

Federico Kereki

Mastering JavaScript Functional Programming

In-depth guide for writing robust and maintainable
JavaScript codes in ES8 and beyond



Packt>

Dedication

Writing a book involves many people, and even if I cannot mention and name all of them, there are some who really deserve being 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 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 on her own with everything in Uruguay, and the rest of the family--this book wouldn't have been possible otherwise, and she is the greatest reason the book could be done!

Mastering JavaScript Functional Programming

In-depth guide for writing robust and maintainable JavaScript code in ES8 and beyond

Federico Kereki



BIRMINGHAM - MUMBAI

Mastering JavaScript Functional Programming

Copyright © 2017 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, and its dealers and distributors will be held liable for any damages caused or alleged to be 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.

First published: November 2017

Production reference: 1281117

Published by Packt Publishing Ltd.
Livery Place
35 Livery Street

Birmingham
B3 2PB, UK.

ISBN 978-1-78728-744-0

www.packtpub.com

Credits

Author Federico Kereki	Copy Editor Safis Editing
Reviewers Gerónimo Garcia Sgritta Steve Perkins	Project Coordinator Ritika Manoj
Commissioning Editor Ashwin Nair	Proofreader Safis Editing
Acquisition Editor Larissa Pinto	Indexer Aishwarya Gangawane
Content Development Editor Mohammed Yusuf Imaratwale	Graphics Jason Monteiro

Technical Editor Ralph Rosario	Production Coordinator Melwyn Dsa

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 also wrote booklets on computer security (*Linux in the Time of Malware* and *SSH: A Modern Lock for your Server*) and a book (*Essential GWT: Building for the Web with Google Web Toolkit*) on GWT programming.

Kereki has given talks on Functional Programming with JavaScript in public conferences (such as JSConf 2016) 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 he is currently on a project in India, and he wrote this book in that country.

About the Reviewer

Steve Perkins is the author of *Hibernate Search by Example*. He has been working with Java and JavaScript since the late-1990s, with forays into Scala and Go. Steve lives in Atlanta, GA, USA, with his wife and two children, and is currently a lead developer at BetterCloud, where he works on a microservice-based platform for managing SaaS applications.

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

www.PacktPub.com

For support files and downloads related to your book, please visit www.PacktPub.com.

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.PacktPub.com and as a print book customer, you are entitled to a discount on the eBook copy. Get in touch with us at service@packtpub.com for more details.

At www.PacktPub.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.



<https://www.packtpub.com/mapt>

Get the most in-demand software skills with Mapt. Mapt gives you full access to all Packt books and video courses, as well as industry-leading tools to help you plan your personal development and advance your career.

Why subscribe?

- Fully searchable across every book published by Packt
- Copy and paste, print, and bookmark content
- On demand and accessible via a web browser

Customer Feedback

Thanks for purchasing this Packt book. At Packt, quality is at the heart of our editorial process. To help us improve, please leave us an honest review on this book's Amazon page at <https://www.amazon.com/dp/1787285731>.

If you'd like to join our team of regular reviewers, you can e-mail us at customerreviews@packtpub.com. We award our regular reviewers with free eBooks and videos in exchange for their valuable feedback. Help us be relentless in improving our products!

Table of Contents

Preface

- What this book covers
- What you need for this book
- Who this book is for
- Conventions
- Reader feedback
- Customer support
 - Downloading the example code
- Errata
- Piracy
- Questions

1. Becoming Functional - Several Questions

- What is Functional Programming?
 - Theory versus practice
 - A different way of thinking
 - What Functional Programming is not
- Why use Functional Programming?
 - What we need
 - What we get
 - Not all is gold...
- Is JavaScript functional?
 - JavaScript as a tool
 - Going functional with JavaScript
 - Key features of JavaScript
 - Functions as First Class Objects
 - Recursion
 - Closures
 - Arrow functions
 - Spread
- How do we work with JavaScript?
 - Using transpilers
 - Working online
 - Testing
- Questions
- Summary

2. Thinking Functionally - A First Example

- The problem - do something only once
- Some bad solutions
 - Solution #1 - hope for the best!

- Solution #2 - use a global flag
- Solution #3 - remove the handler
- Solution #4 - change the handle
- Solution #5 - disable the button
- Solution #6 - redefine the handler
- Solution #7- use a local flag

A functional solution

A higher-order solution

Testing the solution manually

Testing the solution automatically

An even better solution

Questions

Summary

3. Starting Out with Functions - A Core Concept

All about functions

Of lambdas and functions

Arrow functions - the modern way

Returning values

Handling the this value

Working with arguments

One argument or many?

Functions as objects

A React+Redux reducer

An unnecessary mistake

Working with methods

Using functions in FP ways

Injection - sorting it out

Callbacks, promises, and continuations

Continuation Passing Style

Polyfills

Detecting Ajax

Adding missing functions

Stubbing

Immediate invocation

Questions

Summary

4. Behaving Properly - Pure Functions

Pure functions

Referential Transparency

Side effects

Usual side effects

Global state

- Inner state
- Argument mutation
- Troublesome functions
- Advantages of pure functions
 - Order of execution
 - Memoization
 - Self-documentation
 - Testing
- Impure functions
 - Avoiding impure functions
 - Avoiding the usage of state
 - Injecting impure functions
 - Is your function pure?
- Testing - pure versus impure
 - Testing pure functions
 - Testing purified functions
 - Testing impure functions
- Questions
- Summary

5. Programming Declaratively - A Better Style

- Transformations
 - Reducing an array to a value
 - Summing an array
 - Calculating an average
 - Calculating several values at once
 - Folding left and right
 - Applying an operation - map
 - Extracting data from objects
 - Parsing numbers tacitly
 - Working with ranges
 - Emulating map() with reduce()
 - More general looping
- Logical higher-order functions
 - Filtering an array
 - A reduce() example
 - Emulating filter() with reduce()
 - Searching an array
 - A special search case
 - Emulating find() and findIndex() with reduce()
 - Higher level predicates - some, every
 - Checking negatives - none
- Questions
- Summary

6. Producing Functions - Higher-Order Functions

- Wrapping functions

 - Logging

 - Logging in a functional way

 - Taking exceptions into account

 - Working in a more pure way

 - Timing

 - Memoizing

 - Simple memoization

 - More complex memoization

 - Memoization testing

- Altering functions

 - Doing things once, revisited

 - Logically negating a function

 - Inverting results

 - Arity changing

- Other higher-order functions

 - Turning operations into functions

 - Implementing operations

 - A handier implementation

 - Turning functions into promises

 - Getting a property from an object

 - Demethodizing - turning methods into functions

 - Finding the optimum

- Questions

- Summary

7. Transforming Functions - Currying and Partial Application

- A bit of theory

- Currying

 - Dealing with many parameters

 - Currying by hand

 - Currying with `bind()`

 - Currying with `eval()`

- Partial application

 - Partial application with arrow functions

 - Partial application with `eval()`

 - Partial application with closures

- Partial currying

 - Partial currying with `bind()`

 - Partial currying with closures

- Final thoughts

 - Parameter order

 - Being functional

Questions

Summary

8. Connecting Functions - Pipelining and Composition

Pipelining

Piping in Unix/Linux

Revisiting an example

Creating pipelines

Building pipelines by hand

Using other constructs

Debugging pipelines

Using tee

Tapping into a flow

Using a logging wrapper

Chaining and fluent interfaces

Pointfree style

Defining pointfree functions

Converting to pointfree style

Composing

Some examples of composition

Unary operators

Counting files

Finding unique words

Composing with higher order functions

Testing composed functions

Questions

Summary

9. Designing Functions - Recursion

Using recursion

Thinking recursively

Decrease and Conquer: searching

Decrease and Conquer: doing powers

Divide and conquer: The Tower of Hanoi

Divide and conquer: sorting

Dynamic programming: making change

Higher order functions revisited

Mapping and filtering

Other higher-order functions

Searching and backtracking

The Eight Queens puzzle

Traversing a tree structure

Recursion techniques

Tail call optimization

Continuation passing style

Trampolines and thunks

- Recursion elimination

- Questions

- Summary

10. Ensuring Purity - Immutability

- The straightforward JS way

- Mutator functions

- Constants

- Freezing

- Cloning and mutating

- Getters and setters

- Getting a property

- Setting a property by path

- Persistent data structures

- Working with lists

- Updating objects

- A final caveat

- Questions

- Summary

11. Implementing Design Patterns - The Functional Way

- What are Design Patterns?

- Design pattern categories

- Do we need design patterns?

- Object-oriented design patterns

- Façade and Adapter

- Decorator or Wrapper

- Strategy, Template, and Command

- Other patterns

- Functional design patterns

- Questions

- Summary

12. Building Better Containers - Functional Data Types

- Data types

- Signatures for functions

- Other type options

- Containers

- Extending current data types

- Containers and functors

- Wrapping a value: a basic container

- Enhancing our container: functors

- Dealing with missing values with Maybe

- Monads

- Adding operations

- Handling alternatives - the Either monad

Calling a function - the Try monad

Unexpected Monads - Promises

Functions as data structures

Binary trees in Haskell

Functions as binary trees

Questions

Summary

Bibliography

Answers to Questions

Preface

In computer programming, paradigms abound: Some examples are imperative programming, structured (*go to less*) programming, object-oriented programming, 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. **Functional Programming (FP)** emphasizes writing functions, and connecting them in simple ways to produce 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 is thought to be *theoretical* rather than *practical*, even *mathematical*, and possibly uses vocabulary and concepts that are foreign to developers--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 programming.

FP is not something 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 everything you 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 functional programming 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 JS* versions can be even simpler to understand!

What this book covers

In this book, we'll cover **Functional Programming (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 way.

[Chapter 3](#), *Starting Out with Functions - A Core Concept*, will go over the central concept of FP: functions, and the different options available in JavaScript.

[Chapter 4](#), *Behaving Properly - Pure Functions*, will consider the concept of purity and pure functions, and show 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 working 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 show 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 show how a key concept in FP, recursion, can be applied to designing algorithms and functions.

[Chapter 10](#), *Ensuring Purity - Immutability*, will show some tools that can help you 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 show some more high-level functional patterns, introducing types, containers, functors, monads, and several other more advanced FP concepts.

I tried to keep examples simple and down-to-earth, because I wanted to focus on 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, seeing how to fully work it out with the chosen tools. (Also, 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 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 bricks together, how to wire things up, and so on, so that you will be able to apply whatever you need, as it may fit.

What you need for 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/>) or the like, with absolutely nothing else.

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

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.

Conventions

In this book, you will find a number of text styles that distinguish between different kinds of information. Here are some examples of these styles and an explanation of their meaning.

Code words in text, database table names, folder names, filenames, file extensions, pathnames, dummy URLs, user input, and Twitter handles are shown as follows:

"Just assign the names of the layers you want to activate to the `VK_INSTANCE_LAYERS` environment variable".

A block of code is set as follows:

```
{
  if( (result != VK_SUCCESS) ||
      (extensions_count == 0) ) {
    std::cout << "Could not enumerate device extensions." << std::endl;
    return false;
  }
```

Any command-line input or output is written as follows:

```
| setx VK_INSTANCE_LAYERS VK_LAYER_LUNARG_api_dump;VK_LAYER_LUNARG_core_validation
```

New terms and **important words** are shown in bold. Words that you see on the screen, for example, in menus or dialog boxes, appear in the text like this: "Select System info from the Administration panel."



Warnings or important notes appear in a box like this.



Tips and tricks appear like this.

Reader feedback

Feedback from our readers is always welcome. Let us know what you think about this book-what you liked or disliked. Reader feedback is important for us as it helps us develop titles that you will really get the most out of.

To send us general feedback, simply e-mail feedback@packtpub.com, and mention the book's title in the subject of your message.

If there is a topic that you have expertise in and you are interested in either writing or contributing to a book, see our author guide at www.packtpub.com/authors .

Customer support

Now that you are the proud owner of a Packt book, we have a number of things to help you to get the most from your purchase.

Downloading the example code

You can download the example code files for this book from your account at <http://www.packtpub.com>. If you purchased this book elsewhere, you can visit <http://www.packtpub.com/support> and register to have the files e-mailed directly to you.

You can download the code files by following these steps:

1. Log in or register to our website using your e-mail address and password.
2. Hover the mouse pointer on the SUPPORT tab at the top.
3. Click on Code Downloads & Errata.
4. Enter the name of the book in the Search box.
5. Select the book for which you're looking to download the code files.
6. Choose from the drop-down menu where you purchased this book from.
7. Click on Code Download.

You can also download the code files by clicking on the Code Files button on the book's webpage at the Packt Publishing website. This page can be accessed by entering the book's name in the Search box. Please note that you need to be logged in to your Packt account.

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

Errata

Although we have taken every care to ensure the accuracy of our content, mistakes do happen. If you find a mistake in one of our books-maybe a mistake in the text or the code-we would be grateful if you could report this to us. By doing so, you can save other readers from frustration and help us improve subsequent versions of this book. If you find any errata, please report them by visiting <http://www.packtpub.com/submit-errata>, selecting your book, clicking on the Errata Submission Form link, and entering the details of your errata. Once your errata are verified, your submission will be accepted and the errata will be uploaded to our website or added to any list of existing errata under the Errata section of that title.

To view the previously submitted errata, go to <https://www.packtpub.com/books/content/support> and enter the name of the book in the search field. The required information will appear under the Errata section.

Piracy

Piracy of copyrighted material on the Internet is an ongoing problem across all media. At Packt, we take the protection of our copyright and licenses very seriously. If you come across any illegal copies of our works in any form on the Internet, please provide us with the location address or website name immediately so that we can pursue a remedy.

Please contact us at copyright@packtpub.com with a link to the suspected pirated material.

We appreciate your help in protecting our authors and our ability to bring you valuable content.

Questions

If you have a problem with any aspect of this book, you can contact us at questions@packtpub.com, and we will do our best to address the problem.

Becoming Functional – Several Questions

–Functional Programming (or FP, as it's usually shortened) has been around since the earliest times and is going through a sort of revival, due to its increased use with several frameworks and libraries, and most particularly in JavaScript. In this chapter, we shall:

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

So, let's get started by asking ourselves *What is FP?* and start working on that topic.

What is Functional Programming?

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

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 you'll be told 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 within.



For Trivia 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.

Theory versus practice

In this book, we won't be going about FP in a theoretical way: our point is, rather, to show 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 rather in a very practical way. We won't dismiss useful JS constructs, only because they don't happen to fulfill the academic expectations of FP. We won't avoid practical JS features just to fit the FP paradigm. In fact, we could almost say we'll be doing *SFP*—*Sorta Functional Programming* because our code will be a mixture of FP features and more classical imperative and **Object Oriented Programming (OOP)**.

(This 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, give some vocabulary and definitions, and explain core FP concepts... but we'll always be keeping in sight the idea of helping to produce actual, useful, JS code, and not to try to achieve 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 JS make it possible to develop applications that wouldn't even have been possible just a few years ago; think of mobile (hybrid) apps done with Ionic, Apache Cordova, or React Native, or desktop apps done with Electron or NW.js, for example. JS has also migrated to the backend with Node.js, so today the scope of usage for the language has grown in a serious way, and dealing with all the added complexity taxes all designs.

A different way of thinking

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

Another way of saying this, is that 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 getting accustomed to this style, you'll find that your code becomes simpler, shorter, more elegant, and can be easily tested and debugged. However, don't fall into the trap of considering FP as a goal! Think of FP only as a means towards 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 techniques!

What Functional Programming is not

Since we have been saying some things about what FP is, let's also clear some common misconceptions, and consider some things that 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 in order to prove an important result in theoretical computer science. (This work 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):** Also, it isn't either a case of choosing declarative or imperative ways of programming. You can mix and match as it best suits you, and we'll be doing that sort of thing in 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 JS, but the differences are mostly syntactic. Once you learn the basic concepts, you'll see you can get the same results in JS as with FP languages.

It may also be relevant to mention that several modern frameworks, such as the React+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 *black box* fashion.

Similarly, in Redux you get 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 theoretical advantages (we'll be getting to those in the following section) and of 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 Functional Programming?

Throughout the years, there have been many programming styles and fads. However, FP has proved quite resilient and is of great interest today. Why would you care to use FP? The question should rather first be, *What do you want to get?* and only then *Does FP get you that?*

What we need

We can certainly agree that the following list of concerns are universal. Our code should be:

- **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 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 understand their relationships without undue effort. This is highly correlated with *maintainability*: your code will have to be maintained at some time in the future, to change or add some new functionality.
- **Testable:** *unit tests* try out small parts of your program, verifying their behavior with independence of the rest of the code. Your programming style should favor writing code that simplifies the job of writing unit tests. Also, unit tests are like documentation, insofar 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 only minimally (if at all) the structure and data flow of the original code. Small changes shouldn't imply large, serious refactorings of your code.
- **Reusable:** *code reuse* has the goal of saving resources, time, 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), plus *high cohesion* (all the pieces in a module do 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, now, does FP get you these five characteristics?

- In FP, the goal is writing separate independent functions, which are joined together to produce the final results.
- Programs written in functional style usually tend to be cleaner, shorter, and easier to understand.
- Functions can be tested on its own, and FP code has advantages for that.
- 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 FP ways, code becomes more understandable and easier to extend. So, it seems that all five characteristics can be ensured with FP!



*For a well balanced look at reasons for 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 JS, 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 *automagically* make your code better. Some FP solutions are actually tricky — and there are developers who show much glee in writing code and then asking *What does this do?* If you aren't careful, your code may become *write-only*, practically impossible to maintain... and there go *Understandable*, *Extensible*, and *Reusable* out of the door!

Another disadvantage: you may find it harder to find FP-savvy developers. (Quick question: how many *Functional Programmer Sought* job ads have you ever seen?) The vast majority of today's JS 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 JS, and simple tasks may become hard to do. As we said at the beginning, we'll rather opt for *Sorta FP*, so we won't be drastically rejecting any JS features that aren't 100% functional. 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!

Is JavaScript functional?

About this time, another important question that you should be asking: *Is JS a functional language?* Usually, when thinking about FP, the mentioned languages do not include JS, but do listless common options, such as Clojure, Erlang, Haskell, or 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.

JavaScript as a tool

What is JS? 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 JS consistently is in the *top ten* of popularity. From a more academic point of view, the language is sort of a mixture, with features from several different languages. Several libraries helped the growth of the language, by providing features that weren't so easily available, as classes and inheritance (today's version of JS does support classes, but that was not the case not too long ago) that otherwise had to be simulated 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.

JS has grown to be incredibly powerful. But, as with all power tools, it gives you a way to produce great solutions, and also to do great harm. FP could be considered to be 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 JS code, you cannot expect large reworkings of the language that would cause most sites to fail. You must learn to live on with the good and the bad, and simply avoid the latter parts.

In addition, JS 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 JS on its own, but we will make references to existing, available code.

If we ask if JS is actually functional, the answer will be, once again, *sorta*. JS can be considered to be 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, JS 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 JS language features, and we'll try to minimize the problems caused by the more conventional parts of the language. In this sense, JS will or won't be functional, depending on *your* programming style!

If you want to use FP, you should decide upon 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 — and never mind if JS is or isn't functional!

Going functional with JavaScript

JS has evolved through the years, and the version we'll be using is (informally) called JS8, and (formally) ECMAScript 2017, usually shortened to ES2017 or ES8; this version was finalized in June 2017. The previous versions were:

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



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 JS, including JSON, C#, Dart, and others. See 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 JS in the text without further specification, ES8 (ES2017) is meant. However, in terms of the language features that are used in the book, if you were just to use ES2015, you'd have no problems with this book.

No browsers fully implement ES8; most provide an older version, JavaScript 5 (from 2009), with a (always growing) smattering of ES6, ES7, and ES8 features. This will prove to be a problem, but fortunately, a solvable one; we'll get to this shortly, and we'll be using ES8 throughout the book.



In fact, there are only a little 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.

Key features of JavaScript

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

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

Let's see some examples of each one, to explain why they will be useful to us.

Functions as First Class Objects

Saying that functions are *first class objects* (also: *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, 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:

```
$.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, but for the sake of our example, the older way is enough. We'll be getting back to promises, though, in section Building Better Containers, of [chapter 12](#), Building Better Containers – Functional Data Types, when we discuss Monads; in particular, see section Unexpected Monads: Promises.

Since functions can be stored in variables, you could also write:

```
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*, when we consider Higher–Order Functions.

Recursion

This is a 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 $n!$) as defined for non-negative 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 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 $5! = 5 * 4!$. If you continue to work this example, you'll get that $5! = 5 * 4 * 3 * 2 * 1 = 120$, so $n!$ is the product of all numbers up to n .

This can be immediately turned into JS 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 complete [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 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:

```
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: among others, *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](#)).

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 one allows you to write as much code as you want; the second is short for `{ return expression }`. We could rewrite our earlier Ajax example as:

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

A new version of the factorial code could be:

```
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 see more in section Arrow Functions of [Chapter 3, Starting Out with Functions - A Core Concept](#).

You would probably write the latter as a one-liner -- can you see the equivalence?

```
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. A short comment: 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, and it 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 4 --no-bracket-spacing`.



In lambda calculus, a function as $x \Rightarrow 2 * x$ would be represented as $\lambda x. 2 * x$ -- though there are syntactical differences, the definitions are analogous. Functions with more parameters are a bit more complicated: $(x,y) \Rightarrow x + y$ would be expressed as $\lambda x. \lambda y. x + y$. We'll see more about this in section Of Lambdas and functions, in [Chapter 3](#), Starting Out with Functions - A Core Concept, and in section Currying, in [Chapter 7](#), Transforming Functions - Currying and Partial Application.

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:

```
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:

```
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 *equation*. 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 array, and C is for 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 shorter, more concise code, and we will be taking advantage of it.

How do we work with JavaScript?

All this is quite well, but as we mentioned before, it so happens that the JS version available most everywhere isn't ES8, but rather the earlier JS5. An exception to this is Node.js: it is based on Chrome's V8 high-performance JS engine, which already has several ES8 features available. Nonetheless, as of today, ES8 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 and V8.)

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 may be using older browsers, which don't support the fancy features you're keen on using? Let's see some solutions for that.



If you want to be sure 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/>.

Feature name	Current browser	Compilers/polyfills					Desktop browsers									
		Traceur	Babel + core-js ^[2]	Closure	TypeScript + core-js	es6-shim	KQ 4.14 ^[3]	IE 11	Edge 14 ^[4]	Edge 15 ^[4]	FF 45 ESR	FF 52 ESR	FF 53	CH 58, OP 45 ^[1]	SF 10	SF 10.1
Optimisation																
proper tail calls (tail call optimisation)	0/2	0/2	0/2	0/2	0/2	0/2	0/2	0/2	0/2	0/2	0/2	0/2	0/2	0/2	2/2	2/2
Syntax																
default function parameters	7/7	4/7	4/7	4/7	5/7	0/7	0/7	0/7	7/7	7/7	4/7	6/7	7/7	7/7	7/7	7/7
rest parameters	5/5	4/5	3/5	2/5	4/5	0/5	0/5	0/5	5/5	5/5	5/5	5/5	5/5	5/5	5/5	5/5
spread (...) operator	15/15	15/15	13/15	12/15	4/15	0/15	0/15	0/15	15/15	15/15	15/15	15/15	15/15	15/15	15/15	15/15
object literal extensions	6/6	6/6	6/6	4/6	6/6	0/6	0/6	0/6	6/6	6/6	6/6	6/6	6/6	6/6	6/6	6/6
for...of loops	9/9	9/9	9/9	6/9	3/9	0/9	0/9	0/9	7/9	9/9	7/9	7/9	9/9	9/9	9/9	9/9
octal and binary literals	4/4	2/4	4/4	4/4	4/4	2/4	0/4	0/4	4/4	4/4	4/4	4/4	4/4	4/4	4/4	4/4
template literals	5/5	4/5	4/5	3/5	3/5	0/5	0/5	0/5	5/5	5/5	5/5	5/5	5/5	5/5	5/5	5/5
RegExp "y" and "u" flags	5/5	3/5	3/5	0/5	0/5	0/5	0/5	0/5	5/5	5/5	2/5	5/5	5/5	5/5	5/5	5/5
destructuring declarations	22/22	20/22	21/22	19/22	15/22	0/22	0/22	0/22	21/22	22/22	19/22	21/22	22/22	22/22	22/22	22/22
destructuring assignment	24/24	23/24	24/24	17/24	19/24	0/24	0/24	0/24	23/24	24/24	21/24	23/24	24/24	24/24	24/24	24/24
destructuring parameters	23/23	19/23	20/23	18/23	15/23	0/23	0/23	0/23	22/23	23/23	18/23	20/23	23/23	23/23	23/23	23/23
Unicode code point escapes	2/2	1/2	1/2	1/2	1/2	0/2	0/2	0/2	2/2	2/2	1/2	1/2	2/2	2/2	2/2	2/2

Figure 1.1. - The latest versions of JS aren't yet widely and fully supported, so you'll have to check before using any of their new features

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 ES8 code, and transform it into equivalent JS5 code. (It's a source-to-source transformation, instead of a source-to-object code as in compilation.) You can code using advanced ES8 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 did the word transpiler come from, it is a portmanteau of translate and compiler. There are many such combinations in technological speak: email (electronic+mail), emoticon (emotion+icon), malware (malicious+software), or alphanumeric (alphabetic+numeric), and several more.

The most common transpilers for JS 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 your code will get automatically transpiled and provided to end users. You can also try out transpilation online; see Figure 1.2 for an example using Babel's online environment:

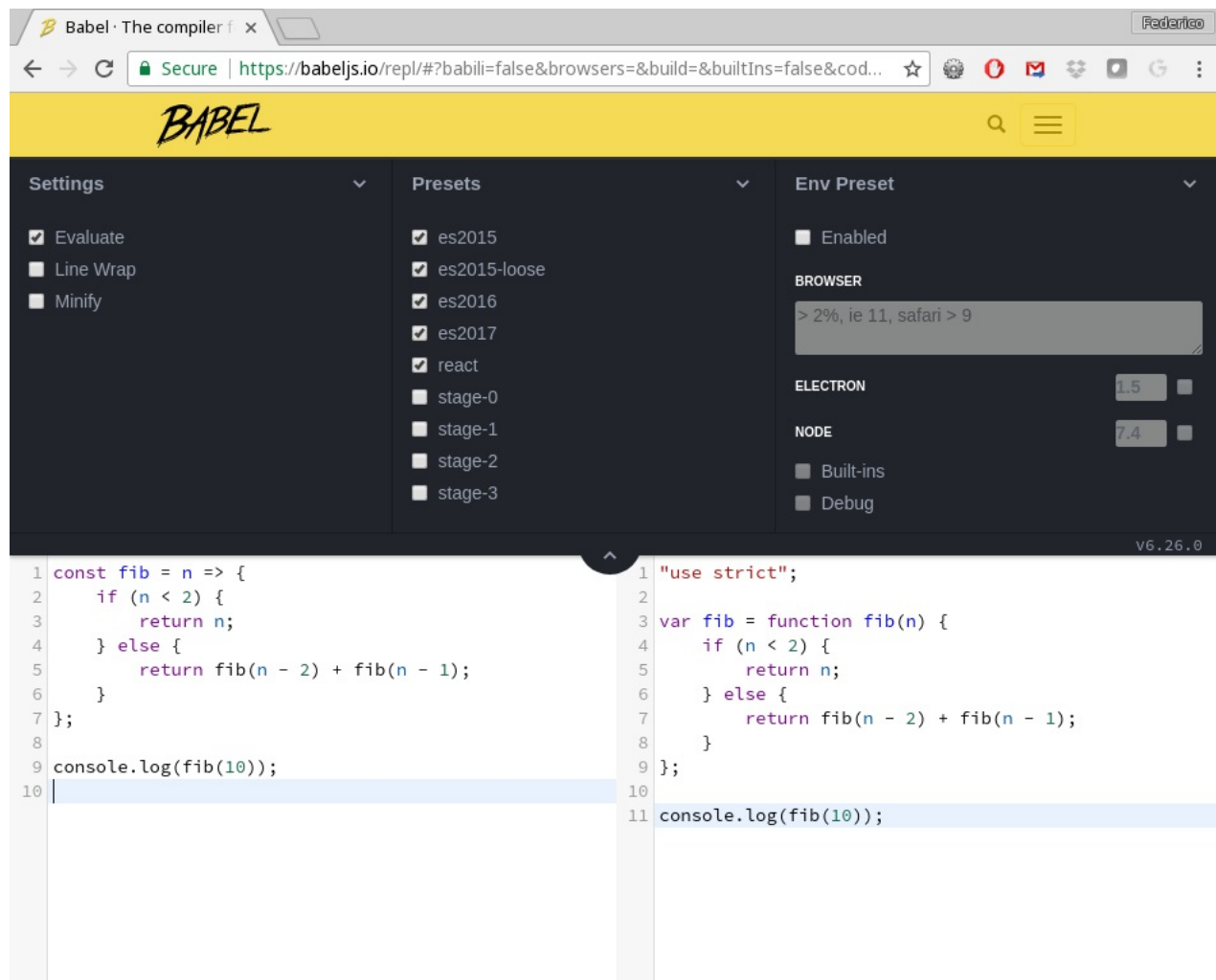


Figure 1.2 - The Babel transpiler converts ES8 code into compatible JS5 code

If you prefer Traceur, 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 this.) Select the EXPERIMENTAL option, to fully enable ES8 support:

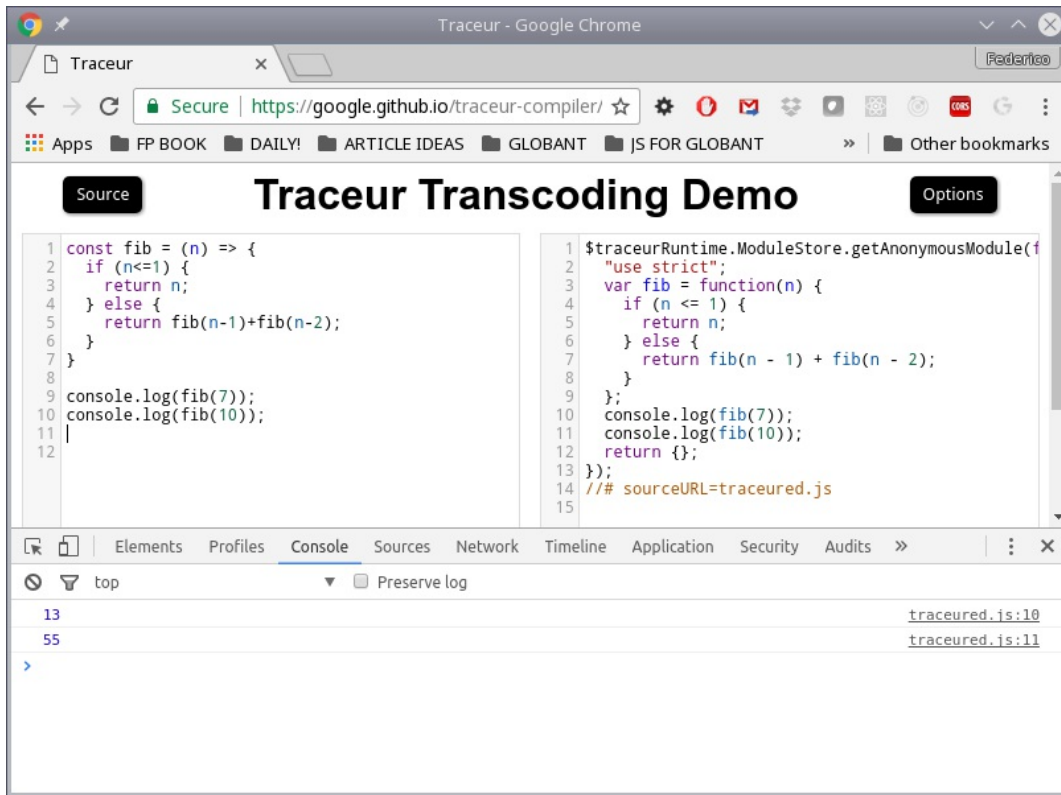


Figure 1.3 - The Traceur transpiler is an equally valid alternative for ES8-to-JS5 translation



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

There's a final possibility you may want to consider: instead of JS, opt for Microsoft's TypeScript (at <http://www.typescriptlang.org/>), a superset of JS, compiled to JS5. The main advantage of TypeScript is adding (optional) static type checks to JS, which helps detect some programming errors at compile time. Beware: as with Babel or Traceur, not all of ES8 will be available.



You can also get 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 types checks, and you can also run your code on the spot. See figure 1.4:

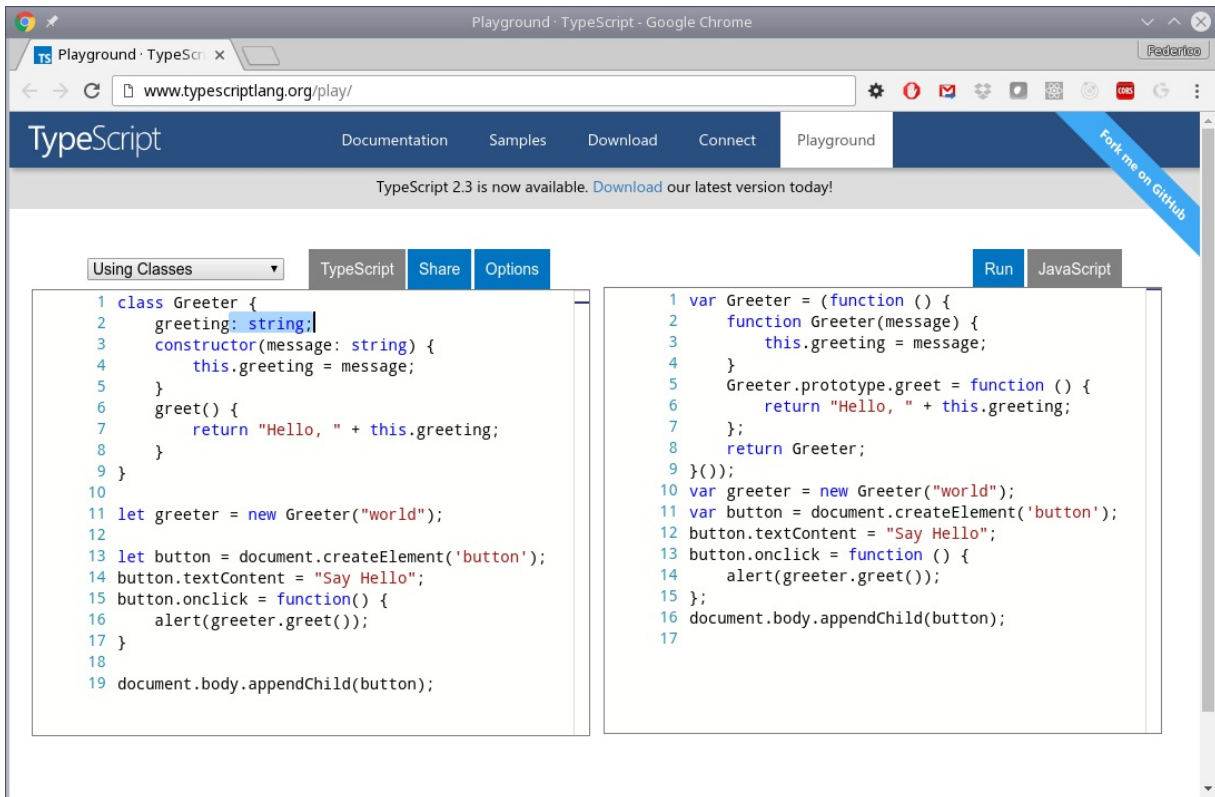


Figure 1.4 - TypeScript adds type checking features, for safer JS programming

Working online

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

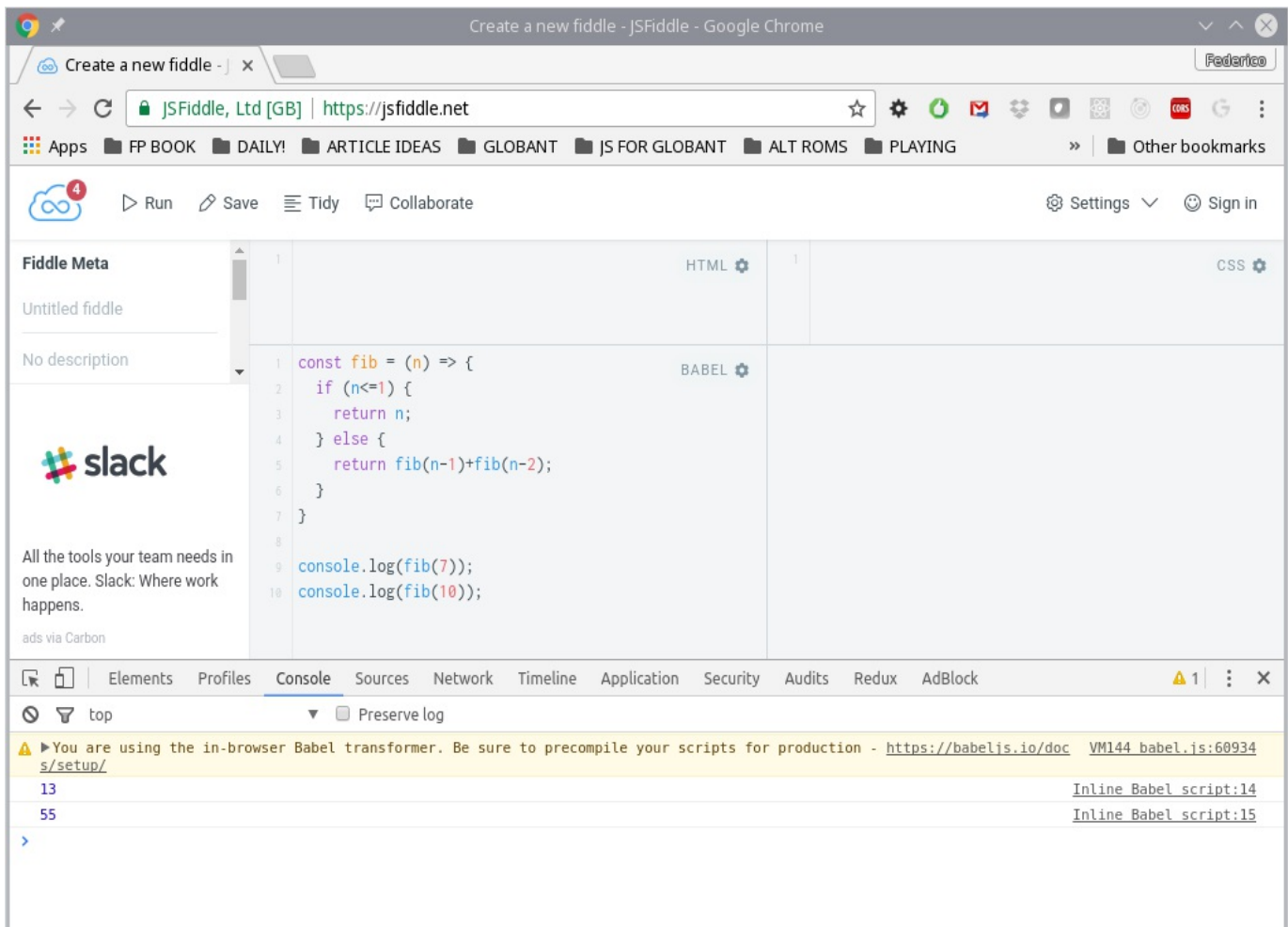


Figure 1.5 - JSFiddle lets you try out ES8 code (plus HTML and CSS) without requiring any further tools

Testing

We will also touch on testing, which is, after all, one of FP's main advantages. For that, 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.

Questions

1.1. Classes as first-class objects: We saw that functions are first class objects, but did you know *classes* also are? (Though, of course, speaking of classes as *objects* does sound weird...) Study this 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 we wrote doesn't verify if its argument is valid or not. Can you add the necessary checks? Try to avoid repeated, redundant tests!

1.3. Climbing factorial: Our implementation of factorial starts 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*?

Summary

In this chapter, we have seen the basics of functional programming, 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 take advantage of the rest of the book.

In [chapter 2](#), *Thinking Functionally - A First Example*, we'll go over an example of a simple problem, and look at it in *common* ways, to finally end by solving it in a functional manner and analyze the advantages of that way of working.

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 JS... but let's now leave theory behind, and start out by considering a simple problem, and how to solve it in a functional way.

In this chapter, we will see:

- A simple, common, e-commerce related problem
- Several usual ways to solve it, with their associated defects
- A way to solve the problem by looking at it functionally
- A higher-order solution, which can be applied to other problems
- How to do unit testing for the functional solutions

In future chapters, We'll be coming back to some of the topics listed here, so we won't be getting very much into details. We'll just show how FP can give a different outlook for our problem, and leave further details for afterward.

The problem - do 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 at 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 would 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 this:

```
| 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, in unobtrusive fashion, you should assign the handler through code. So... Do as I say, not as I do!

This is a very barebones explanation of the problem and your web page, but it's enough for our purposes. Let's now get to think about ways of avoiding repeated clicks on that button... *How can we manage to avoid the user clicking more than once?*

Some bad solutions

OK, how many ways can you think of, in order to solve our problem? Let's go over several solutions, and analyze their quality.

Solution #1 - hope 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 as Figure 2.1.



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

This is a weasel way of avoiding the problem, but 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 as the following code.

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

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

Solution #2 - use 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 change it to `true`, and execute the function; otherwise, you don't do anything at all:

```
let clicked = false;
.
.
.
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 JS 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 do a second buy, 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 - remove 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:

```
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:

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

(note the extra argument) and the called function would be:

```
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 - change the handle

A variant to the previous solution would be not removing the click function, and 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:

```
function alreadyBilled() {  
    window.alert("Your billing process is running; don't click, please.");  
}  
  
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, need to reset the handler, harder testing), so we won't consider it to be quite good anyway.

Solution #5 - disable the button

A similar idea: instead of removing the event handler, disable the button, so the user won't be able to click. You might have a function like the following.

```
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 for the previous solutions (coupling the code to the button, need to re-enable the button, harder testing), so we don't like this solution either.

Solution #6 - redefine 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 line `billTheUser=...` 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- use 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 a *Immediately Invoked Function Expression (IIFE)*: we'll see more on this in [Chapter 3, Starting Out with Functions - A Core Concept](#), and in [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 objection 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 in some ways to it.) OK, it's not too hard to do, but don't forget the *Don't Repeat Yourself (D.R.Y)* advice!

A functional solution

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 that thing, and no other
- 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 over 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!

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. (We'll be seeing more of higher-order functions in [Chapter 6, Producing Functions - Higher-Order Functions](#); in particular, see section *Doing things once, revisited*.)



Underscore and LoDash already has 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 way seems imposing at first, but as you get accustomed to working in FP fashion, you'll get used to this sort of code, and will 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 a 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; we went for a more general term.
- The line `return (...args) => ...` says that `once()` will return a function, with some (0, 1, or more) parameters. Note that we are using the spread syntax we saw in [Chapter 1, Becoming Functional - Several Questions](#). With older versions of JS 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 ES8 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, as 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 sing, *You Can't Always Get What You Want!**

Testing the solution manually

We can run a simple test:

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

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:

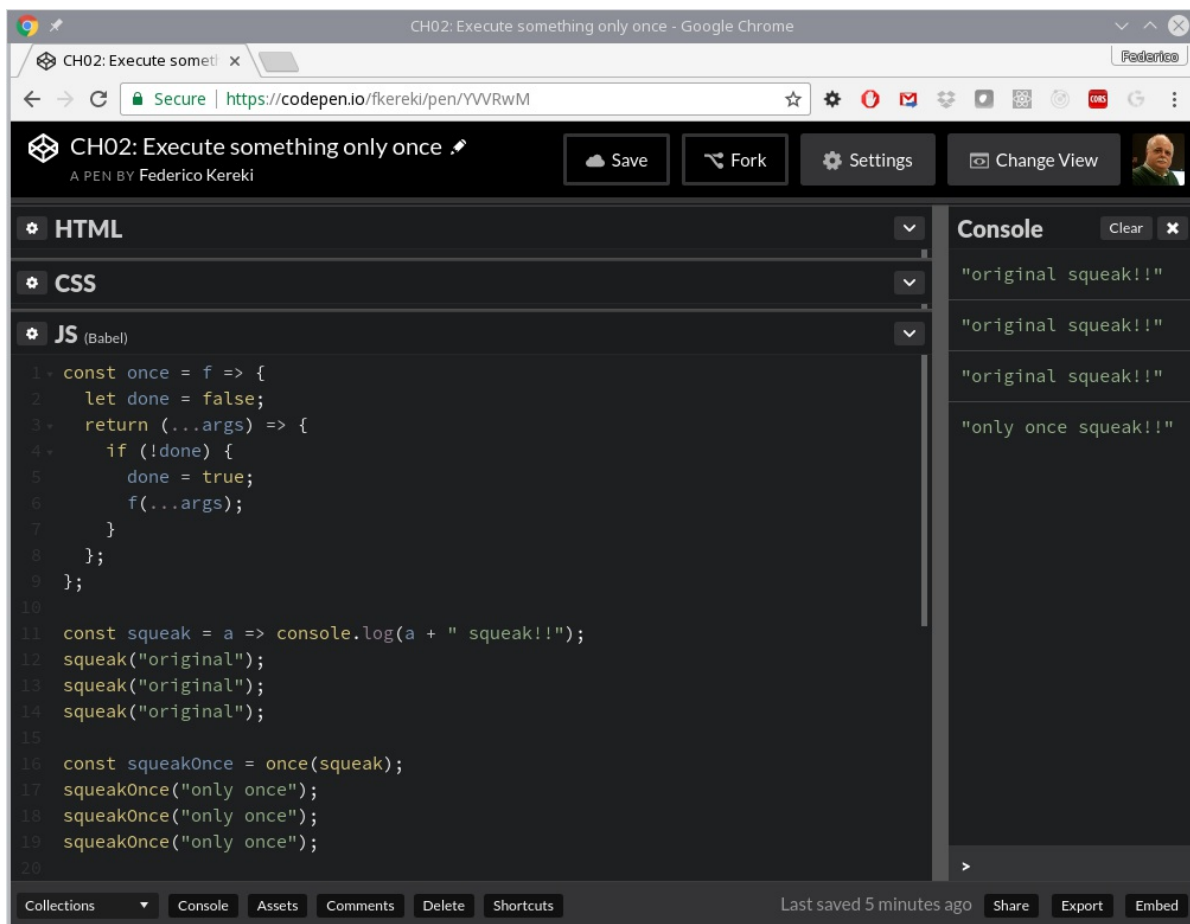


Figure 2.2 - Testing our once() higher-order function

Testing the solution automatically

Running tests by hand is no good; it gets tiresome, boring, and that leads, after time, 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:

```
<!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:

```
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:

- In order 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 what 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()`-ed 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:

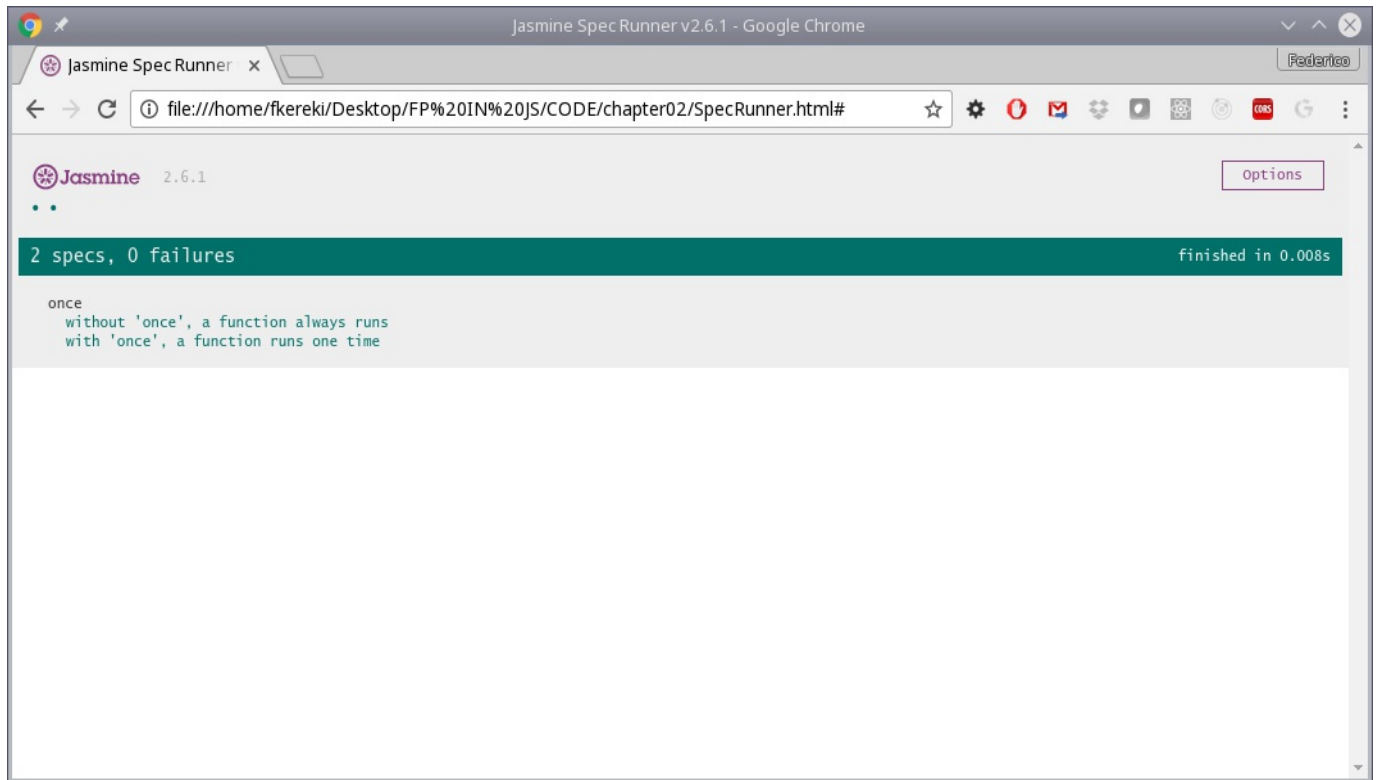


Figure 2.3 - Running automatic tests on our function, with Jasmine

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, and not silently ignoring 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:

```
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 with the same lines as we did earlier:

```
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:

```
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.

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 afterwards do nothing? To give an example, `once(fn)` and `thisManyTimes(fn,1)` would produce functions that behave in exactly the same way.

Summary

In this chapter, we've seen a common, simple problem, based on a real-life situation, and after analyzing several usual ways of solving that, we went for a *functional thinking* solution. We saw how to apply FP to our problem, and we also found a more general higher-order way 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 even 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.

In the next [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.

Starting Out with Functions - A Core Concept

In [chapter 2](#), *Thinking Functionally - A First Example*, we went over an example of FP thinking, but let's now go to the basics, and review functions. In [Chapter 1](#), *Becoming Functional - Several Questions*, we mentioned that two important JS features were functions as first-class objects and closures. Now, in this chapter, let's:

- Examine some key ways of defining functions in JS
- Go in detail regarding arrow functions, which are closest to the lambda calculus functions
- Introduce the concept of *currying*
- Revisit the concept of functions as first-class objects

We'll also consider several FP techniques, such as:

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

All about functions

Let's get started with a short review of functions in JS and their relationship to FP concepts. We can start something we mentioned in previous chapters, about functions as first-class objects, and then go on to several considerations about their usage in JS.

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.



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.

Applying a function means that you provide an actual argument to it, and that 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 JS? That's an interesting question! There are several ways of defining functions, and not all have the same meaning.



A good article showing the many ways of defining functions, methods, and more, is [The Many Faces of Functions in JavaScript](https://bocoup.com/blog/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 JS? The answer is, *probably in more ways than you thought!* At the very least, you could write:

- 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 that's enough.

JS also allows defining generator functions as in `function*(...) {...}`



that actually return a `Generator` object and `async` functions that really are a mix of generators and promises. We won't be using these kinds of functions, but read more about them at [https://developer.mozilla.org/en/docs/Web/JavaScript/Reference/Statements/function](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 in JS and defines a function named `first` (that is, `first.name=="first"`). Due to *hoisting*, this function will be accessible everywhere in the scope where it's defined.



Read more about hoisting at <https://developer.mozilla.org/en-US/docs/Glossary/Hoisting> and keep in mind that it applies only to declarations, but not to initializations.

- The second definition, assigning a function to a variable, also produces a function, but an *anonymous* (that is, no name) one. However, many JS engines are capable of deducing what the name should be, and set `second.name=="second"` (check the following code, which shows a case where the anonymous function gets 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 do 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 do without a name. However, note that named functions are more easily recognized in an error traceback.

- 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 `count`'s initial value (if no initial value is provided, we start at zero). 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 base of the *module* pattern.

- The fifth definition isn't safe, and you shouldn't use it! You pass the arguments names, and then the actual function body as a string in the last argument -- and the equivalent of `eval()` is used to create the function, which could allow for many dangerous hacks, so don't do this! Just to whet your curiosity, let's see an example, 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, such as not creating closures with their creation contexts, and always being global instead. 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 using an arrow `=>` definition is the most compact way

to define a function and the one we'll try to use whenever possible. We'll get into more detail in the next section.

Arrow functions - the modern way

Even if arrow functions pretty much work as the other functions, there are some important differences with usual functions. These functions can implicitly return a value, 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.

Returning values

In lambda 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 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, or otherwise, JS will assume code is meant.



"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 always including them. However, the formatting tool I use, prettier, doesn't approve. Feel free to choose your style!"r!

Handling the this value

A classic problem with JS 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 problems are avoided. To see an example of the possible problems, in the following code, 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 an *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 JS5, plus 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`
- And 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 etas* section.*

Let's see the three solutions in actual code:

```
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
```

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 afterwards `func(...args)`? The key 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 JS? 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 JS5 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 needed to call `somethingElse()`, which requires using `.apply()`.

What would be the equivalent code in ES8? The answer is much shorter, and that's why we'll be seeing several examples of usage of the spread operator throughout the text:

```
function listArguments2(...args) {  
  console.log(args);  
  somethingElse(...args);  
}  
  
listArguments2(12, 4, 56);  
// (3) [12, 4, 56]
```

The points to remember are:

- 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 without any further ado.
- Writing `somethingElse(...args)` is much more clear than the alternative way (using `.apply()`) that we had to use earlier.

By the way, the `arguments` object is still available in ES8. If you want to create an array from it, you have two alternative ways of doing so, without having to recur to the `Array.prototype.slice.call` trick:

- use the `.from()` method, and write `var myArray=Array.from(arguments)`
- or even more simply, say `var myArray=[...arguments]`, which shows yet another type of usage of the spread operator

When we get to higher-order functions, writing functions that deal with other functions, with a possibly unknown number of parameters, will be commonplace. ES8 provides a much shorter way to do so, 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 the lambda calculus, you don't write functions with several parameters, but only with one, by applying something called *currying* (why would you do this? Hold that thought; we'll come to that).



Currying gets its name in memory of Haskell Curry, who developed the concept. Note that he is also remembered in the name of an FP language, Haskell; double recognition!

For instance, the function 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 previously. As is, though it can be used to produce the very same results as our earlier function, it differs in an important way: how do you use it? Say, to sum the numbers 1, 2, and 3? You would have to write:

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



Test yourself before reading on, and mull 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 in many calls can help; this would be the way the previous expression is actually calculated by the JS 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:

```
| 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 do `fn1(2)`, the result is again a function, also with a single parameter, equivalent to:

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

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

You might think 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*](#), or [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, or returned as result of yet other functions, in the very same way that you can do with, say, numbers or strings. Let's start with their definition. When you define a function in the usual way:

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

This is (almost) equivalent to writing:

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

Except for *hoisting*. JS moves all definitions to the top of the current scope, but not 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 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 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 used only once, then you don't need to name it or store it 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, and I included it just for completeness.

By taking advantage of the possibility of storing functions, we can build a *dispatch table* and simplify the preceding code. First, we would 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 have stored 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 usually done. 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)`... don't we already have such a function?



A small bit of theory: In lambda calculus terms, we are replacing $\lambda x. func\ x$ by simply a function -- this is called an eta conversion, 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, the rule we can apply is that whenever you see something like the following:

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

You may replace it 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 way (or, infinitesimally quicker, since you avoid one function call) but simpler to understand... or not?

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 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 you should be aware of: what happens if you are calling an object's method? If your original code had been something along the lines of:

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

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 it is just a `free` function. We can then fix it in a simple way by using `bind()` as:

```
| 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 the correct context will be available. Code like:

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

Should be converted to:

```
| 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 the method will be associated to 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 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. Let's then get to examine them, and point out the functional aspects of the code, so you can get more accustomed to this coding style.

Injection - sorting it out

A 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 something like the following code. For example, to alphabetically sort an array with the colors of the rainbow:

```
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... this needs fixing! The code below 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 keep 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 results wouldn't be correct, anyhow:

```
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 "i") 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:

```
palabras.sort((a, b) => a.localeCompare(b, "es"));
console.log(palabras);
// ["mano", "mítico", "musical", "natural", "ñandú", "oasis"]
```

The `a.localeCompare(b, "es")` call compares strings `a` and `b`, and returns a negative value if `a` should precede `b`, a positive value if `a` should 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 become clearer by introducing a new function, with an understandable name:

```
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: readers of the code, when they get to the sort, will immediately deduce what is being done, even if the comment wasn't present.



This way of changing the way the `sort()` function works by injecting different comparison functions, is actually a case of the Strategy design pattern. We'll be seeing 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:

- `sort()` only works with strings. If you want to sort numbers (as we tried to do earlier above) 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/lower case letters, whether to ignore punctuation and much more -- but be careful; not all browsers may support the needed 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 do 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.JS, reading a file is accomplished asynchronously with something like this:

```
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());
  }
});
```

The `readFile()` function requires a callback, which in this example is just an anonymous function, that gets called when the file reading operation is finished.

With a more modern programming style, you would use promises or `async/await`. For instance, when doing 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.

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 CPS - *Continuation Passing Style*. What is this way of coding? A way of putting it, is by thinking about this 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, if we grant this: you are allowed to pass a callback to the called function, so when that procedure is ready to return to the caller, instead of actually returning, it shall invoke the passed callback. In these terms, the callback provides the called function with the way to continue the process, and thus the name *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 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 may 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 any kind of control structures, 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?
(Note: 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 alternate callbacks. And this, by the way, can provide a solution to another problem: how would you 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 is to provide an alternative callback (that is, a different continuation) to be used whenever an exception would be thrown (in [Chapter 12, Building Better Containers - Functional Data Types](#), we'll find another solution, with *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)  
}
```


Polyfills

Being able to assign functions dynamically (in the same way 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 to the times when Ajax started to appear. Given those different browsers implemented Ajax calls in distinct fashions, you would always have to code around those differences:

```
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 re-do 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 so, 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:

```
(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() {
            return 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 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 IIFE, standing for *Immediately Invoked Function Expression*. The function runs, but *cleans after itself*, for all its variables are local, and won't even exist after the function runs. We'll see 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 instead of writing code like:

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

You'd very much prefer using the newer, clearer way, and just write:

```
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 run*, but only if needed. If `.includes()` is available, you need do nothing, but if it is missing, you then define a polyfill that will provide the very same workings.



You can find polyfills for many modern JS 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.



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 prone to errors. However, in 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 that) and you'll have to use a polyfill too, such as the one at <https://github.com/github/fetch>. Study the code, and you'll see it basically uses the same method as described previously, to see if a polyfill is needed, and to create it.

Stubbing

This is a use case similar in some aspects to polyfill: having a function do different work depending on the environment. The idea is to do *stubbing*, an idea from testing, which means replacing a function with another that does a simpler job, instead of doing the actual work.

A common case is the usage of logging functions. You may want the application to do detailed logging when in development, but not to say a peep when in production. A common solution would be writing something along the lines of:

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

This works, but as in the example about Ajax detection, it does more work than it about Ajax detection, it does more work than it needs since it checks every time if the application is in development. We could simplify the code (and get a really, really tiny performance gain!) if we stub out the logging function, so it won't actually log anything:

```
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 JS allows calling functions with more parameters than arguments, and that we aren't doing anything in `myLog()` when we are





not in development, we could have also written `() => {}` and it would have worked fine. I do prefer, however, keeping the same signature, and that's why I specified the `someText` argument, even if it wouldn't be used; your call!

Immediate invocation

There's yet another common usage of functions, usually seen in popular libraries and frameworks, that lets you bring into JS (even the older versions!) some modularity advantages from other languages. 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 passing some arguments to the function, which 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);
```

The pattern itself is called, as we mentioned, *Immediately Invoked Function Expression* -- usually simplified to 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. This helps the parser understand that we are writing an expression. If you were to omit the first set of parentheses, JS 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, because of JS's function scope, those definitions will be internal, and no other part of your code will be able to access it. Imagine 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 some function with the same name of any of the three here, and hoisting would imply that the *latter* 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 for the rest of the code, which helps keeping the global namespace less polluted:

```
(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 `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.

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 that, 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()` using arrow functions, instead of the *classic* way we used, with the `function` keyword?

3.3. One liner. Some line-of-codes-thrifty programmer suggested rewriting `doAction2()` as a one-liner... though formatting doesn't let it look so! 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;
```

Summary

In this chapter, we went over several ways of defining functions in JS, focusing mainly on arrow functions, which have several advantages over standard functions, including being more terse. We showed the concept of *currying* (which we'll be revisiting later), considered some aspects of functions as first class objects, and we finally considered several JS techniques that happen to be fully FP in concept.

In [Chapter 4](#), *Behaving Properly - Pure Functions*, let's delve even more deeply into functions, and thus introduce the concept of *pure functions*, which will lead us to even better style programming.

Behaving Properly - Pure Functions

In [Chapter 3](#), *Starting Out with Functions - A Core Concept*, we considered functions as the key elements in FP, went into detail about arrow functions, and introduced some concepts such as injection, callbacks, polyfilling, and stubbing. Now, in this chapter, we'll have the opportunity to revisit or apply some of those ideas, while we also...

- Consider the notion of *purity*, and why we should care about *pure functions*
- Examine the concept of *Referential Transparency*
- Recognize the problems implied by side effects
- Show some advantages of pure functions
- Describe the main causes of 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 may be considered to be pure if it satisfies two conditions:

- **Given the same arguments, the function always calculates and returns the same result**, no matter how many times it's invoked, or in which conditions you call it. This result value 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, or some other external variable, not directly controllable, value.
- **When calculating its result, the function doesn't cause any observable *side effect***, including output to I/O devices, mutation of objects, change to program 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 its scope, and do 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, and a simple example showing the difference between purity and idempotency follows. 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, 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 get into these conditions in more detail.

Referential Transparency

In mathematics, *Referential Transparency* is the property that lets you replace an expression with its value, and not change the results of whatever you were doing.



The counterpart of Referential Transparency is, appropriately enough, Referential Opacity. A referentially opaque function cannot guarantee always producing the same result, even when called with the same arguments.

To give a simple example, when an optimizing compiler decides to do *constant folding* and replace a sentence like:

```
| var x = 1 + 2 * 3;
```

with:

```
| var x = 1 + 6;
```

or, even better, directly with:

```
| var x = 7;
```

to save execution time, it's 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, that operation is called a β (beta) reduction. Note that you can only do this safely with referentially transparent functions.

All arithmetic 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 JS functions, you may 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 JS interpreter will return an undefined value in that situation.



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

If you wanted to, you could classify JS functions as:

- **pure functions:** that return a value depending only on its arguments, and have no side effects whatsoever
- **side effects:** that don't return anything (actually, JS has those functions return an `undefined` value, but that's not relevant here), but do produce some kind of side effects
- **functions with side effects:** meaning they return some value (which may not only depend on the function arguments and also involve side effects)

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

Side effects

What are *side effects*? We can define those as some change in state, or some interaction with outside elements (the user, a web service, another computer, whatever) that occurs during the execution of some calculations or 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 of *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 do a `console.log()` call to display some result, that would be considered a side effect, even if it's exactly what you intended the function to do in the first place!

Usual side effects

There are (too many!) things that are considered side effects. In JS programming, including both front- and back-end coding, the more common ones you may find, include:

- Changing global variables.
- Mutating objects received as arguments.
- Doing 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.
- And, finally, just calling some other 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 considering what can cause functional impurity (or *referential opaqueness*, as we saw).

Global state

Of all the preceding points, the most common reason is the usage of non-local variables, sharing 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 did, you must understand how the state got its current values, and that means understanding all the past history from your program: not easy!

```
let limitYear = 1999;

const isOldEnough = birthYear => birthYear <= limitYear;

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

The `isOldEnough()` function correctly detects if 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 possible.

There is an exception to this rule. Check out the following case: is the `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 substituting 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 (a better idea, by the way) the argument would still be the same; the constant cannot be changed, so the function remains pure.

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 if it has been rounding too much upwards or downwards, so next time it will round the other way, to make the accumulated difference closer to zero:

```
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 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 a same value rounded up or down, depending on the accumulated differences, because the function's result depends on its inner state.



This usage of internal state, is why many FPer consider 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 repeated method calls may return different values, despite the same arguments being passed.

Argument mutation

You also need to be aware of the possibility that an impure function will modify its arguments. In JS, 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 effect an actual modification of the original object or array. This can be furthermore 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:

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

Oops, the original array was modified; a side effect by definition! If you were to call `maxStrings(countries)` again, instead of returning the same result as before, it would produce another value; clearly, 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 sort 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"]
```

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 beat 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 thus 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 denomination for them.

For instance, consider this 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.



Check 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:

```
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 array `namePart`, by using `map()`.

Because of its usage of `getRandomLetter()`, `getRandomFileName()` is also impure, though it performs as expected:

```
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 consideration 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 the 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:

```
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 XIX 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, 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 also are impure are those that cause I/O. If a function gets input from some source (a web service, the user himself, a file, and so on), 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 time 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 generally safe, statement such as a `console.log()`, that doesn't change anything internally (at least in a visible way) does cause some 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 does I/O, and also use 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, derives from the fact that they don't have any side effects. When you call a pure function, you need not worry about anything, outside of 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 see more.

Order of execution

Another way of looking at what we have been saying in this chapter, is the consideration that pure functions can be called *robust*. You know that their execution --in whichever order-- won't ever have any sort of impact on the system. This idea may 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.



Unhappily, JS restricts us very much as to parallel programming. We may make do, in very restricted ways, with web workers, but that's about as far as it goes. For Node.js 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 JS 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*, by the way. However, when you deal with impure functions, that can be not true, as shown in the following code:

```
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 shown previously, 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 zero; check it out! 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 learnt about, such as the commutative property. So, while you might think that both expressions should produce the same result, and

code accordingly, with impure functions you may be in for a surprise, 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 re-calculation. 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 see an example done by hand. The Fibonacci sequence is always used for examples, 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!

If you do 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 -- though 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 oneliners, you could also write `const fib = (n) => (n<=1) ? n : fib(n-2)+fib(n-1)` -- do you see why? But, more important... is it worth the loss of clarity?

If you try out this function for growing values of n , you'll soon realize there is a problem, and computation starts taking too much time. For example, on my machine, these are some timings I took, measured in milliseconds -- 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, times for small values of n were really tiny; only from 25 onwards, I got interesting numbers. The chart (see Figure 4.1) shows an exponential growth, which bodes ill:

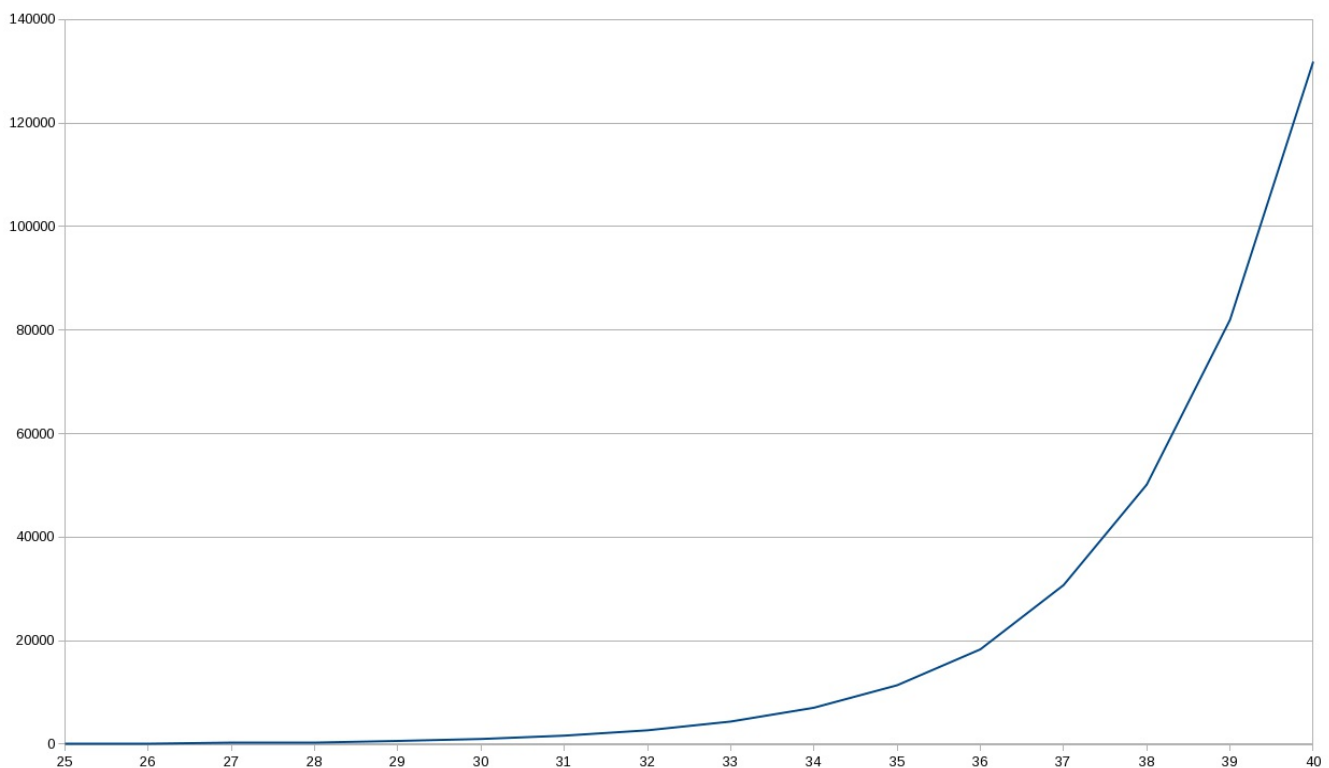


Figure 4.1: Calculation times for the `fib()` recursive function go up exponentially.

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; see Figure 4.2:

- Using an global variable for the cache isn't a very good practice; we could have used an IIFE and a closure to hide 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 that.

Of course, you need not 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 time -- if it were otherwise, the added cache management time would end costing more than whatever you expected to save!

Self-documentation

Pure functions have another advantage. Since all the function needs to work with is given to it through its parameters, having 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 some 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, for they provide examples of what the function returns, when given certain arguments. Most programmers will agree that the best kind of documentation is full with 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 much 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, with independence from the *rest of the world*. We'll see more about testing pure and impure functions, later in this very chapter.

Impure functions

If you decided to completely forego all kinds of side effects, 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 do any web services calls, or to update the DOM; you'd have static pages only. And, for server-side JS, your Node.JS code would be really useless, not being able to do any I/O...

Reducing side effects is a good goal in FP, but we cannot go overboard with it! So, let's think 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 impure functions. Let's now consider how we can minimize their number, if doing away with all of them isn't really feasible.

Avoiding the usage of state

With regard to the usage of global state --both getting and setting it-- the solution is well known. The key to this is:

- Provide whatever is needed of the global state to the function, as arguments
- If the function needs to update the state, it shall not do it directly, but rather produce a new version of the state, and return it
- It will 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 see more about this in [Chapter 10, Ensuring Purity - Immutability](#)).

With regard to our first version of the `isOldEnough()` function, that 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:

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

We'll have to change, obviously, all the calls to provide the needed `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 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. So, our original code (less comments and logging) would change from:

```
| const roundFix1 = (function() {
```

```

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

```

to:

```

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 afterwards, are now the responsibility of the caller code. You would have something like:

```

let accum = 0;

// ...some other code...

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

```

Note that:

- `accum` 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 to easily store each one in a different variable. For more on this, check 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 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 some other 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 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 replacing the impure function, with an injected external one:

```
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 care 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 that in the following examples. The usage of the function will change, and we would have to write:

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

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

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

```
| };
```

Or you may also solve this by 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. In normal use, you'll call `getRandomFileName()` 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 were 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:

```
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 specter of application to other problems. For example, instead of having a function directly access the DOM, we may provide it with injected functions that would do that. 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 a function is actually pure? To show the difficulties of this task, we'll go back to the simple `sum3()` function we saw in earlier chapters. 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 do 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 *OK, 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
```



Observe the way 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](#), to see more on this.

Well, `sum3()` ought to be pure... but it actually depends on whatever parameters you pass to it! You might console yourself 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. Adding some type checking (TypeScript might come in handy) you could at least prevent some cases -- though JS won't ever let you be totally sure that your code is *always* pure!

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 it in a pure way. So, rather on bemoaning the fact that you have to allow for impurity, try to structure your code so you can isolate the impure functions, and let the rest of your code be the best possible.

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, as to its difficulty and complexity, when you deal with pure or impure functions. While coding tests for the former is usually quite simple and follows a basic pattern, the latter usually require scaffolding and complex setups. So, let's finish this chapter by seeing how to go about testing both types of functions.

Testing pure functions

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

- 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 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);
  });
});
```

Another of the pure functions we wrote is equally simple, but for a care about precision. 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 that, 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):

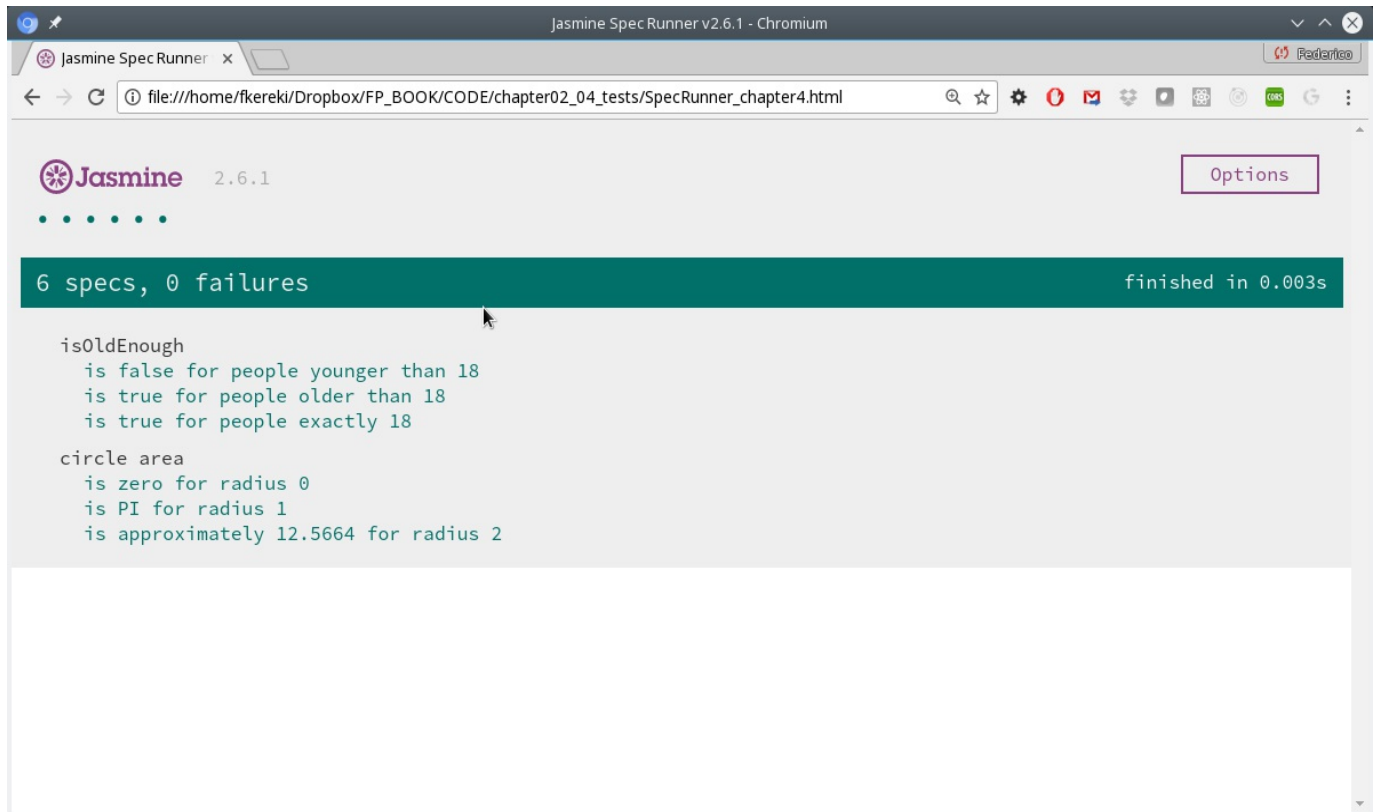


Figure 4.3: A successful test run for a pair of simple pure functions

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

Testing purified functions

When we considered the `roundFix` special function, which required using 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 produce no rounding errors:

```
describe("roundFix2", function() {  
  it("should round 3.14159 to 3 if differences are 0", () => {  
    let {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", () => {  
    let {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", () => {  
    let {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", () => {  
    let {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 if 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`, values close to 1 produce a `Z`, so we can have a little confidence that no extra values are produced. Also, 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!

```
describe("getRandomLetter2", function() {
  it("returns A for values close to 0", () => {
    let letterSmall = getRandomLetter2(() => 0.0001);
    expect(letterSmall).toBe("A");
  });

  it("returns Z for values close to 1", () => {
    let letterBig = getRandomLetter2(() => 0.99999);
    expect(letterBig).toBe("Z");
  });

  it("returns a middle letter for values around 0.5", () => {
    let letterMiddle = getRandomLetter2(() => 0.49384712);
    expect(letterMiddle).toBeGreaterThan("G");
    expect(letterMiddle).toBeLessThan("S");
  });

  it("returns an ascending sequence of letters for ascending values", () => {
    let a = [0.09, 0.22, 0.6];
    const f = () => a.shift(); // impure!!

    let letter1 = getRandomLetter2(f);
    let letter2 = getRandomLetter2(f);
    let 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; 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:

```
describe("getRandomFileName", function() {
  let a = [];
  let f = () => a.shift();

  beforeEach(() => {
    a = "SORTOFRANDOM".split("");
  });

  it("uses the given letters for the file name", () => {
    let fileName = getRandomFileName("", f);
    expect(fileName.startsWith("SORTOFRANDOM")).toBe(true);
  });

  it("includes the right extension, and has the right length", () => {
    let 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'll have 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*, in opposition to *black box testing*, in which 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 we did in the previous section. In the first case, we set `Math.random()` to return 0.0001, and we test that it was actually called, and also that the final return was `A`. In the second case, just for variety, we set things up so `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 a different way of checking how many times `Math.random()` (or, rather, our mocked function) was called:

```
describe("getRandomLetter", function() {
  it("returns A for values close to 0", () => {
    spyOn(Math, "random").and.returnValue(0.0001);
    let 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);
    let letterBig1 = getRandomLetter();
    let 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);
    let 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. Supposedly, 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, to have a bigger chance of catching a bug:

```
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", () => {
    let fileName1 = getRandomFileName(".pdf");
    expect(fileName1.length).toBe(16);
    expect(fileName1.endsWith(".pdf")).toBe(true);
  });

  it("doesn't include any extension if not provided", () => {
    let 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 may decide to implement the Fisher-Yates version, along the lines of the following. 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 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.

Questions

4.1. Minimalistic function: Functional programmers sometimes tend to write code in a minimalistic way. Can you examine this 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?
Suggestion: try to calculate `fib4(6)` by hand, and compare 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:
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

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

Summary

In this chapter, we have introduced the concept of *pure functions* and studied why they matter. We have also seen the problems caused by *side effects*, one of the causes of impure functions; considered some ways of *purifying* such impure functions, and finally, we have seen several ways of doing unit tests, for both pure and impure functions.

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, more powerful code.

Programming Declaratively - A Better Style

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

- `.reduce()` and `.reduceRight()` to apply an operation to a whole array reducing it to a single result
- `.map()`, to transform an array into another, by applying a function to each of its elements
- `.forEach()`, to simplify writing loops, by abstracting the necessary looping code

We'll also be able to do searches and selections with:

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

Using these functions lets you work more declaratively, and you'll see that your focus tends to go to what is needed to do, and not so much on 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 rather focus on using functions as building blocks to specify our desired result.

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

Transformations

The first set of operations 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()` operations; 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 usage of these higher-order functions, trying to avoid them, with longer code, and more probability of bugs simply doesn't make much sense.

Let's start by considering the preceding list of functions in order, starting by 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, doing some operation (say, summing elements) to produce a single value (maybe the sum of all the array values) as a result? Probably many, many times. This kind of operation can be usually 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 working of the `reduce()` function can be illustrated as in Figure 5.1:

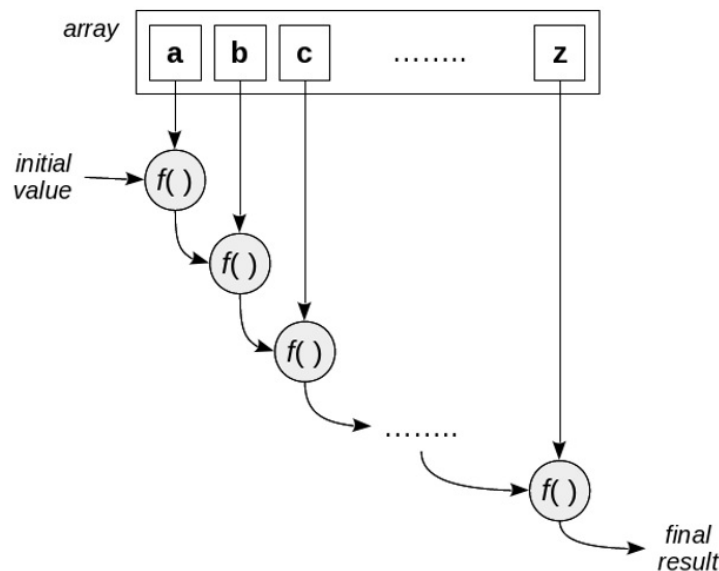


Figure 5.1: The reduce operation traverses the array, applying a function to each element and to the accumulated value

Why should you try to always use `.reduce()` or `.reduceRight()` instead of hand-coded loops?

- All the aspects of loop control are automatically taken care of, so you don't even have the possibility of a, say, *off by one* mistake

- The initialization and handling of the result values are also done implicitly
- And, unless you work really hard at being impure and modifying the original array, your code will be side-effects free

Summing an array

The most common example of application of `.reduce()`, usually seen in all textbooks and web pages, is summing all the elements of an array. So, in order to keep with tradition, let's start with precisely that 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 might have just written `myArray.reduce((x,y) => x+y, 0)`. However, in this fashion the meaning of the code is clearer: you want to reduce the array to a single value by `sum`-ming 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 by saying 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 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. See https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Array/Reduce for more. However, be careful if the array is empty, and if you skipped providing an initial value, you'll get a runtime error!

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:

```
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 (zero) and the first element of the array, and 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 1st part) should now be clear: the reducing operation proceeds from left to right, from the first element up 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!

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 somewhat be 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 zero, and divide by the array's length -- simpler, impossible!



As we mentioned in the previous section, you could have also written `arr.reduce(sum)` without specifying the initial value (zero) for the reduction; it's even shorter and closer to the verbal description of the needed 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 and the array as well, so you could do something different the last time:

```
const myArray = [22, 9, 60, 12, 4, 56];  
  
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
```



Getting the array and the index means you could also turn the function into an impure one; avoid this! Everybody who sees a `.reduce()` call,

TIP

will automatically assume it's a pure function, and will surely introduce bugs when using it.

However, from the point of view of legibility, I'm certain we agree that the first version we saw, was more declarative and closer to the mathematical definition, than this second version.

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

```
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
```

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 a case providing a clear advantage for common loops, but there's a trick you may use. Let's revisit yet once more 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 objection to returning an object, with as many fields as desired:

```
const average3 = arr => {
  const sc = arr.reduce(
    (ac, val) => ({ sum: val + ac.sum, count: ac.count + 1 }),
    { sum: 0, count: 0 }
  );
  return sc.sum / sc.count;
};

console.log(average3(myArray)); // 27.166667
```

Examine the code carefully. We need two variables, for the sum and the count of all numbers. We provide an object as the initial value for the accumulator, with two properties set to zero, 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 just see an example with an array:

```
const average4 = arr => {
  const sc = arr.reduce((ac, val) => [ac[0] + val, ac[1] + 1], [0, 0]);
  return sc[0] / sc[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 as a (not very recommendable) alternative way of calculating many values at once!

Folding left and right

The complementary `.reduceRight()` method works just as `reduce`, 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:

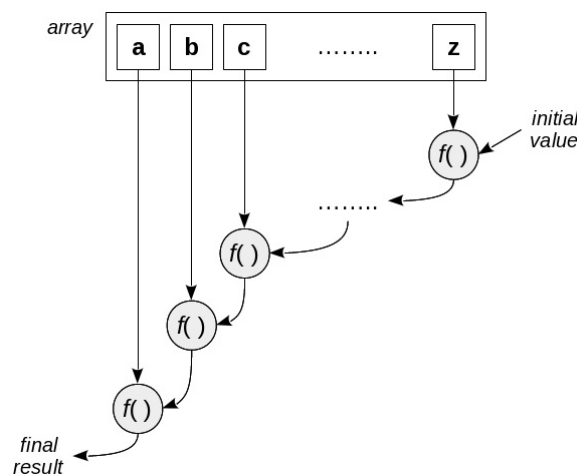


Figure 5.2: The `.reduceRight()` operation works the same way as `.reduce()`, but in reverse order



Read more on `.reduceRight()` at https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Array/ReduceRight.

Suppose we want to implement a function to reverse a string. A solution could be transforming 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 (and yes, it can be shortened, but that's not the point here) works, 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 have also 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. 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; rather keep using `.reduceRight()`!

However, we can draw another conclusion, showing a result we had foretold: it is possible, even if 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.

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 doing some kind of process to 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 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, it 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 pre-specified properties, that we shall see later. And, in Category Theory of which we'll talk a little in [Chapter 12](#), Building Better Containers - Functional Data Types, the mapping operation itself would be called a morphism.

The inner working of the `.map()` operation can be seen in Figure 5.3:

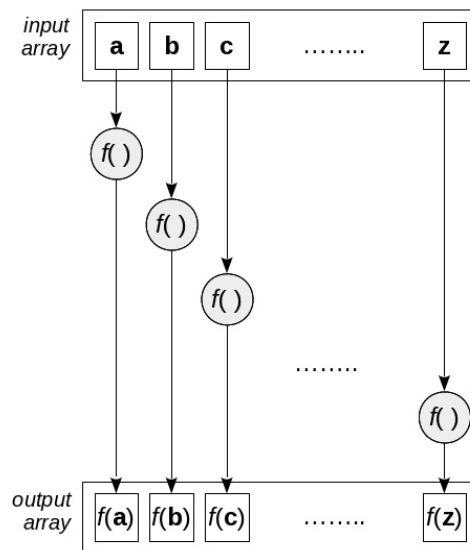


Figure 5.3: The `map()` operation transforms each element of the input array, by applying a mapping function



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 fewer 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
- And lastly, a new array is produced, so your code is pure (though, of course, if you really want to produce side-effects, of course, you can!)



*In JS, `.map()` is basically available only for arrays. (Read more at https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Object/s/Array/map.) However, in section *Extending current data types of Chapter 12, Building Better Containers - Functional Data Types*, we will consider how to make it available for other basic types, such as numbers, booleans, strings, and even functions. Also, libraries such as *LoDash* or *Underscore* or *Ramda*, provide similar functionalities.*

There are only two caveats when using this:

- Always return something from your mapping function. If you forget this, since JS always provides a default `return undefined` for all functions, you'll just produce an `undefined`-filled array.
- If the input array elements are objects or arrays, and you include them in the output array, then JS will still allow the original elements to be accessed.

Extracting data from objects

Let's start with a simple example. Suppose we have some geographic data as shown in the following snippet, related to countries and the coordinates (latitude, longitude) of their capitals. Let's say we happened to 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: "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}  
];
```



In case you are wondering if and why all the data are negative, it's just because the shown countries are all south of the Equator, and west of Greenwich. 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 below, when we study the `some()` and `every()` methods.

We would want to use our `average()` function (that 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. 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));  
let averageLon = average(markers.map(x => x.lon));
```

If you had extended `Array.prototype`, you could have then written an equivalent version, in a different style:

```
let averageLat2 = markers.map(x => x.lat).average();  
let averageLon2 = markers.map(x => x.lon).average();
```

We will be seeing more about these styles in [Chapter 8, Connecting Functions - Pipelining and Composition](#).

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]
```

When you use `parseFloat()` to get floating point results, everything's OK. However, if you wanted to truncate results to integer values, then the output really goes awry... what's happening?

The answer lies in a problem with tacit programming. (We already saw 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. See 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, that is 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.



See more on `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.

Working with ranges

Let's now turn to a helper function, which will come handy for many usages. 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 or 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)` that produces `[0, -2, -4, -6]`, but for our needs, the version we wrote is enough. See the Questions section at the end of this chapter.

How can we use it? In the following section, we'll see some usages for controlled looping with `forEach()`, but we can redo our factorial function by applying `range()` and then `reduce()`. The idea for this is simply to 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 that is that the produced range is empty (the call is `range(1, 1)` that returns an empty array) and then `reduce()` doesn't do any calculations, and simply returns the initial value (one), which is correct.



In [Chapter 8, Connecting Functions - Pipelining and Composition](#), we'll have 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 generating a range with the ASCII codes for the alphabet and mapping 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.

Emulating map() with reduce()

Earlier in this chapter, we saw how `reduce()` could be used to implement `reduceRight()`. Now, let's see that `reduce()` can also be used to provide a polyfill for `map()` --not that you will need it, for browsers usually provide both methods, but just to get more ideas 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 a simple array and function:

```
| 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 `.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.

More general looping

The examples that we've seen above, simply looping through an array. However, sometimes you need to do some loop, but the required process doesn't really fit `.map()` or `.reduce()`... so what can be done? 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.) JS 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, you might have also written `myCopy={...myObj}`, but where's the fun in that? OK, that is better, but I needed a nice example to use `.forEach()`... 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, un-modifiable 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 see more about this later in the book.

If you use the `range()` function we defined previously, you can also do 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 different than 1. This would practically allow you to replace all the loops in your code by `.forEach()` loops.

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.



A bit of terminology: the word predicate can be used in several senses (as in Predicate Logic) but for us, in computer science, we adopt the meaning a function that returns true or false. OK, this isn't a very formal definition, but it's enough for our needs. For example, we will filter an array depending on a predicate, and that 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 is having to filter the elements of an array according to some 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 similarly to `.map()`, `.filter()` doesn't alter the original array, but rather returns a new array with the picked items.

See Figure 5.4 for a diagram showing the input and output:

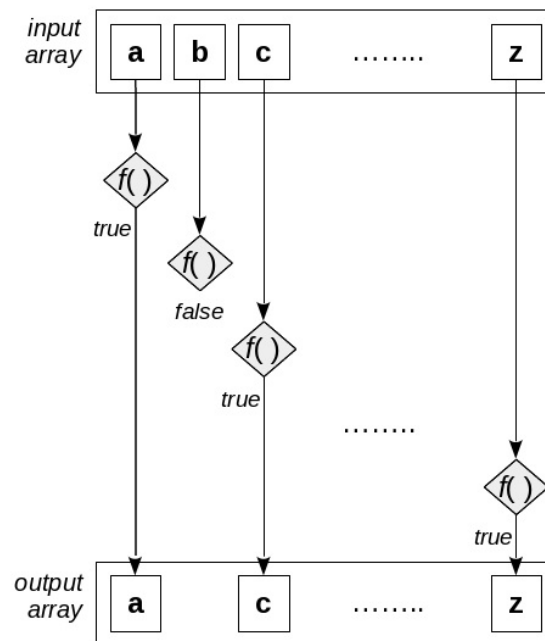


Figure 5.4: The `filter()` method picks the elements of an array, that satisfy a given predicate



Read more on the `.filter()` function at https://developer.mozilla.org/en/docs/Web/JavaScript/Reference/Global_Objects/Array/filter.

The things to remember when filtering an array are these:

- **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, the original elements will still be accessible.

A reduce() example

Let's see 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:

```
{
  accountsData: [
    {
      id: "F220960K",
      balance: 1024
    },
    {
      id: "S120456T",
      balance: 2260
    },
    {
      id: "J140793A",
      balance: -38
    },
    {
      id: "M120396V",
      balance: -114
    },
    {
      id: "A120289L",
      balance: 55000
    }
  ]
}
```

Assuming we got this data in a `serviceResult` variable, we could get the delinquent accounts with something like the following:

```
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 do so 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);
```

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 as 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()` does 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 was found

The analogy is clear to `.includes()` and `.indexOf()`, which search for a specific value, instead of for 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.

```
| 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
```

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 it? A tip: don't try checking the types of the array elements: even though `NaN` stands for *Not a Number*, `typeof NaN === "number"`... And 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 JS 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
```

Emulating `find()` and `findIndex()` with `reduce()`

As with 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 such 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 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 here `-1`, which will be the returned value in case 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.

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:

- `.every()`, that is `true` if and only *every* element in the array satisfies a given predicate
- `.some()`, that 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 && x.lon < 0); // false
markers.some(v => v.lat < 0 && x.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`, we transform one expression into the other, and both are valid.

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 earlier saw -- 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 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 do need `this` to be correctly assigned.

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.

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 JS 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 of [Chapter 11, Implementing Design Patterns - The Functional Way](#). The problem here is to use those functions (and none others!) to produce an unordered list of elements (`<ul>...</ul>`) 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 examples preceding, such as in the

Emulating `map()` with `reduce()` section we didn't write actual unit tests but rather 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, say, for descending ranges, like 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 CSV (comma separated value) file, by using a data URI. (See more at https://developer.mozilla.org/en-US/docs/Web/HTTP/Basics_of_HTTP/Data_URIs /.) Of course, the first problem is to produce the CSV itself! Assume 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"
```

Summary

In this chapter, we have started working with higher-order functions, so as to show a more declarative way of working, with shorter, more expressive code. We have gone over several operations: we have seen `.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; `.filter()`, to pick elements from an array; `.find()` and `.findIndex()`, to search in an array, and `.every()` and `.some()`, to verify general logic conditions.

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.

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 let us write declarative code, gaining in understandability as well as in compactness. In this new chapter, we are going to go further in the direction of higher-order functions, and we are going to develop our own. We can roughly classify the kinds of functions that we are going into three groups:

- **Wrapped functions**, that keep their original functionality, 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* (that caches results to avoid future re-work).
- **Altered functions**, that differ in some key point with their original versions. Here we can include the `once()` function (we wrote it in [Chapter 2](#), *Thinking Functionally - A First Example*) that changes the original function to run only once, functions such as `not()` or `invert()` that alter what the function would return, and arity-related conversions that produce a new function with a fixed number of parameters.
- **Other productions**, that provide for new operations, turn functions into promises, provide enhanced search functions, or allow decoupling a method from objects, so we can use them in other contexts as if they were common functions.

Wrapping functions

In this section, let's consider some higher-order functions that provide a *wrapper* around other function, to enhance it in some way, but without altering its original objective. In terms of *design patterns* (which we'll be revisiting in [Chapter 11, Implementing Design Patterns - The Functional Way](#)), we could 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 JS.



Decorators are being considered for general adoption in JS, but are currently (August 2017) 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 decorators for JS at <https://tc39.github.io/proposal-decorators/> and about the JS adoption process itself, called TC39, in <https://tc39.github.io/process-document/>. See the Questions section in [Chapter 11, Implementing Design Patterns - The Functional Way](#).

As for the term *wrapper*, it's more important and pervasive that 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 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: a wrapper allows methods to be called on things that are not of the appropriate type -- and that's an interesting idea; see [Chapter 12, Building Better Containers - Functional Data Types](#), for more applications of that!

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, and 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, that means that you'll have to modify the code of the function itself, both at entry and on exit. You'll have to code such as the following:

```
function someFunction(param1, param2, param3) {  
    // do something  
    // do something else  
    // and a bit more,  
    // and finally  
    return some expression;  
}
```

to something like:

```
function someFunction(param1, param2, param3) {  
    console.log("entering someFunction: ", param1, param2, param3);  
    // do something  
    // do something else  
    // and a bit more,  
    // and finally  
    let 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. Of course, if you are just calculating the return expression on the fly, you'll need an auxiliar variable to capture that value.

Logging in a functional way

Doing this is not 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.

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:

1. Log the received arguments.
2. Call the original function, catching its returned value.
3. Log that value; and finally.
4. Return 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, was returned



If you were doing this for a Node.js application, you would probably opt for a better way of logging, by using libraries such as Winston, Morgan, or Bunyan -- but our focus is in showing how to wrap the original function, and the needed changes for using those libraries would be small.

For an example, we can use it with an upcoming functions -- which are written, I

agree, in an overtly complicated way, just to have an appropriate example!

```
function subtract(a, b) {  
    b = changeSign(b);  
    return a + b;  
}  
  
function changeSign(a) {  
    return -a;  
}  
  
subtract = addLogging(subtract);  
changeSign = addLogging(changeSign);  
let x = subtract(7, 5);
```

The result of executing the last line would be the production of 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 of not reassigning the wrapped logging function in the Memoizing in the following section.

Taking exceptions into account

Let's enhance our logging function a bit, by considering a needed adjustment. What happens to your log if the function throws an error? Fortunately, that's easy to solve. We just have to add some 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;  
  }  
};
```

Other changes 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.

Working in a more pure way

When we wrote the `addLogging()` preceding function, we laid by the roadside some precepts we saw in [Chapter 4, *Behaving Properly - Pure Functions*](#), because we included an impure element (`console.log()`) right 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, certainly, 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:

```
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.

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 section *Injecting impure functions*: the logging function should be passed as an argument to the wrapper function, so 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.js, we could use *winston* and the results would vary accordingly:



See <https://github.com/winstonjs/winston> for more on the winston logging tool.

```
const winston = require("winston");
const myLogger = t => winston.log("debug", "Logging by winston: %s", t);
winston.level = "debug";

subtract = addLogging3(subtract, myLogger);
changeSign = addLogging3(changeSign, myLogger);
let x = subtract(7, 5);

// debug: Logging by winston: entering subtract: 7,5
// debug: Logging by winston: entering changeSign: 5
// debug: Logging by winston: exiting changeSign: -5
// debug: Logging by winston: exiting subtract: 2
```

Now that we have followed our own earlier advice, we can take advantage of stubs. The code for testing is practically the same as before but we are using a stub, `dummy.logger()`, with no provided functionality or side effects, so it's safer all around. True: in this case, the real function that was being invoked originally, `console.log()`, could do no harm, but that's not always the case, so using a stub is recommended:

```
describe("after addLogging2()", function() {
  let dummy;

  beforeEach(() => {
    dummy = {logger() {}};
    spyOn(dummy, "logger");
  });
```



```

it("should call the provided logger", () => {
  let something = (a, b) => `result=${a}:${b}`;
  something = addLogging2(something, dummy.logger);

  something(22, 9);
  expect(dummy.logger).toHaveBeenCalledTimes(2);
  expect(dummy.logger).toHaveBeenCalledWith(
    "entering something: 22,9"
  );
  expect(dummy.logger).toHaveBeenCalledWith(
    "exiting something: result=22:9"
  );
});

it("a throwing function should be reported", () => {
  let thrower = (a, b, c) => {
    throw "CRASH!";
  };
  thrower = addLogging2(thrower, dummy.logger);

  try {
    thrower(1, 2, 3);
  } catch (e) {
    expect(dummy.logger).toHaveBeenCalledTimes(2);
    expect(dummy.logger).toHaveBeenCalledWith(
      "entering thrower: 1,2,3"
    );
    expect(dummy.logger).toHaveBeenCalledWith(
      "exiting thrower: threw CRASH!"
    );
  }
});
});

```

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.

Timing

Another possible application for wrapped functions is to record and log, in a fully transparent way, the timing of each function invocation.



If you plan to optimize your code, remember the rules: Don't do it, then Don't do it yet, and finally Don't do it without measuring. It has been often 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 parts of your application.

Somehow, 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 the same exact 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 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>



TIP

ance/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>.

Just to be able to fully try out the logging functionality, I modified the `subtract()` function, so it would throw an error if you attempted to subtract zero. You could also list the input parameters, if desired, for more information:

```
subtract = addTiming(subtract);  
  
let x = subtract(7, 5);  
// subtract - normal exit 0.105000000000001819 ms  
  
let y = subtract(4, 0);  
// subtract - exception thrown 0.09499999999999136 ms
```

The 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, that would receive three functions, and would produce a 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 it?

Memoizing

Back in [Chapter 4](#), *Behaving Properly - Pure Functions*, we considered the case of the Fibonacci function and saw how we could transform it, by hand, into a much more efficient version by means of *memoization*: caching calculated values, to avoid recalculations. For simplicity, let's now only consider functions with a single, non-structured parameter, and leave for later functions with more complex parameters (objects, arrays) or more than one parameter for later.



The kind of values we can handle with ease are JS's primitive values: data that isn't objects and has no methods. JS has six of these: `boolean`, `null`, `number`, `string`, `symbol`, and `undefined`. It's likely we would only see the first four as actual arguments. See more in <https://developer.mozilla.org/en-US/docs/Glossary/Primitive>.

Simple memoization

We will work with the Fibonacci function we mentioned, which is a simple case: it receives a single numeric parameter. The function, as we saw it, was the following:

```
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 did there 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 it automatically, in the same fashion as with other wrapped functions. The solution would be a `memoize()` function that wraps any other function, 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, first 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.) We are assuming here 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 auxiliar `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 times may vary, of course, but 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()`, and we should get shorter times -- or 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 web pages. We are timing `memofib()` -- 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, also, 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(testFib)(45); // 0.080 ms
addTiming(testFib)(40); // 0.025 ms
addTiming(testFib)(35); // 0.009 ms
```

Now, when calculating `fib(45)`, in fact, all the intermediate Fibonacci values (from `fib(0)` to `fib(45)` itself) are stored, so the forthcoming calls have practically no work to do.

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 saw 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 do the memoization thing; 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., or you can say that a function is unary, binary, ternary, and so on, or you can also say it's monadic, dyadic, triadic, and so on: take your pick!

```
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 directly use a non-primitive as a cache key. We could attempt to convert the value to a string with something like `strX = String(x)` but we'll have problems. With arrays, it seems it could work, but see these three cases:

```
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"
```

The 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, and 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 is a primitive value, use that argument directly as a cache key; in other cases, use the result of `JSON.stringify()` as 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. And, 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 know 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/>.

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()` to use a global cache, or to somehow allow 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 should omit trying 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 quite longer if the function were not 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. OK, it's possible, but not probable -- but your test isn't fully reliable.

Let's go then for a more direct analysis of the number of actual calls to the memoized function. Working with a non-memoized, original, `fib()`, we could first 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 fact that `fib(6)` equals 8 is easy to verify, but where do you get that the function is called 25 times? For the answer, let's revisit the diagram we saw earlier in [Chapter 4, Behaving Properly - Pure Functions](#):

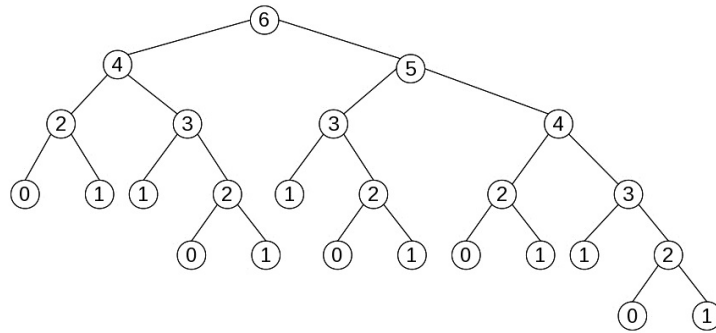


Figure 6.1. All the recursive calls needed for calculating `fib(6)`.

Each node is a call; just counting, we get 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 the question, let's

analyze the figure we saw earlier:

- First, we call `fib(6)`, which calls `fib(4)` and `fib(5)`: 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 done
- `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 -- and now, 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.

Altering functions

In the previous section, we considered some ways of wrapping functions, so they would maintain their original functionality, though enhanced in some ways. Now we'll turn to actually modifying what the functions do, so the new results will actually differ from the original function's ones.

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 a given function would work only once:

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

This is a perfectly fine solution, 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. That's easy to fix, however; all we require is adding a `return` statement. However, that wouldn't be enough; what would the function return if called more times? We can take a page out of the memoizing solution and store the function's return value for future calls:

```
const once2 = func => {  
  let done = false;  
  let result;  
  return (...args) => {  
    if (!done) {  
      done = true;  
      result = func(...args);  
    }  
    return result;  
  };  
};
```

You could also think of making the function work only once, but for each set of arguments... but you wouldn't have to do any work for that: `memoize()` would be enough!

Back in the mentioned [Chapter 2, *Thinking Functionally - A First Example*](#), 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 as earlier, 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:

```
const onceAndAfter2 = (f, g) => {
  let toCall = f;
  return (...args) => {
    let result = toCall(...args);
    toCall = g;
    return result;
  };
};
```

The very same example we saw before 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 just to 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, used everywhere in procedural programming. However, here we manage to skip that usage, and yet produce the same result.

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 it will return the opposite of whatever it originally returned. In the previously mentioned chapters, we saw this 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:

```
| 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:

```
| 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 ES8's syntax:

```
| const not = fn => (...args) => !fn(...args);
```

Working in this way, we could have rewritten the preceding filter as follows:

```
| 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()`:

```
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 that we'll be seeing in [Chapter 8, Connecting Functions - Pipelining and Composition](#). It's more interesting, though, 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 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, 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 more clear; 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?

Inverting results

In the same vein as the preceding filtering problem, let's now revisit the sorting problem from the *Injection - sorting it out* section of [Chapter 3, Starting Out with Functions - A Core Concept](#). We wanted to sort an array with some specific method, and we used `.sort()`, providing it with a comparison function that basically pointed out which of 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 zero if both 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 saw earlier for sorting in Spanish. We had to write a special comparison function, so sorting would take into account the special character order rules from Spanish, such as including letter *ñ* between *n* and *o*, and more.

```
| 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 a *descending* order? Given what we just 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 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 take change the result of a comparison. The code itself is quite similar to that of preceding `not()`:

```
| const invert = fn => (...args) => -fn(...args);
```

Given this higher-order function, we can now sort in descending order by just providing a suitably inverted comparison function:

```
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. By the way, 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 section tacitly* 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:

```
| ["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 the mentioned chapter, we went with an arrow function, that is a simple solution, with the added advantage of being clear to understand. In [Chapter 7, Transforming Functions - Currying and Partial Application](#), we will see yet another, based on partial application. But, here, 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 JS's spread operator and an arrow function, that's 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 so on functions that would turn any function into an equivalent, restricted-arity, version.

You may think 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 it would work with a fixed, non-variable, number of parameters.

If you want a nice list of JavaScript functions and methods, check out the pages at <https://developer.mozilla.org/en/docs/Web/JavaScript/Guide/Functions> and at https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects



TIP

[ference/Methods_Index](#). As for tacit (or point freestyle) programming, we'll be coming back to it in [Chapter 8](#), *Connecting Functions - Pipelining and Composition*.

Other higher-order functions

Let's end this chapter by considering other sundry functions that provide results such as new finders, decoupling method from objects, and more.

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 same chapter, in the section *Working with ranges*, to calculate a factorial, we then needed 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, shown as follows:

```
| const mySum = myArray.reduce(binaryOp("+"), 0);  
| const factorialByRange = n => range(1, n + 1).reduce(binaryOp("*"), 1);
```

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:

```
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!

```
const binaryOp2 = op => new Function("x", "y", `return x ${op} y;`);
```

If you follow this trail of thought, you may also define an `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 the upcoming [Chapter 7, *Transforming Functions - Currying and Partial Application*](#), we will see 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 the *Converting to free point style* section of [Chapter 8, Connecting Functions - Pipelining and Composition](#), we will need a function to check if a number is negative, and we'll consider using `binaryOp2()` to write it:

```
| const isNegative = curry(binaryOp2(">"))(0);
```

Never mind about the `curry()` function now (we'll get to it soon in [Chapter 7, Transforming Functions - Currying and Partial Application](#)) but the idea is that it fixes the first argument to zero, so our function will check, for a given number n , whether $0 > n$. The point here is that the function we just wrote isn't quite clear. We could do better if we defined a binary operation function that also let us specify one of its parameters, the left one or the right one, in addition to the operator to be used:

```
| const binaryLeftOp = (x, op) =>  
|   (y) => binaryOp2(op)(x, y);  
  
| const binaryOpRight = (op, y) =>  
|   (x) => binaryOp2(op)(x, y);
```

Alternatively, you could have gone back to the `new Function()` style of code:

```
| const binaryLeftOp2 = (x, op) => y => binaryOp2(op)(x, y);  
| const binaryOpRight2 = (op, y) => x => binaryOp2(op)(x, y);
```

With these new functions, we could simply write either of the following -- though I think the second is clearer: I'd rather test whether a number is less than zero, rather than whether zero 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, true -- but if you can do better, and simplify your coding, by also allowing to specify one of the two parameters for the operation, just do it! The idea of FP is helping 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 or binary operators or point freestyle or anything else, and I'd just write the following with no further ado:

```
| const isNegative3 = x => x < 0;
```

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, and 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 `.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.)

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:

```
const promisify = fn => (...args) =>
  new Promise((resolve, reject) =>
    fn(...args, (err, data) => (err ? reject(err) : resolve(data)))
  );
```

With this function, instead of writing code like this:

```
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("./doesn't exist.txt", cb); // failure, show exception
```

Instead, you can go with promises:

```
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 would be able to use `fs.promise()` instead of the original method. 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, but often used, 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:

```
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 had another example when we saw how to filter an array; in our example, we wanted to get the IDs for all accounts with a negative balance and, 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 as its result a new function that will be able to extract said attribute from an object. Using ES8 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, able to "go deep" into an object, to get any attribute of it, no matter its location within the object.*

With this function, the coordinates extraction 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 ID's.

```
| const getId = getField("id");  
| const delinquent = serviceResult.accountsData.filter(v => v.balance < 0);  
| const delinquentIds = delinquent.map(getId);
```

Be sure to 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 that's what `getField()` produces.

Demethodizing - turning methods into functions

Methods such as `.filter()` or `.map()` are only available for arrays -- but in fact, you could 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 -- which is rightly frowned upon, and not recommended at all... but do see 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 that, 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 preceding, in the *Logically negating a function*, regarding a possible `filterNot()` function in comparison to the `.filter()` method?) A decoupled method works in a fashion somewhat similar to what are called *generic* functions in other languages since they can be applied to diverse data types.

There are three distinct, but similar, ways to implement this in ES8. The first argument in the list will correspond to the object; the other arguments, to the actual ones for the called method;



See https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Function for explanations on `apply()`, `call()`, and `bind()`. By the way, back in [Chapter 1, Becoming Functional - Several Questions](#), we saw the equivalence between `.apply()` and `.call()` when you used the spread operator.

```
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: `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 see some applications for those! Starting with a simple example, we can use `.map()` to loop over a string, without first converting it into an array of characters. Say you wanted to separate a string into individual letters and turn them into upper case:

```
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 both `.map()` and `.toUpperCase()`, we can simply write the following:

```
const map = demethodize3(Array.prototype.map);
const toUpperCase = demethodize3(String.prototype.toUpperCase);

const result2 = map(name, toUpperCase);
// ["F", "U", "N", "C", "T", "I", "O", "N", "A", "L"]
```



Yes, for this particular case, we could have first 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 no less, would it?

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 = demethodize3(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 `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 the 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:

```
| 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 *lowest* interest rate could be the best, couldn't it? Assuming that we would always want 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 produce 4 -- but it's not easily understandable code.

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 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, that takes two elements as arguments, and returns the best of those. If you could associate a numeric value to 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:

```
| const findOptimum2 = fn => arr => arr.reduce(fn);
```

With this, we can easily replicate maximum- and minimum-finding 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 superheroes card game: each card represents a hero, with 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:

```
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:

```
function Hero(n, s, p, t) {
  this.name = n;
  this.strength = s;
  this.powers = p;
  this.tech = t;
}

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)
];

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 such that the first beats the second, the second beat the third, but the third beats

the first! In mathematical terms, this means that the comparison function is not transitive and you don't have a total ordering for the set.

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 `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 just write a single `opposite()` function that will behave as `not()` or `negate()` as needed?

Summary

In this chapter, we have seen how to write higher-order functions of our own, which can either wrap another function to provide some new feature, alter a function's objective so it will do something else, or even totally new features, such as decoupling methods from objects or creating better finders.

In [Chapter 7](#), *Transforming Functions - Currying and Partial Application*, we'll still keep working with higher-order functions, and we'll see how to produce specialized versions of existing functions, with predefined arguments, by currying and partial application.

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
- Something I'll call *partial currying*, that 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, with possibly 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.

A bit of theory

The concepts that we are going to use 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 which 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. OK, I agree, this isn't very clear! So, let's start by clearing the air, and providing some short definitions, which we will expand later. (If you feel that your eyes are glazing over, please just skip this section and come to it later!) Yes, you may find the following descriptions a bit perplexing, but bear with us: we'll be getting into more detail just in 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, the second function receives the second 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 a 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 preceding ideas: you provide n arguments (from left to right) to a 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 the 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 about interesting ways of JS coding, that 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 device 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) had been 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!

Dealing with many parameters

The idea of currying, by itself, is simple. If you need a function with, say, three parameters, instead of writing (with arrow functions) something like the following:

```
| const make3 = (a, b, c) => String(100 * a + 10 * b + c);
```

You can have a sequence of functions, each with a single parameter:

```
| const make3curried = a => b => c => String(100 * a + 10 * b + c);
```

Alternatively, you might want to consider them to be nested functions:

```
| 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 usual fashion, such as `make3(1, 2, 4)`, that wouldn't work with the second definition. Let's work out why: `make3curried()` is an *unary* (single parameter), so we should write `make3curried(1)...` but what does this return? According to the definition above, this also returns an 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:

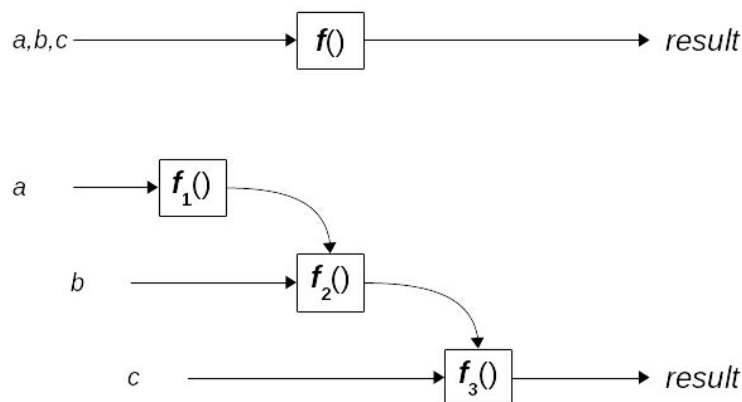


Figure 7.1. The difference between a common function, and a curried equivalent.

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 also exists an uncurrying one! In our case, we would write `make3uncurried = (a,b,c) => make3curried(a)(b)(c)` to revert the currying process and make usable, once again, to provide all parameters at a 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 see a simple example, and down below we will be seeing more examples. Suppose you had a function that calculated the VAT (*Value Added Tax*) for an amount:

```
| 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 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, that 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 a 6% tax.

Of course, you would be probably justified to say that this currying thing is a bit too much just to add a 6% tax, but the simplification is what counts. Let's see 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 see later:

```
| 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");
```

Currying by hand

If we want to implement currying just 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 like in the following examples:

```
sum(3, 5); // 8; did you expect otherwise?  
  
const add2 = sum(2);  
add2(3); // 5  
  
sum(2)(7); // 9 -- 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 desire. (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, can 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 fixate one argument (or more, if need be; we won't be needing 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 more) 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 more times 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 `curry()` always returns a new function, which depends on the function `fn` 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 doing `fn()`. Otherwise, the result of currying the function will be a new function, that receives a single argument, and produces itself 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);

const f1 = curryByBind(make3); // f1 is a function, that will fix make3's 1st parameter
const f2 = f1(6); // f2 is a function, that will fix make3's 2nd parameter
const f3 = f2(5); // f3 is a function, that will fix make3's last parameter
const f4 = f3(8); // "658" is calculated, since there are no more parameters to fix
```

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, that 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()`, that will by itself produce a curried version of `make3()` -- but with its first argument set to 6, so actually the

new function will end fixing the second parameter of `make3()`.

- Similarly, calling `f2(5)` produces yet a third unary function, `f3()`, that will produce a version of `make3()`, but fixing its third argument, since the first two had 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's 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 `curry2()` 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 code listed above, 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(sum(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(sum(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 by 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:

```
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:

```
const make3curried = x1 => x2 => x3 => make3(x1, x2, x3);
```

The needed code to generate this equivalent version is the following. 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 rather be coded in several separate lines to make it more understandable. Let's follow it, with the `make3()`

function as input:

- The `range()` function produces an array with values `[0, 1, 2]`. If we don't provide a `len` argument, `make3.length` (that is, 3) will be used.
- We use `.map()` to generate a new array with values `["x0", "x1", "x2"]`.
- We `join()` the values in that array to produce `x0=>x1=>x2`, which will be the beginning of the code that we will `eval()`.
- 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(`.
- We use `range()`, `map()`, and `join()` again, but this time to generate a list of arguments: `x0, x1, x2`.
- We finally add a closing parenthesis, and after applying `eval()`, we get the curried version of `make3()`:

```
| 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 the *About 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! All the following produce the same result:

```
| 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!

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 open the other three 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 we 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. Let's assume we have a `partial()` function that implements this kind of application and see how we'd use that.



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 go around this limitation with a polyfill, such as the one found at <https://github.com/github/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, because, 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 can simplify the problem, it will still remain hard to write and maintain. See Figure 7.2:

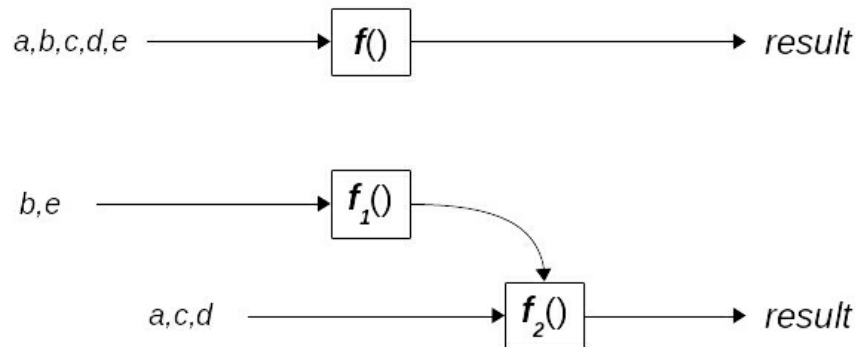


Figure 7.2. Partial application may let you first provide some parameters, and then provide the rest, to finally get the result.

Doing partial application with arrow functions, however, is much simpler. With the example we were mentioning above, 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 fix any number of parameters, and all you do is create a new function out the previous one but fixing some more parameters. For instance, you might now want to also fix the last parameter of the new `fix2and5()` function to 9; 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 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 not applied variables. 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!

```

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. What 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 in correct order

We could write something like the following, allowing for fixing 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 needed 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 of doing partial application, and consider another solution.

Partial application with closures

Let's examine yet another way of doing partial application, which will behave in a fashion somewhat reminiscent of the `curry()` functions we wrote earlier in this chapter, and solve the deficiency we mentioned at the end of the previous section:

```
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 function `partialize()`. 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 we construct a partial version of it:

```
const make3 = (a, b, c) => String(100 * a + 10 * b + c);
const f1 = partialByClosure(make3, undefined, 4);
```

Now we 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. If we now were to write:

```
| 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 earlier wrote in the *Currying with* `bind()` section:

- If all the arguments have been provided, the original function is called
- 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 out applying arguments in sequence, so we should get the final result after two or more steps of applications. However, since we can now call our intermediate functions with any number of parameters, we cannot test arities: for all functions, `function.length===0`:

```
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!

Partial currying

The last transformation we will see is a sort of mixture of currying and partial application. If you google around, in some places you 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:

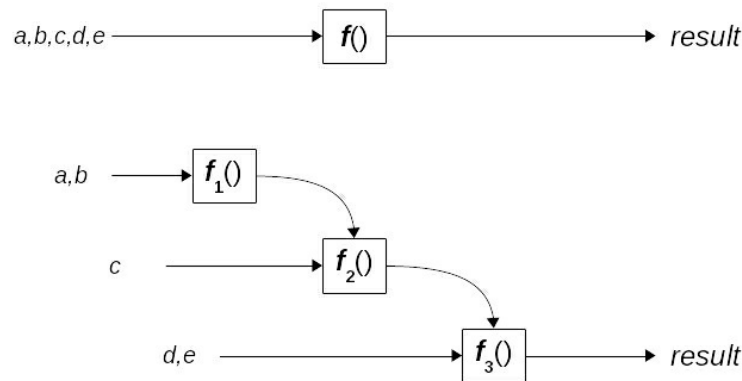


Figure 7.3. "Partial currying" is a mixture of currying and partial application. You may provide arguments from the left, in any quantity, until all have been provided, and then the result is calculated.

To see an example, let's go back to the `nonsense()` function we have been using in previous sections. 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 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:

- 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 needs not be so. However, if on 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:

```
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 fixing any number of arguments, we cannot test the arity of the intermediate functions:

```
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:

```

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 some pages earlier:

```

const sum = (...args) => args.reduce((x, y) => x + y, 0);

pcSum5 = partialCurryingByBind2(sum, 5); // curriedSum5 will expect 5 parameters
pcSum5(1, 5)(3)(7, 4); // 20

```

The new `pcSum5()` function first collected two arguments (1,5) and produced a new function that expected three more. One single parameter was given (3) and a third function was created, to wait for the last two. When those two (7,4) were provided, 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:

```
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, that 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 curried();
};
```

The results are exactly the same as in the previous section, *Partial currying by 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, that many libraries are just wrong as to the order of their parameters, making them harder to use
- Second, that I don't usually even use the higher-order functions in this chapter, going for simpler JS code!

That's not probably what you were expecting by 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, 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 below, 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:

```
| 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 case of use 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 the 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 I thought it important that you should be 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(reduce)));
const sum = (x, y) => x + y;

const sumAll = myReduce(sum, 0);
sumAll(anArray);
sumAll(anotherArray);
```

I used partial currying, to simplify the expression for `sumAll()`. The alternative would have been using common currying, and then I 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 rather 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 usage of arrow functions, I think the other solutions aren't really better:

```
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(map);
const myMap = curry(_plainMap, 2);
const myMapX = curry(flipTwo(_plainMap));

const _plainReduce = demethodize(reduce);
const myReduce = curry(_plainReduce, 3);
const myReduceX = curry(flip3(_plainReduce));

const _plainFilter = demethodize(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 (initial value for reducing) to `.reduce()`, so the arity I chose for that function is three.
- When currying the 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 saw in this chapter and see which ones you prefer!

Questions

7.1. Sum as you will. The following exercise will help you understand some of the concepts we dealt with above, even if you solve it without using any of the functions we saw in the chapter. 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 text. Completing the code below 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 a function `unCurry(fn, arity)` 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
```

Summary

In this chapter, we have considered a new way of producing functions, by fixing the arguments to an existing function in several different ways: currying, a theoretical way; partial application, with more flexibility; and partial currying, which combines good aspects from both 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 saw in the chapter on pure functions, and we will be considering ways of ensuring that functions cannot become *impure by accident*, by seeking for ways to make their arguments immutable, making them impossible to mutate.

Connecting Functions - Pipelining and Composition

In [Chapter 7](#), *Transforming Functions - Currying and Partial Application*, we saw several different ways to build new functions by applying higher-order functions. In this chapter, we will go to the core of FP and see how to create sequences of function calls, so their combination will produce a more complex result out of several simpler components. We will include the following:

- **Pipelining**, a way to join functions together in similar way to Unix/Linux pipes
- **Chaining**, which may be considered a variant of pipelining, but restricted to objects
- **Composing**, which is a classic operation with its origins in basic computer theory

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 of composed or piped functions, for which we'll whip up some auxiliary tools
- Testing of composed or piped functions, which won't prove to be of high complexity

Pipelining

Pipelining and composition are techniques for setting up functions to work in sequence, so the output from a function becomes the input to the next function. There are two ways of looking at this: from a computer point of view and from a mathematical point of view. Usually, most FP texts start with the latter, but since I assume that most readers are closer to computers than to math, let's start with the former.

Piping in Unix/Linux

In Unix/Linux, the execution of a command and passing its output as an input to a second command, whose output will yet the input of a third command, and so on, is called a *pipeline*. This is quite common, and an application of the philosophy of Unix, as explained in a Bell Laboratories article, written by the creator of the pipelining concept himself, Doug McIlroy:

1. Make each program do one thing well. To do a new job, build afresh rather than complicating old programs by adding new *features*.
2. Expect the output of every program to become the input to another, as yet 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 `cd /home/fkerekki/Documents` and then do the following:

```
$ ls -l | grep "odt$" | wc -l
4
```

What does this mean? How does it work? (Ignore the dollar sign: it's just the console prompt.) We have to analyze the 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 file name 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 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 above.

From the point of view of FP, this is a key concept. We want to build up more complex operations out of simple, single-purpose, shorter functions. Pipelining is the way the Unix shell uses to apply that concept, simplifying the job of executing a command, taking its output, and providing it as an input to yet another command. We will be applying similar concepts in our own functional style in JS, as we'll see; check out Figure 8.1:

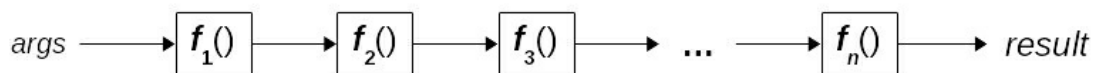


Figure 8.1. Pipelines in JS are similar to Unix/Linux pipelines. The output of each function becomes the input for the next.

By the way (and, no, rest assured, this isn't turning into a shell tutorial!) you can also make pipelines to accept parameters. For example, if I happened to often desire 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  
}
```

I could then use `cfe` as a command, giving it the desired extension as an argument:

```
$ cfe odt  
4  
$ cfe pdf  
6
```

We will also want to write similar parametric pipelines: we are not constrained to only have fixed functions in our flow, but rather have full liberty as to what we want to include.

Revisiting an example

We can start tying ends together by revisiting a problem from earlier chapters. Remember having to calculate the average latitude and longitude for some geographic data, which we saw in the *Extracting data from objects* section of [Chapter 5, Programming Declaratively - A Better Style?](#) Basically, we started with some data like the following and the problem was to calculate the average latitude and longitude of the given points:

```
let 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}
];
```

With what we now 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 the *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](#), plus 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. Written out in long, our solution could be the following:

```
const average = arr => arr.reduce(sum, 0) / arr.length;
const getField = attr => obj => obj[attr];
const myMap = curry(flipTwo(demethodize(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 mind: is it really clearer, 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, so let's now turn to seeing how we can pipeline functions in the right way.

Creating pipelines

We want to be able to generate a pipeline of several functions. We can go at 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 see both types of solution.

Building pipelines by hand

Let's go with a Node.js example, similar to the command-line pipeline we built earlier in this chapter. We need a function to read all files in a directory and we can do that (in a not very recommendable way, because of the synchronous call, 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));
```

So, we can now write the following:

```
const filterOdt = arr => filterByText(".odt", arr);
```

Better still, we can applying currying, and go for pointfree style, as seen in the *An unnecessary mistake* section of [Chapter 3, Starting Out with Functions - A Core Concept](#):

```
const filterOdt2 = curry(filterByText)(".odt");
```

Finally, to count elements in an array, we can simply write the following code. Since `.length` is not a function, we cannot apply our demethodizing trick:

```
const count = arr => arr.length;
```

With these functions available, 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/fkerekki/Documents"); // 4, as with the command line solution
```

If you wanted to get rid of all the intermediate variables, you could also go for a *one-*

liner definition:

```
const countOdtFiles2 = path => count(filterOdt(getDir(path)));  
countOdtFiles2("/home/fkerekki/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 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 -- but those functions appear *the other way round* in our definition!

We certainly can implement pipelining by hand, as we have seen, but it will be better if we can go for a more declarative style. Let's then move on to try to build a better pipeline in a more clear and understandable way, trying to apply some of the concepts we have 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, this could do:

```
| const pipeTwo = (f, g) => (...args) => g(f(...args));
```

This is not so useless as it may seem because we can compose longer pipelines -- though, I'll admit, it requires too much writing! We can write our three functions' pipeline 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 does this work? Following in detail 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 that fortunately matches what we already knew:

```
| countOdtFiles3("/home/fkerekki/Documents") ===  
  pipeTwo(pipeTwo(getDir, filterOdt), count)("/home/fkerekki/Documents") ===  
  count(pipeTwo(getDir, filterOdt)("/home/fkerekki/Documents")) ===  
  count(filterOdt(getDir("/home/fkerekki/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
```

OK, so we now know that we can make do with just a basic *pipe of two* higher-order function... 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/fkereki/Documents"); // still 4
```

This does work -- and the way of specifying our file-counting pipeline is much clearer since now the functions are given in their proper order. However, the implementation itself of the `pipeline()` function is not very functional and rather goes back to old, imperative, loop by hand methods. We can do better using `.reduce()`, as in [Chapter 5, Programming Declaratively - A Better Style](#).



If you check some FP libraries, the function that we are here calling `pipeline()` may also be known as `flow()` - because data flows from the left to the 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 evaluation with the first function, pass the result to the second, then that result to the third, and so on. We can then achieve our pipelining with shorter code:

```
const pipeline2 = (...fns) =>
  fns.reduce((result, f) => (...args) => f(result(...args)));

pipeline2(getDir, filterOdt, count)("/home/fkereki/Documents"); // 4
```

This code is more declarative, and you could have even gone one better by writing it using our `pipeTwo()` function, which does the same:

```
const pipeline3 = (...fns) => fns.reduce(pipeTwo);

pipeline3(getDir, filterOdt, count)("/home/fkereki/Documents"); // again 4
```

You can understand this code also by realizing that, basically, it uses the associative property that we mentioned, and it first 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 possibly not so simple to understand.

Debugging pipelines

Now, let's turn to a practical question: how do you debug your code? With pipelining, you cannot 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 (most appropriately 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 may execute something as follows 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 quite more concise, and just write `const tee = (arg) => (console.log(arg), arg)` -- do you see why? Check https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/Comma_Operator for the answer!

Our logging function will receive a single argument, list it, and pass it on to the next function in the pipe. We can see it working:

```
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 doing 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.

However, this would just be a particular enhancement: let's now 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 creating personalized tapping functions. See Figure 8.2:

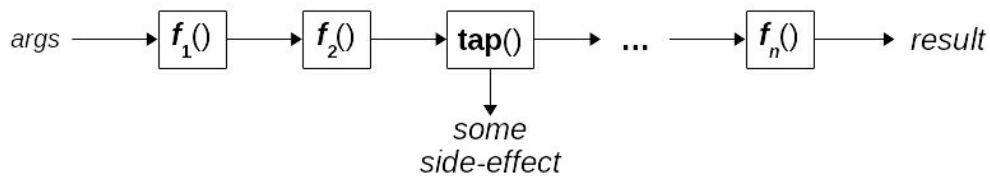


Figure 8.2. Tapping allows you to apply some function to inspect data as it flows through the pipeline.

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 storing data somewhere, or calling a service, or some other kind of side effect. We could have a generic `tap()` function, which would behave in this way:

```
| const tap = curry((fn, x) => (fn(x), x));
```

This is probably a candidate for a *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 of 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 code like `(a, b, c)`, JS will evaluate the three expressions in order and will use the last value as the expression's value.



The comma has several uses in JS 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.

We can now 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 admit it loses something of its mystery!

```
| const tap2 = fn => x => (fn(x), x);
```

You'll recognize this way of currying as we saw it in the *Currying by hand* section of [Chapter 7, Transforming Functions - Currying and Partial Application](#).

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 need not add logging to *every* function in the pipeline: you would probably do in the places where you suspected an error was occurring.

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, to which you can apply a new further method, and so forth. We have already used such methods, as 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 did generate a new array, to which we could have applied further mappings, filtering, or any other available method.

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 -- and the following example, taken from <https://bl.ocks.org/mbostock/4063269> shows it:

```
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 either provides access to a new object to which future method calls will apply (such as the `.selectAll()` or `.append()` methods) or updates the current one (as with the `.attr()` attribute setting calls). This style is not unique and several other well-known libraries (jQuery comes to mind, just to start) 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()` is just as well, and manages the same result. With object chaining, you are limited to returning new objects or arrays or something

to which methods can be applied. (Remember, if you are working with standard types, such as strings or numbers, you cannot 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 some other people'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 they will also cause an (albeit slight) slowdown in your proxied code.

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];
  }
}
```

We could use this class as follows, with details for 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 a fluent handling of the setters, 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, JS will include by default a `return undefined` statement, so 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 a 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 directly 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);
```

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 fluent fashion and they are working fine since our proxy is taking care of providing the needed 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 this worth it? That's up to you -- but remember my comment that there may be cases in which this approach fails, so be wary!

Pointfree style

When you join functions together, either in pipeline fashion as here, or with composition as we'll be seeing 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, and this is called pointfree style.



Pointfree style is also called tacit programming -- and pointless programming by detractors! The term point itself means a function parameter and pointfree refers to not naming those parameters.

Defining pointfree functions

You can easily recognize a pointfree function definition because it doesn't need either the `function` keyword or the `=>` symbol. We can revisit some of the previous functions we wrote in this chapter and check that out. For example, the definition of our original file-counting functions:

```
| 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 do not make reference to the parameter for the newly defined functions. You can deduce it 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 a good 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 just saw it in the *Revisiting an example* section), to decide that it expects an object as an argument. However, making that need explicit by writing:

```
| const getLat = obj => curry(getField)("lat")(obj);
```

This wouldn't make much sense: if you were willing to write all this, you might simply stick with the following:

```
| const getLat = obj => obj.lat;
```

Then you could simply not care about currying or anything like it!

Converting to pointfree style

On the other hand, you had better pause a bit, and not try 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 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 and the next will check whether it's negative, so we will write our alternative version of the balance-checking function in a fashion like the following:

```
| 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:

```
| const isNegative = x => x < 0;
```

There goes our pointfree goal! Instead, we can use the `binaryOp()` function, also from the same earlier cited chapter, 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 using the enhanced functions I mentioned in the *A handier implementation* section of the same chapter -- a bit less complex!

```
| 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 advance? 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 poin-free programming, stop and roll back your changes. Remember our doctrine for the book: we want to do FP, but we don't want to go overboard with it -- and using pointfree style is not a requirement!

Composing

Composing is quite similar to pipelining, but has its roots in mathematical theory. The concept of composition is simple - 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. In the latter, the first function to be applied is the leftmost one, but in 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 it to x , this is equivalent to what you would write as $f(g(h(x)))$. It's important to note that, as with pipelining, the arity of the first function to be applied 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) thus 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 it becomes clear that h is the first function to be applied, and f is the last.

Given the 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.

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 also work with some new examples.

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 in order to negate a check for negative balances; sample code 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 the challenge of writing a `unaryOp()` function that would provide unary functions equivalent to common JS operators. So, if you are able to write the following:

```
| const logicalNot = unaryOp("!");
```

Then, 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 see just one more example in the same vein, you could have managed to get the same results as in the *Inverting results* section, in the same chapter. Remember, we had a function that could compare strings according to the Spanish language rules, but we wanted to invert the sense of the comparison, to sort in descending order:

```
| const changeSign = unaryOp("-");  
| palabras.sort(compose(changeSign, spanishComparison));
```

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 so 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 better 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 so 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 -- that's easy to see, and applies the idea of building large solutions out of simpler pieces of code.

Finding unique words

Finally, let's go for another example, which, I agree, could also have been used for pipelining. Suppose you have a text, and you want to extract all unique words from it: how would you go about it? 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:

- Ignore all non-alphabetic characters
- Put everything in uppercase
- Split the text into words
- 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.

Now, in FP way, 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 to any of the composed functions, you really don't need to show the parameter for `getUniqueWords()` either, and so pointfree style is a natural in this case.

We can test our function out; let's apply this function to the first two sentences of Abraham Lincoln's address at Gettysburg, from November 19, 1863, 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 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 might have written `getUniqueWords()` in possibly shorter ways, 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!

Composing with higher order functions

It's pretty obvious that composing by hand could easily be done in a similar fashion as we saw above with pipelining. For example, the unique word counting function that we wrote a couple of sections earlier, could be written in simple JS 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' ]
```

However, as with pipelining, let's go 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 just a small change with regard to our `pipeTwo()` function, which we saw 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, and with composing, you start with the rightmost one. This variation suggests 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?

```
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, 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 -- and 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 as with composition, by reversing the order of functions and doing pipelining instead; a very functional solution, which I really like!

```
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 once again spread its elements, to correctly call `pipeline()`.

The other solution, less declarative, is by using `.reduceRight()` so 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 working of this call. We can replace `pipeTwo()` with its definition, to make this clearer:

```

const compose2b = (...fns) =>
  fns.reduceRight((f,g) => (...args) => g(f(...args)));

```

OK, let's see!

- Since no initial value is provided, the first time `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, and `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 last time, the result is `(...args) => setToList(step3(...args))`, a function `step4()`

The final result correctly works out to be a function that receives `(...args)`, and starts by applying `removeNonAlpha()` to it, then `toUpperCase()`, and so on, finishing by applying `setToList()`.

It may be a surprise that we can also make this work with `.reduce()` -- can you see why? The reasoning is similar to what we did, so we'll let it be *an exercise for the reader!*

```

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 can end this section by mentioning that in terms of testing and debugging, we can apply the same ideas as for debugging; only 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 now a common way of chaining operations when using objects and see if 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 give examples for both of them and they won't differ, other than the 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 seeing how to test the `pipeTwo()` function, because the setup will be similar for `pipeline()`. We need to create some spies and then check whether they were called the right number of times and whether they received the correct arguments each time. We will set the spies so they will provide a known answer to a call, so we can see if 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 tests is that one shows a pipeline applied to a single argument and the other shows application to several arguments.

Moving on to `pipeline()`, tests would be quite similar. We can, though, 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 just see a single test -- I just changed the order of the functions in the preceding test:

```
var fn1, fn2, fn3, fn4;
```

```

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 `City` object I created above -- I didn't want to mess with mocks, stubs, spies, and the like, but rather wanted to ensure that the code worked in 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 the tests is shown in the following figure:

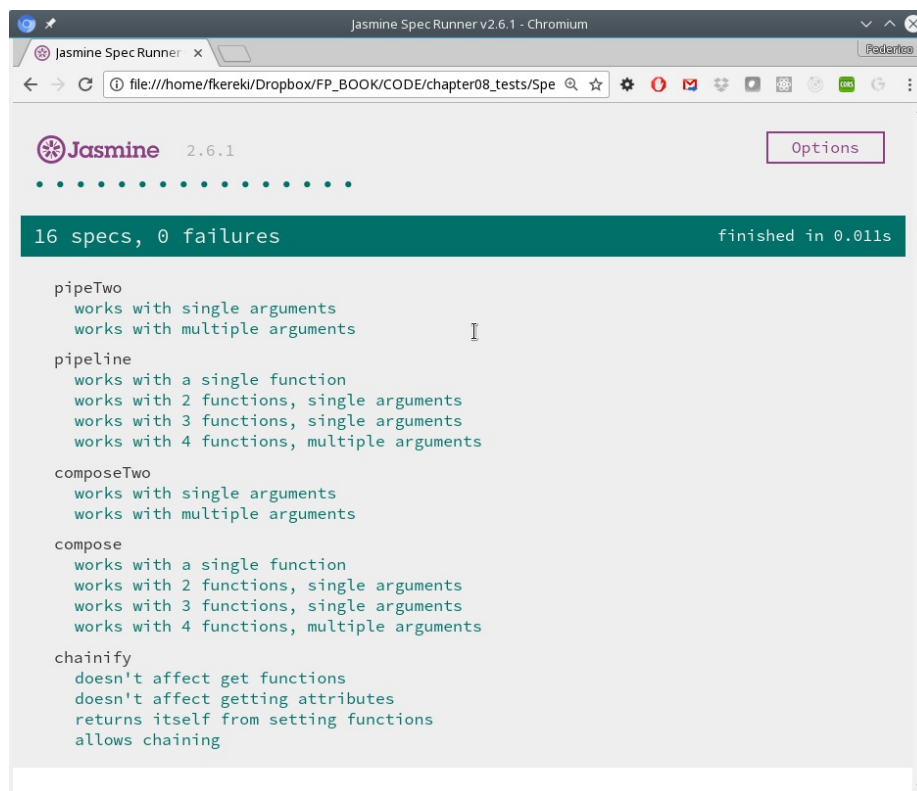


Figure 8.3. A successful run of testing for composed functions.

Questions

8.1. Headline capitalization. Let's define *headline-style capitalization* to require 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 function `headline(sentence)` 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 follows, 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 the book, *Going Further On*, we will see a different way of solving this, by using *Maybe* monads, that will greatly simplify 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 names vague and abstract, so you can focus on the structure and not on the actual functionality.) Can you transform this to Pointfree style?

```
| function getSomeResults(things) {  
|   return sort(group(filter(select(things))));  
| };
```

Summary

In this chapter, we have seen ways of creating new functions by joining several other functions in different ways, through pipelining (with a variant that we don't recommend, chaining) and composition.

In [Chapter 9](#), *Designing Functions - Recursion*, we will move on to function design, and study the usage of recursion, which classically is a basic tool in functional programming and allows for very clean algorithm designs.

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 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 JS 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. So, in this section we shall see several examples of recursive thinking, so you can adapt it for your functional coding.



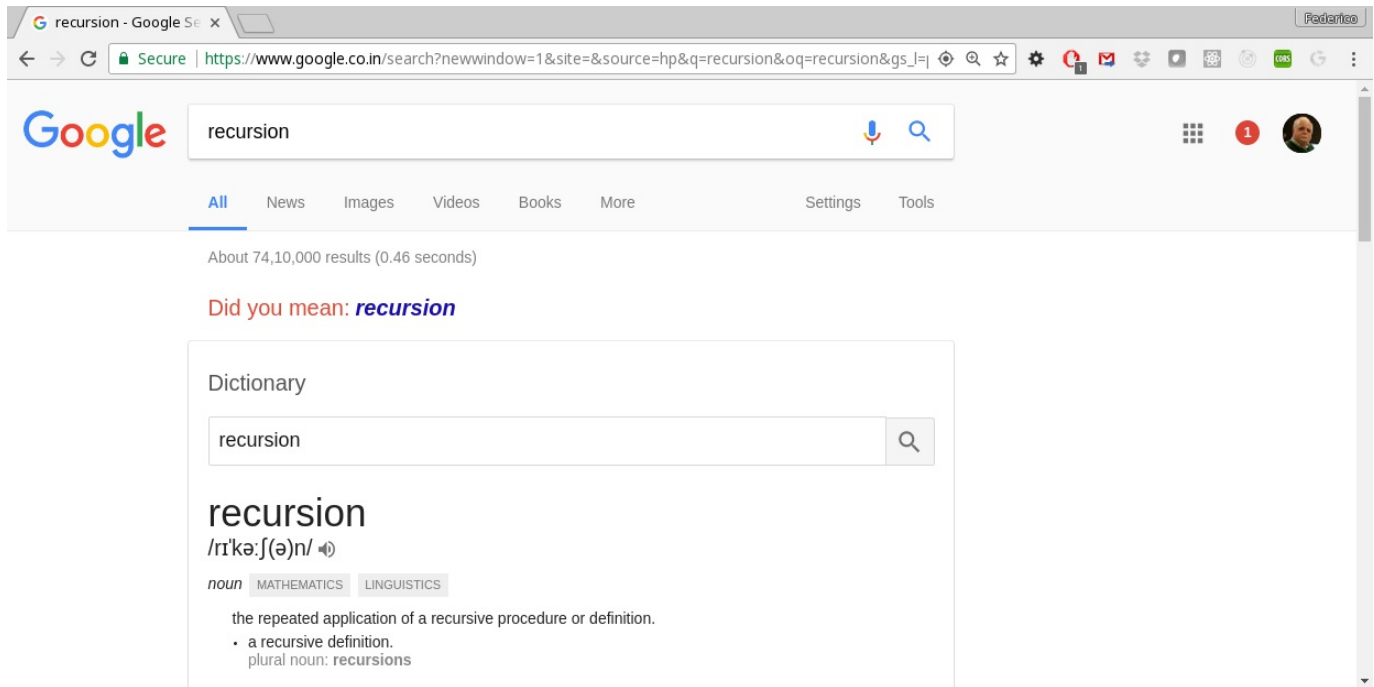
A typical, oft-quoted, and very old, computer joke!

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:

- Mathematical definitions, such as the Fibonacci numbers or the Factorial of a number
- Data structure related algorithms, with recursively defined structures, such as *lists* (a list is either empty or consist 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



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 now, so let's see how we can think in a recursive fashion and then solve several common problems by applying that technique.

Thinking recursively

The key to solving problems recursively is assuming that you already have a function that does whatever you need, and just call it normally. (Doesn't this sound weird? Actually, it is quite appropriate: to solve a problem with recursion, you must first solve the problem...) 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:

1. Assume you already have an appropriate function to solve your problem.
2. Then, 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 are your *base cases*, simple enough that they be solved directly, not requiring any more calls.

With these elements, 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 such solutions to solve the original problem. The only difference between *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 whenever you find yourself needing, again, the solution to a simpler case, you won't directly apply recursion, but rather use the stored result and avoid unnecessary repeated calculations

In this section, we shall see 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 needed to create such an algorithm.

Decrease and Conquer: searching

The most usual case of recursion involves just a single simpler 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 non-mathematical example.

To search for an element in an array, you would also use this *decrease and conquer* strategy. If the array is empty, obviously the searched 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:

```
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 see two further implementations of the same concept. You can shorten the search function a bit -- is it still clear?

```
const search2 = (arr, key) =>  
  arr.length === 0  
    ? false  
    : arr[0] === key || search2(arr.slice(1), key);
```

Sparseness can go even farther!

```
const search3 = (arr, key) =>  
  arr.length && (arr[0] === key || search3(arr.slice(1), key));
```

I'm not really suggesting you code the function in this way -- rather, consider it a warning against the tendency that some FP developers show 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 powers of numbers in an efficient way. If you want to calculate, say, 2 to the 13th power (2^{13}), you may do this with 12 multiplications. However, you can do much better by writing 2^{13} as:

$$= 2 \text{ times } 2^{12}$$

$$= 2 \text{ times } 4^6$$

$$= 2 \text{ times } 16^3$$

$$= 2 \text{ times } 16 \text{ times } 16^2$$

$$= 2 \text{ times } 16 \text{ times } 256^1$$

$$= 8192$$

This reduction in the total number of multiplications may not look very impressive, but, in terms of algorithms complexity, it allows bringing 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:

```
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 if 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 some 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()`, or `filter()`, also apply this technique; we'll look into this later on in this chapter.

Divide and conquer: The Tower of Hanoi

With this 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 XIX century. It seems that there is a temple in India, with three posts with 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 less than 10 disks. See figure 9.1:

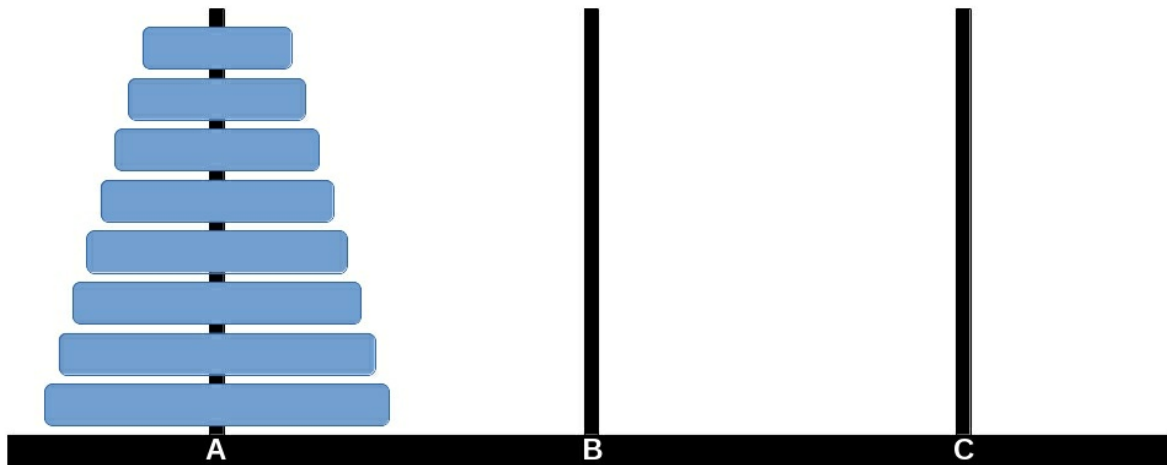


Figure 9.1- The classic Towers of Hanoi puzzle has a simple recursive solution.



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 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. So, now 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 a post to another, you could solve it easily by using this (still unwritten!) function, by:

- Moving all of the disks but one to the extra post
- Moving the larger disk to the destination post
- Using your function, again, to move all of the disks from the extra post (where you had earlier placed them) to the destination

But, what about our bases cases? We could decide that to move a single disk, you needn't use the function; you just go ahead and move it. Coded, it becomes:

```
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 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` equal one. You could also solve it in a different way, by letting disks go down to zero and simply not doing anything -- after all, moving zero disks from a post to another is achieved by doing nothing at all!

```
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 if 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, if 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 premises:

1. If your array has 0 or 1 elements, do nothing; it's already sorted (this is the base case).
2. Otherwise, pick some element of the array (called the "pivot") and split the rest of the array into two sub-arrays: the elements less than your pick, and the elements greater than your pick, or equal to it.
3. Recursively sort each sub-array.
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 sub-array is underlined. Splitting is shown with dotted arrows and is joined with full lines:

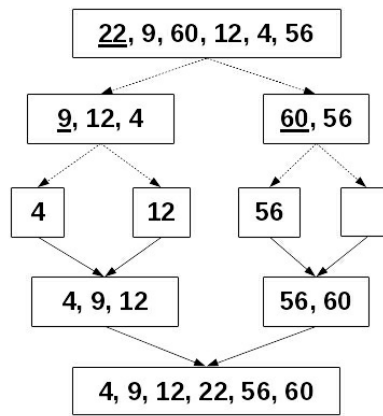


Figure 9.2. Quicksort sorts an array recursively, applying the divide and conquer strategy, to reduce the original problem to smaller ones.

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 having stored the previously found solutions... memoization, in other terms! 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 amount 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:

- Paying negative values is not possible, so in such cases, we should return 0
- 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 have pruned 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:

```
| 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 re-calculate 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!) But, 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:

```

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 already saw in [Chapter 5, *Programming Declaratively - A Better Style*](#), such as `map()`, `reduce()`, `find()`, and `filter()`, to see how we can make do with the only recursion.

We are not planning to exchange the basic JS 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 our efforts are more pedagogical than practical, OK?

Mapping and filtering

Mapping and filtering are quite similar, insofar both imply going through all the elements in an array and applying a callback to each to produce output. Let's work out first the mapping logic, which will have several points to solve, and then filtering will become almost trivially easy, requiring just small changes.

For mapping, according to the way we are using to develop recursive functions, we need a base case. Fortunately, that's easy: mapping an empty array just produces a new empty array. Mapping a non-empty 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 that idea, we can work out a simple initial version: let's call it `mapR()` just to remember 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:

```
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 than `.map()` ... but, shouldn't our callback function receive a couple more parameters, specifically the index at the array and the original array itself? Our implementation isn't quite ready yet.



Check out the definition for the callback function for `.map()` at: https://developer.mozilla.org/en/docs/Web/JavaScript/Reference/Global_Objects/Array/map

```
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 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:

```
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 looping is achieved when you only use recursion:

```
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:

```
[1, 2, , , 5].map(tenTimes)
// [10, 20, undefined * 2, 50]
```

Fortunately, fixing that is simple -- and be glad that all the experience gained here will help us write the other functions in this section!

```
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 * 2, 50]

```

Wow! This was more than we bargained for, but we saw several techniques: replacing iteration with recursion, how to accumulate a result across iterations, 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 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]

```

OK, we managed to implement two of our basic higher-order functions with pretty similar recursive functions. What about 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 wouldn'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:

```
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 on its own:

- 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: if you have no places left, where to look into, then you know you won't be successful in your search:

```
const findR = (arr, cb) => {
  if (arr.length === 0) {
    return undefined;
  }
  if (cb(arr[0])) {
    return arr[0];
  }
  return findR(arr.slice(1), cb);
};
```

```

    } else {
      return cb(arr[0]) ? arr[0] : findR(arr.slice(1), cb);
    }
  };

```

Equivalently:

```

const findR2 = (arr, cb) =>
  arr.length === 0
    ? undefined
    : cb(arr[0]) ? arr[0] : findR(arr.slice(1), cb);

```

We can quickly verify that it 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 a quick implementation.

- If we want to pipeline a single function, then that's the result of the pipeline
- Otherwise, 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:

```

const plus1 = x => x + 1;
const by10 = x => x * 10;

pipelineR(
  by10,
  plus1,
  plus1,
  plus1,
  by10,
  plus1,
  by10,
  by10,
  plus1,
  plus1,
  plus1

```

```
| ) (2);  
| // 23103
```

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:

- 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 variants, you can also apply a 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 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 has been found.

There are many problems for which this logic applies:

- Finding a way out of mazes -- pick any path, mark it as *already followed* and try to find some way out of the maze that won't re-use that path: if you succeed, you are done, and if 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 you rather opt for the best-estimated position

Let's apply these techniques for two problems: solving the 8 *Queens* puzzle and traversing a complete file directory.

The Eight Queens puzzle

The *Eight Queens* puzzle was invented in the XIX Century and requires placing eight chess queens on a standard chessboard. The special condition is that no queen may attack another -- implying that no pair of queens may share a row, column, or diagonal line. The puzzle may ask for any solution or, as we shall do it, for the total number of distinct solutions.



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 on the center, all of the board is threatened, and if you place a queen on a side, only two squares are unthreatened--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 do the recursive search for solutions:

```
const SIZE = 8;
let places = Array(SIZE);
let solutions = 0;

finder();
console.log(`Solutions found: ${solutions}`);
```

When we want to place a queen in a given row within a certain column, we must check if any of the previously placed queens was already placed on the same row, or in a diagonal with respect to it. Let's write a `checkPlace(column, row)` function to verify if a queen can be safely placed in the given square. The most straightforward way is by using `.every()`, as 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 diagonals. 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: 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 recursively now check the following one:

```
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 that 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).

```
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, 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 here, for example, how to traverse a complete file system directory, listing all of its contents. Where's the recursion? The answer follows 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, when we encounter a directory, we proceed to also list its contents and, if those include more directories, we also list them, and so on. We'll be using the same Node.js 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:

```

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 may 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:

```

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 as follows. 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:

```

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 the would-be reader that we are trying to make an array out of something that looks like one, but really isn't.

Recursion techniques

While recursion is a very good technique, it may face some problems due to details in the actual implementations. Each function call, recursive or not, requires an entry in the internal JS stack. When you are working with recursion, each recursive call itself counts as another call and you might find some situations in which your code will crash and throw an error, due to having run out of memory, just because of multiple calls. On the other hand, with most current JS 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 imaginably 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 because, even if you don't get to actually apply them, they represent valid FP ideas for which you may find place in yet other problems.

Tail call optimization

When is a recursive call not a recursive call? Put in this way, the question may make little sense, but there's a common optimization --for other languages, alas, but not JS!-- which 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 was no need to further save any of the elements that had been pushed into the stack on 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, may be considered an ultimate irony!

These calls are known as *tail calls* (for obvious reasons) and imply higher efficiency, not only because of the saved stack space, but also because a jump is quite faster than any alternative. If the browser implements this enhancement, it is doing a *Tail Call Optimization*, or TCO for short. However, a glance at compatibility tables at <http://kangax.github.io/compat-table/es6/> shows that now (mid-2017), the only browser that provides TCO is Safari.

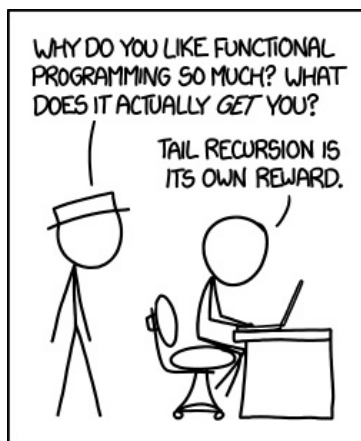


Figure 9.3. To understand this joke, you must previously understand it!
(Note: this XKCD comic is available online at <https://xkcd.com/1270/>.)

There's a simple (though non-standard) test that lets you to verify if your browser provides TCO. (I found this snippet of code at several places on the web, but I'm sorry to say I cannot attest to the original author. However, I believe it is Csaba Hellinger, from Hungary.) Calling `detectTCO()` lets you know if 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 JS 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 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
- Should get longer without TCO, since a complete new stack entry would be created, without doing away with the original one

I'm using Chrome at 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 if 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 may be fairly sure that your JS engine is doing TCO!

```
function justLoop(n) {
  n && justLoop(n - 1); // until n is zero
}
```

Let's finish this section with a very short quiz, to be sure 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 would see it more clearly if you wrote the function in this equivalent way:

```
function fact2(n) {  
  if (n === 0) {  
    return 1;  
  } else {  
    const aux = fact2(n - 1);  
    return n * aux;  
  }  
}
```

So... the takeaways from this section should be two: TCO isn't usually offered by browsers, and even if it were, you may not 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 for 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 at that point.)

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) which will be called at return time.

Let's see a trivial example. Suppose you have a function that returns the time of the day, and you want to show that on the console. The usual way to do it could be as follows:

```
function getTime() {  
    return new Date().toString();  
}  
  
console.log(getTime()); // "21:00:24 GMT+0530 (IST)"
```

If you were doing CPS (**Continuation Passing Style**), 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().toString());  
}  
  
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:

```
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 to `factC()` to make it clear we are working with continuations:

```
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 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? Let's then go for a more complex case, with the Fibonacci function, which has *two* recursive calls in it:

```
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, then, call `fibC()` with `n-1`, and when *that* call returns, then sum the results of both calls and pass that result to the original continuation.

Let's just see one more example, one that involves a loop with an undefined number of recursive calls, and 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 designing 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 result:

```
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 with the previous cases is that the function doesn't return anything, so we won't be passing any arguments to the continuation. Also, it's 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 with the potential high number of pending calls hasn't been solved; let's look for a solution for that now.

Trampolines and thunks

For the last key 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 your 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 sonorous 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 for a form of *lazy evaluation*. If you have a thunk, 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* gets 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 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.

```
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.

```
const sumAll = n => (n == 0 ? 0 : n + sumAll(n - 1));

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 it will become tail recursive.

```
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.

```
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()` a 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 wouldn't probably 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:

```
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 (meant to be invoked and executed) from any other kind of result (meant to be returned).

So, if you happen to have a really complex algorithm, for which a recursive solution is best, but that won't run because of stack limits, you can fix it in a reasonable way by:

1. Changing all recursive calls to tail recursion, by using continuations.
2. Replacing all return statements so 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 non-working solution to a problem!

Recursion elimination

There's yet one other possibility you might want to explore, but that falls beyond the realm of FP, and rather 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 rather depends on a stack. There are ways to systematically transform recursive algorithms into iterative ones, so if you run out of all options (meaning: not even continuations or thunks help you) then you'd have a final opportunity, by replacing all recursion with iteration. We won't be getting into it --as I said, this elimination has little to do with FP-- but it's important to know that the tool exists and you might be able to use it.

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 as 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 may opt to take 1 or 2 steps. In how many different ways can you climb up that ladder? As an example, you may climb a four steps 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 again 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 solved above, there is only one solution which 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 at the end of the sorted rest
- **Insertion sort:** take the first element of the array; sort the rest; 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; 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?

Summary

In this chapter, we have seen 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 is recursion and how to think recursively in order to solve problems, then moved on to see 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 do any side effects, by ensuring the immutability of arguments and data structures.

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 for impurity. Now, after several chapters dealing with many aspects and tools of FP, let's get to 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 modifying the original data, but produces new objects instead) then we'll have an enforceable solution. In this chapter, we will see two distinct approaches to working with such immutable objects and data structures:

- Basic JS ways, such as freezing objects, plus cloning to create new ones instead of modifying existing objects
- Persistent data structures, with methods that allow updating 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, 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.

The straightforward JS way

One of the biggest causes of side-effects was the possibility of a function modifying either global objects or its arguments themselves. All non-primitive objects are passed as references, so when/if 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 JS techniques to disallow those side-effects.

Mutator functions

A common source of unexpected problems comes from the fact that several JS methods actually modify the underlying object. In this case, by merely using them, you will be causing a side-effect, which you may even not recognize. Arrays are the basic source of problems and the list of troublesome methods is not short. (See https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Array#Mutator_methods for more on each method.)

- `.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 the same way, 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

For some of these operations, you might generate a copy of the array and then work with that. In the *Argument mutation* section of [Chapter 4, Behaving Properly - Pure Functions](#), we did just that with the spread operator; we could have used `.slice()` as well:

```
const maxStrings2 = a => [...a].sort().pop();
const maxStrings3 = a => a.slice().sort().pop();

let countries = ["Argentina", "Uruguay", "Brasil", "Paraguay"];
console.log(maxStrings3(countries)); // "Uruguay"
console.log(countries); // ["Argentina", "Uruguay", "Brasil", "Paraguay"] - unchanged
```

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 described later.

Constants

If the mutations do not happen because of using some JS methods, then we might want to attempt using `const` definitions, but that just won't work. In JS, a `const` definition means only that the *reference* to the object or array cannot change (so you cannot assign a different object to it) but you can still modify the properties of the object itself.

```
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}
```

So, if you decide to use `const` everywhere, you will be safe only 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.

What can work is using *freezing* to provide un-modifiable structures and *cloning* to produce modified new ones. These are probably not the best way to go about forbidding objects to be changed but can be used as a makeshift solution. Let's go with both of them in some detail.

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.

```
const myObj = { d: 22, m: 9 };
Object.freeze(myObj);

myObj.d = 12; // won't have effect...
console.log(myObj);
// Object {d: 22, m: 9}
```



Don't confuse freezing with sealing: `Object.seal()`, applied to an object, prohibits adding or deleting properties to it, so 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 similarly 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:

```
myObj3.d = 8888; // wont' work
myObj3.o.f.a = 9999; // oops, does work!!
console.log(myObj3);
// {d:22, m:9, o:{c:"MVD", i:"UY", f:{ a:9999 }}}
```

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 that by applying recursion. Mainly, the idea is to first freeze the object itself and then recursively freeze each of its properties. We must take care 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 the returned object will always be the original one. If we wanted to work in a purer fashion, we should first make a copy of the original object (we'll be seeing how to do this in the next section) and then freeze that.

There remains a small possible problem, but with a very bad result: what would happen if an object included, somewhere down there, a reference to itself? We can avoid that if we skip freezing already frozen objects: backward circular references would then be ignored since objects they refer to would be already 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 will not work, because the changes will be silently ignored. But, then, how can we return a result from a function, if it involves a received object? This can be solved in many ways, and 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.

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. Then, in order to fulfill the reducer requirements, we will have to be able to clone the original state, plus mutate it accordingly to the received action, and that resulting object will then 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, as 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 it by hand, but that's not something you'd really want to consider when working with large, complex objects.

```
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}}
};
```

Now, going for more automatic solutions, there are a couple of straightforward ways of copying arrays or objects in JS, but they have the same *shallowness* problem.

```
let newObject1 = Object.assign({}, myObj);
let newObject2 = {...myObj};

let myArray = [1, 2, 3, 4];
let newArray1 = myArray.slice();
let newArray2 = [...myArray];
```

If an object or array includes objects (which may themselves include objects, and so on) we get the same problem as with 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!!
```

There is a simple solution, based on JSON. If we `stringify()` the original object and then `parse()` the result, we'll get a new object but its totally separate from the old one.

```
const jsonCopy = obj => JSON.parse(JSON.stringify(obj));
```

This works with both arrays and objects, but there's a problem, anyway. If any of the properties of the object have a constructor, it won't get invoked: the result will always be composed of plain JS 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
```

We could do a recursive solution, as 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;
};
```

This solves the problem we found with dates or, in fact, with any object! If we run the code above, but using `deepCopy()` instead of `jsonCopy()`, we'll get object object as

output, as it should be. If we check 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!
```

Now that we know how to copy an object, we can work in this way:

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, to use in your 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

Doing all the work we listed at the end of the previous section, every time you want to update a field, would probably become troublesome, and prone to errors. Let's add a pair of functions to be able to get values from a frozen object, but unfreezing them so they become usable by you, and to allow modifying any property of the object, creating a new copy of it, so the original won't be actually modified.

Getting a property

Back in the *Getting a property from an object* section in [Chapter 6, Producing Functions - Higher-Order Functions](#), we wrote a simple `getField()` function that could handle getting a single attribute out of an object.

```
| const getField = 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. Rather, let's have a function that will receive a *path* -an array of field names- and will return the corresponding part of the object, or will be undefined if the path doesn't exist. Using recursion is quite appropriate and simplifies coding!

```
| 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;  
|   }  
| };
```

Once an object has been frozen, you cannot *defrost* it, so we must resort to making a new copy of it; `deepCopy()` is quite appropriate for that. 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
```

Setting a property by path

Now that we wrote this, we can code a similar `setByPath()` function that will take a path, a value, and an object, and update an object.

```
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;
  }
};
```

We are using recursion here 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](https://github.com/rtfeldman/seamless-immutable) library, which works exactly in this fashion. The seamless part of the name alludes to the fact that you still work with normal objects, albeit frozen!, so you can use `.map()`, `.reduce()`, and so on. Read more about it at <https://github.com/rtfeldman/seamless-immutable>.

We can then 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);
};
```

We can check out how it works: let's 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()`



If you want to see another way of using setters and getters to accomplish functional access and updates to objects, check out lenses, provided by libraries such as Ramda. Lenses can be seen as a functional way of not only getting and setting variables, but also running functions over them, in a composable way: a something that lets you focus on a specific part of a data structure, access it, and possibly also change it or apply functions to it. Check out more starting at <http://ramdajs.com/docs/#lens>.

Persistent data structures

If every time you want to change something in a data structure, 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 way, with persistent data structures, which, if handled correctly, let you apply changes while creating new structures, in an efficient way.

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 it? We can assume 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. See figure 10.1:

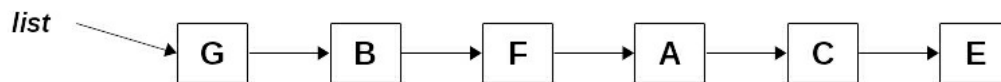


Figure 10.1. The initial list. (Can you tell what is missing in this list, and where?)

If you wanted to add D between B and F (this is something musicians will understand: we have here the *Circle of Thirds*, but D is missing) the simplest solution would be to just add a new node and change an existing one, to get the following result. See figure 10.2:

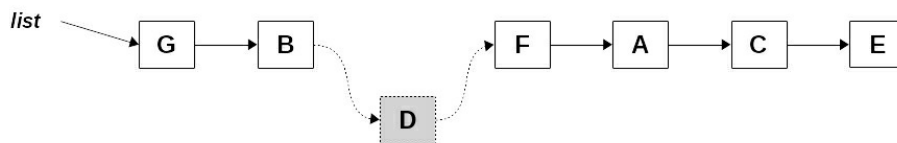


Figure 10.2. The list has now a new element: we had to modify an existing one to do the addition.

However, working in this way is obviously non-functional and it's clear we are modifying data. There is a different way of working, by creating a persistent data structure, in which all 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. See figure

10.3:

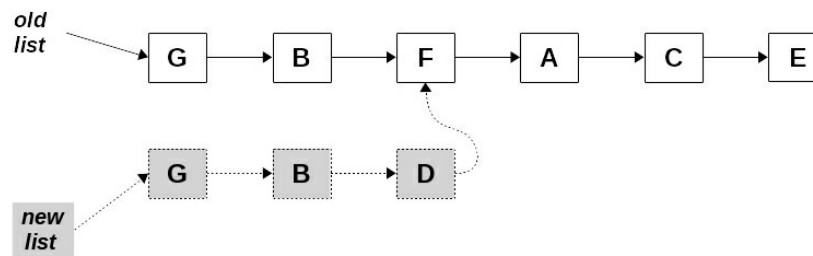


Figure 10.3. The dotted elements show the newly returned list: some elements had to be duplicated to avoid modifying the original structure. Old list refers to the original structure and new list to the result of the insertion.

Of course, we will also deal with updates or deletions. Starting again with the list as in Figure 10.4, 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.

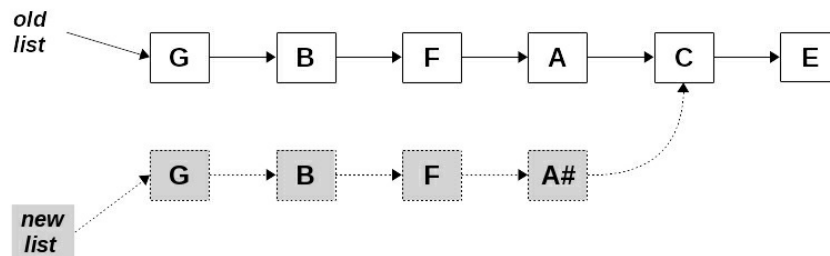


Figure 10.4. Our list, with a changed element.

Removing an element would also be similar. Let's do away with the third element, F, in the original list. See figure 10.5:

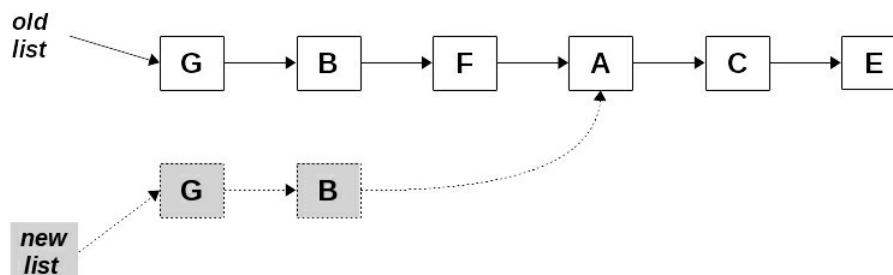


Figure 10.5. The original list, after removing the 3rd element in a persistent way.

Working with lists or other structures can always be solved to provide data persistence. But, let's now focus on what will probably be the most important kind of

work for us: dealing with simple JS objects. After all, all data structures are JS objects, so if we can work with any 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 that 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 an object as follows:

```
myObj = {  
  a: ...,  
  b: ...,  
  c: ...,  
  d: {  
    e: ...,  
    f: ...,  
    g: {  
      h: ...,  
      i: ...  
    }  
  }  
};
```

If you wanted to modify `myObj.d.f`, and wanted to do it in a persistent way, you would create a new object, which would have several attributes in common with the previous object, but would define new ones for the modified ones. See figure 10.6:

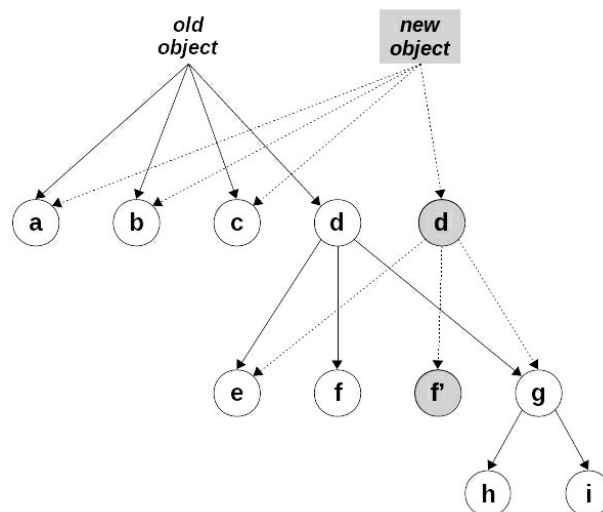


Figure 10.6. A persistent way of editing an object, by creating a new one with some shared attributes, and some new ones.

If you had wanted to do this by hand, you would have had 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 already saw similar code earlier in this chapter, when we decided to work on a cloning function, but let's now 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 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.

We can give this logic a try. Let's 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 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 re-using the value from `myObj`.

Further updating the array in the second object lets us test also 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, and you can see that the array is a new one, but two of the elements (the ones that weren't updated) are still the same objects as 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 require the possibility of eliminating some attribute. Libraries usually provide many more functions, but let's at least work on the deletion of an attribute, to see other important structures change in 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 is similar to that of `setIn()`. The difference is that we do not always copy all attributes from the original object to the new one: we only do that while we haven't yet arrived at the end of the array of path properties. Continuing 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, doing changes, additions, and deletions, in an efficient way that won't create new objects needlessly.



*Probably the best-known library for working with immutable objects is the appropriately named `immutable.js`, 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 basically 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 slow-downs might be observed for other operations, such as mapping, reducing, and so on.

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 the persistence library provides the needed interface so you'll be still able to use it as an array, not noticing the internal difference.

When using this kind of libraries, 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 even have to go so far as changing the way you implement immutability or even work out some way of changing your basic data structures to avoid the time loss, or at least minimize it.

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 for getting operations in order to provide for 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 as we earlier described. Implement an `insertAfter(list, newKey, oldKey)` function that will create a new list, but adding a new node with key `newKey` just after the node with key `oldKey`. You may assume the nodes of 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"))))));
```

Summary

In this chapter, we have seen two different approaches (actually used by commonly available immutability libraries) to avoiding side-effects by working with immutable objects and data structures: one was based on using JavaScript's *object freezing* plus some special logic for cloning and the other applied the concept of persistent data structures, with methods that allowed all kinds of updates without either changing the original or requiring full cloning.

In [Chapter 11](#), *Implementing Design Patterns - The Functional Way*, we will focus on a question 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 they differ from the usual OOP practices.

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 formulae and solutions, which are often used in imperative coding. Since design patterns are well known, and programmers will be 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. 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 alternative we have in FP, if we need one
- A discussion about FP design patterns, not related to the OOP ones

What are Design Patterns?

One of the most relevant books in software engineering was *design patterns: Elements of Reusable Object-Oriented Software*, 1994, written by the **GoF (Gang of Four)**: 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 usually seen, common 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*, which can be used by developers working with different kinds of systems, programming languages, and environments.

The 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 FP languages already provide standard solutions to the corresponding OOP 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:

1. A simple, short, *name* that is used to describe the problem, its solutions, and their consequences. The name is useful for talking with colleagues, explaining a design decision, or describing a specific implementation.
2. The *context* to which the pattern applies: this implies specific situations that require a solution, with possibly some extra conditions that must be met.
3. A *solution* that lists the elements (classes, objects, functions, relationships, and so on) that you'll need in order to solve the given situation.

4. 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 much 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 the GoF patterns; we'll just focus on those for which applying FP is more interesting, bringing out more differences with 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:

- **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:** They deal with ways to create objects in a manner that is suitable for the current problem, possibly guiding the selection between several alternative objects, so the program can work in different ways depending on parameters that may be known at compilation time or at runtime.
- **Structural design patterns:** They 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 towards being able to dynamically change the way objects are composed at runtime.
- **Concurrency patterns:** They are related to dealing with multithreaded programming. Although FP is generally quite appropriate for that (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:** They 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 the 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 60's when Structured Design by Larry Constantine was 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.

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 language, in a simple, trivial way, then you may not need a design pattern at all.

In any case, it's interesting, for OOP developers, to really understand why FP helps solve some problems without need of 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 to 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 SFP, *Sorta Functional Programming*, from the initial part of the first chapter?), 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 one's 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, there are some design patterns that don't get an FP solution. For example, there's no equivalent for a Singleton, which implies the foreign concept of a globally accessed object. Also, 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?

Façade and Adapter

Out of these two patterns, let's start with the *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 in an easier way (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 need. 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 ES8's modules, you might write something as the following, working with an imagined hard 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 things to convert params
  // into the way that the hard library requires
};

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 a `GET` or `POST`, instead of having to go through all the complications of the provided hard Ajax library, you can use the new façade that provides a simpler way of working. Developers would just `import {getUrl, postUrl}` from `"simpleAjax"` and could then work in a more reasonable way.

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](#), by means of a *revealing module* pattern:

```

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* should be now obvious. With the preceding code, because of the scope rules in JS, the only visible attributes of `simpleAjax` will be `simpleAjax.getUrl` and `simpleAjax.postUrl`; using an IIFE lets us implement the module (and thus the façade) in a safe way, 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.

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). Suppose we want to show some element on screen, and for debugging purposes, we want to show a thin red border around the object. How can you do it?

If you were programming using OO, you would probably have to create a new subclass, with the extended functionality. For this particular example, you may just provide some attribute with the name of some CSS class that would provide the required style, but let's keep our focus to the OO; 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 on 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* section of [Chapter 6, Producing Functions - Higher-Order Functions](#). For example, in that section, we saw how to wrap functions in order to produce new versions that could log their input and output, provide timing information, or even memorize calls to avoid future delays. In this occasion, for a 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 the code for every component in the single, same 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 for *react online sandbox* if you want some other options. Results aren't much to talk about, but we are interested in a design pattern right now, and not in UI design; refer to Figure 11.1:

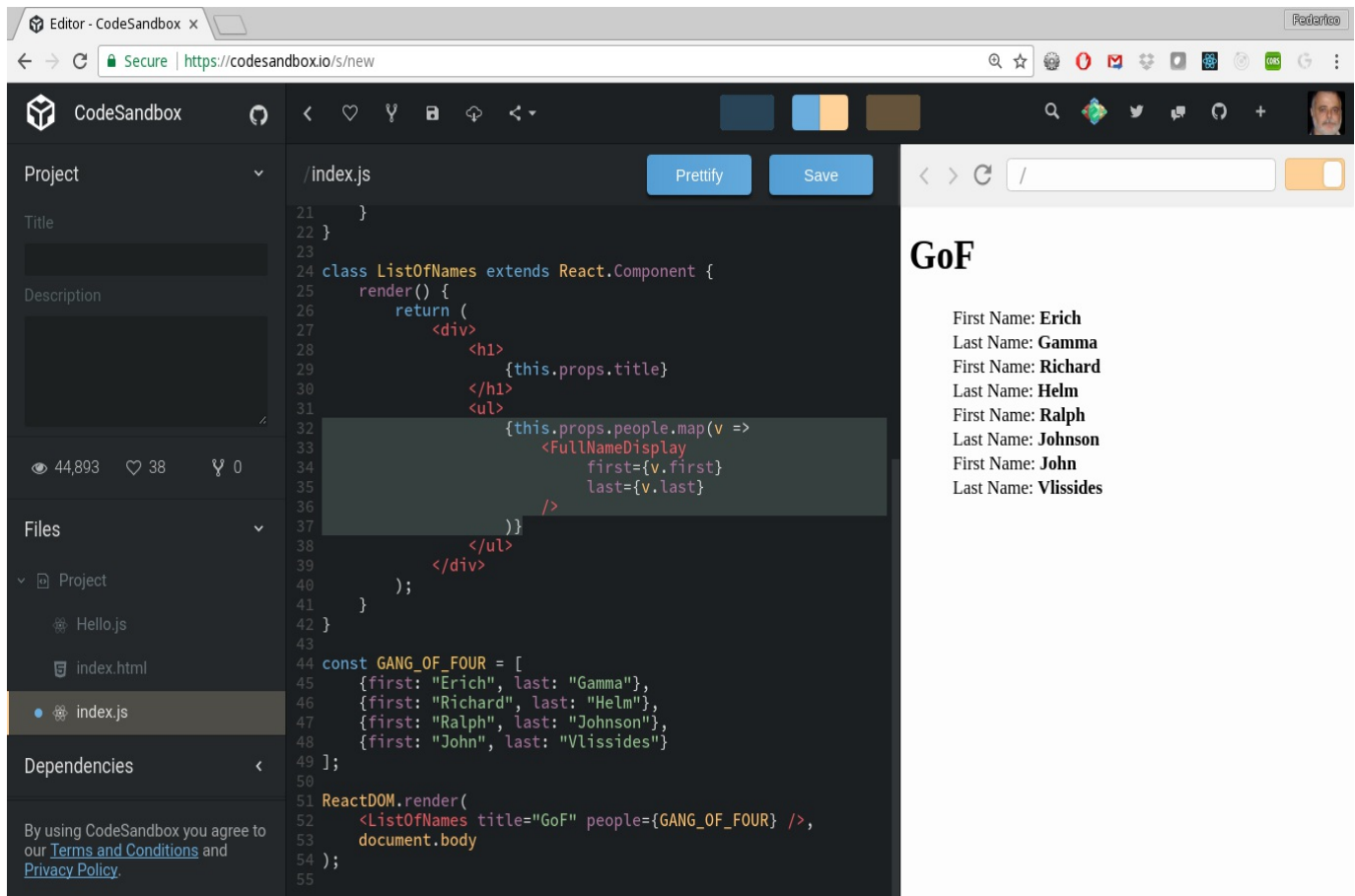


Figure 11.1: The original version of our components shows a (not much to speak about) list of names

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, if we write a function that will take a component as a parameter, and return new JSX, which will be a wrapped object. In our case, we'd like to wrap the original component within a `<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 in 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:

```
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:

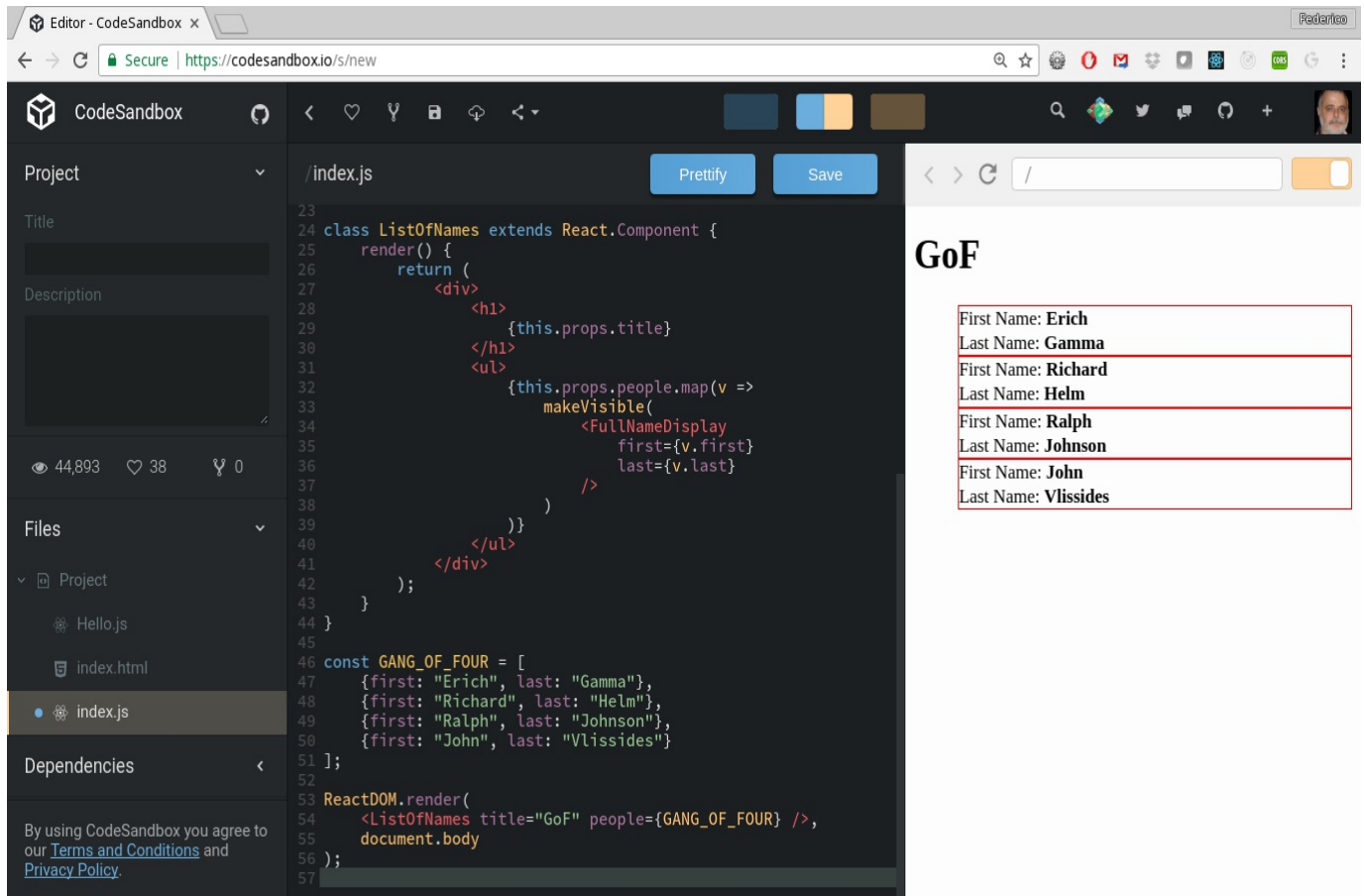


Figure 11.2: The decorated ListOfNames component is still nothing much to look at, but now it shows an added border

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 solution to provide a *higher-order component* (as called in React parlance) wrapped in an extra `<div>` to provide some visual 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, connected to the store, component class for usage in your forms; refer to <https://github.com/reactjs/react-redux> for more details.

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 *does its thing*. For example, a GPS application might want to find a route between two places, but 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 OO 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 JS, implementing strategies is trivial, instead of using a variable such as `byMeans` to switch, you can just pass a function around, which will implement the desired path logic:

```

function findRoute(routeAlgorithm, fromPoint, toPoint) {
    return routeAlgorithm(fromPoint, toPoint);
}

```

You would still have to implement all the desired strategies (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 pattern *Command* also benefits from the ability of being 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 replacement for plain functions working as callbacks.

Other patterns

Let's end this section by glossing over 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 an 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 any 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, 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.

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 the book.

What are candidates for a possible list of patterns? Let's attempt preparing 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, as `filter()`, `map()`, and `reduce()`, as we saw 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, sequentially applied, applying 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 JS (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 can do instead of doing a calculation so that whenever the actual value is needed, it will be calculated at that time, but not earlier.

You may also want to look at JS generators, which is another way of delaying evaluation, though it's not particularly related to FP at all.



Read more about generators 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, and those 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 in a more functional way. 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 dozens 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.

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 draft and take a look at what makes the next code tick.

Some questions: Do you see the need for the `savedMethod` variable? Why do we use `function()` when assigning `new descriptor.value`, instead of an arrow function? Can you understand why `.bind()` is used? What is `descriptor`?

```
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
```

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 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 a property:

```
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`?

Summary

In this chapter, we have done a bridge from the object-oriented way of thinking, and the usual patterns that we use when coding that way, to the functional programming style, by showing how we can solve the same basic problems, but rather more easily than with classes and objects.

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 look abstruse or remote.

Building Better Containers - Functional Data Types

In [Chapter 12](#), *Implementing Design Patterns - The Functional Way*, we have gone over many ways of using functions to achieve different results, and in this chapter, we will go more deeply into data types from a functional point of view. We'll be considering ways of actually implementing our own data types, with several features to help composing operations or ensuring purity, so your FP coding will become actually simpler and shorter. We'll be touching on several themes:

- **Data types** from a functional point of view, because even though JavaScript is not a typed language, a better understanding of types and functions are needed
- **Containers**, including *functors* and the mystifying *monads*, to better 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

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. Let's now get into the theme of seeing how we can specify types, for that we will have at least some advantages:

- Even if you don't have runtime data type checking, there are several tools, such as Facebook's *flow* static type checker or Microsoft's *TypeScript* language, which 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, to let future developers understand what type of arguments they have to pass to the function, and what type it will return. As an example of this, 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 directly want to know 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, so we shall now begin considering how we can define, most importantly, the types of functions and their parameters and result.

Signatures for functions

The specification of a function's arguments and result, is given by a *signature*. Type signatures are based on a *type system* called Hindley-Milner, which influenced several (preferably 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 that kind of figuring out, so the developer need not specify *all* types. Instead of going for a dry, formal, explanation about the rules for writing correct signatures, let's work by examples. We need to only know that:

1. We will be writing the type declaration as a comment.
2. The function name is written first, and then `::` that can be read as *is of type* or *has type*.
3. Optional constraints may follow, with a double (*fat*) arrow `⇒` (or `=>` in basic ASCII fashion, if you cannot key in the arrow) afterwards.
4. The input type of the function follows, with a `→` (or `->` depending on your keyboard).
5. The result type of the function comes last.



Note that instead of this vanilla JS style, Flow and TypeScript have their own syntax for specifying type signatures.

Now we can begin with some examples:

```
// firstToUpper :: String → String
const firstToUpper = s => s[0].toUpperCase() + s.substr(1).toLowerCase();

// Math.random :: () → Number
```

These are simple cases -- and mind the signatures; we are not interested in the actual functions here. 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 is so) 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 that it doesn't receive arguments.

We saw functions with zero or one parameter: what about functions with more than one? There are two answers to this. If we are working in strict functional style, we would always be doing currying (as we saw in [Chapter 7, Transforming Functions - Currying and Partial Application](#)), so all functions would be unary. The other solution is enclosing a list of argument types in parentheses. We can see the following both ways:

```
| // 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;
```

The first signature can also be read as:

```
| // sum3C :: Number → (Number → (Number → (Number)))
```

This is correct when you remember the idea of currying. After you provide the first argument to the function, you are left with a new function, that 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, but of any type. Also, the mapping function can produce any type of result. For these cases, we can specify *generic types*, identified by lower case 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 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's described previously. Also notice 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 in a similar fashion the following:

```
| // filter :: [a] → (a → Boolean) → [a]
```

```
| const filter = curry((arr, fn) => arr.filter(fn));
```

And, the big one: how'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 $((b, a) \rightarrow b)$:

```
| // 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
```

Other type options

What else are we missing? Let's see some other options that you might use. *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. We can then 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 later 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
```

Afterwards, we could specify that a comparison function could 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 quite precise, because you actually can compare any types, even if it doesn't make much sense. However, bear with me for the sake of the example! And 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 last definition would allow writing 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:

```
| // 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 requiring much more typing, would have been writing all the possibilities with a union:

```
| // compareFunction ::  
| //   ((Number, Number) | (String, String) | (Boolean, Boolean)) → Number
```

So far, we have been using the standard type definitions. But, 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 an added `?` to represent an optional type:

```
| // unary :: ((b, ...) → a) → (b → a)  
| const unary = fn => (...args) => fn(args[0]);
```

The `unary()` higher-order function that we defined in the same chapter cited previously, took as a parameter any function, and returned a unary function as its result: we can show that the original function could receive any number of arguments, but the result used only the first:

```
| // parseInt :: (String, Number?) -> Number
```

The standard `parseInt()` function provides an example of optional arguments: though it's highly recommended not to 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 often be adding signatures to methods and functions. This will not only be so you can get accustomed to them, but because when we start delving into more complex containers, it will help to understand what we are dealing with: some cases can be hard to understand!

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 of being 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, as in [Chapter 6](#), *Producing Functions - Higher-Order Functions*), is available only 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 apply that only to basic JS types) or we can wrap types in some type of container, which will allow mapping and chaining.

Let's start first by extending current types, and then move on to using wrappers, which will lead us into deep functional territory, with entities such as functors and monads.

Extending current data types

If we want to add mapping to basic JS data types, let's 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: you 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 the book, we are sticking to plain JS, but you should look into libraries such as `LoDash`, `Underscore`, or `Ramda`, which already provide functionalities such as we are developing here.

A key point, 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()` implementations (you could observe that the resulting array may have different element types as the original one, but it still is an array).

What could we do with a boolean? First, let's accept that booleans are not containers, so they do not 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 elements. However, accepting that difference, we can extend the `Boolean.prototype` (though, as I've already mentioned, that's not usually recommended) by adding a new `.map()` method to it, and make sure that whatever the mapping function returns is turned into a new boolean value. And, for the latter, the solution will be similar:

```
// 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:

```

// 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 first 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)`:

```

// Function.map :: (a → b) → (b → c) → (a → c)
Function.prototype.map = function(fn) {
  return (...args) => fn(this(...args));
};

```

Verifying that this works, is simple:

```

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 as to what we can do with basic JS 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 some container; let's do this.

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, which we could apply to any data type. Since not all things in JS 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.

Wrapping a value: a basic container

Let's pause a bit, and consider what we do need from this wrapper. There are two basic requirements:

- We must have a `.map()` method
- We need a simple way to wrap a value

Let's create a basic container to get started with -- but we'll require some changes:

```
const VALUE = Symbol("Value");

class Container {
  constructor(x) {
    this[VALUE] = x;
  }

  map(fn) {
    return fn(this[VALUE]);
  }
}
```

Some basic considerations:

- We want to be able to store some value in a container, so the constructor takes care of that
- Using a `Symbol` helps to *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

Our basic barebones container is ready, but we can, however, add some other methods for convenience:

- In order to get the value of a container, we could do `.map(x => x)`, but that won't work with more complex containers, so let's add a `.valueOf()` method to get the contained value
- Being able to list a container can certainly help with debugging: a `.toString()` method will come in handy
- Because we need not write `new Container()` all the time, we can add a static `.of()` method to do the equivalent 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 we do not 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 `.valueOf()` is sometimes too handy, so it won't be that restrictive.

Our Container becomes 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 to what we could do with a variable! Let's enhance this a bit.

Enhancing our container: functors

We wanted 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, with which we can keep on working.



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()`. But, 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. The `.of()` method will require a small change:

```
class Functor extends Container {  
  static of(x) {  
    return new Functor(x);  
  }  
  
  map(fn) {  
    return Functor.of(fn(this[VALUE]));  
  }  
}
```

With these properties, we have just defined what's called a *Functor* 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 just some kind of container that allows applying `.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 types) as it happens with arrays. Second, there must exist a function, whose mapping produces the same contained value -- and $x \Rightarrow x$ does this job. Finally, applying two consecutive mappings must produce the same result as applying their composition: `container.map(f).map(g)` must be the same as

```
container.map(compose(g,f)).
```

Let's pause a moment to consider the signature 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 some 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 above.

As is, functors are not allowed or expected to produce side effects, throw exceptions, or any other behavior outside from producing a containered result. Their main usage is to provide a way to manipulate a value, apply operations to it, compose results, and so on, without changing the original -- in this last sense we are once again coming back to immutability.



You could also compare functors to promises, at least in one aspect. In functor, instead of acting on its value directly, you have to apply a function with `.map()` -- and 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 having to deal with exceptions, undefined or null values, and so on. So, let's start seeing more examples of functors, and after a while, we'll be entering into the realms of monads, for even more sophisticated kinds of processing. So, let's now 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, or a dataset could be empty, or an optional attribute might be missing from an object, and so on. Dealing with this kind of situation, in normal imperative fashion, requires adding `if` statements or ternary operators everywhere, to catch the possible missing value, avoiding a certain runtime error. We can do a bit better by implementing a `Maybe` functor, to represent a value that may be (or may *not* be) present! We will use two classes, `Just` (as in *just some value*) and `Nothing`, one for each functor:

```
class Nothing extends Functor {
  isNothing() {
    return true;
  }

  toString() {
    return "Nothing()";
  }

  map(fn) {
    return this;
  }
}

class Just extends Functor {
  isNothing() {
    return false;
  }

  map(fn) {
    return Maybe.of(fn(this[VALUE]));
  }
}

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:

```
const plus1 = x => x + 1;

Maybe.of(2209).map(plus1).map(plus1).toString(); // "Just(2211)"
Maybe.of(null).map(plus1).map(plus1).toString(); // "Nothing()"
```

We just applied `plus1()` several times to a `Maybe.of(null)` value, and there was no error whatsoever. 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 basically including an abstracted check, which won't let an error happen. Let's give a more realistic example of its usage.



Later in the chapter, we'll see that Maybe can actually be a Monad instead of a Functor, and we'll also examine more examples of monads.

Suppose we are writing a small server-side service in Node, and we want 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 to register for usage) to get the alarms, our code would be something like this; all quite normal... Do notice 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 FOR
      uri:
        "https://alerts.weather.gov/cap/wwacapget.php?x=TX125862DD4F88.Tropic
    },
    ...
    {
      title: "Hurricane Local Statement",
      regions: ["Austin", ..., "Wharton"],
      severity: "advisory",
      time: 1503748800,
      expires: 1503683100,
      description:
        "This product covers Southeast Texas **HURRICANE HARVEY DANGEROUSLY
      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 totally exclude the `alerts:[...]` part. So, 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>` +
                `

|  |

```

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 whatever you needed to do. In any case, the resulting output would match something like this:

```

<table><tr><td>Tropical Storm Warning</td><td>...TROPICAL STORM WARNING REMAINS IN E
<tr><td>Flash Flood Watch</td><td>...FLASH FLOOD WATCH REMAINS IN EFFECT THROUGH MONI
<tr><td>Hurricane Local Statement</td><td>This product covers Southeast Texas **PREP

```

If we had instead called `getAlerts(-34.9, -54.60, ...)` with the coordinates for Montevideo, Uruguay, since there were no alerts for that city, the `getField("alerts")` function would have returned `undefined` -- and as 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. See figure 12.1:

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.

We did take advantage of this behavior also when coding the error logic. If an error happens when calling the service, we would still call the original callback to produce a table, but providing 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:

```
class Maybe extends Functor {  
  //  
  // everything as before...  
  //  
  orElse(v) {  
    return this.isNothing() ? v : this.valueOf();  
  }  
}
```

Using this new method instead of `valueOf()`, if trying to get the alerts for someplace without them, you would just get whatever default value you wanted. In the case we cited before when attempting to get the alerts for Montevideo, instead of a `null` value, we would now get an appropriate result:

```
getAlerts(-34.9, -54.6, x =>  
  console.log(  
    produceAlertsTable(x).orElse("<span>No alerts today.</span>")  
  )  
);
```

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 could 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 something to our functors and enter the domain of *monads*.

Monads

Monads have a weird fame among programmers. Well known developer Douglas Crockford has famously spoken of *their* "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 a book like *Categories for the Working Mathematician* by Saunders Mac Lane (one of the creators of Category Theory) you may find a somewhat disconcerting explanation: *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.* Not too illuminating!

The difference between monads and functors is just that the former adds some extra functionality. Let's start by seeing the new requirements, and afterwards move on to consider 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 solving specific cases.



If you want to read a precise and careful description of functors, monads, and all 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, whatever) 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:

```
const fakeSearchForSomething = key => {
  if (key % 2 === 0) {
    return {key, some: "whatever", other: "more data"};
  } else {
    throw new Error("Not found");
  }
};

const findSomething = key => {
  try {
    const something = fakeSearchForSomething(key);
    return Maybe.of(something);
  } catch (e) {
    return Maybe.of(null);
  }
};

const getSome = something => Maybe.of(something.map(getField("some")));

const getSomeFromSomething = key => getSome(findSomething(key));
```

What's the problem here? The problem is that the output from `getSome()` is a `Maybe` value, which itself contains a `Maybe` value, so the result we want is double wrapped:

```
let xxx = getSomeFromSomething(2222).valueOf().valueOf(); // "whatever"
let yyy = getSomeFromSomething(9999).valueOf().valueOf(); // null
```

This problem can be easily solved in this toy problem (just avoid using `Maybe.of()` in `getSome()`), but this kind of result can happen in many ways, in 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 if you'd get the same situation when accessing that attribute: you would end up with some doubly wrapped value.

Monads should provide the following operations:

- A constructor.
- A function that inserts a value into a monad: our `.of()` method.
- A function that allows for chaining operations: our `.map()` method.
- A function that can remove extra wrappers: we will call it `.unwrap()`, and it will solve our preceding multiple wrapper problems. Sometimes it's 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 itself a container anymore. Using this method, we could avoid doubly wrapping easily:

```
| const getSomeFromSomething = key => getSome(findSomething(key)).unwrap();
```

However, this sort of problem could recur at different levels. For example, if we were doing a series of `.map()` operations, any of the intermediate results may end up being doubly wrapped. You could easily solve this by remembering to call `.unwrap()` after each `.map()` -- note that you could do it even if it is not actually needed, for in that case 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, which will do both things for us (sometimes `.chain()` is called `.flatMap()`):

```
| 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 provided only one argument to `add`, the result of that application will be a function... not just any function, 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? In order to be able to apply this wrapped function to a value, we'll need a new method: `.ap()`. What could the 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:

```

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)

```

Now, you can use monads to hold either values or functions, and to interact with other monads and chaining operations as you may wish. So, as you can see, there's no big trick to monads, which are just functors with some extra methods. Let's see now 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 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 it represents some kind of special value (for example, an error message or a thrown exception), which 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 than 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:

```
class Left extends Monad {
  isLeft() {
    return true;
  }

  map(fn) {
    return this;
  }
}

class Right extends Monad {
  isLeft() {
    return false;
  }

  map(fn) {
    return Either.of(null, fn(this[VALUE]));
  }
}

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 a `AJAX FAILURE` message or with the result from the API call:

```

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 become as follows. We use an `Either` again: if there are no alerts, instead of an array, we 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>` +
                    `

|  |

```

Notice how we used `.chain()`, so multiple wrappers would be no problem. We can now 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 for a better way of achieving this, through 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 it 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

```
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 called with a null argument:

```
// getField :: String → attr → a | undefined  
const getField = attr => obj => obj[attr];
```

We can rewrite it using the *Try* monad, so it will *play nice* with other composed functions:

```
const getField2 = attr => obj => Try.of(() => obj[attr], "NULL OBJECT");  
  
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 do visit just one more, which you have been using, 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*! We already commented, earlier in this chapter, 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, whose value 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()` and also `Monad.chain()`, given the mentioned unwrapping.
- And 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. And now, in the same way, that Monsieur Jourdain (a character in Molière's play *Le Bourgeois Gentilhomme*, *The Burgeois Gentleman*) discovered that all his life he had been speaking in prose, you now know you had already been using monads, even without knowing it!

Functions as data structures

So far, we have seen how to use functions to work with or transform, other functions, to process data structures, or to create data types. Let's then finish this chapter by showing 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 might very well say that we have come around to the point where we began this book, at the origins of Functional Programming!

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.



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 particular 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 that 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
```

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 `empty _ = False` because you don't actually care for 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 similarly written:

```
| 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
```

An empty tree doesn't contain the searched value. For other trees, if the root matches the searched value, we are done. If the root is greater than the searched value, the answer is found searching in the left subtree; otherwise, search in the right subtree.

There's an important point to be remembered: for this data type, a union of two possible types, we have to provide two conditions, and pattern matching will be used to decide which one is 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 -- and mind: not with a data structure that is processed by a set of functions, and not either with an object with some methods, but by just a function. Furthermore, we will get a functional data structure, 100% immutable, which if updated produces a new copy of itself. And, we will do all this without using objects; rather, closures will provide the desired results.

How can this work? We shall be applying similar concepts as those we saw earlier in the 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 backwards, and start by showing how we'll use the new data type, and then go to the implementation details.

Creating a tree will be done by using two functions: `EmptyTree()` and `Tree(value, leftTree, rightTree)`. For example, creating a tree as shown in figure 12.2, would be done by using the following code:

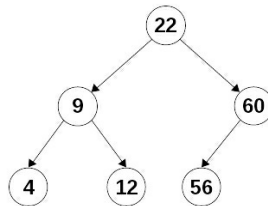


Figure 12.2. A binary search tree, created by the following code.

```
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, and the second function will receive none. So, to get the root we could write something as follows:

```
| const myRoot = myTree((value, left, right) => value, () => null);
```

If we were dealing with a non-empty tree, we 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 are there 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 zero. Get the idea?

We can now show the `Tree()` and `EmptyTree()` functions:

```
| 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 JS, 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 just as a placeholder, to stand for an otherwise ignored argument, but showing that two arguments are assumed.

This can be hard to understand, so let's see some more examples. If we need to access specific elements of a tree, we have the following three functions, one of

which (`treeRoot()`) we already saw -- 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 might find it elsewhere.

How can we decide if a tree is empty? See if you can see why this short line works:

```
const treeIsEmpty = tree => tree(() => false, () => true);
```

Let's just go over a few more samples 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 shorter:

```
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 more clear:

```
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 closely the definition we saw in the previous section (we could have shortened the code a bit, but I did want to parallel the Haskell version):

```

const treeSearch = (findValue, tree) =>
  tree(
    (value, left, right) =>
      findValue === value
        ? true
        : findValue < value
          ? treeSearch(findValue, left)
          : treeSearch(findValue, right),
    () => false
  );

```

And, to round out this section, let's also include how to add new nodes to a tree. Study the code carefully, and you'll notice how the current tree isn't modified, and 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:

```

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 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 would not have been symmetrical, but analogous. And if we try to insert a new key, and we find ourselves an empty tree, we just replace that empty structure with a new tree, with just the new value as its root, and empty left and right subtrees.

We can test out this logic easily -- but the simplest way is to verify that the binary tree is shown 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:

```

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 may program diverse functions: deleting a node, counting nodes, determining a tree's height, comparing two trees, and so on. But, 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:

```

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 won't change the important conclusions we can derive from this work:

- We are handling a data structure (a recursive one, at that) representing it with a function
- We are not 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 it is immutable and all changes always produce new structures
- And, finally, the tree is a functor, providing all the corresponding advantages

So, we have seen even one more application of functional programming -- and we've seen how a function can actually become a structure by itself, which isn't what one is usually accustomed to!

Questions

12.1. Maybe tasks? In the questions section of [Chapter 8, Connecting Functions - Pipelining and Composition](#), a question had to do with getting the pending tasks for a person, but taking into account errors or border situations, such as the possibility that the selected person might not even exist. Redo that exercise, but using Maybe or Either monads to simplify that coding.

12.2. Extending your trees. In order 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 of the 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-liner 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
- Append one list to the end of another list
- Concatenating two lists

Don't forget a `listMap()` function! Also, `listReduce()` and `listFilter()` functions would 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 style if only because you're likely to find it.

Summary

In this chapter, we turned a bit closer to theory and seeing how to use and implement data types from a functional point of view. We started with ways of defining function signatures, to help understand the transformations implied by the multiple operations we later met; then, we went on to define several containers, including functors and monads, and see how they could be used to enhance function composition, and finally we saw how functions could be directly used by themselves, with no extra baggage, to implement functional data structures.

As of now, in this book we have seen several features of functional programming for JavaScript. We started out with some definitions, 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 data flow based upon function connections and data containers. We have seen a lot of concepts, but I'm confident that you'll be able to put them in practice, and start writing even higher quality code - give it a try!

Bibliography

The following texts are freely available online:

- *ECMA-262: ECMAScript 2017 LANGUAGE SPECIFICATION*, 8th Edition, at www.ecma-international.org/ecma-262/8.0/. This provides the official standard to the current version of JS
- *ELOQUENT JAVASCRIPT*, 2nd 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/>. *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 DrBoolean (Brian Lonsdorf), at <https://github.com/MostlyAdequate/mostly-adequate-guide>.

If you prefer printed books, you can go with this list:

- *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
- *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 HIGH PERFORMANCE*, Chad Adams, Packt Publishing, 2015
- *PRO JAVASCRIPT PERFORMANCE*, Tom Barker, Apress, 2012

Answers to Questions

Here are solutions (partial, or worked out in full) to the questions posed along the text. In many cases, there are extra questions for you to do further work.

1.1. Classes as first class objects. If you remember that a class is basically a function that can be used with `new`, then it stands to reason that we should be able to pass classes as parameters to other functions. The `makeSaluteClass()` basically creates a class (that is, a special function) that uses a closure to remember the value of `term`. We'll be seeing more examples of these kind of things in the rest of the book.

1.2. Factorial errors. The key to avoiding repeated tests is to write a function that will:

- First check the value of the argument to see it 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 on recognizing a wrong argument; I just ignored it, and let the function return `undefined`.

1.3. Climbing factorial. The following code does the trick. We add an auxilliary variable `f`, and we make it "climb" from 1 to `n`. We must be careful so `factUp(0)===1`.

```
| const factUp = (n, f = 1) => (n <= f ? f : f * factUp(n, f + 1));
```

2.1. No extra variables. We can make do by using the `fn` variable itself as a flag; after having called `fn()`, we set the variable to null. Before calling `fn()`, we check if 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. We used a destructuring assignment to write the swap in a more compact manner: see 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 if the `limit` variable is greater than zero. If so, we decrement it by 1, and we call the original function; otherwise, we do nothing.

```
| const thisManyTimes = (fn, limit) => {  
|   return (...args) => {  
|     if (limit > 0) {  
|       limit--;  
|       return fn(...args);  
|     }  
|   };  
|};
```

3.1. Uninitialized object? The key is that we missed wrapping the returned object in parentheses, so JS thinks the braces enclose code to be executed. In this case, "`type`" is considered to be labelling a statement, which doesn't really do nothing: it's an expression (`t`) with which nothing is done. So, 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 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! (Yes, a one line answer is appropriate in this case!)

4.1. Minimalistic function. It works, just 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 you'd calculate a Fibonacci number by hand. You'd start writing down $fib(0)=0$ and $fib(1)=1$, add them to get $fib(2)=1$, add 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. A simple idea: 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, the two produced JSON strings, which should be equal. This does away with all the other tests, since it ensures that the array doesn't change length, nor change 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 the examples, let's assume two numbers are close to each other if they differ by no more than 0.1. Then:

- 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 $0.6*0.8=0.48$
- 0.6 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.

5.1. Filtering... but what? `Boolean(x)` is the same as `!!x`, and it turns an expression from "truthy" or "falsy", into true or false. 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 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><ul> ... </ul></div>` tags is tricky, so we had to make the first `.reduce()` operation produce an array, so 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 to 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, "");

```

Or also:

```

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 for further work of your own.

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()`-ed results: they should match.

5.4. Ranging far and wide. This requires a bit of careful arithmetic, but shouldn't be much trouble. We 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 that.

```

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 of options.

```

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 of `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 (current value, index, array) and that causes unexpected results. Using `unary()` from section "Arity Changing" of [Chapter 6, Producing Functions - Higher-Order Functions](#), would also work. See more on https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/String/fromCharCode.

5.6. Producing a CSV. A first solution, with some auxiliary functions, is as follows; can you understand what does each function do?

```
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 so clear, in my opinion... do you agree?

```
const makeCSV2 = t =>
  t.reduce(
    (c, d) => c + "\n" + d,
    " ",
    t.map(x => x.reduce((a, b) => (a == " " ? b : a + "," + b), " "))
  );
```

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 might opt to produce `undefined` instead, or work with monads as in the last chapter of the 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 needed to evaluate $fib(n)$. Analyzing the tree that shows all the needed calculations, we get this:

- $calc(0)=1$
- $calc(1)=1$
- 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 Maths 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. 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 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 that `randomizer()` is written. Can you provide a better version, that won't have this small defect, so that *all* 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 see if the returned value is numeric or boolean, before deciding what to return.

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 => `

7.3. Currying by prototype. Basically, we are just transforming the curryByBind() version to use 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.



7.4. Uncurrying the curried. 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 a 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 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
```


```


8.1. Headline Capitalization. We can make use of several functional equivalents of different methods, such as `.split()`, `.map()`, or `.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 tasks are done; can you see why? Also note that if `allTasks` is `null`, an object must be provided with the `.byPerson` property, so future functions won't crash! For an even better solution, I think monads are better: see question 12.1.

8.3. Thinking in abstract terms. The simple solution implies composing. I preferred it to pipelining, to keep the list of functions in the same order.

```
| const getSomeResults2 = compose(sort, group, filter, select);
```

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 doing `reverse("ONTEVIDEO")+"M"`. In the same way, `reverse("ONTEVIDEO")` would equal `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 it in two ways:

- climbing one single step, and then climbing a $(n-1)$ steps ladder, or
- climbing two steps at once, and then climbing a $(n-2)$ steps ladder.

So, if we call *ladder*(n) the number of ways to climb a n steps ladder, we have that *ladder*(n) = *ladder*($n-1$) + *ladder*($n-2$). Adding the fact that *ladder*(0)=1 (there's only 1 way to climb a ladder with no steps: do nothing) and *ladder*(1)=1, the solution is that *ladder*(n) equals the $(n-1)$ Fibonacci number! Check it out: *ladder*(2)=2, *ladder*(3)=3, *ladder*(4)=5, etc.

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; and
- 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

An implementation follows. We do memoization "by hand" to avoid repeating calculations; we could also have 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 involved.

9.4. Symmetrical queens. The key to finding only symmetric solutions, is that after the first 4 queens have been (tentatively) placed in the first half of the board, we don't have to try all 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!

```
[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 see just 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. Check how we dealt with the mutator functions, 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]
```

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 the subject.) 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");
    }
  });
};
```

We can see the results of this. You'd probably require something other than showing

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, 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));
  }
};
```

We can see this working. The new list is just as 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, so I included several comparisons. The last comparison below shows that from the "F" node onwards, 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
```

As we implemented this, if `oldKey` isn't found, nothing is inserted. Could you change the logic so in that case, the new node would be added at the end of the list?

11.1. Decorating methods, the future way. As we said, decorators aren't yet a fixed, definitive feature. However, 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 saving its return value, log that, and finally return. If the original method throws an exception, we catch it, report it, and throw it again so it can be processed as expected. A simple example 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 first calls the original constructor, and then creates the `.barValue` attribute. The new class has both the original's `.doSomething()` method, and the new `.somethingElse()`

one.

```
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
```

12.1. Maybe tasks? The following code shows a simpler solution than the one we saw for 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();
```

We just 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. At the end you will either get an array of task id's, or a null value.

12.2. Extending your trees. Calculating the tree's height is simple, in a recursive fashion. The height of an empty tree is zero, and 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 build, you first list the left subtree's keys, 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
- and if the node has two subtrees, then you have to find the minimum key in the tree with a greater key, and 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.

```
|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 already given. 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, or appending a value to a list, have simple recursive implementations. And, we can 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
  );

```

Some exercises left to you:

- 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. A first approach would 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.