toBeCloseTo()` matcher, which allows for approximate equality when dealing with floating-point numbers. Other than this, the tests are just about the same—call the function with known arguments and check the expected results:

```
describe("circle area", function() {
  it("is zero for radius 0", () => {
    let area = circleArea(0);
    expect(area).toBe(0);
  });
  it("is PI for radius 1", () => {
    let area = circleArea(1);
    expect(area).toBeCloseTo(Math.PI);
  });
  it("is approximately 12.5664 for radius 2", () => {
    let area = circleArea(2);
    expect(area).toBeCloseTo(12.5664);
  });
});
```

No difficulty whatsoever! The test run reports success for both suites (see *Figure 4.3*):

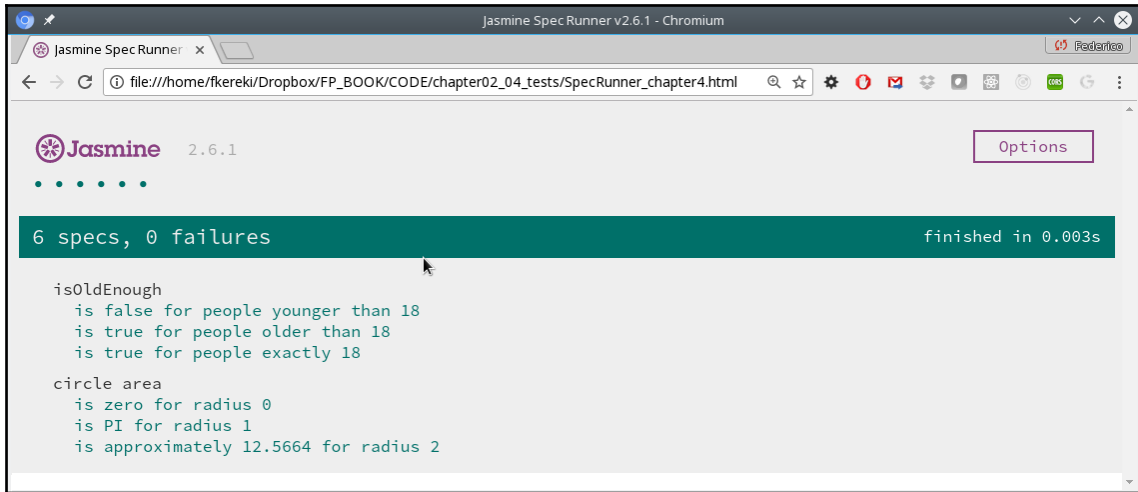


Figure 4.3: A successful test run for a pair of simple pure functions

Now that we don't have to worry about pure functions, let's move on to the impure ones that we dealt with by transforming them into pure equivalents.

Testing purified functions

When we considered the following `roundFix` special function, which required us to use the state to accumulate the differences due to rounding, we produced a new version by providing the current state as an added parameter and by having the function return two values—the rounded one and the updated state:

```
const roundFix2 = (a, n) => {
  let r = a > 0 ? Math.ceil(n) : Math.floor(n);
  a += n - r;
  return {a, r};
};
```

This function is now pure, but testing it requires validating not only the returned values but also the updated states. We can base our tests on the experiments we did previously. Once again, we have to use `toBeCloseTo()` for dealing with floating-point numbers, but we can use `toBe()` with integers, which produces no rounding errors. We could write our tests as follows:

```
describe("roundFix2", function() {
  it("should round 3.14159 to 3 if differences are 0", () => {
    const {a, r} = roundFix2(0.0, 3.14159);
    expect(a).toBeCloseTo(0.14159);
    expect(r).toBe(3);
  });
  it("should round 2.71828 to 3 if differences are 0.14159", () => {
    const {a, r} = roundFix2(0.14159, 2.71828);
    expect(a).toBeCloseTo(-0.14013);
    expect(r).toBe(3);
  });
  it("should round 2.71828 to 2 if differences are -0.14013", () => {
    const {a, r} = roundFix2(-0.14013, 2.71828);
    expect(a).toBeCloseTo(0.57815);
    expect(r).toBe(2);
  });
  it("should round 3.14159 to 4 if differences are 0.57815", () => {
    const {a, r} = roundFix2(0.57815, 3.14159);
    expect(a).toBeCloseTo(-0.28026);
    expect(r).toBe(4);
  });
});
```

We took care to include several cases, with positive, zero, or negative accumulated differences, and checking whether it rounded up or down on each occasion. We could certainly go further by rounding negative numbers, but the idea is clear: if your function takes the current state as a parameter and updates it, the only difference with the pure functions tests are that you will also have to test whether the returned state matches your expectations.

Let's now consider the alternative way of testing for our pure `getRandomLetter()` variant; let's call it `getRandomLetter2()`. This is simple: you just have to provide a function that will itself produce *random* numbers. (This kind of function, in testing parlance, is called a **stub**). There's no limit to the complexity of a stub, but you'll want to keep it simple.

We can then do some tests, based on our knowledge of the workings of the function, to verify that low values produce an A and values close to 1 produce a Z, so we can have a little confidence that no extra values are produced. We should also test that a middle value (around 0.5) should produce a letter around the middle of the alphabet; however, keep in mind that this kind of test is not very good—if we substituted an equally valid `getRandomLetter()` variant, it might be the case that the new function could work perfectly well, but not pass this test, because of a different internal implementation! Our tests could be written as follows:

```
describe("getRandomLetter2", function() {
  it("returns A for values close to 0", () => {
    const letterSmall = getRandomLetter2(() => 0.0001);
    expect(letterSmall).toBe("A");
  });
  it("returns Z for values close to 1", () => {
    const letterBig = getRandomLetter2(() => 0.99999);
    expect(letterBig).toBe("Z");
  });
  it("returns a middle letter for values around 0.5", () => {
    const letterMiddle = getRandomLetter2(() => 0.49384712);
    expect(letterMiddle).toBeGreaterThan("G");
    expect(letterMiddle).toBeLessThan("S");
  });
  it("returns an ascending sequence of letters for ascending values", () =>
  {
    const a = [0.09, 0.22, 0.6];
    const f = () => a.shift(); // impure!!
    const letter1 = getRandomLetter2(f);
    const letter2 = getRandomLetter2(f);
    const letter3 = getRandomLetter2(f);
    expect(letter1).toBeLessThan(letter2);
    expect(letter2).toBeLessThan(letter3);
  });
});
```

Testing our filename generator can be done in a similar way, by using stubs. We can provide a simple stub that will return the letters of "SORTOFRANDOM" in sequence (this function is quite impure; can you see why?). So we can verify that the returned filename matches the expected name and a couple more properties of the returned filename, such as its length and its extension. Our test could then be written as follows:

```
describe("getRandomFileName", function() {
  let a = [];
  const f = () => a.shift();
  beforeEach(() => {
    a = "SORTOFRANDOM".split("");
  });
});
```



```
});  
it("uses the given letters for the file name", () => {  
  const fileName = getRandomFileName("", f);  
  expect(fileName.startsWith("SORTOFRANDOM")).toBe(true);  
});  
it("includes the right extension, and has the right length", () => {  
  const fileName = getRandomFileName(".pdf", f);  
  expect(fileName.endsWith(".pdf")).toBe(true);  
  expect(fileName.length).toBe(16);  
});  
});
```

Testing *purified* impure functions is very much the same as testing originally pure functions. Now we need to consider some cases of truly impure functions, because, as we said, it's quite certain that at some time or another you'll have to use such functions.

Testing impure functions

For starters, we'll go back to our `getRandomLetter()` function. With insider knowledge about its implementation (this is called **white-box testing**, as opposed to **black-box testing**, where we know nothing about the function code itself), we can *spy* (a Jasmine term) on the `Math.random()` method and set a *mock* function that will return whatever values we desire.

We can revisit some of the test cases that we went through in the previous section. In the first case, we set `Math.random()` to return `0.0001` (and test that it was actually called) and we also test that the final return was `A`. In the second case, just for variety, we set things up so that `Math.random()` can be called twice, returning two different values. We also verify that there were two calls to the function and that both results were `Z`. The third case shows yet another way of checking how many times `Math.random()` (or rather, our mocked function) was called. Our revisited tests could look as follows:

```
describe("getRandomLetter", function() {  
  it("returns A for values close to 0", () => {  
    spyOn(Math, "random").and.returnValue(0.0001);  
    const letterSmall = getRandomLetter();  
    expect(Math.random()).toHaveBeenCalled();  
    expect(letterSmall).toBe("A");  
  });  
  it("returns Z for values close to 1", () => {  
    spyOn(Math, "random").and.returnValues(0.98, 0.999);  
    const letterBig1 = getRandomLetter();  
    const letterBig2 = getRandomLetter();  
    expect(Math.random()).toHaveBeenCalledTimes(2);  
  });  
});
```

```

    expect(letterBig1).toBe("Z");
    expect(letterBig2).toBe("Z");
  });
  it("returns a middle letter for values around 0.5", () => {
    spyOn(Math, "random").and.returnValue(0.49384712);
    const letterMiddle = getRandomLetter();
    expect(Math.random.calls.count()).toEqual(1);
    expect(letterMiddle).toBeGreaterThan("G");
    expect(letterMiddle).toBeLessThan("S");
  });
});

```



Of course, you wouldn't go around inventing whatever tests came into your head. In all likelihood, you'll work from the description of the desired `getRandomLetter()` function, which was written before you started to code or test it. In our case, I'm making do as if that specification did exist, and it pointedly said, for example, that values close to 0 should produce an A, values close to 1 should return Z, and the function should return ascending letters for ascending random values.

Now, how would you test the original `getRandomFileName()` function, the one that called the impure `getRandomLetter()` function? That's a much more complicated problem. What kind of expectations do you have? You cannot know the results it will give, so you won't be able to write any `.toBe()` type of tests. What you can do is to test for some properties of the expected results, and also, if your function implies randomness of some kind, you can repeat the tests as many times as you want so that you have a bigger chance of catching a bug. We could do some tests along the lines of the following code:

```

describe("getRandomFileName, with an impure getRandomLetter function",
function() {
  it("generates 12 letter long names", () => {
    for (let i = 0; i < 100; i++) {
      expect(getRandomFileName().length).toBe(12);
    }
  });
  it("generates names with letters A to Z, only", () => {
    for (let i = 0; i < 100; i++) {
      let n = getRandomFileName();
      for (j = 0; j < n.length; j++) {
        expect(n[j] >= "A" && n[j] <= "Z").toBe(true);
      }
    }
  });
  it("includes the right extension if provided", () => {
    const fileName1 = getRandomFileName(".pdf");
    expect(fileName1.length).toBe(16);
  });
});

```

```
    expect(fileName1.endsWith(".pdf")).toBe(true);
  });
  it("doesn't include any extension if not provided", () => {
    const fileName2 = getRandomFileName();
    expect(fileName2.length).toBe(12);
    expect(fileName2.includes(".")).toBe(false);
  });
});
```

We are not passing any random letter generator function to `getFileName()`, so it will use the original, impure one. We ran some of the tests a hundred times, as extra insurance.



When testing code, always remember that *absence of evidence* isn't *evidence of absence*. Even if our repeated tests succeed, there is no guarantee that, with some other random input, they won't produce an unexpected, and hitherto undetected, error.

Let's do another *property* test. Suppose we want to test a shuffling algorithm; we might decide to implement the Fisher-Yates version along the lines of the following code. As implemented, the algorithm is doubly impure: it doesn't always produce the same result (obviously!) and it modifies its input parameter:

```
const shuffle = arr => {
  const len = arr.length;
  for (let i = 0; i < len - 1; i++) {
    let r = Math.floor(Math.random() * (len - i));
    [arr[i], arr[i + r]] = [arr[i + r], arr[i]];
  }
  return arr;
};

var xxx = [11, 22, 33, 44, 55, 66, 77, 88];
console.log(shuffle(xxx));
// [55, 77, 88, 44, 33, 11, 66, 22]
```



For more on this algorithm—including some pitfalls for the unwary programmer—see https://en.wikipedia.org/wiki/Fisher-Yates_shuffle.

How could you test this algorithm? Given that the result won't be predictable, we can check for the properties of its output. We can call it with a known array and then test some properties of it:

```
describe("shuffleTest", function() {
  it("shouldn't change the array length", () => {
```

```
    let a = [22, 9, 60, 12, 4, 56];
    shuffle(a);
    expect(a.length).toBe(6);
  });
  it("shouldn't change the values", () => {
    let a = [22, 9, 60, 12, 4, 56];
    shuffle(a);
    expect(a.includes(22)).toBe(true);
    expect(a.includes(9)).toBe(true);
    expect(a.includes(60)).toBe(true);
    expect(a.includes(12)).toBe(true);
    expect(a.includes(4)).toBe(true);
    expect(a.includes(56)).toBe(true);
  });
});
```

We had to write the second part of the unit tests in that way because, as we saw, `shuffle()` modifies the input parameter.

Summary

In this chapter, we introduced the concept of *pure functions* and studied why they matter. We also saw the problems caused by *side effects*, one of the causes of impure functions, looked at some ways of *purifying* such impure functions, and finally, we saw several ways of performing unit tests, for both pure and impure functions. With these techniques, you'll be able to favor using pure functions in your programming, and when impure functions are needed, you'll have some ways of using them in a controlled way.

In Chapter 5, *Programming Declaratively – A Better Style*, we'll show other advantages of FP: how you can program in a declarative fashion at a higher level for simpler and more powerful code.

Questions

4.1. Minimalistic function: Functional programmers sometimes tend to write code in a minimalistic way. Can you examine the following version of the Fibonacci function and explain whether it works, and if so, how?

```
const fib2 = n => (n < 2 ? n : fib2(n - 2) + fib2(n - 1));
```

4.2. A cheap way: The following version of the Fibonacci function is quite efficient and doesn't do any unnecessary or repeated computations. Can you see how? Here's a suggestion: try to calculate `fib4(6)` by hand and compare it with the example given earlier in the book:

```
const fib4 = (n, a = 0, b = 1) => (n === 0 ? a : fib4(n - 1, b, a + b));
```

4.3. A shuffle test: How would you write unit tests for `shuffle()` to test whether it works correctly with arrays with *repeated* values?

4.4. Breaking laws: Using `toBeCloseTo()` is very practical, but it can cause some problems. Some basic mathematics properties are as follows:

- A number should equal itself: for any number a , a should equal a .
- If a number a equals number b , then b should equal a .
- If a equals b , and b equals c , then a should equal c .
- If a equals b , and c equals d , then $a+c$ should equal $b+d$.
- If a equals b , and c equals d , then $a-c$ should equal $b-d$.
- If a equals b , and c equals d , then $a*c$ should equal $b*d$.
- If a equals b , and c equals d , then a/c should equal b/d .

Does `toBeCloseTo()` also satisfy all these properties?

4.5. Must return? A simple, almost philosophical question: must pure functions always return something? Could you have a pure function that didn't include a `return`?

4.6. JavaScript does math? In the *Testing purified functions* section, we mentioned the need for `toBeCloseTo()` because of precision problems. A related question, often asked in job interviews, is: *what will the following code output, and why?*

```
const a = 0.1;
const b = 0.2;
const c = 0.3;

if (a + b === c) {
  console.log("Math works!");
} else {
  console.log("Math failure?");
}
```

5 Programming Declaratively - A Better Style

Up to now, we haven't really been able to appreciate the possibilities of **Functional Programming (FP)** as it pertains to working in a higher-level, declarative fashion. In this chapter, we will correct this, and start getting shorter, more concise, and easier to understand code, by using some **higher-order functions (HOF)**; that is, functions that take functions as parameters, such as the following:

- `reduce()` and `reduceRight()` to apply an operation to a whole array, reducing it to a single result
- `map()` to transform one array into another by applying a function to each of its elements
- `flat()` to make a single array out of an array of arrays
- `flatMap()` to mix together mapping and flattening
- `forEach()` to simplify writing loops by abstracting the necessary looping code

We'll also be able to perform searches and selections with the following:

- `filter()` to pick some elements from an array
- `find()` and `findIndex()` to search for elements that satisfy a condition
- A pair of predicates, `every()` and `some()`, to check an array for a Boolean test

Using these functions lets you work more declaratively, and you'll see that your focus will shift to what you need to do and not so much how it's going to be done; the dirty details are hidden inside our functions. Instead of writing a series of possibly nested `for` loops, we'll focus on using functions as building blocks to specify our desired result.



We will also be using these functions to work with events in a declarative style, as we'll see in [Chapter 11, *Implementing Design Patterns – The Functional Way*](#), when we use the *observer* pattern.

We will also be able to work in a *fluent* fashion, in which the output of a function becomes the input of the next one, a style we will look at later.

Transformations

The first set of operations that we are going to consider work on an array and process it in the base of a function to produce some results. There are several possible results: a single value with the `reduce()` operation; a new array with `map()`; or just about any kind of result with `forEach()`.



If you Google around, you will find some articles that declare that these functions are not efficient because a loop done by hand can be faster. This, while possibly true, is practically irrelevant. Unless your code really suffers from speed problems and you are able to measure that the slowness derives from the use of these higher-order functions, trying to avoid them using longer code, with a higher probability of bugs, simply doesn't make much sense.

Let's start by considering the preceding list of functions in order, starting with the most general of all, which, as we'll see, can even be used to emulate the rest of the transformations in this chapter!

Reducing an array to a value

Answer this question: how many times have you had to loop through an array, performing an operation (say, summing) to produce a single value (maybe the sum of all the array values) as a result? Probably many, many, many times. This kind of operation can usually be implemented functionally by applying `reduce()` and `reduceRight()`. Let's start with the former!



Time for some terminology! In usual FP parlance, we speak of **folding** operations: `reduce()` is **foldl** (for *fold left*) or just plain **fold**, and `reduceRight()` is correspondingly known as **foldr**. In category theory terms, both operations are **catamorphisms**: the reduction of all the values in a *container* down to a single result.

The inner workings of the `reduce()` function are illustrated in *Figure 5.1*. See how it traverses the array, applying a reducing function to each element and to the accumulated value:

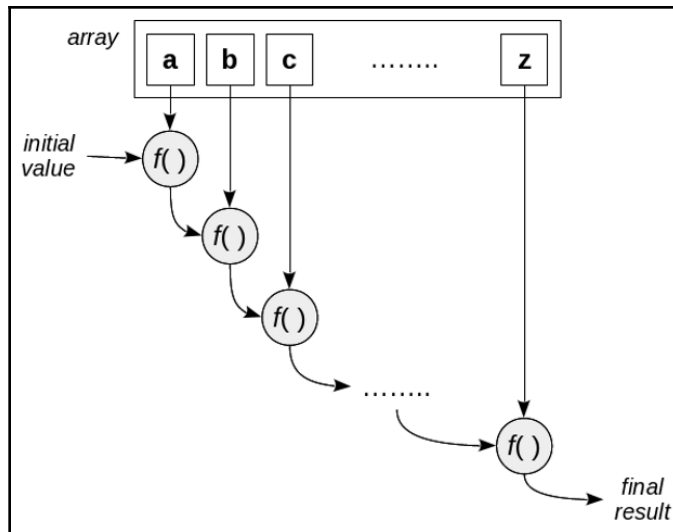


Figure 5.1: The workings of the reduce operation

Why should you always try to use `reduce()` or `reduceRight()` instead of hand-coded loops? The following points might answer this question:

- All the aspects of loop control are automatically taken care of, so you don't even have the possibility of, say, an *off-by-one* mistake.
- The initialization and handling of the result values are also done implicitly.
- Unless you work really hard at being impure and modifying the original array, your code will be side effect free.

Now that we can `reduce()` an array, let's see some of its practical use cases.

Summing an array

The most common example of the application of `reduce()`, usually seen in all textbooks and on all web pages, is the summing of all of the elements of an array. So, in order to keep with tradition, let's start with precisely this example!

Basically, to reduce an array, you must provide a **dyadic** function (that is, a function with two parameters; **binary** would be another name for that) and an initial value. In our case, the function will sum its two arguments. Initially, the function will be applied to the provided initial value and the first element of the array, so for us, the first result we have to provide is a zero, and the first result will be the first element itself. Then, the function will be applied again, this time to the result of the previous operation, and the second element of the array, and so the second result will be the sum of the first two elements of the array. Progressing in this fashion along the whole array, the final result will be the sum of all its elements:

```
const myArray = [22, 9, 60, 12, 4, 56];
const sum = (x, y) => x + y;
const mySum = myArray.reduce(sum, 0); // 163
```

You don't actually need the `sum` definition—you could have just written `myArray.reduce((x, y) => x+y, 0)`—however, when written in this fashion, the meaning of the code is clearer: you want to reduce the array to a single value by summing all its elements. Instead of having to write out the loop, initializing a variable to hold the result of the calculations, and going through the array doing the sums, you just declare what operation should be performed. This is what I meant when I said that programming with functions such as those that we'll see in this chapter allows you to work more declaratively, focusing on *what* rather than *how*.



You can also even do this without providing the initial value: if you skip it, the first value of the array will be used, and the internal loop will start with the second element of the array; however, be careful if the array is empty, and if you skipped providing an initial value, as you'll get a runtime error! See https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Array/Reduce for more details.

We can change the reducing function to see how it progresses through its calculations by just including a little bit of impurity!

```
const sumAndLog = (x, y) => {
  console.log(` ${x}+${y}=${x + y}`);
  return x + y;
};

myArray.reduce(sumAndLog, 0);
```

The output would be as follows:

```
0+22=22
22+9=31
31+60=91
91+12=103
103+4=107
107+56=163
```

You can see how the first sum was done by adding the initial value (0) and the first element of the array, how that result was used in the second addition, and so on.



Part of the reason for the *foldl* name seen previously (at least, its ending `l`) should now be clear: the reducing operation proceeds from left to right, from the first element to the last. You may wonder, however, how it would have been named if it had been defined by a right-to-left language (such as Arabic, Hebrew, Farsi, or Urdu) speaker!

This example is common and well-known; let's now do something a bit more complicated. As we'll find out, `reduce()` will be quite useful for many different objectives!

Calculating an average

Let's work a bit more. How do you calculate the average of a list of numbers? If you were explaining this to someone, your answer would surely be something like *sum all the elements in the list and divide that by the number of elements*. This, in programming terms, is not a **procedural** description (you don't explain how to sum elements, or how to traverse the array), but rather a **declarative** one, since you say what to do, not how to do it.

We can transform that description of the calculation into an almost self-explanatory function:

```
const average = arr => arr.reduce(sum, 0) / arr.length;

console.log(average(myArray)); // 27.166667
```

The definition of `average()` follows what would be a verbal explanation: sum the elements of the array, starting from 0, and divide by the array's length—simpler: impossible!



As we mentioned in the previous section, you could also have written `arr.reduce(sum)` without specifying the initial value (0) for the reduction; it's even shorter and closer to the verbal description of the required calculation. This, however, is less safe, because it would fail (producing a runtime error) should the array be empty. So it's better to always provide the starting value.

This isn't, however, the only way of calculating the average. The reducing function also gets passed the index of the current position of the array as well as the array itself, so you could do something different than last time:

```
const average2 = (sum, val, ind, arr) => {
  sum += val;
  return ind === arr.length - 1 ? sum / arr.length : sum;
};

console.log(myArray.reduce(average2, 0)); // 27.166667
```

Given the current index (and, obviously, having access to the array's length), we can do some trickery: in this case, we always sum values, but if we are at the end of the array, we also throw in a division so that the average value of the array will be returned. This is slick, but from the point of view of legibility, I'm certain we can agree that the first version we saw was more declarative and closer to the mathematical definition than this second version.



Getting the array and the index means that you could also turn the function into an impure one. Avoid this! Everybody who sees a `reduce()` call will automatically assume it's a pure function, and will surely introduce bugs when using it.

It would also be possible to modify `Array.prototype` to add the new function. Modifying prototypes is usually frowned upon because of the possibility of clashes with different libraries, at the very least. However, if you accept this idea, you could then write the following code:

```
Array.prototype.average = function() {
  return this.reduce((x, y) => x + y, 0) / this.length;
};

let myAvg = [22, 9, 60, 12, 4, 56].average(); // 27.166667
```

Do take note of the need for the outer `function()` (instead of an arrow function) because of the implicit handling of `this`, which wouldn't be bound otherwise. We have now extended the `Array.prototype` so that `average()` becomes globally available as a method.

Both this example and the previous one required calculating a single result, but it's possible to go beyond this and calculate several values in a single pass. Let's see how.

Calculating several values at once

What would you do if, instead of a single value, you needed to calculate two or more results? This would seem to be a case for providing a clear advantage for common loops, but there's a trick that you can use. Let's yet again revisit the average calculation. We might want to do it the old-fashioned way, by looping and at the same time summing and counting all numbers. Well, `reduce()` only lets you produce a single result, but there's no reason you can't return an object with as many fields as desired:

```
const average3 = arr => {
  const sumCount = arr.reduce(
    (accum, value) => ({sum: value + accum.sum, count: accum.count + 1}),
    {sum: 0, count: 0}
  );

  return sumCount.sum / sumCount.count;
};

console.log(average3(myArray)); // 27.166667
```

Examine the code carefully. We need two variables: one for the sum and one for the count of all numbers. We provide an object as the initial value for the accumulator, with two properties set to 0, and our reducing function updates those two properties.

By the way, using an object isn't the only option. You could also produce any other data structure; let's see an example with an array. The resemblance is pretty obvious:

```
const average4 = arr => {
  const sumCount = arr.reduce(
    (accum, value) => [accum[0] + value, accum[1] + 1],
    [0, 0]
  );
  return sumCount[0] / sumCount[1];
};

console.log(average4(myArray)); // 27.166667
```

To be frank, I think it's way more obscure than the solution with the object. Just consider this an alternative (not very recommendable) way of calculating many values at once!

We have now seen several examples of the use of `reduce()`, so it's high time to meet a variant of it, `reduceRight()`, which works in a very similar fashion.

Folding left and right

The complementary `reduceRight()` method works just as `reduce()` does, only starting at the end and looping until the beginning of the array. For many operations (such as the calculation of averages that we saw previously), this makes no difference, but there are some cases in which it will.

We shall be seeing a clear case of this in Chapter 8, *Connecting Functions – Pipelining and Composition*, when we compare pipelining and composition: let's go with a simpler example here:

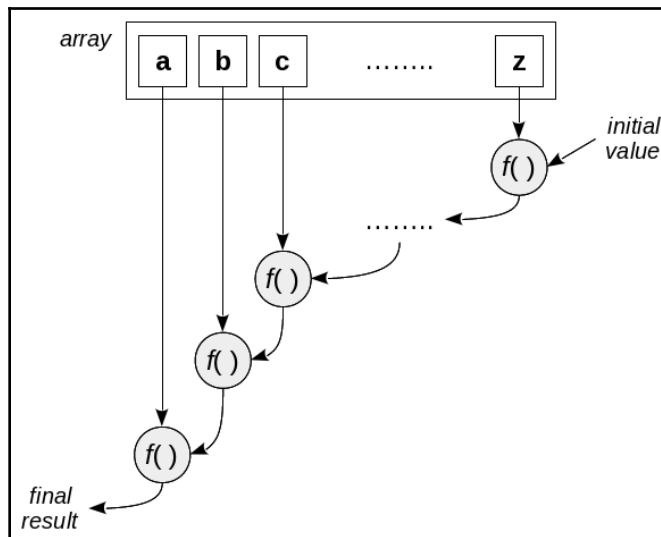


Figure 5.2: The `reduceRight()` operation works the same way as `reduce()`, but in reverse order



You can read more about `reduceRight()` at https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Array/ReduceRight.

Suppose that we want to implement a function to reverse a string. A solution could be to transform the string into an array by using `split()`, then reversing that array, and finally using `join()` to make it whole again:

```
const reverseString = str => {
  let arr = str.split("");
  arr.reverse();
  return arr.join("");
};

console.log(reverseString("MONTEVIDEO")); // OEDIVETNOM
```

This solution works (and yes, it can be shortened, but that's not the point here), but let's do it in another way, just to experiment with `reduceRight()`:

```
const reverseString2 = str =>
  str.split("").reduceRight((x, y) => x + y, "");

console.log(reverseString2("OEDIVETNOM")); // MONTEVIDEO
```



Given that the addition operator also works with strings, we could also have written `reduceRight(sum, "")`. And, if instead of the function we used, we had written `(x, y) => y+x`, the result would have been our original string; can you see why?

From the previous examples, you can also get an idea: if you first apply `reverse()` to an array and then use `reduce()`, the effect will be the same as if you had just applied `reduceRight()` to the original array. There is only one point to take into account: `reverse()` alters the given array, so you would be causing an unintended side effect by reversing the original array! The only way out would be first generating a copy of the array and only then doing the rest. Too much work; best to keep using `reduceRight()`!

However, we can draw another conclusion, showing a result we had foretold: it is possible, albeit more cumbersome, to use `reduce()` to simulate the same result as `reduceRight()`—and in later sections, we'll also use it to emulate the other functions in the chapter. Let's now move on to another common and powerful operation: mapping.

Applying an operation – map

Processing lists of elements and applying some kind of operation to each of them is a quite common pattern in computer programming. Writing loops that systematically go through all the elements of an array or collection, starting at the first and looping until finishing with the last, and performing some kind of process on each of them is a basic coding exercise, usually learned in the first days of all programming courses. We already saw one such kind of operation in the previous section with `reduce()` and `reduceRight()`; let's now turn to a new one, called `map()`.

In mathematics, a **map** is a transformation of elements from a **domain** into elements of a **codomain**. For example, you might transform numbers into strings or strings into numbers, but also numbers to numbers, or strings to strings: the important point is that you have a way to transform an element of the first **kind** or **domain** (think **type**, if it helps) into an element of the second kind, or **codomain**. In our case, this will mean taking the elements of an array and applying a function to each of them to produce a new array. In more computer-like terms, the map function transforms an array of inputs into an array of outputs.



Some more terminology: We would say that an array is a **functor** because it provides a mapping operation with some prespecified properties, which we shall see later. And, in category theory, which we'll talk about a little in Chapter 12, *Building Better Containers – Functional Data Types*, the mapping operation itself would be called a **morphism**.

The inner workings of the `map()` operation can be seen in Figure 5.3:

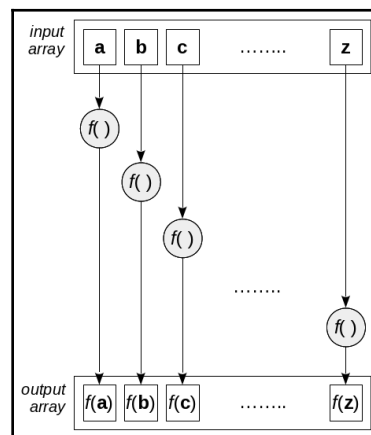


Figure 5.3: The `map()` operation transforms each element of the input array by applying a mapping function



The jQuery library provides a function, `$.map(array, callback)`, that is similar to the `map()` method. Be careful, though, for there are important differences. The jQuery function processes the undefined values of the array, while `map()` skips them. Also, if the applied function produces an array as its result, jQuery *flattens* it and adds each of its individual elements separately, while `map()` just includes those arrays in the result.

What are the advantages of using `map()` over using a straightforward loop?

- First, you don't have to write any loops, so that's one less possible source of bugs.
- Second, you don't even have to access the original array or the index position, even though they are there for you to use if you really need them.
- Lastly, a new array is produced, so your code is pure (though, of course, if you really want to produce side effects, you can!).



In JavaScript, `map()` is basically available only for arrays (you can read more about this at https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Array/map); however, in the *Extending current data types* section in Chapter 12, *Building Better Containers – Functional Data Types*, we will learn how to make it available for other basic types, such as numbers, Booleans, strings, and even functions. Also, libraries, such as LoDash, Underscore, and Ramda, provide similar functionalities.

There are only two caveats when using this:

- Always return something from your mapping function. If you forget this, then you'll just produce an undefined-filled array, because JavaScript always provides a default `return undefined` for all functions.
- If the input array elements are objects or arrays, and you include them in the output array, then JavaScript will still allow the original elements to be accessed.



There's an alternative way of doing `map()`: check the `Array.from()` method at https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Array/from and pay special attention to its second argument!

As we did earlier with `reduce()`, let's now look at some examples of the use of `map()` for common processes so that you'll better appreciate its power and convenience.

Extracting data from objects

Let's start with a simple example. Suppose that we have some geographic data (as shown in the following snippet) related to some South American countries and the coordinates (latitude and longitude) of their capitals. Let's say that we want to calculate the average position of those cities. (No, I don't have a clue why we'd want to do that.) How would we go about it?

```
const markers = [
  {name: "AR", lat: -34.6, lon: -58.4},
  {name: "BO", lat: -16.5, lon: -68.1},
  {name: "BR", lat: -15.8, lon: -47.9},
  {name: "CL", lat: -33.4, lon: -70.7},
  {name: "CO", lat:  4.6, lon: -74.0},
  {name: "EC", lat: -0.3, lon: -78.6},
  {name: "PE", lat: -12.0, lon: -77.0},
  {name: "PY", lat: -25.2, lon: -57.5},
  {name: "UY", lat: -34.9, lon: -56.2},
  {name: "VE", lat:  10.5, lon: -66.9},
];
```



In case you are wondering if and why all the data are negative, it's just because the countries shown here are all south of the Equator and west of the Greenwich Meridian; however, there are some South American countries with positive latitudes, such as Colombia or Venezuela, so not all have negative data. We'll come back to this question a little later when we study the `some()` and `every()` methods.

We would want to use our `average()` function (which we developed earlier in this chapter), but there is a problem: that function can only be applied to an array of *numbers*, and what we have here is an array of *objects*. We can, however, do a trick: we can focus on calculating the average latitude (we can deal with the longitude later, in a similar fashion). We can map each element of the array to just its latitude, and we would then have an appropriate input for `average()`. The solution would be something like the following:

```
let averageLat = average(markers.map(x => x.lat)); // -15.76
let averageLon = average(markers.map(x => x.lon)); // -65.53
```

If you had extended `Array.prototype`, you could then have written an equivalent version, in a different style, using `average()` as a method instead of a function:

```
let averageLat2 = markers.map(x => x.lat).average();
let averageLon2 = markers.map(x => x.lon).average();
```



We will be learning more about these styles in Chapter 8, *Connecting Functions – Pipelining and Composition*.

Mapping an array to extract data is powerful, but you must be careful. Let's now look at a case that *seems* right, but produces incorrect results!

Parsing numbers tacitly

Working with the `map` is usually far safer and simpler than looping by hand, but some edge cases may trip you up. Say you received an array of strings representing numeric values, and you wanted to parse them into actual numbers. Can you explain the following results?

```
["123.45", "67.8", "90"].map(parseFloat);  
// [123.45, 67.8, 90]  
  
["123.45", "-67.8", "90"].map(parseInt);  
// [123, NaN, NaN]
```

Let's analyze the results. When we used `parseFloat()` to get floating-point results, everything was okay; however, when we wanted to truncate the results to integer values, then the output was really awry, and weird `NaN` values appeared. What happened?

The answer lies in a problem with tacit programming. (We have already seen some uses of tacit programming in the *An unnecessary mistake* section of Chapter 3, *Starting Out with Functions – A Core Concept*, and we'll be seeing more in Chapter 8, *Connecting Functions – Pipelining and Composition*.) When you don't explicitly show the parameters to a function, it's easy to have some oversights. Look at the following code, which will lead us to the solution:

```
["123.45", "-67.8", "90"].map(x => parseFloat(x));  
// [123.45, -67.8, 90]  
  
["123.45", "-67.8", "90"].map(x => parseInt(x));  
// [123, -67, 90]
```

The reason for the unexpected behavior with `parseInt()` is that this function can also receive a second parameter—namely, the radix to be used when converting the string to a number. For instance, a call such as `parseInt("100010100001", 2)` will convert the binary number 100010100001 to decimal.



You can read more about `parseInt()` at https://developer.mozilla.org/en/docs/Web/JavaScript/Reference/Global_Objects/parseInt, where the radix parameter is explained in detail. You should always provide it because some browsers might interpret strings with a leading zero to be octal, which would once again produce unwanted results.

So what happens when we provide `parseInt()` to `map()`? Remember that `map()` calls your mapping function with three parameters: the array element value, its index, and the array itself. When `parseInt` receives these values, it ignores the array, but assumes that the provided index is actually a radix and NaN values are produced, since the original strings are not valid numbers in the given radix.

Okay, we saw that some functions can lead you astray when doing mapping, and you now know what to look out for. Let's keep enhancing the way we that we work, by using ranges to help you write code that would usually require a hand-written loop.

Working with ranges

Let's now turn to a helper function, which will come in handy for many uses. We want a `range(start, stop)` function that generates an array of numbers, with values ranging from `start` (inclusive) to `stop` (exclusive):

```
const range = (start, stop) =>
  new Array(stop - start).fill(0).map((v, i) => start + i);

let from2To6 = range(2, 7); // [2, 3, 4, 5, 6]
```

Why `fill(0)`? All undefined array elements are skipped by `map()`, so we need to fill them with something or our code will have no effect.



Libraries such as Underscore and Lodash provide a more powerful version of our `range` function, letting you go in ascending or descending order and also specifying the step to use—as in `_.range(0, -8, -2)`, which produces `[0, -2, -4, -6]`—but for our needs, the version we wrote is enough. Read the *Questions* section at the end of this chapter.

How can we use it? In the following section, we'll see some uses for controlled looping with `forEach()`, but we can redo our factorial function by applying `range()` and then `reduce()`. The idea of this is to simply generate all the numbers from 1 to n and then multiply them together:

```
const factorialByRange = n => range(1, n + 1).reduce((x, y) => x * y, 1);

factorialByRange(5); // 120
factorialByRange(3); // 6
```

It's important to check the border cases, but the function also works for zero; can you see why? The reason for this is that the produced range is empty (the call is `range(1, 1)`, which returns an empty array) and then `reduce()` doesn't do any calculations, and simply returns the initial value (one), which is correct.



In Chapter 7, *Transforming Functions - Currying and Partial Application*, we'll have the opportunity to use `range()` to generate source code; check out the *Currying with eval()* and *Partial application with eval()* sections.

You could use these numeric ranges to produce other kinds of ranges. For example, should you need an array with the alphabet, you could certainly (and tediously) write `["A", "B", "C" ... up to ... "X", "Y", "Z"]`. A simpler solution would be to generate a range with the ASCII codes for the alphabet and map those into letters:

```
const ALPHABET = range("A".charCodeAt(), "Z".charCodeAt() + 1).map(x =>
  String.fromCharCode(x)
);

// ["A", "B", "C", ... "X", "Y", "Z"]
```

Note the use of `charCodeAt()` to get the ASCII codes for the letters and `String.fromCharCode(x)` to transform the ASCII code back into a character.

Mapping is very important and quite often used, so let's now analyze how you could implement it on your own, which could help you develop code of your own for more complex cases.

Emulating `map()` with `reduce()`

Earlier in this chapter, we saw how `reduce()` could be used to implement `reduceRight()`. Now let's see how `reduce()` can also be used to provide a polyfill for `map()` (not that you will need it, because browsers usually provide both methods, but just to get more of an idea of what you can achieve with these tools).

Our own `myMap()` is a one-liner, but it can be hard to understand. The idea is that we apply the function to each element of the array and we `concat()` the result to (an initially empty) result array. When the loop finishes working with the input array, the result array will have the desired output values:

```
const myMap = (arr, fn) => arr.reduce((x, y) => x.concat(fn(y)), []);
```

Let's test this with an array and a simple function. We will use both the original `map()` method and our `myMap()`, and obviously the results should match!

```
const myArray = [22, 9, 60, 12, 4, 56];
const dup = x => 2 * x;

console.log(myArray.map(dup)); // [44, 18, 120, 24, 8, 112]
console.log(myMap(myArray, dup)); // [44, 18, 120, 24, 8, 112]
console.log(myArray); // [22, 9, 60, 12, 4, 56]
```

The first log shows the expected result, produced by `map()`. The second output gives the same result, so it seems that `myMap()` works! And the final output is just to check that the original input array wasn't modified in any way; mapping operations should always produce a new array.

All the previous examples in the chapter focused on simple arrays. But what happens if things get more complicated, say if you had to deal with an array whose elements were arrays themselves? Fortunately, there's a way out for that. Let's move on.

Dealing with arrays of arrays

So far, we have worked with an array of (single) values as an input, but what would happen if your input happened to be an array of arrays? If you consider this to be a farfetched case, there are many possible scenarios where this could apply:

- For some applications, you could have a table of distances, which in JavaScript would actually be an array of arrays: `distance[i][j]` would be the distance between points `i` and `j`. How could you find the maximum distance between any two points? With a common array, finding a maximum would be simple, but how do you deal with an array of arrays?
- A more complex example, also in a geographical vein: you could query a geographical API for cities matching a string and the response could be an array of countries, each with an array of states, each itself with an array of matching cities: an array of arrays of arrays!

In the first case, you could want to have a single array with all distances, and in the second, an array with all cities; how can you manage this? A new operation, **flattening**, is required; let's take a look.

Flattening an array

In ES2019, two operations were added to JavaScript: `flat()`, which we'll look at now, and `flatMap()`, which we'll look at a bit later. It's easier to show what they do than to explain—bear with me!



As often happens, not all browsers have been updated to include these new methods, and Microsoft's Internet Explorer and Edge (among others) are both deficient in this regard, so for web programming, you'll probably have to include a polyfill, or use some kind of implementation, which we'll be learning about soon. As usual, for updated compatibility data, check out the **Can I use?** site, in this case, at <https://caniuse.com/#feat=array-flat>.

The `flat()` method creates a new array, concatenating all elements of its subarrays to the desired level, which is, by default, 1:

```
const a = [[1, 2], [3, 4, [5, 6, 7]], 8, [[[9, 10]]]];

console.log(a.flat()); // or a.flat(1)
// [ 1, 2, 3, 4, [ 5, 6, 7 ], 8, [ 9, 10 ] ]

console.log(a.flat(2));
// [ 1, 2, 3, 4, 5, 6, 7, 8, [ 9, 10 ] ]

console.log(a.flat(Infinity));
// [ 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 ]
```

So how could we use this function to solve our problems? Using `flat()`, spreading, and `Math.max()` answers the first question (in a way that we saw back in Chapter 1, *Becoming Functional – Several Questions*, in the section called *Spread*; we could use the `maxArray()` function we wrote back then), and we can also use `reduce()` for variety. Suppose we have the following table of distances:

```
const distances = [
  [0, 20, 35, 40],
  [20, 0, 10, 50],
  [35, 10, 0, 30],
  [40, 50, 30, 0],
];
```

Then, we can find our maximum distance in a couple of ways: we either flatten the array, spread it, and use `Math.max()`, or flatten the array and use reducing to explicitly find the maximum:

```
const maxDist1 = Math.max(...distances.flat());
// 50

const maxDist2 = distances.flat().reduce((p, d) => Math.max(p, d), 0);
// also 50
```

Let's go back to the second question. Suppose we queried a geographical API for cities that have "LINCOLN" in their names and we got the following answer:

```
const apiAnswer = [
  {
    country: "AR",
    name: "Argentine",
    states: [
      {
        state: "1",
```

```
        name: "Buenos Aires",
        cities: [{city: 3846864, name: "Lincoln"}],
    },
],
},
{
    country: "GB",
    name: "Great Britain",
    states: [
        {
            state: "ENG",
            name: "England",
            cities: [{city: 2644487, name: "Lincoln"}],
        },
    ],
},
{
    country: "US",
    name: "United States of America",
    states: [
        {
            state: "CA",
            name: "California",
            cities: [{city: 5072006, name: "Lincoln"}],
        },
        .
        . several lines clipped out
        .
        {
            state: "IL",
            name: "Illinois",
            cities: [
                {city: 4899911, name: "Lincoln Park"},
                {city: 4899966, name: "Lincoln Square"},
            ],
        },
    ],
},
],
};
```

Extracting the list of cities can be done by applying `flatMap()` twice:

```
console.log(
    apiAnswer
    .map(x => x.states)
    .flat()
    .map(y => y.cities)
    .flat()
);
```



```
);  
  
/*  
[ { city: 3846864, name: 'Lincoln' },  
  { city: 2644487, name: 'Lincoln' },  
  { city: 5072006, name: 'Lincoln' },  
  { city: 8531960, name: 'Lincoln' },  
  { city: 4769608, name: 'Lincolnia' },  
  { city: 4999311, name: 'Lincoln Park' },  
  { city: 5072006, name: 'Lincoln' },  
  { city: 4899911, name: 'Lincoln Park' },  
  { city: 4899966, name: 'Lincoln Square' } ]  
*/
```

We have seen how to use `flat()` to flatten an array; let's now see how to use `flatMap()`, an interesting mixture of `flat()` and `map()`, to further streamline our coding, and even further shorten our preceding second solution!



Think this exercise wasn't hard enough and that its output was sort of lame? Try out exercise 5.8 for a more challenging version!

Mapping and flattening – flatMap()

Basically, what `flatMap()` does is first apply a `map()` function and then apply `flat()` to the result of the mapping operation. This is an interesting combination because it lets you produce a new array with a different number of elements. (With the usual `map()` operation, the output array will be exactly the same length as the input array). If your mapping operation produces an array with two or more elements, then the output array will include many output values, and if you produce an empty array, the output array will include fewer values.

Let's look at a (somehow nonsensical) example. Assume that we have a list of names, such as "Winston Spencer Churchill", "Abraham Lincoln", and "Socrates". Our rule is that if a name has several words, exclude the initial one (the first name, we assume) and separate the rest (last names), but if a name is a single word, just drop it (assuming the person had no last name):

```
const names = [  
  "Winston Spencer Churchill",  
  "Plato",  
  "Abraham Lincoln",  
  "Socrates",
```

```
"Charles Darwin",
];

const lastNames = names.flatMap(x => {
  const s = x.split(" ");
  return s.length === 1 ? [] : s.splice(1);
}); // [ 'Spencer', 'Churchill', 'Lincoln', 'Darwin' ]
```

As we can see, the output array has a different number of elements than the input one: just because of this, we could consider `flatMap()` to be an upgraded version of `map()`, even including some aspects of `filter()`, like when we excluded single names.

Let's now move on to a simple example. Keeping with the Lincolnian theme from the last section, let's count how many words are in Lincoln's Gettysburg address, given as an array of sentences.



Usually, this address is considered to be 272 words long, but the version I found doesn't produce that number! This may be because there are five manuscript copies of the address written by Lincoln himself, plus another version transcribed from shorthand notes taken at the event. In any case, I will leave the discrepancy to historians and stick to coding!

We can use `flatMap()` to split each sentence into an array of words and then just see the length of the flattened array:

```
const gettysburg = [
  "Four score and seven years ago our fathers brought forth, ",
  "on this continent, a new nation, conceived in liberty, and ",
  "dedicated to the proposition that all men are created equal.",
  "Now we are engaged in a great civil war, testing whether that ",
  "nation, or any nation so conceived and so dedicated, can long ",
  "endure.",
  "We are met on a great battle field of that war.",
  "We have come to dedicate a portion of that field, as a final ",
  "resting place for those who here gave their lives, that that ",
  "nation might live.",
  "It is altogether fitting and proper that we should do this.",
  "But, in a larger sense, we cannot dedicate, we cannot consecrate, ",
  "we cannot hallow, this ground.",
  "The brave men, living and dead, who struggled here, have ",
  "consecrated it far above our poor power to add or detract.",
  "The world will little note nor long remember what we say here, ",
  "but it can never forget what they did here.",
  "It is for us the living, rather, to be dedicated here to the ",
  "unfinished work which they who fought here have thus far so nobly ",
  "advanced.",
  "It is rather for us to be here dedicated to the great task ",
```

```
    "remaining before us— that from these honored dead we take increased ",
    "devotion to that cause for which they here gave the last full ",
    "measure of devotion— that we here highly resolve that these dead ",
    "shall not have died in vain— that this nation, under God, shall have ",
    "a new birth of freedom- and that government of the people, by the ",
    "people, for the people, shall not perish from the earth.",
  ];

  console.log(gettysburg.flatMap(s => s.split(" ")).length);
```

Let's go back to the problem with the cities. If we notice that each `map()` was followed by a `flat()`, an alternative solution is immediately obvious. Compare this solution with the one we wrote in the *Flattening an array* section; it's essentially the same, but conflates each `map()` with its following `flat()`:

```
console.log(apiAnswer.flatMap(x => x.states).flatMap(y => y.cities));
/*
[ { city: 3846864, name: 'Lincoln' },
  { city: 2644487, name: 'Lincoln' },
  { city: 5072006, name: 'Lincoln' },
  { city: 8531960, name: 'Lincoln' },
  { city: 4769608, name: 'Lincolnia' },
  { city: 4999311, name: 'Lincoln Park' },
  { city: 5072006, name: 'Lincoln' },
  { city: 4899911, name: 'Lincoln Park' },
  { city: 4899966, name: 'Lincoln Square' } ]
*/
```

We have now seen the new operations. Let's now learn how to emulate them, should you not have them readily available.



It's perfectly possible to solve the problems in this section without using any sort of mapping, but that wouldn't do as a proper example for this section! See exercise 5.9 for an alternative to the word counting problem.

Emulating `flat()` and `flatMap()`

We have already seen how `reduce()` could be used to emulate `map()`. Let's now see how to work out equivalents for `flat()` and `flatMap()` to get more practice. We'll also throw in a recursive version, a topic we'll come back to in Chapter 9, *Designing Functions – Recursion*. As was mentioned earlier, we are not aiming for the fastest or smallest or any particular version of the code; rather, we want to focus on using the concepts we've been looking at in this book.

Totally flattening an array can be done with a recursive call. We use `reduce()` to process the array element by element, and if an element happens to be an array, we recursively flatten it:

```
const flatAll = arr =>
  arr.reduce((f, v) => f.concat(Array.isArray(v) ? flatAll(v) : v), []);
```

Flattening an array to a given level (not infinity; let's leave that for later) is easy if you can first flatten an array one level. We can do this either by using spreading or with `reduce`. Let's write a `flatOne()` function that flattens just a single level of an array. There are two versions of this; pick whichever you prefer:

```
const flatOne1 = arr => [].concat(...arr);

const flatOne2 = arr => arr.reduce((f, v) => f.concat(v), []);
```

Using either of these two functions, we can manage to flatten an array of several levels, and we can do this in two different ways. Our two versions of a `flat()` function use our previous `flatOne()` and `flatAll()` functions, but the first one only uses common looping, while the second one works in a fully recursive way. Which one do you prefer?

```
const flat1 = (arr, n = 1) => {
  if (n === Infinity) {
    return flatAll(arr);
  } else {
    let result = arr;
    range(0, n).forEach(() => {
      result = flatOne(result);
    });
    return result;
  }
};

const flat2 = (arr, n = 1) =>
  n === Infinity
  ? flatAll(arr)
  : n === 1
  ? flatOne(arr)
  : flat2(flatOne(arr), n - 1);
```

Personally, I think the recursive one is nicer, and more aligned with the theme of this book, but it's up to you, really (though if you don't feel comfortable with the ternary operator, then the recursive version is definitely not for you!).

If you wish to polyfill these functions (despite our suggestions not to), it's not complex, and is similar to what we did some pages back with the `average()` method. I took care not to create any extra methods:

```
if (!Array.prototype.flat) {
  Array.prototype.flat = function(n = 1) {
    this.flatAllX = () =>
      this.reduce(
        (f, v) => f.concat(Array.isArray(v) ? v.flat(Infinity) : v),
        []
      );

    this.flatOneX = () => this.reduce((f, v) => f.concat(v), []);

    return n === Infinity
      ? this.flatAllX()
      : n === 1
      ? this.flatOneX()
      : this.flatOneX().flat(n - 1);
  };
}
```

Our `flatOneX()` and `flatAllX()` methods are just copies of what we developed before, and you'll recognize the code of our previous `flat2()` function at the end of our implementation.

Finally, emulating `flatMap()` is simplicity itself, and we can just skip it because it's just a matter of applying `map()` first, and then `flat()`; no big deal!

We have seen how to work with arrays in several ways, but sometimes what you need isn't really well served by any of the functions we have seen. Let's move on to more general ways of doing loops, for greater power.

More general looping

The preceding examples that we've seen simply loop through arrays, doing some work. However, sometimes you need to do a loop, but the required process doesn't really fit `map()` or `reduce()`. So what can be done in such cases? There is a `forEach()` method that can help.



Read more about the specification of the `forEach()` method at https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Array/forEach.

You must provide a callback that will receive the value, the index, and the array on which you are operating. (The last two arguments are optional.) JavaScript will take care of the loop control, and you can do whatever you want at each step. For instance, we can program an object copy method by using some `Object` methods to copy the source object attributes one at a time and generate a new object:

```
const objCopy = obj => {
  let copy = Object.create(Object.getPrototypeOf(obj));
  Object.getOwnPropertyNames(obj).forEach(prop =>
    Object.defineProperty(
      copy,
      prop,
      Object.getOwnPropertyDescriptor(obj, prop)
    )
  );
  return copy;
};

const myObj = {fk: 22, st: 12, desc: "couple"};
const myCopy = objCopy(myObj);
console.log(myObj, myCopy); // {fk: 22, st: 12, desc: "couple"}, twice
```



Yes, of course we could have written `myCopy={...myObj}`, but where's the fun in that? Okay, it would be better, but I needed a nice example to use `forEach()` with. Sorry about that! Also, there are some hidden inconveniences in that code, which we'll explain in Chapter 10, *Ensuring Purity – Immutability*, when we try to get really frozen, unmodifiable objects. Just a hint: the new object may share values with the old one because we have a *shallow* copy, not a *deep* one. We'll learn more about this later in the book.

If you use the `range()` function that we defined previously, you can also perform common loops of the `for(i=0; i<10; i++)` variety. We might write yet another version of factorial (!) using that:

```
const factorial4 = n => {
  let result = 1;
  range(1, n + 1).forEach(v => (result *= v));
  return result;
};

console.log(factorial4(5)); // 120
```

This definition of factorial really matches the usual description: it generates all the numbers from 1 to n inclusive and multiplies them—simple!



For greater generality, you might want to expand `range()` so it can generate ascending and descending ranges of values, possibly also stepping by a number other than 1. This would practically allow you to replace all the loops in your code with `forEach()` loops.

At this point, we have seen many ways of processing arrays to generate results, but there are other objectives that might interest you, so let's now move on to logical functions, which will also simplify your coding needs.

Logical higher-order functions

Up to now, we have been using higher-order functions to produce new results, but there are also some other functions that produce logical results by applying a predicate to all the elements of an array. (By the way, we'll be seeing much more about higher-order functions in the next chapter.)



A bit of terminology: the word **predicate** can be used in several senses (as in *predicate logic*), but for us, in computer science, it has the meaning of *a function that returns true or false*. Okay, this isn't a very formal definition, but it's enough for our needs. For example, saying that we will filter an array depending on a predicate just means that we get to decide which elements are included or excluded depending on the result of the predicate.

Using these functions implies that your code will become shorter: you can, with a single line of code, get the results corresponding to the whole set of values.

Filtering an array

A common need that we will encounter is to filter the elements of an array according to a certain condition. The `filter()` method lets you inspect each element of an array in the same fashion as `map()`. The difference is that instead of producing a new element, the result of your function determines whether the input value will be kept in the output (if the function returned `true`) or if it will be skipped (if the function returned `false`). Also similar to `map()`, `filter()` doesn't alter the original array, but rather returns a new array with the chosen items.

See *Figure 5.4* for a diagram showing the input and output:

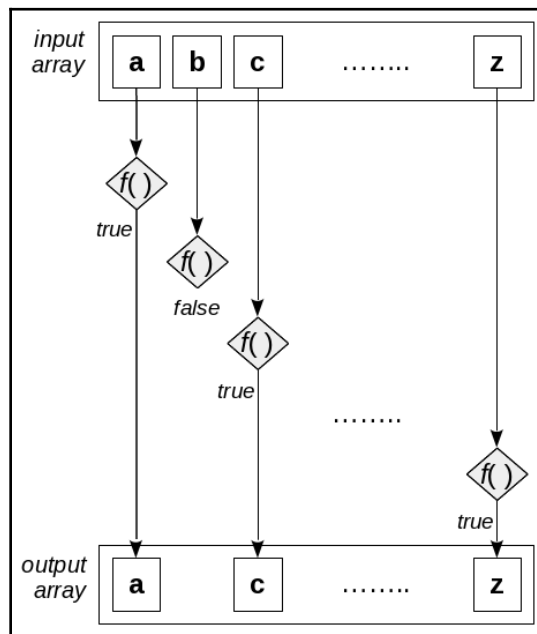


Figure 5.4: The `filter()` method picks the elements of an array that satisfy a given predicate



Read more on the `filter()` function at https://developer.mozilla.org/en/docs/Web/JavaScript/Reference/Global_Objects/Array/filter.

There are a couple of things to remember when filtering an array:

- **Always return something from your predicate:** If you forget to include a `return`, the function will implicitly return `undefined`, and since that's a *falsy* value, the output will be an empty array.
- **The copy that is made is shallow:** If the input array elements are objects or arrays, then the original elements will still be accessible.

Let's get into more detail by first seeing a practical example of `filter()` and then moving on to look at how you could implement that functionality on your own by using `reduce()`.

A `reduce()` example

Let's look at a practical example. Suppose a service has returned a JSON object, which itself has an array of objects containing an `account id` and the `account balance`. How can we get the list of IDs that are *in the red*, with a negative balance? The input data could be as follows:

```
const serviceResult = {
  accountsData: [
    {
      id: "F220960K",
      balance: 1024,
    },
    {
      id: "S120456T",
      balance: 2260,
    },
    {
      id: "J140793A",
      balance: -38,
    },
    {
      id: "M120396V",
      balance: -114,
    },
    {
      id: "A120289L",
      balance: 55000,
    },
  ],
};
```

We could get the delinquent accounts with something like the following. You can check that the value of the `delinquent` variable correctly includes the two IDs of accounts with a negative balance:

```
const delinquent = serviceResult.accountsData.filter(v => v.balance < 0);  
  
console.log(delinquent); // two objects, with id's J140793A and M120396V
```

By the way, given that the filtering operation produced yet another array, if you just wanted the accounts IDs, you could get them by mapping the output to just get the ID field:

```
const delinquentIds = delinquent.map(v => v.id);
```

And if you didn't care for the intermediate result, a one-liner would have done as well:

```
const delinquentIds2 = serviceResult.accountsData  
  .filter(v => v.balance < 0)  
  .map(v => v.id);
```

Filtering is a very useful function, so now, to get a better handle on it, let's see how you could emulate it, which you could even use as a basis for more sophisticated, powerful functions of your own.

Emulating `filter()` with `reduce()`

As we did before with `map()`, we can also create our own version of `filter()` by using `reduce()`. The idea is similar: loop through all the elements of the input array, apply the predicate to it, and if the result is `true`, add the original element to the output array. When the loop is done, the output array will only have those elements for which the predicate was `true`:

```
const myFilter = (arr, fn) =>  
  arr.reduce((x, y) => (fn(y) ? x.concat(y) : x), []);
```

We can quickly see that our function works as expected:

```
console.log(myFilter(serviceResult.accountsData, v => v.balance < 0));  
// two objects, with id's J140793A and M120396V
```

The output is the same pair of accounts that we saw earlier in this section.

Searching an array

Sometimes, instead of filtering all the elements of an array, you want to find an element that satisfies a given predicate. There are a couple of functions that can be used for this, depending on your specific needs:

- `find()` searches through the array and returns the value of the first element that satisfies a given condition, or `undefined` if no such element is found
- `findIndex()` performs a similar task, but instead of returning an element, it returns the index of the first element in the array that satisfies the condition, or `-1` if none were found

The similarity to `includes()` and `indexOf()` is clear; these functions search for a specific value instead of an element that satisfies a more general condition. We can easily write equivalent one-liners:

```
arr.includes(value); // arr.find(v => v === value)
arr.indexOf(value); // arr.findIndex(v => v === value)
```

Going back to the geographic data we used earlier, we could easily find a given country by using the `find()` method. For instance, let's get data for Brazil ("BR"); it just takes a single line of code:

```
markers = [
  {name: "UY", lat: -34.9, lon: -56.2},
  {name: "AR", lat: -34.6, lon: -58.4},
  {name: "BR", lat: -15.8, lon: -47.9},
  //...
  {name: "BO", lat: -16.5, lon: -68.1}
];

let brazilData = markers.find(v => v.name === "BR");
// {name:"BR", lat:-15.8, lon:-47.9}
```

We couldn't use the simpler `includes()` method because we have to delve into the object to get the field we want. If we wanted the position of the country in the array, we would have used `findIndex()`:

```
let brazilIndex = markers.findIndex(v => v.name === "BR"); // 2

let mexicoIndex = markers.findIndex(v => v.name === "MX"); // -1
```

Okay, this was simple! What about a special case, which could even be a trick interview question? Read on!

A special search case

Now, for variety, a little quiz. Suppose you had an array of numbers and wanted to run a sanity check, studying whether any of them was `NaN`. How would you do this? A tip: don't try checking the types of the array elements—even though `NaN` stands for **not a number**, `typeof NaN === "number"`. You'll get a surprising result if you try to do the search in an *obvious way*:

```
[1, 2, NaN, 4].findIndex(x => x === NaN); // -1
```

What's going on here? It's a bit of interesting JavaScript trivia: `NaN` is the only value that isn't equal to itself. Should you need to look for `NaN`, you'll have to use the new `isNaN()` function, as follows:

```
[1, 2, NaN, 4].findIndex(x => isNaN(x)); // 2
```

This was a particular case, but worth knowing about; I actually had to deal with this case once! Now, let's continue as we have done previously, by emulating the searching methods with `reduce()` so that we can see more examples of the power of that function.

Emulating `find()` and `findIndex()` with `reduce()`

As with the other methods, let's finish this section by studying how to implement the methods we showed by using the omnipotent `reduce()`. This is a good exercise to get accustomed to working with higher-order functions, even if you are never going to actually use these polyfills!

The `find()` function requires a bit of work. We start the search with an undefined value, and if we find an array element so that the predicate is `true`, we change the accumulated value to that of the array:

```
arr.find(fn);  
// arr.reduce((x, y) => (x === undefined && fn(y) ? y : x), undefined);
```

For `findIndex()`, we must remember that the callback function receives the accumulated value, the array's current element, and the index of the current element, but other than that, the equivalent expression is quite similar to the one for `find()`; comparing them is worth the time:

```
arr.findIndex(fn);  
// arr.reduce((x, y, i) => (x == -1 && fn(y) ? i : x), -1);
```

The initial accumulated value is `-1` here, which will be the returned value if no element fulfills the predicate. Whenever the accumulated value is still `-1`, but we find an element that satisfies the predicate, we change the accumulated value to the array index.

Okay, we are now done with searches: let's move on to consider higher-level predicates that will simplify testing arrays for a condition, but always in the declarative style we've been using so far.

Higher-level predicates – some, every

The last functions we are going to consider greatly simplify going through arrays to test for conditions. These functions are as follows:

- `every()`, which is `true` if and only if *every* element in the array satisfies a given predicate
- `some()`, which is `true` if at least *one* element in the array satisfies the predicate

For example, we could easily check our hypothesis about all the countries having negative coordinates:

```
markers.every(v => v.lat < 0 && v.lon < 0); // false
markers.some(v => v.lat < 0 && v.lon < 0); // true
```

If we want to find equivalents to these two functions in terms of `reduce()`, the two alternatives show nice symmetry:

```
arr.every(fn);
// arr.reduce((x, y) => x && fn(y), true);

arr.some(fn);
// arr.reduce((x, y) => x || fn(y), false);
```

The first folding operation evaluates `fn(y)`, and ANDs the result with the previous tests; the only way the final result will be `true` is if every test comes out `true`. The second folding operation is similar, but ORs the result with the previous results, and will produce `true`, unless every test comes out `false`.



In terms of Boolean algebra, we would say that the alternative formulations for `every()` and `some()` exhibit duality. This duality is the same kind that appears in the expressions `x === x && true` and `x === x || false`; if `x` is a Boolean value, and we exchange `&&` and `||`, and also `true` and `false`, then we transform one expression into the other, and both are valid.

In this section, we saw how to check for a given Boolean condition. Let's finish by seeing how to check a negative condition by inventing a method of our own.

Checking negatives – none

If you wanted, you could also define `none()` as the complement of `every()`. This new function would be `true` only if none of the elements of the array satisfied the given predicate. The simplest way of coding this would be by noting that if no elements satisfy the condition, then all elements satisfy the negation of the condition:

```
const none = (arr, fn) => arr.every(v => !fn(v));
```

If you want, you can turn it into a method by modifying the array prototype, as we saw earlier; it's still a bad practice, but it's what we have until we start looking into better methods for composing and chaining functions, as we will see in Chapter 8, *Connecting Functions – Pipelining and Composition*:

```
Array.prototype.none = function(fn) {  
  return this.every(v => !fn(v));  
};
```

We had to use `function()` instead of an arrow function for the same reasons we saw earlier; in this sort of case, we need `this` to be correctly assigned. Other than that, it's simple coding, and we now have a `none()` method available for all arrays.



In Chapter 6, *Producing Functions – Higher-Order Functions*, we will see other ways of negating a function by writing an appropriate higher-order function of our own.

In this and the preceding section, we worked with everyday problems, and we saw how to solve them in a declarative way. However, things change a bit when you start working with `async` functions. New solutions will be needed, as we will see in the following section.

Working with `async` functions

All the examples and code that we studied in the previous sections were meant to be used with common functions, specifically meaning not `async` ones. When you want to do mapping, filtering, reducing, and so on, but the function you use is an `async` one, the results may surprise you. In order to simplify our work and not have to deal with actual API calls, let's create a `fakeAPI(delay, value)` function that will just delay a while and then return the given value:

```
const fakeAPI = (delay, value) =>
  new Promise(resolve => setTimeout(() => resolve(value), delay));
```

Let's also have a function to display what `fakeAPI()` returns, so we can see that things are working as expected:

```
const useResult = x => console.log(new Date(), x);
```

We are using the modern `async/await` features to simplify our code:

```
(async () => {
  console.log("START");
  console.log(new Date());
  const result = await fakeAPI(1000, 229);
  useResult(result);
  console.log("END");
})();
/*
START
2019-10-13T19:11:56.209Z
2019-10-13T19:11:57.214Z 229
END
*/
```

The results are previsible: we get the `START` text, then about 1 second (1000 milliseconds) later, we get the result of the fake API call (229), and finally the `END` text. What could go wrong?



Why are we using the *immediate invocation* pattern that we saw in Chapter 3, *Starting Out with Functions – A Core Concept*? The reason is that you can only use `await` within an `async` function. There is a proposal that will allow the use of `await` with top-level modules (see <https://v8.dev/features/top-level-await> for more on this), but it hasn't made its way into JavaScript yet, and it applies to modules only, not general scripts.

The key problem is that all the functions we saw earlier in this chapter are not `async aware`, so they won't really work as you'd expect. Let's start looking at this.

Some strange behaviors

Let's start with a simple quiz: are the results what you expected? Let's look at a couple of examples of code involving `async` calls and maybe we'll see some unexpected results. First, let's look at a common straightforward sequence of `async` calls:

```
(async () => {
  console.log("START SEQUENCE");

  const x1 = await fakeAPI(1000, 1);
  useResult(x1);
  const x2 = await fakeAPI(2000, 2);
  useResult(x2);
  const x3 = await fakeAPI(3000, 3);
  useResult(x3);
  const x4 = await fakeAPI(4000, 4);
  useResult(x4);

  console.log("END SEQUENCE");
})();
```

If you run this code, you'll get the following results, which are surely what you would expect—a `START SEQUENCE` text, four individual lines with the results of the fake API calls, and a final `END SEQUENCE` text. Nothing special here—everything is fine!

```
START SEQUENCE
2019-10-12T13:38:42.367Z 1
2019-10-12T13:38:43.375Z 2
2019-10-12T13:38:44.878Z 3
2019-10-12T13:38:46.880Z 4
END SEQUENCE
```


Now let's go for an alternative second version, which you'd probably expect to be equivalent to the first one. The only difference here is that here we are using looping to do the four API calls; it should be the same, shouldn't it? (We could also have used a `forEach` loop with the `range()` function that we saw earlier, but that makes no difference.) I kept using an IIFE, though in this particular case it wasn't needed; can you see why?

```
(() => {
  console.log("START FOREACH");

  [1, 2, 3, 4].forEach(async n => {
    const x = await fakeAPI(n * 1000, n);
    useResult(x);
  });

  console.log("END FOREACH");
})();
```

This piece of code certainly looks equivalent to the first one, but it produces something quite different!

```
START FOREACH
END FOREACH
2019-10-12T13:34:57.876Z 1
2019-10-12T13:34:58.383Z 2
2019-10-12T13:34:58.874Z 3
2019-10-12T13:34:59.374Z 4
```

The `END FOREACH` text appears before the results of the API calls. What's happening? The answer is what we mentioned before: methods similar to `forEach` and the like are meant to be used with common, sync function calls, and behave strangely with `async` function calls. The key concept is that `async` functions always return promises, so that after getting the `START FOREACH` text, the loop is actually creating four promises (which will get resolved at some time), but without waiting for them, and our code goes on to print the `END FOREACH` text.



You can verify this yourself by looking at the polyfill for `reduce()` at https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Array/Reduce#Polyfill.

The problem is not only with `forEach()`, but rather affects all similar methods as well. Let's see how we can work around this situation and write `async`-aware functions to let us keep working in a declarative fashion, as we did earlier in the chapter.

Async-ready looping

If we cannot directly use methods such as `forEach()`, `map()`, and the like, we'll have to develop new versions of our own. Let's see how to achieve this.

Looping over async calls

Since `async` calls return promises, we can emulate `forEach()` with `reduce()` by starting with a resolved promise and chaining to it the promises for each value in the array. The `then()` methods will be called in the right order, so the results will be correct. The following piece of code manages to get the right, expected results:

```
const forEachAsync = (arr, fn) =>
  arr.reduce(
    (promise, value) => promise.then(() => fn(value)),
    Promise.resolve()
  );

(async () => {
  console.log("START FOREACH VIA REDUCE");
  await forEachAsync([1, 2, 3, 4], async n => {
    const x = await fakeAPI(n * 1000, n);
    useResult(x);
  });
  console.log("END FOREACH VIA REDUCE");
})();

/*
START FOREACH VIA REDUCE
2019-10-13T20:02:23.437Z 1
2019-10-13T20:02:24.446Z 2
2019-10-13T20:02:25.949Z 3
2019-10-13T20:02:27.952Z 4
END FOREACH VIA REDUCE
*/
```

As `forEachAsync()` returns a promise, we mustn't forget to `await` it before showing the final text message. Other than not forgetting all the `await` statements, the code is pretty much similar to what we build using `forEach()`, with the crucial difference that this does work as expected!

Mapping async calls

Can we use the other functions? Writing `mapAsync()`, a version of `map` that can work with an async mapping function, is simple because you can take advantage of `Promise.all()` to create a promise out of an array of promises:

```
const mapAsync = (arr, fn) => Promise.all(arr.map(fn));

(async () => {
  console.log("START MAP");

  const mapped = await mapAsync([1, 2, 3, 4], async n => {
    const x = await fakeAPI(n * 1000, n);
    return x;
  });

  useResult(mapped);
  console.log("END MAP");
})();

/*
START MAP
2019-10-13T20:06:21.149Z [ 1, 2, 3, 4 ]
END MAP
*/
```

The structure of the solution is similar to the `forEachAsync()` code. As before, we must remember to `await` the result of `mapAsync()` before continuing the process. Other than that, the logic is straightforward, and the results are as expected.

Filtering with async calls

Filtering with an `async` function is a tad more complicated. We will have to use `mapAsync()` to produce an array of `true/false` values, and then use the standard `filter()` method to pick values out of the original array depending on what the `async` filtering function returned. Let's try out a simple example, calling the API and accepting only even results by means of a `fakeFilter()` function, which, for our example, accepts even numbers and rejects odd ones:

```
const filterAsync = (arr, fn) =>
  mapAsync(arr, fn).then(arr2 => arr.filter((v, i) => Boolean(arr2[i])));

const fakeFilter = (value) =>
  new Promise(resolve =>
    setTimeout(() => resolve(value % 2 === 0), 1000)
  );

(async () => {
  console.log("START FILTER");

  const filtered = await filterAsync([1, 2, 3, 4], async n => {
    const x = await fakeFilter(n);
    return x;
  });

  useResult(filtered);
  console.log("END FILTER");
})();
/*
START FILTER
2019-10-13T21:24:36.478Z [ 2, 4 ]
END FILTER
*/
```

Note that the result of the mapping of `async` calls is a Boolean array (`arr2`), which we then use with `filter()` to select elements from the original array of values (`arr`); this can be tricky to understand!

Reducing async calls

Finally, finding an equivalent for `reduce()` is a bit more complex, but after the other functions that we've seen, not so much. The key idea is the same as in `forEachAsync`: each function call will return a promise, which must be awaited in order to update the accumulator in an upcoming `then()`. To do the reducing, let's have a `fakeSum()` `async` function that will just sum the API-returned values:

```
const reduceAsync = (arr, fn, init) =>
  Promise.resolve(init).then(accum =>
    forEachAsync(arr, async (v, i) => {
      accum = await fn(accum, v, i);
    }).then(() => accum)
  );

const fakeSum = (value1, value2) =>
  new Promise(resolve => setTimeout(() => resolve(value1 + value2), 1000));

(async () => {
  console.log("START REDUCE");

  const summed = await reduceAsync(
    [1, 2, 3, 4],
    async (_accum, n) => {
      const accum = await _accum;
      const x = await fakeSum(accum, n);
      useResult(`accumulator=${accum} value=${x} `);
      return x;
    },
    0
  );

  useResult(summed);
  console.log("END REDUCE");
})();
/*
START REDUCE
2019-10-13T21:29:01.841Z 'accumulator=0 value=1 '
2019-10-13T21:29:02.846Z 'accumulator=1 value=3 '
2019-10-13T21:29:03.847Z 'accumulator=3 value=6 '
2019-10-13T21:29:04.849Z 'accumulator=6 value=10 '
2019-10-13T21:29:04.849Z 10
END REDUCE
*/
```

Note the important detail: in our reducing function, we must `await` the value of the accumulator, and only afterward `await` the result of our `async` function. This is an important point, which you must not miss: since we are reducing in `async` fashion, getting the accumulator is also an `async` matter, so we get to `await` both the accumulator and the new API call.

By looking at these equivalents, we have seen that `async` functions, despite producing problems with the usual declarative methods that we studied at the beginning of the chapter, may also be handled by similar new functions of our own, so we can keep the new style even for these cases. Even if we have to use a somewhat different set of functions, your code will still be declarative, tighter, and clearer; an all-round win!

Summary

In this chapter, we started working with higher-order functions in order to show a more declarative way of working, with shorter, more expressive code. We went over several operations: we used `.reduce()` and `.reduceRight()` to get a single result from an array, `.map()` to apply a function to each element of an array, `.forEach()` to simplify looping, `flat()` and `flatMap()` to work with arrays of arrays, `.filter()` to pick elements from an array, `.find()` and `.findIndex()` to search in the arrays, and `.every()` and `.some()` to verify general logic conditions. Furthermore, we considered some unexpected situations that happen when you deal with `async` functions and we wrote special functions for those cases.

In Chapter 6, *Producing Functions – Higher-Order Functions*, we will continue working with higher-order functions, but we will then turn to writing our own ones to gain more expressive power for our coding.

Questions

5.1. **Filtering... but what?** Suppose you have an array, called `someArray`, and you apply the following `.filter()` to it, which at first sight doesn't even look like valid JavaScript code. What will be in the new array and why?

```
let newArray = someArray.filter(Boolean);
```

5.2. Generating HTML code, with restrictions: Using the `filter()...map()...reduce()` sequence is quite common (even allowing that sometimes you won't use all three), and we'll come back to this in the *Functional design patterns* section in Chapter 11, *Implementing Design Patterns – The Functional Way*. The problem here is how to use those functions (and no others!) to produce an unordered list of elements (`...`) that can later be used onscreen. Your input is an array of objects like the following (does the list of characters date me?) and you must produce a list of each name that corresponds to chess or checkers players:

```
var characters = [
  {name: "Fred", plays: "bowling"},
  {name: "Barney", plays: "chess"},
  {name: "Wilma", plays: "bridge"},
  {name: "Betty", plays: "checkers"},
  .
  .
  .
  {name: "Pebbles", plays: "chess"}
];
```

The output would be something like the following (though it doesn't matter if you don't generate spaces and indentation). It would be easier if you could use, say, `.join()`, but in this case, it won't be allowed; only the three mentioned functions can be used:

```
<div>
  <ul>
    <li>Barney</li>
    <li>Betty</li>
    .
    .
    .
    <li>Pebbles</li>
  </ul>
</div>;
```

5.3. More formal testing: In some of the preceding examples, such as those in the *Emulating map() with reduce()* section, we didn't write actual unit tests, but instead were satisfied with doing some console logging. Can you write appropriate unit tests instead?

5.4. Ranging far and wide: The `range()` function that we saw here can have many uses, but lacks a bit in generality. Can you expand it to allow for, say, descending ranges, as in `range(10, 1)`? (What should the last number in the range be?) And could you also allow for a step size to be included to specify the difference between successive numbers in the range? With this, `range(1, 10, 2)` would produce `[1, 3, 5, 7, 9]`.

5.5. Doing the alphabet: What would have happened in the *Working with ranges* section if instead of writing `map(x => String.fromCharCode(x))`, you had simply written `map(String.fromCharCode)`? Can you explain the different behavior? Hint: we already saw a similar problem elsewhere in this chapter.

5.6. Producing a CSV: In a certain application, you want to enable the user to download a set of data as a **comma-separated value (CSV)** file by using a data URI. (You can read more about this at https://developer.mozilla.org/en-US/docs/Web/HTTP/Basics_of_HTTP/Data_URIs/.) Of course, the first problem is producing the CSV itself! Assume that you have an array of arrays of numeric values, as shown in the following snippet, and write a function that will transform that structure into a CSV string that you will then be able to plug into the URI. As usual, `\n` stands for the newline character:

```
let myData = [[1, 2, 3, 4], [5, 6, 7, 8], [9, 10, 11, 12]];
let myCSV = dataToCsv(myData); // "1,2,3,4\n5,6,7,8\n9,10,11,12\n"
```

5.7. An empty question: Check that `flat1()` and `flat2()` properly work if applied to arrays with empty places, such as `[22, , 9, , , 60, ,]`. Why do they work?

5.8. Producing better output: Modify the cities query to produce a list of strings that includes not only the name of the city, but the state and country as well.

5.9. Old-style code only! Can you rewrite the word-counting solution without using any mapping or reducing at all? This is more of a JavaScript problem than a functional programming one, but why not?

5.10. Async chaining: Our `...Async()` functions are not methods; can you modify them and add them to `Array.prototype` so that we can write, for example, `[1, 2, 3, 4].mapAsync(...)`? And by the way, will chaining work with your solution?

5.11. Missing equivalents: We wrote `forEach()`, `map()`, `filter()`, and `reduce()` async equivalents, but we didn't do the same for `find()`, `findIndex()`, `some()`, and `every()`; can you do that?

6

Producing Functions - Higher-Order Functions

In [Chapter 5, *Programming Declaratively – A Better Style*](#), we worked with some predefined higher-order functions and were able to see how their usage lets us write declarative code so that we can gain in understandability as well as in compactness. In this chapter, we are going to go further in the direction of higher-order functions and develop our own. We can roughly classify the kinds of results that we are going to get into three groups:

- **Wrapped functions:** These keep their original functionality while adding some kind of new feature. In this group, we can consider *logging* (adding log production capacity to any function), *timing* (producing time and performance data for a given function), and *memoization* (this caches results to avoid future rework).
- **Altered functions:** These differ in some key points from their original versions. Here, we can include the `once()` function (we wrote it in [Chapter 2, *Thinking Functionally – A First Example*](#)), which changes the original function so that it only runs once, functions such as `not()` or `invert()`, which alter what the function returns, and arity-related conversions, which produce a new function with a fixed number of parameters.
- **Other productions:** These provide new operations, turn functions into promises, allow enhanced search functions, or decouple methods from objects so that we can use them in other contexts as if they were common functions. We shall leave a special case – transducers – for [Chapter 8, *Connecting Functions – Pipelining and Composition*](#).

Wrapping functions – keeping behavior

In this section, we'll consider some higher-order functions that provide a *wrapper* around other functions to enhance them in some way but without altering their original objective. In terms of *design patterns* (which we'll be revisiting in Chapter 11, *Implementing Design Patterns – The Functional Way*), we can also speak of *decorators*. This pattern is based on the concept of adding some behavior to an object (in our case, a function) without affecting other objects. The term *decorator* is also popular because of its usage in frameworks such as Angular or (in an experimental mode) for general programming in JavaScript.



Decorators are being considered for general adoption in JavaScript, but are currently (December 2019) still at Stage 2, *Draft* level, and it may be a while until they get to Stage 3 (*Candidate*) and finally Stage 4 (*Finished*, meaning officially adopted). You can read more about the proposal for decorators at <https://tc39.github.io/proposal-decorators/> and about the JavaScript adoption process itself, called TC39, at <https://tc39.github.io/process-document/>. See the *Questions* section in Chapter 11, *Implementing Design Patterns – The Functional Way*, for more information.

As for the term *wrapper*, it's more important and pervasive than you might have thought; in fact, JavaScript uses it widely. Where? You already know that object properties and methods are accessed through dot notation. However, you also know that you can write code such as `myString.length` or `22.9.toPrecision(5)`—where are those properties and methods coming from, given that neither strings nor numbers are objects? JavaScript actually creates a *wrapper object* around your primitive value. This object inherits all the methods that are appropriate to the wrapped value. As soon as the needed evaluation has been done, JavaScript throws away the just-created wrapper. We cannot do anything about these transient wrappers, but there is a concept we will come back to regarding a wrapper that allows methods to be called on things that are not of the appropriate type. This is an interesting idea; see Chapter 12, *Building Better Containers – Functional Data Types*, for more applications of that!

In this section, we'll look at three examples:

- Adding logging to a function
- Getting timing information from functions
- Using caching (*memoizing*) to improve the performance of functions

Let's get to work!

Logging

Let's start with a common problem. When debugging code, you usually need to add some kind of logging information to see if a function was called, with what arguments, what it returned, and so on. (Yes, of course, you can simply use a debugger and set breakpoints, but bear with me for this example!) Working normally, this means that you'll have to modify the code of the function itself, both at entry and on exit, to produce some logging output. For example, your original code could be something like the following:

```
function someFunction(param1, param2, param3) {
  // do something
  // do something else
  // and a bit more,
  // and finally
  return some expression;
}
```

In this case, you would have to modify so that it looks something like the following. Here, we need to add an `auxValue` variable to store the value that we want to log and return:

```
function someFunction(param1, param2, param3) {
  console.log("entering someFunction: ", param1, param2, param3);
  // do something
  // do something else
  // and a bit more,
  // and finally
  const auxValue = some expression;
  console.log("exiting someFunction: ", auxValue);
  return auxValue;
}
```

If the function can return at several places, you'll have to modify all the `return` statements to log the values that are to be returned. And if you are just calculating the return expression on the fly, you'll need an auxiliary variable to capture that value.

In the next section, we'll learn about logging and some special cases of it, such as functions that throw exceptions, as well as working in a purer way.

Logging in a functional way

Doing logging by modifying your functions as we showed isn't difficult, but modifying code is always dangerous and prone to *accidents*. So, let's put our FP hats on and think of a new way of doing this. We have a function that performs some kind of work and we want to know the arguments it receives and the value it returns.

Here, we can write a higher-order function that will have a single parameter – the original function – and return a new function that will do the following, in sequence:

1. Log the received arguments
2. Call the original function, catching its returned value
3. Log that value
4. Return it to the caller

A possible solution would be as follows:

```
const addLogging = fn => (...args) => {
  console.log(`entering ${fn.name}: ${args}`);
  const valueToReturn = fn(...args);
  console.log(`exiting ${fn.name}: ${valueToReturn}`);
  return valueToReturn;
};
```

The function returned by `addLogging()` behaves as follows:

- The first `console.log(...)` line shows the original function's name and its list of arguments.
- Then, the original function, `fn()`, is called and the returned value is stored.
- The second `console.log(...)` line shows the function name (again) and its returned value.
- Finally, the value that `fn()` calculated is returned.



If you were doing this for a Node application, you would probably opt for a better way of logging by using libraries such as Winston, Morgan, or Bunyan, depending on what you wanted to log. However, our focus is on showing you how to wrap the original function, and the needed changes for using those libraries would be small.

For example, we can use it with the upcoming functions—which are written, I agree, in an overly complicated way, just to have an appropriate example! We'll have a function that accomplishes subtraction by changing the sign of the second number and then adding it to the first. The following code does this:

```
function subtract(a, b) {
  b = changeSign(b);
  return a + b;
}

function changeSign(c) {
  return -c;
}
```

```
}  
  
subtract = addLogging(subtract);  
  
changeSign = addLogging(changeSign);  
  
let x = subtract(7, 5);
```

The result of executing the previous line would be the following lines of logging:

```
entering subtract: 7, 5  
entering changeSign: 5  
exiting changeSign: -5  
exiting subtract: 2
```

All the changes we had to do in our code were the reassignments of `subtract()` and `changeSign()`, which essentially replaced them everywhere with their new log-producing wrapped versions. Any call to those two functions will produce this output.



We'll see a possible error because we're not reassigning the wrapped logging function while memoizing in the following section.

This works fine for most functions, but what would happen if the wrapped function threw an exception? Let's take a look.

Taking exceptions into account

Let's enhance our logging function a bit by considering an adjustment. What happens to your log if the function throws an error? Fortunately, this is easy to solve. We just have to add a `try/catch` structure, as shown in the following code:

```
const addLogging2 = fn => (...args) => {  
  console.log(`entering ${fn.name}: ${args}`);  
  try {  
    const valueToReturn = fn(...args);  
    console.log(`exiting ${fn.name}: ${valueToReturn}`);  
    return valueToReturn;  
  } catch (thrownError) {  
    console.log(`exiting ${fn.name}: threw ${thrownError}`);  
    throw thrownError;  
  }  
};
```

With this change, if the function threw an error, you'd also get an appropriate logging message, and the exception would be rethrown for processing.

Other changes to get an even better logging output would be up to you – adding date and time data, enhancing the way parameters are listed, and so on. However, our implementation still has an important defect; let's make it better and purer.

Working in a purer way

When we wrote the `addLogging()` function, we looked at some precepts we saw in Chapter 4, *Behaving Properly – Pure Functions*, because we included an impure element (`console.log()`) in our code. With this, not only did we lose flexibility (would you be able to select an alternate way of logging?) but we also complicated our testing. We could manage to test it by spying on the `console.log()` method, but that isn't very clean: we depend on knowing the internals of the function we want to test, instead of doing a purely black-box test. Take a look at the following example for a clearer understanding of this:

```
describe("a logging function", function() {
  it("should log twice with well behaved functions", () => {
    let something = (a, b) => `result=${a}:${b}`;
    something = addLogging(something);

    spyOn(window.console, "log");
    something(22, 9);
    expect(window.console.log).toHaveBeenCalledTimes(2);
    expect(window.console.log).toHaveBeenCalledWith(
      "entering something: 22,9"
    );
    expect(window.console.log).toHaveBeenCalledWith(
      "exiting something: result=22:9"
    );
  });

  it("should report a thrown exception", () => {
    let thrower = (a, b, c) => {
      throw "CRASH!";
    };
    spyOn(window.console, "log");
    expect(thrower).toThrow();

    thrower = addLogging(thrower);
    try {
      thrower(1, 2, 3);
    } catch (e) {
      expect(window.console.log).toHaveBeenCalledTimes(2);
    }
  });
});
```

```
    expect(window.console.log).toHaveBeenCalledWith(
      "entering thrower: 1,2,3"
    );
    expect(window.console.log).toHaveBeenCalledWith(
      "exiting thrower: threw CRASH!"
    );
  }
});
});
```

Running this test shows that `addLogging()` behaves as expected, so this is a solution. Our first test just does a simple subtraction and verifies that logging was called with appropriate data, while the second test checks an error-throwing function to also verify that the correct logs were produced.

Even so, being able to test our function in this way doesn't solve the lack of flexibility we mentioned. We should pay attention to what we wrote in the *Injecting impure functions* section – the logging function should be passed as an argument to the wrapper function so that we can change it if we need to:

```
const addLogging3 = (fn, logger = console.log) => (...args) => {
  logger(`entering ${fn.name}: ${args}`);
  try {
    const valueToReturn = fn(...args);
    logger(`exiting ${fn.name}: ${valueToReturn}`);
    return valueToReturn;
  } catch (thrownError) {
    logger(`exiting ${fn.name}: threw ${thrownError}`);
    throw thrownError;
  }
};
```

If we don't do anything, the logging wrapper will obviously produce the same results as in the previous section. However, we could provide a different logger – for example, with Node, we could use *winston*, a common logging tool, and the results would vary accordingly:



See <https://github.com/winstonjs/winston> for more on *winston*.

```
const winston = require("winston");
const myLogger = t => winston.log("debug", "Logging by winston: %s", t);
winston.level = "debug";
```



```
        "exiting thrower: threw CRASH!"
    );
  }
  });
});
```

The preceding tests work exactly like the previous ones we wrote earlier, but use and inspect the dummy logger instead of dealing with the original `console.log()` calls. Writing the test in this way avoids all possible problems due to side effects, so it's much cleaner and safer.

When applying FP techniques, always keep in mind that if you are somehow complicating your own job – for example, making it difficult to test any of your functions – then you must be doing something wrong. In our case, the mere fact that the output of `addLogging()` was an impure function should have raised an alarm. Of course, given the simplicity of the code, in this particular case, you might decide that it's not worth a fix, that you can do without testing, and that you don't need to be able to change the way logging is produced. However, long experience in software development suggests that, sooner or later, you'll come to regret that sort of decision, so try to go with the cleaner solution instead.

Now that we have dealt with logging, we'll look at another need: timing functions for performance reasons.

Timing functions

Another possible application for wrapped functions is to record and log the timing of each function invocation in a fully transparent way. Simply put, we want to be able to tell how long a function call takes, most likely for performance studies. However, in the same way we dealt with logging, we don't want to have to modify the original function and will use a higher-order function instead.



If you plan to optimize your code, remember the following three rules: *Don't do it*, *Don't do it yet*, and *Don't do it without measuring*. It has been mentioned that much bad code arises from early attempts at optimization, so don't start by trying to write optimal code, don't try to optimize until you recognize the need for it, and don't do it haphazardly, without trying to determine the reasons for the slowdown by measuring all the parts of your application.

Along the lines of the preceding example, we can write an `addTiming()` function that, given any function, will produce a wrapped version that will write out timing data on the console but will otherwise work in exactly the same way:

```
const myPut = (text, name, tStart, tEnd) =>
  console.log(`${name} - ${text} ${tEnd - tStart} ms`);

const myGet = () => performance.now();

const addTiming = (fn, getTime = myGet, output = myPut) => (...args) => {
  let tStart = getTime();

  try {
    const valueToReturn = fn(...args);
    output("normal exit", fn.name, tStart, getTime());
    return valueToReturn;
  } catch (thrownError) {
    output("exception thrown", fn.name, tStart, getTime());
    throw thrownError;
  }
};
```

Note that, along the lines of the enhancement we applied in the previous section to the logging function, we are providing separate logger and time access functions. Writing tests for our `addTiming()` function should prove easy, given that we can inject both impure functions.



Using `performance.now()` provides the highest accuracy. If you don't need such precision as what's provided by that function (and it's arguable that it is overkill), you could simply substitute `Date.now()`. For more on these alternatives, see <https://developer.mozilla.org/en-US/docs/Web/API/Performance/now> and https://developer.mozilla.org/en/docs/Web/JavaScript/Reference/Global_Objects/Date/now. You could also consider using `console.time()` and `console.timeEnd()`; see <https://developer.mozilla.org/en-US/docs/Web/API/Console/time> for more information.

To be able to try out the logging functionality, I've modified the `subtract()` function so that it throws an error if you attempt to subtract 0. (Yes, of course, you can subtract 0 from another number, but I wanted to have some kind error-throwing situation, at any cost!)

You could also list the input parameters, if desired, for more information:

```
subtract = addTiming(subtract);

let x = subtract(7, 5); // subtract - normal exit 0.10500000000001819 ms

let y = subtract(4, 0); // subtract - exception thrown 0.0949999999999136
// ms
```

The preceding code is quite similar to the previous `addLogging()` function, and that's reasonable—in both cases, we are adding some code before the actual function call, and then some new code after the function returns. You might even consider writing a *higher-order* function, which would receive three functions and produce a new higher-order function as output (such as `addLogging()` or `addTiming()`) that would call the first function at the beginning, and then the second function if the wrapped function returned a value, or the third function if an error had been thrown! What about that?

Memoizing functions

In Chapter 4, *Behaving Properly – Pure Functions*, we considered the case of the Fibonacci function and learned how we could transform it, by hand, into a much more efficient version by means of *memoization*: caching calculated values to avoid recalculations. A *memoized* function is one that will avoid redoing a process if the result was found earlier. We want to be able to turn any function into a memoized one so that we can get a more optimized version.



A real-life memoizing solution should also take into account the available RAM and have some ways of avoiding filling it up; however, this is beyond the scope of this book. Also, we won't be looking into performance issues; those optimizations are also beyond the scope of this book.

For simplicity, let's only consider functions with a single, non-structured parameter and leave functions with more complex parameters (objects, arrays) or more than one parameter for later.



The kind of values we can handle with ease are JavaScript's primitive values: data that aren't objects and have no methods. JavaScript has six of these: `boolean`, `null`, `number`, `string`, `symbol`, and `undefined`. Usually, we only see the first four as actual arguments. You can find out more by going to <https://developer.mozilla.org/en-US/docs/Glossary/Primitive>.

We won't be aiming to produce the best-ever memoizing solution, but let's study the subject a bit and produce several variants of a memoizing higher-order function. First, we'll deal with functions with a single parameter and then consider functions with several parameters.

Simple memoization

We will work with the Fibonacci function we mentioned previously, which is a simple case: it receives a single numeric parameter. This function is as follows:

```
function fib(n) {
  if (n == 0) {
    return 0;
  } else if (n == 1) {
    return 1;
  } else {
    return fib(n - 2) + fib(n - 1);
  }
}
```

The solution we created previously was general in concept, but particularly in its implementation: we had to directly modify the code of the function in order to take advantage of said memoization. Now, we should look into a way of doing this automatically, in the same fashion as we do it with other wrapped functions. The solution would be a `memoize()` function that wraps any other function in order to apply memoization:

```
const memoize = fn => {
  let cache = {};
  return x => (x in cache ? cache[x] : (cache[x] = fn(x)));
};
```

How does this work? The returned function, for any given argument, checks whether the argument was already received, that is, whether it can be found as a key in the cache object. If so, there's no need for calculation, and the cached value is returned. Otherwise, we calculate the missing value and store it in the cache. (We are using a closure to hide the cache from external access.) Here, we are assuming that the memoized function receives only one argument (`x`) and that it is a primitive value, which can then be directly used as a key value for the cache object; we'll consider other cases later.

Is this working? We'll have to time it – and we happen to have a useful `addTiming()` function for that! First, we take some timings for the original `fib()` function. We want to time the complete calculation and not each individual recursive call, so we write an auxiliary `testFib()` function and that's the one we'll time.

We should repeat the timing operations and do an average but, since we just want to confirm that memoizing works, we'll tolerate differences:

```
const testFib = n => fib(n);

addTiming(testFib)(45); // 15,382.255 ms
addTiming(testFib)(40); // 1,600.600 ms
addTiming(testFib)(35); // 146.900 ms
```

Your own times will vary, of course, depending on your specific CPU, RAM, and so on. However, the results seem logical: the exponential growth we mentioned in Chapter 4, *Behaving Properly – Pure Functions*, appears to be present, and times grow quickly. Now, let's memoize `fib()`. We should get shorter times... shouldn't we?

```
const testMemoFib = memoize(n => fib(n));

addTiming(testMemoFib)(45); // 15,537.575 ms
addTiming(testMemoFib)(45); // 0.005 ms... good!
addTiming(testMemoFib)(40); // 1,368.880 ms... recalculating?
addTiming(testMemoFib)(35); // 123.970 ms... here too?
```

Something's wrong! The times should have gone down, but they are just about the same. This is because of a common error, which I've even seen in some articles and on some web pages. We are timing `testMemoFib()`, but nobody calls that function, except for timing, and that only happens once! Internally, all recursive calls are to `fib()`, which isn't memoized. If we called `testMemoFib(45)` again, *that* call would be cached, and it would return almost immediately, but that optimization doesn't apply to the internal `fib()` calls. This is the reason why the calls for `testMemoFib(40)` and `testMemoFib(35)` weren't optimized – when we did the calculation for `testMemoFib(45)`, that was the only value that got cached.

The correct solution is as follows:

```
fib = memoize(fib);

addTiming(fib)(45); // 0.080 ms
addTiming(fib)(40); // 0.025 ms
addTiming(fib)(35); // 0.009 ms
```

Now, when calculating `fib(45)`, all the intermediate Fibonacci values (from `fib(0)` to `fib(45)` itself) are stored, so the forthcoming calls have practically no work to do.

Now that we know how to memoize single-argument functions, let's look at functions with more arguments.

More complex memoization

What can we do if we have to work with a function that receives two or more arguments, or that can receive arrays or objects as arguments? Of course, like in the problem that we looked at in [Chapter 2, Thinking Functionally – A First Example](#), about having a function do its job only once, we could simply ignore the question: if the function to be memoized is unary, we go through the memoization process; otherwise, if the function has a different arity, we just don't do anything!



The number of parameters of a function is called the *arity* of the function, or its *valence*. You may speak in three different ways: you can say a function has arity 1, 2, 3, and so on; you can say that a function is unary, binary, ternary, and so on; or you can say it's monadic, dyadic, triadic, and so on. Take your pick!

Our first attempt could be just memoizing unary functions, and leave the rest alone, as in the following code:

```
const memoize2 = fn => {
  if (fn.length === 1) {
    let cache = {};
    return x => (x in cache ? cache[x] : (cache[x] = fn(x)));
  } else {
    return fn;
  }
};
```

Working more seriously, if we want to be able to memoize any function, we must find a way to generate cache keys. To do this, we have to find a way to convert any kind of argument into a string. We cannot use a non-primitive as a cache key directly. We could attempt to convert the value into a string with something like `strX = String(x)`, but we'll have problems. With arrays, it seems this could work. However, take a look at the following three cases, involving different arrays but with a twist:

```
var a = [1, 5, 3, 8, 7, 4, 6];
String(a); // "1,5,3,8,7,4,6"

var b = [[1, 5], [3, 8, 7, 4, 6]];
String(b); // "1,5,3,8,7,4,6"

var c = [[1, 5, 3], [8, 7, 4, 6]];
String(c); // "1,5,3,8,7,4,6"
```

These three cases produce the same result. If we were only considering a single array argument, we'd probably be able to make do, but when different arrays produce the same key, that's a problem. Things become worse if we have to receive objects as arguments, because the `String()` representation of any object is, invariably, "[object Object]":

```
var d = {a: "fk"};
String(d); // "[object Object]"

var e = [{p: 1, q: 3}, {p: 2, q: 6}];
String(e); // "[object Object],[object Object]"
```

The simplest solution is to use `JSON.stringify()` to convert whatever arguments we have received into a useful, distinct string:

```
var a = [1, 5, 3, 8, 7, 4, 6];
JSON.stringify(a); // "[1,5,3,8,7,4,6]"

var b = [[1, 5], [3, 8, 7, 4, 6]];
JSON.stringify(b); // "[[1,5],[3,8,7,4,6]]"

var c = [[1, 5, 3], [8, 7, 4, 6]];
JSON.stringify(c); // "[[1,5,3],[8,7,4,6]]"

var d = {a: "fk"};
JSON.stringify(d); // "{\"a\":\"fk\"}"

var e = [{p: 1, q: 3}, {p: 2, q: 6}];
JSON.stringify(e); // "[{\"p\":1,\"q\":3},{\"p\":2,\"q\":6}]"
```

For performance, our logic should be as follows: if the function that we are memoizing receives a single argument that's a primitive value, we can use that argument directly as a cache key. In other cases, we would use the result of `JSON.stringify()` that's applied to the array of arguments. Our enhanced memoizing higher-order function could be as follows:

```
const memoize3 = fn => {
  let cache = {};
  const PRIMITIVES = ["number", "string", "boolean"];
  return (...args) => {
    let strX =
      args.length === 1 && PRIMITIVES.includes(typeof args[0])
        ? args[0]
        : JSON.stringify(args);
    return strX in cache ? cache[strX] : (cache[strX] = fn(...args));
  };
};
```

In terms of universality, this is the safest version. If you are sure about the type of parameters in the function you are going to process, it's arguable that our first version was faster. On the other hand, if you want to have easier-to-understand code, even at the cost of some wasted CPU cycles, you could go with a simpler version:

```
const memoize4 = fn => {
  let cache = {};
  return (...args) => {
    let strX = JSON.stringify(args);
    return strX in cache ? cache[strX] : (cache[strX] = fn(...args));
  };
};
```



If you want to learn about the development of a top-performance memoizing function, read Caio Gondim's *How I wrote the world's fastest JavaScript memoization library* article, available online at <https://community.risingstack.com/the-worlds-fastest-javascript-memoization-library/>.

So far, we have achieved several interesting memoizing functions, but how will we manage to write tests for them? Let's analyze this problem now.

Memoization testing

Testing the memoization higher-order function poses an interesting problem – just how would you go about it? The first idea would be to look into the cache – but that's private and not visible. Of course, we could change `memoize()` so that it uses a global cache or somehow allows external access to the cache, but doing that sort of internal exam is frowned upon: you should try to do your tests based on external properties only.

Accepting that we shouldn't try to examine the cache, we could go for a time control: calling a function such as `fib()`, for a large value of `n`, should take longer if the function isn't memoized. This is certainly possible, but it's also prone to possible failures: something external to your tests could run at just the wrong time and it could be possible that your memoized run would take longer than the original one. Okay, it's possible, but not probable – but your test isn't fully reliable.

So, let's go for a more direct analysis of the number of actual calls to the memoized function. Working with a non-memoized, original `fib()`, we could test whether the function works normally and check how many calls it makes:

```
var fib = null;
beforeEach(() => {
```



```

fib = n => {
  if (n == 0) {
    return 0;
  } else if (n == 1) {
    return 1;
  } else {
    return fib(n - 2) + fib(n - 1);
  }
};

describe("the original fib", function() {
  it("should produce correct results", () => {
    expect(fib(0)).toBe(0);
    expect(fib(1)).toBe(1);
    expect(fib(5)).toBe(5);
    expect(fib(8)).toBe(21);
    expect(fib(10)).toBe(55);
  });

  it("should repeat calculations", () => {
    spyOn(window, "fib").and.callThrough();
    expect(fib(6)).toBe(8);
    expect(fib).toHaveBeenCalledTimes(25);
  });
});

```

The preceding code is fairly straightforward: we are using the Fibonacci function we developed earlier and testing that it produces correct values. For instance, the fact that `fib(6)` equals 8 is easy to verify, but where do you find out that the function is called 25 times? For the answer to this, let's revisit the diagram we looked at in Chapter 4, *Behaving Properly – Pure Functions*:

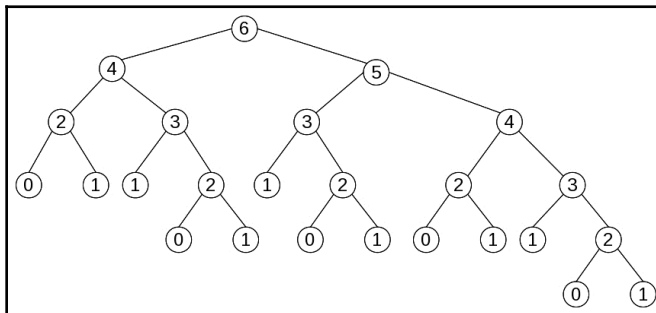


Figure 6.1: All the recursive calls needed for calculating `fib(6)`

Each node is a call; just by counting, we can see that in order to calculate `fib(6)`, 25 calls are actually made to `fib()`. Now, let's turn to the memoized version of the function.

Testing that it still produces the same results is easy:

```
describe("the memoized fib", function() {
  beforeEach(() => {
    fib = memoize(fib);
  });

  it("should produce same results", () => {
    expect(fib(0)).toBe(0);
    expect(fib(1)).toBe(1);
    expect(fib(5)).toBe(5);
    expect(fib(8)).toBe(21);
    expect(fib(10)).toBe(55);
  });

  it("shouldn't repeat calculations", () => {
    spyOn(window, "fib").and.callThrough();

    expect(fib(6)).toBe(8); // 11 calls
    expect(fib).toHaveBeenCalledTimes(11);

    expect(fib(5)).toBe(5); // 1 call
    expect(fib(4)).toBe(3); // 1 call
    expect(fib(3)).toBe(2); // 1 call
    expect(fib).toHaveBeenCalledTimes(14);
  });
});
```

But why is it called 11 times for calculating `fib(6)`, and then three times more after calculating `fib(5)`, `fib(4)`, and `fib(3)`? To answer the first part of this question, let's analyze the diagram we looked at earlier:

- First, we call `fib(6)`, which calls `fib(4)` and `fib(5)`. This is three calls.
- When calculating `fib(4)`, `fib(2)` and `fib(3)` are called; the count is up to five.
- When calculating `fib(5)`, `fib(3)` and `fib(4)` are called; the count climbs to 11.
- Finally, `fib(6)` is calculated and cached.
- `fib(3)` and `fib(4)` are both cached, so no more calls are made.
- `fib(5)` is calculated and cached.
- When calculating `fib(2)`, `fib(0)` and `fib(1)` are called; now, we have seven calls.
- When calculating `fib(3)`, `fib(1)` and `fib(2)` are called; the count is up to nine.

- `fib(4)` is calculated and cached.
- `fib(1)` and `fib(2)` are both already cached, so no further calls are made.
- `fib(3)` is calculated and cached.
- When calculating `fib(0)` and `fib(1)`, no extra calls are made and both are cached.
- `fib(2)` is calculated and cached.

Whew! So, the count of calls for `fib(6)` is 11. Given that all the values of `fib(n)` have been cached, for `n` from 0 to 6, it's easy to see why calculating `fib(5)`, `fib(4)`, and `fib(3)` only adds three calls: all the other required values are already cached.

In this section, we've dealt with several examples that implied wrapping functions so that they keep working, but with some kind of extra feature added in. Now, let's look at a different case where we want to change the way a function actually works.

Altering a function's behavior

In the previous section, we considered some ways of wrapping functions so that they maintain their original functionality, even though they've been enhanced in some way. Now, we'll turn to modify what the functions do so that the new results will differ from the original function's ones.

We'll be covering the following topics:

- Revisiting the problem of having a function work, but just once
- Negating or inverting a function's result
- Changing the arity of a function

Let's get started!

Doing things once, revisited

Back in [Chapter 2, *Thinking Functionally – A First Example*](#), we went through an example of developing an FP-style solution for a simple problem: fixing things so that a given function works only once. The following code is what we wrote back then:

```
const once = func => {
  let done = false;
  return (...args) => {
    if (!done) {
```

```
    done = true;
    func(...args);
  }
};
};
```

This is a perfectly fine solution; it works well and we have nothing to object to. We can, however, think of a variation. We could observe that the given function gets called once, but its return value gets lost. This is easy to fix: all we need to do is add a `return` statement. However, that wouldn't be enough; what would the function return if it's called more? We can take a page out of the memoizing solution and store the function's return value for future calls.

Let's store the function's value in a variable (`result`) so that we can return it later:

```
const once2 = func => {
  let done = false;
  let result;
  return (...args) => {
    if (!done) {
      done = true;
      result = func(...args);
    }
    return result;
  };
};
```

The first time the function gets called, its value is stored in `result`; further calls just return that value with no further process. You could also think of making the function work only once, but for each set of arguments. You wouldn't have to do any work for that – `memoize()` would be enough!

Back in Chapter 2, *Thinking Functionally – A First Example*, in the *An even better solution* section, we considered a possible alternative to `once()`: another higher-order function that took two functions as parameters and allowed the first function to be called only once, calling the second function from that point on. Adding a `return` statement to the code from before, it would have been as follows:

```
const onceAndAfter = (f, g) => {
  let done = false;
  return (...args) => {
    if (!done) {
      done = true;
      return f(...args);
    } else {
      return g(...args);
    }
  };
};
```

```
    }  
  };  
};
```

We can rewrite this if we remember that functions are first-order objects. Instead of using a flag to remember which function to call, we can use a variable (`toCall`) to directly store whichever function needs to be called. Logically, that variable will be initialized to the first function, but will then change to the second one. The following code implements that change:

```
const onceAndAfter2 = (f, g) => {  
  let toCall = f;  
  return (...args) => {  
    let result = toCall(...args);  
    toCall = g;  
    return result;  
  };  
};
```

The `toCall` variable is initialized with `f`, so `f()` will get called the first time, but then `toCall` gets the `g` value, implying that all future calls will execute `g()` instead. The very same example we looked at earlier in this book would still work:

```
const squeak = (x) => console.log(x, "squeak!!");  
  
const creak = (x) => console.log(x, "creak!!");  
  
const makeSound = onceAndAfter2(squeak, creak);  
  
makeSound("door"); // "door squeak!!"  
makeSound("door"); // "door creak!!"  
makeSound("door"); // "door creak!!"  
makeSound("door"); // "door creak!!"
```

In terms of performance, the difference may be negligible. The reason for showing this further variation is to show that you should keep in mind that, by storing functions, you can often produce results in a simpler way. Using flags to store state is a common technique that's used everywhere in procedural programming. However, here, we manage to skip that usage and produce the same result. Now, let's look at some new examples of wrapping functions to change their behaviors.

Logically negating a function

Let's consider the `filter()` method from Chapter 5, *Programming Declaratively – A Better Style*. Given a predicate, we can filter the array to only include those elements for which the predicate is true. But how would you do a reverse filter and *exclude* the elements for which the predicate is true?

The first solution should be pretty obvious: rework the predicate so that it returns the opposite of whatever it originally returned. In Chapter 5, *Programming Declaratively – A Better Style*, we looked at the following example:

```
const delinquent = serviceResult.accountsData.filter(v => v.balance < 0);
```

So, we could just write it the other way round, in either of these two equivalent fashions. Note the different ways of writing the same predicate to test for non-negative values:

```
const notDelinquent = serviceResult.accountsData.filter(
  v => v.balance >= 0
);

const notDelinquent2 = serviceResult.accountsData.filter(
  v => !(v.balance < 0)
);
```

That's perfectly fine, but we could also have had something like the following in our code:

```
const isNegativeBalance = v => v.balance < 0;

// ...many lines later...

const delinquent2 = serviceResult.accountsData.filter(isNegativeBalance);
```

In this case, rewriting the original function isn't possible. However, working in a functional way, we can just write a higher-order function that will take any predicate, evaluate it, and then negate its result. A possible implementation would be quite simple, thanks to modern JavaScript syntax:

```
const not = fn => (...args) => !fn(...args);
```

Working in this way, we could have rewritten the preceding filter as follows; to test for non-negative balances, we use the original `isNegativeBalance()` function, which is negated via our `not()` higher-order function:

```
const isNegativeBalance = v => v.balance < 0;

// ...many lines later...
```

```
const notDelinquent3 = serviceResult.accountsData.filter(  
  not (isNegativeBalance)  
);
```

There is an additional solution we might want to try out – instead of reversing the condition (as we did), we could write a new filtering method (possibly `filterNot()`?) that would work in the opposite way to `filter()`. The following code shows how this new function would be written:

```
const filterNot = arr => fn => arr.filter(not(fn));
```

This solution doesn't fully match `filter()` since you cannot use it as a method, but we could either add it to `Array.prototype` or apply some methods. We'll look at these methods in Chapter 8, *Connecting Functions – Pipelining and Composition*. However, it's more interesting to note that we used the negated function, so `not()` is actually necessary for both solutions to the reverse filtering problem. In the upcoming *Demethodizing – turning methods into functions* section, we will see that we have yet another solution since we will be able to decouple methods such as `filter()` from the objects they apply to, thereby changing them into common functions.

As for negating the function *versus* using a new `filterNot()`, even though both possibilities are equally valid, I think using `not()` is clearer; if you already understand how filtering works, then you can practically read it aloud and it will be understandable: we want those that don't have a negative balance, right? Now, let's consider a related problem: inverting the results of a function.

Inverting the results

In the same vein as the preceding filtering problem, let's revisit the sorting problem from the *Injection – sorting it out* section of Chapter 3, *Starting Out with Functions – A Core Concept*. Here, we wanted to sort an array with a specific method. Therefore, we used `.sort()`, providing it with a comparison function that basically pointed out which of the two strings should go first. To refresh your memory, given two strings, the function should do the following:

- Return a negative number if the first string should precede the second one
- Return 0 if the strings are the same
- Return a positive number if the first string should follow the second one

Let's go back to the code we looked at for sorting in Spanish. We had to write a special comparison function so that sorting would take into account the special character order rules from Spanish, such as including the letter *ñ* between *n* and *o*, and more. The code for this was as follows:

```
const spanishComparison = (a, b) => a.localeCompare(b, "es");

palabras.sort(spanishComparison); // sorts the palabras array according to
Spanish rules
```

We are facing a similar problem: how can we manage to sort in *descending* order? Given what we saw in the previous section, two alternatives should immediately come to mind:

- Write a function that will invert the result from the comparing function. This will invert the result of all the decisions as to which string should precede, and the final result will be an array sorted in exactly the opposite way.
- Write a `sortDescending()` function or method that does its work in the opposite fashion to `sort()`.

Let's write an `invert()` function that will change the result of a comparison. The code itself is quite similar to that of `not()`:

```
const invert = fn => (...args) => -fn(...args);
```

Given this higher-order function, we can sort in descending order by providing a suitably inverted comparison function. Take a look at the last few lines, where we use `invert()` to change the result of the sorting comparison:

```
const spanishComparison = (a, b) => a.localeCompare(b, "es");

var palabras = ["ñandú", "oasis", "mano", "natural", "mítico", "musical"];

palabras.sort(spanishComparison);
// ["mano", "mítico", "musical", "natural", "ñandú", "oasis"]

palabras.sort(invert(spanishComparison));
// ["oasis", "ñandú", "natural", "musical", "mítico", "mano"]
```

The output is as expected: when we `invert()` the comparison function, the results are in the opposite order. Writing unit tests would be quite easy, given that we already have some test cases with their expected results, wouldn't it?

Arity changing

Back in the *Parsing numbers tacitly* section of Chapter 5, *Programming Declaratively – A Better Style*, we saw that using `parseInt()` with `reduce()` would produce problems because of the unexpected arity of that function, which took more than one argument—remember the example from earlier?

```
["123.45", "-67.8", "90"].map(parseInt); // problem: parseInt isn't
                                         // monadic!
                                         // [123, NaN, NaN]
```

We have more than one way to solve this. In Chapter 5, *Programming Declaratively – A Better Style*, we went with an arrow function. This was a simple solution, with the added advantage of being clear to understand. In Chapter 7, *Transforming Functions – Currying and Partial Application*, we will look at yet another, based on partial application. For now, let's go with a higher-order function. What we need is a function that will take another function as a parameter and turn it into a unary function. Using JavaScript's spread operator and an arrow function, this is easy to manage:

```
const unary = fn => (...args) => fn(args[0]);
```

Using this function, our number parsing problem goes away:

```
["123.45", "-67.8", "90"].map(unary(parseInt)); // [123, -67, 90]
```

It goes without saying that it would be equally simple to define further `binary()`, `ternary()`, and other functions that would turn any function into an equivalent, restricted-arity, version. Let's not go overboard and just look at a couple of all the possible functions:

```
const binary = fn => (...args) => fn(args[0], args[1]);
const ternary = fn => (...args) => fn(args[0], args[1], args[2]);
```

This works, but spelling out all the parameters can become tiresome. We can even go one better by using array operations and spreading and make a generic function to deal with all of these cases, as follows:

```
const arity = (fn, n) => (...args) => fn(...args.slice(0, n));
```

With this generic `arity()` function, we can give alternative definitions for `unary()`, `binary()`, and so on. We could even rewrite the earlier functions as follows:

```
const unary = fn => arity(fn, 1);
const binary = fn => arity(fn, 2);
const ternary = fn => arity(fn, 3);
```

You may be thinking that there aren't many cases in which you would want to apply this kind of solution, but in fact, there are many more than you would expect. Going through all of JavaScript's functions and methods, you can easily produce a list starting with `apply()`, `assign()`, `bind()`, `concat()`, `copyWithin()`, and many more! If you wanted to use any of those in a tacit way, you would probably need to fix its arity so that it would work with a fixed, non-variable number of parameters.



If you want a nice list of JavaScript functions and methods, check out <https://developer.mozilla.org/en/docs/Web/JavaScript/Guide/Functions> and https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Methods_Index. As for tacit programming (or pointfree style), we'll be coming back to it in Chapter 8, *Connecting Functions – Pipelining and Composition*.

So far, we have learned how to wrap functions while keeping their original behavior or by changing it in some fashion. Now, let's consider some other ways of modifying functions.

Changing functions in other ways

Let's end this chapter by considering some other sundry functions that provide results such as new finders, decoupling methods from objects, and more. Our examples will include the following:

- Turning operations (such as adding with the `+` operator) into functions
- Turning functions into promises
- Accessing objects to get the value of a property
- Turning methods into functions
- A better way of finding optimum values

Turning operations into functions

We have already seen several cases in which we needed to write a function just to add or multiply a pair of numbers. For example, in the *Summing an array* section of Chapter 5, *Programming Declaratively – A Better Style*, we had to write code equivalent to the following:

```
const mySum = myArray.reduce((x, y) => x + y, 0);
```

In the *Working with ranges* section of Chapter 5, *Programming Declaratively – A Better Style*, to calculate a factorial, we wrote this:

```
const factorialByRange = n => range(1, n + 1).reduce((x, y) => x * y, 1);
```

It would have been easier if we could just turn a binary operator into a function that calculates the same result. The preceding two examples could have been written more succinctly, as follows. Can you understand the change we made?

```
const mySum = myArray.reduce(binaryOp("+"), 0);

const factorialByRange = n => range(1, n + 1).reduce(binaryOp("*"), 1);
```

We haven't looked at how `binaryOp()` is implemented yet, but the key notion is that instead of an infix operator (like we use when we write `22+9`), we now have a function (as if we could write our sum like `+(22, 9)`, which certainly isn't valid JavaScript). Let's see how we can make this work.

Implementing operations

How would we write this `binaryOp()` function? There are at least two ways of doing so: a safe but long one and a riskier and shorter alternative. The first would require listing each possible operator. The following code does this by using a longish `switch`:

```
const binaryOp1 = op => {
  switch (op) {
    case "+":
      return (x, y) => x + y;
    case "-":
      return (x, y) => x - y;
    case "*":
      return (x, y) => x * y;
    //
    // etc.
    //
  }
};
```

This solution is perfectly fine but requires too much work. The second is more dangerous, but shorter. Please consider this just as an example, for learning purposes; using `eval()` isn't recommended, for security reasons! Our second version would simply use `Function()` to create a new function that uses the desired operator, as follows:

```
const binaryOp2 = op => new Function("x", "y", `return x ${op} y;`);
```

If you follow this trail of thought, you may also define a `unaryOp()` function, even though there are fewer applications for it. (I leave this implementation to you; it's quite similar to what we already wrote.) In [Chapter 7, Transforming Functions – Currying and Partial Application](#), we will look at an alternative way of creating this unary function by using partial application.

A handier implementation

Let's get ahead of ourselves. Doing FP doesn't mean always getting down to the very basic, simplest possible functions. For example, in an upcoming section of this book, we will need a function to check whether a number is negative, and we'll consider (see the *Converting into pointfree style* section of [Chapter 8, Connecting Functions – Pipelining and Composition](#)) using `binaryOp2()` to write it:

```
const isNegative = curry(binaryOp2(">"))(0);
```

Don't worry about the `curry()` function now (we'll get to it soon, in [Chapter 7, Transforming Functions – Currying and Partial Application](#)) – the idea is that it fixes the first argument to 0 so that our function will check for a given number, n , if $0 > n$. The point here is that the function we just wrote isn't very clear. We could do better if we defined a binary operation function that also lets us specify one of its parameters – the left one or the right one – in addition to the operator to be used. Here, we can write the following couple of functions, which define the functions where the left or right operators are missing:

```
const binaryLeftOp = (x, op) => y => binaryOp2(op)(x, y);
```

```
const binaryOpRight = (op, y) => x => binaryOp2(op)(x, y);
```

With these new functions, we could simply write either of the following two definitions, though I think the second is clearer. I'd rather test whether a number is less than 0 than whether 0 is greater than the number:

```
const isNegative1 = binaryLeftOp(0, ">");
```

```
const isNegative2 = binaryOpRight("<", 0);
```

What is the point of this? Don't strive for some kind of *basic simplicity* or *going down to basics* code. We can transform an operator into a function, but if you can do better and simplify your coding by also specifying one of the two parameters for the operation, just do it! The idea of FP is to help write better code, and creating artificial limitations won't help anybody.

Of course, for a simple function such as checking whether a number is negative, I would never want to complicate things with currying, binary operators, pointfree style, or anything else, and I'd just write the following with no further ado:

```
const isNegative3 = x => x < 0;
```

So far, we have seen several ways of solving the same problem. Keep in mind that FP doesn't force you to pick one single way of doing things; instead, it allows you a lot of freedom in deciding on which way to go!

Turning functions into promises

In Node, most asynchronous functions require a callback such as `(err, data) => { ... }`: if `err` is `null`, the function was successful and `data` is its result, while if `err` has some value, the function failed and `err` gives the cause. (See https://nodejs.org/api/errors.html#errors_node_js_style_callbacks for more on this.)

However, you might prefer to work with promises instead. So, we can think of writing a higher-order function that will transform a function that requires a callback into a promise that lets you use the `.then()` and `.catch()` methods. (In Chapter 12, *Building Better Containers – Functional Data Types*, we will see that promises are actually monads, so this transformation is interesting in yet another way.)



Node, since version 8, already provides the `util.promisify()` function, which turns an async function into a promise. See https://nodejs.org/dist/latest-v8.x/docs/api/util.html#util_util_promisify_original for more on that.

How can we manage this? The transformation is rather simple. Given a function, we produce a new one: this will return a promise that, upon calling the original function with some parameters, will either `reject()` or `resolve()` the promise appropriately. The `promisify()` function does exactly that:

```
const promisify = fn => (...args) =>
  new Promise((resolve, reject) =>
    fn(...args, (err, data) => (err ? reject(err) : resolve(data)))
  );
```

When working in Node, the following style is fairly common:

```
const fs = require("fs");

const cb = (err, data) =>
```

```
err ? console.log("ERROR", err) : console.log("SUCCESS", data);

fs.readFile("./exists.txt", cb); // success, list the data
fs.readFile("./doesnt_exist.txt", cb); // failure, show exception
```

However, you can use promises instead by using the `promisify()` function. However, in current versions of Node, you would use `util.promisify()`:

```
const fsPromise = promisify(fs.readFile.bind(fs));

const goodRead = data => console.log("SUCCESSFUL PROMISE", data);
const badRead = err => console.log("UNSUCCESSFUL PROMISE", err);

fsPromise("./readme.txt") // success
  .then(goodRead)
  .catch(badRead);

fsPromise("./readmenot.txt") // failure
  .then(goodRead)
  .catch(badRead);
```

Now, you can use `fsPromise()` instead of the original method. To do so, we had to bind `fs.readFile`, as we mentioned in the *An unnecessary mistake* section of Chapter 3, *Starting Out with Functions – A Core Concept*.

Getting a property from an object

There is a simple function that we could also produce. Extracting an attribute from an object is a commonly required operation. For example, in Chapter 5, *Programming Declaratively – A Better Style*, we had to get latitudes and longitudes to be able to calculate an average. The code for this was as follows:

```
markers = [
  {name: "UY", lat: -34.9, lon: -56.2},
  {name: "AR", lat: -34.6, lon: -58.4},
  {name: "BR", lat: -15.8, lon: -47.9},
  ...
  {name: "BO", lat: -16.5, lon: -68.1}
];

let averageLat = average(markers.map(x => x.lat));
let averageLon = average(markers.map(x => x.lon));
```

We saw another example of this when we learned how to filter an array; in our example, we wanted to get the IDs for all the accounts with a negative balance. After filtering out all other accounts, we still needed to extract the ID field:

```
const delinquent = serviceResult.accountsData.filter(v => v.balance < 0);
const delinquentIds = delinquent.map(v => v.id);
```



We could have joined those two lines and produced the desired result with a one-liner, but that's not relevant here. In fact, unless the `delinquent` intermediate result was needed for some reason, most FP programmers would go for the one-line solution.

What do we need? We need a higher-order function that will receive the name of an attribute and produce a new function that will be able to extract an attribute from an object. Using the arrow function syntax, this function is easy to write:

```
const getField = attr => obj => obj[attr];
```



In the *Getters and setters* section of Chapter 10, *Ensuring Purity – Immutability*, we'll write an even more general version of this function that's able to "go deep" into an object to get an attribute of it, regardless of its location within the object.

With this function, the coordinates extraction process could have been written as follows:

```
let averageLat = average(markers.map(getField("lat")));
let averageLon = average(markers.map(getField("lon")));
```

For variety, we could have used an auxiliary variable to get the delinquent IDs, as follows:

```
const getId = getField("id");
const delinquent = serviceResult.accountsData.filter(v => v.balance < 0);
const delinquentIds = delinquent.map(getId);
```

Make sure that you fully understand what's going on here. The result of the `getField()` call is a function, which will be used in further expressions. The `map()` method requires a mapping function and is what `getField()` produces.

Demethodizing – turning methods into functions

Methods such as `filter()` and `map()` are only available for arrays; however, you may want to apply them to, say, a `NodeList` or a `String`, and you'd be out of luck. Also, we are focusing on strings, so having to use these functions as methods is not exactly what we had in mind. Finally, whenever we create a new function (such as `none()`, which we saw in the *Checking negatives* section of Chapter 5, *Programming Declaratively – A Better Style*), it cannot be applied in the same way as its peers (`some()` and `every()`, in this case) unless you do some prototype trickery. This is rightly frowned upon and not recommended.



Read the *Extending current data types* section of Chapter 12, *Building Better Containers – Functional Data Types*, where we will make `map()` available for most basic types.

So... what can we do? We can apply the old saying *If the mountain won't come to Muhammad, then Muhammad must go to the mountain* and, instead of worrying about not being able to create new methods, we will turn the existing methods into functions. We can do this if we convert each method into a function that will receive, as its first parameter, the object it will work on.

Decoupling methods from objects can help you because once you achieve this separation, everything turns out to be a function and your code will be simpler. (Remember what we wrote in the *Logically negating a function* section, regarding a possible `filterNot()` function in comparison to the `filter()` method?) A decoupled method works similarly to how *generic* functions do in other languages since they can be applied to diverse data types.



Take a look at https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Function for explanations on `apply()`, `call()`, and `bind()`. We are going to use these for our implementation. Back in Chapter 1, *Becoming Functional – Several Questions*, we saw the equivalence between `apply()` and `call()` when we used the spread operator.

There are three distinct, but similar, ways to implement this in JavaScript. The first argument in the list (`arg0`) will correspond to the object, the other arguments (`args`) to the actual ones for the called method. The three equivalent versions would be as follows. Note that any of them could be used as a `demethodize()` function; pick whichever you prefer!

```
const demethodize1 = fn => (arg0, ...args) => fn.apply(arg0, args);
const demethodize2 = fn => (arg0, ...args) => fn.call(arg0, ...args);
const demethodize3 = fn => (...args) => fn.bind(...args)();
```




There's yet another way of doing this: `const demethodize = Function.prototype.bind.bind(Function.prototype.call)`. If you want to understand how this works, read Leland Richardson's *Clever Way to Demethodize Native JS Methods*, at <http://www.intelligiblebabble.com/clever-way-to-demethodize-native-js-methods>.

Let's look at some applications of these! Starting with a simple example, we can use `map()` to loop over a string without converting it into an array of characters first. Say you wanted to separate a string into individual letters and make them uppercase; we could do this by using `split()` and `toUpperCase()`:

```
const name = "FUNCTIONAL";
const result = name.split("").map(x => x.toUpperCase());
/*
["F", "U", "N", "C", "T", "I", "O", "N", "A", "L"]
*/
```

However, if we demethodize `map()` and `toUpperCase()`, we can simply write the following:

```
const map = demethodize(Array.prototype.map);
const toUpperCase = demethodize(String.prototype.toUpperCase);

const result2 = map(name, toUpperCase);
/*
["F", "U", "N", "C", "T", "I", "O", "N", "A", "L"]
*/
```



For this particular case, we could have turned the string into uppercase and then split it into separate letters, as in `name.toUpperCase().split("")`, but it wouldn't have been such a nice example, with two usages of demethodizing being used.

In a similar way, we could convert an array of decimal amounts into properly formatted strings, with thousands of separators and decimal points:

```
const toLocaleString = demethodize(Number.prototype.toLocaleString);

const numbers = [2209.6, 124.56, 1048576];
const strings = numbers.map(toLocaleString);
/*
["2,209.6", "124.56", "1,048,576"]
*/
```

Alternatively, given the preceding demethodized `map()` function, this would have also worked:

```
const strings2 = map(numbers, toLocaleString);
```

The idea of demethodizing a method to turn it into a function will prove to be quite useful in diverse situations. We have already seen some examples where we could have applied it, and there will be more such cases in the rest of this book.

Finding the optimum

Let's end this section by creating an extension of the `find()` method. Suppose we want to find the optimum value—let's suppose it's the maximum—of an array of numbers. We could make do with this:

```
const findOptimum = arr => Math.max(...arr);

const myArray = [22, 9, 60, 12, 4, 56];
findOptimum(myArray); // 60
```

Now, is this sufficiently general? There are at least a pair of problems with this approach. First, are you sure that the optimum of a set will always be the maximum? If you were considering several mortgages, the one with the *lowest* interest rate could be the best, couldn't it? That is, assuming that always wanting the *maximum* of a set is too constrictive.



You could do a roundabout trick: if you change the signs of all the numbers in an array, find its maximum, and change its sign, then you actually get the minimum of the array. In our case, `-findOptimum(myArray.map((x) => -x))` would correctly produce 4—but it's not easily understandable code, is it?

Second, this way of finding the maximum depends on each option having a numeric value. But how would you find the optimum if such a value didn't exist? The usual way depends on comparing elements with each another and picking the one that comes on top of the comparison: compare the first element with the second and keep the best of those two; then, compare that value with the third element and keep the best; and then keep at it until you have finished going through all the elements.

The way to solve this problem with more generality is to assume the existence of a `comparator()` function, which takes two elements as arguments and returns the best of those. If you could associate a numeric value with each element, then the comparator function could simply compare those values. In other cases, it could do whatever logic is needed in order to decide what element comes out on top.

Let's try to create an appropriate higher-order function; our newer version will use `reduce()`, as follows:

```
const findOptimum2 = fn => arr => arr.reduce(fn);
```

With this, we can easily replicate the maximum- and minimum-finding functions – we just have to provide the appropriate reducing functions:

```
const findMaximum = findOptimum2((x, y) => (x > y ? x : y));
const findMinimum = findOptimum2((x, y) => (x < y ? x : y));

findMaximum(myArray); // 60
findMinimum(myArray); // 4
```

Let's go one better and compare non-numeric values. Let's imagine a superhero card game: each card represents a hero and has several numeric attributes, such as Strength, Powers, and Tech. When two heroes fight each other, the one with more categories with higher values than the other is the winner. Let's implement a comparator for this; a suitable `compareHeroes()` function could be as follows:

```
const compareHeroes = (card1, card2) => {
  const oneIfBigger = (x, y) => (x > y ? 1 : 0);
  const wins1 =
    oneIfBigger(card1.strength, card2.strength) +
    oneIfBigger(card1.powers, card2.powers) +
    oneIfBigger(card1.tech, card2.tech);
  const wins2 =
    oneIfBigger(card2.strength, card1.strength) +
    oneIfBigger(card2.powers, card1.powers) +
    oneIfBigger(card2.tech, card1.tech);
  return wins1 > wins2 ? card1 : card2;
};
```

Then, we can apply this to our *tournament* of heroes. Let's create a constructor to build the heroes:

```
function Hero(n, s, p, t) {
  this.name = n;
  this.strength = s;
  this.powers = p;
  this.tech = t;
}
```

Now, let's create our own league of heroes:

```
const codingLeagueOfAmerica = [
  new Hero("Forceful", 20, 15, 2),
```

```
new Hero("Electrico", 12, 21, 8),  
new Hero("Speediest", 8, 11, 4),  
new Hero("TechWiz", 6, 16, 30)  
];
```

With these definitions, we can write a `findBestHero()` function to get the top hero:

```
const findBestHero = findOptimum2(compareHeroes);  
  
findBestHero(codingLeagueOfAmerica); // Electrico is the top hero!
```



When you rank elements according to one-to-one comparisons, unexpected results may be produced. For instance, with our superheroes comparison rules, you could find three heroes where the results show that the first beats the second, the second beats the third, but the third beats the first! In mathematical terms, this means that the comparison function is not transitive and that you don't have a *total ordering* for the set.

With this, we have seen several ways of modifying functions in order to produce newer variants with enhanced processing; think of particular cases you might be facing and consider whether a higher-order function might help you out.

Summary

In this chapter, we learned how to write higher-order functions of our own that can either wrap another function to provide some new feature, alter a function's objective so that it does something else, or even provide totally new features, such as decoupling methods from objects or creating better finders. The main takeaway from this chapter is that you have a way of modifying the behavior of a function without actually having to modify its own code; higher-order functions can manage this in an orderly way.

In [Chapter 7, Transforming Functions – Currying and Partial Application](#), we'll keep working with higher-order functions and learn how to produce specialized versions of existing functions with predefined arguments by using currying and partial application.

Questions

6.1. **A border case:** What happens with our `getField()` function if we apply it to a null object? What should its behavior be? If necessary, modify the function.

6.2. How many? How many calls would be needed to calculate `fib(50)` without memoizing? For example, to calculate `fib(0)` or `fib(1)`, one call is enough with no further recursion needed, and for `fib(6)`, we saw that 25 calls were required. Can you find a formula to do this calculation?

6.3. A randomizing balancer: Write a higher-order function, that is, `randomizer(fn1, fn2, ...)`, that will receive a variable number of functions as arguments and return a new function that will, on each call, randomly call one of `fn1`, `fn2`, and so on. You could possibly use this to balance calls to different services on a server if each function was able to do an Ajax call. For bonus points, ensure that no function will be called twice in a row.

6.4. Just say no! In this chapter, we wrote a `not()` function that worked with Boolean functions and a `negate()` function that worked with numerical ones. Can you go one better and write a single `opposite()` function that will behave as `not()` or `negate()` as needed?

6.5. Missing companion: If we have a `getField()` function, we should also have a `setField()` one, so can you define it? We'll be needing both `getField()` and `setField()` in Chapter 10, *Ensuring Purity – Immutability*, when we work with getters, setters, and lenses. Note that `setField()` shouldn't directly modify an object; instead, it should return a new object with a changed value – it should be a pure function!

6.6. Wrong function length: Our `arity()` function works well, but the produced functions don't have the correct `length` attribute. Can you write a different arity-changing function without this defect?

```
const f1 = arity(parseInt, 1);
const f2 = arity(parseInt, 2);
/*
  f1.length === 0
  f2.length === 0
*/
```

6.7. Not reinventing the wheel: When we wrote `findMaximum()` and `findMinimum()`, we wrote our own functions to compare two values – but JavaScript already provides appropriate functions for that! Can you figure out alternative versions of our code based on that hint?

7

Transforming Functions - Currying and Partial Application

In Chapter 6, *Producing Functions – Higher-Order Functions*, we saw several ways of manipulating functions, to get new versions with some change in their functionality. In this chapter, we will go into a particular kind of transformation, a sort of *factory* method, that lets you produce new versions of any given function, with some fixed arguments.

We will be considering the following:

- **Currying:** A classic FP theoretical function that transforms a function with many parameters into a sequence of unary functions.
- **Partial application:** Another time-honored FP transformation, which produces new versions of functions by fixing some of their arguments.
- **Partial currying (a name of my own):** Can be seen as a mixture of the two previous transformations.

To be fair, we'll also see that some of these techniques can be emulated, possibly with greater clarity, by simple arrow functions. However, since you are quite liable to find currying and partial application in all sorts of texts and web pages on FP, it is quite important that you are aware of their meaning and usage, even if you opt for a simpler way out. Using the techniques in this chapter will provide you with a different way of producing functions out of other functions, and we'll look at several applications of the ideas in the following sections.

A bit of theory

The concepts that we are going to discuss in this chapter are in some ways very similar, and in other ways quite different. It's common to find some confusion as to their real meanings and there are plenty of web pages that misuse terms. You could even say that all the transformations in this chapter are roughly equivalent since they let you transform a function into another one that fixes some parameters, leaving others free and eventually leading to the same result. Okay, I agree, this isn't very clear! So, let's start by clearing the air, and providing some short definitions, which we will expand on later. (If you feel that your eyes are glazing over, please just skip this section and come back to it later!) Yes, you may find the following descriptions a bit perplexing, but bear with us—we'll go into more detail in just a bit:

- *Currying* is the process of transforming an m -ary function (that is, a function of arity m) into a sequence of m unary functions, each of which receives one argument of the original function, from left to right. (The first function receives the first argument of the original function, and returns a second function that receives the second argument, and returns a third function that receives the third argument, and so on.) Upon being called with an argument, each function produces the next one in the sequence, and the last one does the actual calculations.
- *Partial application* is the idea of providing n arguments to an m -ary function, being n less than or equal to m , to transform it into a function with $(m-n)$ parameters. Each time you provide some arguments, a new function is produced, with smaller arity. When you provide the last arguments, the actual calculations are performed.
- *Partial currying* is a mixture of both of the preceding ideas: you provide n arguments (from left to right) to an m -ary function and you produce a new function of arity $(m-n)$. When this new function receives some other arguments, also from left to right, it will produce yet another function. When the last parameters are provided, the function produces the correct calculations.

In this chapter, we are going to see these three transformations, what they require, and ways of implementing them. With respect to this, we will go into more than one way of coding each higher-order function and that will give us several insights into interesting ways of coding JavaScript, which you might find interesting for other applications.

Currying

We already mentioned currying back in the *Arrow functions* section of Chapter 1, *Becoming Functional – Several Questions*, and in the *One argument or many?* section of Chapter 3, *Starting Out with Functions – A Core Concept*, but let's be more thorough here. Currying is a technique that enables you to only work with single-variable functions, even if you need a multiple-variable one.



The idea of converting a multi-variable function into a series of single-variable functions (or, more rigorously, reducing operators with several operands, to a sequence of applications of a single operand operator) was worked on by Moses Schönfinkel, and there have been some authors who suggest, not necessarily tongue-in-cheek, that currying would be more correctly named *Schönfinkeling!*

In the next sections, we will first see how to deal with functions that have many parameters, and then we'll move on to see how to do currying by hand, or by using `bind()` or `eval()`.

Dealing with many parameters

The idea of currying, by itself, is simple. If you need a function with, say, three parameters, you could write something like the following by using arrow functions:

```
const make3 = (a, b, c) => String(100 * a + 10 * b + c);
```

Alternatively, you can have a sequence of functions, each with a single parameter, as shown here:

```
const make3curried = a => b => c => String(100 * a + 10 * b + c);
```

Alternatively, you might want to consider them as nested functions, like the following code snippet:

```
const make3curried2 = function(a) {  
  return function(b) {  
    return function(c) {  
      return String(100 * a + 10 * b + c);  
    };  
  };  
};
```


In terms of usage, there's an important difference in how you'd use each function. While you would call the first in the usual fashion, such as `make3(1, 2, 4)`, that wouldn't work with the second definition. Let's work out why: `make3curried()` is a *unary* (single parameter), so we should write `make3curried(1)`. But what does this return? According to the preceding definition, this also returns a unary function—and *that* function also returns a unary function! So, the correct call to get the same result as with the ternary function would be `make3curried(1)(2)(4)`! See *Figure 7.1*:

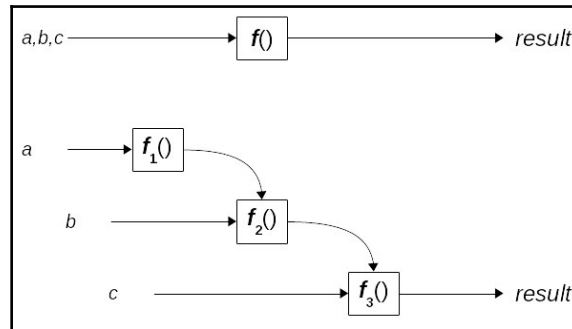


Figure 7.1: The difference between a common function and a curried equivalent.

Study this carefully—we have the first function, and when we apply an argument to it, we get a second function. Applying an argument to it produces a third function and a final application produces the desired result. This can be seen as a needless exercise in theoretical computing, but it actually brings some advantages, because you can then always work with unary functions, even if you need functions with more parameters.



Since there is a currying transformation, there is also an uncurrying one! In our case, we would write `make3uncurried = (a, b, c) => make3curried(a)(b)(c)` to revert the currying process and make it usable, once again, to provide all parameters in one sitting.

In some languages, such as Haskell, functions are only allowed to take a single parameter—but then again, the syntax of the language allows you to invoke functions as if multiple parameters were permitted. For our example, in Haskell, writing `make3curried 1 2 4` would have produced the result `124`, without anybody even needing to be aware that it involved *three* function calls, each with one of our arguments. Since you don't write parentheses around parameters, and you don't separate them with commas, you cannot tell that you are not providing a triplet of values instead of three singular ones.

Currying is basic in Scala or Haskell, which are fully functional languages, but JavaScript has enough features to allow us to define and use currying in our work. It won't be as easy since, after all, it's not built-in—but we'll be able to manage.

So, to review the basic concepts, the key differences between our original `make3()` and `make3curried()` are as follows:

- `make3()` is a ternary function, but `make3curried()` is unary.
- `make3()` returns a string; `make3curried()` returns another function—which, itself, returns a *second* function, which returns yet a *third* function, which finally does return a string!
- You can produce a string by writing something like `make3(1, 2, 4)`, which returns `124`, but you'll have to write `make3curried(1)(2)(4)` to get the same result.

Why would you go to all this bother? Let's just look at a simple example, and further on we will look at more examples. Suppose you had a function that calculated the **Value-added Tax (VAT)** for an amount, as shown here:

```
const addVAT = (rate, amount) => amount * (1 + rate / 100);

addVAT(20, 500); // 600 -- that is, 500 + 20%
addVAT(15, 200); // 230 -- 200 +15%
```

If you had to apply a single, constant rate, you could then curry the `addVAT()` function, to produce a more specialized version that is always applied your given rate. For example, if your national rate was 6%, you could then have something like the following:

```
const addVATcurried = rate => amount => amount * (1 + rate / 100);

const addNationalVAT = addVATcurried(6);

addNationalVAT(1500); // 1590 -- 1500 + 6%
```

The first line defines a curried version of our VAT-calculating function. Given a tax rate, `addVATcurried()` returns a new function, which when given an amount of money, finally adds the original tax rate to it. So, if the national tax rate were 6%, then `addNationalVAT()` would be a function that added 6% to any amount given to it. For example, if we were to calculate `addNationalVAT(1500)`, as in the preceding code, the result would be 1590: \$1500, plus 6% tax.

Of course, you would probably be justified in saying that this currying thing is a bit too much just to add 6% tax, but the simplification is what counts. Let's look at one more example. In your application, you may want to include some logging, with a function such as the following:

```
let myLog = (severity, logText) => {  
  // display logText in an appropriate way,  
  // according to its severity ("NORMAL", "WARNING", or "ERROR")  
};
```

However, with this approach, every time you wanted to display a normal log message, you would write `myLog("NORMAL", some normal text)`, and for warnings, you'd write `myLog("WARNING", some warning)`—but you could simplify this a bit with currying, by fixing the first parameter of `myLog()` as follows, with a `curry()` function that we'll look at later. Our code could then be as follows:

```
myLog = curry(myLog);  
// replace myLog by a curried version of itself  
  
const myNormalLog = myLog("NORMAL");  
const myWarningLog = myLog("WARNING");  
const myErrorLog = myLog("ERROR");
```

What do you gain? Now you can just write `myNormalLog("some normal text")` or `myWarningLog("some warning")`, because you have curried `myLog()` and then fixed its argument—this makes for simpler, easier-to-read code!

By the way, if you prefer, you could have also achieved the same result in a single step, with the original uncurried `myLog()` function, by currying it case by case:

```
const myNormalLog2 = curry(myLog)("NORMAL");  
const myWarningLog2 = curry(myLog)("WARNING");  
const myErrorLog2 = curry(myLog)("ERROR");
```

So, having a `curry()` function lets you fix some arguments while leaving others still open; let's see how to do this in three different ways.

Currying by hand

Before trying more complex things, we could curry a function by hand, without any special auxiliary functions or anything else. And, in fact, if we just want to implement currying for a special case, there's no need to do anything complex, because we can manage with simple arrow functions: we saw that for both `make3curried()` and `addVATcurried()`, so there's no need to revisit that idea.

Instead, let's look into some ways of doing that automatically, so we will be able to produce an equivalent curried version of any function, even without knowing its arity beforehand. Going further, we might want to code a more intelligent version of a function that could work differently depending on the number of received arguments. For example, we could have a `sum(x, y)` function that behaved as in the following examples:

```
sum(3, 5); // 8; did you expect otherwise?

const add3 = sum(3);

add3(5);    // 8

sum(3)(5); // 8 -- as if it were curried
```

We can achieve that behavior by hand. Our function would be something like the following:

```
const sum = (x, y) => {
  if (x !== undefined && y !== undefined) {
    return x + y;
  } else if (x !== undefined && y == undefined) {
    return z => sum(x, z);
  } else {
    return sum;
  }
};
```

Let's recap what we did here. Our curried-by-hand function has this behavior:

- If we call it with two arguments, it adds them, and returns the sum; this provides our first use case, as in `sum(3, 5) == 8`.
- If only one argument is provided, it returns a new function. This new function expects a single argument, and will return the sum of that argument and the original one: this behavior is what we expected in the other two use cases, such as `add2(3) == 5` or `sum(2)(7) == 9`.
- Finally, if no arguments are provided, it returns itself. This means that we would be able to write `sum()(1)(2)` if we desired. (No, I cannot think of a reason for wanting to write that.)

So, if we want, we can incorporate currying in the definition itself of a function. However, you'll have to agree that having to deal with all the special cases in each function could easily become troublesome, as well as error-prone. So, let's try to work out some more generic ways of accomplishing the same result, without any kind of particular coding.

Currying with bind()

We can find a solution to currying by using the `bind()` method. This allows us to fix one argument (or more, if need be; we won't be needing to do that here, but later on we will use it) and provide a function with that fixed argument. Of course, many libraries (such as Lodash, Underscore, Ramda, and others) provide this functionality, but we want to see how to implement that by ourselves.



Read more on `.bind()` at https://developer.mozilla.org/en/docs/Web/JavaScript/Reference/Global_objects/Function/bind—it will be useful since we'll take advantage of this method at other points in this chapter.

Our implementation is quite short but will require some explanation:

```
const curryByBind = fn =>
  fn.length === 0 ? fn() : p => curryByBind(fn.bind(null, p));
```

Start by noticing that `curryByBind()` always returns a new function, which depends on the `fn` function given as its parameter. If the function has no (more) parameters left (when `fn.length===0`) because all parameters have already been fixed, we can simply evaluate it by using `fn()`. Otherwise, the result of currying the function will be a new function that receives a single argument, and itself produces a newly curried function, with another fixed argument. Let's see this in action, with a detailed example, using the `make3()` function we saw at the beginning of this chapter once again:

```
const make3 = (a, b, c) => String(100 * a + 10 * b + c);

// f1 is a function that will fix make3's 1st parameter
const f1 = curryByBind(make3);

// f2 is a function that will fix make3's 2nd parameter
const f2 = f1(6);

// f3 is a function that will fix make3's last parameter
const f3 = f2(5);

// "658" will be now calculated, since there are
// no more parameters to fix
const f4 = f3(8);
```

The explanation of this code is as follows:

- The first function, `f1()`, has not received any arguments yet. Its result is a function of a single parameter, which will itself produce a curried version of `make3()`, with its first argument fixed to whatever it's given.
- Calling `f1(6)` produces a new unary function, `f2()`, which will itself produce a curried version of `make3()`—but with its first argument set to 6, so actually the new function will end up fixing the second parameter of `make3()`.
- Similarly, calling `f2(5)` produces yet a third unary function, `f3()`, which will produce a version of `make3()`, but fixing its third argument, since the first two have already been fixed.
- Finally, when we calculate `f3(8)`, this fixes the last parameter of `make3()` to 8, and since there are no more arguments left, the thrice-bound `make3()` function is called and the result "658" is produced.

If you wanted to curry the function by hand, you could use JavaScript's `.bind()` method. The sequence would be as follows:

```
const step1 = make3.bind(null, 6);
const step2 = step1.bind(null, 5);
const step3 = step2.bind(null, 8);

step3(); // "658"
```

In each step, we provide a further parameter. (The `null` value is required, to provide context. If it were a method attached to an object, we would provide that object as the first parameter to `.bind()`. Since that's not the case, `null` is expected.) This is equivalent to what our code does, with the exception that the last time, `curryByBind()` does the actual calculation, instead of making you do it, as in `step3()`.

Testing this transformation is rather simple—because there are not many possible ways of currying:

```
const make3 = (a, b, c) => String(100 * a + 10 * b + c);

describe("with curryByBind", function() {
  it("you fix arguments one by one", () => {
    const make3a = curryByBind(make3);
    const make3b = make3a(1)(2);
    const make3c = make3b(3);
    expect(make3c).toBe(make3(1, 2, 3));
  });
});
```

What else could you test? Maybe functions with just one parameter could be added, but there are no more to try.

If we wanted to curry a function with a variable number of parameters, then using `fn.length` wouldn't work; it only has a value for functions with a fixed number of parameters. We can solve this simply, by providing the desired number of arguments:

```
const curryByBind2 = (fn, len = fn.length) =>
  len === 0 ? fn() : p => curryByBind2(fn.bind(null, p), len - 1);

const sum2 = (...args) => args.reduce((x, y) => x + y, 0);
sum2.length; // 0; curryByBind() wouldn't work

sum2(1, 5, 3); // 9
sum2(1, 5, 3, 7); // 16
sum2(1, 5, 3, 7, 4); // 20

curriedSum5 = curryByBind2(sum2, 5); // curriedSum5 will expect 5
parameters
curriedSum5(1)(5)(3)(7)(4); // 20
```

The new `curryByBind2()` function works as before, but instead of depending on `fn.length`, it works with the `len` parameter, which defaults to `fn.length`, for standard functions with a constant number of parameters. Notice that when `len` isn't 0, the returned function calls `curryByBind2()` with `len-1` as its last argument—this makes sense, because if one argument has just been fixed, then there is one fewer parameter left to fix.

In our example, the `sum()` function can work with any number of parameters, and JavaScript informs us that `sum.length` is zero. However, when currying the function, if we set `len` to 5, currying will be done as if `sum()` was a five-parameter function—and the last line in the preceding code shows that this is really the case.

As before, testing is rather simple, given that we have no variants to try:

```
const sum2 = (...args) => args.reduce((x, y) => x + y, 0);

describe("with curryByBind2", function() {
  it("you fix arguments one by one", () => {
    const suma = curryByBind2(sum2, 5);
    const sumb = suma(1)(2)(3)(4)(5);
    expect(sumb).toBe(sum2(1, 2, 3, 4, 5));
  });
  it("you can also work with arity 1", () => {
    const suma = curryByBind2(sum2, 1);
    const sumb = suma(111);
    expect(sumb).toBe(sum2(111));
  });
});
```

```
});  
});
```

We tested setting the arity of the curried function to 1, as a border case, but there are no more possibilities.

Currying with eval()

There's another interesting way of currying a function—by creating a new one by means of `eval()`. Yes—that unsafe, dangerous `eval()`! (Remember what we said earlier: this is for learning purposes, but you'll be better off avoiding the potential security headaches that `eval()` can bring!) We will also be using the `range()` function that we wrote in the *Working with ranges* section of Chapter 5, *Programming Declaratively – A Better Style*.



Languages such as LISP have always had the possibility of generating and executing LISP code. JavaScript shares that functionality, but it's not often used—mainly because of the dangers it may entail! However, in our case, since we want to generate new functions, it seems logical to take advantage of this neglected capability.

The idea is simple: in the *A bit of theory* section (earlier in this chapter), we saw that we could easily curry a function by using arrow functions, as shown here:

```
const make3 = (a, b, c) => String(100 * a + 10 * b + c);  
  
const make3curried = a => b => c => String(100 * a + 10 * b + c);
```

Let's apply a couple of changes to the second version, to rewrite it in a way that will help us, as you'll see. First, we can just change the names of the parameters, and directly call the original `make3()` function:

```
const make3curried = x1 => x2 => x3 => make3(x1, x2, x3);
```

Why are we doing this? The answer is short: to help generate the required code automatically. We will be using the `range()` function we wrote back in the *Working with ranges* section of Chapter 5, *Programming Declaratively – A Better Style*, to avoid needing to write an explicit loop:

```
const range = (start, stop) =>  
  new Array(stop - start).fill(0).map((v, i) => start + i);  
  
const curryByEval = (fn, len = fn.length) =>  
  eval(`${range(0, len).map(i => `x${i}`)}.join(">")} =>  
    ${fn.name}${range(0, len).map(i => `x${i}`).join(",")}`);
```


This is quite a chunk of code to digest and, in fact, it should instead be coded in several separate lines to make it more understandable. Let's see how this works when applied to the `make3()` function as input:

1. The `range()` function produces an array with the values `[0, 1, 2]`. If we don't provide a `len` argument, `make3.length` (that is, `3`) will be used.
2. We use `map()` to generate a new array with the values `["x0", "x1", "x2"]`.
3. We `join()` the values in that array to produce `x0=>x1=>x2`, which will be the beginning of the code that we will `eval()`.
4. We then add an arrow, the name of the function, and an opening parenthesis, to make the middle part of our newly generated code: `=> make3(`.
5. We use `range()`, `map()`, and `join()` again, but this time to generate a list of arguments: `x0, x1, x2`.
6. We finally add a closing parenthesis, and after applying `eval()`, we get the curried version of `make3()`.

After following all these steps, in our case, the resulting function would be as follows:

```
curryByEval(make3); // x0=>x1=>x2=> make3(x0, x1, x2)
```

There's only one problem: if the original function didn't have a name, the transformation wouldn't work. (For more about that, check out the *Of lambdas and functions* section of Chapter 3, *Starting Out with Functions – A Core Concept*.) We could work around the function name problem by including the actual code of the function to be curried:

```
const curryByEval2 = (fn, len = fn.length) =>
  eval(`${range(0, len).map(i => `x${i}`).join(">")} =>
    (${fn.toString()}) (${range(0, len).map(i => `x${i}`).join(",")})`);
```

The only change is that instead of including the original function name, we substitute its actual code:

```
curryByEval2(make3); // x0=>x1=>x2=> ((a,b,c) => 100*a+10*b+c)(x0, x1, x2)
```

The produced function is surprising, having a full function followed by its parameters—but that's actually valid JavaScript! In fact, instead of the `add()` function, as follows, you could also write the function definition followed by its arguments, as in the last line in the following code:

```
const add = (x, y) => x + y;
add(2, 5); // 7

((x, y) => x + y)(2, 5); // 7
```

When you want to call a function, you write it, and follow with its arguments within parentheses—so that's all we are doing, even if it looks weird! We are now done with currying, possibly the best known FP technique, so let's move on to partial application, so you'll have even more flexibility for your own coding.

Partial application

The second transformation that we will be considering lets you fix some of the parameters of the function, creating a new function that will receive the rest of them. Let's make this clear with a nonsense example. Imagine you have a function with five parameters. You might want to fix the second and fifth parameters, and partial application would then produce a new version of the function that fixed those two parameters but left the other three open for new calls. If you called the resulting function with the three required arguments, it would produce the correct answer, by using the original two fixed parameters plus the newly provided three.



The idea of specifying only some of the parameters in function application, producing a function of the remaining parameters, is called **projection**: you are said to be *projecting* the function onto the remaining arguments. We will not use this term, but I wanted to cite it, just in case you happen to find it somewhere else.

Let's consider an example, using the `fetch()` API, which is widely considered to be the modern way to go for Ajax calls. You might want to fetch several resources, always specifying the same parameters for the call (for example, request headers) and only changing the URL to search. So, by using partial application, you could create a new `myFetch()` function that would always provide fixed parameters.



TIP

You can read more on `fetch()` at https://developer.mozilla.org/en-US/docs/Web/API/Fetch_API/Using_Fetch. According to <http://caniuse.com/#search=fetch>, you can use it in most browsers, except for (oh, surprise!) Internet Explorer, but you can get around this limitation with a polyfill, such as the one found at <https://github.com/github/fetch>.

Let's assume we have a `partial()` function that implements this kind of application and see how we'd use that to produce our new version of `fetch()`:

```
const myParameters = {
  method: "GET",
  headers: new Headers(),
  cache: "default"
```

```
};

const myFetch = partial(fetch, undefined, myParameters);
// undefined means the first argument for fetch is not yet defined
// the second argument for fetch() is set to myParameters

myFetch("a/first/url")
  .then(/* do something */)
  .catch(/* on error */);

myFetch("a/second/url")
  .then(/* do something else */)
  .catch(/* on error */);
```

If the request parameters had been the first argument for `fetch()`, currying would have worked. (We'll have more to say about the order of parameters later.) With partial application, you can replace any arguments, no matter which, so in this case, `myFetch()` ends up as a unary function. This new function will get data from any URL you wish, always passing the same set of parameters for the GET operation.

Partial application with arrow functions

Trying to do partial application by hand, as we did with currying, is too complicated. For instance, for a function with five parameters, you would have to write code that would allow the user to provide any of the 32 possible combinations of fixed and unfixed parameters, 32 being equal to 2 raised to the fifth power. And, even if you could simplify the problem, it would still remain hard to write and maintain. See *Figure 7.2* for one of many possible combinations:

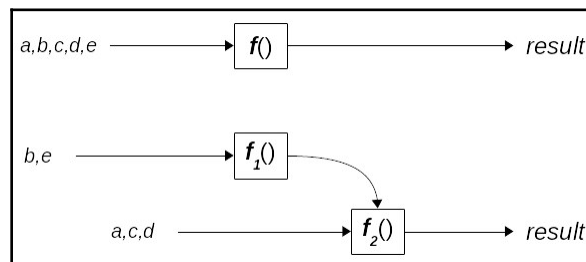


Figure 7.2: Partial application may let you first provide some parameters, and then provide the rest, to finally get the result.

Doing partial application with arrow functions, however, is much simpler. With the example we mentioned previously, we would have something like the following code. In this case, we will assume we want to fix the second parameter to 22, and the fifth parameter to 1960:

```
const nonsense = (a, b, c, d, e) => `${a}/${b}/${c}/${d}/${e}`;  
  
const fix2and5 = (a, c, d) => nonsense(a, 22, c, d, 1960);
```

Doing partial application in this way is quite simple, though we may want to find a more general solution. You can set any number of parameters, by creating a new function out of the previous one but fixing some more parameters. (Wrappers as in the previous [Chapter 6, Producing Functions - Higher Order Functions](#), could be used.) For instance, you might now want to also fix the last parameter of the new `fix2and5()` function to 9, as shown in the following code; there's nothing easier:

```
const fixLast = (a, c) => fix2and5(a, c, 9);
```

You might also have written `nonsense(a, 22, c, 9, 1960)`, if you wished to, but the fact remains that fixing parameters by using arrow functions is simple. Let's now consider, as we said, a more general solution.

Partial application with `eval()`

If we want to be able to do partial application fixing of any combination of parameters, we must have a way to specify which arguments are to be left free and which will be fixed from that point on. Some libraries, such as Underscore or Lodash, use a special object, `_`, to signify an omitted parameter. In this fashion, still using the same `nonsense()` function, we would write the following:

```
const fix2and5 = _.partial(nonsense, _, 22, _, _, 1960);
```

We could do the same sort of thing by having a global variable that would represent a pending, not yet fixed argument, but let's make it simpler, and just write `undefined` to represent a missing parameter.



When checking for `undefined`, remember to always use the `===` operator; with `==`, it happens that `null===undefined`, and you don't want that. See https://developer.mozilla.org/en/docs/Web/JavaScript/Reference/Global_Objects/undefined for more on this.

We want to write a function that will partially apply some arguments and leave the rest open for the future. We want to write code similar to the following and produce a new function in the same fashion as we earlier did with arrow functions:

```
const nonsense = (a, b, c, d, e) => `${a}/${b}/${c}/${d}/${e}`;

const fix2and5 = partialByEval(
  nonsense,
  undefined,
  22,
  undefined,
  undefined,
  1960
);
// fix2and5 would become (X0, X2, X3) => nonsense(X0, 22, X2, X3, 1960);
```

We can go back to using `eval()` and work out something like the following:

```
const range = (start, stop) =>
  new Array(stop - start).fill(0).map((v, i) => start + i);

const partialByEval = (fn, ...args) => {
  const rangeArgs = range(0, fn.length);

  const leftList = rangeArgs
    .map(v => (args[v] === undefined ? `x${v}` : null))
    .filter(v => !!v)
    .join(",");

  const rightList = rangeArgs
    .map(v => (args[v] === undefined ? `x${v}` : args[v]))
    .join(",");

  return eval(`${leftList} => ${fn.name}(${rightList})`);
};
```

Let's break down this function step by step. Once again, we are using our `range()` function:

- `rangeArgs` is an array with numbers from zero up to (but not including) the number of parameters in the input function.
- `leftList` is a string, representing the list of variables that haven't been applied. In our example, it would be `"X0,X2,X3"`, since we did provide values for the second and fifth arguments. This string will be used to generate the left part of the arrow function.

- `rightList` is a string, representing the list of the parameters for the call to the provided function. In our case, it would be `"X0, 'Z', X2, X3, 1960"`. We will use this string to generate the right part of the arrow function.

After having generated both lists, the remaining part of the code consists of just producing the appropriate string and giving it to `eval()` to get back a function.



If we were doing partial application on a function with a variable number of arguments, we could have substituted `args.length` for `fn.length`, or provided an extra (optional) parameter with the number to use, as we did in the *Currying* section of this chapter.

By the way, I deliberately expressed this function in this long way, to make it more clear. (We already saw somewhat similar—though shorter—code, when we did currying using `eval()`.) However, be aware that you might also find a shorter, more intense and obscure version, and that's the kind of code that gives FP a bad name! Our new version of the code could be:

```
const partialByEval2 = (fn, ...args) =>
  eval(
    `(${range(0, fn.length)
      .map(v => (args[v] === undefined ? `x${v}` : null))
      .filter(v => !!v)
      .join(",")})) => ${fn.name}(${range(0, fn.length)
      .map(v => (args[v] === undefined ? `x${v}` : args[v]))
      .join(",")}))`
  );
```

Let's finish this section by writing some tests. Here are some things we should consider:

- When we do partial application, the arity of the produced function should decrease.
- The original function should be called when arguments are in the correct order.

We could write something like the following, allowing the fixing of arguments in different places. Instead of using a spy or mock, we can directly work with the `nonsense()` function we had because it's quite efficient:

```
const nonsense = (a, b, c, d, e) => `${a}/${b}/${c}/${d}/${e}`;

describe("with partialByEval()", function() {
  it("you could fix no arguments", () => {
    const nonsensePC0 = partialByEval(nonsense);
    expect(nonsensePC0.length).toBe(5);
    expect(nonsensePC0(0, 1, 2, 3, 4)).toBe(nonsense(0, 1, 2, 3, 4));
  });
});
```

```
});
it("you could fix only some initial arguments", () => {
  const nonsensePC1 = partialByEval(nonsense, 1, 2, 3);
  expect(nonsensePC1.length).toBe(2);
  expect(nonsensePC1(4, 5)).toBe(nonsense(1, 2, 3, 4, 5));
});
it("you could skip some arguments", () => {
  const nonsensePC2 = partialByEval(
    nonsense,
    undefined,
    22,
    undefined,
    44
  );
  expect(nonsensePC2.length).toBe(3);
  expect(nonsensePC2(11, 33, 55)).toBe(nonsense(11, 22, 33, 44, 55));
});
it("you could fix only some last arguments", () => {
  const nonsensePC3 = partialByEval(
    nonsense,
    undefined,
    undefined,
    undefined,
    444,
    555
  );
  expect(nonsensePC3.length).toBe(3);
  expect(nonsensePC3(111, 222, 333)).toBe(
    nonsense(111, 222, 333, 444, 555)
  );
});
it("you could fix ALL the arguments", () => {
  const nonsensePC4 = partialByEval(nonsense, 6, 7, 8, 9, 0);
  expect(nonsensePC4.length).toBe(0);
  expect(nonsensePC4()).toBe(nonsense(6, 7, 8, 9, 0));
});
});
```

We wrote a partial application higher-order function, but it's not as flexible as we would like. For instance, we can fix a few arguments in the first instance, but then we have to provide all the rest of the arguments in the next call. It would be better if, after calling `partialByEval()`, we got a new function, and if we didn't provide all required arguments, we would get yet another function, and another, and so on, until all parameters had been provided—somewhat along the lines of what happens with currying. So, let's change the way we're doing partial application and consider another solution.

Partial application with closures

Let's examine yet another way of doing partial application, by using closures. (You may want to go over that topic in [Chapter 1, *Becoming Functional – Several Questions*](#).) This way of doing partial application will behave in a fashion somewhat reminiscent of the `curry()` functions we wrote earlier in this chapter, and solve the lack of flexibility that we mentioned at the end of the previous section. Our new implementation would be as follows:

```
const partialByClosure = (fn, ...args) => {
  const partialize = (...args1) => (...args2) => {
    for (let i = 0; i < args1.length && args2.length; i++) {
      if (args1[i] === undefined) {
        args1[i] = args2.shift();
      }
    }

    const allParams = [...args1, ...args2];
    return (allParams.includes(undefined) || allParams.length < fn.length
      ? partialize
      : fn)(...allParams);
  };

  return partialize(...args);
};
```

Wow—a longish bit of code! The key is the inner `partialize()` function. Given a list of parameters (`args1`), it produces a function that receives a second list of parameters (`args2`):

- First, it replaces all possible undefined values in `args1` with values from `args2`.
- Then, if any parameters are left in `args2`, it also appends them to those of `args1`, producing `allParams`.
- Finally, if that list of arguments does not include any more undefined values, and it is sufficiently long, it calls the original function.
- Otherwise, it partializes itself, to wait for more parameters.

An example will make it more clear. Let's go back to our trusty `make3()` function and construct a partial version of it:

```
const make3 = (a, b, c) => String(100 * a + 10 * b + c);
const f1 = partialByClosure(make3, undefined, 4);
```


Now let's write a second function:

```
const f2 = f1(7);
```

What happens? The original list of parameters (`[undefined, 4]`) gets merged with the new list (a single element—in this case, `[7]`), producing a function that now receives 7 and 4 as its first two arguments. However, this isn't yet ready, because the original function requires three arguments. We could write the following:

```
const f3 = f2(9);
```

Then, the current list of arguments would be merged with the new argument, producing `[7, 4, 9]`. Since the list is now complete, the original function will be evaluated, producing 749 as the final result.

There are important similarities between the structure of this code and the other higher-order function we wrote earlier, in the *Currying with bind()* section:

- If all the arguments have been provided, the original function is called.
- Otherwise, if some arguments are still required (when currying, it's just a matter of counting arguments; when doing partial application, you must also consider the possibility of having some undefined parameters), the higher-order function calls itself to produce a new version of the function that will *wait* for the missing arguments.

Let's finish by writing some tests that will show the enhancements in our new way of doing partial application. Basically, all the tests we did earlier would work, but we must also try applying arguments in sequence, so we should get the final result after two or more application steps. However, since we can now call our intermediate functions with any number of parameters, we cannot test arities: for all those intermediate functions, we get that `function.length===0`. Our tests could be as follows:

```
describe("with partialByClosure()", function() {
  it("you could fix no arguments", () => {
    const nonsensePC0 = partialByClosure(nonsense);
    expect(nonsensePC0(0, 1, 2, 3, 4)).toBe(nonsense(0, 1, 2, 3, 4));
  });

  it("you could fix only some initial arguments, and then some more", () =>
  {
    const nonsensePC1 = partialByClosure(nonsense, 1, 2, 3);
    const nonsensePC1b = nonsensePC1(undefined, 5);
    expect(nonsensePC1b(4)).toBe(nonsense(1, 2, 3, 4, 5));
  });
});
```

```
});

it("you could skip some arguments", () => {
  const nonsensePC2 = partialByClosure(
    nonsense,
    undefined,
    22,
    undefined,
    44
  );
  expect(nonsensePC2(11, 33, 55)).toBe(nonsense(11, 22, 33, 44, 55));
});

it("you could fix only some last arguments", () => {
  const nonsensePC3 = partialByClosure(
    nonsense,
    undefined,
    undefined,
    undefined,
    444,
    555
  );
  expect(nonsensePC3(111)(222, 333)).toBe(
    nonsense(111, 222, 333, 444, 555)
  );
});

it("you could simulate currying", () => {
  const nonsensePC4 = partialByClosure(nonsense);
  expect(nonsensePC4(6)(7)(8)(9)(0)).toBe(nonsense(6, 7, 8, 9, 0));
});

it("you could fix ALL the arguments", () => {
  const nonsensePC5 = partialByClosure(nonsense, 16, 17, 18, 19, 20);
  expect(nonsensePC5()).toBe(nonsense(16, 17, 18, 19, 20));
});
});
```

The code is longer than before, but the tests themselves are easy to understand. The next-to-last test should remind you of currying, by the way! We have now seen how to do currying and partial application. Let's finish the chapter with a hybrid method, partial currying, which includes aspects of both techniques.

Partial currying

The last transformation we will look at is a sort of mixture of currying and partial application. If you google it, in some places you will find it called *currying*, and in others, *partial application*, but as it happens, it fits neither, so I'm sitting on the fence and calling it *partial currying*!

The idea of this is, given a function, to fix its first few arguments and produce a new function that will receive the rest of them. However, if that new function is given fewer arguments, it will fix whatever it was given and produce a newer function, to receive the rest of them, until all the arguments are given and the final result can be calculated. See *Figure 7.3*:

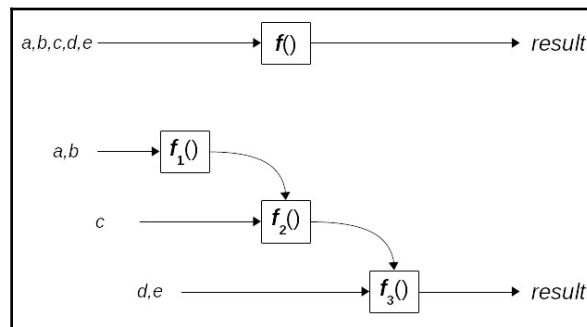


Figure 7.3: Partial currying is a mixture of currying and partial application. You may provide arguments from the left, in any quantity, until all have been provided, and then the result is calculated.

To look at an example, let's go back to the `nonsense()` function we have been using in previous sections, as follows. Assume we already have a `partialCurry()` function:

```

const nonsense = (a, b, c, d, e) => `${a}/${b}/${c}/${d}/${e}`;

const pcNonsense = partialCurry(nonsense);
const fix1And2 = pcNonsense(9, 22); // fix1And2 is now a ternary function
const fix3 = fix1And2(60); // fix3 is a binary function
const fix4and5 = fix3(12, 4); // fix4and5 === nonsense(9,22,60,12,4),
"9/22/60/12/4"
  
```

The original function had an arity of 5. When we *partial curry* that function, and give it arguments 9 and 22, it becomes a ternary function, because out of the original five parameters, two have become fixed. If we take that ternary function and give it a single argument (60), the result is yet another function: in this case, a binary one, because now we have fixed the first three of the original five parameters. The final call, providing the last two arguments, then does the job of actually calculating the desired result.

There are some points in common with currying and partial application, but also some differences, as follows:

- The original function is transformed into a series of functions, each of which produces the next one until the last in the series actually carries out its calculations.
- You always provide parameters starting from the first one (the leftmost one), as in currying, but you can provide more than one, as in partial application.
- When currying a function, all the intermediate functions are unary, but with partial currying that need not be so. However, if in each instance we were to provide a single argument, then the result would require as many steps as plain currying.

So, we have our definition—let's now see how we can implement our new higher-order function; we'll probably be reusing a few concepts from the previous sections in this chapter.

Partial currying with `bind()`

Similar to what we did with currying, there's a simple way to do partial currying. We will take advantage of the fact that `bind()` can actually fix many arguments at once:

```
const partialCurryingByBind = fn =>
  fn.length === 0
  ? fn()
  : (...pp) => partialCurryingByBind(fn.bind(null, ...pp));
```

Compare the code to the previous `curryByBind()` function and you'll see the very small differences:

```
const curryByBind = fn =>
  fn.length === 0
  ? fn()
  : p => curryByBind(fn.bind(null, p));
```

The mechanism is exactly the same. The only difference is that in our new function, we can bind many arguments at the same time, while in `curryByBind()` we always bind just one. We can revisit our earlier example—and the only difference is that we can get the final result in fewer steps:

```
const make3 = (a, b, c) => String(100 * a + 10 * b + c);

const f1 = partialCurryingByBind(make3);
```

```
const f2 = f1(6, 5); // f2 is a function, that fixes make3's first two
arguments
const f3 = f2(8); // "658" is calculated, since there are no more
parameters to fix
```

By the way, and just to be aware of the existing possibilities, you can fix some parameters when currying as shown here:

```
const g1 = partialCurryingByBind(make3)(8, 7);
const g2 = g1(6); // "876"
```

Testing this function is easy and the examples we provided are a very good starting point. Note, however, that since we allow the fixing of any number of arguments, we cannot test the arity of the intermediate functions. Our tests could be as follows, then:

```
const make3 = (a, b, c) => String(100 * a + 10 * b + c);

describe("with partialCurryingByBind", function() {
  it("you could fix arguments in several steps", () => {
    const make3a = partialCurryingByBind(make3);
    const make3b = make3a(1, 2);
    const make3c = make3b(3);
    expect(make3c).toBe(make3(1, 2, 3));
  });

  it("you could fix arguments in a single step", () => {
    const make3a = partialCurryingByBind(make3);
    const make3b = make3a(10, 11, 12);
    expect(make3b).toBe(make3(10, 11, 12));
  });

  it("you could fix ALL the arguments", () => {
    const make3all = partialCurryingByBind(make3);
    expect(make3all(20, 21, 22)).toBe(make3(20, 21, 22));
  });

  it("you could fix one argument at a time", () => {
    const make3one = partialCurryingByBind(make3)(30)(31)(32);
    expect(make3one).toBe(make3(30, 31, 32));
  });
});
```

Now, let's consider functions with a variable number of parameters. As before, we'll have to provide an extra value, and we'll get the following implementation:

```
const partialCurryingByBind2 = (fn, len = fn.length) =>
  len === 0
    ? fn()
    : (...pp) =>
      partialCurryingByBind2(
        fn.bind(null, ...pp),
        len - pp.length
      );
```

We can try this out in a simple way, revisiting our currying example from earlier in the chapter, but now using partial currying, as shown here:

```
const sum = (...args) => args.reduce((x, y) => x + y, 0);

pcSum5 = partialCurryingByBind2(sum2, 5);
// curriedSum5 will expect 5 parameters

pcSum5(1, 5)(3)(7, 4); // 20
```

When we called the new `pcSum5()` function with arguments (1,5), it produced a new function that expected three more. Providing it with one single parameter (3), a third function was created, to wait for the last two. Finally, when we provided the last two values (7,4) to that last function, the original function was called, to calculate the result (20).

We can also add some tests for this alternate way of doing partial currying:

```
const sum2 = (...args) => args.reduce((x, y) => x + y, 0);

describe("with partialCurryingByBind2", function() {
  it("you could fix arguments in several steps", () => {
    const suma = partialCurryingByBind2(sum2, 3);
    const sumb = suma(1, 2);
    const sumc = sumb(3);
    expect(sumc).toBe(sum2(1, 2, 3));
  });

  it("you could fix arguments in a single step", () => {
    const suma = partialCurryingByBind2(sum2, 4);
    const sumb = suma(10, 11, 12, 13);
    expect(sumb).toBe(sum(10, 11, 12, 13));
  });

  it("you could fix ALL the arguments", () => {
    const sumall = partialCurryingByBind2(sum2, 5);
```

```
    expect(sumall(20, 21, 22, 23, 24)).toBe(sum2(20, 21, 22, 23, 24));
  });

  it("you could fix one argument at a time", () => {
    const sumone = partialCurryingByBind2(sum2, 6)(30)(31)(32)(33)(34)(35);
    expect(sumone).toBe(sum2(30, 31, 32, 33, 34, 35));
  });
});
```

Trying out different arities is better than sticking to just one, so we did that for variety.

Partial currying with closures

As with partial application, there's a solution that works with closures. Since we have gone over many of the required details, let's jump directly into the code:

```
const partialCurryByClosure = fn => {
  const curryize = (...args1) => (...args2) => {
    const allParams = [...args1, ...args2];
    return (allParams.length < func.length ? curryize : fn)(
      ...allParams
    );
  };

  return curryize();
};
```

If you compare `partialCurryByClosure()` and `partialByClosure()`, the main difference is that with partial currying, since we are always providing arguments from the left, and there is no way to skip some, you concatenate whatever arguments you had with the new ones, and check whether you got enough. If the new list of arguments has reached the expected arity of the original function, you can call it and get the final result. In other cases, you just use `curryize()` to get a new intermediate function, which will wait for more arguments.

As earlier, if you have to deal with functions with a varying number of parameters, you can provide an extra argument to the partial currying function:

```
const partialCurryByClosure2 = (fn, len = fn.length) => {
  const curryize = (...args1) => (...args2) => {
    const allParams = [...args1, ...args2];
    return (allParams.length < len ? curryize : fn)(...allParams);
  };
  return curryize();
};
```

The results are exactly the same as in the previous section, *Partial currying with bind()*, so it's not worth repeating them. You could also easily change the tests we wrote to use `partialCurryByClosure()` instead of `partialCurryByBind()` and they would work.

Final thoughts

Let's finish this chapter with two more philosophical considerations regarding currying and partial application, which may cause a bit of a discussion:

- First, many libraries are just wrong as to the order of their parameters, making them harder to use.
- Second, I don't usually even use the higher-order functions in this chapter, going for simpler JavaScript code!

That's probably not what you were expecting at this time, so let's go over those two points in more detail, so you'll see it's not a matter of *do as I say, not as I do...* or *as the libraries do!*

Parameter order

There's a problem that's common to not only functions such as Underscore's or LoDash's `_.map(list, mappingFunction)` or `_.reduce(list, reducingFunction, initialValue)`, but also to some that we have produced in this book, such as the result of `demethodize()`, for example. (See the *Demethodizing - turning methods into functions* section of Chapter 6, *Producing Functions - Higher-Order Functions*, to review that higher-order function.) The problem is that the *order* of their parameters doesn't really help with currying.

When currying a function, you will probably want to store intermediate results. When we do something like in the code that follows, we assume that you are going to reuse the curried function with the fixed argument and that means that the first argument to the original function is the least likely to change. Let's now consider a specific case. Answer this question: what's more likely—that you'll use `map()` to apply the same function to several different arrays, or that you'll apply several different functions to the same array? With validations or transformations, the former is more likely, but that's not what we get!

We can write a simple function to flip the parameters for a binary function, as shown here:

```
const flipTwo = fn => (p1, p2) => fn(p2, p1);
```



Note that even if the original `fn()` function could receive more or fewer arguments, after applying `flipTwo()` to it, the arity of the resulting function will be fixed to 2. We will be taking advantage of this fact in the following section.

With this, you could then write code as follows:

```
const myMap = curry(flipTwo(demethodize(map)));
const makeString = v => String(v);

const stringify = myMap(makeString);
let x = stringify(anArray);
let y = stringify(anotherArray);
let z = stringify(yetAnotherArray);
```

The most common use case is that you'll want to apply the function to several different lists, and neither the library functions nor our own *demethodized* ones provide for that. However, by using `flipTwo()`, we can work in a fashion we would prefer.



In this particular case, we might have solved our problem by using partial application instead of currying, because with that we could fix the second argument to `map()` without any further bother. However, flipping arguments to produce new functions that have a different order of parameters is also an often-used technique, and it's important that you're aware of it.

For situations such as with `reduce()`, which usually receives three arguments (the list, the function, and the initial value), we may opt for this:

```
const flip3 = fn => (p1, p2, p3) => fn(p2, p3, p1);

const myReduce = partialCurry(flip3(demethodize(Array.prototype.reduce)));

const sum = (x, y) => x + y;
const sumAll = myReduce(sum, 0);
sumAll(anArray);
sumAll(anotherArray);
```

Here, we used partial currying to simplify the expression for `sumAll()`. The alternative would have been using common currying, and then we would have defined `sumAll = myReduce(sum)(0)`.

If you want, you can also go for more esoteric parameter rearranging functions, but you usually won't need more than these two. For really complex situations, you may instead opt for using arrow functions (as we did when defining `flipTwo()` and `flip3()`) and make it clear what kind of reordering you need.

Being functional

Now that we are nearing the end of this chapter, a confession is in order: I do not always use currying and partial application as shown above! Don't misunderstand me, I *do* apply those techniques—but sometimes it makes for longer, less clear, not necessarily better code. Let me show you what I'm talking about.

If I'm writing my own function and then I want to curry it in order to fix the first parameter, currying (or partial application, or partial currying) doesn't really make a difference, in comparison to arrow functions. I'd have to write the following:

```
const myFunction = (a, b, c) => { ... };
const myCurriedFunction = curry(myFunction)(fixed_first_argument);

// and later in the code...
myCurriedFunction(second_argument)(third_argument);
```

Currying the function, and giving it a first parameter, all in the same line, may be considered not so clear; the alternative calls for an added variable and one more line of code. Later, the future call isn't so good either; however, partial currying makes it simpler: `myPartiallyCurriedFunction(second_argument, third_argument)`. In any case, when I compare the final code with the use of arrow functions, I think the other solutions aren't really any better; make your own evaluation of the sample that follows:

```
const myFunction = (a, b, c) => { ... };
const myFixedFirst = (b, c) => myFunction(fixed_first_argument, b, c);

// and later...
myFixedFirst(second_argument, third_argument);
```

Where I do think that currying and partial application is quite good is in my small library of demethodized, pre-curried, basic higher-order functions. I have my own set of functions, such as the following:

```
const _plainMap = demethodize(Array.prototype.map);
const myMap = curry(_plainMap, 2);
const myMapX = curry(flipTwo(_plainMap));

const _plainReduce = demethodize(Array.prototype.reduce);
```

```
const myReduce = curry(_plainReduce, 3);
const myReduceX = curry(flip3(_plainReduce));

const _plainFilter = demethodize(Array.prototype.filter);
const myFilter = curry(_plainFilter, 2);
const myFilterX = curry(flipTwo(_plainFilter));

// ...and more functions in the same vein
```

Here are some points to note about the code:

- I have these functions in a separate module, and I only export the `myXXX()` named ones.
- The other functions are private, and I use the leading underscore to remind me of that.
- I use the `my...` prefix to remember that these are *my* functions and not the normal JavaScript ones. Some people would rather keep the standard names such as `map()` or `filter()`, but I prefer distinct names.
- Since most of the JavaScript methods have a variable arity, I had to specify it when currying.
- I always provide the third argument (the initial value for reducing) to `reduce()`, so the arity I chose for that function is three.
- When currying flipped functions, you don't need to specify the number of parameters, because flipping already does that for you.

In the end, it all comes down to a personal decision; experiment with the techniques that we've looked at in this chapter and see which ones you prefer!

Summary

In this chapter, we have considered a new way of producing functions, by fixing arguments to an existing function in several different ways: currying—which originally came from computer theory; partial application—which is more flexible; and partial currying, which combines good aspects from both of the previous methods. Using these transformations, you can simplify your coding, because you can generate more specialized versions of general functions, without any hassle.

In Chapter 8, *Connecting Functions – Pipelining and Composition*, we will turn back to some concepts we looked at in the chapter on pure functions, and we will consider ways of ensuring that functions cannot become *impure by accident*, by seeking ways to make their arguments immutable, making them impossible to mutate.

Questions

7.1. Sum as you will: The following exercise will help you understand some of the concepts we dealt with in this chapter, even if you solve it without using any of the functions we looked at. Write a `sumMany()` function that lets you sum an indeterminate quantity of numbers, in the following fashion. Note that when the function is called with no arguments, the sum is returned:

```
let result = sumMany((9) (2) (3) (1) (4) (3) ());  
// 22
```

7.2. Working stylishly: Write an `applyStyle()` function that will let you apply basic styling to strings, in the following way. Use either currying or partial application:

```
const makeBold = applyStyle("b");  
document.getElementById("myCity").innerHTML =  
makeBold("Montevideo");  
// <b>Montevideo</b>, to produce Montevideo  
  
const makeUnderline = applyStyle("u");  
document.getElementById("myCountry").innerHTML =  
makeUnderline("Uruguay");  
// <u>Uruguay</u>, to produce Uruguay
```

7.3. Currying by prototype: Modify `Function.prototype` to provide a `curry()` method that will work like the `curry()` function we saw in the chapter. Completing the following code should produce the following results:

```
Function.prototype.curry = function() {  
  // ...your code goes here...  
};  
  
const sum3 = (a, b, c) => 100 * a + 10 * b + c;  
sum3.curry() (1) (2) (4); // 124  
  
const sum3C = sum3.curry() (2) (2);  
sum3C(9); // 229
```

7.4. Uncurrying the curried: Write an `unCurry(fn, arity)` function that receives as arguments a (curried) function and its expected arity, and returns an uncurried version of `fn()`; that is, a function that will receive n arguments and produce a result (providing the expected arity is needed because you have no way of determining it on your own):

```
const make3 = (a, b, c) => String(100 * a + 10 * b + c);

const make3c = curry(make3);
console.log(make3c(1)(2)(3)); // 123

const remake3 = uncurry(make3c, 3);
console.log(remake3(1, 2, 3)); // 123
```

7.5. Mystery questions function: What does the following function, purposefully written in an unhelpful way, actually do?

```
const what = who => (...why) =>
  who.length <= why.length
    ? who(...why)
    : (...when) => what(who)(...why, ...when);
```

7.6. Yet more curry! Here is another proposal for a `curry()` function: can you see why it works? A hint: the code is related to something we saw in the chapter:

```
const curry = fn => (...args) =>
  args.length >= fn.length ? fn(...args) : curry(fn.bind(null, ...args));
```

8

Connecting Functions - Pipelining and Composition

In Chapter 7, *Transforming Functions – Currying and Partial Application*, we looked at several ways we can build new functions by applying higher-order functions. In this chapter, we will go to the core of FP and learn how to create sequences of function calls and how to combine them to produce a more complex result out of several simpler components. To do this, we will cover the following topics:

- **Pipelining:** A way to join functions together in a similar way to Unix/Linux pipes.
- **Chaining:** This may be considered a variant of pipelining, but is restricted to objects.
- **Composing:** This is a classic operation with its origins in basic computer theory.
- **Transducing:** An optimized way to compose map/filter/reduce operations.

Along the way, we will be touching on related concepts, such as the following:

- **Pointfree style**, which is often used with pipelining and composition
- **Debugging** composed or piped functions, for which we'll whip up some auxiliary tools
- **Testing** composed or piped functions, which won't prove to be of high complexity

Armed with these techniques, you'll be able to combine small functions to create larger ones, which is a characteristic of functional programming and will help you develop better code.

Pipelining

Pipelining and composition are techniques that are used to set up functions so that they work in sequence so that the output from a function becomes the input for the next function. There are two ways of looking at this: from a computer point of view, and from a mathematical point of view. We'll look at both in this section. Most FP texts start with the latter, but since I assume that most of you will prefer computers over math, let's start with the former instead.

Piping in Unix/Linux

In Unix/Linux, executing a command and passing its output as input to a second command, whose output will yield the input of a third command, and so on, is called a *pipeline*. This is quite a common application of the philosophy of Unix, as explained in a Bell Laboratories article, written by the creator of the pipelining concept himself, Doug McIlroy:

- Make each program do one thing well. To do a new job, build afresh rather than complicate old programs by adding new *features*.
- Expect the output of every program to become the input to another, so far unknown, program.



Given the historical importance of Unix, I'd recommend reading some of the seminal articles describing the (then new) operating system, in the *Bell System Technical Journal*, July 1978, at http://emulator.pdp-11.org.ru/misc/1978.07_-_Bell_System_Technical_Journal.pdf. The two quoted rules are in the *Style* section, in the *Foreword* article.

Let's consider a simple example to get started. Suppose I want to know how many LibreOffice text documents there are in a directory. There are many ways to do this, but this will do. We will execute three commands, piping (that's the meaning of the `|` character) each command's output as input to the next one. Suppose we go to `cd /home/fkerekki/Documents` and then do the following (please ignore the dollar sign, which is just the console prompt):

```
$ ls -l | grep "odt$" | wc -l
4
```

What does this mean? How does it work? We have to analyze this process step by step:

- The first part of the pipeline, `ls -l`, lists all the files in the directory (`/home/fkerekki/Documents`, as per our `cd` command), in a single column, one filename per line.
- The output from the first command is provided as input to `grep "odt$"`, which filters (lets pass) only those lines that finish with "odt", the standard file extension for LibreOffice Writer.
- The filtered output is provided to the counting command, `wc -l`, which counts how many lines there are in its input.



You can find out more about pipelines in Section 6.2, *Filters*, of the *UNIX Time-Sharing System* article by Dennis Ritchie and Ken Thompson, also in the issue of the Bell Laboratories journal that I mentioned previously.

From the point of view of FP, this is a key concept. We want to build more complex operations out of simple, single-purpose, shorter functions. Pipelining is what the Unix shell uses to apply that concept, which it does by simplifying the job of executing a command, taking its output, and providing it as input to yet another command. We will be applying similar concepts in our own functional style in JavaScript later:

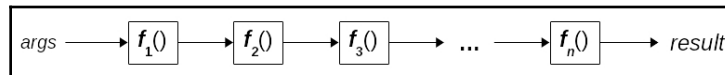


Figure 8.1: Pipelines in JavaScript are similar to Unix/Linux pipelines. The output of each function becomes the input for the next

By the way (and no—rest assured, this isn't turning into a shell tutorial!), you can also make pipelines accept parameters. For example, if I happened to want to count how many files I had with this or that extension, I could create a function such as `cfe`, standing for *count for extension*:

```

$ function cfe() {
  ls -l | grep "$1\$" | wc -l
}

```

Then, I could use `cfe` as a command, giving it the desired extension as an argument:

```

$ cfe odt
4
$ cfe pdf
6

```


`cfe` executes my pipeline and tells me I have 4 `odt` files (LibreOffice) and 6 `pdf` ones; nice! We will also want to write similar parametric pipelines: we are not constrained to only have fixed functions in our flow; we have full liberty as to what we want to include. Having worked in Linux, we can now go back to coding. Let's see how.

Revisiting an example

We can start tying ends together by revisiting a problem from a previous chapter. Remember when we had to calculate the average latitude and longitude for some geographic data that we looked at in the *Extracting data from objects* section of Chapter 5, *Programming Declaratively – A Better Style?* Basically, we started with some data such as the following and the problem was to calculate the average latitude and longitude of the given points:

```
const markers = [
  {name: "AR", lat: -34.6, lon: -58.4},
  {name: "BO", lat: -16.5, lon: -68.1},
  {name: "BR", lat: -15.8, lon: -47.9},
  {name: "CL", lat: -33.4, lon: -70.7},
  {name: "CO", lat: 4.6, lon: -74.0},
  {name: "EC", lat: -0.3, lon: -78.6},
  {name: "PE", lat: -12.0, lon: -77.0},
  {name: "PY", lat: -25.2, lon: -57.5},
  {name: "UY", lat: -34.9, lon: -56.2},
  {name: "VE", lat: 10.5, lon: -66.9},
];
```

With what we know, we can write a solution in terms of the following:

- Being able to extract the latitude (and, afterward, the longitude) from each point
- Using that function to create an array of latitudes
- Pipelining the resulting array to the average function we wrote in *Calculating an average* section of the aforementioned chapter

To do the first task, we can use the `myMap()` function from the *Parameters order* section of Chapter 7, *Transforming Functions – Currying and Partial Application*. For the second task, we can make do with the `getField()` function from the *Getting a property from an object* section of Chapter 6, *Producing Functions – Higher-Order Functions*, plus a bit of currying to fix some values. Finally, for the third task, we'll just use the (yet unwritten!) pipelining function we'll be developing soon! In full, our solution could look like this:

```
const average = arr => arr.reduce(sum, 0) / arr.length;
const getField = attr => obj => obj[attr];
```

```
const myMap = curry(flipTwo(demethodize(array.prototype.map)));

const getLat = curry(getField) ("lat");
const getAllLats = curry(myMap) (getLat);

let averageLat = pipeline(getAllLats, average);
// and similar code to average longitudes
```

Of course, you can always yield to the temptation of going for a couple of *one-liners*, but would it be much clearer or better?

```
let averageLat2 = pipeline(curry(myMap) (curry(getField) ("lat")), average);
let averageLon2 = pipeline(curry(myMap) (curry(getField) ("lon")), average);
```

Whether this makes sense to you will depend on your experience with FP. In any case, no matter which solution you take, the fact remains that adding pipelining (and later on, composition) to your set of tools can help you write tighter, declarative, simpler-to-understand code.

Now, let's learn how to pipeline functions in the right way.

Creating pipelines

We want to be able to generate a pipeline of several functions. We can do this in two different ways: by building the pipeline by hand, in a problem-specific way, or by seeking to use more generic constructs that can be applied with generality. Let's look at both.

Building pipelines by hand

Let's go with a Node example, similar to the command-line pipeline we built earlier in this chapter. Here, we'll build the pipeline we need by hand. We need a function to read all the files in a directory. We can do that (this isn't recommended because of the synchronous call, which is normally not good in a server environment) with something like this:

```
function getDir(path) {
  const fs = require("fs");
  const files = fs.readdirSync(path);
  return files;
}
```

Filtering the `odt` files is quite simple. We start with the following function:

```
const filterByText = (text, arr) => arr.filter(v => v.endsWith(text));
```

This function takes an array and filters out any elements that do not end with the given text. So, we can now write the following:

```
const filterOdt = arr => filterByText(".odt", arr);
```

Better still, we can apply currying and go for pointfree style, as shown in the *An unnecessary mistake* section of Chapter 3, *Starting Out with Functions – A Core Concept*:

```
const filterOdt2 = curry(filterByText)(".odt");
```

Both versions of the filtering function are equivalent; which one you use comes down to your tastes. Finally, to count elements in an array, we can simply write the following. Since `length` is not a function, we cannot apply our demethodizing trick:

```
const count = arr => arr.length;
```

With these functions, we could write something like this:

```
const countOdtFiles = path => {
  const files = getDir(path);
  const filteredFiles = filterOdt(files);
  const countOfFiles = count(filteredFiles);
  return countOfFiles;
};
```

```
countOdtFiles("/home/fkereki/Documents"); // 4, as with the command line
solution
```

We are essentially doing the same process as in Linux: getting the files, keeping only the `odt` ones, and counting how many files result from this. If you wanted to get rid of all the intermediate variables, you could also go for a *one-liner* definition that does exactly the same job in the very same way, albeit with fewer lines:

```
const countOdtFiles2 = path => count(filterOdt(getDir(path)));
```

```
countOdtFiles2("/home/fkereki/Documents"); // 4, as before
```

This gets to the crux of the matter: both implementations of our file-counting function have disadvantages. The first definition uses several intermediate variables to hold the results and makes a multiline function out of what was a single line of code in the Linux shell. The second, much shorter, definition, on the other hand, is quite harder to understand, insofar as we are writing the steps of the computation in seemingly reverse order! Our pipeline has to read files first, then filter them, and finally count them, but those functions appear *the other way round* in our definition!

We can certainly implement pipelining by hand, as we have seen, but it would be better if we could go for a more declarative style.

Let's move on and try to build a better pipeline in a more clear and understandable way by trying to apply some of the concepts we've already seen.

Using other constructs

If we think in functional terms, what we have is a list of functions and we want to apply them sequentially, starting with the first, then applying the second to whatever the first function produced as its result, and then applying the third to the second function's results, and so on. If we were just fixing a pipeline of two functions, you could use the following code:

```
const pipeTwo = (f, g) => (...args) => g(f(...args));
```

This is the basic definition we provided earlier in this chapter: we evaluate the first function, and its output becomes the input for the second function; quite straightforward! You might object, though, that this pipeline, of only two functions, is a bit too limited! This is not as useless as it may seem because we can compose longer pipelines—though I'll admit that it requires too much writing! Suppose we wanted to write our three-function pipeline (from the previous section); we could do so in two different, equivalent ways:

```
const countOdtFiles3 = path =>
  pipeTwo(pipeTwo(getDir, filterOdt), count)(path);

const countOdtFiles4 = path =>
  pipeTwo(getDir, pipeTwo(filterOdt, count))(path);
```



We are taking advantage of the fact that piping is an associative operation. In mathematics, the associative property is the one that says that we can compute $1+2+3$ either by adding $1+2$ first and then adding that result to 3, or by adding 1 to the result of adding $2+3$: in other terms, $1+2+3$ is the same as $(1+2)+3$ or $1+(2+3)$.

How do they work? How is it that they are equivalent? Following the execution of a given call will be useful; it's quite easy to get confused with so many calls! The first implementation can be followed step by step until the final result, which matches what we already know:

```
countOdtFiles3("/home/fkereki/Documents") ===
  pipeTwo(pipeTwo(getDir, filterOdt), count)("/home/fkereki/Documents") ===
  count(pipeTwo(getDir, filterOdt)("/home/fkereki/Documents")) ===
  count(filterOdt(getDir("/home/fkereki/Documents"))) // 4
```

The second implementation also comes to the same final result:

```
countOdtFiles4("/home/fkerekki/Documents") ===  
  pipeTwo(getDir, pipeTwo(filterOdt, count))("/home/fkerekki/Documents") ===  
  pipeTwo(filterOdt, count)(getDir("/home/fkerekki/Documents")) ===  
  count(filterOdt(getDir("/home/fkerekki/Documents"))) // 4
```

Both derivations arrived at the same final expression—the same we had written by hand earlier, in fact—so we now know that we can make do with just a basic *pipe of two* higher-order functions, but we'd really like to be able to work in a shorter, more compact way. A first implementation could be along the lines of the following:

```
const pipeline = (...fns) => (...args) => {  
  let result = fns[0](...args);  
  for (let i = 1; i < fns.length; i++) {  
    result = fns[i](result);  
  }  
  return result;  
};  
  
pipeline(getDir, filterOdt, count)("/home/fkerekki/Documents"); // still 4
```

This does work—and the way of specifying our file-counting pipeline is much clearer since the functions are given in their proper order. However, the implementation of the `pipeline()` function is not very functional and goes back to old, imperative, loop by hand methods. We can do better using `reduce()`, as we did in Chapter 5, *Programming Declaratively – A Better Style*.



If you check out some FP libraries, the function that we are calling `pipeline()` here may also be known as `flow()`—because data flows from left to right – or `sequence()`—alluding to the fact that operations are performed in ascending sequence—but the semantics are the same.

The idea is to start the evaluation with the first function, pass the result to the second, then that result to the third, and so on. By doing this, we can pipeline with shorter code:

```
const pipeline2 = (...fns) =>  
  fns.reduce((result, f) => (...args) => f(result(...args)));  
  
pipeline2(getDir, filterOdt, count)("/home/fkerekki/Documents"); // 4
```

This code is more declarative. However, you could have gone one better by writing it using our `pipeTwo()` function, which does the same thing but in a more concise manner:

```
const pipeline3 = (...fns) => fns.reduce(pipeTwo);

pipeline3(getDir, filterOdt, count)("/home/fkereki/Documents"); // again 4
```

You can understand this code by realizing that it uses the associative property that we mentioned previously and pipes the first function to the second; then, it pipes the result of this to the third function, and so on.

Which version is better? I would say that the version that refers to the `pipeTwo()` function is clearer: if you know how `reduce()` works, you can readily understand that our pipeline goes through the functions two at a time, starting from the first—and that matches what you know about how pipes work. The other versions that we wrote are more or less declarative, but not as simple to understand.

Before we look at other ways in which we can compose functions, let's consider how we would go about debugging our pipelines.

Debugging pipelines

Now, let's turn to a practical question: how do you debug your code? With pipelining, you can't really see what's passing on from function to function, so how do you do it? We have two answers for that: one (also) comes from the Unix/Linux world, and the other (the most appropriate for this book) uses wrappers to provide some logs.

Using tee

The first solution we'll use implies adding a function to the pipeline, which will just log its input. We want to implement something similar to the `tee` Linux command, which can intercept the standard data flow in a pipeline and send a copy to an alternate file or device. Remembering that `/dev/tty` is the usual console, we could execute something similar to the following and get an onscreen copy of everything that passes through the `tee` command:

```
$ ls -l | grep "odt$" | tee /dev/tty | wc -l

...the list of files with names ending in odt...
4
```

We could write a similar function with ease:

```
const tee = arg => {
  console.log(arg);
  return arg;
};
```



If you are aware of the uses of the comma operator, you can be more concise and just write `const tee = (arg) => (console.log(arg), arg)`—do you see why? Check out https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/Comma_Operator for the answer!

Our logging function is short and to the point: it will receive a single argument, list it, and pass it on to the next function in the pipe. We can see it working in the following code:

```
console.log(
  pipeline2(getDir, tee, filterOdt, tee, count)(
    "/home/fkerekki/Documents"
  )
);

[...the list of all the files in the directory...]
[...the list of files with names ending in odt...]
4
```

We could do even better if our `tee()` function could receive a logger function as a parameter, as we did in the *Logging in a functional way* section of Chapter 6, *Producing Functions – Higher-Order Functions*; it's just a matter of making the same kind of change we managed there. The same good design concepts are applied again!

```
const tee2 = (arg, logger = console.log) => {
  logger(arg);
  return args;
};
```



Be aware that there might be a binding problem when passing `console.log` in that way. It would be safer to write `console.log.bind(console)` just as a precaution.

This function works exactly in the same way as the previous `tee()`, though it will allow us to be flexible when it comes to applying and testing. However, in our case, this would just be a particular enhancement.

Now, let's consider an even more generic tapping function, with more possibilities than just doing a bit of logging.

Tapping into a flow

If you wish, you could write an enhanced `tee()` function that could produce more debugging information, possibly send the reported data to a file or remote service, and so on—there are many possibilities you can explore. You could also explore a more general solution, of which `tee()` would just be a particular case and which would also allow us to create personalized tapping functions. This can be seen in the following diagram:

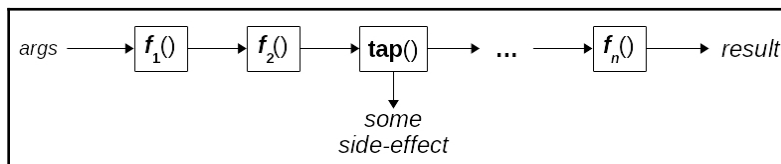


Figure 8.2: Tapping allows you to apply a function so that you can inspect data as it flows through the pipeline

When working with pipelines, you might want to put a logging function in the middle of it, or you might want some other kind of *snooping* function—possibly for storing data somewhere, calling a service, or some other kind of side effect. We could have a generic `tap()` function, which would allow us to inspect data as it moves along our pipeline, that would behave in the following way:

```
const tap = curry((fn, x) => (fn(x), x));
```

This is probably a candidate for the *trickiest-looking-code-in-the-book* award, so let's explain it. We want to produce a function that, given a function, `fn()`, and an argument, `x`, will evaluate `fn(x)` (to produce whatever sort of side effect we may be interested in) but return `x` (so the pipeline goes on without interference). The comma operator has exactly that behavior: if you write something similar to `(a, b, c)`, JavaScript will evaluate the three expressions in order and use the last value as the expression's value.



The comma has several uses in JavaScript and you can read more about its usage as an operator at https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/Comma_Operator.

Now, we can take advantage of currying to produce several different tapping functions. The one we wrote in the previous section, `tee()`, could also be written in the following fashion:

```
const tee3 = tap(console.log);
```

By the way, you could have also written `tap()` without currying, but you'll have to admit it loses some of its mystery! This is demonstrated here:

```
const tap2 = fn => x => (fn(x), x);
```

This does exactly the same job, and you'll recognize this way of currying as what we looked at in the *Currying by hand* section of Chapter 7, *Transforming Functions – Currying and Partial Application*. Now that we have learned how to tap into a pipeline, let's move on to a different way of doing logging by revisiting some concepts we looked at in previous chapters.

Using a logging wrapper

The second idea we mentioned is based on the `addLogging()` function that we wrote in the *Logging* section of Chapter 6, *Producing Functions – Higher-Order Functions*. The idea was to wrap a function with some logging functionality so that, on entry, the arguments would be printed and, on exit, the result of the function would be shown:

```
pipeline2(
  addLogging(getDir),
  addLogging(filterOdt),
  addLogging(count))("/home/fkerekki/Documents");
```

```
entering getDir: /home/fkerekki/Documents
exiting getDir: ...the list of all the files in the directory...
entering filterOdt: ...the same list of files...
exiting filterOdt: ...the list of files with names ending in odt...
entering count: ...the list of files with names ending in odt...
exiting count: 4
```

We can trivially verify that the `pipeline()` function is doing its thing correctly—whatever a function produces, as a result, is given as input to the next function in the line and we can also understand what's happening with each call. Of course, you don't need to add logging to *every* function in the pipeline: you would probably do so in the places where you suspected an error was occurring.

Now that we've looked at how to join functions, let's take a look at a very common way of defining functions in FP, *pointfree style*, which you may encounter.

Pointfree style

When you join functions together, either in pipeline fashion or with composition, as we'll see later in this chapter, you don't need any intermediate variables to hold the results that will become arguments to the next function in line: they are implicit. Similarly, you can write functions without mentioning their parameters; this is called the pointfree style.



Pointfree style is also called *tacit* programming and *pointless* programming by detractors! The term *point* itself means a function parameter, while pointfree refers to not naming those parameters.

Defining pointfree functions

You can easily recognize a pointfree function definition because it doesn't need the `function` keyword or the `=>` symbol. Let's revisit some of the previous functions we wrote in this chapter and check them out. For example, the definition of our original file-counting functions is as follows:

```
const countOdtFiles3 = path =>
  pipeTwo(pipeTwo(getDir, filterOdt), count)(path);

const countOdtFiles4 = path =>
  pipeTwo(getDir, pipeTwo(filterOdt, count))(path);
```

The preceding code could be rewritten as follows:

```
const countOdtFiles3b = pipeTwo(pipeTwo(getDir, filterOdt), count);

const countOdtFiles4b = pipeTwo(getDir, pipeTwo(filterOdt, count));
```

The new definitions don't make reference to the parameter for the newly defined functions. You can deduce this by examining the first function in the pipeline (`getDir()`, in this case) and seeing what it receives as arguments. (Using type signatures, as we'll see in Chapter 12, *Building Better Containers – Functional Data Types*, would be of great help in terms of documentation.) Similarly, the definition for `getLat()` is pointfree:

```
const getLat = curry(getField)("lat");
```

What should be the equivalent full style definition? You'd have to examine the `getField()` function (we looked at this in the *Revisiting an example* section) to decide that it expects an object as an argument. However, making that need explicit by writing the following wouldn't make much sense:

```
const getLat = obj => curry(getField)("lat")(obj);
```

If you were willing to write all this, you may wish to stick with the following:

```
const getLat = obj => obj.lat;
```

Then, you could simply not care about currying!

Converting to pointfree style

On the other hand, you had better pause for a minute and try not to write *everything* in pointfree code, whatever it might cost. For example, consider the `isNegativeBalance()` function we wrote back in Chapter 6, *Producing Functions – Higher-Order Functions*:

```
const isNegativeBalance = v => v.balance < 0;
```

Can we write this in a pointfree style? Yes, we can, and we'll see how—but I'm not sure we'd want to code this way! We can consider building a pipeline of two functions: one will extract the balance from the given object, while the other will check whether it's negative. Due to this, we will write our alternative version of the balance-checking function like so:

```
const isNegativeBalance2 = pipeline(getBalance, isNegative);
```

To extract the balance attribute from a given object, we can use `getField()` and a bit of currying, and then write the following:

```
const getBalance = curry(getField)("balance");
```

For the second function, we could write the following code:

```
const isNegative = x => x < 0;
```

There goes our pointfree goal! Instead, we can use the `binaryOp()` function, also from the same chapter we mentioned earlier, plus some more currying, to write the following:

```
const isNegative = curry(binaryOp(">"))(0);
```

I wrote the test the other way around ($0 > x$ instead of $x < 0$) just for ease of coding. An alternative would have been to use the enhanced functions I mentioned in the *A handier implementation* section of Chapter 6, *Producing Functions – Higher-Order Functions*, which is a bit less complex, as follows:

```
const isNegative = binaryOpRight("<", 0);
```

So, finally, we could write the following:

```
const isNegativeBalance2 = pipeline(
  curry(getField)("balance"),
  curry(binaryOp(">"))(0)
);
```

Alternatively, we could write the following:

```
const isNegativeBalance3 = pipeline(
  curry(getField)("balance"),
  binaryOpRight("<", 0)
);
```

Do you really think that's an improvement? Our new versions of `isNegativeBalance()` don't make a reference to their argument and are fully pointfree, but the idea of using pointfree style should be to help improve the clarity and readability of your code, and not to produce obfuscation and opaqueness! I doubt anybody would look at our new versions of the function and consider them to be an advantage over the original, for any possible reason.

If you find that your code is becoming harder to understand, and that's only due to your intent on using pointfree programming, stop and roll back your changes. Remember our doctrine for this book: we want to do FP, but we don't want to go overboard with it—and using the pointfree style is not a requirement!

In this section, we've learned how to build pipelines of functions—this is a powerful technique. For objects and arrays, however, we have another special technique that you may have used already: chaining. Let's take a look at this now.

Chaining and fluent interfaces

When you work with objects or arrays, there is another way of linking the execution of several calls together: by applying *chaining*. For example, when you work with arrays, if you apply a `map()` or `filter()` method, the result is a new array, which you can then apply a new further `map()` or `filter()` to, and so forth. We used such methods when we defined the `range()` function back in the *Working with ranges* section of Chapter 5, *Programming Declaratively – A Better Style*:

```
const range = (start, stop) =>
  new Array(stop - start).fill(0).map((v, i) => start + i);
```

First, we created a new array; then, we applied the `fill()` method to it, which updated the array in place (side effect) and returned the updated array, to which we finally applied a `map()` method. The latter method generated a new array, to which we could have applied further mappings, filtering, or any other available method.

Let's take a look at a common example of fluent, chained APIs, and then consider how we can do this on our own.

An example of fluent APIs

This style of continuous chained operation is also used in fluent APIs or interfaces. To give just one example, the graphic `D3.js` library (see <https://d3js.org/> for more on it) frequently uses this style. The following example, taken from <https://bl.ocks.org/mbostock/4063269>, shows it in action:

```
var node = svg
  .selectAll(".node")
  .data(pack(root).leaves())
  .enter()
  .append("g")
  .attr("class", "node")
  .attr("transform", function(d) {
    return "translate(" + d.x + "," + d.y + ")";
  });
```

Each method works on the previous object and provides access to a new object that future method calls will be applied to (such as the `selectAll()` or `append()` methods) or updates the current one (like the `attr()` attribute setting calls do). This style is not unique and several other well-known libraries (jQuery comes to mind) also apply it.

Can we automate this? In this case, the answer is *possibly, but I'd rather not*. In my opinion, using `pipeline()` or `compose()` works just as well, and manages the same result. With object chaining, you are limited to returning new objects or arrays or something that methods can be applied to. (Remember, if you are working with standard types, such as strings or numbers, you can't add methods to them unless you mess with their prototype, which isn't recommended!). With composition, however, you can return any kind of value; the only restriction is that the next function in line must be expecting the data type that you are providing.

On the other hand, if you are writing your own API, then you can provide a fluent interface by just having each method return this—unless, of course, it needs to return something else! If you were working with someone else's API, you could also do some trickery by using a proxy, but be aware there could be cases in which your proxied code might fail: maybe another proxy is being used, or there are some getters or setters that somehow cause problems, and so on.



You may want to read up on proxy objects at https://developer.mozilla.org/en/docs/Web/JavaScript/Reference/Global_Objects/Proxy—they are very powerful and allow for interesting metaprogramming functionalities, but they can also trap you with technicalities and will also cause an (albeit slight) slowdown in your proxied code.

Let's now take a look at how to chain calls so that we can apply them to any class.

Chaining method calls

Let's go for a basic example. We could have a `City` class with `name`, `latitude (lat)`, and `longitude (long)` attributes:

```
class City {
  constructor(name, lat, long) {
    this.name = name;
    this.lat = lat;
    this.long = long;
  }

  getName() {
    return this.name;
  }

  setName(newName) {
    this.name = newName;
  }
}
```

```
    }

    setLat(newLat) {
      this.lat = newLat;
    }

    setLong(newLong) {
      this.long = newLong;
    }

    getCoords() {
      return [this.lat, this.long];
    }
  }
}
```

This is a common class with a few methods; everything's quite normal. We could use this class as follows and provide details about my native city, Montevideo, Uruguay:

```
let myCity = new City("Montevideo, Uruguay", -34.9011, -56.1645);
console.log(myCity.getCoords(), myCity.getName());
// [ -34.9011, -56.1645 ] 'Montevideo, Uruguay'
```

If we wanted to allow the setters to be handled in a fluent manner, we could set up a proxy to detect such calls and provide the missing `return this`. How can we do that? If the original method doesn't return anything, JavaScript will include a `return undefined` statement by default so that we can detect whether that's what the method is returning and substitute `return this` instead. Of course, this is a problem: what would we do if we had a method that could legally return an `undefined` value on its own because of its semantics? We could have some kind of *exceptions list* to tell our proxy not to add anything in those cases, but let's not get into that.

The code for our handler is as follows. Whenever the method of an object is invoked, a `get` is implicitly called and we catch it. If we are getting a function, then we wrap it with some code of our own that will call the original method and then decide whether to return its value or a reference to the proxied object instead. If we weren't getting a function, then we would return the requested property's value. Our `chainify()` function will take care of assigning the handler to an object and creating the needed proxy:

```
const getHandler = {
  get(target, property, receiver) {
    if (typeof target[property] === "function") {
      // requesting a method? return a wrapped version
      return (...args) => {
        const result = target[property](...args);
        return result === undefined ? receiver : result;
      };
    }
  }
};
```

```
    } else {
      // an attribute was requested - just return it
      return target[property];
    }
  },
};
const chainify = obj => new Proxy(obj, getHandler);
```

We need to check whether the invoked `get()` was for a function or for an attribute. In the first case, we wrap the method with extra code so that it will execute it and then return its results (if any) or a reference to the object itself. In the second case, we just return the attribute, which is the expected behavior.

With this, we can *chainify* any object, so we'll get a chance to inspect any called methods. As I'm writing this, I'm currently living in Pune, India, so let's reflect that change:

```
myCity = chainify(myCity);

console.log(myCity
  .setName("Pune, India")
  .setLat(18.5626)
  .setLong(73.8087)
  .getCoords(),
  myCity.getName());

// [ 18.5626, 73.8087 ] 'Pune, India'
```

Notice the following:

- We changed `myCity` to be a proxified version of itself.
- We are calling several setters in a fluent fashion and they are working fine since our proxy is taking care of providing the value for the following call.
- The calls to `getCoords()` and `getName()` are intercepted, but nothing special is done because they already return a value.

Is working in a chained way worth it? That's up to you—but remember that there may be cases in which this approach fails, so be wary! Now, let's move on to composing, the other most common way of joining functions together.

Composing

Composing is quite similar to pipelining, but has its roots in mathematical theory. The concept of composition is simply—a sequence of function calls, in which the output of one function is the input for the next one—but the order is reversed from the one in pipelining. So, if you have a series of functions, from left to right, when pipelining, the first function of the series to be applied is the leftmost one, but when you use composition, you start with the rightmost one.

Let's investigate this a bit more. When you define the composition of, say, three functions as $(f \circ g \circ h)$, and apply this composition to x , this is equivalent to writing $f(g(h(x)))$. It's important to note that, as with pipelining, the arity of the first function to be applied (actually the last one in the list) can be anything, but all the other functions must be unary. Also, apart from the difference as to the sequence of function evaluations, composing is an important tool in FP because it also abstracts implementation details (putting your focus on what you need to accomplish, rather than on the specific details for achieving that), thereby letting you work in a more declarative fashion.



If it helps, you can read $(f \circ g \circ h)$ as *f after g after h*, so that it becomes clear that h is the first function to be applied, while f is the last.

Given its similarity to pipelining, it will be no surprise that implementing composition won't be very hard. However, there will still be some important and interesting details to go over. Let's take a look at some examples of composition before moving on to using higher-order functions and finishing with some considerations about testing composed functions.

Some examples of composition

It may not be a surprise to you, but we have already seen several examples of composition—or, at the very least, cases in which the solutions we achieved were functionally equivalent to using composition. Let's review some of these and work with some new examples too.

Unary operators

In the *Logically negating a function* section of Chapter 6, *Producing Functions – Higher-Order Functions*, we wrote a `not()` function that, given another function, would logically invert its result. We used that function to negate a check for negative balances; the sample code for this could be as follows:

```
const not = fn => (...args) => !fn(...args);

const positiveBalance = not(isNegativeBalance);
```

In another section of that very same chapter, *Turning operations into functions*, I left you with the challenge of writing a `unaryOp()` function that would provide unary functions equivalent to common JavaScript operators. If you met that challenge, you should be able to write something like the following:

```
const logicalNot = unaryOp("!");
```

Assuming the existence of a `compose()` function, you could have also written the following:

```
const positiveBalance = compose(logicalNot, isNegativeBalance);
```

Which one do you prefer? It's a matter of taste, really—but I think the second version makes it clearer what we are trying to do. With the `not()` function, you have to check what it does in order to understand the general code. With composition, you still need to know what `logicalNot()` is, but the global construct is open to see.

To look at just one more example in the same vein, you could have managed to get the same results that we got in the *Inverting results* section, in the same chapter. Recall that we had a function that could compare strings according to the Spanish language rules, but we wanted to invert the sense of the comparison so that it was sorted in descending order:

```
const changeSign = unaryOp("-");

palabras.sort(compose(changeSign, spanishComparison));
```

This code produces the same result that our previous sorting problem did, but the logic is expressed more clearly and with less code: a typical FP result! Let's look at some more examples of composing functions by reviewing another task we went over earlier.

Counting files

We can also go back to our pipeline. We had written a single-line function to count the `odt` files in a given path:

```
const countOdtFiles2 = path => count(filterOdt(getDir(path)));
```

Disregarding (at least for the moment) the observation that this code is not as clear as the pipeline version that we got to develop later, we could have also written this function with composition:

```
const countOdtFiles2b = path => compose(count, filterOdt, getDir)(path);
```

```
countOdtFiles2b("/home/fkerekki/Documents"); // 4, no change here
```



We could have also written the function in pointfree fashion, without specifying the `path` parameter, with `const countOdtFiles2 = compose(count, filterOdt, getDir)`, but I wanted to parallel the previous definition.

It would also be possible to see this written in *one-liner* fashion:

```
compose(count, filterOdt, getDir)("/home/fkerekki/Documents");
```

Even if it's not as clear as the pipeline version (and that's just my opinion, which may be biased by my liking of Linux!), this declarative implementation makes it clear that we depend on combining three distinct functions to get our result—this is easy to see and applies the idea of building large solutions out of simpler pieces of code.

Let's take a look at another example that's designed to compose as many functions as possible.

Finding unique words

Finally, let's go for another example, which, I agree, could have also been used for pipelining. Suppose you have some text and want to extract all the unique words from it: how would you go about doing this? If you think about it in steps (instead of trying to create a full solution in a single bit step), you would probably come up with a solution similar to this:

1. Ignore all non-alphabetic characters
2. Put everything in uppercase

3. Split the text into words
4. Create a set of words



Why a set? Because it automatically discards repeated values; check out https://developer.mozilla.org/en/docs/Web/JavaScript/Reference/Global_Objects/Set for more on this. By the way, we will be using the `Array.from()` method to produce an array out of our set; see https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Array/from for more information.

Now, using FP, let's solve each problem:

```
const removeNonAlpha = str => str.replace(/[^a-z]/gi, " ");
const toUpperCase = demethodize(String.prototype.toUpperCase);
const splitInWords = str => str.trim().split(/\s+/);
const arrayToSet = arr => new Set(arr);
const setToList = set => Array.from(set).sort();
```

With these functions, the result can be written as follows:

```
const getUniqueWords = compose(
  setToList,
  arrayToSet,
  splitInWords,
  toUpperCase,
  removeNonAlpha
);
```

Since you don't get to see the arguments of any of the composed functions, you really don't need to show the parameter for `getUniqueWords()` either, so the pointfree style is natural to use in this case.

Now, let's test our function. To do this, let's apply this function to the first two sentences of Abraham Lincoln's address at Gettysburg (which we already used in an example back in the *Mapping and flattening – flatMap* section of Chapter 5, *Programming Declaratively – A Better Style*) and print out the 43 different words (trust me, I counted them!) they comprised:

```
const GETTYSBURG_1_2 = `Four score and seven years ago
our fathers brought forth on this continent, a new nation,
conceived in liberty, and dedicated to the proposition
that all men are created equal. Now we are engaged in a
great civil war, testing whether that nation, or any
nation so conceived and so dedicated, can long
endure.`;
```

```
console.log(getUniqueWords(GETTYSBURG_1_2));  
[ 'A',  
  'AGO',  
  'ALL',  
  'AND',  
  'ANY',  
  'ARE',  
  'BROUGHT',  
  'CAN',  
  'CIVIL',  
  .  
  .  
  'TESTING', |  
  'THAT',  
  'THE',  
  'THIS',  
  'TO',  
  'WAR',  
  'WE',  
  'WHETHER',  
  'YEARS' ]
```

Of course, you could have written `getUniqueWords()` in a shorter way, but the point I'm making is that by composing your solution out of several shorter steps, your code is clearer and easier to grasp. However, if you wish to say that a pipelined solution seems better, then it's just a matter of opinion!

At this point, we have looked at many examples of function composition, but there's another way to manage this—by using higher-order functions.

Composing with higher-order functions

It's pretty obvious that composing by hand could easily be done in a similar fashion to pipelining. For example, the unique word counting function that we wrote previously could be written in simple JavaScript style:

```
const getUniqueWords1 = str => {  
  const str1 = removeNonAlpha(str);  
  const str2 = toUpperCase(str1);  
  const arr1 = splitInWords(str2);  
  const set1 = arrayToSet(arr1);  
  const arr2 = setToList(set1);  
  return arr2;  
};
```

Alternatively, it could be written more concisely (and more obscurely!) in *one-liner* style:

```
const getUniqueWords2 = str =>
  setToList (arrayToSet (splitInWords (toUpperCase (removeNonAlpha (str)))));

console.log(getUniqueWords2(GETTYSBURG_1_2));
// [ 'A', 'AGO', 'ALL', 'AND', ... 'WAR', 'WE', 'WHETHER', 'YEARS' ]
```

This works fine, but like we did when we studied pipelining, let's go and look for a more general solution that won't require writing a special function each time we want to compose some other functions.

Composing two functions is quite easy and requires making a small change with regard to our `pipeTwo()` function, which we looked at earlier in this chapter:

```
const pipeTwo = (f, g) => (...args) => g(f(...args));
const composeTwo = (f, g) => (...args) => f(g(...args));
```

The only difference is that, with piping, you apply the leftmost function first, while with composing, you start with the rightmost function first. This variation suggests that we could have used the `flipTwo()` higher-order function from the *Parameters order* section of Chapter 7, *Transforming Functions – Currying and Partial Application*. Is it clearer? Here is the code:

```
const composeTwoByFlipping = flipTwo(pipeTwo);
```

In any case, if we wanted to compose more than two functions, we could have also taken advantage of the associative property in order to write something like the following:

```
const getUniqueWords3 = composeTwo(
  setToList,
  composeTwo(
    arrayToSet,
    composeTwo(splitInWords, composeTwo(toUpperCase, removeNonAlpha))
  )
);

console.log(getUniqueWords3(GETTYSBURG_1_2));
// [ 'A', 'AGO', 'ALL', 'AND', ... 'WAR', 'WE', 'WHETHER', 'YEARS' ] OK
again
```

Even though this works, let's go for a better solution—we can provide at least two. The first way has to do with the fact that pipelining and composing work *in reverse* of each other. We apply functions from left to right when pipelining, and from right to left when composing. Thus, we can achieve the same result that we achieved with a composition by reversing the order of the functions and doing pipelining instead; a very functional solution, which I really like! This is as follows:

```
const compose = (...fns) => pipeline(...(fns.reverse()));

console.log(
  compose(
    setToList,
    arrayToSet,
    splitInWords,
    toUpperCase,
    removeNonAlpha
  )(GETTYSBURG_1_2)
);

/*
[ 'A', 'AGO', 'ALL', 'AND', ... 'WAR', 'WE', 'WHETHER', 'YEARS' ]
OK once more
*/
```

The only tricky part is the usage of the spread operator before calling `pipeline()`. After reversing the `fns` array, we must spread its elements in order to call `pipeline()` correctly.

The other solution, which is less declarative, is to use `reduceRight()` so that instead of reversing the list of functions, we reverse the order of processing them:

```
const compose2 = (...fns) => fns.reduceRight(pipeTwo);

console.log(
  compose2(
    setToList,
    arrayToSet,
    splitInWords,
    toUpperCase,
    removeNonAlpha
  )(GETTYSBURG_1_2)
);

/*
[ 'A', 'AGO', 'ALL', 'AND', ... 'WAR', 'WE', 'WHETHER', 'YEARS' ]
still OK
*/
```

Why/how does this work? Let's follow the inner workings of this call. To make this clearer, we can replace `pipeTwo()` with its definition:

```
const compose2b = (...fns) =>
  fns.reduceRight((f, g) => (...args) => g(f(...args)));
```

Let's take a closer look:

- Since no initial value is provided, `f()` is `removeNonAlpha()` and `g()` is `toUpperCase()`, so the first intermediate result is a function, `(...args) => toUpperCase(removeNonAlpha(...args))`; let's call it `step1()`.
- The second time, `f()` is `step1()` from the previous step, while `g()` is `splitInWords()`, so the new result is a function, `(...args) => splitInWords(step1(...args))`, which we can call `step2()`.
- The third time around, in the same fashion, we get `(...args) => arrayToSet(step2(...args))`, which we call `step3()`.
- Finally, the result is `(...args) => setToList(step3(...args))`, a function; let's call it `step4()`.

The final result turns out to be a function that receives `(...args)` and starts by applying `removeNonAlpha()` to it, then `toUpperCase()`, and so on, before finishing by applying `setToList()`.

It may come as a surprise that we can also make this work with `reduce()`—can you see why? The reasoning is similar to what we did previously, so we'll leave this as an exercise to you:

```
const compose3 = (...fns) => fns.reduce(composeTwo);
```



After working out how `compose3()` works, you might want to write a version of `pipeline()` that uses `reduceRight()`, just for symmetry, to round things out!

We will end this section by mentioning that, in terms of testing and debugging, we can apply the same ideas that we applied to pipelining; however, remember that composition *goes the other way!* We won't gain anything by providing yet more examples of the same kind, so let's consider a common way of chaining operations when using objects and see whether it's advantageous or not, given our growing FP knowledge and experience.

Testing composed functions

Let's finish this chapter by giving some consideration to testing for pipelined or composed functions. Given that the mechanism for both operations is similar, we will look at examples of both. They won't differ, other than their logical differences due to the left-to-right or right-to-left order of function evaluation.

When it comes to pipelining, we can start by looking at how to test the `pipeTwo()` function since the setup will be similar to `pipeline()`. We need to create some spies and check whether they were called the correct number of times and whether they received the correct arguments each time. We will set the spies so that they provide a known answer to a call. By doing this, we can check whether the output of a function becomes the input of the next in the pipeline:

```
var fn1, fn2;

describe("pipeTwo", function() {
  beforeEach(() => {
    fn1 = () => {};
    fn2 = () => {};
  });

  it("works with single arguments", () => {
    spyOn(window, "fn1").and.returnValue(1);
    spyOn(window, "fn2").and.returnValue(2);

    const pipe = pipeTwo(fn1, fn2);
    const result = pipe(22);

    expect(fn1).toHaveBeenCalledTimes(1);
    expect(fn2).toHaveBeenCalledTimes(1);
    expect(fn1).toHaveBeenCalledWith(22);
    expect(fn2).toHaveBeenCalledWith(1);
    expect(result).toBe(2);
  });

  it("works with multiple arguments", () => {
    spyOn(window, "fn1").and.returnValue(11);
    spyOn(window, "fn2").and.returnValue(22);

    const pipe = pipeTwo(fn1, fn2);
    const result = pipe(12, 4, 56);

    expect(fn1).toHaveBeenCalledTimes(1);
    expect(fn2).toHaveBeenCalledTimes(1);
    expect(fn1).toHaveBeenCalledWith(12, 4, 56);
  });
});
```

```
    expect(fn2).toHaveBeenCalledWith(11);
    expect(result).toBe(22);
  });
});
```

There isn't much to test, given that our function always receives two functions as parameters. The only difference between the tests is that one shows a pipeline that's been applied to a single argument, while the other shows it being applied to several arguments.

Moving on to `pipeline()`, the tests would be quite similar. However, we can add a test for a single-function pipeline (border case!) and another with four functions:

```
describe("pipeline", function() {
  beforeEach(() => {
    fn1 = () => {};
    fn2 = () => {};
    fn3 = () => {};
    fn4 = () => {};
  });

  it("works with a single function", () => {
    spyOn(window, "fn1").and.returnValue(11);

    const pipe = pipeline(fn1);
    const result = pipe(60);

    expect(fn1).toHaveBeenCalledTimes(1);
    expect(fn1).toHaveBeenCalledWith(60);
    expect(result).toBe(11);
  });

  // we omit here tests for 2 functions,
  // which are similar to those for pipeTwo()

  it("works with 4 functions, multiple arguments", () => {
    spyOn(window, "fn1").and.returnValue(111);
    spyOn(window, "fn2").and.returnValue(222);
    spyOn(window, "fn3").and.returnValue(333);
    spyOn(window, "fn4").and.returnValue(444);

    const pipe = pipeline(fn1, fn2, fn3, fn4);
    const result = pipe(24, 11, 63);

    expect(fn1).toHaveBeenCalledTimes(1);
    expect(fn2).toHaveBeenCalledTimes(1);
    expect(fn3).toHaveBeenCalledTimes(1);
    expect(fn4).toHaveBeenCalledTimes(1);
    expect(fn1).toHaveBeenCalledWith(24, 11, 63);
  });
});
```

```
    expect(fn2).toHaveBeenCalledWith(111);
    expect(fn3).toHaveBeenCalledWith(222);
    expect(fn4).toHaveBeenCalledWith(333);
    expect(result).toBe(444);
  });
});
```

Finally, for composition, the style is the same (except that the order of function evaluation is reversed), so let's take a look at a single test—here, I simply changed the order of the functions in the preceding test:

```
describe("compose", function() {
  beforeEach(() => {
    fn1 = () => {};
    fn2 = () => {};
    fn3 = () => {};
    fn4 = () => {};
  });

  // other tests omitted...

  it("works with 4 functions, multiple arguments", () => {
    spyOn(window, "fn1").and.returnValue(111);
    spyOn(window, "fn2").and.returnValue(222);
    spyOn(window, "fn3").and.returnValue(333);
    spyOn(window, "fn4").and.returnValue(444);

    const pipe = compose(fn4, fn3, fn2, fn1);
    const result = pipe(24, 11, 63);

    expect(fn1).toHaveBeenCalledTimes(1);
    expect(fn2).toHaveBeenCalledTimes(1);
    expect(fn3).toHaveBeenCalledTimes(1);
    expect(fn4).toHaveBeenCalledTimes(1);

    expect(fn1).toHaveBeenCalledWith(24, 11, 63);
    expect(fn2).toHaveBeenCalledWith(111);
    expect(fn3).toHaveBeenCalledWith(222);
    expect(fn4).toHaveBeenCalledWith(333);
    expect(result).toBe(444);
  });
});
```

Finally, to test the `chainify()` function, I opted to use the preceding `City` object I created—I didn't want to mess with mocks, stubs, spies, and the like; I wanted to ensure that the code worked under normal conditions:

```
class City {
  // as above
}

var myCity;

describe("chainify", function() {
  beforeEach(() => {
    myCity = new City("Montevideo, Uruguay", -34.9011, -56.1645);
    myCity = chainify(myCity);
  });

  it("doesn't affect get functions", () => {
    expect(myCity.getName()).toBe("Montevideo, Uruguay");
    expect(myCity.getCoords()[0]).toBe(-34.9011);
    expect(myCity.getCoords()[1]).toBe(-56.1645);
  });

  it("doesn't affect getting attributes", () => {
    expect(myCity.name).toBe("Montevideo, Uruguay");
    expect(myCity.lat).toBe(-34.9011);
    expect(myCity.long).toBe(-56.1645);
  });

  it("returns itself from setting functions", () => {
    expect(myCity.setName("Other name")).toBe(myCity);
    expect(myCity.setLat(11)).toBe(myCity);
    expect(myCity.setLong(22)).toBe(myCity);
  });

  it("allows chaining", () => {
    const newCoords = myCity
      .setName("Pune, India")
      .setLat(18.5626)
      .setLong(73.8087)
      .getCoords();

    expect(myCity.name).toBe("Pune, India");
    expect(newCoords[0]).toBe(18.5626);
    expect(newCoords[1]).toBe(73.8087);
  });
});
```

The final result of all of these tests can be seen in the following screenshot:

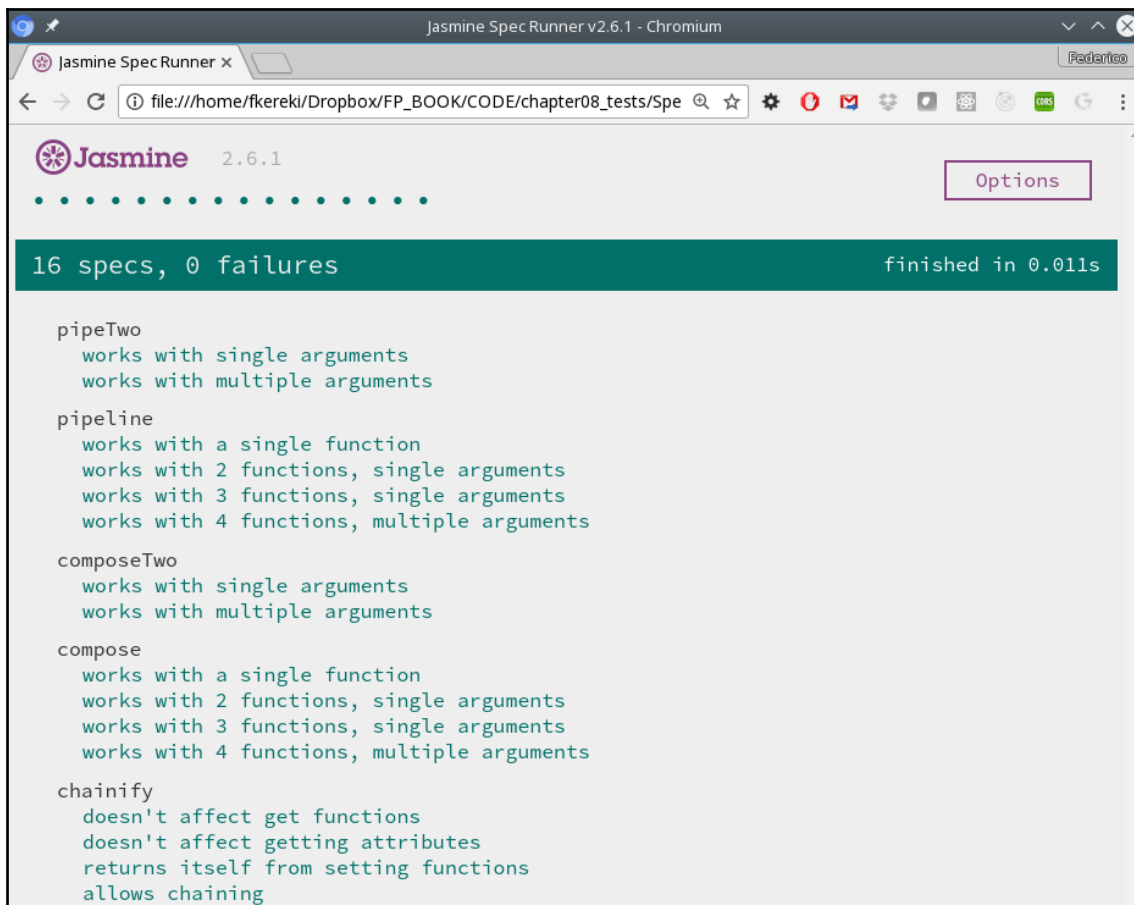


Figure 8.3: A successful run of testing for composed functions

As we can see, all our tests passed successfully; good!

Here, we have looked at the important methods we can use to build functions by using pipelining, chaining, and composition. This works very well, but we'll see that there's a particular case in which the performance of your code could be affected and that we'll need a new way to handle composition: *transducing*.

Transducing

Now, let's consider a performance problem in JavaScript that happens when we're dealing with large arrays and applying several map/filter/reduce operations. If you start with an array and apply such operations (via chaining, as we saw earlier in this chapter), you get the desired result, but many intermediate arrays are created, processed, and discarded—and that causes delays. If you are dealing with short arrays, the extra time won't make an impact, but if you are processing larger arrays (as in a big data process, maybe in Node, where you're working with the results of a large database query), then you will have cause to look for some optimization. We'll do this by learning about a new tool for composing functions: *transducing*.

First, let's create some functions and data. We'll make do with a basically nonsensical example since we aren't focusing on the actual operations but on the general process. We'll start with some filtering functions and some mappings:

```
const testOdd = x => x % 2 === 1;
const testUnderFifty = x => x < 50;
const duplicate = x => x + x;
const addThree = x => x + 3;
```

Now, let's apply those maps and filters to an array. First, we drop the even numbers, duplicate the kept odd numbers, drop results over 50, and end by adding three to all the results:

```
const myArray = [22, 9, 60, 24, 11, 63];

const a0 = myArray
  .filter(testOdd)
  .map(duplicate)
  .filter(testUnderFifty)
  .map(addThree);

/*
[ 21, 25 ]
*/
```

The following diagram shows how this sequence of operations works:

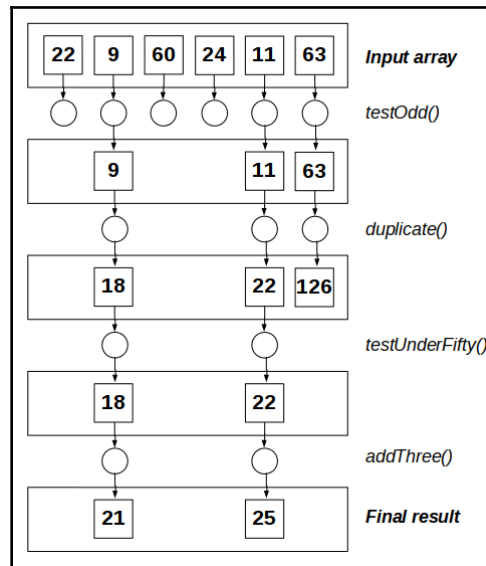


Figure 8.4: Chaining map/filter/reduce operations causes intermediate arrays to be created and later discarded

Here, we can see that chaining together several map/filter/reduce operations causes intermediate arrays (three, in this case) to be created and later discarded—and for large arrays, that can become cumbersome.

How can we optimize this? The problem here is that processing applies the first transformation to the input array; then, the second transformation is applied to the resulting array; then the third, and so on. The alternative solution would be to take the first element of the input array and apply all the transformations in sequence to it. Then, you would need to take the second element of the input array and apply all the transformations to it, then take the third, and so on. In a sort of pseudocode, the difference is between the following schemes:

```

for each transformation to be applied:
  for each element in the input list:
    apply the transformation to the element
  
```

With this logic, we go transformation by transformation, applying it to each list and generating a new one. This will require several intermediate lists to be produced. The alternative is as follows:

```
for each element in the input list:
  for each transformation to be applied:
    apply the transformation to the element
```

In this variant, we go element by element and apply all the transformations to it in sequence so that we arrive at the final output list without any intermediate ones.

Now, the problem is being able to transpose the transformations; how can we do this? We saw this key concept in Chapter 5, *Programming Declaratively – A Better Style*, and that we can define `map()` and `filter()` in terms of `reduce()`. By using those definitions, instead of a sequence of different functions, we will be applying the same operation (reduce) at each step, and here is the secret! As shown in the following diagram, we change the order of evaluation by composing all the transformations so that they can be applied in a single pass, with no intermediate arrays whatsoever:

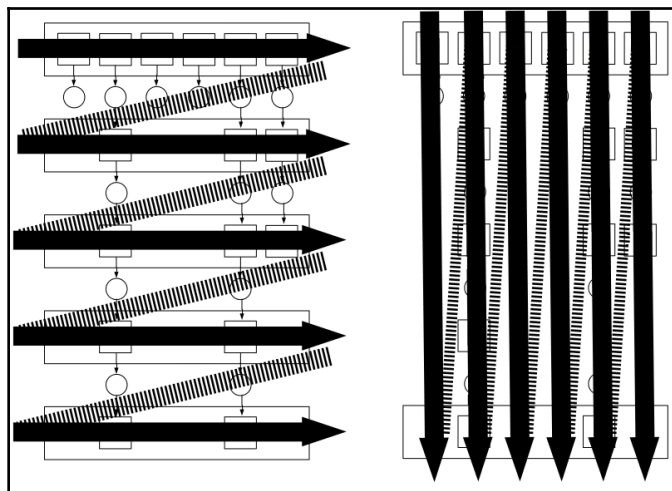


Figure 8.5: By applying transducers, we will change the order of evaluation but get the same result

Instead of applying a first reduce operation, passing its result to a second, its result to a third, and so on, we will compose all the reducing functions into a single one! Let's analyze this.

Composing reducers

Essentially, what we want is to transform each function (`testOdd()`, `duplicate()`, and so on) into a reducing operation that will call the following reducer. A couple of higher-order functions will help; one for mapping functions and an other for filtering ones. With this idea, the result of an operation will be passed to the next one, avoiding intermediate arrays:

```
const mapTR = fn => reducer => (accum, value) => reducer(accum, fn(value));

const filterTR = fn => reducer => (accum, value) =>
  fn(value) ? reducer(accum, value) : accum;
```

These two transforming functions are transducers: functions that accept a reducing function and return a new reducing function.



The word *transduce* comes from Latin, meaning *transform, transport, convert, change over*, and is applied in many different fields, including biology, psychology, machine learning, physics, electronics, and more.

How do we use these transducers? We can write code such as the following, though we'll want a more abstract, generic version later:

```
const testOddR = filterTR(testOdd);
const testUnderFiftyR = filterTR(testUnderFifty);
const duplicateR = mapTR(duplicate);
const addThreeR = mapTR(addThree);
```

Each of our original four functions is transformed, so they will calculate their result and call a reducer to deal with this further. As an example, `addThreeR()` will add three to its input and pass the incremented value to the next reducer, which in this case is `addToArray()`. This will build up the final resulting array. Now, we can write our whole transformation in a single step:

```
const addToArray = (a, v) => {
  a.push(v);
  return a;
};

const a1 = myArray.reduce(
```

```
    testOddR(duplicateR(testUnderFiftyR(addThreeR(addToArray))))),  
  []  
);  
  
/*  
 [ 21, 25 ]  
*/
```

This is quite a mouthful, but it works! However, we can simplify our code by using the `compose()` function:

```
const makeReducer1 = (arr, fns) =>  
  arr.reduce(compose(...fns)(addToArray), []);  
  
const a2 = makeReducer1(myArray, [  
  testOddR,  
  duplicateR,  
  testUnderFiftyR,  
  addThreeR,  
]);  
  
/*  
 [ 21, 25 ]  
*/
```

The code is the same, but pay particular attention to the `compose(...fns)(addToArray)` expression: we compose all the mapping and filtering functions—with the last one being `addToArray`—to build up the output. However, this is not as general as we may want it to be: why do we have to create an array? Why can't we have a different final reducing function? We can go one better by generalizing a bit more.

Generalizing for all reducers

To be able to work with all kinds of reducers and produce whatever kind of result they build, we'll need to make a small change. The idea is simple: let's modify our `makeReducer()` function so that it will accept a final reducer and a starting value for the accumulator:

```
const makeReducer2 = (arr, fns, reducer = addToArray, initial = []) =>  
  arr.reduce(compose(...fns)(reducer), initial);  
  
const a3 = makeReducer2(myArray, [  
  testOddR,  
  duplicateR,  
  testUnderFiftyR,
```

```
    addThreeR,  
  ] );  
  
  /*  
  [ 21, 25 ]  
  */
```

To make this function more usable, we specified our array-building function (and `[]` as a starting accumulator value) so that if you skip those two parameters, you'll get a reducer that produces an array. Now, let's look at the other option: instead of an array, let's calculate the sum of the resulting numbers after all the mapping and filtering:

```
const sum = makeReducer2(  
  myArray,  
  [testOddR, duplicateR, testUnderFiftyR, addThreeR],  
  (acc, value) => acc + value,  
  0  
);  
  
/*  
46  
*/
```

By using transducers, we have been able to optimize a sequence of map/filter/reduce operations so that the input array is processed once and directly produces the output result (whether that be an array or a single value) without creating any intermediate arrays; a good gain!

Summary

In this chapter, we have learned how to create new functions by joining several other functions in different ways through pipelining and composition. We also looked at fluent interfaces, which apply chaining, and transducing, a way to compose reducers in order to get higher speed sequences of transformations. With these methods, you'll be able to create new functions out of existing ones and keep programming in the declarative way we've been favoring.

In [Chapter 9, *Designing Functions – Recursion*](#), we will move on to function design and study the usage of recursion, which is a basic tool in functional programming and allows for very clean algorithm designs.

Questions

8.1. *Headline capitalization*: Let's define *headline-style capitalization*, so ensure that a sentence is all written in lowercase, except the first letter of each word. (The real definition of this style is more complicated, so let's simplify it for this question.) Write a `headline(sentence)` function that will receive a string as an argument and return an appropriately capitalized version. Spaces separate words. Build this function by composing smaller functions:

```
console.log(headline("Alice's ADVENTURES in WoNdErLaNd"));  
// Alice's Adventures In Wonderland
```

8.2. *Pending tasks*: A web service returns a result such as the following, showing, person by person, all their assigned tasks. Tasks may be finished (`done===true`) or pending (`done===false`). Your goal is to produce an array with the IDs of the pending tasks for a given person, identified by name, which should match the `responsible` field. Solve this by using composition or pipelining:

```
const allTasks = {  
  date: "2017-09-22",  
  byPerson: [  
    {  
      responsible: "EG",  
      tasks: [  
        {id: 111, desc: "task 111", done: false},  
        {id: 222, desc: "task 222", done: false}  
      ]  
    },  
    {  
      responsible: "FK",  
      tasks: [  
        {id: 555, desc: "task 555", done: false},  
        {id: 777, desc: "task 777", done: true},  
        {id: 999, desc: "task 999", done: false}  
      ]  
    },  
    {  
      responsible: "ST",  
      tasks: [{id: 444, desc: "task 444", done: true}]  
    }  
  ]  
};
```

Make sure your code doesn't throw an exception if, for example, the person you are looking for doesn't appear in the web service result!



In the last chapter of this book, Chapter 12, *Building Better Containers – Functional Data Types*, we will look at a different way of solving this by using `Maybe` monads. This greatly simplifies the problem of dealing with possibly missing data.

8.3. Thinking in abstract terms: Suppose you are looking through somewhat old code and you find a function that looks like the following one. (I'm keeping the names vague and abstract so that you can focus on the structure and not on the actual functionality). Can you transform this into pointfree style?

```
function getSomeResults(things) {  
    return sort(group(filter(select(things))));  
};
```

8.4. Undetected impurity? Did you notice that the `addToArray()` function we wrote is actually impure? (Check out the *Argument mutation* section of Chapter 4, *Behaving Properly – Pure Functions*, if you aren't convinced!) Would it be better if we wrote it as follows? Should we go for it?

```
const addToArray = (a, v) => [...a, v];
```

8.5. Needless transducing? We used transducers to simplify any sequence of mapping and filtering operations. Would you have needed this if you only had `map()` operations? What if you only had `filter()` operations?

9

Designing Functions - Recursion

In Chapter 8, *Connecting Functions – Pipelining and Composition*, we considered yet more ways to create new functions out of combining previous existing ones. Here, we are going to get into a different theme: how to actually design and write functions, in a typically functional way, by applying recursive techniques.

We will be covering the following topics:

- Understanding what recursion is and how to think in order to produce recursive solutions
- Applying recursion to some well-known problems, such as making a change or the Tower of Hanoi
- Using recursion instead of iteration to re-implement some higher-order functions from earlier chapters
- Writing search and backtrack algorithms with ease
- Traversing data structures, such as trees, to work with file system directories or with the browser DOM
- Getting around some limitations caused by browser JavaScript engine considerations

Using recursion

Recursion is a key technique in FP, to the degree that there are some languages that do not provide for any kind of iteration or loops and work exclusively with recursion (Haskell, which we already mentioned, is a prime example of that). A basic fact of computer science is that whatever you can do with recursion, you can also do with iteration (loops), and vice versa. The key concept is that there are many algorithms whose definition is far easier if you work recursively. On the other hand, recursion is not always taught, or many programmers, even knowing about it, prefer not to use it. Therefore, in this section, we shall see several examples of recursive thinking, so that you can adapt it for your functional coding.

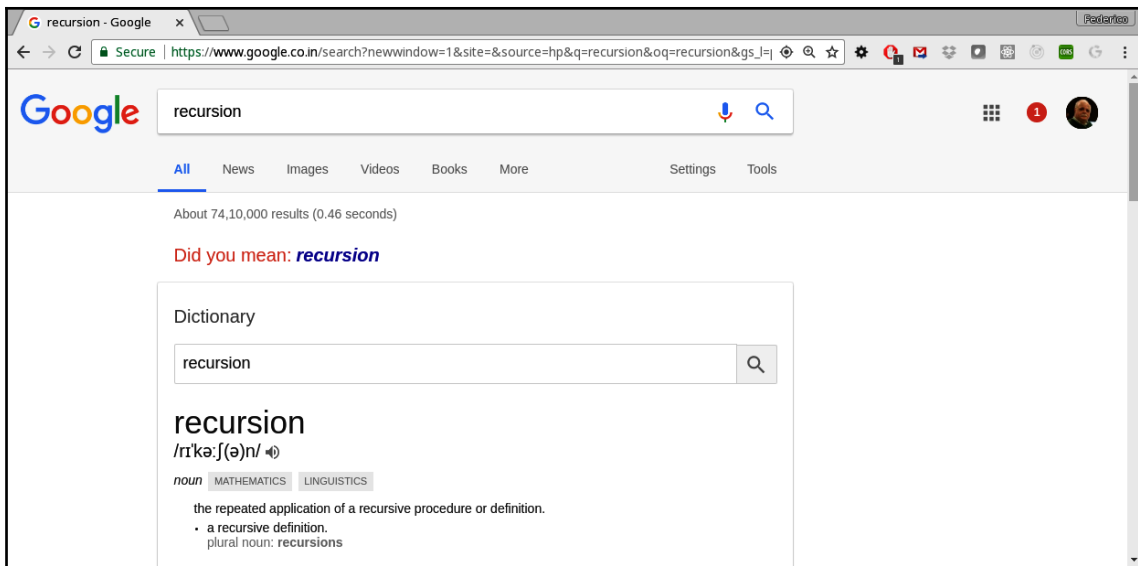


A typical, oft-quoted, and very old computer joke!

Dictionary definition:
recursion: (n) see recursion

But what is recursion? There are many ways to define what recursion is, but the simplest one I've seen runs along the lines of *a function calls itself again and again, until it doesn't*. Recursion is a natural technique for several kinds of problems, such as the following:

- Mathematical definitions, such as the Fibonacci sequence or the factorial of a number.
- Data-structure-related algorithms, with recursively defined structures, such as **lists** (a list is either empty or consists of a head node followed by a list of nodes) or **trees** (a tree might be defined as a special node, called the root, linked to zero or more trees).
- Syntax analysis for compilers, based on grammar rules, which themselves depend on other rules, which also depend on other rules, and so on.
- And many more! It even appears in art and humor, as shown in the following screenshot:



Google itself jokes about it: if you ask about recursion, it answers "Did you mean: recursion!"

In any case, a recursive function, apart from some easy, base cases, in which no further computation is required, always needs to call itself one or more times in order to perform part of the required calculations. This concept may be not very clear at the moment, so, in the following sections, we will see how we can think in a recursive fashion and then solve several common problems by applying this technique.

Thinking recursively

The key to solving problems recursively is assuming that you already have a function that does whatever you need and just calling it normally. (Doesn't this sound weird? Actually, it is quite appropriate: if you want to solve a problem by using recursion, you must first have solved it before...) On the other hand, if you try to work out in your head how the recursive calls work and attempt to follow the flow in your mind, you'll probably just get lost. So what you need to do is the following:

1. Assume you already have an appropriate function to solve your problem.
2. See how the big problem can be solved by solving one (or more) smaller problems.
3. Solve those problems by using the imagined function from step 1.
4. Decide what your base cases are. Make sure that they are simple enough that they are solved directly, without requiring any more calls.

With these points in mind, you can solve problems by recursion because you'll have the basic structure for your recursive solution.

There are three usual methods for solving problems by applying recursion:

- **Decrease and conquer** is the simplest case, in which solving a problem directly depends on solving a single, simpler case of itself.
- **Divide and conquer** is a more general approach. The idea is to try to divide your problem into two or more smaller versions, solve them recursively, and use these solutions to solve the original problem. The only difference between this technique and decrease and conquer is that, here, you have to solve two or more other problems, instead of only one.
- **Dynamic programming** can be seen as a variant of divide and conquer: basically, you solve a complex problem by breaking it into a set of somewhat simpler versions of the same problem and solving each in order; however, a key idea in this strategy is to store previously found solutions, so that whenever you find yourself needing the solution to a simpler case again you won't directly apply recursion, but rather use the stored result and avoid unnecessary repeated calculations.

In this section, we shall look at a few problems and solve them by thinking in a recursive way. Of course, we shall see more applications of recursion in the rest of the chapter; here, we'll focus on the key decisions and questions that are needed to create such an algorithm.

Decrease and conquer – searching

The most usual case of recursion involves just a single, simple case. We have already seen some examples of this, such as the ubiquitous factorial calculation: to calculate the factorial of n , you previously needed to calculate the factorial of $n-1$. (See Chapter 1, *Becoming Functional – Several Questions*.) Let's turn now to a nonmathematical example.

To search for an element in an array, you would also use this decrease and conquer strategy. If the array is empty, then obviously the searched-for value isn't there; otherwise, the result is in the array if and only if it's the first element in it, or if it's in the rest of the array. The following code does just that:

```
const search = (arr, key) => {
  if (arr.length === 0) {
    return false;

  } else if (arr[0] === key) {
    return true;
  }
}
```

```
    } else {  
      return search(arr.slice(1), key);  
    }  
  };
```

This implementation directly mirrors our explanation, and it's easy to verify its correctness.

By the way, just as a precaution, let's look at two further implementations of the same concept. You can shorten the search function a bit—is it still clear? We are using a ternary operator to detect whether the array is empty, and a Boolean `||` operator to return `true` if the first element is the sought one or else return the result of the recursive search:

```
const search2 = (arr, key) =>  
  arr.length === 0 ? false : arr[0] === key || search2(arr.slice(1), key);
```

Sparseness can go even further! Using `&&` as a shortcut is a common idiom:

```
const search3 = (arr, key) =>  
  arr.length && (arr[0] === key || search3(arr.slice(1), key));
```

I'm not really suggesting that you code the function in this way—rather, consider it a warning against the tendency that some FP developers have to try to go for the tightest, shortest possible solution and never mind clarity!

Decrease and conquer – doing powers

Another classic example has to do with calculating the powers of numbers in an efficient way. If you want to calculate, say, 2 to the 13th power (2^{13}), then you can do this with 12 multiplications; however, you can do much better by writing 2^{13} as the following:

```
= 2 times  $2^{12}$   
= 2 times  $4^6$   
= 2 times  $16^3$   
= 2 times 16 times  $16^2$   
= 2 times 16 times  $256^1$   
= 8192
```

This reduction in the total number of multiplications may not look very impressive, but, in terms of algorithmic complexity, it allows us to bring down the order of the calculations from $O(n)$ to $O(\lg n)$. In some cryptographic-related methods, which have to raise numbers to really high exponents, this makes a very important difference. We can implement this recursive algorithm in a few lines of code, as shown in the following code:

```
const powerN = (base, power) => {
  if (power === 0) {
    return 1;

  } else if (power % 2) { // odd power?
    return base * powerN(base, power - 1);

  } else { // even power?
    return powerN(base * base, power / 2);
  }
};
```



When implemented for production, bit operations are used, instead of modulus and divisions. Checking whether a number is odd can be written as `power & 1`, and division by 2 is achieved with `power >> 1`. These alternative calculations are way faster than the replaced operations.

Calculating a power is simple when the base case is reached (raising something to the zeroth power), or is based upon previously calculating a power for a smaller exponent. (If you wanted to, you could add another base case for raising something to the power of one.) These observations show that we are seeing a textbook case for the decrease and conquer recursive strategy.

Finally, some of our higher-order functions, such as `map()`, `reduce()`, and `filter()`, also apply this technique; we'll look into this later on in this chapter.

Divide and conquer – the Towers of Hanoi

With the divide and conquer strategy, solving a problem requires two or more recursive solutions. For starters, let's consider a classic puzzle, invented by a French mathematician, Édouard Lucas, in the nineteenth century. The puzzle involves a temple in India, with 3 posts a 64 golden disks of decreasing diameter. The priests have to move the disks from the first post to the last one following two rules: only one disk can be moved at a time, and a larger disk can never be placed on top of a smaller disk. According to the legend, when the 64 disks are moved, the world will end. This puzzle is usually marketed under the name *Towers of Hanoi* (yes, they changed countries!) with fewer than 10 disks. See *Figure 9.1*:

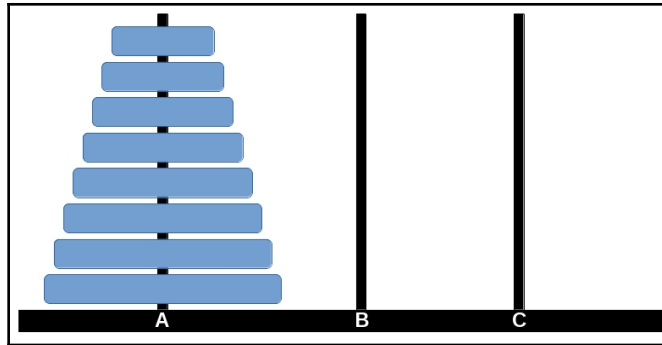


Figure 9.1: The classic Towers of Hanoi puzzle has a simple recursive solution.



The solution for n disks requires $2^n - 1$ movements. The original puzzle, requiring $2^{64} - 1$ movements, at one movement per second, would take more than 584 billion years to finish, a very long time, considering that the universe's age is evaluated to only be 13.8 billion years!

Suppose that we already had a function that was able to solve the problem of moving any number of disks from a source post to a destination post using the remaining post as an extra aid. Think about solving the general problem if you already had a function to solve that problem: `hanoi(disks, from, to, extra)`. If you wanted to move several disks from one post to another, then you could solve it easily using this (still unwritten!) function by carrying out the following steps:

1. Moving all of the disks but the last one to the extra post.
2. Moving the last disk to the destination post.
3. Moving all the disks from the extra post (where you had placed them earlier) to the destination.

But what about our base cases? We could decide that, to move a single disk, you needn't use the function; you just go ahead and move it. When coded, it becomes the following:

```
const hanoi = (disks, from, to, extra) => {
  if (disks === 1) {
    console.log(`Move disk 1 from post ${from} to post ${to}`);
  } else {
    hanoi(disks - 1, from, extra, to);
    console.log(`Move disk ${disks} from post ${from} to post ${to}`);
    hanoi(disks - 1, extra, to, from);
  }
};
```

We can quickly verify that this code works:

```
hanoi (4, "A", "B", "C"); // we want to move all disks from A to B
Move disk 1 from post A to post C
Move disk 2 from post A to post B
Move disk 1 from post C to post B
Move disk 3 from post A to post C
Move disk 1 from post B to post A
Move disk 2 from post B to post C
Move disk 1 from post A to post C
Move disk 4 from post A to post B
Move disk 1 from post C to post B
Move disk 2 from post C to post A
Move disk 1 from post B to post A
Move disk 3 from post C to post B
Move disk 1 from post A to post C
Move disk 2 from post A to post B
Move disk 1 from post C to post B
```

There's only a small detail to consider, which can simplify the function even further. In this code, our base case (the one that needs no further recursion) is when `disks` equals one. You could also solve this in a different way by letting the disks go down to zero and simply not doing anything—after all, moving zero disks from one post to another is achieved by doing nothing at all! The revised code would be as follows:

```
const hanoi2 = (disks, from, to, extra) => {
  if (disks > 0) {
    hanoi(disks - 1, from, extra, to);
    console.log(`Move disk ${disks} from post ${from} to post ${to}`);
    hanoi(disks - 1, extra, to, from);
  }
};
```

Instead of checking whether there are any disks to move before doing the recursive call, we can just skip the check and have the function test, at the next level, whether there's something to be done.



If you are doing the puzzle by hand, there's a simple solution for that: on odd turns, always move the smaller disk to the next post (if the total number of disks is odd) or to the previous post (if the total number of disks is even). On even turns, do the only possible move that doesn't imply the smaller disk.

So, the principle for recursive algorithm design works: assume you already have your desired function and use it to build it!

Divide and conquer – sorting

We can see another example of the divide and conquer strategy with sorting. A way to sort arrays, called *quicksort*, is based upon the following steps:

1. If your array has 0 or 1 elements, do nothing; it's already sorted (this is the base case).
2. Pick an element of the array (called the **pivot**) and split the rest of the array into two subarrays: the elements smaller than your chosen element and the elements greater than or equal to your chosen element.
3. Recursively sort each subarray.
4. Concatenate both sorted results, with the pivot in-between, to produce the sorted version of the original array.

Let's see a simple version of this (there are some better-optimized implementations, but we are interested in the recursive logic now). Usually, picking a random element of the array is suggested to avoid some bad performance border cases, but for our example, let's just take the first one:

```
const quicksort = arr => {
  if (arr.length < 2) {
    return arr;

  } else {
    const pivot = arr[0];
    const smaller = arr.slice(1).filter(x => x < pivot);
    const greaterEqual = arr.slice(1).filter(x => x >= pivot);
    return [...quicksort(smaller), pivot, ...quicksort(greaterEqual)];
  }
};

console.log(quicksort([22, 9, 60, 12, 4, 56]));
// [4, 9, 12, 22, 56, 60]
```

We can see how this works in *Figure 9.2*: the pivot for each array and subarray is underlined. Splitting is shown with dotted arrows and is joined with full lines:

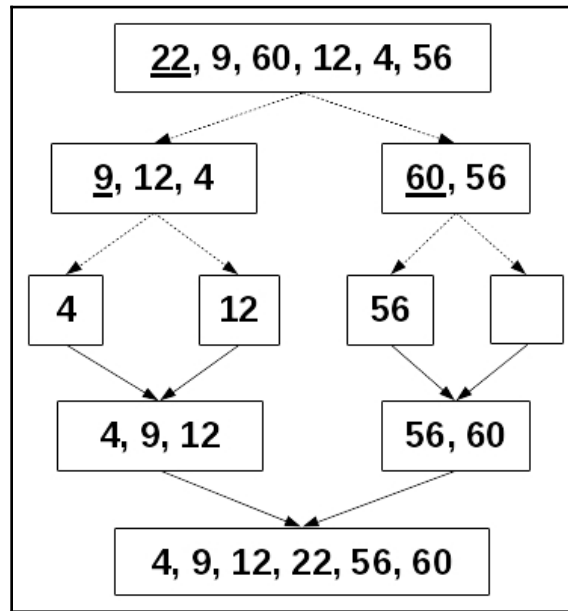


Figure 9.2: Quicksort sorts an array recursively, applying the divide and conquer strategy, to reduce the original problem to smaller ones



Writing Quicksort correctly is not trivial; see question 9.8 at the end of this chapter for an alternative version that happens to be *almost* right, but not totally correct!

We have already seen the basic strategies to reduce a problem to simpler versions of itself. Let's now look at an important optimization that is key for many algorithms.

Dynamic programming – making change

The third general strategy, dynamic programming, assumes that you will have to solve many smaller problems, but, instead of using recursion each and every time, it depends on you having stored the previously found solutions... memoization, in other words! In *Chapter 4, Behaving Properly – Pure Functions*, and later in a better fashion in *Chapter 6, Producing Functions – Higher-Order Functions*, we already saw how to optimize the calculations of the usual Fibonacci series, avoiding unnecessary repeated calls. Let's now consider another problem.

Given a certain number of dollars and the list of existing bill values, calculate in how many different ways we can pay that amount of dollars with different combinations of bills. It is assumed that you have access to an unlimited number of each bill. How can we go about solving this? Let's start by considering the base cases, where no further computation is needed. They are as follows:

- Paying negative values is not possible, so in such cases, we should return zero
- Paying zero dollars is only possible in a single way (by giving no bills), so in this case, we should return 1
- Paying any positive amount of dollars isn't possible if no bills are provided, so in this case, also return 0

Finally, we can answer the question: in how many ways can we pay N dollars with a given set of bills? We can consider two cases: we do not use the larger bill at all and pay the amount using only smaller denomination bills, or we can take one bill of the larger amount and reconsider the question. (Let's forget the avoidance of repeated calculations for now):

- In the first case, we should invoke our supposedly existing function with the same value of N , but prune the largest bill denomination from the list of available bills.
- In the second case, we should invoke our function with N minus the largest bill denomination, keeping the list of bills the same, as shown in the following code:

```
const makeChange = (n, bills) => {
  if (n < 0) {
    return 0; // no way of paying negative amounts

  } else if (n == 0) {
    return 1; // one single way of paying $0: with no bills

  } else if (bills.length == 0) {
    // here, n>0
    return 0; // no bills? no way of paying

  } else {
    return makeChange(n, bills.slice(1)) + makeChange(n - bills[0],
bills);
  }
};

console.log(makeChange(64, [100, 50, 20, 10, 5, 2, 1]));
// 969 ways of paying $64
```


Now let's do some optimizing. This algorithm often needs to recalculate the same values over and over. (To verify this, add `console.log(n, bills.length)` as the first line in `makeChange()`—but be ready for plenty of output!) However, we already have a solution for this: memoization! Since we are applying this technique to a binary function, we'll need a version of the memoization algorithm that deals with more than one parameter. The memoizing function and its application would be as follows:

```
const memoize3 = fn => {
  let cache = {};
  return (...args) => {
    let strX = JSON.stringify(args);
    return strX in cache ? cache[strX] : (cache[strX] = fn(...args));
  };
};

const makeChange = memoize3((n, bills) => {
  // ...same as above
});
```

The memoized version of `makeChange()` is far more efficient, and you can verify it with logging. While it is certainly possible to deal with the repetitions by yourself (for example, by keeping an array of already computed values), the memoization solution is, in my opinion, better, because it composes two functions to produce a better solution for the given problem.

Higher-order functions revisited

Classic FP techniques do not use iteration at all, but work exclusively with recursion as the only way to do some looping. Let's revisit some of the functions that we have already seen in Chapter 5, *Programming Declaratively – A Better Style*, such as `map()`, `reduce()`, `find()`, and `filter()`, to see how we can make do with just recursion.

We are not planning to exchange the basic JavaScript functions for ours, though: it's highly likely that performance will be worse for our recursive polyfills and we won't derive any advantages just from having the functions use recursion. Rather, we want to study how iterations are performed in a recursive way so that our efforts are more pedagogical than practical, OK?

Mapping and filtering

Mapping and filtering are quite similar insofar as both imply going through all the elements in an array and applying a callback to each to produce output. Let's first work out the mapping logic, which will have several points to solve, and then we should see that filtering has become almost trivially easy, requiring just small changes.

For mapping, given how we are developing recursive functions, we need a base case. Fortunately, that's easy: mapping an empty array just produces a new empty array. Mapping a nonempty array can be done by first applying the mapping function to the first element of the array, then recursively mapping the rest of the array, and finally producing a single array accumulating both results.

Based on this idea, we can work out a simple initial version: let's call it `mapR()`, just to remember that we are dealing with our own, recursive version of `map()`; however, be careful: our polyfill has some bugs! We'll deal with them one at a time. Here's our first attempt at writing our own mapping code:

```
const mapR = (arr, cb) =>
  arr.length === 0 ? [] : [cb(arr[0])] .concat (mapR(arr.slice(1), cb));
```

Let's test it out:

```
let aaa = [ 1, 2, 4, 5, 7];
const timesTen = x => x * 10;

console.log(aaa.map(timesTen)); // [10, 20, 40, 50, 70]
console.log(mapR(aaa, timesTen)); // [10, 20, 40, 50, 70]
```

Great! Our `mapR()` function seemingly produces the same results as `map()`. However, shouldn't our callback function receive a couple more parameters, specifically the index at the array and the original array itself?



Check out the definition for the callback function for `map()` at https://developer.mozilla.org/en/docs/Web/JavaScript/Reference/Global_Objects/Array/map.

Our implementation isn't quite ready yet. Let's first see how it fails by using a simple example:

```
const timesTenPlusI = (v, i) => 10 * v + i;

console.log(aaa.map(timesTenPlusI)); // [10, 21, 42, 53, 74]
console.log(mapR2(aaa, timesTenPlusI)); // [NaN, NaN, NaN, NaN, NaN]
```

Generating the appropriate index position will require an extra parameter for the recursion, but it is basically simple: when we start out, we have `index=0`, and when we call our function recursively, it's starting at position `index+1`. Accessing the original array requires yet another parameter, which will never change, and now we have a better mapping function:

```
const mapR2 = (arr, cb, i = 0, orig = arr) =>
  arr.length == 0
    ? []
    : [cb(arr[0], i, orig)].concat(
      mapR2(arr.slice(1), cb, i + 1, orig)
    );

let aaa = [1, 2, 4, 5, 7];
const senseless = (x, i, a) => x * 10 + i + a[i] / 10;
console.log(aaa.map(senseless)); // [10.1, 21.2, 42.4, 53.5, 74.7]
console.log(mapR2(aaa, senseless)); // [10.1, 21.2, 42.4, 53.5, 74.7]
```

Great! When you do recursion instead of iteration, you don't have access to an index, so, if you need it (as in our case), you'll have to generate it on your own. This is an often-used technique, so working out our `map()` substitute was a good idea.

However, having extra arguments in the function is not so good; a developer might accidentally provide them and then the results would be unpredictable. So, using another usual technique, let's define an inner function, `mapLoop()`, to handle looping. This is, in fact, the usual way in which looping is achieved when you only use recursion; look at the following code, in which the extra function just isn't accessible from outside:

```
const mapR3 = (orig, cb) => {
  const mapLoop = (arr, i) =>
    arr.length == 0
      ? []
      : [cb(arr[0], i, orig)].concat(
        mapR3(arr.slice(1), cb, i + 1, orig)
      );
  return mapLoop(orig, 0);
};
```

There's only one pending issue: if the original array has some missing elements, they should be skipped during the loop. Let's look at an example:

```
[1, 2, , , 5].map(tenTimes)
// [10, 20, undefined x 2, 50]
```

Fortunately, fixing this is simple—and be glad that all the experience gained here will help us write the other functions in this section! Can you understand the fix in the following code?

```
const mapR4 = (orig, cb) => {
  const mapLoop = (arr, i) => {
    if (arr.length == 0) {
      return [];

    } else {
      const mapRest = mapR4(arr.slice(1), cb, i + 1, orig);
      if (!(0 in arr)) {
        return [,].concat(mapRest);

      } else {
        return [cb(arr[0], i, orig)].concat(mapRest);
      }
    }
  };

  return mapLoop(orig, 0);
};

console.log(mapR4(aaa, timesTen)); // [10, 20, undefined x 2, 50]
```

Wow! This was more than we bargained for, but we saw several techniques: how to replace iteration with recursion, how to accumulate a result across iterations, and how to generate and provide the index value—good tips! Furthermore, writing filtering code will prove much easier, since we'll be able to apply very much the same logic as we did for mapping. The main difference is that we use the callback function to decide whether an element goes into the output array, so the inner loop function is a tad longer:

```
const filterR = (orig, cb) => {
  const filterLoop = (arr, i) => {
    if (arr.length == 0) {
      return [];

    } else {
      const filterRest = filterR(arr.slice(1), cb, i + 1, orig);
      if (!(0 in arr)) {
        return filterRest;

      } else if (cb(arr[0], i, orig)) {
        return [arr[0]].concat(filterRest);

      } else {
        return filterRest;
      }
    }
  };
};
```

```
    }
  }
};

return filterLoop(orig, 0);
};

let aaa = [1, 12, , , 5, 22, 9, 60];
const isOdd = x => x % 2;
console.log(aaa.filter(isOdd)); // [1, 5, 9]
console.log(filterR(aaa, isOdd)); // [1, 5, 9]
```

Okay, we managed to implement two of our basic higher-order functions with pretty similar recursive functions. What about the others?

Other higher-order functions

Programming `reduce()` is, from the outset, a bit trickier, since you can decide to omit the initial value for the accumulator. Since we mentioned earlier that providing that value is generally better, let's work here under the assumption that it will be given; dealing with the other possibility won't be too hard.

The base case is simple: if the array is empty, the result is the accumulator; otherwise, we must apply the reduce function to the current element and the accumulator, update the latter, and then continue working with the rest of the array. This can be a bit confusing because of the ternary operators, but, after all we've seen, it should be clear enough. Look at the following code for the details:

```
const reduceR = (orig, cb, accum) => {
  const reduceLoop = (arr, i) => {
    return arr.length == 0
      ? accum
      : reduceR(
          arr.slice(1),
          cb,
          !(0 in arr) ? accum : cb(accum, arr[0], i, orig),
          i + 1,
          orig
        );
  };

  return reduceLoop(orig, 0);
};
```

```
let bbb = [1, 2, , 5, 7, 8, 10, 21, 40];
console.log(bbb.reduce((x, y) => x + y, 0)); // 94
console.log(reduce2(bbb, (x, y) => x + y, 0)); // 94
```

On the other hand, `find()` is particularly apt for recursive logic, since the very definition of how you (attempt to) find something, is recursive in itself:

- You look at the first place you think of, and, if you find what you were seeking, you are done.
- Alternatively, you look at the other places to see if what you seek is there.

We are only missing the base case, but that's simple, and we already saw this earlier in the chapter: if you have no places left to search, then you know you won't be successful in your search:

```
const findR = (arr, cb) => {
  if (arr.length === 0) {
    return undefined;

  } else {
    return cb(arr[0]) ? arr[0] : findR(arr.slice(1), cb);
  }
};
```

If you want to shorten the code a bit, you can do this by using the ternary operator a couple of times:

```
const findR2 = (arr, cb) =>
  arr.length === 0
  ? undefined
  : cb(arr[0])
  ? arr[0]
  : findR(arr.slice(1), cb);
```

We can quickly verify whether this works:

```
let aaa = [1, 12, , , 5, 22, 9, 60];

const isTwentySomething = x => 20 <= x && x <= 29;
console.log(findR(aaa, isTwentySomething)); // 22

const isThirtySomething = x => 30 <= x && x <= 39;
console.log(findR(aaa, isThirtySomething)); // undefined
```

Let's finish with our pipelining function. The definition of a pipeline lends itself to quick implementation:

- If we want to pipeline a single function, then that's the result of the pipeline.
- If we want to pipeline several functions, then we must first apply the initial function, and then pass that result as input to the pipeline of the other functions.

We can directly turn this into code:

```
const pipelineR = (first, ...rest) =>
  rest.length == 0
  ? first
  : (...args) => pipelineR(...rest)(first(...args));
```

We can verify its correctness with a simple example. Let's pipeline several calls to a couple of functions, one of which just adds one to its argument and the other of which multiplies by ten:

```
const plus1 = x => x + 1;
const by10 = x => x * 10;

pipelineR(
  by10,
  plus1,
  plus1,
  plus1,
  by10,
  plus1,
  by10,
  by10,
  plus1,
  plus1,
  plus1
)(2);
// 23103
```

If you follow the math, you'll be able to check that the pipelining is working fine. Doing the same for composition is easy, except that you cannot use the spread operator to simplify the function definition, and you'll have to work with array indices—work it out!

Searching and backtracking

Searching for solutions to problems, especially when there is no direct algorithm and you must resort to trial and error, is particularly appropriate for recursion. Many of these algorithms fall into a scheme such as the following:

- Out of many choices available, pick one.
- If no options are available, you've failed.
- If you could pick one, apply the same algorithm, but find a solution to the rest.
- If you succeed, you are done.
- Otherwise, try another choice.

With small variations, you can also apply similar logic to find a good—or possibly, optimum—solution to a given problem. Each time you find a possible solution, you match it with previous ones that you might have found and decide which to keep. This may go on until all possible solutions have been evaluated, or until a good enough solution has been found.

There are many problems to which this logic applies. They are as follows:

- Finding a way out of mazes—pick any path, mark it as already followed, and try to find a way out of the maze that won't reuse that path: if you succeed, you are done, and if you do not, go back to pick a different path.
- Filling out sudoku puzzles—if an empty cell can contain only a single number, then assign it; otherwise, run through all of the possible assignments and, for each one, recursively try to see if the rest of the puzzle can be filled out.
- Playing chess—where you aren't likely to be able to follow through all possible move sequences and so instead you opt for the best-estimated position.

Let's apply these techniques to two problems: solving the eight queens puzzle and traversing a complete file directory.

The eight queens puzzle

The eight queens puzzle was invented in the nineteenth century and involves placing eight chess queens on a standard chessboard. The special condition is that no queen should be able to attack another—implying that no pair of queens may share a row, column, or diagonal line. The puzzle may ask for any solution or for the total number of distinct solutions, which is what we will attempt to find.



The puzzle may also be generalized to n queens, by working on an $n \times n$ square board. It is known that there are solutions for all values of n , except $n=2$ (pretty simple to see why: after placing one queen, all of the board is threatened) and $n=3$ (if you place a queen in the center, all of the board is threatened, and if you place a queen on a side, only two squares are not threatened, but they threaten each other, making it impossible to place queens on them).

Let's start our solution with the top-level logic. Because of the given rules, there will be a single queen in each column, so we use a `places()` array to take note of each queen's row within the given column. The `SIZE` constant could be modified to solve a more general problem. We'll count each found distribution of queens in the `solutions` variable. Finally, the `finder()` function will perform the recursive search for solutions. The basic skeleton for the code would be as follows:

```
const SIZE = 8;
let places = Array(SIZE);
let solutions = 0;

finder();

console.log(`Solutions found: ${solutions}`);
```

Let's get into the required logic. When we want to place a queen in a given row within a certain column, we must check whether any of the previously placed queens were already placed on the same row or in a diagonal leading from the row. See *Figure 9.3*:

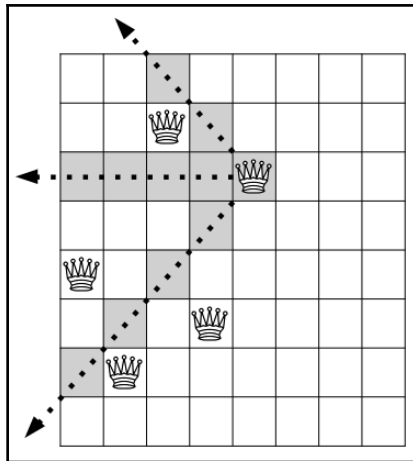


Figure 9.3: Before placing a queen in a column, we must check the previously placed queens' positions

Let's write a `checkPlace(column, row)` function to verify whether a queen can be safely placed in the given square. The most straightforward way is by using `.every()`, as shown in the following code:

```
const checkPlace = (column, row) =>
  places
    .slice(0, column)
    .every((v, i) => v !== row && Math.abs(v - row) !== column - i);
```

This declarative fashion seems best: when we place a queen in a position, we want to make sure that every other previously placed queen is in a different row and diagonal. A recursive solution would have been possible too, so let's see that. How do we know that a square is safe?

- A base case is that, when there are no more columns to check, the square is safe.
- If the square is in the same row or diagonal as any other queen, it's not safe.
- If we have checked a column and found no problem, we can now recursively check the following one.

The required code to check whether a position in a column can be occupied by a queen is therefore as follows:

```
const checkPlace2 = (column, row) => {
  const checkColumn = i => {
    if (i === column) {
      return true;
    } else if (
      places[i] === row ||
      Math.abs(places[i] - row) === column - i
    ) {
      return false;
    } else {
      return checkColumn(i + 1);
    }
  };

  return checkColumn(0);
};
```

The code works, but I wouldn't be using it since the declarative version is clearer. Anyway, having worked out this check, we can pay attention to the main `finder()` logic, which will do the recursive search. The process proceeds as we described at the beginning: trying out a possible placement for a queen, and if that is acceptable, using the same search procedure to try and place the remaining queens. We start at column 0, and our base case is when we reach the last column, meaning that all queens have been successfully placed: we can print out the solution, count it, and go back to search for a new configuration.



Check out how we use `map()` and a simple arrow function to print the rows of the queens, column by column, as numbers between 1 and 8, instead of 0 and 7. In chess, rows are numbered from 1 to 8 (and columns from a to h, but that doesn't matter here).

Check out the following code, which applies the logic that we described previously:

```
const finder = (column = 0) => {
  if (column === SIZE) {
    // all columns tried out?
    console.log(places.map(x => x + 1)); // print out solution
    solutions++; // count it
  } else {
    const testRowsInColumn = j => {
      if (j < SIZE) {
        if (checkPlace(column, j)) {
          places[column] = j;
          finder(column + 1);
        }
        testRowsInColumn(j + 1);
      }
    };

    testRowsInColumn(0);
  }
};
```

The inner `testRowsInColumn()` function also fulfills an iterative role, but recursively. The idea is to attempt placing a queen in every possible row, starting at zero: if the square is safe, `finder()` is called to start searching from the next column onward. No matter whether a solution was or wasn't found, all rows in the column are tried out, since we are interested in the total number of solutions; in other search problems, you might be content with finding just any solution, and you would stop your search right there.

We have come this far, so let's find the answer to our problem!

```
[1, 5, 8, 6, 3, 7, 2, 4]
[1, 6, 8, 3, 7, 4, 2, 5]
[1, 7, 4, 6, 8, 2, 5, 3]
[1, 7, 5, 8, 2, 4, 6, 3]
...
... 84 lines snipped out ...
...
[8, 2, 4, 1, 7, 5, 3, 6]
[8, 2, 5, 3, 1, 7, 4, 6]
[8, 3, 1, 6, 2, 5, 7, 4]
[8, 4, 1, 3, 6, 2, 7, 5]
Solutions found: 92
```

Each solution is given as the row positions for the queens, column by column, and there are 92 solutions in all.

Traversing a tree structure

Data structures, which include recursion in their definition, are naturally appropriate for recursive techniques. Let's consider, for example, how to traverse a complete filesystem directory, listing all of its contents. Where's the recursion? The answer is clear if you consider that each directory can do either of the following:

- Be empty—a base case, in which there's nothing to do
- Include one or more entries, each of which is either a file or a directory itself

Let's work out a full recursive directory listing—meaning that when we encounter a directory, we also list its contents, and if those include more directories, we also list them, and so on. We'll be using the same node functions as in `getDir()` (from the *Building pipelines by hand* section in Chapter 8, *Connecting Functions – Pipelining and Composition*), plus a few more in order to test whether a directory entry is a symbolic link (which we won't follow to avoid possible infinite loops), a directory (which will require a recursive listing), or a common file:

```
const fs = require("fs");

const recursiveDir = path => {
  console.log(path);
  fs.readdirSync(path).forEach(entry => {
    if (entry.startsWith(".")) {
      // skip it!
    } else {
```

```
const full = path + "/" + entry;
const stats = fs.lstatSync(full);

if (stats.isSymbolicLink()) {
  console.log("L ", full); // symlink, don't follow
} else if (stats.isDirectory()) {
  console.log("D ", full);
  recursiveDir(full);
} else {
  console.log(" ", full);
}
}
});
};
```

The listing is long but correct. I opted to list the `/boot` directory in my own OpenSUSE Linux laptop, and this was produced:

```
recursiveDir("/boot");
/boot
  /boot/System.map-4.11.8-1-default
  /boot/boot.readme
  /boot/config-4.11.8-1-default
D  /boot/efi
D  /boot/efi/EFI
D  /boot/efi/EFI/boot
  /boot/efi/EFI/boot/bootx64.efi
  /boot/efi/EFI/boot/fallback.efi
  ...
  ... many omitted lines
  ...
L  /boot/initrd
  /boot/initrd-4.11.8-1-default
  /boot/message
  /boot/symtypes-4.11.8-1-default.gz
  /boot/symvers-4.11.8-1-default.gz
  /boot/sysctl.conf-4.11.8-1-default
  /boot/vmlinuz-4.11.8-1-default.gz
L  /boot/vmlinuz
  /boot/vmlinuz-4.11.8-1-default
```

By the way, we can apply the same structure to a similar problem: traversing a DOM structure. We could list all of the tags, starting from a given element, by using essentially the same approach: we list a node and (by applying the same algorithm) all of its children. The base case is also the same as before: when a node has no children, no more recursive calls are done. You can see this in the following code:

```
const traverseDom = (node, depth = 0) => {
  console.log(`${"| ".repeat(depth)}<${node.nodeName.toLowerCase()}>`);
  for (let i = 0; i < node.children.length; i++) {
    traverseDom(node.children[i], depth + 1);
  }
};
```

We are using the `depth` variable to know how many levels below the original element we are. We could also use it to make the traversing logic stop at a certain level, of course; in our case, we are using it only to add some bars and spaces to appropriately indent each element according to its place in the DOM hierarchy. The result of this function is shown in the following code. It would be easy to list more information and not just the element tag, but I wanted to focus on the recursive process:

```
traverseDom(document.body);
<body>
| <script>
| <div>
| | <div>
| | | <a>
| | | <div>
| | | | <ul>
| | | | | <li>
| | | | | | <a>
| | | | | | <div>
| | | | | | | <div>
| | | | | | | <div>
| | | | | | | | <br>
| | | | | | | <div>
| | | | | | | <ul>
| | | | | | | | <li>
| | | | | | | | <a>
| | | | | | | | <li>
...etc!
```

However, there's an ugly point there: why are we doing a loop to go through all of the children? We should know better! The problem is that the structure we get from the DOM isn't really an array; however, there's a way out: we can use `Array.from()` to create a real array out of it and then write a more declarative solution. The following code solves the problem in a better way:

```
const traverseDom2 = (node, depth = 0) => {
  console.log(` ${" | ".repeat(depth)}<${node.nodeName.toLowerCase()}>`);
  Array.from(node.children).forEach(child =>
    traverseDom2(child, depth + 1)
  );
};
```

Writing `[...node.children].forEach()` would have worked as well, but I think using `Array.from()` makes it clearer to any reader that we are trying to make an array out of something that looks like one, but really isn't.

We have now seen many ideas about the usage of recursion, and we've seen many applications of it; however, there are some cases in which you may run into problems, so let's now consider some tweaks that may come in handy for specific problems.

Recursion techniques

While recursion is a very good technique, it may face some problems because of the details the way it is actually implemented. Each function call, recursive or not, requires an entry in the internal JavaScript stack. When you are working with recursion, each recursive call itself counts as another call, and you might find that there are some situations in which your code will crash and throw an error because it ran out of memory, just because of multiple calls. On the other hand, with most current JavaScript engines, you can probably have several thousand pending recursive calls without a problem (but with earlier browsers and smaller machines, the number could drop into the hundreds and could feasibly go even lower), so it could be argued that at present, you are not likely to suffer from any particular memory problems.

In any case, let's review the problem and go over some possible solutions in the following sections, because even if you don't get to actually apply them, they represent valid FP ideas for which you may find a place in yet other problems. We will be looking at the following solutions:

- **Tail call optimization**, a technique that speeds up recursion
- **Continuation passing style**, an important FP technique that can help with recursion
- A couple of interestingly named techniques, **trampolines** and **thunks**, which are also common FP tools
- **Recursion elimination**, a technique that is rather beyond the scope of this book, but which still may be applied.

Tail call optimization

When is a recursive call not a recursive call? Put this way, the question may make little sense, but there's a common optimization—for other languages, alas, but not JavaScript!—that explains the answer. If the recursive call is the very last thing a function will do, then the call could be transformed to a simple jump to the start of the function without needing to create a new stack entry. (Why? The stack entry wouldn't be required: after the recursive call is done, the function would have nothing else to do, so there is no need to further save any of the elements that have been pushed into the stack upon entering the function.) The original stack entry would then no longer be needed and could simply be replaced by a new one, corresponding to the recent call.



The fact that a recursive call, a quintessential FP technique, is being implemented by a base imperative `GO TO` statement can be considered an ultimate irony!

These calls are known as **tail calls** (for obvious reasons) and have higher efficiency, not only because of the saved stack space, but also because a jump is quite a bit faster than any alternative. If the browser implements this enhancement, then it is using a **tail call optimization (TCO)**; however, a glance at the compatibility tables at <http://kangax.github.io/compat-table/es6/> shows that at the time of writing (at the end of 2019), the only browser that provides TCO is Safari.

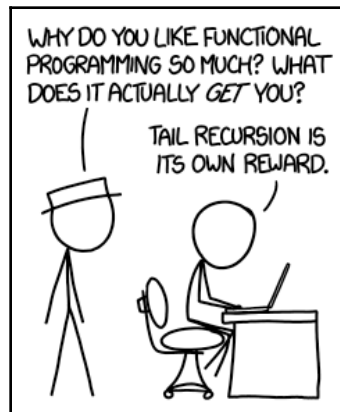


Figure 9.4: To understand this joke, you must have previously understood it!



This XKCD comic is available online at <https://xkcd.com/1270/>

There's a simple (though nonstandard) test that lets you verify whether your browser provides TCO (I found this snippet of code in several places on the web, but I'm sorry to say I cannot attest to the original author (although I believe it is Csaba Hellinger, from Hungary)). Calling `detectTCO()` lets you know whether your browser does or does not use TCO:

```
"use strict";

function detectTCO() {
  const outerStackLen = new Error().stack.length;
  return (function inner() {
    const innerStackLen = new Error().stack.length;
    return innerStackLen <= outerStackLen;
  })();
}
```

The `Error().stack` result is not a JavaScript standard, but modern browsers support it, albeit in somewhat different ways. In any case, the idea is that, when a function with a long name calls another function with a shorter name, the stack trace should do the following:

- It should get shorter if the browser implements TCO, since the old entry for the longer-named function would be replaced with the entry for the shorter-named one.
- It should get longer without TCO, since a completely new stack entry would be created without doing away with the original one.

I'm using Chrome on my Linux laptop, and I added a `console.log()` statement to show `Error().stack`. You can see that both stack entries (for `inner()` and `detectTCO()`) are *live*, so there's no TCO:

```
Error
  at inner (<anonymous>:6:13)
  at detectTCO (<anonymous>:9:6)
  at <anonymous>:1:1
```

Of course, there's also another way of learning whether your environment includes TCO: just try out the following function, which does nothing, with large enough numbers. If you manage to run it with numbers like, say, 100,000 or 1,000,000, you can be fairly sure that your JavaScript engine is doing TCO! A possible such function could be the following:

```
function justLoop(n) {
  n && justLoop(n - 1); // until n is zero
}
```

Let's finish this section with a very short quiz to be sure that we understand what tail calls are. Is the recursive call in the factorial function that we saw in Chapter 1, *Becoming Functional – Several Questions*, a tail call?

```
function fact(n) {
  if (n === 0) {
    return 1;
  } else {
    return n * fact(n - 1);
  }
}
```

Think about it, because the answer is important! You might be tempted to answer in the affirmative, but the correct answer is a *no*. There's good reason for this, and it's a key point: after the recursive call is done, and the value for `fact(n-1)` has been calculated, the function *still* has work to do. (So doing the recursive call wasn't actually the last thing the function would do.) You can see it more clearly if you write the function in this equivalent way:

```
function fact2(n) {
  if (n === 0) {
    return 1;

  } else {
    const aux = fact2(n - 1);
    return n * aux;
  }
}
```

So there should be two takeaways from this section: TCO isn't usually offered by browsers, and, even if it were, you cannot take advantage of it if your calls aren't actual tail calls. Now that we know what the problem is, let's see some FP ways of working around it!

Continuation passing style

If we have recursive calls stacked too high, we already know that our logic will fail. On the other hand, we know that tail calls should alleviate that problem, but don't, because of browser implementations! However, there's a way out of this. Let's first consider how we can transform recursive calls into tail calls by using a well-known FP concept—**continuations**—and we'll leave the problem of solving TCO limitations for the next section. (We mentioned continuations in the *Callbacks, promises, and continuations* section of [Chapter 3, Starting Out with Functions – A Core Concept](#), but we didn't go into detail.)

In FP parlance, a continuation is something that represents the state of a process and allows processing to continue. This may be too abstract, so let's get down to earth for our needs. The key idea is that, when you call a function, you also provide it with a continuation (in reality, a simple function) that will be called at return time.

Let's look at a trivial example. Suppose you have a function that returns the time of the day and you want to show this on the console. The usual way to do this could be as follows:

```
function getTime() {
  return new Date().toLocaleTimeString();
}

console.log(getTime()); // "21:00:24 GMT+0530 (IST)"
```

If you were doing **continuation passing style (CPS)**, you would pass a continuation to the `getTime()` function. Instead of returning a calculated value, the function would invoke the continuation, giving it the value as a parameter:

```
function getTime2(cont) {
  return cont(new Date().toLocaleTimeString());
}

getTime2(console.log); // similar result as above
```

What's the difference? The key is that we can apply this mechanism to make a recursive call into a tail call because all of the code that comes after will be provided in the recursive call itself. To make this clear, let's revisit the factorial function in the version that made it explicit that we weren't doing tail calls. The following code is fully equivalent to the previous one:

```
function fact2(n) {
  if (n === 0) {
    return 1;
  } else {
    const aux = fact2(n - 1);
    return n * aux;
  }
}
```

We will add a new parameter to the function for the continuation. What do we do with the result of the `fact(n-1)` call? We multiply it by `n`, so let's provide a continuation that will do just that. I'll rename the factorial function as `factC()` to make it clear that we are working with continuations, as shown in the following code:

```
function factC(n, cont) {
  if (n === 0) {
    return cont(1);

  } else {
    return factC(n - 1, x => cont(n * x));
  }
}
```

How would we get the final result? Easy: we can call `factC()` with a continuation that will just return whatever it's given:

```
factC(7, x => x); // 5040, correctly
```



In FP, a function that returns its argument as a result is usually called `identity()` for obvious reasons. In combinatory logic (which we won't be using), we would speak of the *I* combinator.

Can you understand how it worked? Then let's try out a more complex case with the Fibonacci function, which has two recursive calls in it, as shown in the following highlighted code:

```
const fibC = (n, cont) => {
  if (n <= 1) {
    return cont(n);

  } else {
    return fibC(n - 2, p => fibC(n - 1, q => cont(p + q)));
  }
};
```

This is trickier: we call `fibC()` with `n-2` and a continuation that says that whatever that call returned, call `fibC()` with `n-1`, and when *that* call returns, sum the results of both calls and pass that result to the original continuation.

Let's see just one more example, one that involves a loop with an undefined number of recursive calls. By then, you should have some idea about how to apply CPS to your code—though I'll readily admit, it can become really complex!

We saw this function in the *Traversing a tree structure* section earlier in this chapter. The idea was to print out the DOM structure, like this:

```
<body>
| <script>
| <div>
| | <div>
| | | <a>
| | | <div>
| | | | <ul>
| | | | | <li>
| | | | | | <a>
| | | | | | | <div>
| | | | | | | | <div>
| | | | | | | | <div>
| | | | | | | | | <br>
| | | | | | | | | <div>
| | | | | | | | | <ul>
| | | | | | | | | <li>
| | | | | | | | | | <a>
| | | | | | | | | | <li>
...etc!
```

The function we ended up designing back then was the following:

```
const traverseDom2 = (node, depth = 0) => {
  console.log(`${"| ".repeat(depth)}<${node.nodeName.toLowerCase()}>`);
  Array.from(node.children).forEach(child =>
    traverseDom2(child, depth + 1)
  );
};
```

Let's start by making this fully recursive, getting rid of the `forEach()` loop. We have seen this technique before, so we can just move on to the following result; note how the following code forms its loops by using recursion:

```
var traverseDom3 = (node, depth = 0) => {
  console.log(`${"| ".repeat(depth)}<${node.nodeName.toLowerCase()}>`);
  const traverseChildren = (children, i = 0) => {
    if (i < children.length) {
      traverseDom3(children[i], depth + 1);
      return traverseChildren(children, i + 1); // loop
    }
    return;
  };
  return traverseChildren(Array.from(node.children));
};
```

Now, we have to add a continuation to `traverseDom3()`. The only difference from the previous cases is that the function doesn't return anything, so we won't be passing any arguments to the continuation. It's also important to remember the implicit `return` at the end of the `traverseChildren()` loop: we must call the continuation:

```
var traverseDom3C = (node, depth = 0, cont = () => {}) => {
  console.log(`${"| ".repeat(depth)}<${node.nodeName.toLowerCase()}>`);
  const traverseChildren = (children, i = 0) => {
    if (i < children.length) {
      return traverseDom3C(children[i], depth + 1, () =>
        traverseChildren(children, i + 1)
      );
    }
    return cont();
  };
  return traverseChildren(Array.from(node.children));
};
```

We opted to give a default value to `cont`, so we can simply call `traverseDom3C(document.body)` as before. If we try out this logic, it works—but the problem of the potentially high number of pending calls hasn't been solved; let's look for a solution to this in the following section.

Trampolines and thunks

For the last solution to our problem, we shall have to think about the cause of the problem. Each pending recursive call creates a new entry stack. Whenever the stack gets too empty, the program crashes and our algorithm is history. So, if we can work out a way to avoid the stack growth, we should be free. The solution, in this case, is quite imposing and requires thunks and a trampoline—let's see what these are!

First, a **thunk** is really quite simple: it's just a nullary function (so, with no parameters) that helps delay a computation, providing a form of **lazy evaluation**. If you have a thunk, then, unless you call it, you won't get its value. For example, if you want to get the current date and time in ISO format, you could get it with `new Date().toISOString()`; however, if you provide a thunk that calculates that, you won't get the value until you actually invoke it:

```
const getIsoDateAndTime = () => new Date().toISOString(); // a thunk

const isoDateAndTime = getIsoDateAndTime(); // getting the thunk's value
```

What's the use of this? The problem with recursion is that a function calls itself, and calls itself, and calls itself, and so on until the stack blows over. Instead of directly calling itself, we are going to have the function return a thunk, which, when executed, will actually recursively call the function. So, instead of having the stack grow more and more, it will actually be quite flat, since the function will never get to actually call itself; the stack will grow by one position, when you call the function, and then get back to its size, as soon as the function returns its thunk.

But who gets to do the recursion? That's where the concept of a **trampoline** comes in. A trampoline is just a loop that calls a function, gets its return, and, if it is a thunk, then it calls it so that recursion will proceed, but in a flat, linear, way! The loop is exited when the thunk evaluation returns an actual value instead of a new function. Look at the following code:

```
const trampoline = (fn) => {
  while (typeof fn === 'function') {
    fn = fn();
  }
  return fn;
};
```

How can we apply this to an actual function? Let's start with a simple one that just sums all numbers from 1 to n , but in a recursive, guaranteed-to-cause-stack-crash fashion. Our simple `sumAll()` function could just be the following:

```
const sumAll = n => (n == 0 ? 0 : n + sumAll(n - 1));
```

However, if we start trying this function out, we'll eventually stumble and get a crash, as you can see in the following examples:

```
sumAll(10); // 55
sumAll(100); // 5050
sumAll(1000); // 500500
sumAll(10000); // Uncaught RangeError: Maximum call stack size exceeded
```

The stack problem will come up sooner or later depending on your machine, your memory size, and so on, but it will come, no doubt about that. Let's rewrite the function in continuation-passing style so that it will become tail recursive. We will just apply the same technique that we saw earlier, as shown in the following code:

```
const sumAllC = (n, cont) =>
  n === 0 ? cont(0) : sumAllC(n - 1, v => cont(v + n));

sumAllC(10000, console.log); // crash as earlier
```


Now, let's apply a simple rule: whenever you are going to return from a call, instead return a thunk that will, when executed, do the call that you actually wanted to do. The following code implements that change:

```
const sumAllT = (n, cont) =>
  n === 0 ? () => cont(0) : () => sumAllT(n - 1, v => () => cont(v + n));
```

Whenever there would have been a call to a function, we now return a thunk. How do we get to run this function? This is the missing detail. You need an initial call that will invoke `sumAllT()` the first time and (unless the function was called with a zero argument) a thunk will be immediately returned. The trampoline function will call the thunk, and that will cause a new call, and so on, until we eventually get a thunk that simply returns a value, and then the calculation will be ended:

```
const sumAll2 = n => trampoline(sumAllT(n, x => x));
```

In fact, you probably wouldn't want a separate `sumAllT()` function, so you'd go for something like this:

```
const sumAll3 = n => {
  const sumAllT = (n, cont) =>
    n === 0 ? () => cont(0) : () => sumAllT(n - 1, v => () => cont(v + n));

  return trampoline(sumAllT(n, x => x));
};
```

There's only one problem left: what would we do if the result of our recursive function wasn't a value, but rather a function? The problem there would be on the `trampoline()` code that, as long as the result of the thunk evaluation is a function, goes back again and again to evaluate it. The simplest solution would be to return a thunk, but wrapped in an object, as shown in the following code:

```
function Thunk(fn) {
  this.fn = fn;
}

var trampoline2 = thk => {
  while (typeof thk === "object" && thk.constructor.name === "Thunk") {
    thk = thk.fn();
  }
  return thk;
};
```

The difference now would be that, instead of returning a thunk, you'd write something as `return (v) => new Thunk(() => cont(v+n))`, so our new trampolining function can now distinguish an actual thunk (which is meant to be invoked and executed) from any other kind of result (which is meant to be returned).

So if you happen to have a recursive algorithm, but it won't run because of stack limits, you can fix it in a reasonable way by going through the following steps:

1. Changing all recursive calls to tail recursion using continuations
2. Replacing all `return` statements so that they'll return thunks
3. Replacing the call to the original function with a trampoline call to start the calculations

Of course, this doesn't come free. You'll notice that, when using this mechanism, there's extra work involving returning thunks, evaluating them, and so on, so you can expect the total time to go up. Nonetheless, this is a cheap price to pay if the alternative is having a nonworking solution to a problem!

Recursion elimination

There's yet one other possibility that you might want to explore, but that falls beyond the realm of FP and into algorithm design. It's a computer science fact that any algorithm that is implemented using recursion has an equivalent version that doesn't use recursion at all and instead depends on a stack. There are ways to systematically transform recursive algorithms into iterative ones, so, if you run out of all options (that is, if not even continuations or thunks can help you), then you'd have a final opportunity to achieve your goals by replacing all recursion with iteration. We won't be getting into this—as I said, this elimination has little to do with FP—but it's important to know that the tool exists and that you might be able to use it.

Summary

In this chapter, we saw how we can use recursion, a basic tool in FP, as a powerful technique to create algorithms for problems that would probably require far more complex solutions otherwise. We started by considering what recursion is and how to think recursively in order to solve problems, then moved on to see some recursive solutions to several problems in different areas, and ended by analyzing potential problems with deep recursion and how to solve them.

In Chapter 10, *Ensuring Purity – Immutability*, we shall get back to a concept we saw earlier in the book, function purity, and see some techniques that will help us guarantee that a function won't have any side effects by ensuring the immutability of arguments and data structures.

Questions

9.1. **Into reverse:** Can you program a `reverse()` function, but implement it in a recursive fashion? Obviously, the best way to go about this would be using the standard string `reverse()` method, as detailed in https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Array/reverse, but that wouldn't do for a question on recursion, would it?

9.2. **Climbing steps:** Suppose you want to climb up a ladder with n steps. At each time you raise your foot, you may opt to climb up one or two rungs. In how many different ways can you climb up that ladder? For example, you can climb a four-rung ladder in five different ways:

- Always taking one step at a time
- Always taking two steps at a time
- Taking two steps first, then one, and then one
- Taking one step first, then two, and then one
- Taking one step first, then another one, and finishing with two

9.3. **Longest common subsequence:** A classic dynamic programming problem is as follows: given two strings, find the length of the longest subsequence present in both of them. Be careful: we define a subsequence as a sequence of characters that appear in the same relative order, but not necessarily next to each other. For example, the longest common subsequence of `INTERNATIONAL` and `CONTRACTOR` is `N...T...R...A...T...O`. Try it out with or without memoizing and see the difference!

9.4. **Symmetrical queens:** In the eight queens puzzle that we previously solved, there is only one solution that shows symmetry in the placement of the queens. Can you modify your algorithm to find it?

9.5. **Sorting recursively:** There are many sorting algorithms that can be described with recursion; can you implement them?

- **Selection sort:** Find the maximum element of the array, remove it, recursively sort the rest, and then push the maximum element to the end of the sorted rest.
- **Insertion sort:** Take the first element of the array, sort the rest, and finish by inserting the removed element into its correct place in the sorted rest.
- **Merge sort:** Divide the array into two parts, sort each one, and finish by merging the two sorted parts into a sorted list.

9.6. **Completing callbacks:** In our `findR()` function, we did not provide all possible parameters to the `cb()` callback. Can you fix that? Your solution should be along the lines of what we did for `map()` and other functions.

9.7. **Recursive logic:** We didn't get to code `every()` and `some()` using recursion: can you do that?

9.8 **What could go wrong?** A developer decided that he could write a somewhat shorter version of Quicksort. He reasoned that the pivot didn't need any special handling since it would be set into its correct place anyway when sorting `greaterEqual`. Can you foresee any possible problems with this? The following code highlights the changes that the developer made with regard to the original version we saw earlier:

```
const quicksort = arr => {
  if (arr.length < 2) {
    return arr;
  } else {
    const pivot = arr[0];
    const smaller = arr.filter(x => x < pivot);
    const greaterEqual = arr.filter(x => x >= pivot);
    return [...quicksort(smaller), ...quicksort(greaterEqual)];
  }
};
```

9.9. More efficiency: Let's try to make `quicksort()` a bit more efficient by avoiding having to call `filter()` twice. Along the lines of what we saw in the *Calculating several values at once* section in Chapter 5, *Programming Declaratively – A Better Style*, write a `partition(arr, pr)` function that, given an array `arr` and a predicate `fn`, will return two arrays: the values of `arr` for which `fn` is `true` in the first one, and the rest of the values of `arr` in the second one:

```
const quicksort = arr => {
  if (arr.length < 2) {
    return arr;

  } else {
    const pivot = arr[0];
    const [smaller, greaterEqual] = partition(arr.slice(1), x => x <
pivot);
    return [...quicksort(smaller), pivot, ...quicksort(greaterEqual)];
  }
};
```

10

Ensuring Purity - Immutability

In [Chapter 4, *Behaving Properly – Pure Functions*](#), when we considered pure functions and their advantages, we saw that side effects such as modifying a received argument or a global variable were frequent causes of impurity. Now, after several chapters dealing with many aspects and tools of FP, let's talk about the concept of *immutability*: how to work with objects in such a way that accidentally modifying them will become harder or, even better, impossible.

We cannot force developers to work in a safe, guarded way, but if we find some way to make data structures immutable (meaning that they cannot be directly changed, except through some interface that never allows us to modify the original data and produces new objects instead), then we'll have an enforceable solution. In this chapter, we will look at two distinct approaches to working with such immutable objects and data structures:

- **Basic JavaScript ways**, such as freezing objects, plus cloning to create new ones instead of modifying existing objects
- **Persistent data structures**, with methods that allow us to update them without changing the original and without the need to clone everything either, for higher performance



A warning: the code in this chapter isn't production-ready; I wanted to focus on the main points and not on all the myriad details having to do with properties, getters, setters, lenses, prototypes, and more that you should take into account for a full, bulletproof, solution. For actual development, I'd very much recommend going with a third-party library, but only after checking that it really applies to your situation. We'll be recommending several such libraries, but of course, there are many more that you could use.

Going the straightforward JavaScript way

One of the biggest causes of side effects was the possibility of a function modifying global objects or its arguments. All non-primitive objects are passed as references, so if/when you modify them, the original objects will be changed. If we want to stop this (without just depending on the goodwill and clean coding of our developers), we may want to consider some straightforward JavaScript techniques to disallow those side effects:

- Avoiding mutator functions that directly modify the object that they are applied to
- Using `const` declarations to prevent variables from being changed
- Freezing objects so that they can't be modified in any way
- Creating (changed) clones of objects to avoid modifying the original
- Using getters and setters to control what is changed and how
- Using a functional concept—lenses—to access and set attributes

Let's take a look at each technique in more detail.

Mutator functions

A common source of unexpected problems comes from the fact that several JavaScript methods are actually mutators that modify the underlying object. In this case, by merely using them, you will be causing a side effect, which you may not even recognize. Arrays are the most basic sources of problems and the list of troublesome methods isn't short:

- `copyWithin()` lets you copy elements within the array.
- `fill()` fills an array with a given value.
- `push()` and `pop()` let you add or delete elements at the end of an array.
- `shift()` and `unshift()` work in the same way as `push()` and `pop()`, but at the beginning of the array.
- `splice()` lets you add or delete elements anywhere within the array.
- `reverse()` and `sort()` modify the array in place, reversing its elements or ordering them.



Refer to https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Array#Mutator_methods for more on each method.

Let's take a look at an example we saw in the *Argument mutation* section of Chapter 4, *Behaving Properly – Pure Functions*:

```
const maxStrings = a => a.sort().pop();

let countries = ["Argentina", "Uruguay", "Brasil", "Paraguay"];

console.log(maxStrings(countries)); // "Uruguay"
console.log(countries); // ["Argentina", "Brasil", "Paraguay"]
```

Our `maxStrings()` function returns the highest value in the array, but also modifies the original array; this is a side effect of the `sort()` and `pop()` mutator functions. In this case and others, you might generate a copy of the array and then work with that: both the spread operator and `.slice()` are useful:

```
const maxStrings2 = a => [...a].sort().pop();

const maxStrings3 = a => a.slice().sort().pop();

let countries = ["Argentina", "Uruguay", "Brasil", "Paraguay"];

console.log(maxStrings2(countries)); // "Uruguay"
console.log(maxStrings3(countries)); // "Uruguay"

console.log(countries); // ["Argentina", "Uruguay", "Brasil", "Paraguay"] -
unchanged
```

Both new versions of our `maxStrings()` functions are now functional, without side effects, because the mutator methods have been applied to copies of the original argument.

Of course, setter methods are also mutators and will logically produce side effects because they can do just about anything. If this is the case, you'll have to go for some of the other solutions that will be described later in this chapter.

Constants

If the mutations don't happen because of using some JavaScript methods, then we might want to attempt to use `const` definitions, but that just won't work. In JavaScript, a `const` definition means that the *reference* to the object or array cannot change (you cannot assign a different object to it) but you can still modify the properties of the object itself. We can see this in the following code:

```
const myObj = {d: 22, m: 9};
console.log(myObj);
```



```
// {d: 22, m: 9}

myObj = {d: 12, m: 4};
// Uncaught TypeError: Assignment to constant variable.

myObj.d = 12; // but this is fine!
myObj.m = 4;
console.log(myObj);
// {d: 12, m: 4}
```

You cannot modify the value of `myObj` by assigning it a new value, but you can modify the current value of `myObj` so that only the reference to an object is constant, and not the object's values themselves. (By the way, this would have also happened with arrays.) So, if you decide to use `const` everywhere, you will only be safe against direct assignments to objects and arrays. More modest side effects, such as changing an attribute or an array element, will still be possible, so this is not a solution.

There are two methods that can work: using *freezing* to provide unmodifiable structures, and *cloning* to produce modified new ones. These are probably not the best ways to go about forbidding objects from being changed but can be used as a makeshift solution. Let's take a look at them in more detail, starting with freezing.

Freezing

If we want to avoid the possibility of a programmer accidentally or willingly modifying an object, freezing it is a valid solution. After an object has been frozen, any attempts at modifying it will silently fail—JavaScript won't report an error or throw an exception, but it won't alter the object either. In the following example, if we attempt to make the same changes we made in the previous section, they just won't have any effect, and `myObj` will be unchanged:

```
const myObj = { d: 22, m: 9 };
Object.freeze(myObj);

myObj.d = 12; // won't have effect...
myObj.m = 4;

console.log(myObj);
// Object {d: 22, m: 9}
```



Don't confuse freezing with sealing: `Object.seal()`, when applied to an object, prohibits adding or deleting properties to it. This means that the structure of the object is immutable, but the attributes themselves can be changed. `Object.freeze()` includes not only sealing properties but also making them unchangeable. See https://developer.mozilla.org/en/docs/Web/JavaScript/Reference/Global_Objects/Object/seal and https://developer.mozilla.org/en/docs/Web/JavaScript/Reference/Global_Objects/Object/freeze for more on this.

There is only one problem with this solution: freezing an object is a *shallow* operation that freezes the attributes themselves, similar to what a `const` declaration does. If any of the attributes are objects or arrays themselves, with further objects or arrays as properties, and so on, they can still be modified. We will only be considering data here; you may also want to freeze, say, functions, but for most use cases, it's data you want to protect:

```
let myObj3 = {
  d: 22,
  m: 9,
  o: {c: "MVD", i: "UY", f: {a: 56}}
};
```

```
Object.freeze(myObj3);
console.log(myObj3); // {d:22, m:9, o:{c:"MVD", i:"UY", f:{ a:56}}}
```

This is only partially successful, as we can see when we try changing some attributes:

```
myObj3.d = 8888; // wont' work, as earlier
myObj3.o.f.a = 9999; // oops, does work!!
console.log(myObj3); // {d:22, m:9, o:{c:"MVD", i:"UY", f:{ a:9999 }}}
```

Modifying `myObj3.d` didn't work because the object is frozen, but that doesn't extend to objects within `myObj3`, so changing `myObj3.o.f.a` did work.

If we want to achieve real immutability for our object, we need to write a routine that will freeze all the levels of an object. Fortunately, it's easy to achieve this by applying recursion. (We saw similar applications of recursion in the *Traversing a tree structure* section of the previous Chapter 9, *Designing Functions - Recursion*.) Mainly, the idea is to freeze the object itself and then recursively freeze each of its properties. We must ensure that we only freeze the object's own properties; we shouldn't mess with the prototype of the object, for example:

```
const deepFreeze = obj => {
  if (obj && typeof obj === "object" && !Object.isFrozen(obj)) {
    Object.freeze(obj);
    Object.getOwnPropertyNames(obj).forEach(prop => deepFreeze(obj[prop]););
  }
};
```

```
    }  
    return obj;  
};
```

Note that, in the same way as `Object.freeze()` works, `deepFreeze()` also freezes the object *in place*. I wanted to keep the original semantics of the operation so that the returned object will always be the original one. If we wanted to work in a purer fashion, we should make a copy of the original object first (we'll learn how to do this in the next section) and then freeze that.

A small possible problem remains, but with a very bad result: what would happen if an object included a reference to itself? We can avoid this if we skip freezing already frozen objects: backward circular references would be ignored since the objects they refer to would already be frozen. So, the logic we wrote took care of that problem and there's nothing more to be done!

If we apply `deepFreeze()` to an object, we can safely pass it to any function, knowing that there simply is no way in which it can be modified. You can also use this property to test whether a function modifies its arguments: deep freeze them, call the function, and if the function depends on modifying its arguments, it won't work because the changes will be silently ignored. So, how can we return a result from a function if it involves a received object? This can be solved in many ways. A simple one uses cloning, as we'll see.



Check the *Questions* section at the end of this chapter for another way of freezing an object by means of proxies.

In this section, we dealt with one of the methods we can use to avoid changes in objects. Now, let's look at an alternative involving cloning.

Cloning and mutating

If mutating an object isn't allowed, then you must create a new object. For example, if you use Redux, a reducer is a function that receives the current state and an action (essentially, an object with new data) and produces the new state. Modifying the current state is totally forbidden and we could avoid this error by always working with frozen objects, as we saw in the previous section. To fulfill the reducer's requirements, we have to be able to clone the original state, as well as mutate it according to the received action. The resulting object will become the new state.



You may want to revisit the *More general looping* section of Chapter 5, *Programming Declaratively – A Better Style*, where we wrote a basic `objCopy()` function that provides a different approach from the one shown here.

To round things off, we should also freeze the returned object, just like we did with the original state. But let's start at the beginning: how do we clone an object? Of course, you can always do this by hand, but that's not something you'd really want to consider when working with large, complex objects. For example, if you wanted to clone `oldObject` to produce `newObject`, doing it by hand would imply a lot of code:

```
let oldObject = {
  d: 22,
  m: 9,
  o: {c: "MVD", i: "UY", f: {a: 56}}
};

let newObject = {
  d: oldObject.d,
  m: oldObject.m,
  o: {c: oldObject.o.c, i: oldObject.o.i, f: {a: oldObject.o.f.a}}
};
```

This manual solution is obviously a lot of work, and error-prone as well; you could easily forget an attribute! Going for more automatic solutions, there are a couple of straightforward ways of copying arrays or objects in JavaScript, but they have the same *shallowness* problem. You can make a (shallow) copy of an object with `Object.assign()` or by using spreading:

```
let newObject1 = Object.assign({}, myObj);
let newObject2 = {...myObj};
```

To create a (again, shallow) copy of an array, you can either use `slice()` or spreading, as we saw in the *Mutator functions* section earlier in this chapter:

```
let myArray = [1, 2, 3, 4];
let newArray1 = myArray.slice();
let newArray2 = [...myArray];
```

What's the problem with these solutions? If an object or array includes objects (which may themselves include objects), we get the same problem that we had when freezing: objects are copied by reference, which means that a change in the new object will also imply changing the old object:

```
let oldObject = {
  d: 22,
```

```
    m: 9,
    o: { c: "MVD", i: "UY", f: { a: 56 } }
  };
let newObject = Object.assign({}, oldObject);

newObject.d = 8888;
newObject.o.f.a = 9999;

console.log(newObject);
// {d:8888, m:9, o: {c:"MVD", i:"UY", f: {a:9999}}} -- ok

console.log(oldObject);
// {d:22, m:9, o: {c:"MVD", i:"UY", f: {a:9999}}} -- oops!!
```

In this case, notice what happened when we changed some properties of `newObject`. Changing `newObject.d` worked fine, but changing `newObject.o.f.a` also impacted `oldObject` since `newObject.o` and `oldObject.o` are actually references to the very same object.

There is a simple solution to this based on JSON. If we `stringify()` the original object and then `parse()` the result, we'll get a new object that's totally separate from the old one:

```
const jsonCopy = obj => JSON.parse(JSON.stringify(obj));
```

By using `JSON.stringify()`, we can convert our object into a string. Then, `JSON.parse()` creates a (new) object out of that string; simple! This works with both arrays and objects, but there's a problem. If any of the properties of the object have a constructor, they won't be invoked: the result will always be composed of plain JavaScript objects. We can see this very simply with a `Date()`:

```
let myDate = new Date();
let newDate = jsonCopy(myDate);
console.log(typeof myDate, typeof newDate); // object string
```

While `myDate` is an object, `newDate` turns out to be a string with a value, `"2019-11-08T01:32:56.365Z"`, which is the current date and time at the moment we did the conversion.

We could do a recursive solution, just like we did with deep freezing, and the logic is quite similar. Whenever we find a property that is really an object, we invoke the appropriate constructor:

```
const deepCopy = obj => {
  let aux = obj;
  if (obj && typeof obj === "object") {
    aux = new obj.constructor();
  }
}
```

```
    Object.getOwnPropertyNames(obj).forEach(
      prop => (aux[prop] = deepCopy(obj[prop]))
    );
  }

  return aux;
};
```

Whenever we find that a property of an object is actually another object, we invoke its constructor before continuing. This solves the problem we found with dates or, in fact, with any object! If we run the preceding code, but using `deepCopy()` instead of `jsonCopy()`, we'll get `object object` as output, as it should be. If we check the types and constructors, everything will match. Furthermore, the data changing experiment will also work fine now:

```
let oldObject = {
  d: 22,
  m: 9,
  o: { c: "MVD", i: "UY", f: { a: 56 } }
};

let newObject = deepCopy(oldObject);
newObject.d = 8888;
newObject.o.f.a = 9999;

console.log(newObject);
// {d:8888, m:9, o:{c:"MVD", i:"UY", f:{a:9999}}}
```

```
console.log(oldObject);
// {d:22, m:9, o:{c:"MVD", i:"UY", f:{a:56}}} -- unchanged!
```

Let's check out the last few lines. Modifying `newObject` had absolutely no impact on `oldObject`, so both objects are completely separate.

Now that we know how to copy an object, we can follow these steps:

1. Receive a (frozen) object as an argument
2. Make a copy of it, which won't be frozen
3. Take values from that copy that we can use in our code
4. Modify the copy at will
5. Freeze it
6. Return it as the result of the function

All of this is viable, though a bit cumbersome. So, let's add a couple of functions that will help bring everything together.

Getters and setters

When following the steps provided at the end of the previous section, you'll notice that every time you want to update a field, things become troublesome and prone to errors. Let's use a common technique to add a pair of functions, getters, and setters. These are as follows:

- *getters* can be used to get values from a frozen object by unfreezing them so that they can be used.
- *setters* allow you to modify any property of an object. You can do this by creating a new and updated version of it, leaving the original untouched.

Let's build our getters and setters.

Getting a property

Back in the *Getting a property from an object* section of Chapter 6, *Producing Functions – Higher-Order Functions*, we wrote a simple `getField()` function that could handle getting a single attribute out of an object. (See question 6.5 in that chapter for the missing companion `setField()` function.) Let's take a look at how we can code this. We can have a straightforward version, as follows:

```
const getField = attr => obj => obj[attr];
```

We can even go one better by applying currying so that we have a more general version:

```
const getField = curry((attr, obj) => obj[attr]);
```

We could get a deep attribute out of an object by composing a series of applications of `getField()` calls, but that would be rather cumbersome. Instead, let's create a function that will receive a *path*—an array of field names—and return the corresponding part of the object or be undefined if the path doesn't exist. Using recursion is appropriate here and simplifies coding! Observe the following code:

```
const getByPath = (arr, obj) => {
  if (arr[0] in obj) {
    return arr.length > 1
      ? getByPath(arr.slice(1), obj[arr[0]])
      : deepCopy(obj[arr[0]]);
  } else {
    return undefined;
  }
};
```

Basically, we look for the first string in the path to see whether it exists in the object. If it doesn't, the operation fails, so we return `undefined`. If successful, and we have still more strings in the path, we use recursion to keep digging into the object; otherwise, we return a deep copy of the value of the attribute.

Once an object has been frozen, you cannot *defrost* it, so we must resort to making a new copy of it; `deepCopy()` is appropriate for doing this. Let's try out our new function:

```
let myObj3 = {
  d: 22,
  m: 9,
  o: {c: "MVD", i: "UY", f: {a: 56}}
};
deepFreeze(myObj3);

console.log(getByPath(["d"], myObj3)); // 22
console.log(getByPath(["o"], myObj3)); // {c: "MVD", i: "UY", f: {a: 56}}
console.log(getByPath(["o", "c"], myObj3)); // "MVD"
console.log(getByPath(["o", "f", "a"], myObj3)); // 56
```

We can also check that returned objects are not frozen:

```
let fObj = getByPath(["o", "f"], myObj3);
console.log(fObj); // {a: 56}

fObj.a = 9999;
console.log(fObj); // {a: 9999} -- it's not frozen
```

Here, you can see that we could directly update the `fObj` object, so that means it wasn't frozen. Now that we've written our getter, we can move on to creating a setter.

Setting a property by path

Now, we can code a similar `setByPath()` function that will take a path, a value, and an object and update an object. This is *not* a pure function, but we'll use it to write a pure one; wait and see! Here is the code:

```
const setByPath = (arr, value, obj) => {
  if (!(arr[0] in obj)) {
    obj[arr[0]] =
      arr.length === 1 ? null : Number.isInteger(arr[1]) ? [] : {};
  }

  if (arr.length > 1) {
    return setByPath(arr.slice(1), value, obj[arr[0]]);
  }
}
```



```
    } else {
      obj[arr[0]] = value;
      return obj;
    }
  };
```

Here, we are using recursion to get into the object, creating new attributes if needed, until we have traveled the full length of the path. One important detail when creating attributes is whether we need an array or an object. We can determine that by checking the next element in the path: if it's a number, then we need an array; otherwise, an object will do. When we get to the end of the path, we simply assign the new given value.



If you like this way of doing things, you should check out the *seamless-immutable* library, which works in this fashion. The *seamless* part of the name alludes to the fact that you still work with normal objects—albeit frozen—which means you can use `map()`, `reduce()`, and so on. You can read more about this at <https://github.com/rtfeldman/seamless-immutable>.

Now, you can write a function that will be able to take a frozen object and update an attribute within it, returning a new, also frozen, object:

```
const updateObject = (arr, obj, value) => {
  let newObj = deepCopy(obj);
  setByPath(arr, value, newObj);
  return deepFreeze(newObj);
};
```

Let's check out how it works. To do this, we'll run several updates on the `myObj3` object we have been using:

```
let new1 = updateObject(["m"], myObj3, "sep");
// {d: 22, m: "sep", o: {c: "MVD", i: "UY", f: {a: 56}}};

let new2 =updateObject(["b"], myObj3, 220960);
// {d: 22, m: 9, o: {c: "MVD", i: "UY", f: {a: 56}}, b: 220960};

let new3 =updateObject(["o", "f", "a"], myObj3, 9999);
// {d: 22, m: 9, o: {c: "MVD", i: "UY", f: {a: 9999}}};

let new4 =updateObject(["o", "f", "j", "k", "l"], myObj3, "deep");
// {d: 22, m: 9, o: {c: "MVD", i: "UY", f: {a: 56, j: {k: "deep"}}}};
```

Given this pair of functions, we have finally gotten ourselves a way to keep immutability:

- Objects must be frozen from the beginning
- Getting data from objects is done with `getByPath()`
- Setting data is done with `updateObject()`, which internally uses `setByPath()`

In this section, we learned how to get and set values from an object in a way that keeps objects immutable. Let's now take a look at a variation of this concept—*lenses*—that will allow us to not only get and set values but also apply a function to the data.

Lenses

There's another way to get and set values, which goes by the name of *optics*, including *lenses* and *prisms* (which we'll look at later in this chapter). What are lenses? They are functional ways of *focusing* (another optical term!) on a given spot in an object so that we can access or modify its value in a non-mutating way. In this section, we'll look at some examples of usage of lenses and consider two implementations: first, a simple one based on objects, and then a more complete one that's interesting because of some of the techniques we will be using.



Several libraries provide full implementations of lenses that are production-ready and more complete than what we saw in this chapter; for example, check out Ramda: <http://ramdajs.com/docs/#lens>

Working with lenses

Both implementations will share basic functionality, so let's start by skipping what lenses are or how they are built and look at some examples of their usage instead. First, let's create a sample object that we will work with: some data about a writer (his name sounds familiar) and his books:

```
const author = {
  user: "fkereki",
  name: {
    first: "Federico",
    middle: "",
    last: "Kereki",
  },
  books: [
    {name: "Google Web Toolkit", year: 2010},
```

```
    {name: "Functional Programming", year: 2017},  
    {name: "Javascript Cookbook", year: 2018},  
  ],  
};
```

We shall assume that several functions exist; we'll see how they are implemented in upcoming sections. A lens depends on having a getter and a setter for a given attribute, and we can build one by directly using `lens()` or by means of `lensProp()` for briefer coding. Let's create a lens for the `user` attribute:

```
const lens1 = lens(getField("user"), setField("user"));
```

This defines a lens that focuses on the `user` attribute. Since this is a common operation, it can also be written more compactly:

```
const lens1 = lensProp("user");
```

Both of these lenses allow us to focus on the `user` attribute of whatever object we use them with. With lenses, there are three basic operations, and we'll follow tradition by using the names that most (if not all) libraries follow:

- `view()`: Used to access the value of an attribute
- `set()`: Used to modify the value of an attribute
- `over()`: Used to apply a function to an attribute and change its value

These functions are curried (as we saw in the previous chapter). So, to access the `user` attribute, we can write something similar to the following:

```
console.log(view(lens1, author));  
console.log(view(lens1)(author));  
/*  
  fkereki, in both cases  
*/
```

The `view()` function takes a lens as its first parameter. When this is applied to an object, it produces the value of whatever the lens focuses on—in our case, the `user` attribute. Of course, you could apply sequences of `view()` functions to get to deeper parts of the object:

```
console.log(view(lensProp("last"), view(lensProp("name"), author)));  
/*  
  Kereki  
*/
```

Instead of writing such a series of `view()` calls, we'll compose lenses so that we can focus more deeply on an object. Let's take a look at one final example, which shows how we access an array:

```
const lensBooks = lensProp("books");
console.log(
  "The author wrote " + view(lensBooks, author).length + " book(s) "
);
/*
  The author wrote 3 book(s)
*/
```

In the future, should there be any change in the `author` structure, a simple change in the `lensBooks` definition would be enough to keep the rest of the code unchanged.



You can also use lenses to access other structures: refer to question 10.5 for a way to use lenses with arrays, and question 10.6 for how to use lenses so that they work with maps.

Moving on, the `set()` function allows us to set the value of the focus of the lens:

```
console.log(set(lens1, "FEFK", author));
/*
  user: "FEFK",
  name: {first: "Federico", middle: "", last: "Kereki"},
  books: [
    {name: "Google Web Toolkit", year: 2010},
    {name: "Functional Programming", year: 2017},
    {name: "Javascript Cookbook", year: 2018},
  ],
}
*/
```

The result of `set()` is a new object with a changed value. Using `over()` is similar in that a new object is returned, but in this case, the value is changed by applying a mapping function to it:

```
const newAuthor = over(lens1, x => x + x + x, author);
console.log(newAuthor);
/*
  user: "fkerekifkerekifkereki",
  name: {first: "Federico", middle: "", last: "Kereki"},
  books: [
    {name: "GWT", year: 2010},
    {name: "FP", year: 2017},
    {name: "CB", year: 2018},
  ]
*/
```

```
    ],  
  }  
  */
```

There are more functions you can do with lenses, but we'll just go with these three for now.



Take a look at question 10.4 for an interesting idea on how to use lenses to access *virtual attributes* that don't actually exist in an object.

To finish this section, I'd recommend looking at some third-party optics libraries to get a glimpse into all the functionality that's available. Now that we have an idea of what to expect when using lenses, let's learn how to implement them.

Implementing lenses with objects

The simplest way to implement a lens is by representing it with an object with just two properties: a getter and a setter. In this case, we'd have something like this:

```
const lens = (getter, setter) => ({getter, setter});
```

This is easy to understand: given a getter and a setter, `lens()` just creates an object with those two attributes. With this definition, `lensProp()` would be as follows:

```
const lensProp = attr => lens(getField(attr), setField(attr));
```

The first function, `lensProp()`, creates a getter/setter pair by using `getField()` and `setField()`; very straightforward. Now that we have our lens, how do we implement the three basic functions that we saw in the previous section? Viewing an attribute just requires applying the getter:

```
const view = curry((lens, obj) => lens.getter(obj));
```

To be consistent with the rest of the functions we've been using, we are going to apply currying. Similarly, setting an attribute is a matter of applying the setter:

```
const set = curry((lens, newVal, obj) => lens.setter(newVal, obj));
```

Finally, applying a mapping function to an attribute is sort of a *two-for-one* operation: we use the getter to get the current value of the attribute, we apply the function to it, and we use the setter to store the calculated result:

```
const over = curry((lens, mapfn, obj) =>
  lens.setter(mapfn(lens.getter(obj)), obj)
);
```

Now that we can do all three operations, we have working lenses! What about composition? Lenses have a peculiar characteristic: they're composed backward, or left-to-right, so you start with the most generic and end with the most specific. That certainly goes against intuition: we'll learn about this in more detail in the next section, but for now, we'll keep with tradition:

```
const composeTwoLenses = (lens1, lens2) => ({
  getter: obj => lens2.getter(lens1.getter(obj)),
  setter: curry((newVal, obj) =>
    lens1.setter(lens2.setter(newVal, lens1.getter(obj)), obj)
  ),
});
```

The code is sort of impressive, but not too hard to understand. The getter for the composition of two lenses is the result of using the first lens' getter and then applying the second lens' getter to that result. The setter for the composition is just a tad more complex, but follows along the same lines; can you see how it works? Now, we can compose lenses easily; let's start with an invented nonsensical object:

```
const deepObject = {
  a: 1,
  b: 2,
  c: {
    d: 3,
    e: {
      f: 6,
      g: {i: 9, j: {k: 11}},
      h: 8,
    },
  },
};
```

Now, we can define a few lenses:

```
const lC = lensProp("c");
const lE = lensProp("e");
const lG = lensProp("g");
const lJ = lensProp("j");
```

We can try composing our new lens in a couple of ways, just for variety, and to check that everything works:

```
const lJK = composeTwoLenses(lJ, lK);
const lGJK = composeTwoLenses(lG, lJK);
const lEGJK = composeTwoLenses(lE, lGJK);
const lCEGJK1 = composeTwoLenses(lC, lEGJK);
console.log(view(lCEGJK1)(deepObject));

const lCE = composeTwoLenses(lC, lE);
const lCEG = composeTwoLenses(lCE, lG);
const lCEGJ = composeTwoLenses(lCEG, lJ);
const lCEGJK2 = composeTwoLenses(lCEGJ, lK);
console.log(view(lCEGJK2)(deepObject));

/*
   11 both times
*/
```

With `lCEGJ1`, we composed some lenses, starting with the latter ones. With `lCEGJ2`, we started with the lenses at the beginning, but the results are the same. Now, let's try setting some values. We want to get down to the `k` attribute and set it to 60. We can do this by using the same lens we just applied:

```
const setTo60 = set(lCEGJ1, 60, deepObject);
/*
  {a: 1, b: 2, c: {d: 3, e: {f: 6, g: {i: 9, j: { k: 60 }}, h: 8}}}
*/
```

The composed lens worked perfectly, and the value was changed. (Also, a new object was returned; the original is unmodified, as we wanted.) To finish, let's also verify that we can do `over()` with our lens and try to duplicate the `k` value so that it becomes 22. Just for variety, let's use the other composed lens, even though we know that it works in the same way:

```
const setToDouble = over(lCEGJK2, x => x * 2, deepObject);
/*
  {a: 1, b: 2, c: {d: 3, e: {f: 6, g: {i: 9, j: { k: 22 }}, h: 8}}}
*/
```

Now, we have learned how to implement lenses in a simple fashion. However, let's consider a different way of achieving the same objective by using actual functions to represent a lens. This will allow us to do composition in the standard way, without the need for any special lens function.

Implementing lenses with functions

The previous implementation of lenses with objects works well, but we want to look at a different way of doing things that will let us work with more advanced functional ideas. This will involve some concepts we'll be analyzing in more detail in [Chapter 12, Building Better Containers – Functional Data Types](#), but here, we'll use just what we need so that you don't have to go and read that chapter now! Our lenses will work in the same way that the preceding ones did, except that since they will be functions, we'll be able to compose them with no special composing code.

What's the key concept here? A lens will be a function, based on a getter and a setter pair, that will construct a *container* (actually an object, but let's go with the container name) with a `value` attribute and a `map` method (in [Chapter 12, Building Better Containers – Functional Data Types](#), we'll see that this is a *functor*, but you don't need to know that now). By having specific mapping methods, we'll implement our `view()`, `set()`, and `over()` functions. Our `lens()` function is as follows. We'll explain the details of this later:

```
const lens = (getter, setter) => fn => obj =>
  fn(getter(obj)).map(value => setter(value, obj));
```

Let's consider its parameters:

- The `getter` and `setter` parameters are the same as before; we can even use the very same `lensProp()` function that we used earlier in this chapter.
- The `fn` function is the magic sauce that makes everything work: depending on what we want to do with the lens, we'll provide a specific function—more on this later!
- The `obj` parameter is the object that we want to apply the lens to.

Let's code our `view()` function. For this, we'll need an auxiliary class, `Constant`, that, given a value, `v`, produces a container with that value, and a `map` function that returns the very same container:

```
class Constant {
  constructor(v) {
    this.value = v;
    this.map = () => this;
  }
}
```

With this, we can now code `view()`:

```
const view = curry(
  (lensAttr, obj) => lensAttr(x => new Constant(x))(obj).value
```



```
);

const user = view(lensProp("user"), author);
/*
   fkereki
*/
```

What happens here? Let's follow this step by step; it's a bit tricky!

1. We use `lensProp()` to create a lens focusing on the user attribute.
2. Our `view()` function passes the constant-building function to `lens()`.
3. Our `lens()` function uses the getter to access the user attribute in the author object.
4. Then, the value that we received is used to create a constant container.
5. The `map()` method is invoked—that method returns the very same container.
6. The value attribute of the container is accessed, and that's the value that the getter retrieved in *step 3*; wow!

With that under our belt, let's move on to `set()` and `over()`, which will require a different auxiliary function, to create a container whose value may vary:

```
class Variable {
  constructor(v) {
    this.value = v;
    this.map = fn => new Variable(fn(v));
  }
}
```

In this case (as opposed to `Constant` objects), the `map()` method really does something: when provided with a function, it applies it to the value of the container and returns a new `Variable` object with the resulting value. The `set()` function can be implemented easily:

```
const set = curry(
  (lensAttr, newVal, obj) =>
    lensAttr(() => new Variable(newVal))(obj).value
);

const changedUser = set(lensProp("user"), "FEFK", author);
/*
{
  user: "FEFK",
  name: {first: "Federico", middle: "", last: "Kereki"},
  books: [
    {name: "GWT", year: 2010},
    {name: "FP", year: 2017},
  ]
}
*/
```

```

    {name: "CB", year: 2018},
  ],
};
*/

```

In this case, when the lens invokes the container's `map()` method, it will produce a new container with a new value, and that makes all the difference. To understand how this works, follow the same six steps we saw for `get()`—the only difference will be in *step 5*, where a new, different container is produced.

Now that we've survived this (tricky indeed!) code, the `over()` function is simple, and the only difference is that instead of mapping to a given value, you use the mapping `mapfn` function provided to compute the new value for the container:

```

const over = curry(
  (lensAttr, mapfn, obj) =>
    lensAttr(x => new Variable(mapfn(x)))(obj).value
);

const newAuthor = over(lensProp("user"), x => x + x + x, author);
/*
  user: "fkerekifkerekifkereki",
  name: {first: "Federico", middle: "", last: "Kereki"},
  books: [
    {name: "GWT", year: 2010},
    {name: "FP", year: 2017},
    {name: "CB", year: 2018},
  ],
}
*/

```

As you can see, the difference between `set()` and `over()` is that, in the former case, you provide the value to replace the original one, while in the latter case, you provide a function to calculate the new value. Other than that, both are similar.

To finish, let's verify that `compose()` can be applied to our functor-based lenses:

```

const lastName = view(
  compose(
    lensProp("name"),
    lensProp("last")
  )
)(author);
/*
  Kereki
*/

```

Here, we created two individual lenses for `name` and `last`, and we composed them with the very same `compose()` function that we developed back in Chapter 8, *Connecting Functions – Pipelining and Composition*. Using this composite lens, we focused on the last name of the author without any problem, so everything worked as expected.



It seems to go against logic that lenses should be composed from left to right; this appears to be backward. This is something that troubles developers, and if you Google for an explanation, you'll find many. To combat this question on your own, I suggest spelling out how `compose()` works in full—two functions will be enough—and then substitute the definitions of lenses; you'll see why and how everything works out.

Now that we've looked at lenses, we can move on and look at prisms, another optics tool.

Prisms

Lenses, as we saw in the previous section, are useful for working with *product* types. However, prisms are useful for working with *sum* types. But what are they? (We'll look at products and unions in more detail in the *Data types* section of the next chapter.) The idea is that a product type is always built out of the same options, such as an object from a class, while a sum type will likely have different structures—extra or missing attributes, for example. When you use a lens, you assume that the object that you'll be applying it to has a known structure with no variations, but what do you use if the object may have different structures? The answer is prisms. Let's take a look at how they are used first; then, we'll look at their implementation.

Working with prisms

Working with prisms is similar to using lenses, except for what happens when an attribute is not present. Let's take a look at an example from the previous section:

```
const author = {
  user: "fkereki",
  name: {
    first: "Federico",
    middle: "",
    last: "Kereki"
  },
  books: [
    { name: "GWT", year: 2010 },
    { name: "FP", year: 2017 },
    { name: "CB", year: 2018 }
  ]
}
```

```
    ]  
  };
```

If we wanted to access the `user` attribute using prisms, we would write something like the following—don't worry about the details; we'll look at the actual implementation later:

```
const pUser = prismProp("user");  
  
console.log(review(pUser, author).toString());  
  
/*  
  fkereki  
*/
```

Here, we define a prism using a `prismProp()` function, which parallels our previous `lensProp()` one. Then, we use the prism with the `preview()` function, which is analog to `get()` with lenses, and the result is the same as if we had used lenses; no surprises there. What would have happened if we asked for a non-existing `pseudonym` attribute? Let's see:

```
const pPseudonym = prismProp("pseudonym");  
  
console.log(preview(pPseudonym, author).toString());  
/*  
  undefined  
*/
```

So far, we may not be able to see any differences, but let's see what happens if we try to compose lenses or prisms with several missing attributes. Say you wanted to access a (missing!) `pseudonym.usedSince` attribute with lenses, without taking precautions and checking that the attributes exist. Here, you would get the following output:

```
const lPseudonym = lensProp("pseudonym");  
const lUsedSince = lensProp("usedSince");  
  
console.log(  
  "PSEUDONYM, USED SINCE",  
  view(compose(lPseudonym, lUsedSince))(author)  
);  
/*  
  TypeError: Cannot read property 'usedSince' of undefined  
  .  
  . many more error lines, snipped out  
  .  
*/
```

On the other hand, since prisms already take missing values into account, this would cause no problems, and we'd simply get an `undefined` result:

```
const pUsedSince = prismProp("usedSince");

console.log(
  "PSEUDONYM, USED SINCE",
  review(compose(pPseudonym, pUsedSince))(author).toString()
);
/*
  undefined
*/
```

What happens if we want to set a value? The analog function to `set()` is `review()`; let's take a look at how it would work. The idea is that whatever attribute we specify will be set, if, and only if, the attribute already exists. So, if we attempt to change the `user.name` attribute, this will work:

```
const fullAuthor2 = review(
  compose(prismProp("name"), prismProp("first")),
  "FREDERICK",
  author
);

/*
{ user: 'fkereki',
  name: { first: 'FREDERICK', middle: '', last: 'Kereki' },
  books:
    [ { name: 'GWT', year: 2010 },
      { name: 'FP', year: 2017 },
      { name: 'CB', year: 2018 } ] }
*/
```

However, if we try to modify the (non-existent) `pseudonym` attribute, the original, unchanged object will be returned:

```
const fullAuthor3 = review(pPseudonym, "NEW ALIAS", author);

/*
{ user: 'fkereki',
  name: { first: 'Federico', middle: '', last: 'Kereki' },
  books:
    [ { name: 'GWT', year: 2010 },
      { name: 'FP', year: 2017 },
      { name: 'CB', year: 2018 } ] }
*/
```

So, using prisms takes care of all possible missing or optional fields. How do we implement this new optic? Let's take a look.

Implementing prisms

How do we implement prisms? We will take our cue from our lenses implementation and make a few changes. When getting an attribute, we must check whether the object we are processing is not `null` or `undefined` and whether the attribute we want is in the object. We can make do by making small changes to our original `getField()` function:

```
const getFieldP = curry((attr, obj) =>
  obj && attr in obj ? obj[attr] : undefined
);
```

Here, we're checking for the existence of the object and the attribute: if everything's fine, we return `obj[attr]`; otherwise, we return `undefined` otherwise. The changes for `setField()` are very similar:

```
const setFieldP = curry((attr, value, obj) =>
  obj && attr in obj ? { ...obj, [attr]: value } : { ...obj }
);
```

If the object and the attribute both exist, we return a new object by changing the attribute's value; otherwise, we return a copy of the object. That's all there is to it!

Now that we've learned how to access objects in functional ways; let's analyze persistent data structures that can be modified in very efficient ways, without the need for a full copy of the original object.

Creating persistent data structures

If you want to change something in a data structure and you just go and change it, your code will be full of side effects. On the other hand, copying complete structures every time is a waste of time and space. There's a middle ground to this that has to do with persistent data structures, which, if handled correctly, let you apply changes while creating new structures in an efficient way.

Given that there are many possible data structures you could work with, let's just take a look at a few examples:

- Working with lists, one of the simplest data structures
- Working with objects, a very common necessity in JavaScript programs
- Dealing with arrays, which will prove to be harder to work with

Let's get started!

Working with lists

Let's consider a simple procedure: suppose you have a list and you want to add a new element to it. How would you do this? Here, we can assume that each node is a `ListNode` object:

```
class ListNode {
  constructor(value, next = null) {
    this.value = value;
    this.next = next;
  }
}
```

A possible list would be as follows, where a `list` variable would point to the first element. Take a look at the following diagram; can you tell what is missing in the list and where?

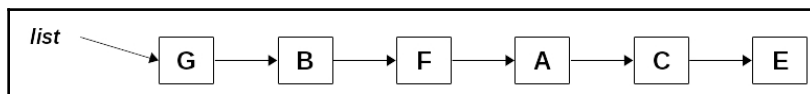


Figure 10.1: The initial list

If you wanted to add **D** between **B** and **F** (the sample list is something musicians will understand: the *Circle of Thirds*, a musical concept, but missing the **D** note), the simplest solution would be to add a new node and change an existing one. This would result in the following:

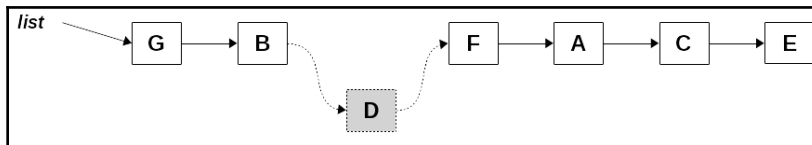


Figure 10.2: The list now has a new element – we had to modify an existing one to perform the addition

However, working in this way is obviously non-functional and it's clear we are modifying data. There is a different way of working, and that's by creating a persistent data structure, in which all the alterations (insertions, deletions, and modifications) are done separately, being careful not to modify existing data. On the other hand, if some parts of the structure can be reused, this is done to gain in performance. Doing a persistent update would return a new list, with some nodes that are duplicates of some previous ones, but with no changes whatsoever to the original list. This can be seen in the following diagram:

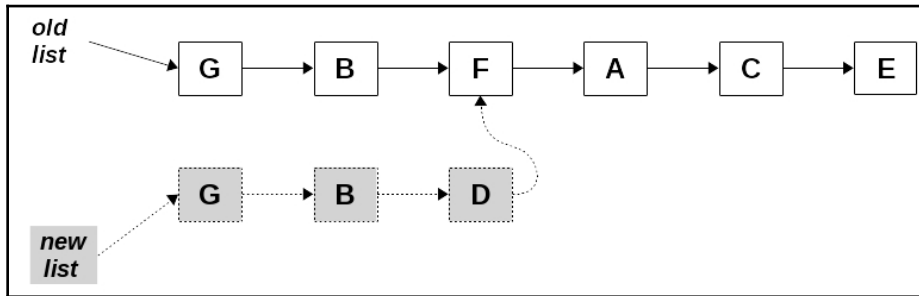


Figure 10.3: The dotted elements show the newly returned list, which shares some elements with the old one

Updating a structure in this way requires duplicating some elements to avoid modifying the original structure, but part of the list is shared.

Of course, we will also deal with updates or deletions. Starting again with the list shown in the following diagram, if we wanted to update its fourth element, the solution would imply creating a new subset of the list, up to and including the fourth element, while keeping the rest unchanged:

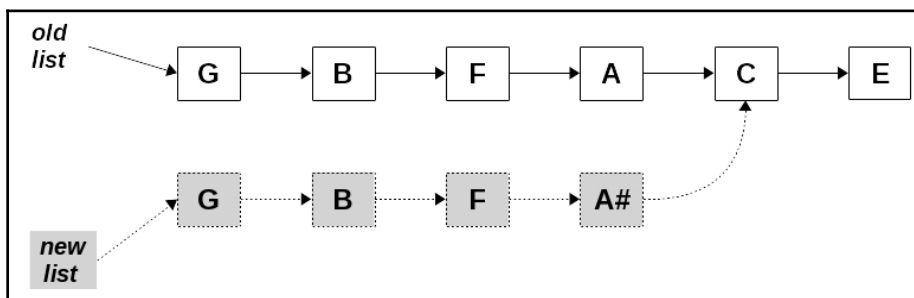


Figure 10.4: Our list, with a changed element

Removing an element would also be similar. Let's do away with the third element, **F**, in the original list, as follows:

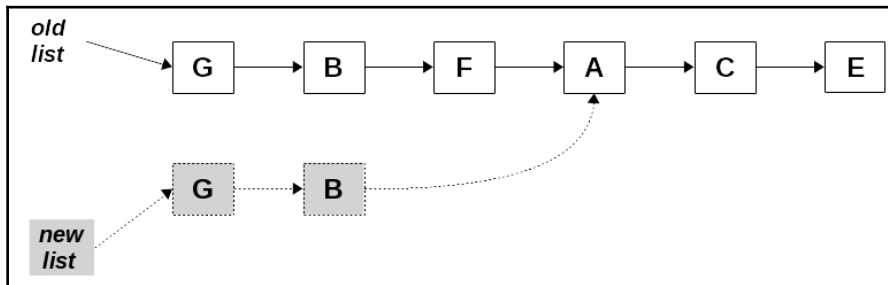


Figure 10.5: The original list, after removing the third element in a persistent way

Working with lists or other structures can always be solved to provide data persistence. For now, focus on what will probably be the most important kind of work for us: dealing with simple JavaScript objects. After all, all data structures are JavaScript objects, so if we can work with objects, we can work with other structures.

Updating objects

This kind of method can also be applied to more common requirements, such as modifying an object. This is a very good idea for, say, Redux users: a reducer can be programmed so that it will receive the old state as a parameter and produce an updated version with the minimum needed changes, without altering the original state in any way.

Imagine you had the following object:

```
myObj = {
  a: ...,
  b: ...,
  c: ...,
  d: {
    e: ...,
    f: ...,
    g: {
      h: ...,
      i: ...
    }
  }
};
```

Let's assume you wanted to modify the value of the `myObj.d.f` attribute, but working in a persistent way. Instead of copying the full object (with the `deepCopy()` function that we used earlier), we could create a new object that has several attributes in common with the previous object, but new ones for the modified ones. This can be seen in the following diagram:

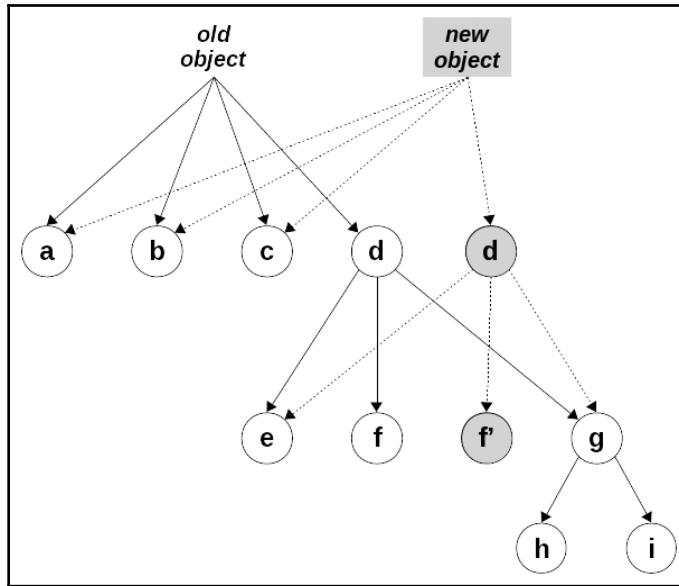


Figure 10.6: A persistent way of editing an object, that is, by sharing some attributes and creating others

The old and new objects share most of the attributes, but there are new `d` and `f` attributes, so you managed to minimize the changes when creating the new object.

If you want to do this by hand, you would have to write, in a very cumbersome way, something like the following. Most attributes are taken from the original object, but `d` and `d.f` are new:

```
newObj = {
  a: myObj.a,
  b: myObj.b,
  c: myObj.c,
  d: {
    e: myObj.d.e,
    f: the new value,
    g: myObj.d.g
  }
};
```

We saw some code similar to this earlier in this chapter when we decided to work on a cloning function. Here, let's go for a different type of solution. In fact, this kind of update can be automated:

```
const setIn = (arr, val, obj) => {
  const newObj = Number.isInteger(arr[0]) ? [] : {};

  Object.keys(obj).forEach(k => {
    newObj[k] = k !== arr[0] ? obj[k] : null;
  });

  newObj[arr[0]] =
    arr.length > 1 ? setIn(arr.slice(1), val, obj[arr[0]]) : val;
  return newObj;
};
```

The logic is recursive, but not too complex. First, we figure out, at the current level, what kind of object we need: either an array or an object. Then, we copy all the attributes from the original object to the new one, except the property we are changing. Finally, we set that property to the given value (if we have finished with the path of property names) or we use recursion to go deeper with the copy.



Note the order of the arguments: first the path, then the value, and finally the object. We are applying the concept of putting the most *stable* parameters first and the most variable last. If you curry this function, you can apply the same path to several different values and objects, and if you fix the path and the value, you can still use the function with different objects.

Let's give this logic a try. We'll start with a nonsensical object, but with several levels and even an array of objects for variety:

```
let myObj1 = {
  a: 111,
  b: 222,
  c: 333,
  d: {
    e: 444,
    f: 555,
    g: {
      h: 666,
      i: 777
    },
    j: [{k: 100}, {k: 200}, {k: 300}]
  }
};
```

We can test this by changing `myObj.d.f` to a new value:

```
let myObj2 = setIn(["d", "f"], 88888, myObj1);

/*
{
  a: 111,
  b: 222,
  c: 333,
  d: {
    e: 444,
    f: 88888,
    g: {h: 666, i: 777},
    j: [{k: 100}, {k: 200}, {k: 300}]
  }
}
*/

console.log(myObj.d === myObj2.d); // false
console.log(myObj.d.f === myObj2.d.f); // false
console.log(myObj.d.g === myObj2.d.g); // true
```

The logs at the bottom verify that the algorithm is working correctly: `myObj2.d` is a new object, but `myObj2.d.g` is reusing the value from `myObj`.

Updating the array in the second object lets us test how the logic works in those cases:

```
let myObj3 = setIn(["d", "j", 1, "k"], 99999, myObj2);
/*
{
  a: 111,
  b: 222,
  c: 333,
  d: {
    e: 444,
    f: 88888,
    g: {h: 666, i: 777},
    j: [{k: 100}, {k: 99999}, {k: 300}]
  }
}
*/

console.log(myObj.d.j === myObj3.d.j); // false
console.log(myObj.d.j[0] === myObj3.d.j[0]); // true
console.log(myObj.d.j[1] === myObj3.d.j[1]); // false
console.log(myObj.d.j[2] === myObj3.d.j[2]); // true
```

We can compare the elements in the `myObj.d.j` array with the ones in the newly created object. You will see that the array is a new one, but two of the elements (the ones that weren't updated) are still the same objects that were in `myObj`.

This obviously isn't enough to get by. Our logic can update an existing field, or even add it if it wasn't there, but you'd also need to eliminate and attribute. Libraries usually provide many more functions, but let's work on the deletion of an attribute for now so that we can look at some of the other important structural changes we can make to an object:

```
const deleteIn = (arr, obj) => {
  const newObj = Number.isInteger(arr[0]) ? [] : {};

  Object.keys(obj).forEach(k => {
    if (k !== arr[0]) {
      newObj[k] = obj[k];
    }
  });

  if (arr.length > 1) {
    newObj[arr[0]] = deleteIn(arr.slice(1), obj[arr[0]]);
  }
  return newObj;
};
```

The logic here is similar to that of `setIn()`. The difference is that we don't always copy all the attributes from the original object to the new one: we only do that if we haven't arrived at the end of the array of path properties. Continuing with the series of tests after the updates, we get the following:

```
myObj4 = deleteIn(["d", "g"], myObj3);
myObj5 = deleteIn(["d", "j"], myObj4);

// {a: 111, b: 222, c: 333, d: {e: 444, f: 88888}};
```

With this pair of functions, we can manage to work with persistent objects by making changes, additions, and deletions in an efficient way that won't create new objects needlessly.



The most well-known library for working with immutable objects is the appropriately named *immutable.js*, which can be found at <https://facebook.github.io/immutable-js/>. The only weak point about it is its notoriously obscure documentation. However, there's an easy solution for that: check out *The Missing Immutable.js Manual with all the Examples you'll ever need* at <http://untangled.io/the-missing-immutable-js-manual/> and you won't have any trouble!

A final caveat

Working with persistent data structures requires some cloning, but how would you implement a persistent array? If you think about this, you'll realize that, in that case, there would be no way out apart from cloning the whole array after each operation. This would mean that an operation such as updating an element in an array, which took a constant time, would now take a length of time proportional to the size of the array.



In algorithm complexity terms, we would say that updates went from being an $O(1)$ operation to an $O(n)$ one. Similarly, access to an element may become an $O(\log n)$ operation, and similar slowdowns might be observed for other operations, such as mapping and reducing.

How do we avoid this? There's no easy solution. For example, you may find that an array is internally represented as a binary search tree (or even more complex data structures) and that the persistence library provides the necessary interface so that you'll still be able to use it as an array, not noticing the internal difference.

When using this kind of library, the advantages of having immutable updates without cloning may be offset in part by some operations that may become slower. If this becomes a bottleneck in your application, you might have to go so far as changing the way you implement immutability or even work out how to change your basic data structures to avoid the time loss, or at least minimize it.

Summary

In this chapter, we looked at two different approaches (used by commonly available immutability libraries) to avoiding side effects by working with immutable objects and data structures: one based on using JavaScript's *object freezing* plus some special logic for cloning, and the other based on applying the concept of persistent data structures with methods that allow all kinds of updates without changing the original or requiring full cloning.

In Chapter 11, *Implementing Design Patterns – The Functional Way*, we will focus on a question that's often asked by object-oriented programmers: how are design patterns used in FP? Are they required, available, or usable? Are they still practiced but with a new focus on functions rather than on objects? We'll answer these questions with several examples, showing where and how they are equivalent or how they differ from the usual OOP practices.

Questions

10.1. Freezing by proxying: In the *Chaining and fluent interfaces* section of Chapter 8, *Connecting Functions – Pipelining and Composition*, we used a proxy to get operations in order to provide automatic chaining. By using a proxy for *setting* and *deleting* operations, you may do your own *freezing* (if, instead of setting an object's property, you'd rather throw an exception). Implement a `freezeByProxy(obj)` function that will apply this idea to forbid all kinds of updates (adding, modifying, or deleting properties) for an object. Remember to work recursively in case an object has other objects as properties!

10.2. Inserting into a list, persistently: In the *Working with lists* section, we described how an algorithm could add a new node to a list, but in a persistent way, by creating a new list. Implement an `insertAfter(list, newKey, oldKey)` function that will create a new list but add a new node with `newKey` just after the node with `oldKey`. Here, you'll need to assume that the nodes in the list were created by the following logic:

```
class Node {
  constructor(key, next = null) {
    this.key = key;
    this.next = next;
  }
}

const node = (key, next) => new Node(key, next);

let c3 = node("G", node("B", node("F", node("A", node("C", node("E")))));
```

10.3. Composing many lenses: Write a `composeLenses()` function that will allow you to compose as many simple lenses as you want, instead of only two as in `composeTwoLenses()`, along the same lines as what we did in Chapter 8, *Connecting Functions – Pipelining and Composition*, when we moved from `composeTwo()` to a generic `compose()` function.

10.4. Lenses by path: In this chapter, we created lenses using `getField()` and `setField()`. Then, we used composition to access deeper attributes. Can you create a lens by giving a path and allow shorter code?

10.5. Accessing virtual attributes: By using lenses, you can view (and even set) attributes that don't actually exist in an object. Here are some tips to let you develop that. First, can you write a getter that will access an object such as `author` and return the author's full name in LAST NAME, FIRST NAME format? Second, can you write a setter that, given a full name, will split it in half and set its first and last names? With those two functions, you could write the following:

```
const fullNameLens = lens(
  ...your getter...,
  ...your setter...
);

console.log(view(fullNameLens, author));
/*
  Kereki, Federico
*/

console.log(set(fullNameLens, "Doe, John", author));
/*
{ user: 'fkereki',
  name: { first: 'John', middle: '', last: 'Doe' },
  books:
    [ { name: 'GWT', year: 2010 },
      { name: 'FP', year: 2017 },
      { name: 'CB', year: 2018 } ] }
*/
```

10.6. Lenses for arrays? What would happen if you created a lens like so and applied it to an array? If there's a problem, could you fix it?

```
const getArray = curry((ind, arr) => arr[ind]);

const setArray = curry((ind, value, arr) => {
  arr[ind] = value;
  return arr;
});

const lensArray = ind => lens(getArray(ind), setArray(ind));
```

10.7. Lenses into maps: Write a `lensMap()` function that will create a lens that you can use to access and modify maps. You may want to look into cloning maps at https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Map for more information. Your function should be declared as follows. You'll have to write a couple of auxiliary functions as well:

```
const lensMap = key => lens(getMap(key), setMap(key));
```


11

Implementing Design Patterns - The Functional Way

In Chapter 10, *Ensuring Purity – Immutability*, we saw several functional techniques to solve different problems. However, programmers who are used to employing OOP may find that we have missed some well-known formula and solutions that are often used in imperative coding. Since design patterns are well known, and programmers will likely already be aware of how they are applied in other languages, it's important to take a look at how a functional implementation would be done.

In this chapter, we shall consider the solutions implied by *design patterns*, which are common in OOP, to see their equivalences in FP. This will help you to transition from OOP to a more functional approach and to learn more about the power and methods of FP, by seeing an alternative solution to problems you already knew.

In particular, we will study the following topics:

- The concept of *design patterns* and to what they apply
- A few OOP standard patterns and what alternatives we have in FP if we need one
- In particular, the *observer* pattern, which leads to *reactive programming*, a declarative way of dealing with events
- A discussion about FP design patterns, not related to the OOP ones

Understanding design patterns

One of the most relevant books in software engineering was *Design Patterns: Elements of Reusable Object-Oriented Software*, 1994, written by the **Gang of Four (GOF)**: Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. This book presented about two dozen different OOP *patterns* and has been recognized as a highly important book in computer science.



Patterns are actually a concept from architectural design, originally defined by an architect, Christopher Alexander.

In software terms, a *design pattern* is a generally applicable, reusable solution to a commonly-seen problem in software design. Rather than a specific finished and coded design, it's a description of a solution (the word *template* is also used) that can solve a given problem that appears in many contexts. Given their advantages, design patterns are on their own *best practices* that can be used by developers working with different kinds of systems, programming languages, and environments.

The GoF book obviously focused on OOP, and some of the patterns within cannot be recommended for or applied in FP. Other patterns are unnecessary or irrelevant because functional languages already provide standard solutions to the corresponding object-oriented problems. Even given this difficulty, since most programmers have been exposed to OOP design patterns and usually try to apply them even in other contexts such as FP, it makes sense to consider the original problems and then take a look at how a new solution can be produced. The standard object-based solutions may not apply, but the problem can still stand, so seeing how to solve it is still valid.

Patterns are often described in terms of four essential, basic elements:

- A simple, short *name* that is used to describe the problem, its solutions, and its consequences. The name is useful for talking with colleagues, explaining a design decision, or describing a specific implementation.
- The *context* to which the pattern applies: specific situations that require a solution, possibly with some extra conditions that must be met.
- A *solution* that lists the elements (classes, objects, functions, relationships, and so on) that you'll need to solve the given situation.
- The *consequences* (results and trade-offs) if you apply the pattern. You may derive some gains from the solution, but it may also imply some losses.

In this chapter, we will assume that the reader is already aware of the design patterns that we will be describing and using, so we won't be providing many details about them. Rather, we will focus on how FP either makes the problem irrelevant (because there is an obvious way of applying functional techniques to solve it) or solves it in some fashion. Also, we won't be going over all of the GoF patterns; we'll just focus on those for which applying FP is more interesting, bringing out more differences to the usual OOP implementations.

Design pattern categories

Design patterns are usually grouped into several distinct categories, according to their focus. The first three in the following list are the ones that appeared in the original GoF book, but more categories have been added. They are as follows:

- **Behavioral design patterns:** These have to do with interactions and communications between objects. Rather than focusing on how objects are created or built, the key consideration is how to connect them so that they can cooperate when performing a complex task, preferably in a way that provides well-known advantages, such as diminished coupling or enhanced cohesiveness.
- **Creational design patterns:** These deal with ways to create objects in a manner that is suitable for the current problem. With it, you can decide between several alternative objects, so the program can work differently depending on parameters that may be known at compilation time or runtime.
- **Structural design patterns:** These have to do with the composition of objects, forming larger structures from many individual parts and implementing relationships between objects. Some of the patterns imply inheritance or implementation of interfaces, whereas others use different mechanisms, all geared toward being able to dynamically change the way objects are composed at runtime.
- **Concurrency patterns:** These are related to dealing with multithreaded programming. Although FP is generally quite appropriate for this (given, for example, the lack of assignments and side effects), since we are working with JavaScript, these patterns are not very relevant to us.
- **Architectural patterns:** These are more high-level oriented, with a broader scope than the previous patterns we've listed, and provide general solutions to software architecture problems. As is, we aren't considering such problems in this book, so we won't deal with these either.



Coupling and cohesiveness are terms that were in use even before OOP came into vogue; they date back to the late '60s when *Structured Design* by Larry Constantine came out. The former measures the interdependence between any two modules, and the latter has to do with the degree to which all components of a module really belong together. Low coupling and high cohesiveness are good goals for software design because they imply that related things are close by and unrelated ones are separate.

Following along these lines, you could also classify design patterns as *object patterns* (which concern the dynamic relationships between objects) and *class patterns* that deal with the relationships between classes and subclasses (which are defined statically at compile time). We won't be worrying much about this classification because our point of view has more to do with behaviors and functions rather than classes and objects.

As we mentioned earlier, we can now readily observe that these categories are heavily oriented toward OOP, and the first three directly mention objects. However, without the loss of generality, we will look beyond the definitions, remember what problem we were trying to solve, and then look into analogous solutions with FP, which, if not 100% equivalent to the OOP ones, will in spirit be solving the same problem in a parallel way. Let's move on and start by considering *why* we want to deal with patterns at all!

Do we need design patterns?

There is an interesting point of view that says that design patterns are only needed to patch shortcomings of a programming language. The rationale is that if you can solve a problem with a given programming language in a simple, direct, and straightforward way, then you may not need a design pattern at all. (An example: if your language doesn't provide recursion, we would have to implement it on our own, but otherwise, you can just use it without further ado.) However, studying patterns lets you think about different ways of solving problems, so that's a point in their favor.

In any case, it's interesting for OOP developers to really understand why FP helps to solve some problems without the need for further tools. In the next section, we shall consider several well-known design patterns and take a look at why we don't need them or how we can easily implement them. It's also a fact that we have already applied several patterns earlier in the text, so we'll point out those examples as well.

We won't try, however, to express or convert all design patterns into FP terms. For example, the *Singleton* pattern basically requires a single, global, object, which is sort of opposed to everything that functional programmers are used to. Given our approach to FP (remember **Sorta Functional Programming (SPF)**, from the initial part of the first chapter of this book?), we won't mind either, and if a *Singleton* is required, we may consider using it, even though FP doesn't have an appropriate equivalent.

Finally, it must be said that our point of view may affect what is considered a pattern and what isn't. What may be a pattern to some may be considered a trivial detail for others. We will find some such situations, given that FP lets us solve some particular problems in easy ways, and we have already seen examples of that in previous chapters.

Object-oriented design patterns

In this section, we'll go over some of the GoF design patterns, check whether they are pertinent to FP, and study how to implement them. Of course, some design patterns don't get an FP solution. As we said, for example, there's no equivalent for a *Singleton*, which implies the foreign concept of a globally accessed object. Additionally, while it's true that you may no longer need OOP-specific patterns, developers will still think in terms of those. Also, finally, since we're not going fully functional if an OOP pattern fits, why not use it?

We will be considering the following:

- *Facade* and *Adapter*, to provide new interfaces to other code
- *Decorator* (also known as *Wrapper*) to add new functionality to existing code
- *Strategy*, *Template*, and *Command*, to let you fine-tune algorithms by passing functions as parameters
- *Observer*, which leads to reactive programming, a declarative way of dealing with events
- Other patterns that do not so fully match the corresponding OOP ones

Now, let's begin our study by analyzing a couple of similar patterns that let you use your code in somewhat different ways.

Facade and adapter

Out of these two patterns, let's start with the *Facade* or, more correctly, *Façade*. This is meant to solve the problem of providing a different interface to the methods of a class or to a library. The idea is to provide a new interface to a system that makes it easier to use. You might say that a Façade provides a better control panel to access certain functionalities, removing difficulties for the user.



Façade or facade? The original word is an architectural term meaning the *front of a building* and comes from the French language. According to this source and the usual sound of the cedilla (ç) character, its pronunciation is something like *fuh-sahd*. The other spelling probably has to do with the lack of international characters in keyboards and poses the following problem: shouldn't you read it as *faKade*? You may see this problem as the reverse of *celtic*, which is pronounced as *Keltic*, changing the *s* sound for a *k* sound.

The main problem that we want to solve is being able to use external code more easily (of course, if it were your code, you could handle such problems directly; we must assume you cannot—or shouldn't—try to modify that other code. This would be the case when you use any library that's available over the web, for example). The key to this is to implement a module of your own that will provide an interface that better suits your needs. Your code will use your module and won't directly interact with the original code.

Suppose that you want to do Ajax calls, and your only possibility is using some hard library with a really complex interface. With modules, you might write something like the following, working with an imagined, hard-to-use Ajax library:

```
// simpleAjax.js

import * as hard from "hardajaxlibrary";
// import the other library that does Ajax calls
// but in a hard, difficult way, requiring complex code

const convertParamsToHardStyle = params => {
  // do some internal steps to convert params
  // into whatever the hard library may require
};

const makeStandardUrl = url => {
  // make sure the url is in the standard
  // way for the hard library
};
```

```
const getUrl = (url, params, callback) => {
  const xhr = hard.createAnXmlHttpRequestObject();
  hard.initializeAjaxCall(xhr);
  const standardUrl = makeStandardUrl(url);
  hard.setUrl(xhr, standardUrl);
  const convertedParams = convertParamsToHardStyle(params);
  hard.setAdditionalParameters(params);
  hard.setCallback(callback);
  if (hard.everythingOk(xhr)) {
    hard.doAjaxCall(xhr);
  } else {
    throw new Error("ajax failure");
  }
};

const postUrl = (url, params, callback) => {
  // some similarly complex code
  // to do a POST using the hard library
};

export {getUrl, postUrl}; // the only methods that will be seen
```

Now, if you need to do GET or POST, instead of having to go through all of the complications of the provided hard Ajax library, you can use the new façade that provides a simpler way of working. Developers would just do `import {getUrl, postUrl} from "simpleAjax"` and could then work more reasonably.

However, why are we showing this code that, though interesting, doesn't show any particular FP aspects? The key is that, at least until modules are fully implemented in browsers, the internal implicit way to do this is with the usage of an IIFE (Immediately Invoked Function Expression) as we saw in the *Immediate invocation* section of Chapter 3, *Starting Out with Functions – A Core Concept*, using a *revealing module* pattern. The way to implement this would then be as follows:

```
const simpleAjax = (function() {
  const hard = require("hardajaxlibrary");

  const convertParamsToHardStyle = params => {
    // ...
  };

  const makeStandardUrl = url => {
    // ...
  };

  const getUrl = (url, params, callback) => {
```

```
    // ...
  };

  const postUrl = (url, params, callback) => {
    // ...
  };

  return {
    getUrl,
    postUrl
  };
}());
```

The reason for the *revealing module* name should be now obvious. With the preceding code, because of the JavaScript scope rules, the only visible attributes of `simpleAjax` will be `simpleAjax.getUrl` and `simpleAjax.postUrl`; using an IIFE lets us implement the module (and hence, the façade) safely, making implementation details private.

Now, the Adapter pattern is similar, insofar it is also meant to define a new interface. However, while Façade defines a new interface to old code, Adapter is used when you need to implement an old interface for a new code, so it will match what you already had. If you are working with modules, it's clear that the same type of solution that worked for Façade will work here, so we don't have to study it in detail. Now, let's continue with a well-known pattern, which you'll recognize we've already seen earlier in this book!

Decorator or wrapper

The *Decorator* pattern (also known as *wrapper*) is useful when you want to add additional responsibilities or functionalities to an object in a dynamic way. Let's consider a simple example, which we will illustrate with some React code. (Don't worry if you do not know this framework; the example will be easy to understand. The idea of going with React is because it can very well take advantage of this pattern. Also, we have already seen pure JavaScript higher-order function examples, so it's good to see something new.) Suppose we want to show some elements on the screen, and for debugging purposes, we want to show a thin red border around the object. How can you do it?

If you were using OOP, you would probably have to create a new subclass with the extended functionality. For this particular example, you might just provide some attribute with the name of some CSS class that would provide the required style, but let's keep our focus on OOP; using CSS won't always solve this software design problem, so we want a more general solution. The new subclass would *know* how to show itself with a border, and you'd use this subclass whenever you wanted an object's border to be visible.

With our experience of higher-order functions, we can solve this in a different way using *wrapping*; wrap the original function within another one, which would provide the extra functionality.

Note that we have already seen some examples of wrapping in the *Wrapping functions – keeping behavior* section of Chapter 6, *Producing Functions – Higher-Order Functions*. For example, in that section, we saw how to wrap functions to produce new versions that could log their input and output, provide timing information, or even memorize calls to avoid future delays. On this occasion, for variety, we are applying the concept to *decorate* a visual component, but the principle remains the same.

Let's define a simple React component, `ListOfNames`, that can display a heading and a list of people, and for the latter, it will use a `FullNameDisplay` component. The code for those elements would be as seen in the following fragment:

```
class FullNameDisplay extends React.Component {
  render() {
    return (
      <div>
        First Name: <b>{this.props.first}</b>
        <br />
        Last Name: <b>{this.props.last}</b>
      </div>
    );
  }
}

class ListOfNames extends React.Component {
  render() {
    return (
      <div>
        <h1>
          {this.props.heading}
        </h1>
        <ul>
          {this.props.people.map(v =>
            <FullNameDisplay first={v.first} last={v.last} />
          )}
        </ul>
      </div>
    );
  }
}
```

The `ListOfNames` component uses mapping to create a `FullNameDisplay` component to show data for each person. The full logic for our application could then be the following:

```
import React from "react";
import ReactDOM from "react-dom";

class FullNameDisplay extends React.Component {
  // ...as above...
}

class ListOfNames extends React.Component {
  // ...as above...
}

const GANG_OF_FOUR = [
  {first: "Erich", last: "Gamma"},
  {first: "Richard", last: "Helm"},
  {first: "Ralph", last: "Johnson"},
  {first: "John", last: "Vlissides"}
];

ReactDOM.render(
  <ListOfNames heading="GoF" people={GANG_OF_FOUR} />,
  document.body
);
```



In real life, you wouldn't put all of the code for every component in the same single source code file—and you would probably have a few CSS files. However, for our example, having everything in one place, and going with inline styles is enough, so bear with me and keep in mind the following saying: *Do as I say, not as I do.*

We can quickly test the result in the online React sandbox at <https://codesandbox.io/>; Google react online sandbox if you want some other options. The interface design isn't much to talk about (so please don't criticize my poor web page!) because we are interested in design patterns right now; refer to *Figure 11.1*, given as follows:

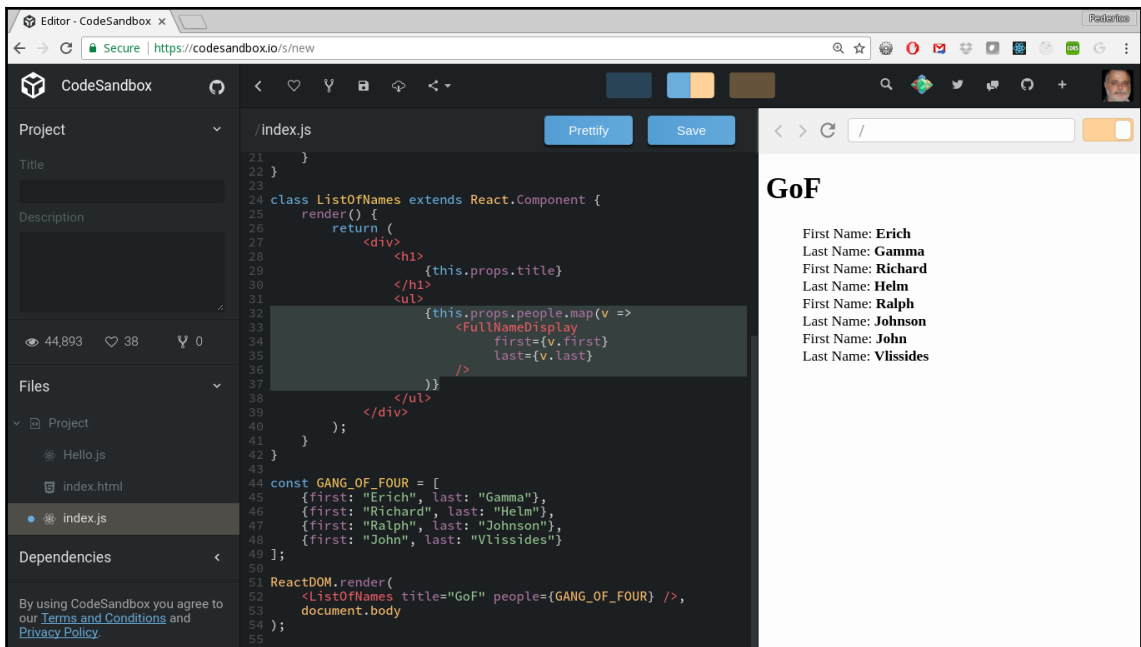


Figure 11.1: The original version of our components shows a (not much to speak about) list of names

In React, inline components are written in JSX (inline HTML style) and are actually compiled into objects, which are later transformed into HTML code to be displayed. Whenever the `render()` method is called, it returns a structure of objects. So, we will write a function that will take a component as a parameter and return a new JSX, a wrapped object. In our case, we'd like to wrap the original component within `<div>` with the required border:

```
const makeVisible = component => {
  return (
    <div style={{border: "1px solid red"}}>
      {component}
    </div>
  );
};
```

If you wish, you could make this function aware of whether it's executing in development mode or production; in the latter case, it would simply return the original component argument without any change, but let's not worry about that now.

We now have to change `ListOfNames` to use wrapped components; the new version would be as follows:

```
class ListOfNames extends React.Component {
  render() {
    return (
      <div>
        <h1>
          {this.props.title}
        </h1>
        <ul>
          {this.props.people.map(v =>
            makeVisible(
              <FullNameDisplay
                first={v.first}
                last={v.last}
              />
            )
          )}
        </ul>
      </div>
    );
  }
}
```

The decorated version of the code works as expected: each of the `ListOfNames` components is now wrapped in another component that adds the desired border to them; refer to *Figure 11.2*, given as follows:

The screenshot shows a CodeSandbox editor with a file named `index.js`. The code defines a `ListOfNames` component that renders a title and a list of people. The `makeVisible` function is used to wrap the list items in a `<div>` with a red border. The rendered output shows the title "GoF" and a list of four people, each with their first and last names displayed in a red-bordered box.

```

23
24 class ListOfNames extends React.Component {
25   render() {
26     return (
27       <div>
28         <h1>
29           {this.props.title}
30         </h1>
31         <ul>
32           {this.props.people.map(v =>
33             makeVisible(
34               <FullNameDisplay
35                 first={v.first}
36                 last={v.last}
37             )
38           )}
39         </ul>
40       </div>
41     );
42   }
43 }
44
45 const GANG_OF_FOUR = [
46   {first: "Erich", last: "Gamma"},
47   {first: "Richard", last: "Helm"},
48   {first: "Ralph", last: "Johnson"},
49   {first: "John", last: "Vlissides"}
50 ];
51
52
53 ReactDOM.render(
54   <ListOfNames title="GoF" people={GANG_OF_FOUR} />,
55   document.body
56 );
57

```

The rendered output shows the following list:

First Name: Erich
Last Name: Gamma
First Name: Richard
Last Name: Helm
First Name: Ralph
Last Name: Johnson
First Name: John
Last Name: Vlissides

Figure 11.2: The decorated ListOfNames component is still nothing much to look at, but now it shows an added border

In earlier chapters, we saw how to decorate a function, wrapping it inside of another function, so it would perform extra code and add a few functionalities. Now, here, we saw how to apply the same style of the solution to provide a *higher-order component* (as it's called in React parlance) wrapped in an extra `<div>` to provide some visually distinctive details.



If you have used Redux and the *react-redux* package, you may note that the latter's `connect()` method is also a decorator in the same sense; it receives a component class, and returns a new component class, connected to the store, for usage in your forms; refer to <https://github.com/reactjs/react-redux> for more details.

Let's move to a different set of patterns that will let us change how functions perform.

Strategy, Template, and Command

The *Strategy* pattern applies whenever you want to have the ability to change a class, method, or function, possibly in a dynamic way, by changing the way it actually does whatever it's expected to do. For example, a GPS application might want to find a route between two places by applying different strategies if the person is on foot, rides a bicycle, or goes by car. In that case, the fastest or the shortest routes might be desired. The problem is the same, but different algorithms must be applied, depending on the given condition.

By the way, does this sound familiar? If so, it is because we have already met a similar problem. When we wanted to sort a set of strings in different ways, in [Chapter 3, *Starting Out with Functions – A Core Concept*](#), we needed a way to specify how the ordering was to be applied or, equivalently, how to compare two given strings and determine which had to go first. Depending on the language, we had to sort applying different comparison methods.

Before trying an FP solution, let's consider more ways of implementing our routing function. You could make do by having a big enough piece of code, which would receive an argument declaring which algorithm to use, plus the starting and ending points. With these arguments, the function could do a switch or something similar to apply the correct path-finding logic. The code would be roughly equivalent to the following fragment:

```
function findRoute(byMeans, fromPoint, toPoint) {
  switch (byMeans) {
    case "foot":
      /*
       * find the shortest road
       * for a walking person
       */

    case "bicycle":
      /*
       * find a route apt
       * for a cyclist
       */

    case "car-fastest":
      /*
       * find the fastest route
       * for a car driver
       */

    case "car-shortest":
      /*
       * find the shortest route
       * for a car driver
       */
  }
}
```

```
    */  
  
    default:  
        /*  
         * plot a straight line,  
         * or throw an error,  
         * or whatever suits you  
         */  
    }  
}
```

This kind of solution is really not desirable, and your function is really the sum of a lot of distinct other functions, which doesn't offer a high level of cohesion. If your language doesn't support lambda functions (as was the case with Java, for example, until Java 8 came out in 2014), the OOP solution for this requires defining classes that implement the different strategies you may want, creating an appropriate object, and passing it around.

With FP in JavaScript, implementing strategies is trivial: instead of using a variable such as `byMeans` to switch, you provide a route-finding function (`routeAlgorithm()` in the following code) that will implement the desired path logic:

```
function findRoute(routeAlgorithm, fromPoint, toPoint) {  
    return routeAlgorithm(fromPoint, toPoint);  
}
```

You would still have to implement all of the desired strategies (there's no way around that) and decide which function to pass to `findRoute()`, but now that function is independent of the routing logic, and if you wanted to add new routing algorithms, you wouldn't touch `findRoute()`.

If you consider the *Template* pattern, the difference is that Strategy allows you to use completely different ways of achieving an outcome, while Template provides an overarching algorithm (or *template*) in which some implementation details are left to methods to be specified. In the same way, you can provide functions to implement the Strategy pattern; you can also provide them for a Template pattern.

Finally, the *Command* pattern also benefits from the ability to be able to pass functions as arguments. This pattern is meant to be enabled to encapsulate a request as an object, so for different requests, you have differently parameterized objects. Given that we can simply pass functions as arguments to other functions, there's no need for the enclosing object.

We also saw a similar use of this pattern back in the *A React-Redux reducer* section of Chapter 3, *Starting Out with Functions – A Core Concept*. There, we defined a table, each of whose entries was a callback that was called whenever needed. We could directly say that the Command pattern is just an **object-oriented (OO)** replacement for plain functions working as callbacks.

Let's now move on to a classic pattern that implies a new term, *reactive programming*, that's being thrown around a lot these days.

Observer and reactive programming

The idea of the *observer* pattern is to define a link between entities, so when one changes, all of the dependent entities are updated automatically. The *observable* can publish changes to its state, and its observer (which subscribed to the observable) will be notified of such changes.



There is a proposal for adding observables to JavaScript (see <https://github.com/tc39/proposal-observable>) but as of December 2019, it's still stuck at stage 1; check <https://github.com/tc39/proposal-observable/issues/191>. Hence, for the time being, it seems that using a library will still be mandatory.

There's an extension to this concept called **reactive programming**, which involves asynchronous streams of events (such as mouse clicks or keypresses) or data (from APIs or web sockets), and different parts of the application subscribing to observe such streams by passing callbacks that will get called whenever something new appears.



We won't be implementing reactive programming on our own; instead, we'll use RxJS, a JavaScript implementation of Reactive Extensions (*ReactiveX*) originally developed by Microsoft. RxJS is widely used in the Angular framework and can also be used in other frontend frameworks, such as React or Vue, or in the backend with Node. Learn more about RxJS at <https://rxjs-dev.firebaseio.com/> and <https://www.learnrxjs.io/>.

The techniques we will be showing in these sections are, confusingly, called both **Functional Reactive Programming (FRP)** and **Reactive Functional Programming (RFP)**; pick whichever you want! There is also a suggestion that FRP shouldn't be applied to discrete streams (so the name is wrong) but the expression is seen all over the web, which gives it some standing. But...what makes this functional, and why should we be interested in it? The key is that we will be using similar methods to `map()`, `filter()`, and `reduce()` to process those streams, and pick which events to process and how. OK, this may be confusing now, so bear with me and let's see some concepts first, and after that, some examples of FRP—or whatever you want to call it! We will be seeing the following:

- Several basic concepts and terms you'll need to work with FRP
- Some of the many available operators you'll use
- A couple of examples: detecting multi-clicks, and providing typeahead searches

Let's move on to analyze each item, starting with the basic ideas you need to know.

Basic concepts and terms

Using FRP requires getting used to several new terms, so let's begin with a short glossary:

- **Observable:** This represents a stream of (present or future) values and can be connected to an observer. You can create observables from practically anything, but the most common case is from events. By convention, observable variable names end with `$`; see <https://angular.io/guide/rx-library#naming-conventions>.
- **Observer:** This is either a callback that is executed whenever the observable it's subscribed to produce a new value or an object with three methods: `next()`, `error()`, and `complete()`, which will be called by the observable when a value is available, when there's an error, and when the stream is ended respectively.
- **Operators:** These are pure functions (similar to `map()`, `filter()`, and so on, from Chapter 5, *Programming Declaratively – A Better Style*) that let you apply transformations to a stream in a declarative way.
- **Pipe:** This is a way to define a pipeline of operators that will be applied to a stream. This is very similar to the `pipeline()` function we developed back in Chapter 8, *Connecting Functions – Pipelining and Composition*.
- **Subscription:** This is the connection to an observable. An observable doesn't do anything until you call the `subscribe()` method, providing an observer.

An interesting way of looking at observables is that they complete the lower row of this table; check it out. You will probably be quite familiar with the *Single* column, but maybe not with the *Multiple one*:

	Single	Multiple
Pull	Function	Iterator
Push	Promise	Observable

How do we interpret this table? The rows distinguish between pull (you call something) and push (you get called), and the columns represent how many values you get: one or many. With these descriptions, we can see the following:

- `function` is called and returns a single value.
- `promise` calls your code (a callback in the `then()` method) also with a single value.
- `iterator` returns a new value each time it's called—at least until the sequence is over.
- `observable` calls your code (provided you `subscribe()` to the observable) for each value in the stream.

Observables and promises can be compared a bit more:

- They are both mostly async in nature, and your callback will be called at an indefinite future time.
- Promises cannot be canceled, but you can `unsubscribe()` from an observable.
- Promises start executing the moment you create them; observables are lazy, and nothing happens until an observer does `subscribe()` to them.

The real power of observables derives from the variety of operators you can use; let's see some of them.

Operators for observables

Basically, operators are just functions: creation operators can be used to create observables out of many different sources, and pipeable operators can be applied to modify a stream, producing a new observable: we'll see many families of these, but for complete lists and descriptions, you should access <https://www.learnrxjs.io/operators/> and <https://rxjs.dev/guide/operators>.



We won't be covering how to install RxJS; see <https://rxjs.dev/guide/installation> for all of the possibilities. In particular, in our examples, meant for a browser, we'll be installing version 6 of RxJS from a CDN, which creates a global `rxjs` variable, similar to jQuery's `$` or LoDash's `_` variables.

Let's begin by creating observables, and then move on to transforming them. For creation, some of the several operators you can use are explained in the following table:

Operator	Usage
<code>ajax</code>	Creates an observable for an Ajax request, for which we'll emit the response that is returned
<code>from</code>	Produces an observable out of an array, an iterable, or a promise
<code>fromEvent</code>	Turns events (for example, mouse clicks) into an observable sequence
<code>interval</code>	Emits values at periodic intervals
<code>of</code>	Generates a sequence out of a given set of value
<code>range</code>	Produces a sequence of values in a range
<code>timer</code>	After an initial delay, emits values periodically

To give a very basic example, the following three observables will all produce a sequence of values from 1 to 10, and we'll be seeing more practical examples a bit later in this chapter:

```
const obs1$ = from([1, 2, 3, 4, 5, 6, 7, 8, 9, 10]);  
const obs2$ = of(1, 2, 3, 4, 5, 6, 7, 8, 9, 10);  
const obs3$ = range(1, 10);
```

The available pipeable operators are way too many for this section, so we'll just go over some families and describe their basic idea with one or two particular mentions. The following table lists the most common families, with their most often used operators:

Family	Description
<i>Combination</i>	<p>These operators allow joining information from several distinct observables, including the following:</p> <ul style="list-style-type: none"> • <code>concat()</code> to put observables in a queue one after the other. • <code>merge()</code> to create a single observable out of many. • <code>pairwise()</code> to emit the previous value and the current one as an array. • <code>startWith()</code> to inject value in an observable.
<i>Conditional</i>	<p>These produce values depending on conditions, and include the following:</p> <ul style="list-style-type: none"> • <code>defaultIfEmpty()</code> emits a value if an observable doesn't emit anything before completing. • <code>every()</code> emits true if all values satisfy a predicate and emits false instead. • <code>iif()</code> subscribes to one of two observables depending on a condition, like the ternary ? operator.
<i>Error handling</i>	<p>These (obviously!) apply to error conditions, and include the following:</p> <ul style="list-style-type: none"> • <code>catchError()</code> to gracefully process an error from an observable. • <code>retry()</code> and <code>retryWhen()</code> to retry an observable sequence (most likely, one linked to HTTP requests.)
<i>Filtering</i>	<p>Probably the most important family, providing many operators to process sequences, by selecting which elements will get processed or dismissed, by applying different types of conditions for your selection. Some of the more common ones include the following:</p> <ul style="list-style-type: none"> • <code>debounce()</code> and <code>debounceTime()</code> to deal with values too close together in time. • <code>distinctUntilChanged()</code> to only emit when the new value is different from the last. • <code>filter()</code> to only emit values that satisfy a given predicate. • <code>find()</code> to emit only the first value that satisfies a condition. • <code>first()</code> and <code>last()</code> to pick only the first or last values of a sequence. • <code>skip()</code> plus <code>skipUntil()</code> and <code>skipWhile()</code> to discard values. • <code>take()</code> and <code>takeLast()</code> to pick a given number of values from the beginning or end of a sequence. • <code>takeUntil()</code> and <code>takeWhile()</code> to pick values and more.

Transforming	<p>The other very commonly used family, which includes operators to transform the values in the sequence. Some of the many possibilities include these:</p> <ul style="list-style-type: none"> • <code>buffer()</code> and <code>bufferTime()</code> to collect values and emit them as an array. • <code>groupBy()</code> to group values together based on some property. • <code>map()</code> to apply a given mapping function to every element in the sequence. • <code>partition()</code> to split an observable into two, based on a given predicate. • <code>pluck()</code> to pick only some attributes from each element. • <code>reduce()</code> to reduce a sequence of values to a single one. • <code>scan()</code> works like <code>reduce()</code>, but emits all intermediate values. • <code>toArray()</code> collects all values and emits them as a single array.
Utilities	<p>A sundry collection of operators with different functions, including the following:</p> <ul style="list-style-type: none"> • <code>tap()</code> to perform a side effect, similar to what we saw in the <i>Tapping into a flow</i> section in Chapter 8, <i>Connecting Functions – Pipelining and Composition</i> • <code>delay()</code> to delay sequence values some time. • <code>finalize()</code> to call a function when an observable completes or produces an error. • <code>repeat()</code> is just like <code>retry()</code> but for normal (that is, non-error) cases. • <code>timeout()</code> to produce an error if no value is produced before a given duration.

Wow, that's a lot of operators! We have excluded many, and you could even write your own, so be sure to look at the documentation.



Understanding operators is made easier with *marbles diagrams*; we won't be using them here, but read <http://reactivex.io/documentation/observable.html> for a basic explanation, and then check out <https://rxmarbles.com/> for many interactive examples of operators and how they function.

Let's finish this section with a couple of examples of the real possibility of application for your own coding.

Detecting multi-clicks

Suppose you decided, for some reason or another, that users should be able to triple-click or four-click on something, and the number of clicks would somehow be meaningful and produce some kind of special result. Browsers do very well detecting single- or double-clicks and let you respond to them, but triple- (or more) clicks aren't available so easily. However, we can make do with a bit of FRP. Let's start with a really basic layout, including a text span that the user should click. The code is given here:

```
<html>
  <head>
    <title>Multiple click example</title>
    <script type="text/javascript" src="rxjs.umd.js"></script>
  </head>
  <body>
    <span id="mySpan">Click this text many times (quickly)</span>
    <script>
      // our code goes here...
    </script>
  </body>
</html>
```

This is as plain as can be; you just get a text onscreen, urging you to multi-click it. See *Figure 11.3*:

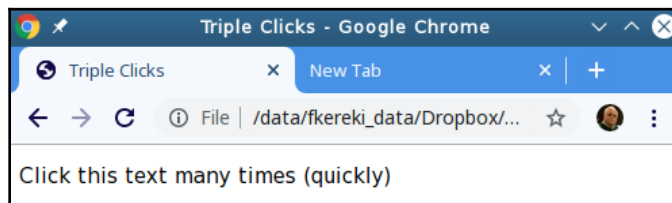


Figure 11.3: A very plain screen, to test detecting triple-clicks

To detect these multi-clicks, we'll need some RxJS functions, so let's start with those:

```
const { fromEvent, pipe } = rxjs;
const { buffer, filter } = rxjs.operators;
```

We will use these functions soon enough. How do we detect triple- (or more) clicks? Let's go straight on to the code given here:

```
const spanClick$ = fromEvent (
  document.getElementById("mySpan"),
  "click"
);
```

```
spanClick$
  .pipe(
    buffer(spanClick$.pipe(debounceTime(250))),
    map(list => list.length),
    filter(x => x >= 3)
  )
  .subscribe(e => {
    console.log(`${e} clicks at ${new Date()}`);
  });

/*
4 clicks at Mon Nov 11 2019 20:19:29 GMT-0300 (Uruguay Standard Time)
3 clicks at Mon Nov 11 2019 20:19:29 GMT-0300 (Uruguay Standard Time)
4 clicks at Mon Nov 11 2019 20:19:31 GMT-0300 (Uruguay Standard Time)
*/
```

The logic is simple:

1. We create an observable with `fromEvent()` listening to mouse clicks on our span.
2. Now, a tricky point: we use `buffer()` to join together many events, which come from applying `debounceTime()` to the sequence of clicks—so all clicks that happen within an interval of 250 milliseconds will get grouped into a single array.
3. We then apply `map()` to transform each array of clicks into just its length—after all, we care about how many clicks there were, and not their specific details.
4. We finish by filtering out values under 3, so only longer sequences of clicks will be processed.
5. The subscription, in this case, just logs the clicks, but in your application, it should do something more relevant.

If you wanted, you could detect multi-clicks by hand, writing your own code; see question 11.3 in the *Questions* section. To finish, let's go with an even longer example and do some typeahead searches invoking some external API.

Providing typeahead searches

Let's do another web example: typeahead searches. The usual setup is that there is some textbox, the user types in it, and the web page queries some API to provide ways of completing the search. The important thing is when and how to do the search and trying to avoid unnecessary calls to the backend server whenever possible. A (totally basic) HTML page could be as follows, and see *Figure 11.4* later in this section:

```
<html>
  <head>
    <title>Cities search</title>
    <script type="text/javascript" src="rxjs.umd.js"></script>
  </head>
  <body>
    Find cities:
    <input type="text" id="myText" />
    <br />
    <h4>Some cities...</h4>
    <div id="myResults"></div>
    <script>
      // typeahead code goes here...
    </script>
  </body>
</html>
```

We have a single textbox in which the user will type and an area below that in which we'll show whatever the API provides. We'll be using the GeoDB Cities API (see <http://geodb-cities-api.wirefreethought.com/>), which provides many search options, but we'll just use it to search for cities starting with whatever the user has typed. Just to get it out of our way, let's see the `getCitiesOrNull()` function, which will return a promise for search results (if something was typed in) or a promise that resolves to null (no cities, if nothing was typed in). The results of this promise will be used to fill the `myResults` division on the page. Let's see how this works out in code:

```
const getCitiesOrNull = text => {
  if (text) {
    const citySearchUrl =
      `http://geodb-free-service.wirefreethought.com/v1/geo/cities?` +
      `hateoasMode=false&` +
      `sort=-population&` +
      `namePrefix=${encodeURIComponent(text)}`;
    return fetch(citySearchUrl);
  } else {
    return Promise.resolve(null);
  }
};
```


The code is simple: if some text was provided, we generate the URL for the cities' search and use `fetch()` to get the API data. With this done, let's see how to generate the needed observable. We will need some RxJS functions, so first, let's have some definitions:

```
const { fromEvent, pipe } = rxjs;
const {
  debounceTime,
  distinctUntilChanged,
  filter,
  map,
  reduce,
  switchMap
} = rxjs.operators;
```

We will be using all of these functions later. Now, we can write the code to do the typeahead:

```
const textInput$ = fromEvent(
  document.getElementById("myText"),
  "input"
).pipe(
  map(e => e.target.value),
  debounceTime(200),
  filter(w => w.length === 0 || w.length > 3),
  distinctUntilChanged(),
  switchMap(w => getCitiesOrNull(w))
);
```

This requires going step by step:

1. We use the `fromEvent()` constructor to observe input events (every time the user types something) from the `myText` input field.
2. We use `map()` to get the event's target value, the complete text of the input field.
3. We use `debounceTime(200)` so the observable won't emit until the user has been 0.2 seconds (200 milliseconds) without typing—what's the use of calling the API if the user isn't done with their query?
4. We then use `filter()` to discard the input if it was only one, two, or three characters long because that's not good enough for our search. We accept empty strings (so we'll empty the results area) and strings four or more characters long.
5. Then, we use `distinctUntilChanged()` so if the search string is the same as before (the user possibly added a character but quickly backspaced, deleting it), nothing will be emitted.
6. We finally change use `switchMap()` to cancel the previous subscription to the observable and create a new one using `getCitiesOrNull()`.

How do we use this? We subscribe to the observable and when we get results, we use them to display values. A possible sample code follows:

```
textInput$.subscribe(async fetchResult => {
  domElem = document.getElementById("myResults");

  if (fetchResult !== null) {
    result = await fetchResult.json();
    domElem.innerHTML = result.data
      .map(x => `${x.city}, ${x.region}, ${x.country}`)
      .join("<br />");
  } else {
    domElem.innerHTML = "";
  }
});
```

An important point: the promise is resolved, and the final value of the sequence is hence whatever the promise produced. If the result wasn't null, we get an array of cities, and we use `map()` and `join()` to produce the (very basic!) HTML output; otherwise, we just empty the results area.

Let's try it out. If you start typing, nothing will happen while you haven't reached four characters and pause a bit; see *Figure 11.4*, as follows:

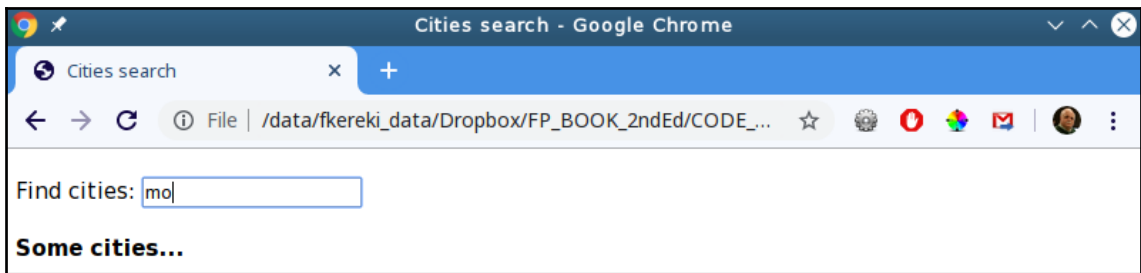


Figure 11.4: Our search for cities doesn't trigger for less than four characters

When you reach four characters and pause a bit, the observable will emit an event, and we'll do a first search: in this case, for cities with names starting with MONT. See *Figure 11.5*, as follows:

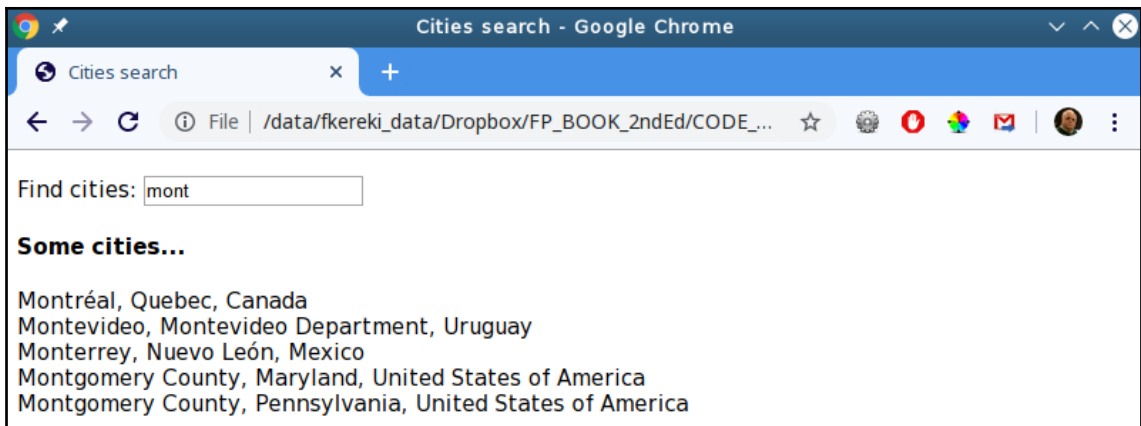


Figure 11.5: After reaching four characters, searches will be fired

Finally, as you add more characters, new API calls will be done, refining the search; see *Figure 11.6*, as follows:

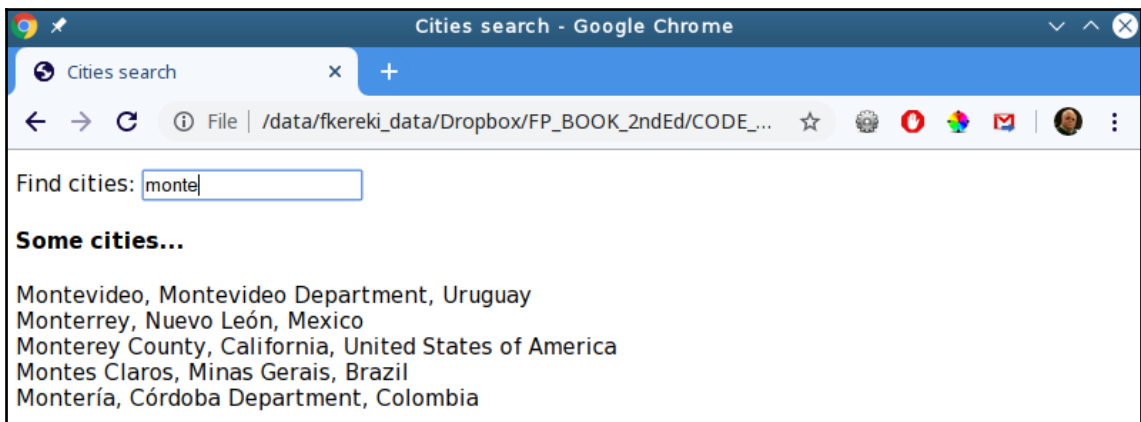


Figure 11.6: Further characters are used to refine the search

What can we learn from these examples? Using observables for events lets us achieve a good separation of concerns as to event production and event consumption, and the declarative style of the stream process makes the data flow clearer. You could even note that the HTML code itself has no reference to click methods or anything like that; the complete code is separate from that.

We have now seen most of the interesting patterns; let's finish with some other ones, which may or may not be exactly equivalent to their classic OOP partners.

Other patterns

Let's end this section by glancing at some other patterns, where the equivalence may or may not be so good:

- **Currying and Partial Application** (which we saw in [Chapter 7, Transforming Functions – Currying and Partial Application](#)): This can be seen as approximately equivalent to a *Factory* for functions. Given a general function, you can produce specialized cases by fixing one or more arguments, and this is, in essence, what a Factory does, of course, speaking about functions and not objects.
- **Declarative functions** (such as `map()` or `reduce()`): They can be considered an application of the *Iterator* pattern. The traversal of the container's elements is decoupled from the container itself. You might also provide different `map()` methods for different objects, so you could traverse all kinds of data structures.
- **Persistent data structures**: As mentioned in [Chapter 10, Ensuring Purity – Immutability](#), they allow for the implementation of the *Memento* pattern. The central idea is, given an object, to be able to go back to a previous state. As we saw, each updated version of a data structure doesn't impact on the previous one(s), so you could easily add a mechanism to provide an earlier state and *roll back* to it.
- A **Chain of Responsibility** pattern: In this pattern, there is a potentially variable number of *request processors* and a stream of requests to be handled, which may be implemented using `find()` to determine which is the processor that will handle the request (the desired one is the first in the list that accepts the request) and then simply doing the required process.

Remember the warning at the beginning: with these patterns, the match with FP techniques may not be so perfect as with others that we have previously seen, but the idea was to show that there are some common FP patterns that can be applied, and it will produce the same results as the OOP solutions, despite having different implementations.

Now, after having seen several OOP equivalent patterns, let's move on to more specific FP ones.

Functional design patterns

After having seen several OOP design patterns, it may seem a cheat to say that there's no approved, official, or even remotely generally accepted similar list of patterns for FP. There are, however, several problems for which there are standard FP solutions, which can be considered design patterns on their own, and we have already covered most of them in this book.

What are candidates for a possible list of patterns? Let's attempt to prepare one—but remember, it's just a personal view. Also, I'll admit that I'm not trying to mimic the usual style of pattern definition; I'll just be mentioning a general problem and refer to the way FP in JS can solve it, and I won't be aiming for nice, short, memorable names for the patterns either:

- **Processing collections using filter/map/reduce:** Whenever you have to process a data collection, using declarative higher-order functions, such as `filter()`, `map()`, and `reduce()`, as we saw in this chapter and previously in Chapter 5, *Programming Declaratively – A Better Style*, is a way to remove complexity from the problem (the usual `MapReduce` web framework is an extension of this concept, which allows for distributed processing among several servers, even if the implementation and details aren't exactly the same). Instead of performing looping and processing as a single step, you should think about the problem as a sequence of steps, applied in order, doing transformations until obtaining the final, desired result.



JS also includes *iterators*, that is, another way of looping through a collection. Using *iterators* isn't particularly functional, but you may want to look at them since they may be able to simplify some situations. Read more at https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Iteration_protocols.

- **Lazy evaluation with thunks:** The idea of *lazy evaluation* is not doing any calculations until they are actually needed. In some programming languages, this is built in. However, in JavaScript (and in most imperative languages as well), *eager evaluation* is applied, in which an expression is evaluated as soon as it is bound to some variable. (Another way of saying this is that JavaScript is a *strict programming language*, with a *strict paradigm*, which only allows calling a function if all of its parameters have been completely evaluated.) This sort of evaluation is required when you need to specify the order of evaluation with precision, mainly because such evaluations may have side effects.

In FP, which is rather more declarative and pure, you can delay such evaluation with *thunks* (which we used in the *Trampolines and thunks* section of Chapter 9, *Designing Functions – Recursion*) by passing a thunk that will calculate the needed value only when it's needed, but not earlier.



You may also want to look at JavaScript *generators*, which is another way of delaying evaluation, though not particularly related to FP at all. Read more about them at https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Generator. The combination of generators and promises is called an async function, which may be of interest to you; refer to https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Statements/async_function.

- **Persistent data structures for immutability:** Having immutable data structures, as we saw in Chapter 10, *Ensuring Purity – Immutability*, is mandatory when working with certain frameworks, and in general, it is recommended because it helps to reason about a program or debugging it. (Earlier in this chapter, we also mentioned how the *Memento* OOP pattern can be implemented in this fashion). Whenever you have to represent structured data, the FP solution of using a persistent data structure helps in many ways.
- **Wrapped values for checks and operations:** If you directly work with variables or data structures, you may modify them at will (possibly violating any restrictions) or you may need to do many checks before using them (such as verifying that a value is not null before trying to access the corresponding object). The idea of this pattern is to wrap a value within an object or function, so direct manipulation won't be possible, and checks can be managed more functionally. We'll be referring to more of this in Chapter 12, *Building Better Containers – Functional Data Types*.

As we have said, the power of FP is such that, instead of having a couple of dozen standard design patterns (and that's only in the GoF book; if you read other texts, the list grows!), there isn't yet a standard or acknowledged list of functional patterns.

Summary

In this chapter, we have made a bridge from the OO way of thinking and the usual patterns that we use when coding that way, to the FP style, by showing how we can solve the same basic problems (but rather more easily) than with classes and objects. We have seen several common design patterns, and we've seen that the same concepts apply in FP, even if implementations may vary, so now you have a way to apply those well-known solution structures to your JavaScript coding.

In Chapter 12, *Building Better Containers – Functional Data Types*, we will be working with a *potpourri* of functional programming concepts, which will give you even more ideas about tools you can use. I promised that this book wouldn't become deeply theoretical, but rather more practical, and we'll try to keep it this way, even if some of the presented concepts may seem abstruse or remote.

Questions

11.1. Decorating methods, the future way: In Chapter 6, *Producing Functions – Higher-Order Functions*, we wrote a decorator to enable logging for any function. Currently, method decorators are being considered for upcoming versions of JavaScript: refer to <https://tc39.github.io/proposal-decorators/> for that (Draft 2 means that inclusion of this feature in the standard is likely, although there may be some additions or small changes). Study the following draft and take a look at what makes the next code tick:

```
const logging = (target, name, descriptor) => {
  const savedMethod = descriptor.value;
  descriptor.value = function(...args) {
    console.log(`entering ${name}: ${args}`);
    try {
      const valueToReturn = savedMethod.bind(this)(...args);
      console.log(`exiting ${name}: ${valueToReturn}`);
      return valueToReturn;
    } catch (thrownError) {
      console.log(`exiting ${name}: threw ${thrownError}`);
      throw thrownError;
    }
  };
  return descriptor;
};
```

A working example would be as follows:

```
class SumThree {
  constructor(z) {
    this.z = z;
  }
  @logging
  sum(x, y) {
    return x + y + this.z;
  }
}

new SumThree(100).sum(20, 8);
// entering sum: 20,8
// exiting sum: 128
```

Following are some questions about the code for the `logging()` decorator:

- Do you see the need for the `savedMethod` variable?
- Why do we use `function()` when assigning a new `descriptor.value`, instead of an arrow function?
- Can you understand why `.bind()` is used?
- What is `descriptor`?

11.2. Decorator with mixins: Back in the *Questions* section of Chapter 1, *Becoming Functional – Several Questions*, we saw that classes are first-class objects. Taking advantage of this, complete the following `addBar()` function, which will add some mixins to the `Foo` class so that the code will run as shown. The created `fooBar` object should have two attributes (`fooValue` and `barValue`) and two methods (`doSomething()` and `doSomethingElse()`) that simply show some text and properties, as shown here:

```
class Foo {
  constructor(fooValue) {
    this.fooValue = fooValue;
  }
  doSomething() {
    console.log("something: foo... ", this.fooValue);
  }
}

var addBar = BaseClass =>
  /*
  your code goes here
  */
;
```



```
var fooBar = new (addBar(Foo))(22, 9);
fooBar.doSomething(); // something: foo... 22
fooBar.somethingElse(); // something else: bar... 9
console.log(Object.keys(fooBar)); // ["fooValue", "barValue"]
```

Could you include a third mixin, `addBazAndQux()`, so that `addBazAndQux(addBar(Foo))` would add even more attributes and methods to `Foo`?

11.3. Multi-clicking by hand: Can you write your own multi-click detection code, which should work exactly as our example?

12

Building Better Containers - Functional Data Types

In Chapter 11, *Implementing Design Patterns – The Functional Way*, we went over how to use functions to achieve different results. In this chapter, we will look at data types from a functional point of view. We'll be considering how we can implement our own data types, along with several features that can help us compose operations or ensure their purity so that our FP coding will become simpler and shorter.

We'll be touching on several themes:

- **Data types** from a functional point of view. Even though JavaScript is not a typed language, a better understanding of types and functions is needed.
- **Containers**, including *functors* and the mystifying *monads*, to structure data flow.
- **Functions as structures**, in which we'll see yet another way of using functions to represent data types, with immutability thrown in as an extra.

With that, let's get started!

Specifying data types

Even though JavaScript is a dynamic language, without static or explicit typing declarations and controls, it doesn't mean that you can simply ignore types. Even if the language doesn't allow you to specify the types of your variables or functions, you still work—even if only in your head—with types. Now, let's learn how we can specify types. When it comes to specifying types, we have some advantages, as follows:

- Even if you don't have compile-time data type checking, there are several tools, such as Facebook's *flow* static type checker or Microsoft's *TypeScript* language, that let you deal with it.

- It will help if you plan to move on from JavaScript to a more functional language such as *Elm*.
- It serves as documentation that lets future developers understand what type of arguments they have to pass to the function and what type it will return. All the functions in the Ramda library are documented in this way.
- It will also help with the functional data structures later in this section, where we will examine a way of dealing with structures, similar in some aspects to what you do in fully functional languages such as Haskell.



If you want to learn more about the tools that I cited, visit <https://flow.org/> for Flow, <https://www.typescriptlang.org/> for TypeScript, and <http://elm-lang.org/> for Elm. If you want to know more about type checks, the corresponding web pages are <https://flow.org/en/docs/types/functions/>, <https://www.typescriptlang.org/docs/handbook/functions.html>, and <https://flow.org/en/docs/types/functions/>.

Whenever you read or work with a function, you will have to reason about types, think about the possible operations on this or that variable or attribute, and so on. Having type declarations will help. Due to this, we will start considering how we can define the types of functions and their parameters. After that, we will consider other type definitions.

Signatures for functions

The specification of a function's arguments and the result is given by a *signature*. Type signatures are based on a *type system* called **Hindley-Milner**, which influenced several (mostly functional) languages, including Haskell, though the notation has changed from that of the original paper. This system can even deduce types that are not directly given; tools such as TypeScript or Flow also do this, so developers don't need to specify *all* types. Instead of going for a dry, formal explanation about the rules for writing correct signatures, let's work by examples. We only need to know the following:

- We will be writing the type declaration as a comment.
- The function name is written first, and then `::`, which can be read as *is of type* or *has type*.
- Optional constraints may follow, with a double (*fat*) arrow \Rightarrow (or \Rightarrow in basic ASCII fashion, if you cannot key in the arrow) afterward.
- The input type of the function follows, with a \rightarrow (or $->$, depending on your keyboard).
- The result type of the function comes last.



Note that instead of this vanilla JavaScript style, Flow and TypeScript have their own syntax for specifying type signatures.

Now, we can begin with some examples. Let's define the type for a simple function that just capitalizes a word and do the same for the `Math.random` function:

```
// firstToUpper :: String → String
const firstToUpper = s => s[0].toUpperCase() + s.substr(1).toLowerCase();

// Math.random :: () → Number
```

These are simple cases—only take the signatures into account here; we are not interested in the actual functions. The first function receives a string as an argument and returns a new string. The second one receives no arguments (the empty parentheses show this) and returns a floating-point number. The arrows denote functions. So, we can read the first signature as `firstToUpper` is a function of the type that receives a string and returns a string and we can speak similarly about the maligned (impurity-wise) `Math.random()` function, with the only difference being that it doesn't receive arguments.

We've already looked at functions with zero or one parameter, but what about functions with more than one? There are two answers to this. If we are working in a strict functional style, we would always be doing currying (as we saw in [Chapter 7, Transforming Functions – Currying and Partial Application](#)), so all the functions would be unary. The other solution is enclosing a list of argument types in parentheses. We can see both of these solutions in the following code:

```
// sum3C :: Number → Number → Number → Number
const sum3C = curry((a, b, c) => a + b + c);

// sum3 :: (Number, Number, Number) → Number
const sum3 = (a, b, c) => a + b + c;
```

Remember that `sum3C` is actually `a => b => c => a + b + c`; this explains the first signature, which can also be read as follows:

```
// sum3C :: Number → (Number → (Number → (Number)))
```

After you provide the first argument to the function, you are left with a new function, which also expects an argument, and returns a third function, which, when given an argument, will produce the final result. We won't be using parentheses because we'll always assume this grouping from right to left.

Now, what about higher-order functions, which receive functions as arguments? The `map()` function poses a problem: it works with arrays of any type. Also, the mapping function can produce any type of result. For these cases, we can specify *generic types*, which are identified by lowercase letters: these generic types can stand for any possible type. For arrays themselves, we use brackets. So, we would have the following:

```
// map :: [a] → (a → b) → [b]
const map = curry((arr, fn) => arr.map(fn));
```

It's perfectly valid to have `a` and `b` represent the same type, as in a mapping that's applied to an array of numbers, which produces another array of numbers. The point is that, in principle, `a` and `b` may stand for different types, and that's what we described previously. Also, note that if we weren't currying, the signature would have been `([a], (a → b)) → [b]`, showing a function that receives two arguments (an array of elements of type `a` and a function that maps from type `a` to type `b`) and produces an array of elements of type `b` as the result. Given this, we can write the following in a similar fashion:

```
// filter :: [a] → (a → Boolean) → [a]
const filter = curry((arr, fn) => arr.filter(fn));
```

And now the big one: what's the signature for `reduce()`? Be sure to read it carefully and see if you can work out why it's written that way. You may prefer thinking about the second part of the signature as if it were `((b, a) → b)`:

```
// reduce :: [a] → (b → a → b) → b → b
const reduce = curry((arr, fn, acc) => arr.reduce(fn, acc));
```

Finally, if you are defining a method instead of a function, you use a squiggly arrow such as `~>`:

```
// String.repeat :: String ~> Number → String
```

So far, we have defined data types for functions, but we aren't done with this subject just yet. Let's consider some other cases.

Other data type options

What else are we missing? Let's look at some other options that you might use. *Product types* are a set of values that are always together and are commonly used with objects. For tuples (that is, an array with a fixed number of elements of (probably) different types), we can write something like the following:

```
// getWeekAndDay :: String → (Number × String)
const getWeekAndDay = yyyy_mm_dd =>
```

```
/* ... */
return [weekNumber, dayOfWeekName];
```

For objects, we can go with a definition very similar to what JavaScript already uses. Let's imagine we have a `getPerson()` function that receives an ID and returns an object with data about a person:

```
// getPerson :: Number → { id:Number × name:String }
const getPerson = personId =>
  /* ... */
  return { id:personId, name:personName }
```

Sum types (also known as *union types*) are defined as a list of possible values. For example, our `getField()` function from Chapter 6, *Producing Functions – Higher-Order Functions*, either returns the value of an attribute or it returns undefined. For this, we can write the following signature:

```
// getField :: String → attr → a | undefined
const getField = attr => obj => obj[attr];
```

We could also define a type (union or otherwise) and use it in further definitions. For instance, the data types that can be directly compared and sorted are numbers, strings, and Booleans, so we could write the following definitions:

```
// Sortable :: Number | String | Boolean
```

Afterward, we could specify that a comparison function can be defined in terms of the `Sortable` type, but be careful: there's a hidden problem here!

```
// compareFunction :: (Sortable, Sortable) → Number
```



Actually, this definition isn't very precise because you can compare any type, even if it doesn't make much sense. However, bear with me for the sake of this example!

If you want to refresh your memory about sorting and comparison functions, see https://developer.mozilla.org/en/docs/Web/JavaScript/Reference/Global_Objects/Array/sort.

The previous definition would allow us to write a function that received, say, a number and a Boolean: it doesn't say that both types should be the same. However, there's a way out. If you have constraints for some data types, you can express them before the actual signature, using a *fat* arrow, as shown in the following code:

```
// compareFunction :: Sortable a ⇒ (a, a) → Number
```

Now, the definition is correct because all occurrences of the same type (denoted by the same letter, in this case, `a`) must be exactly the same. An alternative, but one that requires much more typing, would have been writing all three possibilities with a union:

```
// compareFunction ::  
//   ((Number, Number) | (String, String) | (Boolean, Boolean)) → Number
```

So far, we have been using the standard type definitions. However, when we work with JavaScript, we have to consider some other possibilities, such as functions with optional parameters, or even with an undetermined number of parameters. We can use `...` to stand for any number of arguments and add `?` to represent an optional type, as follows:

```
// unary :: ((b, ...) → a) → (b → a)  
const unary = fn => (...args) => fn(args[0]);
```

The `unary()` higher-order function that we defined in the same chapter we cited previously took any function as a parameter and returned a unary function as its result. We can show that the original function can receive any number of arguments but that the result used only the first of them. The data type definition for this would be as follows:

```
// parseInt :: (String, Number?) -> Number
```

The standard `parseInt()` function provides an example of optional arguments: though it's highly recommended that you don't omit the second parameter (the base radix), you can, in fact, skip it.



Check out <https://github.com/fantasyland/fantasy-land/> and <https://sanctuary.js.org/#types> for a more formal definition and description of types, as applied to JavaScript.

From now on, throughout this chapter, we will be adding signatures to methods and functions. This will not only be so that you can get accustomed to them but because, when we start delving into more complex containers, it will help you understand what we are dealing with: some cases can be hard to understand!

Building containers

Back in Chapter 5, *Programming Declaratively – A Better Style*, and later, in Chapter 8, *Connecting Functions – Pipelining and Composition*, we saw that the ability to be able to apply a mapping to all the elements of an array—and, even better, being able to chain a sequence of similar operations—was a good way to produce better, more understandable code.

However, there is a problem: the `map()` method (or the equivalent, *demethodized* one, which we looked at in [Chapter 6, Producing Functions – Higher-Order Functions](#)), is only available for arrays, and we might want to be able to apply mappings and chaining to other data types. So, what can we do?

Let's consider different ways of doing this, which will give us several new tools for better functional coding. Basically, there are only two possible ways of solving this: we can either add new methods to existing types (though that will be limited because we can only apply that to basic JavaScript types) or we can wrap types in some type of container, which will allow mapping and chaining.

Let's start by extending current types before moving on to using wrappers, which will lead us into the deep functional territory, with entities such as functors and monads.

Extending current data types

If we want to add mapping to basic JavaScript data types, we need to start by considering our options:

- With `null`, `undefined`, and `Symbol`, applying maps doesn't sound too interesting.
- With the `Boolean`, `Number`, and `String` data types, we have some interesting possibilities, so we can examine some of those.
- Applying mapping to an object would be trivial: we just have to add a `map()` method, which must return a new object.
- Finally, despite not being basic data types, we could also consider special cases, such as dates or functions, to which we could also add `map()` methods.



As in the rest of this book, we are sticking to plain JavaScript, but you should look into libraries such as `Lodash`, `Underscore`, or `Ramda`, which already provide functionalities similar to the ones we are developing here.

A key point to consider in all these mapping operations should be that the returned value is of exactly the same type as the original one: when we use `Array.map()`, the result is also an array, and similar considerations must apply to any other `map()` method implementations (you could observe that the resulting array may have different element types to the original one, but it is still an array).

What could we do with a Boolean? First, let's accept that Booleans are not containers, so they don't really behave in the same way as an array: trivially, a Boolean can only have a Boolean value, while an array may contain any type of element. However, accepting that difference, we can extend `Boolean.prototype` (though, as I've already mentioned, that's not usually recommended) by adding a new `map()` method to it and making sure that whatever the mapping function returns is turned into a new Boolean value. For the latter, the solution will be similar to the following:

```
// Boolean.map :: Boolean ~> (Boolean → a) → Boolean
Boolean.prototype.map = function(fn) {
  return !!fn(this);
};
```

The `!!` operator forces the result to be a Boolean: `Boolean(fn(this))` could also have been used. This kind of solution can also be applied to numbers and strings, as shown in the following code:

```
// Number.map :: Number ~> (Number → a) → Number
Number.prototype.map = function(fn) {
  return Number(fn(this));
};

// String.map :: String ~> (String → a) → String
String.prototype.map = function(fn) {
  return String(fn(this));
};
```

As with Boolean values, we are forcing the results of the mapping operations to the correct data types.

Finally, if we wanted to apply mappings to a function, what would that mean? Mapping a function should produce a function. The logical interpretation for `f.map(g)` would be applying `f()`, and then applying `g()` to the result. So, `f.map(g)` should be the same as writing `x => g(f(x))` or, equivalently, `pipe(f, g)`. The definition is more complex than it was for the previous examples, so study it carefully:

```
// Function.map :: (a → b) ~> (b → c) → (a → c)
Function.prototype.map = function(fn) {
  return (...args) => fn(this(...args));
};
```

Verifying that this works is simple, and the following code is an easy example of how to do this. The `by10()` mapping function is applied to the result of calculating `plus1(3)`, so the result is 40:

```
const plus1 = x => x + 1;
const by10 = y => 10 * y;

console.log(plus1.map(by10)(3));
// 40: first add 1 to 3, then multiply by 10
```

With this, we are done talking about what we can achieve with basic JavaScript types, but we need a more general solution if we want to apply this to other data types. We'd like to be able to apply mapping to any kind of values, and for that, we'll need to create a container. We'll do this in the next section.

Containers and functors

What we did in the previous section does work and can be used with no problems. However, we would like to consider a more general solution that we can apply to any data type. Since not all things in JavaScript provide the desired `map()` method, we will have to either extend the type (as we did in the previous section) or apply a design pattern that we considered in [Chapter 11, Implementing Design Patterns – The Functional Way](#): wrapping our data types with a wrapper that will provide the required `map()` operations.

In particular, we will do the following:

- Start by seeing how to build a basic container, wrapping a value
- Convert the container into something more powerful—a functor
- Study how to deal with missing values using a special functor, `Maybe`

Wrapping a value – a basic container

Let's pause for a minute and consider what we need from this wrapper. There are two basic requirements:

- We must have a `map()` method.
- We need a simple way to wrap a value.

To get started, let's create a basic container. Any object containing just a value would do, but we want some additions, so our object won't be that trivial; we'll explain the differences after the code:

```
const VALUE = Symbol("Value");

class Container {
  constructor(x) {
    this[VALUE] = x;
  }

  map(fn) {
    return fn(this[VALUE]);
  }
}
```

Some basic considerations that we need to keep in mind are as follows:

- We want to be able to store some value in a container, so the constructor takes care of that.
- Using a `Symbol` helps hide the field: the property key won't show up in `Object.keys()` or in `for...in` or `for...of` loops, making them more meddle-proof.
- We need to be able to `map()`, so a method is provided for that.



If you haven't worked with JavaScript symbols, possibly the least known of its primitive data types, you might want to check out <https://developer.mozilla.org/en-US/docs/Glossary/symbol>.

Our basic barebones container is ready, but we can also add some other methods for convenience, as follows:

- To get the value of a container, we could use `map(x => x)`, but that won't work with more complex containers, so we'll add a `valueOf()` method to get the contained value.
- Being able to list a container can certainly help with debugging. The `toString()` method will come in handy for this.
- Because we don't need to write `new Container()` all the time, we can add a static `of()` method to do the same job.



Working with classes to represent containers (and later, functors and monads) when living in a functional programming world may seem like heresy or sin... but remember that we don't want to be dogmatic, and `class` and `extends` simplify our coding. Similarly, it could be argued that you must never take a value out of the container—but using a `valueOf()` method is sometimes too handy, so we won't be that restrictive.

By taking all of this into account, our container is as follows:

```
class Container {
    //
    // everything as above
    //

    static of(x) {
        return new Container(x);
    }

    toString() {
        return `${this.constructor.name} (${this[VALUE]})`;
    }

    valueOf() {
        return this[VALUE];
    }
}
```

Now, we can use this container to store a value, and we can use `map()` to apply any function to that value, but this isn't very different from what we could do with a variable! Let's enhance this a bit.

Enhancing our container – functors

We want to have wrapped values, so what exactly should return the `map()` method? If we want to be able to chain operations, then the only logical answer is that it should return a new wrapped object. In true functional style, when we apply a mapping to a wrapped value, the result will be another wrapped value that we can keep working on.



Instead of `map()`, this operation is sometimes called `fmap()`, standing for *functorial map*. The rationale for the name change was to avoid expanding the meaning of `map()`. However, since we are working in a language that supports reusing the name, we can keep it.

We can extend our `Container` class to implement this change and get ourselves an enhanced container: a *functor*. The `of()` and `map()` methods will require a small change. For this, we'll be creating a new class, as shown in the following code:

```
class Functor extends Container {
  static of(x) {
    return new Functor(x);
  }

  map(fn) {
    return Functor.of(fn(this[VALUE]));
  }
}
```

Here, the `of()` method produces a `Functor` object, and so does the `map()` method. With these changes, we have just defined what a `Functor` is in category theory! (Or, if you want to get really technical, a *Pointed Functor* because of the `of()` method – but let's keep it simple.) We won't go into the theoretical details, but roughly speaking, a functor is some kind of container that allows us to apply `map()` to its contents, producing a new container of the same type, and if this sounds familiar, it's because you already know a functor: arrays! When you apply `map()` to an array, the result is a new array containing transformed (mapped) values.



There are more requirements for functors. First, the contained values may be polymorphic (of any type), just like arrays. Second, a function must exist whose mapping produces the same contained value—`x => x` does this for us. Finally, applying two consecutive mappings must produce the same result as applying their composition. This means that `container.map(f).map(g)` must be the same as `container.map(compose(g, f))`.

Let's pause for a moment and consider the signatures for our function and methods:

```
of :: Functor f => a -> f a

Functor.toString :: Functor f => f a ~> String

Functor.valueOf :: Functor f => f a ~> a

Functor.map :: Functor f => f a ~> (a -> b) -> f a -> f b
```

The first function, `of()`, is the simplest: given a value of any type, it produces a `Functor` of that type. The next two are also rather simple to understand: given a `Functor`, `toString()` always returns a string (no surprise there!) and if the functor-contained value is of a given type, `valueOf()` produces a result of that same type. The third one, `map()`, is more interesting. Given a function that takes an argument of type `a` and produces a result of type `b`, applying it to a functor that contains a value of type `a` produces a functor containing a value of type `b`—this is exactly what we described previously.

As is, functors are not allowed or expected to produce side effects, throw exceptions, or exhibit any other behavior outside of producing a containered result. Their main usage is to provide us with a way to manipulate a value, apply operations to it, compose results, and so on, without changing the original—in this sense, we are once again coming back to immutability.



You could also compare functors to promises, at least in one aspect. With functors, instead of acting on its value directly, you have to apply a function with `map()`. In promises, you do exactly the same, but using `then()` instead! In fact, there are more analogies, as we'll be seeing soon.

However, you could well say that this isn't enough since, in normal programming, it's quite usual to have to deal with exceptions, undefined or null values, and so on. So, let's start by looking at more examples of functors. After that, we'll enter the realm of monads so that we can look at even more sophisticated kinds of processing. Let's experiment a bit!

Dealing with missing values with Maybe

A common problem in programming is dealing with missing values. There are many possible causes for this situation: a web service Ajax call may have returned an empty result, a dataset could be empty, an optional attribute might be missing from an object, and so on. Dealing with this kind of situation, in a normal imperative fashion, requires adding `if` statements or ternary operators everywhere to catch the possible missing value in order to avoid a certain runtime error. We can do a bit better by implementing a `Maybe` functor to represent a value that may (or may *not*) be present! We will use two classes, `Just` (as in *just some value*) and `Nothing`, both of which are functors themselves. The `Nothing` functor is particularly simple, with trivial methods:

```
class Nothing extends Functor {  
  isNothing() {  
    return true;  
  }  
  
  toString() {
```

```
        return "Nothing()";
    }

    map(fn) {
        return this;
    }
}
```

The `isNothing()` method returns `true`, `toString()` returns a constant text, and `map()` always returns itself, no matter what function it's given. Moving forward, the `Just` functor is also a basic one, with the added `isNothing()` method (which always returns `true`, since a `Just` object isn't a `Nothing`), and a `map()` method that now returns a `Maybe`:

```
class Just extends Functor {
    isNothing() {
        return false;
    }

    map(fn) {
        return Maybe.of(fn(this[VALUE]));
    }
}
```

Finally, our `Maybe` class packs the logic that's needed to construct either a `Nothing` or a `Just`. If it receives an undefined or null value, a `Nothing` will be constructed, and in other cases, a `Just` will be the result. The `of()` method has exactly the same behavior:

```
class Maybe extends Functor {
    constructor(x) {
        return x === undefined || x === null
            ? new Nothing()
            : new Just(x);
    }

    static of(x) {
        return new Maybe(x);
    }
}
```

We can quickly verify that this works by trying to apply an operation to either a valid value or a missing one. Let's look at two examples of this:

```
const plus1 = x => x + 1;

Maybe.of(2209).map(plus1).map(plus1).toString(); // "Just (2211)"

Maybe.of(null).map(plus1).map(plus1).toString(); // "Nothing()"
```

When we applied `plus1()` to `Maybe.of(2209)`, everything worked fine, and we ended up with a `Just(2011)` value. On the other hand, when we applied the same sequence of operations to a `Maybe.of(null)` value, the end result was a `Nothing`, but there were no errors, even if we tried to do math with a null value. A `Maybe` functor can deal with mapping a missing value by just skipping the operation and returning a wrapped `null` value instead. This means that this functor is including an abstracted check, which won't let an error happen.



Later in this chapter, we'll see that `Maybe` can actually be a monad instead of a functor, and we'll also examine more examples of monads.

Let's look at a more realistic example of its usage.

Dealing with varying API results

Suppose we are writing a small server-side service in Node to get the alerts for a city and produce a not-very-fashionable HTML `<table>` with them, supposedly to be part of some server side-produced web page. (Yes, I know you should try to avoid tables in your pages, but what I want here is a short example of HTML generation, and actual results aren't really important.) If we used the *Dark Sky* API (see <https://darksky.net/> for more on this API and how to register with it) to get the alarms, our code would be something like this; all quite normal. Note the callback in case of an error; you'll see why in the following code:

```
const request = require("superagent");

const getAlerts = (lat, long, callback) => {
  const SERVER = "https://api.darksky.net/forecast";
  const UNITS = "units=si";
  const EXCLUSIONS = "exclude=minutely, hourly, daily, flags";
  const API_KEY = "you.need.to.get.your.own.api.key";
  request
    .get(`${SERVER}/${API_KEY}/${lat}, ${long}?${UNITS}&${EXCLUSIONS}`)
    .end(function(err, res) {
      if (err) {
        callback({});
      } else {
        callback(JSON.parse(res.text));
      }
    });
};
```


The (heavily edited and reduced in size) output of such a call might be something like this:

```
{
  latitude: 29.76,
  longitude: -95.37,
  timezone: "America/Chicago",
  offset: -5,
  currently: {
    time: 1503660334,
    summary: "Drizzle",
    icon: "rain",
    temperature: 24.97,
    .
    .
    .
    uvIndex: 0
  },
  alerts: [
    {
      title: "Tropical Storm Warning",
      regions: ["Harris"],
      severity: "warning",
      time: 1503653400,
      expires: 1503682200,
      description:
        "TROPICAL STORM WARNING REMAINS IN EFFECT... WIND - LATEST LOCAL
        FORECAST: Below tropical storm force wind ... CURRENT THREAT TO LIFE AND
        PROPERTY: Moderate ... Locations could realize roofs peeled off buildings,
        chimneys toppled, mobile homes pushed off foundations or overturned ...",
      uri:
        "https://alerts.weather.gov/cap/wwacapget.php?x=TX125862DD4F88.TropicalStor
        mWarning.125862DE8808TX.HGXTCVHGX.73ee697556fc6f3af7649812391a38b3"
    },
    .
    .
    .
    {
      title: "Hurricane Local Statement",
      regions: ["Austin", ... , "Wharton"],
      severity: "advisory",
      time: 1503748800,
      expires: 1503683100,
      description:
        "This product covers Southeast Texas **HURRICANE HARVEY DANGEROUSLY
        APPROACHING THE TEXAS COAST** ... The next local statement will be issued
        by the National Weather Service in Houston/Galveston TX around 1030 AM CDT,
        or sooner if conditions warrant.\n",
      uri: "https://alerts.weather.gov/cap/wwacapget.php?..."
    }
  ]
}
```

```

    }
  ]
};

```

I got this information for Houston, TX, US, on a day when Hurricane Harvey was approaching the state. If you called the API on a normal day, the data would simply exclude the `alerts: [...]` part. Here, we can use a `Maybe` functor to process the received data without any problems, with or without any alerts:

```

const getField = attr => obj => obj[attr];
const os = require("os");

const produceAlertsTable = weatherObj =>
  Maybe.of(weatherObj)
    .map(getField("alerts"))
    .map(a =>
      a.map(
        x =>
          `|<td>${x.title}</td>` +
            `console.log(produceAlertsTable(x).valueOf())
);


```

Of course, you would probably do something more interesting than just logging the value of the contained result of `produceAlertsTable()`! The most likely option would be to `map()` again with a function that would output the table, send it to a client, or do whatever you needed to do. In any case, the resulting output would look something like this:

```

<table><tr><td>Tropical Storm Warning</td><td>...TROPICAL STORM WARNING
REMAINS IN EFFECT... ..STORM SURGE WATCH REMAINS IN EFFECT... * WIND -
LATEST LOCAL FORECAST: Below tropical storm force wind - Peak Wind
Forecast: 25-35 mph with gusts to 45 mph - CURRENT THREAT TO LIFE AND
PROPERTY: Moderate - The wind threat has remained nearly steady from the
previous assessment. - Emergency plans should include a reasonable threat
for strong tropical storm force wind of 58 to 73 mph. - To be safe,
earnestly prepare for the potential of significant...</td></tr>
<tr><td>Flash Flood Watch</td><td>...FLASH FLOOD WATCH REMAINS IN EFFECT
THROUGH MONDAY MORNING... The Flash Flood Watch continues for * Portions of
Southeast Texas...including the following
counties...Austin...Brazoria...Brazos...Burleson...
Chambers...Colorado...Fort Bend...Galveston...Grimes...

```

```
Harris...Jackson...Liberty...Matagorda...Montgomery...Waller... Washington
and Wharton. * Through Monday morning * Rainfall from Harvey will cause
devastating and life threatening flooding as a prolonged heavy rain and
flash flood thre...</td></tr>
<tr><td>Hurricane Local Statement</td><td>This product covers Southeast
Texas **PREPARATIONS FOR HARVEY SHOULD BE RUSHED TO COMPLETION THIS
MORNING** NEW INFORMATION ----- * CHANGES TO WATCHES AND
WARNINGS: - None * CURRENT WATCHES AND WARNINGS: - A Tropical Storm Warning
and Storm Surge Watch are in effect for Chambers and Harris - A Tropical
Storm Warning is in effect for Austin, Colorado, Fort Bend, Liberty,
Waller, and Wharton - A Storm Surge Warning and Hurricane Warning are in
effect for Jackson and Matagorda - A Storm S...</td></tr></table>
```

The output of the preceding code can be seen in the following screenshot:

Tropical Storm Warning	...TROPICAL STORM WARNING REMAINS IN EFFECT... ...STORM SURGE WATCH REMAINS IN EFFECT... * WIND - LATEST LOCAL FORECAST: Below tropical storm force wind - Peak Wind Forecast: 25-35 mph with gusts to 45 mph - CURRENT THREAT TO LIFE AND PROPERTY: Moderate - The wind threat has remained nearly steady from the previous assessment. - Emergency plans should include a reasonable threat for strong tropical storm force wind of 58 to 73 mph. - To be safe, earnestly prepare for the potential of significant...
Flash Flood Watch	...FLASH FLOOD WATCH REMAINS IN EFFECT THROUGH MONDAY MORNING... The Flash Flood Watch continues for * Portions of Southeast Texas...including the following counties...Austin...Brazoria...Brazos...Burleson... Chambers...Colorado...Fort Bend...Galveston...Grimes... Harris...Jackson...Liberty...Matagorda...Montgomery...Waller... Washington and Wharton. * Through Monday morning * Rainfall from Harvey will cause devastating and life threatening flooding as a prolonged heavy rain and flash flood thre...
Hurricane Local Statement	This product covers Southeast Texas **PREPARATIONS FOR HARVEY SHOULD BE RUSHED TO COMPLETION THIS MORNING** NEW INFORMATION ----- * CHANGES TO WATCHES AND WARNINGS: - None * CURRENT WATCHES AND WARNINGS: - A Tropical Storm Warning and Storm Surge Watch are in effect for Chambers and Harris - A Tropical Storm Warning is in effect for Austin, Colorado, Fort Bend, Liberty, Waller, and Wharton - A Storm Surge Warning and Hurricane Warning are in effect for Jackson and Matagorda - A Storm S...

Figure 12.1: The output table is not much to look at, but the logic that produced it didn't require a single if statement

If we had called `getAlerts(-34.9, -54.60, ...)` with the coordinates for Montevideo, Uruguay, instead, since there were no alerts for that city, the `getField("alerts")` function would have returned `undefined`—and since that value is recognized by the `Maybe` functor, and even though all the following `map()` operations would still be executed, no one would actually do anything, and a `null` value would be the final result.

We took advantage of this behavior when we coded the error logic. If an error occurs when calling the service, we would still call the original callback to produce a table but provide an empty object. Even if this result is unexpected, we would be safe because the same guards would avoid causing a runtime error.

As a final enhancement, we can add an `orElse()` method to provide a default value when no one is present. The added method will return the default value if `Maybe` is a `Nothing`, or the `Maybe` value itself otherwise:

```
class Maybe extends Functor {
  //
  // everything as before...
  //
  orElse(v) {
    return this.isNothing() ? v : this.valueOf();
  }
}
```

Using this new method instead of `valueOf()`, if you're trying to get the alerts for someplace without them, would just get whatever default value you wanted. In the case we mentioned previously when we attempted to get the alerts for Montevideo, instead of a `null` value, we would get the following appropriate result:

```
getAlerts(-34.9, -54.6, x =>
  console.log(
    produceAlertsTable(x).orElse("<span>No alerts today.</span>")
  )
);
```

With this, we have looked at an example of dealing with different situations when working with an API. Let's quickly revisit another topic from the previous chapter and look at a better implementation of Prisms.

Implementing Prisms

The more common implementations of Prisms (which we first met in the *Prisms* section of Chapter 10, *Ensuring Purity – Immutability*) we came across was that instead of returning either some value or `undefined` and leaving it up to the caller to check what happened, we could opt to return a `Maybe`, which already provides us with easy ways to deal with missing values. In our new implementation (which we'll look at soon), our example from the aforementioned chapter would look like this:

```
const author = {
  user: "fkereki",
  name: {
    first: "Federico",
    middle: "",
    last: "Kereki"
  },
  books: [
```

```
    { name: "GWT", year: 2010 },
    { name: "FP", year: 2017 },
    { name: "CB", year: 2018 }
  ]
};
```

If we wanted to access the `author.user` attribute, the result would be different:

```
const pUser = prismProp("user");

console.log(review(pUser, author).toString());

/*
   Just("fkereki")
*/
```

Similarly, if we asked for a non-existent `pseudonym` attribute, instead of `undefined` (as in our previous version of Prisms), we would get a `Nothing`:

```
const pPseudonym = prismProp("pseudonym");

console.log(review(pPseudonym, author).toString());

/*
   Nothing()
*/
```

So, this new version of Prisms is better to work with if you are already used to dealing with `Maybe` values. What do we need to implement this? We need just a single change; our `Constant` class now needs to return a `Maybe` instead of a value, so we'll have a new `ConstantP` (P for Prism) class:

```
class ConstantP {
  constructor(v) {
    this.value = Maybe.of(v);
    this.map = () => this;
  }
}
```

We will have to rewrite `preview()` to use the new class, and that finishes the change:

```
const preview = curry(
  (prismAttr, obj) => prismAttr(x => new ConstantP(x))(obj).value
);
```

So, getting Prisms to work with Maybes wasn't that hard, and now we have a consistent way of dealing with possibly missing attributes. Working in this fashion, we can simplify our coding and avoid many tests for nulls and other similar situations. However, we may want to go beyond this; for instance, we may want to know *why* there were no alerts: was it a service error? Or just a normal situation? Just getting a `null` at the end isn't enough, and in order to work with these new requirements, we will need to add some extra functionality to our functors (as we'll see in the next section) and enter the domain of *monads*.

Monads

Monads have weird fame among programmers. Well-known developer Douglas Crockford has famously spoken of a curse, maintaining that *Once you happen to finally understand monads, you immediately lose the ability to explain them to other people!* On a different note, if you decide to go to the basics and read *Categories for the Working Mathematician* by Saunders Mac Lane (one of the creators of category theory), you may find a somewhat disconcerting explanation – which is not too illuminating!

"A monad in X is just a monoid in the category of endofunctors of X, with product \times replaced by composition of endofunctors and unit set by the identity endofunctor."

The difference between monads and functors is that the former adds some extra functionality; we'll see what functionality they add soon. Let's start by looking at the new requirements before moving on and considering some common, useful monads. As with functors, we will have a basic monad, which you could consider to be an *abstract* version, and specific *monadic types*, which are *concrete* implementations, geared to solve specific cases.



If you want to read a precise and careful description of functors, monads, and their family (but leaning heavily to the theoretical side, with plenty of algebraic definitions to go around), you can try the *Fantasy Land Specification* at <https://github.com/fantasyland/fantasy-land/>. Don't say we didn't warn you: the alternative name for that page is *Algebraic JavaScript Specification*!

Adding operations

Let's consider a simple problem. Suppose you have the following pair of functions, working with `Maybe` functors: the first function tries to search for *something* (say, a client or a product) given its *key*, and the second attempts to extract *some* attribute from it (I'm being purposefully vague because the problem does not have anything to do with whatever objects or things we may be working with). Both functions produce `Maybe` results to avoid possible errors. We are using a mocked search function, just to help us see the problem: for even keys, it returns fake data, and for odd keys, it throws an exception. The code for this search is very simple:

```
const fakeSearchForSomething = key => {
  if (key % 2 === 0) {
    return {key, some: "whatever", other: "more data"};

    } else {
    throw new Error("Not found");
  }
};
```

Using this search, our `findSomething()` function will try to do a search, return a `Maybe.of()` for a successful call, or a `Maybe.of(null)` (in other terms, a `Nothing`) in case of an error:

```
const findSomething = key => {
  try {
    const something = fakeSearchForSomething(key);
    return Maybe.of(something);
  } catch (e) {
    return Maybe.of(null);
  }
};
```

With this, we could think of writing these two functions to do some searching, but not everything would be fine; can you see what the problem is here?

```
const getSome = something => Maybe.of(something.map(getField("some")));

const getSomeFromSomething = key => getSome(findSomething(key));
```

The problem in this sequence is that the output from `getSome()` is a `Maybe` value, which itself contains a `Maybe` value, so the result we want is double wrapped, as we can see by executing a couple of calls, for an even number (which will return a "whatever") and for an odd number (which will be an error), as follows:

```
let xxx = getSomeFromSomething(2222).valueOf().valueOf(); // "whatever"

let yyy = getSomeFromSomething(9999).valueOf().valueOf(); // null
```

This problem can be easily solved in this toy problem if we just avoid using `Maybe.of()` in `getSome()`, but this kind of result can happen in many more complex ways. For instance, you could be building a `Maybe` out of an object, one of whose attributes happened to be a `Maybe`, and you'd get the same situation when accessing that attribute: you would end up with a double wrapped value.

Now, we are going to look into monads. Monads should provide the following operations:

- A constructor.
- A function that inserts a value into a monad: our `of()` method.
- A function that allows us to chain operations: our `map()` method.
- A function that can remove extra wrappers: we will call it `unwrap()`. It will solve our preceding multiple wrapper problems. Sometimes, this function is called `flatten()`.

We will also have a function to chain calls, just to simplify our coding, and another function to apply functions, but we'll get to those later. Let's see what a monad looks like in actual JavaScript code. Data type specifications are very much like those for functors, so we won't repeat them here:

```
class Monad extends Functor {
  static of(x) {
    return new Monad(x);
  }

  map(fn) {
    return Monad.of(fn(this[VALUE]));
  }

  unwrap() {
    const myValue = this[VALUE];
    return myValue instanceof Container ? myValue.unwrap() : this;
  }
}
```


We use recursion to successively remove wrappers until the wrapped value isn't a container anymore. Using this method, we could avoid double wrapping easily, and we could rewrite our previous troublesome function like this:

```
const getSomeFromSomething = key => getSome(findSomething(key)).unwrap();
```

However, this sort of problem could reoccur at different levels. For example, if we were doing a series of `map()` operations, any of the intermediate results may end up being double wrapped. You could easily solve this by remembering to call `unwrap()` after each `map()`—note that you could do this even if it is not actually needed since the result of `unwrap()` would be the very same object (can you see why?). But we can do better! Let's define a `chain()` operation (sometimes named `flatMap()` instead, which is a bit confusing since we already have another meaning for that; see Chapter 5, *Programming Declaratively – A Better Style*, for more on this) that will do both things for us:

```
class Monad extends Functor {  
  //  
  // everything as before...  
  //  
  chain(fn) {  
    return this.map(fn).unwrap();  
  }  
}
```

There's only one operation left. Suppose you have a curried function with two parameters; nothing outlandish! What would happen if you were to provide that function to a `map()` operation?

```
const add = x => y => x+y; // or curry((x,y) => x+y)
```

```
const something = Monad.of(2).map(add);
```

What would something be? Given that we have only provided one argument to `add`, the result of that application will be a function—not just any function, though, but a *wrapped* one! (Since functions are first-class objects, there's no logical obstacle to wrapping a function in a `Monad`, is there?) What would we want to do with such a function? To be able to apply this wrapped function to a value, we'll need a new method: `ap()`. What could its value be? In this case, it could either be a plain number, or a number wrapped in a `Monad` as a result of other operations. Since we can always `Map.of()` a plain number into a wrapped one, let's have `ap()` work with a `Monad` as its parameter; the new method would be as follows:

```
class Monad extends Functor {  
  //
```

```
// everything as earlier...  
//  
ap(m) {  
  return m.map(this.valueOf());  
}  
}
```

With this, you could then do the following:

```
const monad5 = something.ap(Monad.of(3)); // Monad(5)
```

You can use monads to hold values or functions and to interact with other monads and chaining operations as you wish. So, as you can see, there's no big trick to monads, which are just functors with some extra methods. Now, let's look at how we can apply them to our original problem and handle errors in a better way.

Handling alternatives – the Either monad

Knowing that a value was missing may be enough in some cases, but in others, you'll want to be able to provide an explanation. We can get such an explanation if we use a different functor, which will take one of two possible values: one associated with a problem, error, or failure, and another associated with normal execution, or success:

- A *left* value, which should be null, but if present then it represents some kind of special value (for example, an error message or a thrown exception) that cannot be mapped over
- A *right* value, which represents the *normal* value of the functor and can be mapped over

We can construct this monad in a similar way to what we did for `Maybe` (actually, the added operations make it better for `Maybe` to extend `Monad` as well). The constructor will receive a left and a right value: if the left value is present, it will become the value of the `Either` monad; otherwise, the right value will be used. Since we have been providing `of()` methods for all our functors, we need one for `Either` too. The `Left` monad is very similar to our previous `Nothing`:

```
class Left extends Monad {  
  isLeft() {  
    return true;  
  }  
  map(fn) {  
    return this;  
  }  
}
```

Similarly, `Right` resembles our previous `Just`:

```
class Right extends Monad {
  isLeft() {
    return false;
  }

  map(fn) {
    return Either.of(null, fn(this[VALUE]));
  }
}
```

And with these two monads under our belt, we can write our `Either` monad. It shouldn't be a surprise that this resembles our previous `Maybe`, should it?

```
class Either extends Monad {
  constructor(left, right) {
    return right === undefined || right === null
      ? new Left(left)
      : new Right(right);
  }

  static of(left, right) {
    return new Either(left, right);
  }
}
```

The `map()` method is key. If this functor has got a *left* value, it won't be processed any further; in other cases, the mapping will be applied to the *right* value, and the result will be wrapped. Now, how can we enhance our code with this? The key idea is for every involved method to return an `Either` monad; `chain()` will be used to execute operations one after another. Getting the alerts would be the first step—we invoke the callback either with an `AJAX FAILURE` message or with the result from the API call, as follows:

```
const getAlerts2 = (lat, long, callback) => {
  const SERVER = "https://api.darksky.net/forecast";
  const UNITS = "units=si";
  const EXCLUSIONS = "exclude=minutely,hourly,daily,flags";
  const API_KEY = "you.have.to.get.your.own.key";

  request
    .get(`${SERVER}/${API_KEY}/${lat},${long}?${UNITS}&${EXCLUSIONS}`)
    .end((err, res) =>
      callback(
        err
          ? Either.of("AJAX FAILURE", null)
          : Either.of(null, JSON.parse(res.text))
      )
    )
}
```

```

    )
  );
};

```

Then, the general process would be as follows. We use an `Either` again: if there are no alerts, instead of an array, we would return a `NO ALERTS` message:

```

const produceAlertsTable2 = weatherObj => {
  return weatherObj
    .chain(obj => {
      const alerts = getField("alerts")(obj);
      return alerts
        ? Either.of(null, alerts)
        : Either.of("NO ALERTS", null);
    })

    .chain(a =>
      a.map(
        x =>
          `|<td>${x.title}</td>` +
          `

|  |

```

Note how we used `chain()` so that multiple wrappers would be no problem. Now, we can test multiple situations and get appropriate results—or at least, for the current weather situation around the world!

- For Houston, TX, we still get an HTML table.
- For Montevideo, UY, we get a text saying there were no alerts.
- For a point with wrong coordinates, we learn that the AJAX call failed: nice!

```

// Houston, TX, US:
getAlerts2(29.76, -95.37, x =>
  console.log(produceAlertsTable2(x).toString());
  Right("...a table with alerts: lots of HTML code..."));

// Montevideo, UY
getAlerts2(-34.9, -54.6, x =>
  console.log(produceAlertsTable2(x).toString());
  Left("NO ALERTS"));

```

```
// A point with wrong coordinates
getAlerts2(444, 555, x => console.log(produceAlertsTable2(x).toString()));
Left("AJAX FAILURE");
```

We are not done with the `Either` monad. It's likely that much of your code will involve calling functions. Let's look at a better way of achieving this by using a variant of this monad.

Calling a function – the Try monad

If we are calling functions that may throw exceptions and we want to do so in a functional way, we could use the `Try` monad to encapsulate the function result or the exception. The idea is basically the same as the `Either` monad: the only difference is in the constructor, which receives a function and calls it:

- If there are no problems, the returned value becomes the right value for the monad.
- If there's an exception, it will become the left value.

This can be seen in the following code:

```
class Try extends Either {
  constructor(fn, msg) {
    try {
      return Either.of(null, fn());
    } catch (e) {
      return Either.of(msg || e, null);
    }
  }

  static of(fn, msg) {
    return new Try(fn, msg);
  }
}
```

Now, we can invoke any function, catching exceptions in a good way. For example, the `getField()` function that we have been using would crash if it were called with a null argument:

```
// getField :: String → attr → a | undefined
const getField = attr => obj => obj[attr];
```

In the *Implementing Prisms* section of Chapter 10, *Ensuring Purity – Immutability*, we wrote a `getFieldP()` function that could deal with null values, but here, we will rewrite it using the `Try` monad, so, in addition, it will play nice with other composed functions. The alternative implementation of our getter would be as follows:

```
const getField2 = attr => obj => Try.of(() => obj[attr], "NULL OBJECT");
```

We can check that this works by trying to apply our new function to a `null` value:

```
const x = getField2("somefield")(null);

console.log(x.isLeft()); // true

console.log(x.toString()); // Left(NULL OBJECT)
```

There are many more monads and, of course, you can even define your own, so we couldn't possibly go over all of them. However, let's visit just one more—one that you have been using already, without being aware of its *monad-ness*!

Unexpected monads – promises

Let's finish this section on monads by mentioning yet another one that you may have used, though under a different name: *Promises*! Previously, we mentioned that functors (and, remember, monads are functors) had at least something in common with promises: using a method in order to access the value. However, the analogy is greater than that!

- `Promise.resolve()` corresponds with `Monad.of()`—if you pass a value to `.resolve()`, you'll get a promise resolved to that value, and if you provide a promise, you will get a new promise, the value of which will be that of the original one (see https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Promise/resolve for more on this). This is an *unwrapping* behavior!
- `Promise.then()` stands for `Monad.map()` as well as `Monad.chain()`, given the mentioned unwrapping.
- We don't have a direct match to `Monad.ap()`, but we could add something like the following code:

```
Promise.prototype.ap = function(promise2) {
  return this.then(x => promise2.map(x));
};
```



Even if you opt for the modern `async` and `await` features, internally, they are based on promises. Furthermore, in some situations, you may still need `Promise.race()` and `Promise.all()`, so it's likely you will keep using promises, even if you opt for full ES8 coding.

This is an appropriate ending for this section. Earlier, you found out that common arrays were, in fact, functors. Now, in the same way that *Monsieur Jourdain* (a character in Molière's play *Le Bourgeois Gentilhomme*, *The Bourgeois Gentleman*) discovered that all his life he had been speaking in prose, you now know you had already been using monads, without even knowing it! So far, we have learned how to build different types of containers. Now, let's learn how functions can also make do as containers, as well as for all kinds of data structures!

Functions as data structures

So far, we have learned how to use functions to work with or transform other functions to process data structures or to create data types. Now, we'll finish this chapter by showing you how a function can actually implement a data type by itself, becoming a sort of container of its own. In fact, this is a basic theoretical point of the lambda calculus (and if you want to learn more, look up *Church Encoding* and *Scott Encoding*), so we may very well say that we have come back to where we began this book, at the origins of FP! We will start with a detour that considers binary trees in a different functional language, Haskell, and then move on to implementing trees as functions, but in JavaScript; this experience will help you work out how to deal with other data structures.

Binary trees in Haskell

Consider a binary tree. Such a tree may either be empty or consist of a node (the tree *root*) with two sons: a *left* binary tree and a *right* one. A node that has no sons is called a *leaf*.



In [Chapter 9, *Designing Functions – Recursion*](#), we worked with more general tree structures, such as a filesystem or the browser DOM itself, which allow a node to have any number of sons. In the case of the trees we are working with in this section, each node always has two sons, although each of them may be empty. The difference may seem minor, but allowing for empty subtrees is what lets you define that all nodes are binary.

Let's make a digression with the Haskell language. In it, we might write something like the following; *a* would be the type of whatever value we hold in the nodes:

```
data Tree a = Nil | Node a (Tree a) (Tree a)
```

In the Haskell language, *pattern matching* is often used for coding. For example, we could define an `empty` function as follows:

```
empty :: Tree a -> Bool
empty Nil = True
empty (Node root left right) = False
```

What does this mean? Apart from the data type definition, the logic is simple: if the tree is `Nil` (the first possibility in the definition of the type), then the tree is certainly empty; otherwise, the tree isn't empty. The last line would probably be written as `empty _ = False` while using `_` as a placeholder, because you don't actually care about the components of the tree; the mere fact that it's not `Nil` suffices.

Searching for a value in a binary search tree (in which the root is greater than all the values of its left subtree and less than all the values of its right subtree) would be written in a similar fashion, as follows:

```
contains :: (Ord a) => (Tree a) -> a -> Bool
contains Nil _ = False
contains (Node root left right) x
  | x == root = True
  | x < root = contains left x
  | x > root = contains right x
```

What patterns are matched here? We have four patterns now, which must be considered in order:

- An empty tree (`Nil`—it doesn't matter what we are looking for, so just write `_`) doesn't contain the searched value.
- If the tree isn't empty, and the root matches the searched value (`x`), we are done.
- If the root doesn't match and is greater than the searched value, the answer is found while searching in the left subtree.
- Otherwise, the answer is found by searching in the right subtree.

There's an important point to remember: for this data type, which is a union of two possible types, we have to provide two conditions, and pattern matching will be used to decide which one is going to be applied. Keep this in mind!

Functions as binary trees

Can we do something similar with functions? The answer is yes: we will represent a tree (or any other structure) with a function itself—not with a data structure that is processed by a set of functions, nor with an object with some methods, but by just a function. Furthermore, we will get a functional data structure that's 100% immutable, which, if updated, produces a new copy of itself. We will do all this without using objects; here, closures will provide the desired results.

How can this work? We shall be applying similar concepts to the ones we looked at earlier in this chapter, so the function will act as a container and it will produce, as its result, a mapping of its contained values. Let's walk backward and start by looking at how we'll use the new data type. Then, we'll go to the implementation details.

Creating a tree can be done by using two functions: `EmptyTree()` and `Tree(value, leftTree, rightTree)`. For example, let's say we wish to create a tree similar to the one shown in the following diagram:

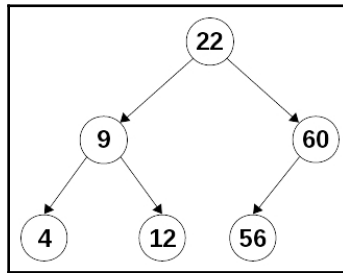


Figure 12.2: A binary search tree, created by the following code

We can create this using the following code:

```
const myTree = Tree(
  22,
  Tree(
    9,
    Tree(4, EmptyTree(), EmptyTree()),
    Tree(12, EmptyTree(), EmptyTree())
  ),
  Tree(
    60,
    Tree(56, EmptyTree(), EmptyTree()),
    EmptyTree()
  )
);
```

How do you work with this structure? According to the data type description, whenever you work with a tree, you must consider two cases: a non-empty tree or an empty one. In the preceding code, `myTree()` is actually a function that receives two functions as arguments, one for each of the two data type cases. The first function will be called with the node value and left and right trees as arguments, while the second function will receive none. So, to get the root, we could write something similar to the following:

```
const myRoot = myTree((value, left, right) => value, () => null);
```

If we were dealing with a non-empty tree, we would expect the first function to be called and produce the value of the root as the result. With an empty tree, the second function should be called, and then a `null` value would be returned.

Similarly, if we wanted to count how many nodes there are in a tree, we would write the following:

```
const treeCount = aTree => aTree(
  (value, left, right) => 1 + treeCount(left) + treeCount(right),
  () => 0
);

console.log(treeCount(myTree));
```

For non-empty trees, the first function would return 1 (for the root), plus the node count from both the root's subtrees. For empty trees, the count is simply 0. Get the idea?

Now, we can show the `Tree()` and `EmptyTree()` functions. They are as follows:

```
const Tree = (value, left, right) => (destructure, __) =>
  destructure(value, left, right);

const EmptyTree = () => (__, destructure) => destructure();
```

The `destructure()` function is what you will pass as an argument (the name comes from the destructuring statement in JavaScript, which lets you separate an object attribute into distinct variables). You will have to provide two versions of this function. If the tree is non-empty, the first function will be executed; for an empty tree, the second one will be run (this mimics the *case* selection in the Haskell code, except we are placing the non-empty tree case first and the empty tree last). The `__` variable is used as a placeholder that stands for an otherwise ignored argument but shows that two arguments are assumed.

This can be hard to understand, so let's look at some more examples. If we need to access specific elements of a tree, we have the following three functions, one of which (`treeRoot()`) we've already looked at—let's repeat it here for completeness:

```
const treeRoot = tree => tree((value, left, right) => value, () => null);
const treeLeft = tree => tree((value, left, right) => left, () => null);
const treeRight = tree => tree((value, left, right) => right, () => null);
```



Functions that access the component values of structures (or *constructions*, to use another term) are called **projector functions**. We won't be using this term, but you may find it being used elsewhere.

How can we decide if a tree is empty? See if you can figure out why the following short line of code works:

```
const treeIsEmpty = tree => tree(() => false, () => true);
```

Let's go over a few more examples of this. For example, we can build an object out of a tree, and that would help with debugging. I added logic to avoid including left or right empty subtrees, so the produced object would be more compact; check out the two `if` statements in the following code:

```
const treeToObject = tree =>
  tree(
    (value, left, right) => {
      const leftBranch = treeToObject(left);
      const rightBranch = treeToObject(right);
      const result = { value };

      if (leftBranch) {
        result.left = leftBranch;
      }

      if (rightBranch) {
        result.right = rightBranch;
      }

      return result;
    },
    () => null
  );
```

Note the usage of recursion, as in the *Traversing a tree structure* section of Chapter 9, *Designing Functions – Recursion*, in order to produce the object equivalents of the left and right subtrees. An example of this function is as follows; I edited the output to make it clearer:

```
console.log(treeToObject(myTree));  
/*  
{  
  value: 22,  
  left: {  
    value: 9,  
    left: {  
      value: 4  
    },  
    right: {  
      value: 12  
    }  
  },  
  right: {  
    value: 60,  
    left: {  
      value: 56  
    }  
  }  
}  
*/
```

Can we search for a node? Of course, and the logic follows the definition we saw in the previous section closely. (We could have shortened the code a bit, but I wanted to parallel the Haskell version.) Our `treeSearch()` function could be as follows:

```
const treeSearch = (findValue, tree) =>  
  tree(  
    (value, left, right) =>  
      findValue === value  
        ? true  
        : findValue < value  
          ? treeSearch(findValue, left)  
          : treeSearch(findValue, right),  
    () => false  
  );
```

To round off this section, let's also look at how to add new nodes to a tree. Study the code carefully; you'll notice how the current tree isn't modified and that a new one is produced instead. Of course, given that we are using functions to represent our tree data type, it should be obvious that we wouldn't have been able to just modify the old structure: it's immutable by default. The tree insertion function would be as follows:

```
const treeInsert = (newValue, tree) =>
  tree(
    (value, left, right) =>
      newValue <= value
        ? Tree(value, treeInsert(newValue, left), right)
        : Tree(value, left, treeInsert(newValue, right)),
    () => Tree(newValue, EmptyTree(), EmptyTree())
  );
```

When trying to insert a new key, if it's less than or equal to the root of the tree, we produce a new tree that has the current root as its own root, maintains the old right subtree, but changes its left subtree to incorporate the new value (which will be done in a recursive way). If the key was greater than the root, the changes wouldn't have been symmetrical; they would have been analogous. If we try to insert a new key and we find ourselves with an empty tree, we just replace that empty structure with a new tree where the new value is its root and it has empty left and right subtrees.

We can test out this logic easily, but the simplest way is to verify that the binary tree that we showed earlier (*Figure 12.2*) is generated by the following sequence of operations:

```
let myTree = EmptyTree();
myTree = treeInsert(22, myTree);
myTree = treeInsert(9, myTree);
myTree = treeInsert(60, myTree);
myTree = treeInsert(12, myTree);
myTree = treeInsert(4, myTree);
myTree = treeInsert(56, myTree);

// The resulting tree is:
{
  value: 22,
  left: { value: 9, left: { value: 4 }, right: { value: 12 } },
  right: { value: 60, left: { value: 56 } }
};
```

We could make this insertion function even more general by providing the comparator function that would be used to compare values. In this fashion, we could easily adapt a binary tree to represent a generic map. The value of a node would actually be an object such as `{key:... , data:...}` and the provided function would compare `newValue.key` and `value.key` to decide where to add the new node. Of course, if the two keys were equal, we would change the root of the current tree. The new tree insertion code would be as follows:

```
const compare = (obj1, obj2) =>
  obj1.key === obj2.key ? 0 : obj1.key < obj2.key ? -1 : 1;

const treeInsert2 = (comparator, newValue, tree) =>
  tree(
    (value, left, right) =>
      comparator(newValue, value) === 0
        ? Tree(newValue, left, right)
        : comparator(newValue, value) < 0
          ? Tree(value, treeInsert2(comparator, newValue, left), right)
          : Tree(value, left, treeInsert2(comparator, newValue, right)),
    () => Tree(newValue, EmptyTree(), EmptyTree())
  );
```

What else do we need? Of course, we can program diverse functions: deleting a node, counting nodes, determining a tree's height, comparing two trees, and so on. However, in order to gain more usability, we should really turn the structure into a functor by implementing a `map()` function. Fortunately, using recursion, this proves to be easy—we apply the mapping function to the tree root and use `map()` recursively on the left and right subtrees, as follows:

```
const treeMap = (fn, tree) =>
  tree(
    (value, left, right) =>
      Tree(fn(value), treeMap(fn, left), treeMap(fn, right)),
    () => EmptyTree()
  );
```

We could go on with more examples, but that wouldn't change the important conclusions we can derive from this work:

- We are handling a data structure (a recursive one, at that) and representing it with a function.
- We aren't using any external variables or objects for the data: closures are used instead.

- The data structure itself satisfies all the requirements we analyzed in `Chapter 10`, *Ensuring Purity – Immutability*, insofar that it is immutable and all the changes always produce new structures.
- The tree is a functor, providing all the corresponding advantages.

In this section, we have looked at one more application of functional programming, as well as how a function can actually become a structure by itself, which isn't what we are usually accustomed to!

Summary

In this chapter, we looked at the theory of data types and learned how to use and implement them from a functional point of view. We started with how to define function signatures to help us understand the transformations that are implied by the multiple operations we looked at later. Then, we went on to define several containers, including functors and monads, and saw how they can be used to enhance function composition. Finally, we learned how functions can be directly used by themselves, with no extra baggage, to implement functional data structures to simplify dealing with errors.

In this book, we have looked at several features of functional programming for JavaScript. We started out with some definitions, and a practical example, and then moved on to important considerations such as pure functions, side effects avoidance, immutability, testability, building new functions out of other ones, and implementing a data flow based upon function connections and data containers. We have looked at a lot of concepts, but I'm confident that you'll be able to put them to practice and start writing even higher-quality code – give it a try, and thank you very much for reading my book!

Questions

12.1. **Maybe tasks?** In the *Questions* section of `Chapter 8`, *Connecting Functions – Pipelining and Composition*, a question (8.2) had to do with getting the pending tasks for a person while taking errors or border situations into account, such as the possibility that the selected person might not even exist. Redo that exercise but using `Maybe` or `Either` monads to simplify that code.

12.2. Extending your trees: To get a more complete implementation of our functional binary search trees, implement the following functions:

- Calculate the tree's height or, equivalently, the maximum distance from the root to any other node
- List all the tree's keys, in ascending order
- Delete a key from a tree

12.3. Functional lists: In the same spirit as binary trees, implement functional lists. Since a list is defined to be either empty or a node (`head`), followed by another list (`tail`), you might want to start with the following:

```
const List = (head, tail) => (destructure, __) =>
  destructure(head, tail);
const EmptyList = () => (__, destructure) => destructure();
```

Here are some easy one-line operations to get you started:

```
const listHead = list => list((head, __) => head, () => null);
const listTail = list => list(__, tail) => tail, () => null);
const listIsEmpty = list => (() => false, () => true);
const listSize = list => list((head, tail) => 1 + listSize(tail),
  () => 0);
```

You could consider having these operations:

- Transforming a list into an array and vice versa
- Reversing a list
- Appending one list to the end of another list
- Concatenating two lists

Don't forget the `listMap()` function! Also, the `listReduce()` and `listFilter()` functions will come in handy.

12.4. Code shortening: We mentioned that the `treeSearch()` function could be shortened—can you do that? Yes, this is more of a JavaScript problem than a functional one, and I'm not saying that shorter code is necessarily better, but many programmers act as if it were, so it's good to be aware of such a style if only because you're likely to find it.

Bibliography

The following texts are freely available online:

- *ECMA-262: ECMAScript 2019 Language Specification*, latest edition (currently the 10th) at <http://www.ecma-international.org/ecma-262/>. This provides the official standard to the current version of JS.
- *ELOQUENT JAVASCRIPT, third Edition*, by *Marijn Haverbeke*, at <http://eloquentjavascript.net/>.
- *EXPLORING ES6*, by *Dr. Axel Rauschmayer*, at <http://exploringjs.com/es6/>.
- *Exploring ES2016 AND ES2017*, by *Dr. Axel Rauschmayer*, at <http://exploringjs.com/es2016-es2017/>.
- *Exploring ES2018 AND ES2019*, by *Dr. Axel Rauschmayer*, at <http://exploringjs.com/es2018-es2019/>. This text will let you get up to date with the latest features in JS.
- *Functional-Light JavaScript*, by *Kyle Simpson*, at <https://github.com/getify/Functional-Light-JS>.
- *JavaScript Allongé*, by *Reginald Braithwaite*, at <https://leanpub.com/javascript-allonge/read>.
- *Professor Frisby's Mostly Adequate Guide to Functional Programming*, by *Dr Boolean (Brian Lonsdorf)*, at <https://github.com/MostlyAdequate/mostly-adequate-guide>

If you prefer printed books, you can go with this list:

- *Beginning Functional JavaScript*, *Anto Aravinth*, Apress, 2017
- *Discover Functional JavaScript*, *Cristian Salcescu*, (independently published) 2019
- *Functional JavaScript*, *Michael Fogus*, O'Reilly Media, 2013
- *Functional Programming in JavaScript*, *Dan Mantyla*, Packt Publishing, 2015
- *Functional Programming in JavaScript*, *Luis Atencio*, Manning Publications, 2016
- *Hands-on Functional Programming with TypeScript*, *Remo Jansen*, Packt Publishing, 2019
- *Introduction to Functional Programming*, *Richard Bird & Philip Wadler*, Prentice Hall International, 1988. A more theoretical point of view, not dealing specifically with JavaScript.
- *Pro JavaScript Design Patterns*, *Ross Harmes & Dustin Díaz*, Apress, 2008

- *Secrets of the JavaScript Ninja*, John Resig & Bear Bibeault, Manning Publications, 2012

Also interesting, though with a lesser focus on Functional Programming:

- *High-Performance JavaScript*, Nicholas Zakas, O'Reilly Media, 2010
- *JavaScript Patterns*, Stoyan Stefanov, O'Reilly Media, 2010
- *JavaScript: The Good Parts*, Douglas Crockford, O'Reilly Media, 2008
- *JavaScript with Promises*, Daniel Parker, O'Reilly Media, 2015
- *Learning JavaScript Design Patterns*, Addy Osmani, O'Reilly Media, 2012
- *Mastering JavaScript Design Patterns*, 2nd Edition, Simon Timms, Packt Publishing, 2016
- *Mastering JavaScript High Performance*, Chad Adams, Packt Publishing, 2015
- *Pro JavaScript Performance*, Tom Barker, Apress, 2012

On the subject of Reactive Functional Programming:

- *Mastering Reactive JavaScript*, Erich de Souza Oliveira, Packt Publishing, 2017
- *Reactive Programming with Node.js*, Fernando Doglio, Apress, 2016
- *Reactive Programming with RxJS*, Sergi Mansilla, The Pragmatic Programmers, 2015

Answers to Questions

Here are the solutions (partial, or worked out in full) to the questions that were contained within the chapters in this book. In many cases, there are extra questions so that you can do further work if you choose to.

Chapter 1, Becoming Functional – Several Questions

1.1. **Classes as first-class objects:** As you may recall, a class is basically a function that can be used with `new`. Therefore, it stands to reason that we should be able to pass classes as parameters to other functions. `makeSaluteClass()` creates a class (that is, a special function) that uses a closure to remember the value of `term`. We'll be looking at more examples like this throughout this book.

1.2. **Factorial errors:** The key to avoiding repeating tests is to write a function that will check the value of the argument to ensure it's valid, and if so call an inner function to do the factorial itself, without worrying about erroneous arguments:

```
const carefulFact = n => {
  if (
    typeof n !== "undefined" &&
    Number(n) === n &&
    n >= 0 &&
    n === Math.floor(n)
  ) {
    const innerFact = n => (n === 0 ? 1 : n * innerFact(n - 1));
    return innerFact(n);
  }
};
```

```
console.log(carefulFact(3)); // 6, correct
console.log(carefulFact(3.1)); // undefined
console.log(carefulFact(-3)); // undefined
console.log(carefulFact(-3.1)); // undefined
console.log(carefulFact("3")); // undefined
console.log(carefulFact(false)); // undefined
console.log(carefulFact([])); // undefined
console.log(carefulFact({})); // undefined
```

You could throw an error when an incorrect argument is recognized, but here, I just ignored it and let the function return `undefined`.

1.3. Climbing factorial: The following code does the trick. We add an auxiliary variable, `f`, and we make it *climb* from 1 to `n`. We must be careful so that `factUp(0) === 1`:

```
const factUp = (n, f = 1) => (n <= f ? f : f * factUp(n, f + 1));
```

1.4. Code squeezing: Using arrow functions, as suggested, as well as the prefix `++` operator (for more information, see https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/Arithmetic_Operators#Increment), you can condense `newCounter()` down to the following:

```
const shorterCounter = () => {  
  let count = 0;  
  return () => ++count;  
};
```

Using arrow functions isn't hard to understand, but be aware that many developers may have questions or doubts about using `++` as a prefix operator, so this version could prove to be harder to understand.

Chapter 2, Thinking Functionally – a First Example

2.1. No extra variables: We can make do by using the `fn` variable itself as a flag. After calling `fn()`, we set the variable to `null`. Before calling `fn()`, we check that it's not `null`:

```
const once = fn => {  
  return (...args) => {  
    fn && fn(...args);  
    fn = null;  
  };  
};
```

2.2. Alternating functions: In a manner similar to what we did in the previous question, we call the first function and then switch functions for the next time. Here, we used a destructuring assignment to write the swap in a more compact manner. For more information, refer to https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/Destructuring_assignment#Swapping_variables:

```
const alternator = (fn1, fn2) => {  
  return (...args) => {
```

```
    fn1(...args);
    [fn1, fn2] = [fn2, fn1];
  };
};
```

2.3. **Everything has a limit!** We simply check whether the `limit` variable is greater than 0. If so, we decrement it by 1 and call the original function; otherwise, we do nothing:

```
const thisManyTimes = (fn, limit) => {
  return (...args) => {
    if (limit > 0) {
      limit--;
      return fn(...args);
    }
  };
};
```

Chapter 3, Starting Out with Functions – a Core Concept

3.1. **Uninitialized object?** The key is that we didn't wrap the returned object in parentheses, so JavaScript thinks the braces enclose the code to be executed. In this case, `type` is considered to be labeling a statement, which doesn't really do anything: it's an expression (`t`) that isn't used. Due to this, the code is considered valid, and since it doesn't have an explicit `return` statement, the implicit returned value is `undefined`. See <https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Statements/label> for more on labels, and https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Functions/Arrow_functions#Returning_object_literals for more on returning objects. The corrected code is as follows:

```
const simpleAction = t => ({
  type: t;
});
```

3.2. **Are arrows allowed?** There would be no problems with `listArguments2()`, but with `listArguments()`, you would get an error since `arguments` is not defined for arrow functions:

```
listArguments(22, 9, 60);
Uncaught ReferenceError: arguments is not defined
```

3.3. **One-liner:** It works! (And yes, a one-line answer is appropriate in this case!).

3.4. Spot the bug! Initially, many people look at the weird `(console(...), window.store.set(...))` code, but the bug isn't there: because of how the comma operator works, JavaScript does the logging first, and then the setting. The real problem is that `oldSet()` is not bound to the `window.store` object, so the second line should be as follows instead:

```
const oldSet = window.store.set.bind(window.store);
```

Reread the *Working with methods* section for more on this, as well as question 11.1 for another way of doing logging, that is, with decorators.

3.5. Bindless binding: If `bind()` wasn't available, you could use a closure, the `that` trick (which we saw in the *Handling the this value* section), and the `apply()` method, as follows:

```
function bind(context) {
  var that = this;
  return function() {
    return that.apply(context, arguments);
  };
}
```

We could do something similar to what we did in the *Adding missing functions* section. Alternatively, just for variety, we could use a common idiom based on the `||` operator: if `Function.prototype.bind` exists, evaluation stops right there, and the existing `bind()` method is used; otherwise, our new function is applied:

```
Function.prototype.bind =
Function.prototype.bind ||
function(context) {
  var that = this;
  return function() {
    return that.apply(context, arguments);
  };
};
```

Chapter 4, Behaving Properly – Pure Functions

4.1. Minimalistic function: It works because $fib(0)=0$ and $fib(1)=1$, so it's true that for $n < 2$, $fib(n)=n$.

4.2. A cheap way: Basically, this algorithm works the same way as you'd calculate a Fibonacci number by hand. You'd start by writing down $fib(0)=0$ and $fib(1)=1$, adding them to get $fib(2)=1$, adding the last two to get $fib(3)=2$, and so on. In this version of the algorithm, a and b stand for two consecutive Fibonacci numbers. This implementation is quite efficient!

4.3. A shuffle test: Before shuffling the array, sort a copy of it, `JSON.stringify()` it, and save the result. After shuffling, sort a copy of the shuffled array and `JSON.stringify()` it as well. Finally, two JSON strings should be produced, which should be equal. This does away with all the other tests since it ensures that the array doesn't change length, nor its elements:

```
describe("shuffleTest", function() {
  it("shouldn't change the array length or its elements", () => {
    let a = [22, 9, 60, 12, 4, 56];
    let old = JSON.stringify([...a].sort());
    shuffle(a);
    let new = JSON.stringify([...a].sort());
    expect(old).toBe(new);
  });
});
```

4.4. Breaking laws: Some of the properties are no longer always valid. To simplify our examples, let's assume two numbers are close to each other if they differ by no more than 0.1. If this is the case, then we have the following:

- 0.5 is close to 0.6, and 0.6 is close to 0.7, but 0.5 is not close to 0.7.
- 0.5 is close to 0.6, and 0.7 is close to 0.8, but $0.5+0.7=1.2$ is not close to $0.6+0.8=1.4$; with the same numbers, $0.5*0.7=0.35$ is not close to $0.6*0.8=0.48$.
- 0.5 is close to 0.4, and 0.2 is close to 0.3, but $0.5-0.2=0.3$ is not close to $0.4-0.3=0.1$.
- 0.6 is close to 0.5, and 0.9 is close to 1.0, but $0.6/0.9=0.667$ is not close to $0.5/1.0=0.5$.

The other cited properties are always true.

4.5. Must return? If a pure function doesn't return anything, it means that the function doesn't do anything since it can't modify its inputs or any other side effect.

4.6. JavaScript does math? If you run the code, you'll (unexpectedly) get the `Math failure?` message. The problem has to do with the fact that JavaScript internally uses binary instead of decimal, and floating-point precision is limited. In decimal, 0.1, 0.2, and 0.3 have a fixed, short representation, but in binary, they have infinite representation, much like $1/3=0.33333\dots$ has in decimal.

If you write out the value of `a+b` after the test, you'll get `0.30000000000000004` – and that's why you must be very careful when testing for equality in JavaScript.

Chapter 5, Programming Declaratively – a Better Style

5.1. Filtering... but what? `Boolean(x)` is the same as `!!x`, and it turns an expression from being *truthy* or *falsy* into `true` or `false`, respectively. Thus, the `.filter()` operation removes all falsy elements from the array.

5.2. Generating HTML code, with restrictions: In real life, you wouldn't limit yourself to using only `filter()`, `map()`, and `reduce()`, but the objective of this question was to make you think about how to manage with only those. Using `join()` or other extra string functions would make the problem easier. For instance, finding out a way to add the enclosing `<div> ... </div>` tags is tricky, so we had to make the first `reduce()` operation produce an array so that we could keep on working on it:

```
var characters = [
  { name: "Fred", plays: "bowling" },
  { name: "Barney", plays: "chess" },
  { name: "Wilma", plays: "bridge" },
  { name: "Betty", plays: "checkers" },
  { name: "Pebbles", plays: "chess" }
];

let list = characters
  .filter(x => x.plays === "chess" || x.plays === "checkers")
  .map(x => `- ${x.name}</li>`)
  .reduce((a, x) => [a[0] + x], [""])
  .map(x => `

<ul>${x}</ul></div>`)
  .reduce((a, x) => x);

console.log(list);
// <div><ul><li>Barney</li><li>Betty</li><li>Pebbles</li></ul></div>

```

Accessing the array and index arguments for the `map()` or `reduce()` callbacks would also provide solutions:

```
let list2 = characters
  .filter(x => x.plays === "chess" || x.plays === "checkers")
  .map(
    (x, i, t) =>
      `${i === 0 ? "<div><ul>" : ""}` +
      `- ${x.name}</li>` +

```



```
    `${i == t.length - 1 ? "</ul></div>" : ""}`  
  )  
  .reduce((a, x) => a + x, "");
```

We could also do the following:

```
let list3 = characters  
  .filter(x => x.plays === "chess" || x.plays == "checkers")  
  .map(x => `- ${x.name}</li>`)  
  .reduce(  
    (a, x, i, t) => a + x + (i === t.length - 1 ? "</ul></div>" : ""),  
    "<div><ul>"  
  );

```

Study the three examples: they will help you gain insight into these higher-order functions and provide you with ideas so that you can do independent work.

5.3. More formal testing: Use an idea from question 4.3: select an array and a function, find the result of mapping using both the standard `map()` method and the new `myMap()` function, and compare the two `JSON.stringify()` results: they should match.

5.4. Ranging far and wide: This requires a bit of careful arithmetic, but shouldn't be much trouble. Here, we need to distinguish two cases: upward and downward ranges. The default step is 1 for the former and -1 for the latter. We used `Math.sign()` for this:

```
const range2 = (start, stop, step = Math.sign(stop - start)) =>  
  new Array(Math.ceil((stop - start) / step))  
    .fill(0)  
    .map((v, i) => start + i * step);
```

A few examples of calculated ranges show the diversity in terms of the options we have:

```
console.log(range2(1, 10)); // [1, 2, 3, 4, 5, 6, 7, 8, 9]  
console.log(range2(1, 10, 2)); // [1, 3, 5, 7, 9]  
console.log(range2(1, 10, 3)); // [1, 4, 7]  
console.log(range2(1, 10, 6)); // [1, 7]  
console.log(range2(1, 10, 11)); // [1]  
  
console.log(range2(21, 10)); // [21, 20, 19, ... 13, 12, 11]  
console.log(range2(21, 10, -3)); // [21, 18, 15, 12]  
console.log(range2(21, 10, -4)); // [21, 17, 13]  
console.log(range2(21, 10, -7)); // [21, 14]  
console.log(range2(21, 10, -12)); // [21]
```

Using this new `range2()` function means that you can write a greater variety of loops in a functional way, with no need for `for(...)` statements.

5.5. Doing the alphabet: The problem is that `String.fromCharCode()` is not unary. This method may receive any number of arguments, and when you write `map(String.fromCharCode)`, the callback gets called with three parameters (the current value, the index, and the array) and that causes unexpected results. Using `unary()` from the *Arity Changing* section of Chapter 6, *Producing Functions – Higher-Order Functions*, would also work. To find out more, go to https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/String/fromCharCode.

5.6. Producing a CSV: A first solution, along with some auxiliary functions, is as follows; can you understand what each function does?

```
const concatNumbers = (a, b) => (a == " " ? b : a + "," + b);
const concatLines = (c, d) => c + "\n" + d;
const makeCSV = t =>
  t.reduce(concatLines, " ", t.map(f => f.reduce(concatNumbers, " ")));
```

An alternative one-liner is also possible, but not as clear – do you agree?

```
const makeCSV2 = t =>
  t.reduce(
    (c, d) => c + "\n" + d,
    " ",
    t.map(x => x.reduce((a, b) => (a == " " ? b : a + "," + b), " "))
  );
```

5.7 Producing better output: For this, you'll have to do some extra mapping, as follows:

```
const better = apiAnswer
  .flatMap(c => c.states.map(s => ({...s, country: c.name})))
  .flatMap(s => s.cities.map(t => ({...t, state: s.name, country:
s.country})))
  .map(t => `${t.name}, ${t.state}, ${t.country}`);

/*
[ 'Lincoln, Buenos Aires, Argentine',
  'Lincoln, England, Great Britain',
  'Lincoln, California, United States of America',
  'Lincoln, Rhode Island, United States of America',
  'Lincolnia, Virginia, United States of America',
  'Lincoln Park, Michigan, United States of America',
  'Lincoln, Nebraska, United States of America',
  'Lincoln Park, Illinois, United States of America',
  'Lincoln Square, Illinois, United States of America' ]
*/
```

5.8 Old-style code only! One way of doing this is by using `join()` to build a single long string out of the individual sentences, then using `split()` to split it into words, and finally looking at the length of the resulting array:

```
const words = gettysburg.join(" ").split(" ").length;
```

5.9 Async chaining: An article by Valeri Karpov, which can be found at <https://thecodebarbarian.com/basic-functional-programming-with-async-await.html>, provides polyfills for methods such as `forEach()`, `map()`, and so on, and also develops a class for async arrays that allows chaining.

5.10 Missing equivalents: Start by using `mapAsync()` to get the async values and apply the original function to the returned array. An example for `some()` would be as follows:

```
const someAsync = (arr, fn) =>
  mapAsync(arr, fn).then(mapped => mapped.some(Boolean));

(async () => {
  const someEven = await someAsync([1, 2, 3, 4], fakeFilter);
  useResult(someEven);

  const someEven2 = await someAsync([1, 3, 5, 7, 9], fakeFilter);
  useResult(someEven2);
})();
/*
2019-10-13T22:05:32.215Z true
2019-10-13T22:05:33.257Z false
*/
```

Chapter 6, Producing Functions – Higher-Order Functions

6.1. A border case: Just applying the function to a null object will throw an error:

```
const getField = attr => obj => obj[attr];

getField("someField")(null);
// Uncaught TypeError: Cannot read property 'a' of null
```

Having functions throw exceptions isn't usually good in FP. You may opt to produce `undefined` instead, or work with monads, just like we did in the last Chapter 12, *Building Better Containers – Functional Data Types* of this book. A safe version of `getField()` is as follows:

```
const getField2 = attr => obj => (attr && obj ? obj[attr] : undefined);
```

6.2. How many? Let's call $calc(n)$ the number of calls that are needed to evaluate $fib(n)$. Analyzing the tree that shows all the needed calculations, we get the following:

- $calc(0)=1$
- $calc(1)=1$
- For $n>1$, $calc(n)=1 + calc(n-1) + calc(n-2)$

The last line follows from the fact that when we call $fib(n)$, we have one call, plus calls to $fib(n-1)$ and $fib(n-2)$. A spreadsheet shows that $calc(50)$ is 40,730,022,147 – rather high!

If you care for some algebra, it can be shown that $calc(n)=5fib(n-1)+fib(n-4)-1$, or that as n grows, $calc(n)$ becomes approximately $(1+\sqrt{5})=3.236$ times the value of $fib(n)$ – but since this is not a math book, I won't even mention those results!

6.3. A randomizing balancer: Using our `shuffle()` function from Chapter 4, *Behaving Properly – Pure Functions*, we can write the following code. Here, we remove the first function from the list before shuffling the rest and we add it back at the end of the array to avoid repeating any calls:

```
const randomizer = (...fns) => (...args) => {
  const first = fns.shift();
  fns = shuffle(fns);
  fns.push(first);
  return fns[0](...args);
};
```

A quick verification shows it fulfills all our requirements:

```
const say1 = () => console.log(1);
const say22 = () => console.log(22);
const say333 = () => console.log(333);
const say4444 = () => console.log(4444);

const rrr = randomizer(say1, say22, say333, say4444);
rrr(); // 333
rrr(); // 4444
rrr(); // 333
rrr(); // 22
```

```
rrr(); // 333
rrr(); // 22
rrr(); // 333
rrr(); // 4444
rrr(); // 1
rrr(); // 4444
```

A small consideration: the first function in the list can never be called the first time around because of the way `randomizer()` is written. Can you provide a better version that won't have this small defect so that *all* the functions in the list will have the same chance of being called the first time?

6.4. Just say no! Call the original function and then use `typeof` to check whether the returned value is numeric or Boolean, before deciding what to return.

6.5. Missing companion: A simple one-line version could be as follows. Here, we use spreading to get a shallow copy of the original object and then set the specified attribute to its new value by using a computed property name. See https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/Object_initializer for more details:

```
const setField = (attr, value, obj) => ({...obj, [attr]: value});
```

In Chapter 10, *Ensuring Purity – Immutability*, we wrote `deepCopy()`, which would be better than spreading when it comes to creating a totally new object instead of a shallow copy. By using this, we would have the following:

```
const setField = (attr, value, obj) => ({...deepCopy(obj), [attr]: value});
```

Finally, you could also look into modifying the `updateObject()` function, also from Chapter 10, *Ensuring Purity – Immutability*, by removing the freezing code; I'll leave it up to you.

6.6. Wrong function length: We can solve this problem by using `eval()` – which, in general, isn't such a good idea! If you persist and insist, though, we can write a `function.length` preserving version of `arity()` as follows; let's call it `arityL()`:

```
const arityL = (fn, n) => {
  const args1n = range(0, n)
    .map(i => `x${i}`)
    .join(",");
  return eval(`(${args1n}) => ${fn.name}(${args1n})`);
};
```

If you were to apply `arityL()` to `Number.parseInt`, the results would be as follows (note that the produced functions have the correct `length` property):

```
const parseInt1 = arityL(parseInt, 1);
/*
  (x0) => parseInt(x0,x1)
  parseInt1.length === 1
*/

const parseInt2 = arity(Number.parseInt, 2)
/*
  (x0, x1) => parseInt(x0, x1)
  parseInt2.length === 2
*/
```

6.7. **Not reinventing the wheel:** We can use `Math.max()` and `Math.min()` as follows:

```
const findMaximum2 = findOptimum2((x, y) => Math.max(x, y));
const findMinimum2 = findOptimum2((x, y) => Math.min(x, y));
```

Another way of writing this could be achieved by defining the following first:

```
const max = (...args) => Math.max(...arr);
const min = (...args) => Math.min(...arr);
```

Then, we could write in point-free style:

```
const findMaximum3 = findOptimum2(max);
const findMinimum3 = findOptimum2(min);
```

Chapter 7, Transforming Functions – Currying and Partial Application

7.1. **Sum as you will:** The following `sumMany()` function does the job:

```
const sumMany = total => number =>
  number === undefined ? total : sumMany(total + number);

sumMany(2)(2)(9)(6)(0)(-3)(); // 16
```

7.2. **Working stylishly:** We can do currying by hand for `applyStyle()`:

```
const applyStyle = style => string => `<${style}>${string}</${style}>`;
```

7.3. Currying by prototype: Basically, we are just transforming the `curryByBind()` version so that it uses `this`:

```
Function.prototype.curry = function() {
  return this.length === 0 ? this() : p => this.bind(this, p).curry();
};
```

You could work in a similar fashion and provide a `partial()` method instead.

7.4. Uncurrying the currying: We can work in a similar fashion to what we did in `curryByEval()`:

```
const uncurryByEval = (fn, len) =>
  eval(
    `(${range(0, len)
      .map(i => `x${i}`)
      .join(",")} ) => ${fn.name}${range(0, len)
      .map(i => `(x${i})` )
      .join("")}`
  );
```

Earlier, when currying, given an `fn()` function of arity 3, we would have generated the following:

```
x0=>x1=>x2=> make3(x0,x1,x2)
```

Now, to uncurry a function (say, `curriedFn()`), we want to do something very similar: the only difference is the placement of the parentheses:

```
(x0, x1, x2) => curriedFn(x0)(x1)(x2)
```

The expected behavior is as follows:

```
const make3 = (a, b, c) => String(100 * a + 10 * b + c);
const make3c = a => b => c => make3(a, b, c);
console.log(make3c(1)(2)(3)); // 123

const remake3 = uncurryByEval(make3c, 3);
console.log(remake3(4, 5, 6)); // 456
```

7.5. Mystery questions function: It implements partial currying; the following is an example of this:

```
const sum3 = (a, b, c) => 100 * a + 10 * b + c;
const alt3 = what(sum3);

console.log(alt3(1, 2, 4));
console.log(alt3(1, 2)(4));
```

```
console.log(alt3(1)(2, 4));
console.log(alt3(1)(2)(4));
/*
  "124", four times
*/
```

A more understandable and better-named version of the `what()` function is as follows:

```
const partial = fn => (...params) =>
  fn.length <= params.length
  ? fn(...params)
  : (...otherParams) => partial(fn)(...params, ...otherParams);
```

7.6. Yet more curry! This is just an alternative version of our `partialCurryingByBind()`. The only difference is that if you provide all the arguments to a function, this new `curry()` directly calls the curried function, while `partialCurryingByBind()` would bind the function to all its arguments first and then recursively call it to return the final result. We can check that the results are exactly the same by using the following code:

```
const make3 = (a, b, c) => String(100 * a + 10 * b + c);

const make3curried = curry(make3);

console.log(make3curried(1)(2)(3));
console.log(make3curried(4, 5)(6));
console.log(make3curried(7, 8, 9));

/*
123
456
789
*/
```

Chapter 8, Connecting Functions – Pipelining and Composition

8.1. Headline capitalization: We can make use of several functional equivalents of different methods, such as `split()`, `map()`, and `join()`. Using `demethodize()` from Chapter 6, *Producing Functions – Higher-Order Functions*, and `flipTwo()` from Chapter 7, *Transforming Functions – Currying and Partial Application*, would have also been possible:

```
const split = str => arr => arr.split(str);
const map = fn => arr => arr.map(fn);
const firstToUpper = word =>
```



```
word[0].toUpperCase() + word.substr(1).toLowerCase();
const join = str => arr => arr.join(str);

const headline = pipeline(split(" "), map(firstToUpper), join(" "));
```

The pipeline works as expected: we split the string into words, we map each word to make its first letter uppercase, and we join the array elements to form a string again. We could have used `reduce()` for the last step, but `join()` already does what we need, so why reinvent the wheel?

```
console.log(headline("Alice's ADVENTURES in WoNdErLaNd"));
// Alice's Adventures In Wonderland
```

8.2. Pending tasks: The following pipeline does the job:

```
const getField = attr => obj => obj[attr];
const filter = fn => arr => arr.filter(fn);
const map = fn => arr => arr.map(fn);
const reduce = (fn, init) => arr => arr.reduce(fn, init);

const pending = (listOfTasks, name) =>
  pipeline(
    getField("byPerson"),
    filter(t => t.responsible === name),
    map(t => t.tasks),
    reduce((y, x) => x, []),
    filter(t => t && !t.done),
    map(getField("id"))
  )(allTasks || {byPerson: []}); //
```

The `reduce()` call may be mystifying. By that time, we are handling an array with a single element – an object – and we want the object in the pipeline, not the array. This code works even if the responsible person doesn't exist, or if all the tasks have been completed; can you see why? Also, note that if `allTasks` is `null`, an object must be provided with the `byPerson` property so that future functions won't crash! For an even better solution, I think monads are better: see question 12.1 for more.

8.3. Thinking in abstract terms: The simple solution implies composing. I preferred it to pipelining in order to keep the list of functions in the same order:

```
const getSomeResults2 = compose(sort, group, filter, select);
```

8.4 Undetected impurity? Yes, the function is impure, but using it as-is would fall squarely under the SFP **Sorta Functional Programming (SFP)** style we mentioned back in the *Theory versus practice* section of Chapter 1, *Becoming Functional – Several Questions*. The version we used is not pure, but in the way we use it, the final results are pure: we modify an array in place, but it's a new array that we are creating. The alternate implementation is pure and also works, but will be slower since it creates a completely new array every time we call it. So, accepting this bit of impurity helps us get a function that performs better; we can accept that!

8.5 Needless transducing? If you only had a sequence of `map()` operations, you could apply a single `map` and pipeline all the mapping functions into a single one. For `filter()` operations, it becomes a bit harder, but here's a tip: use `reduce()` to apply all the filters in sequence with a carefully thought out accumulating function.

Chapter 9, Designing Functions – Recursion

9.1. Into reverse: An empty string is reversed by simply doing nothing. To reverse a non-empty string, remove its first character, reverse the rest, and append the removed character at the end. For example, `reverse("MONTEVIDEO")` can be found by using `reverse("ONTEVIDEO")+"M"`. In the same way, `reverse("ONTEVIDEO")` would be equal to `reverse("NTEVIDEO")+"O"`, and so on:

```
const reverse = str =>
  str.length === 0 ? "" : reverse(str.slice(1)) + str[0];
```

9.2. Climbing steps: Suppose we want to climb a ladder with n steps. We can do this in two ways:

- Climbing one single step and then climbing an $(n-1)$ steps ladder
- Climbing two steps at once and then climbing an $(n-2)$ steps ladder

So, if we call $ladder(n)$ the number of ways to climb n steps ladder, we know that $ladder(n) = ladder(n-1) + ladder(n-2)$. Adding the fact that $ladder(0)=1$ (there's only one way to climb a ladder with no steps: do nothing) and $ladder(1)=1$, the solution is that $ladder(n)$ equals the $(n-1)^{th}$ Fibonacci number! Check it out: $ladder(2)=2$, $ladder(3)=3$, $ladder(4)=5$, and so on.

9.3. Longest common subsequence: The length of the **longest common sequence (LCS)** of two strings, *a* and *b*, can be found with recursion, as follows:

- If the length of *a* is zero, or if the length of *b* is zero, return zero.
- If the first characters of *a* and *b* match, the answer is 1 plus the LCS of *a* and *b*, both minus their initial characters.
- If the first characters of *a* and *b* do not match, the answer is the largest of the following two results:
 - The LCS of *a* minus its initial character, and *b*
 - The LCS of *a*, and *b* minus its initial character

We can implement this as follows. We do memoization "by hand" to avoid repeating calculations; we could have also used our memoization function:

```
const LCS = (strA, strB) => {
  let cache = {}; // memoization "by hand"

  const innerLCS = (strA, strB) => {
    const key = strA + "/" + strB;

    if (!(key in cache)) {
      if (strA.length === 0 || strB.length === 0) {
        ret = 0;

      } else if (strA[0] === strB[0]) {
        ret = 1 + innerLCS(strA.substr(1), strB.substr(1));

      } else {
        ret = Math.max(
          innerLCS(strA, strB.substr(1)),
          innerLCS(strA.substr(1), strB)
        );
      }

      cache[key] = ret;
    }

    return cache[key];
  };

  return innerLCS(strA, strB);
};

console.log(LCS("INTERNATIONAL", "CONTRACTOR")); // 6, as in the text
```

As an extra exercise, you could try to produce not only the length of the LCS but also the characters that are involved.

9.4. Symmetrical queens: The key to finding only symmetric solutions is as follows. After the first four queens have been (tentatively) placed on the first half of the board, we don't have to try all the possible positions for the other queens; they are automatically determined with regard to the first ones:

```
const SIZE = 8;
let places = Array(SIZE);
const checkPlace = (column, row) =>
  places
    .slice(0, column)
    .every((v, i) => v !== row && Math.abs(v - row) !== column - i);

const symmetricFinder = (column = 0) => {
  if (column === SIZE) {
    console.log(places.map(x => x + 1)); // print out solution
  } else if (column <= SIZE / 2) {
    // first half of the board?
    const testRowsInColumn = j => {
      if (j < SIZE) {
        if (checkPlace(column, j)) {
          places[column] = j;
          symmetricFinder(column + 1);
        }
        testRowsInColumn(j + 1);
      }
    };
    testRowsInColumn(0);
  } else {
    // second half of the board
    let symmetric = SIZE - 1 - places[SIZE - 1 - column];
    if (checkPlace(column, symmetric)) {
      places[column] = symmetric;
      symmetricFinder(column + 1);
    }
  }
};
```

Calling `symmetricFinder()` produces four solutions, which are essentially the same. Make drawings and check it to make sure it's correct!

```
[3, 5, 2, 8, 1, 7, 4, 6]
[4, 6, 8, 2, 7, 1, 3, 5]
[5, 3, 1, 7, 2, 8, 6, 4]
[6, 4, 7, 1, 8, 2, 5, 3]
```

9.5. Sorting recursively: Let's look at the first of these algorithms; many of the techniques here will help you write the other sorts. If the array is empty, sorting it produces a (new) empty array. Otherwise, we find the maximum value of the array (`max`), create a new copy of the array but without that element, sort the copy, and then return the sorted copy with `max` added at the end. Take a look at how we dealt with the mutator functions in order to avoid modifying the original string:

```
const selectionSort = arr => {
  if (arr.length === 0) {
    return [];
  } else {
    const max = Math.max(...arr);
    const rest = [...arr];
    rest.splice(arr.indexOf(max), 1);
    return [...selectionSort(rest), max];
  }
};

selectionSort([2, 2, 0, 9, 1, 9, 6, 0]);
// [0, 0, 1, 2, 2, 6, 9, 9]
```

9.6. What could go wrong? This would fail if, at any time, the array (or sub-array) to be sorted consisted of all equal values. In that case, `smaller` would be an empty array and `greaterEqual` would be equal to the whole array to sort, so the logic would enter an infinite loop.

9.7. More efficiency: The following code does the work for us. Here, we use a ternary operator to decide where to push the new item:

```
const partition = (arr, fn) =>
  arr.reduce(
    (result, elem) => {
      result[fn(elem) ? 0 : 1].push(elem);
      return result;
    },
    [[], []]
  );
```

Chapter 10, Ensuring Purity – Immutability

10.1. Freezing by proxying: As requested, using a proxy allows you to intercept changes on an object. (See https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Proxy for more on this.) We use recursion to apply the proxy *all the way down* in case some attributes are objects themselves:

```
const proxySetAll = obj => {
  Object.keys(obj).forEach(v => {
    if (typeof obj[v] === "object") {
      obj[v] = proxySetAll(obj[v]);
    }
  });
};

return new Proxy(obj, {
  set(target, key, value) {
    throw new Error("DON'T MODIFY ANYTHING IN ME");
  },
  deleteProperty(target, key) {
    throw new Error("DON'T DELETE ANYTHING IN ME");
  }
});
};
```

The following is the output of the preceding code. You'd probably require something other than a DON'T MODIFY ANYTHING IN ME message, of course!

```
let myObj = {a: 5, b: 6, c: {d: 7, e: 8}};
myObj = proxySetAll(myObj);

myObj.a = 777; // Uncaught Error: DON'T MODIFY ANYTHING IN ME
myObj.f = 888; // Uncaught Error: DON'T MODIFY ANYTHING IN ME
delete myObj.b; // Uncaught Error: DON'T DELETE ANYTHING IN ME
```

10.2. Inserting into a list, persistently: Using recursion helps out:

- If the list is empty, we cannot insert the new key.
- If we are at a node and its key isn't `oldKey`, we create a clone of the node and insert the new key somewhere in the rest of the original node's list.

- If we are at a node and its key is `oldKey`, we create a clone of the node that's pointing at a list that starts with a new node, with `newKey` as its value, and itself pointing to the rest of the original node's list:

```
const insertAfter = (list, newKey, oldKey) => {
  if (list === null) {
    return null;

  } else if (list.key !== oldKey) {
    return node(list.key, insertAfter(list.next, newKey, oldKey));

  } else {
    return node(list.key, node(newKey, list.next));
  }
};
```

In the following code, we can see this working. The new list is similar to the one shown in *Figure 10.2*. However, printing out the lists (`c3` and `newList`) wouldn't be enough; you wouldn't be able to recognize the new or old nodes from doing this, so I've included several comparisons. The following last comparison shows that from the "F" node onward, the list is the same:

```
class Node {
  constructor(key, next = null) {
    this.key = key;
    this.next = next;
  }
}

const node = (key, next) => new Node(key, next);

let c3 = node("G", node("B", node("F", node("A", node("C", node("E")))))));
let newList = insertAfter(c3, "D", "B");

c3 === newList // false
c3.key === newList.key // true (both are "G")
c3.next === newList.next // false

c3.next.key === newList.next.key // true (both are "B")
c3.next.next === newList.next.next // false

c3.next.next.key === "F" // true
newList.next.next.key === "D" // true
c3.next.next.next === newList.next.next.next.next // true
```

When we implement this, if `oldKey` isn't found, nothing is inserted. Could you change the logic so that the new node would be added at the end of the list?

10.3. Composing many lenses: We want to compose lenses from left to right so that we can use `reduce()` in a direct way. Let's write the `composeManyLenses()` function and apply it to the same example that was shown in the text:

```
const composeManyLenses = (...lenses) =>
  lenses.reduce((acc, lens) => composeTwoLenses(acc, lens));

console.log(view(composeManyLenses(1C, 1E, 1G, 1J, 1K), deepObject));
/*
  11, same as earlier
*/
```

10.4. Lenses by path: Hint: the needed changes would be along the lines of what we did when we went from `getField()` to `getByPath()`.

10.5. Accessing virtual attributes: Using a getter is always viable, and for this question, you'd write something like the following:

```
const lastNameLens = composeTwoLenses(lensProp("name"), lensProp("last"));

const fullNameGetter = obj => `${view(lastNameLens)(obj)},
${view(firstNameLens)(obj)}`;
```

Being able to set several attributes based on a single value isn't always possible, but if we assume the incoming full name is in the right format, we can split it by the comma and assign the two parts to first and last name, respectively:

```
const fullNameSetter = (fullName, obj) => {
  const parts = fullName.split(",");
  return set(firstNameLens, parts[1], set(lastNameLens, parts[0], obj));
};
```

10.6. Lenses for arrays? The `view()` function would work well, but `set()` and `over()` wouldn't work in a pure way since `setArray()` doesn't return a new array; instead, it modifies the current one in place. Take a look at the next question for a related problem.

10.7. Lenses into maps: Getting a value from the map poses no problem, but for setting, we need to clone the map:

```
const getMap = curry((key, map) => map.get(key));

const setMap = curry((key, value, map) => new Map(map).set(key, value));

const lensMap = key => lens(getMap(key), setMap(key));
```


Chapter 11, Implementing Design Patterns – the Functional Way

11.1. Decorating methods, the future way: As we've already mentioned, decorators aren't a fixed, definitive feature at the moment. However, by following <https://tc39.github.io/proposal-decorators/>, we can write the following:

```
const logging = (target, name, descriptor) => {
  const savedMethod = descriptor.value;
  descriptor.value = function(...args) {
    console.log(`entering ${name}: ${args}`);

    try {
      const valueToReturn = savedMethod.bind(this)(...args);
      console.log(`exiting ${name}: ${valueToReturn}`);
      return valueToReturn;

    } catch (thrownError) {
      console.log(`exiting ${name}: threw ${thrownError}`);
      throw thrownError;
    }
  };

  return descriptor;
};
```

We want to add a `@logging` decoration to a method. We save the original method in `savedMethod` and substitute a new method that will log the received arguments, call the original method to save its return value, log that, and finally return it. If the original method throws an exception, we catch it, report it, and throw it again so that it can be processed as expected. A simple example of this is as follows:

```
class SumThree {
  constructor(z) {
    this.z = z;
  }
  @logging
  sum(x, y) {
    return x + y + this.z;
  }
}

new SumThree(100).sum(20, 8);
// entering sum: 20,8
// exiting sum: 128
```

11.2. Decorator with mixins: Working along the same lines as in question 1.1, we write an `addBar()` function that receives a `Base` class and extends it. In this case, I decided to add a new attribute and a new method. The constructor for the extended class calls the original constructor and creates the `.barValue` attribute. The new class has both the original's `doSomething()` method and the new `somethingElse()` method:

```
class Foo {
  constructor(fooValue) {
    this.fooValue = fooValue;
  }

  doSomething() {
    console.log("something: foo...", this.fooValue);
  }
}

var addBar = Base =>
  class extends Base {
    constructor(fooValue, barValue) {
      super(fooValue);
      this.barValue = barValue;
    }

    somethingElse() {
      console.log("something added: bar... ", this.barValue);
    }
  };

var fooBar = new (addBar(Foo))(22, 9);
fooBar.doSomething(); // something: foo... 22
fooBar.somethingElse(); // something added: bar... 9
```

11.3. Multi-clicking by hand: There are various ways to achieve this with timers and counting, but make sure that you don't interfere with single- or double-click detection! You can also use a common listener and look at `event.detail`; you can find out more at <https://developer.mozilla.org/en-US/docs/Web/API/UIEvent/detail>.

Chapter 12, Building Better Containers – Functional Data Types

12.1. **Maybe tasks?** The following code shows a simpler solution than the one we looked at in question 8.2:

```
const pending = Maybe.of(listOfTasks)
  .map(getField("byPerson"))
  .map(filter(t => t.responsible === name))
  .map(t => tasks)
  .map(t => t[0])
  .map(filter(t => !t.done))
  .map(getField("id"))
  .valueOf();
```

Here, we apply one function after the other, secure in the knowledge that if any of these functions produces an empty result (or even if the original `listOfTasks` is null), the sequence of calls will go on. In the end, you will either get an array of task IDs or a null value.

12.2. **Extending your trees:** Calculating the tree's height is simple if you do this in a recursive fashion. The height of an empty tree is zero, while the height of a non-empty tree is one (for the root) plus the maximum height of its left and right subtrees:

```
const treeHeight = tree =>
  tree(
    (val, left, right) =>
      1 + Math.max(treeHeight(left), treeHeight(right)),
    () => 0
  );
```

Listing the keys in order is a well-known requirement. Because of the way that the tree is built, you list the left subtree's keys first, then the root, and finally the right subtree's keys, all in a recursive fashion:

```
const treeList = tree =>
  tree(
    (value, left, right) => {
      treeList(left);
      console.log(value);
      treeList(right);
    },
    () => {
      // nothing
    }
  );
```

Finally, deleting a key from a binary search tree is a bit more complex. First, you must locate the node that is going to be removed, and then there are several cases:

- If the node has no subtrees, deletion is simple.
- If the node has only one subtree, you just replace the node by its subtree.
- If the node has two subtrees, then you have to do the following:
 - Find the minimum key in the tree with a greater key.
 - Place it in the node's place.

Since this algorithm is well covered in all computer science textbooks, I won't go into more detail about this here:

```
const treeRemove = (toRemove, tree) =>
  tree(
    (val, left, right) => {
      const findMinimumAndRemove = (tree /* never empty */) =>
        tree((value, left, right) => {
          if (treeIsEmpty(left)) {
            return { min: value, tree: right };

          } else {
            const result = findMinimumAndRemove(left);
            return {
              min: result.min,
              tree: Tree(value, result.tree, right)
            };
          }
        });

      if (toRemove < val) {
        return Tree(val, treeRemove(toRemove, left), right);

      } else if (toRemove > val) {
        return Tree(val, left, treeRemove(toRemove, right));

      } else if (treeIsEmpty(left) && treeIsEmpty(right)) {
        return EmptyTree();

      } else if (treeIsEmpty(left) !== treeIsEmpty(right)) {
        return tree((val, left, right) => left(() => left, () => right));

      } else {
        const result = findMinimumAndRemove(right);
        return Tree(result.min, left, result.tree);
      }
    },
```

```
    () => tree
  );
```

12.3. Functional lists: Let's add to the samples that have already been provided. We can simplify working with lists if we can transform a list into an array and vice versa:

```
const listToArray = list =>
  list((head, tail) => [head, ...listToArray(tail)], () => []);

const listFromArray = arr =>
  arr.length
    ? NewList(arr[0], listFromArray(arr.slice(1)))
    : EmptyList();
```

Concatenating two lists together and appending a value to a list have simple recursive implementations. We can also reverse a list by using the appending function:

```
const listConcat = (list1, list2) =>
  list1(
    (head, tail) => NewList(head, listConcat(tail, list2)),
    () => list2
  );

const listAppend = value => list =>
  list(
    (head, tail) => NewList(head, listAppend(value)(tail)),
    () => NewList(value, EmptyList)
  );

const listReverse = list =>
  list(
    (head, tail) => listAppend(head)(listReverse(tail)),
    () => EmptyList
  );
```

Finally, the basic `map()`, `filter()`, and `reduce()` operations are good to have:

```
const listMap = fn => list =>
  list(
    (head, tail) => NewList(fn(head), listMap(fn)(tail)),
    () => EmptyList
  );

const listFilter = fn => list =>
  list(
    (head, tail) =>
      fn(head)
        ? NewList(head, listFilter(fn)(tail))
```

```
      : listFilter(fn) (tail),
    () => EmptyList
  );

const listReduce = (fn, accum) => list =>
  list((head, tail) => listReduce(fn, fn(accum, head))(tail), () => accum);
```

The following are some exercises that have been left for you to tackle:

- Generate a printable version of a list.
- Compare two lists to see if they have the same values, in the same order.
- Search a list for a value.
- Get, update, or remove the value at the n -th position of a list.

12.4. Shortening code: The first thing you would do is get rid of the first ternary operator by taking advantage of the short circuit evaluation of the `||` operator:

```
const treeSearch2 = (findValue, tree) =>
  tree(
    (value, left, right) =>
      findValue === value ||
      (findValue < value
        ? treeSearch2(findValue, left)
        : treeSearch2(findValue, right)),
    () => false
  );
```

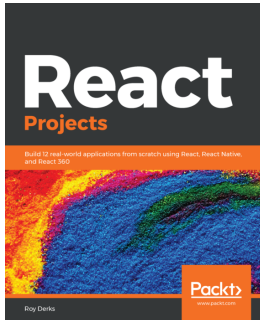
Also, seeing that both alternatives in the second ternary operator are very similar, you could also do some shortening there:

```
const treeSearch3 = (findValue, tree) =>
  tree(
    (value, left, right) =>
      findValue === value ||
      treeSearch3(findValue, findValue < value ? left : right),
    () => false
  );
```

Remember: shorter doesn't imply better! However, I've found many examples of this kind of code tightening, and it's better if you have been exposed to it, too.

Other Books You May Enjoy

If you enjoyed this book, you may be interested in these other books by Packt:

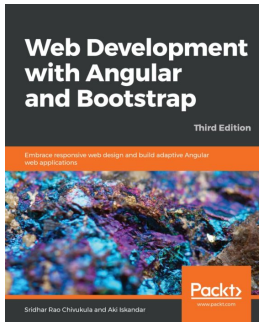


React Projects

Roy Derks

ISBN: 978-1-78995-493-7

- Create a wide range of applications using various modern React tools and frameworks
- Discover how React Hooks modernize state management for React apps
- Develop progressive web applications using React components
- Build test-driven React applications using the Jest and Enzyme frameworks
- Understand full stack development using React, Apollo, and GraphQL
- Perform server-side rendering using React and React Router
- Design gestures and animations for a cross-platform game using React Native



Web Development with Angular and Bootstrap - Third Edition

Sridhar Rao Chivukula, Aki Iskandar

ISBN: 978-1-78883-810-8

- Develop Angular single-page applications using an ecosystem of helper tools
- Get familiar with Bootstrap's new grid and helper classes
- Embrace TypeScript and ECMAScript to write more maintainable code
- Implement custom directives for Bootstrap 4 with the ng2-bootstrap library
- Understand the component-oriented structure of Angular and its router
- Use the built-in HTTP library to work with API endpoints
- Manage your app's data and state with observables and streams
- Combine Angular and Bootstrap 4 with Firebase to develop a solid example

Leave a review - let other readers know what you think

Please share your thoughts on this book with others by leaving a review on the site that you bought it from. If you purchased the book from Amazon, please leave us an honest review on this book's Amazon page. This is vital so that other potential readers can see and use your unbiased opinion to make purchasing decisions, we can understand what our customers think about our products, and our authors can see your feedback on the title that they have worked with Packt to create. It will only take a few minutes of your time, but is valuable to other potential customers, our authors, and Packt. Thank you!

Index

▪

- `.charCodeAt()` method
 - reference link 81
- `.filter()` method
 - URL 131
- `.forEach()`
 - URL 128
- `.map()`
 - URL 114, 268

A

- Adapter patterns 335, 336, 337, 338
- Ajax
 - detecting 64, 65
- altered functions 147
- anonymous 19
- arguments
 - reference link 51
- `Array.from()`
 - reference link 238
- `Array.prototype.sort()`
 - URL 368
- array
 - `.reduce()`, using 111, 112
 - average, calculating 108, 109, 110
 - filtering 130, 131
 - `find()`, emulating with `reduce()` 134
 - `findIndex()`, emulating with `reduce()` 134
 - flattening 120, 121, 122, 123
 - reducing, to value 105, 106
 - reference link 107
 - searching 133
 - special search case 134
 - summing 107, 108
 - values, calculating 110, 111
- arrays of arrays

- dealing with 120
- arrow functions
 - about 19, 20, 48
 - multiple arguments 53, 54
 - using 198
 - values, handling 49, 51
 - values, returning 49
 - working, with arguments 51, 52
- async calls
 - filtering with 142
 - looping over 140, 141
 - mapping 141
 - reducing 143, 144
- async functions
 - strange behaviors 138, 139, 140
 - working with 137, 138
- async-ready
 - looping 140

B

- Babel
 - URL 23
- backtracking 274
- basic container 372, 373, 374
- Bell System Technical Journal
 - URL 217
- binary 107
- binary trees
 - using, in Haskell 393, 394
- `bind()` method
 - using, for currying 191, 192, 194
- `bind()`
 - reference link 59
- black-box testing 99

C

- callback 17, 62
- chaining
 - about 216, 231
 - method calls 232, 233, 234
- cloning 301, 302, 303, 304
- closures
 - about 18, 19
 - using, in partial application 202, 203
 - using, in partial currying 209
- CodePen
 - URL 25
- Colossal Cave Adventure Game
 - reference link 55
- comma operator
 - URL 225, 226
- command line interface (CLI) 24
- Command patterns 335, 344, 345
- commutative property 84
- compatibility table, ES6
 - reference link 22
- compatibility tables
 - URL 283
- complex memoization 160, 161, 162
- composing
 - about 216, 235
 - composed function, testing 243, 244, 245, 246, 247
 - examples 235
 - files, counting 237
 - unary operators 236
 - unique words, searching 237, 238, 239
 - with higher-order functions 239, 240, 241, 242
- constants 298, 299
- containers
 - about 372
 - API results, dealing with 378, 380, 381, 382
 - building 369, 370
 - data types, extending 370, 371, 372
 - enhancing 374, 375, 376
 - missing values, dealing with 376, 377
 - monads 384
 - prisms, implementing 382, 384
- continuation-passing style (CPS) 63, 64, 285, 286, 287, 288, 289

- continuations 62, 285
- currying
 - about 53, 184, 185, 186
 - curried-by-hand function 189, 190
 - parameters 186, 187, 188, 189
 - with bind() 191, 192, 194
 - with eval() 194, 195

D

- D3.js library
 - URL 231
- Dark Sky API
 - URL 378
- data types
 - about 364, 365
 - extending 370, 371, 372
 - options 367, 369
- data
 - extracting, from objects 115, 116
- debugging 216
- decorator patterns 335, 338, 339, 340, 342
- decorators
 - reference link 148
- demethodizing 178, 179, 180
- design patterns
 - about 332, 333
 - architectural patterns 333
 - behavioral design patterns 333
 - categories 333, 334
 - concurrency patterns 333
 - creational design patterns 333
 - need for 334, 335
 - structural design patterns 333
- destructuring assignment
 - reference link 91
- Don't Repeat Yourself (DRY) 35
- dyadic function 107

E

- e-commerce, related issues 29, 30
- eight queens puzzle 274, 275, 276, 277, 278
- Either monad 388, 389, 391
- ES6
 - reference link 22
- eta abstraction 57

- eta conversion 57
- eta reduction 57
- European Computer Manufacturers Association (ECMA)
 - about 15
 - URL 15
- eval() method
 - using 198
 - using, for currying 194, 195

F

- Facade patterns 335, 336, 337, 338
- Facebook's Flow
 - URL 25
- Fantasy Land
 - URL 384
- fetch()
 - reference link 67
 - references 196
- Fibonacci
 - reference link 85
- first-class objects
 - about 16, 54
 - functions, using as 16
- Fisher-Yates shuffle
 - algorithm, reference link 101
 - reference link 101
- flat()
 - emulating 125, 126, 127
- flatMap()
 - emulating 125, 126, 127
 - flattening 123, 124, 125
 - mapping 123, 124, 125
- flattening 120
- Flow
 - URL 365
- fluent APIs
 - examples 231, 232
- fluent interfaces 231
- foldr 106
- freezing 299, 301
- functional design patterns 359, 360
- functional programming (FP)
 - about 8, 9, 73, 104
 - characteristics 12
 - disadvantages 13
 - extensible functionality 12
 - misconceptions 10, 11
 - modular functionality 11
 - need for 11
 - program, customizing 10
 - qualities 11
 - reusable functionality 12
 - testable functionality 11
 - theoretical way, versus practical way 9
 - understandable functionality 11
- Functional Reactive Programming (FRP)
 - about 347
 - basic concepts and terms 347, 348
 - multi-clicks, detecting 352
 - observable 347
 - observer 347
 - operators 347
 - operators, for observables 349, 351
 - pipeline 347
 - subscription 347
 - typeahead searches 354, 355, 356, 357
- functional solution, e-commerce related issues
 - about 35
 - higher-order solution 36, 37
 - solution, producing 41, 42
 - solution, testing automatically 39, 40, 41
 - solution, testing manually 37, 38
- functions, using in FP ways
 - about 59
 - callbacks 62
 - continuation-passing style (CPS) 63, 64
 - continuations 62
 - immediate invocation 68, 69, 70, 71
 - injection 59, 60, 61
 - polyfills 64
 - promises 62
 - stubbing 67, 68
- functions, with side effects 75
- functions
 - about 44
 - arity changing 171, 172
 - arrow functions 48
 - behavior, altering 165
 - common mistake 57

- example 165, 167
- lambda function 45, 46, 47, 48
- logging, adding to 149
- methods, turning into 178, 179, 180
- modifying 172
- negating, logically 168, 169
- operations, turning into 172, 173
- optimum value, finding 180, 181, 182
- React-Redux reducer 55, 56
- references 46
- results, inverting 169, 170
- signatures 365, 366, 367
- turning, into promises 175, 176
- types, URL 365
- URL 48
- used, as objects 54, 55
- using 212
- using, as binary trees 395, 396, 397, 398, 399, 400
- using, as data structures 393
- using, as first-class objects 16
- working, with methods 58, 59

functors 372, 374, 375, 376

G

- Gang of Four (GoF) 332
- getRandomLetter() function
 - reference link 81
- getters 305
 - writing 305, 306

H

- Haskell
 - binary trees, using 393, 394
- higher-order functions (HOF)
 - about 104, 267, 271, 272, 273
 - composing with 239, 240, 241, 242
- Hindley-Milner 365
- hoisting
 - reference link 46

I

- idempotency 74
- immediate invocation 68, 69, 70, 71
- Immediately Invoked Function Expression (IIFE)

- 34
- immutable objects
 - references 327
- impure functions
 - about 89
 - avoiding 89
 - injecting 91, 92, 93
 - purity, ensuring 93, 94
 - state usage, avoiding 89, 91
 - testing 99, 100, 101

J

- Jasmine
 - URL 26
- JavaScript (JS)
 - about 8, 297
 - arrow functions 19, 20
 - cloning 301, 302, 303, 304
 - closures 18, 19
 - constants 298, 299
 - features 16
 - freezing 299, 301
 - functionalities 15, 16
 - functions, using as first-class objects 16, 17
 - lenses 308
 - mutating 301, 302, 303, 304
 - mutator functions 297, 298
 - prisms 317
 - recursion 17, 18
 - spread operator 20, 21
 - testing 26
 - transpilers, using 23, 24, 25
 - using, as functional language 13
 - using, as tool 14, 15
 - working with 22
 - working, with online tools 25, 26
- JavaScript functions
 - reference link 172
- JavaScript, getters
 - about 305
 - writing 305, 306
- JavaScript, setters
 - about 305
 - creating 306, 308
- JS prettier

- reference link 20

JSBin

- URL 25

JSFiddle

- reference link 25

K

Karma

- URL 26

L

lambda function 45, 46, 47, 48

lazy evaluation 289

lenses

- about 308
- implementing, with functions 314, 315, 317
- implementing, with objects 311, 312, 313
- working with 308, 309, 311

Linux

- pipelining, using in 217, 218, 219

lists

- working with 321, 322, 323

localeCompare()

- reference link 61

logging function

- enhancing 151

logging

- adding, to function 149
- in functional way 149, 150

logical higher-order functions

- about 129
- array, filtering 130, 131
- array, searching 133
- filter(), emulating with reduce() 132
- higher-level predicates 135, 136
- negatives, checking 136, 137
- reduce() example 131, 132

looping 127, 129

M

map()

- about 113, 114
- emulating, with reduce() 119
- using, advantages 114

memoization 84

memoization higher-order function

- testing 162, 163, 165

memoizing functions 157

methods

- turning, into functions 178, 179, 180

Mocha

- URL 26

monads

- about 384
- function, calling 391, 392
- operations, adding 385, 386, 387, 388
- promises 392, 393

morphism 113

mutating 301, 302, 303, 304

mutator functions 298

mutator methods

- about 80, 297
- URL 297

N

Node.js

- reference link 22

numbers

- parsing 116, 117

O

Object Oriented Programming (OOP) 9

object-oriented (OO) 346

object-oriented design patterns

- about 335, 358
- Adapter patterns 336, 337, 338
- Chain of Responsibility pattern 358
- Command patterns 344, 345
- currying and partial application 358
- declarative functions 358
- decorator patterns 338, 339, 340, 342
- Facade patterns 336, 337, 338
- observer programming 346
- persistent data structures 358
- reactive programming 346
- Strategy patterns 344, 345
- Template patterns 344, 345
- wrapper patterns 338, 339, 342

Object.freeze()

- URL 300

- Object.seal()
 - URL 300
- objects
 - property, obtaining from 176, 177
 - updating 323, 324, 325, 327
- observables
 - creating, with operators 349
 - reference link 346
- Observer patterns 335
- observer programming 346
- operations
 - implementing 173, 174
 - turning, into functions 172, 173

P

- parameter order 210, 211
- parseInt()
 - URL 117
- partial application
 - about 184, 185, 196
 - with arrow functions 198
 - with closures 202, 203
 - with eval() 198, 199, 201
- partial currying
 - about 184, 185, 205, 206
 - with bind() 206, 208, 209
 - with closures 209
- persistent data structures
 - creating 320
 - limitations 328
 - lists, working 321, 322, 323
 - objects, updating 323, 324, 325, 327
- pipelines
 - building 220, 221, 222
 - constructs, using 222, 223, 224
 - creating 220
 - debugging 224
 - logging wrapper, using 227
 - tapping, into flow 226, 227
 - tee function, using 224, 225
- pipelining
 - about 216, 217
 - example 219, 220
 - pipelines, creating 220
 - pipelines, debugging 224
 - point-free style 228
 - using, in Linux 217, 218, 219
 - using, in Unix 217, 218, 219
- pivot 264
- point-free style
 - about 216, 228
 - converting to 229, 230
 - functions, defining 228, 229
- pointfree 57
- polyfills
 - about 64
 - Ajax, detecting 64, 65
 - missing functions, adding 66, 67
 - reference link 66
- popularity indices
 - reference link 14
- prisms
 - about 317
 - implementing 320
 - working with 317, 318, 320
- product types 367
- Promise.resolve()
 - URL 392
- promises
 - about 62
 - functions, turning into 175, 176
 - reference link 62
- property
 - obtaining, from object 176, 177
- proxy
 - URL 232
- pure functions, advantages
 - about 83
 - memoization 84, 85, 86, 87, 88
 - order of execution 83, 84
 - self-documentation 88
 - testing 88
- pure functions
 - about 73, 75
 - conditions 73
 - referential transparency 74, 75, 76
 - testing 95, 96
 - versus impure function 94
 - working in 152, 153, 155
- purified functions

testing 96, 97, 98

R

ranges

working with 117, 118

React sandbox

reference link 340

React-Redux package

reference link 343

React-Redux reducer 55

working 55, 56

Reactive Functional Programming (RFP) 347

reactive programming 346

recursion techniques

about 281

continuation-passing style (CPS) 285, 286, 287, 288, 289

elimination 292

tail calls, optimization 282, 283, 284, 285

thunks 289, 290, 291, 292

trampolines 289, 290, 291, 292

recursion

about 17, 18

applying 258, 259

applying, methods 259

backtracking 274

decrease and conquer strategy 259, 260, 261

divide and conquer strategy 261, 262, 263, 264, 265

dynamic programming 265, 267

eight queens puzzle 274, 275, 276, 277, 278

filtering 268, 269, 270, 271

higher-order functions 267, 271, 272, 273

mapping 268, 269, 270, 271

searching 274

tree structure, traversing 278, 280, 281

using 257, 258

reduce()

map(), emulating with 119

referential opacity 74

referential transparency 74, 75, 76

RxJS Operators

reference link 349

RxJS

about 347

installation link 349

S

Sanctuary

URL 369

searching 274

set()

reference link 238

setters

about 305

creating 306, 308

side effects, pure functions

about 75, 76

argument mutation 80

global state 77

inner state 78, 79

troublesome functions 81, 82, 83

usual side effects 76

simple memoization 158, 159

solutions, e-commerce related issues

about 30, 31

button, disabling 33

global flag, using 31

handler, modifying 33

handler, redefining 34

handler, removing 32

local flag, using 34, 35

Sorta Functional Programming (SFP) 9, 335

spread operator

about 20, 21

reference link 20

Strategy patterns 335, 344, 345

stub 97

stubbing 67, 68

sum types 368

T

tacit 57

tail calls

about 283

optimization 282, 283, 284, 285

TC39

reference link 148

Template patterns 335, 344, 345

testing 216

thunks 289, 290, 291, 292
timing functions 155, 156, 157
Traceur
 reference link 23, 24
trampolines 290, 291, 292
transducing
 about 248, 249, 250, 251
 reducers, composing 251, 252
 reducers, generalizing 252, 253
transformations
 about 105
 array, reducing to value 105, 106
 looping 127, 129
 operation, applying 113, 114
transpilers
 using 23, 24, 25
Try monad 391, 392

TypeScript
 reference link 25
 URL 24, 365

U

unary() higher-order function 369
Unix
 pipelining, using in 217, 218, 219

V

Value-added Tax (VAT) 188

W

white-box testing 99
wrapped functions 147
wrapper patterns 335, 338, 339, 340, 342
wrapping functions 148