Refactoring JavaScript

Turning Bad Code into Good Code

Evan Burchard

Refactoring JavaScript
by Evan Burchard

Printed in the United States of America.
Published by O'Reilly Media, Inc., 1005 Gravenstein Highway North, Sebastopol, CA 95472.
O'Reilly books may be purchased for educational, business, or sales promotional use. Online editions are also available for most titles (**http://oreilly.com/safari**). For more information, contact our corporate/institutional sales department: 800-998-9938 or *corporate@oreilly.com*.

- Editors: Nan Barber and Allyson MacDonald
- Production Editor: Kristen Brown
- Copyeditor: Rachel Monaghan
- Proofreader: Rachel Head
- Indexer: Ellen Troutman-Zaig
- Interior Designer: David Futato
- Cover Designer: Karen Montgomery
- Illustrator: Rebecca Demarest
- March 2017: First Edition

Revision History for the First Edition

- 2017-03-10: First Release

See **http://oreilly.com/catalog/errata.csp?isbn=9781491964927** for release details.

978-1-491-96492-7
[LSI]

For Jade, again and always.

Table of Contents

Foreword

I still remember reading Martin Fowler's book *Refactoring: Improving the Design of Existing Code* when it came out in 1999. It was a revelation: I had never before seen code being considered to be something malleable. Programmers tend to have the urge to rewrite code bases from scratch, but this book argued that it is possible to evolve and clean up existing code via small, principled, and comparatively safe steps. While doing so, tests provide an additional safety net and enable you to move forward with confidence. One piece of advice from the book will always stick with me—whenever you code, always keep two kinds of activity completely separate: implementing new functionality and refactoring existing code. If you do that, you'll avoid doing too many things at the same time and will produce less bugs.

Refactoring JavaScript takes the ideas of refactoring and applies them to the world of JavaScript. JavaScript's dynamic nature means that you need different techniques compared to more static languages such as Java. In Java, you have static typing. And inheritance and polymorphism are used quite often. For JavaScript, you often rely on static checking tools (such as ESLint and Flow) and can adapt objects flexibly according to your needs. Functional programming techniques are also more popular. Additionally, tests play an even more important role, but they also tend to be more lightweight. With all of these issues and more (e.g., asynchronous code), this book has you covered!

Happy reading!

—Axel Rauschmayer
—November 2016

Preface

Welcome to *Refactoring JavaScript*. Throughout this book, we'll be looking at ways to write better JavaScript, drawing inspiration from classical refactoring techniques while exploring various styles of coding.

Why This Book Exists

Like it or not, JavaScript is not going away. No matter what framework or "compiles-to-JS" language or library you use, bugs and performance concerns will always be an issue if the underlying quality of your JavaScript is poor. Rewrites, including porting to the framework of the month, are terribly expensive and unpredictable. The bugs won't magically go away, and can happily reproduce themselves in a new context. To complicate things further, features will get dropped, at least temporarily.

This book provides clear guidance on how best to avoid these pathological approaches to writing JavaScript. Bad code doesn't have to stay that way. And making it better doesn't have to be intimidating or unreasonably expensive.

Who This Book Is For

This book is meant for programmers who have some experience writing bad code, and an interest in writing better code. It's for those writing JavaScript on the frontend or the backend. It's for those writing JavaScript by choice as well as those who are "stuck with it" due to JavaScript's monopoly of the browser platform.

If you're an absolute beginner, you might want to write some bad code for a couple of months first. If you're not interested in writing better code, you might not have the patience for this book. If neither of those situations describes you, we're good to go.

Interestingly enough, there are numerous efforts working to make JavaScript better, while at the same time others aim to make it obsolete. The number of ways to write good and bad JavaScript continues to expand. Frameworks

can go a long way toward handling complexity, but programmers constrained by frameworks will be limited. If you find that you (or your codebase) are struggling to work outside of a framework (or at some of the more confusing edges of it), this book should give you new ideas for how to approach your work.

If you have trouble testing, debugging, or having confidence in your codebase, this book should be helpful.

Most of us don't work on perfect codebases, especially in JavaScript, where engineers might primarily use Ruby, Python, Java, and so on. What this book does is help you identify what specific parts of a codebase are bad, while providing a multitude of options for improvement.

How To Use This Book

Chapters **1**–**5** describe the interplay between JavaScript, refactoring, quality, confidence, and testing. In many books, it is common to tack on testing at the end. In this book, for the types of code we are exploring, this wouldn't be appropriate. Testing is essential for confidence. Confidence is essential to refactoring. Refactoring is essential to quality, which is the goal:

```
testing -> confidence -> refactoring -> quality
```

JavaScript (and its ecosystem) happens to provide the space in which that transformation takes place, so these opening chapters necessarily include an exploration of the language itself. If you're completely comfortable with the transformation just described, you might want to skim or skip these chapters. Although that is not recommended, it is your book, so you can use it however you want. If you think it's best used as a doorstop or to start a fire for warmth or a sacrifice of some sort, go for it. If you do find an unconventional use for the book, email me a picture or video. I'm at ***http://evanburchard.com/contact*** or @evanburchard on Twitter and GitHub.

CAN I BURN OR DOORSTOPIFY DIGITAL COPIES TOO?

Unfortunately, no. *However*, since this book is under a Creative Commons license, you're free to share links to the HTML version and any other files available at ***http://refactoringjs.com***, for example.

After **Chapter 5**, things get harder, especially if you skipped 1–5. There's more code to write and follow along with. In Chapters **6** and **7**, we go through refactoring functions and objects, and we don't shy away from some of the more complicated bits of JavaScript. Generally, the goal of these chapters is to provide options for improving code without radically changing the interface.

Through applying techniques found in these two chapters, you'll be able to turn a mess of a codebase into one that has a decent baseline of quality.

Chapter 8 expands our view of architecture to those that include (or avoid) hierarchies.

Chapters **9**, **10**, and **11** are dedicated to specific topics (design patterns, asynchronous programming, and functional programming, respectively) that can take your code beyond that baseline, but necessarily involve more aggressive changes. With the design patterns in **Chapter 9**, we recognize ways to extend and draw from JavaScript's object-oriented side, and cover the historical connection between refactoring and object-oriented programming (OOP). In **Chapter 10**, we deal with the reality that many JavaScript codebases have more than one thing to do at once. In **Chapter 11**, we take a tour of functional programming through a couple of libraries that provide useful interfaces that go beyond those that our standard `Array` functions (`forEach`, `map`, `reduce`, etc.) give us.

In some sense, those last three chapters in particular break away from our initial goal of refactoring by changing the *implementation details* without changing the *interface*. On the other hand, these interfaces are both useful and sometimes unavoidable. We may easily find ourselves wanting to write code that is necessarily asynchronous for performance reasons. Or we could find ourselves "stuck" in a codebase that has much invested in *good or bad* attempts at OOP or functional programming (FP). So, either through choice or code we inherit, these are parts of JavaScript that we should pay attention to and be able to improve upon. If you adopt a completely different paradigm to a codebase, it is unlikely that you'll be "refactoring" in the sense that we mean throughout most of the book.

If we want to be rigid about it, these chapters still refactor *within* their paradigms (OOP to better OOP, async to better async, and FP to better FP), and if we wish to think in the broadest terms about the execution of our program (e.g., "running **`node myprogram.js`**" as *input* and "being satisfied with how it ran" as *output*), then we can be refactoring even while jumping from paradigm to paradigm. I encourage you to first work with smaller, incremental changes that are easy to test and be confident in.

To quote William Opdyke's original thesis on refactoring:

> This definition of semantic equivalence allows changes throughout the program, as long as this mapping of input to output values remains the same. Imagine that a circle is drawn around the parts of a program affected by a refactoring. The behavior as viewed from outside the circle does not change. For some refactorings, the circle surrounds most or all of the program. For example, if a variable is referenced throughout a program, the refactoring that changes its name will affect much of the program. For other refactorings, the circle covers a much smaller area;

> for example, only part of one function body is effected when a particular function call contained in it is inline expanded. In both cases, the key idea is that the results (including side effects) of operations invoked and references made from outside the circle do not change, as viewed from outside the circle.[1]

Although we're free to draw "the circle" as large as we'd like, it's very common for the term *refactoring* to get thrown around as though it simply meant "changing code." As we discuss in **Chapter 1**, it does not. That is easier to see on a small scale, like the ones that we spend the most pages on. Think of Chapters **8**, **9**, and **10** first presenting as architectural options, and second possibilities for creating better code (safely and incrementally) within those options. As an example, if someone says something about "*refactoring* to use asynchronous code" it is likely too broad of a problem to execute in a safe and incremental way. But if you want to think of **Chapter 9** as giving you the power to do so, I can't stop you. It's your book now. You can draw the circle as big as you want.

If you find any of the tools or concepts confusing, you will probably find the appendix helpful. If you are looking for code samples and other information, visit the book's **website** (http://refactoringjs.com). You can also find an HTML version of the book there if you prefer to read that way.

So in summary, use this book to learn about:

- Refactoring
- Testing
- JavaScript
- Refactoring and testing JavaScript
- A few JavaScript paradigms
- Refactoring and testing within those JavaScript paradigms

Alternatively (under adult supervision, of course), the paper version of the book can be set on fire and used for:

- Warmth
- Protest
- Ritual tech book sacrifices

The digital files from ***http://refactoringjs.com*** can get passed around in accordance with the Creative Commons license restrictions.

If you have any problems, questions, complaints, or compliments, feel free to reach out to me through **my website** (http://evanburchard.com/contact).

1 William Opdyke, "Refactoring Object-Oriented Frameworks" (PhD thesis, University of Illinois at Urbana-Champaign, 1992), 40.

Some Words in This Book

App, Application, Program

Some words in this book are imprecise. *App*, *application*, *program*, and *website* are interchangeable much of the time. In case there is any confusion, this book describes general principles for improving the quality of JavaScript, so none of those terms should be taken too literally. Maybe your code is a library or framework? In any case, the techniques in this book should apply just fine.

Inclusivity Through Words and Diagrams

Some words in this book may not feel inclusive to everyone. I tried to balance the use of *he* and *she*, which I realize isn't everyone's ideal. Although I'd prefer to use the singular they, that's not within publisher guidelines at the moment.

Additionally, I am realizing (too late) that my reliance on diagrams, especially those in Chapter 5, may do a terrible disservice to readers with a visual impairment. If you feel like you've missed out on any content for this reason, please feel free to reach out to me with any questions.

Users

There's also one word in this book I really hate, and that's *users*. It's imprecise and also creates some distance between the creators (developers/designers) and consumers (users). More precise and charitable words are often specific to the problem domain, or else we're stuck with terms like "people" or "people who use the program/website." If there is no more specific term than *person* or *user* (even including *customer*), it might be a hint that the business model is based purely on selling people as data, but that's another discussion.

The point is that the term *user* is used in this book to convey a familiar concept: a person who uses the program/website. Also, there are not yet magnanimous and accurate terms to supplant the related terms of *user experience* (UX) or *user interface* (UI). Rather than explaining this in several places or using nonstandard or specific terms for frequently abstract concepts, I chose to save the effort and just talk about it here.

In any case, I fully endorse the following quote (and its implications) by "the Leonardo da Vinci of Data," Edward Tufte:

> There are only two industries that refer to their customers as users: illegal drugs, and software houses.

There is a movement called “ethical design” that hopefully will help the industry shed this term (and the inconsiderate practices that stem from it) at some point.

Third-Party Libraries and Communities

Although I tried very hard to present the best tools to demonstrate the fundamentals of refactoring and testing in JavaScript, there may be times where you find that a particular tool isn’t working for you. The great news here is that JavaScript has a rich ecosystem of options. I have a preference for tools that are simple, flexible, and atomic, but you may feel differently. Large frameworks in particular are not explored in this text, as they tend to come with their own ecosystems of other tools (often themselves quite active and varied). I would absolutely recommend a framework when you’re starting out, but they are most useful when combined with facility in the underlying language, which I believe this book will teach you very well.

Additionally, every tool, framework, and library will come with some community and history. Just as I don’t believe in any one true way for tooling, I also don’t endorse the community behind any given third-party code or project. Many projects will come with a code of conduct that will let you know if participating in them will be an enjoyable use of your time.

API, Interface, Implementation, “Client Code”

This gets a little murky, but one thing I wish I could highlight more is the hierarchy not in terms of objects, but in the interface of a well-designed codebase. When code is a simple script, we expect it to run top to bottom, as a procedure. As a codebase matures (through design, not butchery mixed with entropy), we expect it to develop in three main layers (although this is obviously extended in more complex codebases).

The first layer—the code behind the scenes, deeper in the codebase—is referred to in this book as the *implementation*. For refactoring, the most important distinction is between the implementation and the next layer. This second layer can be called the *interface* or *API* and describes the “public” functions and objects that one should expect to interact with if a codebase is used as a module. The third layer of consequence is sometimes called the *client code* or *calling code*. It refers to the code that is written to interact with the interface layer. This is the code that people using a module would write, as well as the testing code that we will write to test the interface layer.

BASICS OF ARCHITECTURE

Throughout this book, we're creating programs that start out very unstructured, and our main thrust (regardless of a paradigm like OOP or FP) is to make divisions between these three layers. This is what allows code to be testable and portable. If you're mostly reliant on frameworks that provide their own organization, this process might be unfamiliar.

Inputs (Nonlocal and Free Variables)

Throughout the book (especially in Chapter 5), we distinguish between three types of inputs:

- Explicit inputs (the parameters passed into a function)
- Implicit inputs (the `this` that refers to the containing context object, function, or class)
- Nonlocal inputs (the values used within a function or object that are defined elsewhere)

There are two things of note here. First, local variables (or constants) created within the scope of a function are not considered "inputs" for the sake of diagramming or otherwise. Second, although the term *nonlocal input* is used as a precise term in this text, *free variable* is a more common name. However, it is a bit imprecise given that nonlocal inputs may be constants rather than variables. Similarly, some use the term *bound variables* to refer to what we call *explicit inputs* and, to some degree, *implicit inputs* as well.

Conventions Used in This Book

The following typographical conventions are used in this book:

Italic
: Indicates new terms, URLs, email addresses, filenames, and file extensions. Also used on occasion for emphasis and contrast.

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

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

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

> This element signifies a tip or suggestion.

> This element signifies a general note.

> This element indicates a warning or caution.

Using Code Examples

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

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

We appreciate, but do not require, attribution. An attribution usually includes the title, author, publisher, and ISBN. For example: "*Refactoring JavaScript* by Evan Burchard (O'Reilly). Copyright 2017 O'Reilly Media, 978-1-491-96492-7."

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

Tools used within the code of this book can be found in the appendix, along with resources for further information on topics covered. For reference, the tools used in this book along with their versions are:

- node 6.7.0
- npm 3.10.3
- wish 0.1.2
- mocha 3.2.0
- deep-equal 1.0.1
- testdouble 1.10.0
- tape 4.6.3

- lodash 4.17.2
- assert 1.4.1
- underscore 1.8.3
- ramda 0.22.1
- sanctuary 0.11.1

Later versions are unlikely to cause problems, but earlier ones might. At lower versions, node in particular is known to not fully support the code in this book.

O'Reilly Safari

> ***Safari*** (formerly Safari Books Online) is a membership-based training and reference platform for enterprise, government, educators, and individuals.

Members have access to thousands of books, training videos, Learning Paths, interactive tutorials, and curated playlists from over 250 publishers, including O'Reilly Media, Harvard Business Review, Prentice Hall Professional, Addison-Wesley Professional, Microsoft Press, Sams, Que, Peachpit Press, Adobe, Focal Press, Cisco Press, John Wiley & Sons, Syngress, Morgan Kaufmann, IBM Redbooks, Packt, Adobe Press, FT Press, Apress, Manning, New Riders, McGraw-Hill, Jones & Bartlett, and Course Technology, among others.

For more information, please visit ***http://oreilly.com/safari***.

How to Contact Us

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

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

We have a web page for this book, where we list errata, examples, and any additional information. You can access this page at ***http://bit.ly/refactoring-js_1e***.

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

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

Find us on Facebook: ***http://facebook.com/oreilly***
Follow us on Twitter: ***http://twitter.com/oreillymedia***
Watch us on YouTube: ***http://www.youtube.com/oreillymedia***

Acknowledgments

Thanks to my family for their support in making this book happen: Mom, Dad, Amy, Scott, Gretchen, Max, and Jade.

Special thanks to the people who helped kick everything off: Zeke Templin, Steve Souders, Mary Treseler, Simon St. Laurent, and Tyler Ortman.

And to those who gave technical inspiration and feedback: Jacob Barss-Bailey, Matt Blake, Charles Baakel, Stefano De Vuono, and Ryan Duchin.

And the rest of the O'Reilly staff that helped along the way: Annalis Clint, Nena Caviness, Michelle Gilliland, Rita Scordamalgia, Josh Garstka, Kristen Brown, Rebecca Demarest, Rachel Monaghan, Shiny Kalapurakkel, and especially my editors, Nan Barber and Ally MacDonald.

And to my technical reviewers: Steve Suering, Shelley Powers, Chris Deely, Darrell Heath, and Jade Applegate.

And to those whose work I found useful and inspirational: William F. Opdyke, Martin Fowler, Kent Beck, John Brant, Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides, Douglas Crockford, Tony Hoare, Alexis Deveria, Addy Osmani, Robert Nystrom, Brian Lonsdorf, Reginald Braithwaite, Miran Lipovaca, Kyle Simpson, Tom Stuart, Michael Fogus, David Chambers, Michael Hurley, Scott Sauyet, Yehuda Katz, Jay Fields, Shane Harvie, Russ Olsen, Joshua Kerievsky, James Halliday, TJ Holowaychuk, Justin Searls, Eric Elliot, Jake Archibald, Arnau Sanchez, Alex Chaffee, Eric Hodel, Sean Hussey, Brian Cardarella, Foy Savas, and Katrina Owen, and Bryan Liles.

A special thanks to Dr. Axel Rauschmayer for his amazing work interpreting specs for us mere mortals, as well as providing the foreword to this book.

PSST...HEY, READER!

I know it looks like just a big list of names, but the people in this section are all really awesome. The resources in the appendix are less important than this list. A lot of these people made that stuff. And searching their names will let you know about their new stuff, which is probably better than their old stuff. Look these people up.

And thanks in general to all the people at TC39 and MDN.

And to my dog for taking me on walks, even when I was right in the middle of something.

Also, to you. Thanks for supporting my work. Hit me up if you need anything.

1 What Is Refactoring?

Refactoring is not changing code.

Okay, yes, it is, but there's more to it. Refactoring is a type of changing code, but has one major constraint that makes "changing code" an imprecise way to describe it: you don't change the *behavior* of the code. Two immediate questions should come to mind:

- How do you guarantee that behavior does not change?
- What is the point of changing the code if the behavior doesn't change?

In the rest of the chapter, we will pursue the answers to these questions. We're not going to go through the full history of JavaScript, as that's covered extensively on the web already.

How Can You Guarantee Behavior Doesn't Change?

Unqualified, the answer to that question is that it is incredibly hard. Fortunately, many types of behavior are not our primary concern when refactoring. We'll cover these next:

- Details of implementation
- Unspecified and untested behavior
- Performance

The shorter answer, for us, is using tests and version control.

Another approach, supported by William Opdyke, whose **thesis** (http://www.ai.univ-paris8.fr/~lysop/opdyke-thesis.pdf) is the foundational work on refactoring, stresses using automated tools that are responsible for changing the code as well as guaranteeing safety *before* doing so. Professional coders might find that removing the human element limits the types of changes that can be made, as the number of changes that can be guaranteed as "safe" is confined to the functionality of the tools.

Writing tools to encompass the whole refactoring catalog proposed by Martin Fowler in his seminal book, *Refactoring: Improving the Design of Existing Code* (Addison-Wesley), would prove extremely difficult. And in JavaScript, a dynamic, multiparadigmatic language with an ecosystem teeming with variants (see **Chapter 2**), these tools are bound to lag behind a refactorer's imagination even more so.

Fowler's approach pulls away from automation, while at the same time stressing the "mechanics" of the refactoring: steps of altering code that minimize unsafe states.

If we relied on an "Opdykian," automated approach for this book, the tooling would hold us back significantly. And we're straying from Fowler's emphasis on mechanics (step-by-step processes) as well. The reason is that, as we move toward confidence in our code through a given refactoring, if it is backed up by tests, verifying the success of our changes should be straightforward. And when we fail to execute a refactoring properly, version control (we'll be using Git) should give us an easy way to simply "roll back" to the state of the code beforehand.

WARNING! USE VERSION CONTROL!

Any form of "changing code" carries significant risk to your codebase if you cannot easily revert it to a previous, safe version. If you don't have versioned backups of the codebase that you plan on refactoring, put this book down and don't pick it up again until you have your code under version control.

If you're not using version control already, you probably want to use **Git** (http://git-scm.com/), and you probably want to back your work up on **GitHub** (http://github.com).

Admittedly, the approach of this book might seem reactive and cavalier in comparison to the earlier paths of automation and mechanics. However, the process—the "red" (failure state of a test), "green" (passing state of a test), "refactor" cycle, with an eye on rolling back quickly if things go wrong—employed in this book is based upon how quality-focused teams operate with tools that are popular among them. Perhaps later, automated refactoring will catch up with Fowler's extensive catalog of refactorings, as well as all that are presented in this book, but I wouldn't count on it happening soon.

Our goal here is to get JavaScript developers out of the muck. Although it is tempting to try to automate and mechanize that process, the most valuable parts of the work of the giants (Opdyke, Fowler, Johnson, et al.) whose shoulders this book humbly stands on is that they gave us a new mindset around making code better and doing it safely.

Why Don't We Care About Details of Implementation?

Let's say we have a simple function that multiplies numbers by 2:

```
function byTwo(number){
  return number * 2;
}
```

We could instead write a function that accomplishes the same goal in a slightly different way:

```
function byTwo(number){
  return number << 1;
}
```

And either of these will work fine for many applications. Any tests that we used for the byTwo function would basically just be a mapping between an input number, and an output number that is twice the value. But most of the time, we are more interested in the results, rather than whether the * or << operator is used. We can think of this as an *implementation detail*. Although you could think of implementation details like this as *behavior*, it is behavior that is insignificant if all we care about is the input and output of a function.

If we happened to use the second version of byTwo for some reason, we might find that it breaks when our `number` argument gets too large (try it with a trillion: `1000000000000 << 1`). Does this mean we suddenly care about this implementation detail?

No. We care that our output is broken. This means that our test suite needs to include more cases than we initially thought. And we can happily swap this implementation out for one that satisfies all of our test cases; whether that is `return number * 2` or `return number + number` is not our main concern.

We changed the implementation details, but doubling our number is the behavior we care about. What we care about is also what we test (either manually or in an automated way). Testing the specifics is not only unnecessary in many cases, but it also will result in a codebase that we can't refactor as freely.

TESTING JAVASCRIPT ITSELF

If you are testing extremely specific implementation details, you will at some point no longer be testing your program, but rather the environment itself. We'll go over testing in detail later, but for now, you can run this either in a node console or by saving it to a file and running **`node file_name.js`**. That said, it's not critical to do so at this point.

Let's say that you're testing something like this:

```
assert = require('assert');
assert(2 + 2 === 4);
```

By the way, that first line will produce some scary-looking output if you're in a node console. Don't worry about that, though; it's just showing what loaded. Also, the node console already has `assert`, so you can leave out the first line if you're not working from a *.js* file. All the second line reports is `undefined`, which might seem weird. That's normal. Try asserting something untrue like `assert(3 === 2)`, and you'll see an error.

If you make assertions like these, you're testing JavaScript itself: its numbers, `+` operator, and `===` operator. Similarly, and more frequently, you might find yourself testing libraries:

```
_ = require('underscore');
assert(.first([3, 2]) === 3);
```

This is testing to see if the underscore library behaves as expected. Testing low-level implementation details like this is useful if you're exploring a new library or an unfamiliar part of JavaScript, but it's generally enough to rely on the tests that the packages themselves have (any good library will have its own tests). There are two caveats to this, however. First, "sanity tests" are necessary to ensure that some function or library has been made available to your environment, although those tests need not stick around once you've stabilized your environment. Second, if you're writing tests to help yourself be confident in the code (more on this later), then testing a library's behavior is appropriate to demonstrate the code in action.

Why Don't We Care About Unspecified and Untested Behavior?

The degree to which we specify and test our code is literally the effort that demonstrates our care for its behavior. Not having a test, a manual procedure for execution, or at least a description of how it should work means that the code is basically unverifiable.

Let's assume that the following code has no supporting tests, documentation, or business processes that are described through it:

```
function doesThings(args, callback){
  doesOtherThings(args);
  doesOtherOtherThings(args, callback);
  return 5;
};
```

Do we care if the behavior changes? Actually, yes! This function could be holding a lot of things together. Just because we can't understand it doesn't make it less important. However, it does make it much more dangerous.

But in the context of refactoring, we don't care if this behavior changes yet, *because we won't be refactoring it*. When we have any code that lacks tests (or at least a documented way of executing it), we do *not* want to change this code. *We can't refactor it*, because we won't be able to verify that the behavior doesn't change. Later in the book, we'll cover creating "characterization tests" to deal with untested code.

This situation isn't confined to legacy code either. It's also impossible to refactor new, untested code without tests, whether they are automated or manual.

HOW CONVERSATIONS ABOUT REFACTORING SHOULD GO UNTIL TESTS ARE WRITTEN

"I refactored login to take email address and username."

"No, you didn't."

"I'm refactoring the code to _____"

"No, you aren't."

"Before we can add tests, we need to refactor."

"No."

"Refactoring th–"

"No."

"Refa–"

"No."

Why Don't We Care About Performance?

As far as refactoring goes, we don't *initially* care about performance. Like with our doubling function a couple of sections back, we care about our inputs delivering expected outputs. Most of the time, we can lean on the tools given to us and our first guess at an implementation will be *good enough*. The mantra here is to "write for humans first."

Good enough for a first implementation means that we're able to, in a reasonable amount of time, determine that the inputs yield expected outputs. If our implementation does not allow for that because it takes too long, then we need to change the implementation. But by that time, we should have tests in place to verify inputs and outputs. When we have tests in place, then we have enough confidence to refactor our code and change the implementation. If we don't have those tests in place, we are putting behavior we really care about (the inputs and outputs) at risk.

Although it is part of what is called *nonfunctional* testing, and generally not the focus of refactoring, we can prioritize performance characteristics (and other "nonfunctional" aspects of the code like usability) by making them falsifiable, just like the program's correctness. In other words, we can test for performance.

Performance is privileged among nonfunctional aspects of a codebase in that it is relatively easy to elevate to a similar standard of correctness. By "benchmarking" our code, we can create tests that fail when performance (e.g., of a function) is too slow, and pass when performance is acceptable. We do this by making the running of the function (or other process) itself the "input," and designating the time (or other resource) taken as the "output."

However, until the performance is under some verifiable testing structure of this format, it would not be considered "behavior" that we are concerned about changing or not changing. If we have functional tests in place, we can adjust our implementations freely until we decide on some standard of performance. At that point, our test suite grows to encompass performance characteristics.

So in the end, caring about performance (and other nonfunctional aspects) is a secondary concern until we decide to create expectations and tests around it.

JAVASCRIPT'S PARTICULAR DIFFICULTIES WITH PERFORMANCE

For some types of JavaScript (see **Chapter 2**), it will be completely impossible to either change (unsafe) or refactor (safe) your code without changing performance. Adding a few lines of frontend code that don't get minimized will increase the time for downloading and processing. The incredible amount of build tools, compilers, and implementations might perform any number of tricks because of how you structured your code.

These things might be important for your program, but just to be clear, refactoring's promise of "not changing behavior" must not apply to these situations. Subtle

and difficult-to-control performance implications should be treated as a separate concern.

What Is the Point of Refactoring if Behavior Doesn't Change?

The point is to *improve* quality while *preserving* behavior. This is not to say that fixing bugs in broken code and creating new features (writing new code) are not important. In fact, these two types of tasks are tied more closely to business objectives, and are likely to receive much more direct attention from project/product managers than concerns about the quality of the codebase. However, those actions are both about changing behavior and therefore are distinct from refactoring.

We now have two more items to address. First, why is quality important, in the context of "getting things done"? And second, what is quality, and how does refactoring contribute to it?

Balancing Quality and Getting Things Done

It may seem as though everyone and every project operates on a simple spectrum between quality and getting things done. On the one end, you have a "beautiful" codebase that doesn't do anything of value. And on the other hand, you have a codebase that tries to support many features, but is full of bugs and half-completed ideas.

A metaphor that has gained popularity in the last 20 years is that of *technical debt*. Describing things this way puts code into a pseudofinancial lingo that noncoders can understand easily, and facilitates a more nuanced conversation about how quickly tasks can and should be done.

The aforementioned spectrum of quality to speed is accurate to a degree. On small projects, visible and addressable technical debt may be acceptable. As a project grows, however, quality becomes increasingly important.

What Is Quality and How Does It Relate to Refactoring?

There have been countless efforts to determine what makes for quality code. Some are determined by collections of principles:

- SOLID: Single responsibility, open/closed, Liskov substitution, interface segregation, and dependency inversion
- DRY: Don't repeat yourself

- KISS: Keep it simple, stupid
- GRASP: General responsibility assignment software patterns
- YAGNI: Ya ain't gonna need it

There are metrics like code/test coverage, complexity, numbers of arguments, and length of a file. There are tools to monitor for syntax errors and style guide violations. Some languages go as far as to eliminate the possibility of certain styles of code being written.

There is no one grand metric for quality. For the purposes of this book, quality code is code that works properly and is able to be extended easily. Flowing from that, our tactical concerns are to write tests for code, and write code that is easily testable. Here I not so humbly introduce the EVAN principles of code quality:

- Extract functions and modules to simplify interfaces
- Verify code behavior through tests

- Avoid impure functions when possible
- Name variables and functions well

Feel free to make up your own "principles of software quality" with your own name.

HUMAN READABILITY AS QUALITY

"Human readability" is sometimes cited as the chief concern for quality, but this is a fairly intractable metric. Humans come with varying experiences with and exposure to concepts. New coders, or even seasoned ones in a new paradigm or codebase, can struggle with abstractions, properly applied or not.

One could conclude from that line of thinking that only the simplest abstractions can be a part of a high-quality codebase.

In practice, teams find a balance between avoiding using esoteric and confusing features, and making time to mentor junior members to understand well-applied and sensible abstractions.

In the context of refactoring, quality is the goal.

Because you solve such a wide range of problems in software and have so many tools at your disposal, your first guess is rarely optimal. To demand of yourself only to write the best solutions (and never revisit them) is completely impractical.

With refactoring, you write your best guess for the code and the test (although not in that order if you're doing test-driven development, or TDD; see **Chapter 4**). Then, your tests ensure that as you change the *details* of the code, the overall *behavior* (inputs and outputs of the test, aka the interface) remains the same. With the freedom that provides, you can change your code, approaching whatever version of quality (possibly including performance and other nonfunctional characteristics) and whatever forms of abstraction you see fit. Aside from the benefit of being able to improve your code gradually, one significant additional perk of practicing refactoring is that you will learn to be less wrong the first time around: not by insisting on it up front, but by having experience with transforming bad code into good code.

So we use refactoring to safely change code (but not behavior), in order to improve quality. You may rightfully be wondering what this looks like in action. This is what is covered in the rest of the book; however, we have a few chapters of background to get through before that promise can be delivered upon.

Chapter 2 provides background on JavaScript itself. Chapters **3** and **4** give a justification for tests, followed by an actionable approach for testing, derived from natural inclinations to write code confidently and iterate quickly, rather than a dogmatic insistence on testing simply being understood as something unquestionably good and proper. **Chapter 5** explores quality in depth, aided by function visualizations called *Trellus diagrams*, which you can learn more about at **trell.us**.

In Chapters **6** and **7**, we look at general refactoring techniques. Following that, we look at refactoring object-oriented code with hierarchies in **Chapter 8** and patterns in **Chapter 9**. Then we finish with asynchronous refactoring (**Chapter 10**), and refactoring through functional programming (**Chapter 11**).

Nailing down what exactly quality is can be tough in a language as broad as JavaScript, but with the range of skills covered in these chapters, you should be left with a ton of options.

Refactoring as Exploration

Although for most of this book we commit to refactoring as a process for improving code, it is not the only purpose. Refactoring also helps build confidence in coding generally, as well as familiarity with what you are working on.

Constraints have their place, and in many ways a lack of them is what makes JavaScript so difficult to learn and work with. But at the same time, reverence breeds unnecessary constraints. I saw Ben Folds perform once, and he ended the show by throwing his chair at the piano. Who would attack the traditionally revered (and expensive) piano? Someone in control. Someone more important than his tools.

You're more important than your code. Break it. Delete it all. Change everything you want. Folds had the money for a new piano. You have version control. What happens between you and your editor is no one else's business, and you're working in the cheapest, most flexible, and most durable medium of all time.

By all means, refactor your code to improve it when it suits you. My guess is that will happen frequently. But if you want to delete something you don't like, or just want to break it or tear it apart to see how it works, go for it. You will learn a lot by writing tests and moving in small steps, but that's not always the easiest or most fun and liberating path to exploration.

What Is and Isn't Refactoring

Before we leave off, let's once again distinguish between refactoring and other lookalike processes. Here is a list of things that are not refactoring. Instead, they create new code and features:

- Adding square root functionality to a calculator application
- Creating an app/program from scratch
- Rebuilding an existing app/program in a new framework
- Adding a new package to an application or program
- Addressing a user by first and last name instead of first name
- Localizing
- Optimizing performance
- Converting code to use a different interface (e.g., synchronous to asynchronous or callbacks to promises)

And the list goes on. For existing code, any changes made to the interface (aka behavior) *should* break tests. Otherwise, this indicates poor coverage. However, changes to the underlying details of implementation *should not* break tests.

Additionally, any code changes made without tests in place (or at least a commitment to manually test the code) cannot be guaranteed to preserve behavior, and therefore are not refactoring, just changing code.

"REFACTORING" VS. "REFACTORING"

Initially for this book, we considered designating "refactoring," as it is colloquially used to mean "changing code," by a lowercase *r*, and reserving the capitalized version for our more specific definition (confidently restructuring code in a way that preserves behavior). Because this is cumbersome and we *never* mean lowercase "refactoring" in the context of this book, we decided not to use this distinction. However, when you hear someone say "refactoring," it is worth pausing to consider whether they mean "Refactoring" or "refactoring" (i.e., restructuring or just changing code).

Wrapping Up

Hopefully, this chapter has helped to reveal what refactoring is, or at least provide some examples of what it is not.

If we could define and achieve quality code through refactoring in JavaScript in the abstract or by simply looking at inspiring source examples from other languages (notably Java), our JavaScript codebases would not suffer from the

"broken or new" dichotomy of today, where codebases are poorly maintained until they are rewritten using tool A or framework B: a costly and risky approach.

Frameworks can't save us from our quality issues. jQuery didn't save us, and neither will ESNext, Ramda, Sanctuary, Immutable.js, React, Elm, or whatever comes next. Reducing and organizing code is useful, but through the rest of this book, you will be developing a process to make improvements that don't involve a cycle of suffering with poor quality followed by investing an unknowable amount of time to rebuild it in the "Framework of the Month," followed by more suffering in that framework, followed by rebuilding, and so on.

Which JavaScript Are You Using? 2

This might seem like it has an easy answer. How varied can one language be? Well, in JavaScript's case, any of these can greatly impact your tooling and workflows:

- Versions and specifications
- Platforms and implementations
- Precompiled languages
- Frameworks
- Libraries
- What JavaScript do you need?
- What JavaScript are *we* using?

These can represent not only different ways of doing things, but also a significant time investment to decide upon, learn to proficiency, and eventually write fluently. Throughout this chapter, we will explore these complexities in order to uncover what JavaScript we *can* write, and throughout the rest of the book, we'll get more specific about what JavaScript we *should* write.

Some of the choices involved in what JavaScript to use will be imposed by the project, some by the framework, and some by your own personal tastes.

Developing coding *style* is one of the biggest challenges in any language. Because of the complexity and diversity of the JavaScript ecosystem, this can be especially challenging in JavaScript. To become a well-rounded coder, some people recommend learning a new programming language every year. But with JavaScript, you might not even be able to learn every dialect in your whole lifetime. For a lot of languages "learning the language" means being competent with core APIs, resources, and one or two popular extensions/libraries. Applying that same standard to JavaScript leaves many contexts unexplored.

WHICH FRAMEWORK SHOULD I USE?

This is a perennial question posed by JavaScript developers, and a language-specific form of probably the biggest question from new developers, "Which language should I learn?" The framework treadmill can give programmers a sense that they genuinely need to know everything. Job descriptions with inflated and even contradictory requirements don't help. When it comes to JavaScript, there are so many frameworks, platforms, and ultimately distinct types of code you might write that some form of this question comes up time and time again.

In the end, you can't possibly learn everything. If you have a job or target job that genuinely requires certain skills, spend your time on those first. If you don't have a particular job in mind, find friends and mentors at meetups and follow what they do. Or, if you're just concerned with learning something that's interesting to you, pick something that seems cool* and go as deep as you want, then move on, and at some point consider getting very adept with a handful of these technologies.

You can apply this same process (filtering by job requirement, what your friends are using, and what you think is cool) to languages, frameworks, testing libraries, or musical instruments. In all of them, go deep occasionally, and, if you'll pardon a bit of crass advice, keep an eye on where the money is.

*"Seems cool" might sound vague, and it is. For *me*, that means finding the most novel or mind-bending technology *to me*. I'm not likely to learn 14 variants of something that solve the same problem. For others, cool means new and hip. To other people it means popular or lucrative. If you don't know what "seems cool" means to you, solve that by taking shallow trips into a few possibilities rather than spending too much time wondering which to choose.

Versions and Specifications

If you want to know where JavaScript is, and where it's headed, you should follow what's going on with the **ECMAScript specification** (https://tc39.github.io/ecma262/). Although features of JavaScript can bubble up from libraries and specific implementations, if you are looking for the canonical source of features that are likely to stick around, watch the ECMAScript specification.

Specifically (as of this writing), the committee responsible for tracking and adopting new features into the spec is called TC39. Spec proposals go through a multistaged process for adoption. You can track proposals in all five stages (0–4) on **GitHub** (http://github.com/tc39/proposals).

STRICT MODE

Because there are a variety of implementations (mostly browsers) in web usage, and "breaking old websites" is generally seen as a bad thing, JavaScript features are unlikely to be deprecated so much as fall out of favor.

Unfortunately, JavaScript contains certain features that cause unpredictability, and others that hurt performance just by being made available.

There is a safer, faster, opt-in subset of JS that is available to developers scoping files or functions with `"use strict"`.

It is generally seen as good practice to use strict mode. Some frameworks and precompiled languages include it as part of the build/compilation process. Additionally, the bodies of class expressions and declarations include `"use strict"` by default.

That said, following the ECMAScript specs has two major downsides. First, the raw documentation can be intimidating in both size and emphasis. It is generally written for those creating browsers rather than applications or websites. In other words, for most of us mere mortals, it is overkill. Some exposure to it is useful, however, for those who either enjoy describing the features in a more accessible way through blog posts, or prefer not having to rely on said blog posts as their source of truth.

The second downside is that even when a spec proposal is finalized (stage 4), there is no guarantee that your target implementations (i.e., node or your target browsers) have made the functionality described available. The spec is far from hypothetical, however, as the specs are influenced by implementers (e.g., browser vendors), and in many cases a particular implementation may have a feature in place before it is even finalized in the spec.

If you are interested in features that a spec makes available but that are not yet supported by your chosen implementation, three words you'll want to know are *shims*, *polyfills*, and *transpilers*. Searching "how do I support <whatever feature> in <some platform>" (node, Firefox, Chrome, etc.), combined with these terms, will likely give you the answer you're looking for.

Platforms and Implementations

When node hit the scene, web developers experienced some combination of relief and enthusiasm about the prospect of writing the same language on both the backend and the frontend. Others lamented the fact that JavaScript was the language to have such a prominent role.

This promise has seen mixed results. The JavaScript ecosystem has flourished with the backend pushing the frontend to be treated more like real code (organizationally and paradigmatically), while the pure mass of frontend develop-

ers available ensured that the backend platforms would always attract fresh and curious contributors.

On the other hand, as of this writing, although attempts have been made, full-stack JavaScript frameworks (sometimes called "isomorphic" for running the same code in two places) have not been as popular among developers as dedicated frontend and backend frameworks have been. Whereas within Ruby's Rails and Python's Django there is a clear hub of framework activity, no "grand unifying framework" has emerged in the vibrant but volatile JavaScript landscape.

In the browser, JavaScript code naturally gravitates toward the `window` base object, interactions with the DOM, and other browser/device capabilities. On the server side of a web app, data management and processing requests are the fundamental concern. Even if the language on the frontend and backend happens to be "the same," the kinds of code written conform to the task at hand, such that they are unlikely to follow similar patterns of code organization.

While the ECMAScript spec determines what features are likely to be implemented and supported, you can only use what is supported by your implementation (or libraries you bring in). What version of node or what version of the browser you're relying on is where the spec meets reality.

For tracking what features are natively available in browsers, **caniuse.com** keeps an updated list of what is available with respect not only to JavaScript APIs, but also to HTML and CSS. For considering both frontend and backend implementations, you can find broader feature tracking in **Kangax's ECMAScript compatibility tables** (https://kangax.github.io/compat-table/es6/).

If you're specifically interested in what new ECMA/TC39 proposals are implemented on a given platform, you can filter that table to show **proposals that are not yet part of the current standard** (https://kangax.github.io/compat-table/esnext/).

Implementations of a programming language can be called *runtimes* or *installs* as well. Particular versions of JavaScript, especially when implementations exist in a browser, are sometimes referred to as *JavaScript engines*. As far as versions, in addition to having traditional versioning numbers, the relationship of the version with its release cycle makes it likely to see the words *build* or *release* used to describe where it is in the process. You could see terms like *nightly build*, *weekly build*, *stable release*, or *long-term support release*.

Experimental features (not from the ECMAScript spec), nonnormative features (not specified by the spec), and gaps in the spec vary from browser to browser and build to build. Some features that eventually ended up in a finalized spec existed in libraries or implementations for years prior. Other features inside of implementations wither and deprecate when they are either replaced or ignored by the ECMAScript spec and other implementers.

Some places that JavaScript is popping up don't fit neatly into the language of "platform" or "implementation." JavaScript can be used as the source language for applications running on mobile devices, desktop operating systems, and even microcontrollers.

Although these are all exciting avenues for the ecosystem, keeping track of which JavaScript features are available in each context is, unfortunately, not a trivial task.

Precompiled Languages

So far, we've seen that implementations, platforms, and each version of the ECMAScript spec all have their own concept of what JavaScript is. So which JavaScript should you write?

First, let's take the simplest case of "compiled" versus "source" JavaScript: a process called *minification*. A *minifier* will compress your code to reduce the size of the file, while leaving the same meaning. In a browser context, this means a smaller download for the website user, and consequently, a faster page load.

However, minification is far from the only use case for compiling JavaScript into other JavaScript. One particularly successful project, Babel.js, began with the purpose of allowing developers to make use of future features of JavaScript by taking as input source code that would potentially not yet work in the target implementation and then compiling it into older syntax with better adoption.

Other precompiled languages specifically target a feature set that may not have anything to do with the ECMAScript spec, but still feels JavaScript-inspired. Sweet.js allows developers to add new keywords and macros to JavaScript. React as a whole is out of scope for this section, but the language often used with it, JSX, is also a precompiled language that reads like a mixture of JavaScript and HTML. As evidence of Babel's expansion, both Sweet.js and JSX are compiled into JavaScript using it.

One interesting effect of the love/hate relationship that many developers have with JavaScript is that it has led to an explosion of libraries, some with a compilation step that defines them as precompiled languages. These aim to treat JavaScript as a "compilation target."

In a rage against curly braces and a (former) lack of classes, CoffeeScript gained popularity as a way to write (preferable to some) code that compiled into JavaScript. Many other precompilations take a similar approach: write code the way you want to (using either a new or a preexisting language), and it will be compiled into JavaScript. Although CoffeeScript has fallen out of favor, other precompiled languages are stepping up to fill in other perceived gaps (or just lack of consistency) in JavaScript.

It would be great to give an overview of all of the languages that compile into JavaScript, but there are 337 of these documented on the **CoffeeScript project's wiki** (https://github.com/jashkenas/coffeescript/wiki/List-of-languages-that-compile-to-JS) as of this writing. If you needed any more evidence of the importance of JavaScript platforms, the distrust of the language itself, or the complexity of JavaScript's ecosystem, this is a good number to have in mind.

Frameworks

Let's step back to the original question of the chapter: "which JavaScript are you using?"

With our current knowledge of specifications, platforms, implementations, and precompiled languages, we would be able to "choose a JavaScript" for a website for whatever browsers we wanted to target by using supported features. Or we could choose to build a program outside of the browser using node. We could even use a precompiled language to backfill or extend the code we want to write and avoid writing *actual* JavaScript altogether. But frameworks provide for another possibility.

Frameworks can unify platforms and implementations as well as extend the vocabulary of the JavaScript we're using. And if you feel, for some reason, that you don't have quite enough decisions to make in order to determine *which JavaScript* you're using, frameworks are here to provide you with more choices (please clap).

VANILLA.JS

A parodic JavaScript framework called Vanilla.js makes a case for using no framework ("vanilla" is sometimes used as a synonym for "plain" or "unadorned") at all. As standards improve and implementations coalesce around common features, the case seems to get stronger.

On the other hand, a lack of willingness to deprecate confusing, nonstandard, and duplicated functionality (looking at you, `prototype`, `__proto__`, and `Object.getPrototypeOf`) guarantees a fractured and sprawling feature set.

Perhaps something like `"use strict"` will allow for unification of implementations (and obviation of frameworks) in the future, but I wouldn't bet on it.

The jQuery, Ember, React, Angular, and similar frameworks are basically super-libraries. Many, such as Ember, handle code organization, packaging, distribution, and testing concerns. Some create new syntax for writing HTML, like Angular. React even contains its own precompiled language (the JSX mentioned earlier), which will not run without compilation.

jQuery's footprint is still keenly felt in many apps. Newcomers find the difference between JavaScript and jQuery to be significant enough in syntax and purpose that they still ask which they should learn. This is an evergreen question that any framework (frontend or backend) will face.

The line between frameworks and libraries is a little murky, but whenever you see this question about a library, that indicates (in addition to a bit of confusion on the part of the questioner) that you are dealing with a framework in that it does not resemble JavaScript enough to be recognizable to the beginner.

The term *framework* is incredibly overloaded. Some frameworks that deal specifically with simplifying and unifying browser interactions, like jQuery, use the term *JavaScript framework*, whereas you might see things like Ember referred to as *web frameworks* or *app frameworks*. App/web frameworks tend to come with their own build/compile step, an app server, a base app structure, and a test runner.

To confuse things further, virtually any library can attach the word *framework* to itself (e.g., "testing framework") and appear more important. On the other hand, Electron, which allows desktop OS apps to be built using HTML, CSS, and JavaScript, also uses the word *framework*, whereas in the taxonomy of this chapter, it is closer to a *platform* unto itself.

Libraries

Regardless of what they call themselves, libraries are generally distinguished from frameworks in that they tend to have a more specialized purpose and be smaller (or at least humbler). As of this writing, Underscore.js calls itself a "library that provides a whole mess of useful functional programming helpers."

Which JavaScript are you writing?

So far, you have the choice to target specific platforms and implementations. Additionally, you can decide to use "frameworks," which may simplify processes, unify implementations, enable `"use strict"`, and introduce a build/compile step that may include a precompiled language before the JavaScript is generated.

All that libraries tend to add to this is some combination of more features and possible deprecation of others (à la `"use strict"`).

What JavaScript Do You Need?

It's a tough question. Here are four things to try.

1. Follow the hype.
   Honestly, if you're unsure, following the hype is the best thing you can do. Choose popular frameworks. Choose popular libraries. Target the most popular platforms and implementations that make sense for your applications and programs. Having a big community means they will also likely have a decent amount of documentation and example code.
2. Try something obscure.
   After you're done exploring the most popular options, look for a framework that is unique and helps you think about things in a different way. Look for similarities and differences in the more popular version you tried.
3. Use every tool possible.
   See how bloated and complicated you can make your testing process by introducing every library you can.
4. Go minimalist.
   See how far you can get with Vanilla.js on a given implementation. After you gain some experience with tooling, it can be refreshing to start fresh, bringing in tools only when they justify themselves. This process is covered when we gradually introduce a testing framework in **Chapter 4**.

What JavaScript Are We Using?

With so many options for JavaScript, it might seem impossible to choose any given form for this book. Here's how we're handling that:

- No frameworks (except for a few mentions).
- No compilation/transpilation/minifying steps.
- Most code is runnable without a browser, using standard node core packages.
- A few libraries are brought in for testing (mocha, tape, testdouble, wish).
- Two more are used for functional programming (Ramda, Sanctuary).

As far as style goes, this book is meant to prepare you to adapt and improve upon codebases of varying styles and quality. We'll explore procedural programming, OOP, and functional programming.

You will see a lot of bad code *before* improvements are applied. But the code *after* might not be optimal, or in your preferred style (or even mine), either. We take safe, small steps, and in order to demonstrate a wide variety of techniques, we can't take every code snippet all the way to perfection from its first form.

You'll find the same situation in legacy codebases, and overall, the amount of bad code likely outnumbers the good, both in size and in variations. Through this book, we move *incrementally* to make things better. There is a temptation when looking at a legacy codebase to think, "This is garbage. We need to throw it all away and use this framework/style/whatever." But there you're probably describing a *rewrite*, which is an ambitious (read "expensive and risky") process. In the styles and changes presented in this book, sometimes we move from bad to okay, and other times from good to better. But through this breadth, we will be exploring a range of options for you to apply to your own work.

Wrapping Up

As we covered in this chapter, your options for how to use JavaScript are incredibly broad. This situation may change in the future, but I wouldn't bet too heavily on any *one true JavaScript*. The ecosystem is so varied that you can explore completely different ways of coding and still stay somewhat close to home. Although the fact that "knowing JavaScript" is something of a moving target has its frustrations, it also means that the outlook is great for finding new interests and work within JavaScript(s).

3 Testing

Let's start with what's wrong with testing.

"Writing tests takes too long. We're moving too fast for that."

"It's extra code to maintain."

"That's what QA is for."

"It doesn't catch enough errors."

"It doesn't catch the important errors."

"This is just a little script/simple change. It doesn't need a test."

"But the code works."

"The boss/client is paying for features, not tests."

"No one else on the team cares and will break the test suite and I'll be trying to be Commissioner Gordon in a Gotham gone mad."

ABOUT THAT LAST QUOTE...

This one is actually a lot harder. Here, testing isn't the problem. This dynamic suggests a small and/or inexperienced team with a lack of leadership. Changes to this outlook on testing and quality could evolve over time (slowly, as in one person at a time), or be mandated from above (unlikely without new leadership).

Unfortunately, if you're in a team full of "cowboys"—coders who just push code, ignoring quality and testing—the most likely outcomes are frustration and unpredictable breakages (and unpredictable hours).

"Leave this toxic team immediately" is not the only solution, as there may be other benefits and constraints in your situation. But projects that have some focus on quality typically offer more stability (less turnover), better compensation, and more learning opportunities.

Of all the possible strawman quotes listed, this one stands out as one where the fix is not simply to "recognize and enjoy the benefits of testing after becoming comfortable with it." But if the *engineering culture* actively discourages testing, it is actively discouraging quality. It's harder to change culture than your own personal

outlook and experience with testing. If you're not in a leadership role, your personal development is best served by avoiding or leaving these projects.

At first glance, "The boss/client is paying for features, not tests" may also look like a cultural problem. However, it's unlikely that this is actually enforced at a code level. If you're efficient at writing tests, only the shortest professional engagements would move faster without having a test suite in place. In these cases, it's best to use your judgment to write quality software. No reasonable boss or client would refuse *any* verification that the software works correctly. If you can automate the process efficiently, your professional standard of quality should include writing tests. If you can't do that efficiently due to your lack of skill with testing tools, you can't help but give in to lower standards until you can gain experience. First, recognize that when you do manual checks, you are testing; you're just not automating your tests. That should be sufficient motivation.

All this is to say that the solution to your internal resistance to testing is to get more comfortable with testing as you develop your standard of quality. External resistance, if strong enough, is resistance to quality and professional development. This book should help you overcome the internal resistance. The external resistance is all about networking and experience in choosing projects.

If you have these opinions, you're certainly not alone, and on a given project you might even be right. There are real difficulties with writing tests. One thing to keep in mind, though, is that when things are frustrating, beneficial as they might be, some coders may feel a sense of "cognitive dissonance," which can escalate a need to gather more evidence as to why testing is useless/costly/not your job. Here, you're acting like Aesop's fox who, after grasping for grapes and failing, consoles itself by saying that they must have been sour anyway. If ensuring software quality through testing is hard, then it must not be important, right?

That might be a helpful adaptation for dealing with regrets and disappointments in life that are outside of your control, but without recognizing the benefits of testing, you're closed off to a completely different way of writing code. The grapes aren't sour, and unlike the fox, you can find a ladder to reach them.

The main purpose of testing is to have *confidence* in your code. This confidence cannot be born in a vacuum. It's forged in the skepticism developed from seeing code that errs and resists change. As we'll see in **"Debugging and Regression Tests"**, confidence is the best indicator of what and how to test. The biggest reason to learn testing is to develop your senses of confidence and skepticism when looking at a codebase. If that sounds a bit abstract, don't worry. More concrete reasons for testing are coming up in the next section.

First, a quick note on some terms we'll be using:

Coverage (also code coverage or test coverage)
: This is a measurement, most usefully a percentage, of the lines of code that are covered by tests.

High-level and low-level
: Just like ordinary code, tests can be broad (high-level) or more involved in details (low-level). These are general terms, but for the two most important types of tests we'll cover, *high-level* will generally correspond to "end-to-end tests," whereas *low-level* will correspond to "unit tests."

Complexity
: This is a measurement of the pathways through the code. It tends to be more casually and generally referred to as *complexity*, rather than its ancestor *cyclomatic complexity*.

Confidence
: This is, ultimately, why we test. Full test coverage gives us confidence that the whole codebase behaves as we intended. There are some caveats to this, covered by **"Nonfunctional Testing"** and **"Test-driven development"**.

Exercised
: A line of code is said to be *exercised* if it is run by the test suite. If a line is exercised, then it has coverage.

Technical debt
: This is the situation where a lack of confidence and trust (via complexity and a lack of test coverage) in the codebase results in more guesswork and slower development overall.

Feedback loop
: This is the gap between writing code and knowing if it is correct. A "tight" or "small" feedback loop (versus a "loose" or "long" one) is good, because you know right away when your code is functioning as expected.

Mocking and stubbing
: These are both ways of avoiding directly exercising a function, by replacing it with a dummy version. The difference between the two is that mocking creates an *assertion* (pass/fail part of a test), whereas stubbing does not.

The Many Whys of Testing

1. You're already doing it!
   Well, this is a bit of an assumption, but if you run your code in the console or open up an HTML file in the browser to verify behavior, you are testing

already, albeit in a slow and error-prone way. Automated testing is just making that process repeatable. See **"Manual Testing"** for an example.

2. Refactoring is impossible without it.
   Refactoring, as discussed in **Chapter 1**, is completely impossible without guaranteeing behavior, which, in turn, is impossible without testing. We want to refactor and improve code quality, right?
3. It makes working with a team easier.
   If your coworker writes some broken code, the test suite should let you both know that there's a potential problem.
4. Tests are ideal for demonstrating (not documenting) functionality.
   Making and maintaining documentation is a whole different discussion. But assuming you don't have documentation in place, tests can demonstrate the behavior you want out of your code. Exclusively relying on tests to document the code (especially for an externally facing interface) is not a great idea, but it's a step up from having only the source code as a reference.
5. You're not just verifying the behavior of *your* code.
   Not every software library you use will have the same policy on updates and versioning. You could unintentionally upgrade to a bad or conflicting version, and without testing, you wouldn't realize it had broken your code. Not only that, but when you bring a new library in or use a new part of it, do you want to exclusively rely on the tests the library has in place for itself? What if you modify that library? Can you still trust just its developer's tests then?
6. Tests are crucial for big upgrades.
   You really want to start using the latest version of big framework/new runtime. How can you upgrade responsibly and quickly? Just quickly? YOLO. Deploy it. It's probably okay, right? Right? Probably not. Just responsibly? Manually go through every possible code path and verify that everything does what it is supposed to, constructing necessary data objects as you need them. For both speed and responsibility at the same time, you need a test suite. There is no other way.
7. You'll catch bugs early.
   The earlier that bugs are caught in the development cycle, the easier they are to fix. Before you write them is ideal. If the quality assurance (QA) or product department finds a bug, that means more people committing time to the fix. Once it hits customers, you're talking about another production cycle as well as potential loss of business or trust.
8. Tests help you smooth out development cycles and not "crunch."

Testing, refactoring, improving quality, and ultimately carrying a low amount of technical debt will help to prevent times when you need to move fast and "can't help but break things." That means long hours, delayed releases, and time away from whatever you enjoy outside of work.

9. 9. Your feedback loop will be tighter.
   When you develop without tests, you're increasing the amount of time between development and verifying that your code is working. With tests in place, your feedback loop can be reduced to a few seconds. Without them, you're stuck with either assuming the code works or manually testing. If that takes five minutes every time (and gets longer as the program's complexity grows), how often will you do it? Odds are, the tighter your feedback loop is, the more often you'll verify that your code works, and the more confident you can be in making further changes.

The Many Ways of Testing

In this section, we'll look at methods of testing. One important thing to note is that each testing method we are looking at has three stages: the setup, the assertion, and the teardown.

The taxonomy of tests varies by organization, industry, language, framework, and point in history. The categories here highlight the broader types that are interesting to us in refactoring, but this list is not exhaustive. For instance, there are many variations of manual testing, and what we are calling "end-to-end" tests could be called integration tests, system tests, functional tests, or something else depending on the context.

We are mostly interested in tests that aid us in refactoring—that is to say, tests that protect and help to improve the quality of the software itself, rather than the experience of using it.

We consider a codebase to have full coverage when every code path is exercised by either unit tests, end-to-end tests, or ideally, both. It might seem overly picky to insist that *every* line is covered, but generally the worse the code is, the worse the coverage is, and vice versa. Practically speaking, 100% coverage is very hard to accomplish for many reasons, especially when you're writing JavaScript and relying on external libraries. There are diminishing returns in confidence gained through more coverage as you approach 100% (or even "five nines": 99.999%).

Testing is a tool to produce confidence. It is not the only tool. The original author of a codebase or a problem domain expert can derive confidence from other sources. Tests (along with code simplicity and, where appropriate, comments) are special sources of confidence because they allow the confidence to be transmitted along with the code. In any case, they are not the goal. Confi-

dence is the goal. Tests are a practical way of creating specific types of confidence in the code that can be transferred to other team members or the future you.

For the sake of refactoring, end-to-end tests and unit tests are the most important of the types we'll cover in this chapter.

Manual Testing

This was hinted at earlier in the chapter, but instinctively, everyone wants to test their code. If you're working on a web app, that likely means just loading the page with the appropriate data objects in place and clicking around a bit. Maybe you throw a `console.log()` statement in somewhere to ensure variables have the expected values. Whether you call it "monkey testing" or "manual testing," "making sure it works" or "QAing," this testing strategy is useful for exploration and debugging.

If you're faced with an undertested codebase, it's a good way to experiment. This applies to the feature and debugging level of development as well. Sometimes, you just need to see what is happening. This is also a large component in *spiking*, the process of research that is sometimes needed before a "red/green/refactor" cycle can be entered.

Depending on which JavaScript you're using (see **Chapter 2**), build/compiler errors can also be caught during this step.

Documented Manual Testing

One step toward automation from manual testing is to develop a testing/QA plan. The best manual tests are either temporary or well documented. Although you want to move on to feature tests and unit tests as quickly as possible, sometimes the fastest way to get a section of code "covered" is by writing a detailed set of steps to execute the relevant code paths. Although not automated, having a list (similar to what a QA team would have) can ensure that you are exercising the code in all the ways you need to, and makes the process less error prone by not relying on your memory. Also, it gives other members of your team a chance to contribute to and execute the plan as a "checklist." This is handy to take some of the weight from a QA team, or fulfill that role if none exists.

If you find yourself lacking confidence in the code, with a big deploy looming and a dysfunctional or incomplete test suite, this is your best option. Documenting your code paths in text allows manual steps to be repeatable and distributed among team members.

Even if coverage is good, a QA department or developers filling that role may elect to manually run checks on a particular system if it is especially vital that it does not break (sometimes called a *smoke test*) or contains complexity that is resistant to automated testing.

Approval Tests

This method is a bit tricky, but may work for some projects and team configurations. Sometimes, the outcome of code execution would be difficult to automatically assert. For instance, let's say that you have an image processing feature on a website for automatically cropping and resizing an avatar. Setup is no problem: you simply feed it an image that you have on hand. But it could be difficult to write a simple assertion that proves the cropping tool worked well. Do you hardcode the bytes or pixels of the images for the desired input and output? That would make the tests very brittle, as they would break with every new image it crops. So do you skip testing altogether?

With an approval test, you automate the test setup and teardown, but the assertion (or "approval") is left to human input. This test suite keeps a record of what output you have approved. When the test runs a second time, the same output will "pass," and the other tests (that are new or have different results than the approved output) are added to the "unapproved" queue for human observation. When a member of the queue is approved, the new version of the output replaces or augments the memory of the old output (these can be files/database entries on the development machine or on a staging server). If everything is approved, then the code is assumed to be functioning correctly. If everything is not approved, the code gets the same treatment as it would for any failing test: it gets fixed, hopefully with a few regression tests that reproduce the specific conditions that caused the bug.

This process may work well when the output is something like an image, video, or audio file, where it's difficult to write an assertion based on programmatically inspecting it. HTML may seem like a good fit for this type of testing, but often, end-to-end tests or unit tests are more appropriate if assertions are things like testing for text or an element on a page. Because you can use an HTML parser, or even a regular expression parser (most of the time) for HTML/CSS, that output is better tested through unit tests or end-to-end tests.

Just as manual tests are an organic process for testing as an individual, in some ways, approval tests are a natural way to test in a group. In any situation where a QA department, product owner, or designer is in a position to approve something's output, an ad hoc approval test system is in place. Depending on the technical ability of the approver, he may be responsible for handling the setup programmatically, through a documented manual test, or by asking the developer for a demo.

The weaknesses of the ad hoc approval test system are that setup may be onerous for the approver, and that it may be difficult to remember what outputs were approved/rejected. This is not an insult toward the memory of anyone involved. Even if they are singularly focused on this process, if the team using the process changes, the departing members take all their knowledge of the old approvals with them.

But recognize that any attempt to framework-itize this process will produce something that looks different to an approver than what she might be used to. Keeping things simple for approvers probably means providing a list of URLs to be reviewed in the queue.

For the developer, there are nontrivial annoyances in setting up the queue, setting up each test, and providing some easy way for the approver to review.

Even though this process is popularly practiced by teams in an ad hoc way, there has been little innovation in the seemingly likely space of "approval test frameworks." The questions raised by a system like this suggest some reasons why:

- If development and approval are two distinct processes, where does the canonical list of approved/unapproved tests/checklists live?
- Who is tasked with the "extra work" of translating product or design requirements into these lists?
- Should the list of features be contained within the source code?
- Should approval test failures (after a manual check) be integrated tightly enough to fail a test build?
- If so, how can you avoid slowing down testing cycles?
- If not, what is the feedback mechanism when someone rejects an approval spec?
- Will there be a mismatch of expectations if developers see approval tests passing as the completion of a task?
- What interaction, if any, should an approval test framework have with an issue/feature/bug tracking system?

It can be difficult enough for a process to be adopted when it is confined to a development team, but in the case of an approval test framework, everyone involved has to understand and agree to the answers to the preceding questions. Perhaps some Software as a Service product would be able to manage these concerns, but the preferred interfaces across different teams are varied and productized decisions about what types of processes to change with a tool like this are unlikely to make everyone happy.

RELATED: ACCEPTANCE TESTING

There have been attempts to formalize "acceptance testing," which rigorously mandates creation of specifications that correspond with user stories. If you are interested in this type of testing, Cucumber.js would be a framework to investigate.

Although it seems appealing, and it specifies a lot of uncertainty about approval tests, getting an internal cross-functional team or client on board may be more challenging than expected.

In the worst (but fairly likely) case, developers end up writing another entire layer of testing (rather than the client or "product owner"), but the client/product owner still requires the same level of process flexibility that the tests are intended to guard against.

Confusingly, some frameworks that are described as "acceptance test" frameworks do not insist on "English-like" syntax and do not imply a complex process that starts with a nondeveloper writing the spec in said English-like syntax. Instead, they provide high-level APIs for events like clicking and logging in, but these are clearly code, and not obscured by a layer of language that is converted into code.

These high-level APIs are useful for end-to-end tests, but be wary of frameworks that try to automate the actual acceptance of tasks. Requirements *magically* turning into code, and code *magically* fulfilling requirements, tends to actually be the *magic* that is software engineering, which isn't really magic and probably involves humans talking to each other. Ta-da.

End-to-End Tests

Finally, the good stuff. These tests are meant to automate the actual interactions that a manual tester could perform with the interface provided to the end user. In the case of the web app, this means signing up, clicking on a button, viewing a web page, downloading a file, and so on.

For tests like these, code should be exercised in collaboration with other pieces of the codebase. Mocking and stubbing should be avoided except when absolutely necessary (usually involving the filesystem or remote web requests) so that these tests can cover the integration between different system components.

These tests are slow and simulate end users' experience. If you want to split test suites into two, one fast and one slow, an end-to-end suite and unit test (covered next) suite are what you want. These are also called *high-level* and *low-level* tests. As you'll recall from the beginning of this chapter, "high-level"

means to take a broader view and have your code be more concerned with an integration of parts, while "low-level" means being more focused on the details.

Unit Tests

Unit tests are fast and singularly focused on "units." What is a *unit*? In some languages, the answer would be "mostly classes and their functions." In JavaScript, we could mean files, modules, classes, functions, objects, or packages. Whatever method of abstraction forms a unit, though, the focus of unit tests is on the behavior of the inputs and outputs of functions for each unit.

If we pretended that JavaScript's ecosystem was simpler and just contained classes, this would mean testing the inputs and outputs inside that class, along with creating objects from the class.

"PRIVATE" FUNCTIONS

It's a bit of a simplification to say that JavaScript units, whatever those may be (classes? function scope? modules?), are split into "private" and "public" methods. In packages/modules, private methods would be the functions that are not exported. In classes and objects, you have explicit control over what is "private" in a sense, but that necessarily involves extra setup and/or awkward testing scenarios.

In any case, a popular recommendation is to test only public methods. This allows your public methods to focus on the interface of inputs and outputs, as the code for private methods will still be exercised when you run the public methods that make use of them. This leaves you with a bit of flexibility to change private methods' implementation details, without having to rewrite the tests (as mentioned earlier, tests that break when an interface hasn't changed can be called *brittle*).

This is a good guideline in general, with tests also following the mantra "code to an interface, not an implementation," but it doesn't always line up with a priority of "code confidence." If you can't be confident in a public method without testing its implementation, feel free to ignore this advice. We'll see an example of when this could happen in the next chapter when we deal with randomness.

In contrast to end-to-end tests, we're only concerned with the independent behavior of functions of our units. These are low-level tests, rather than high-level tests. That means at integration points between units, we should feel free to mock and stub more liberally than with end-to-end tests. This helps us to keep the focus on the details, and leave the end-to-end tests tasked with the integration points. Additionally, if you avoid loading your entire framework and

every package you use, as well as faking the calls to remote services and possibly the filesystem and database too, this test suite will stay fast, even as it grows.

WHERE FRAMEWORKS CAN LET YOU DOWN

Frameworks often come with their own patterns of testing. These may relate to the directory that code lives in, rather than whether it is a unit test or an end-to-end test. This is unfortunate for two reasons. First, it encourages *one* test suite, rather than a fast suite (run frequently) and a slow suite (run somewhat often).

Division by app directory can also encourage tests that are part unit test and part end-to-end test. If you're testing a signup in a web app, should you need to load the database at all? For an end-to-end test, the answer is probably yes. For a unit test, the answer is probably no. Having slow tests (end-to-end) and fast tests (unit) to differentiate this behavior would be ideal.

This division is eroded in part because colloquially, tests can be known as feature tests, model tests, services tests, functional tests, or something else, if the test directory mirrors the app directory. As an application grows, this could result in one big, slow test suite (with fast tests mixed in). Once you have one slow test suite, it is difficult to dig in and split it into *necessarily* slow tests and *probably* fast tests. Fortunately, your app's structure may provide hints as to what folders should be fast and what folders should be slow (based on whether the files inside them tend to do high-level or low-level operations). Following that division, additional work is likely needed to remove loading dependencies of the potentially fast unit tests. As they are built without performance in mind, they are likely to load too much of the app, the database, and external dependencies.

One advantage to the default organization that a framework might provide is that tests may be easier to reason about, and to write to begin with. Although at some point this approach may be undesirable, having one test suite with good coverage is better than two with slightly better architecture but worse coverage.

All this is to say, solve the immediate problem, but look out for future problems as well. If you have bad coverage, that takes priority. If your test suite is so slow that no one will run it, a problem that you're only likely to have as a *result* of many tests and good coverage, then focus your efforts on *that* problem.

Your test suite, just like any code, should be written to serve its primary purpose before performance tuning. Coverage and confidence come first, but if your suite gets bogged down to the point where it isn't run frequently, then you must address the performance. You can do so by renting test running architecture online, parallelizing tests, splitting your tests into slow and fast suites, and mocking/stubbing calls to systems that are particularly slow.

Nonfunctional Testing

In the context of refactoring, nonfunctional testing does not directly contribute to accomplishing our code quality goals. Nor does it contribute to confidence that the code works. Nonfunctional testing techniques include:

- Performance testing
- Usability testing
- Play testing
- Security testing
- Accessibility testing
- Localization testing

Results of these types of tests contribute to new feature creation and issues (more broadly than what might initially be considered bugs) to fix. Is your game fun? Can people use your program? What about expert users? Is the first-time experience interesting? What about people with visual impairments? Will lax security policies lead to a data breach or prevent collaborations with partner companies or clients?

Unit and end-to-end testing will lead to coverage, confidence, and the chance to refactor, but they will not address these questions directly.

TRY AUDIO CAPTCHA SOMETIME

Some audio captchas on websites are horribly difficult. But it's worse than that. The experience often starts with being presented with an image of someone in a wheelchair, which certainly doesn't apply to all people who are visually impaired. And the audio that follows is often disturbing and haunting as well as being difficult to parse. All in all, it's a terribly unwelcoming and frustrating process.

In a conference talk, accessibility expert Robert Christopherson conducted an experiment with attendees using an audio captcha. He played it twice, and had everyone write down what they thought they heard, then compare it to the person next to them. Finally, he asked how many people had written down the same thing as their neighbor—zero out of about a thousand.

Ignoring nonfunctional testing will necessarily lead to ignoring some people, which is somewhere between mean and illegal. Nonfunctional testing is very important, but just not the focus of this book.

Nonfunctional testing, whether for accessibility or otherwise, is critical to the heart of a project. Because this book is focused strictly on technical quality, we can't help but gloss over the many disciplines involved in nonfunctional testing, but we also can't leave the topic without a recommendation to spend

more time learning about these ideas. Not only that, but also consider that a team that is diverse, functionally, demographically, and in life experiences, can help to root out the most egregious problems quickly and make the nuances more clear.

All this said, these testing techniques tend to generate tasks (bugs/features/enhancements) that will need attention in addition to what the obvious product road map indicates. This means more code changes and potentially more complexity. You want to meet that complexity with confidence, just as you would with mainline product features. The confidence to be flexible, fix bugs, and add features comes from testing.

Other Test Types of Interest

We'll cover these in detail in the next chapter, but this classification concerns different ways that your tests may interact with the implementation code. All of these three types may be either high-level or low-level:

Feature tests
: These are the tests that you write for new features. It is helpful to write the tests first (use test-driven development, or TDD), as will be demonstrated in the next chapter.

Regression tests
: These tests are intended initially to reproduce a bug, after which changes are made to the implementation code in order to fix the bug. This guarantees coverage so that that bug does not pop up again.

Characterization tests
: You write these for untested code to add coverage. The process begins with observing the code through the test, and ends with unit or end-to-end tests that work just as if you had written a feature test. If you want to follow TDD, but the implementation code is already written (even if *you* just did it), you should consider either writing this type of test or temporarily rolling the implementation code back (or commenting it out) in order to write the test first. This is how you can ensure good coverage.

Tools and Processes

Hopefully at this point, you're totally on board with the "why" behind testing, and understand how unit and end-to-end tests form the core to being confident in your code.

There is one last problem we need to address in this chapter: *testing is hard*. We already wrote the code, and now we have to do extra work to write the code that uses it in a structured and comprehensive way? That's frustrating (and also backward if you're doing TDD, but we'll get to that in a few short pages).

The larger problem is that *quality is hard*. Fortunately, there's more than just advice on how to do this: there are processes and tools as well as a good half-century's worth of research on how to encourage quality software.

Not all of these tools and processes are great for every project and every team. You might even spot an occasional lone-wolf programmer who avoids process, tools, and sometimes testing altogether. She might be deeply creative and prolific, to the point where you see her and wonder if this is how all software should be developed.

In complex software, maintained by teams of coders of varying skill levels, responsible for real deadlines and budgets, this approach will result in technical debt and "siloed" information that is not well understood by the whole team. Quality processes and tools scale up with complexity of projects and teams, but there's really no one-size-fits-all situation.

Processes for Quality

CODING STANDARDS AND STYLE GUIDES

The simplest process to use in order to help with test coverage and quality is for a team to adopt certain standards. These typically live in a *style guide* and include specifics like "Use `const` and `let` instead of `var`" as well as more general guidelines like "All new code must be tested" and "All bug fixes must have a test that reproduces the bug." One document may cover all the systems and languages that a team uses, or they could be broken out into several documents (a CSS style guide, a frontend style guide, a testing style guide, etc.).

Developing such a guide collaboratively with team members, and revisiting it from time to time, can make it flexible enough to adopt to changing ideas of what "good" means for the team. Having this guide in place provides a good foundation for the other processes covered here. Also, in an ideal world, most of this style guide is also executable, meaning that you have the ability to check for a large class of style guide violations very easily.

The term "style guide" may also refer to a design document describing the look and feel of the site. Aspects of this may even be in an external "press kit"–type form with instructions ("Use this logo when talking about us," "Our name is spelled in ALL CAPS," etc.) These are distinct documents. Putting documents like these in the same place as the *coding* style guide is not recommended.

DEVELOPER HAPPINESS MEETING

Another lightweight quality-focused process worth considering is a weekly, short meeting sometimes referred to as a *developer happiness* (this name may not be well received by noncoders in the organization), *engineering quality*, or *technical debt* meeting. The purpose of the meeting is to recognize areas in the codebase that have quality problems worth addressing. If product managers tightly control the queue of upcoming tasks, then this process enables one of the developers to perpetually make the case to them for adding a given task to the queue. Alternatively, developers or product managers may allocate a weekly budget for these types of tasks.

However the tasks are given priority, this process allows for things like speeding up the test suite; adding test coverage to a legacy module; or refactoring tasks to be addressed in a way that doesn't surprise anyone, maintains a quality focus, and allows the worst technical debt to be surfaced, widely understood, and then paid off.

PAIR PROGRAMMING

Pairing, aka *pair programming*, is a simple idea, but implementing it in a team can be difficult. Basically, two programmers tackle a problem, side-by-side, with one person running the keyboard ("driving") while the other watches the screen and sometimes has an additional computer available for research and quick sanity checks without tying up the main development machine. Typically the "driver" role is passed back and forth, sometimes at regular, predetermined intervals.

Pairing can lead to higher quality because it helps catch small bugs, even just typos, right away. Additionally, questions about quality ("Is this too hacky?" or "Should we test this as well?") come up frequently. Openly discussing these details can lead to more robust code and higher test coverage. Another benefit is that information about the system is held by at least two people. Not only that, but it helps other knowledge, from algorithms to keyboard shortcuts, to spread through an organization.

Knowledge sharing, focus, high-quality code, and company—what's the downside? First, like with other quality-based initiatives, it can be difficult to justify the upfront expense. "Two programming 'resources' are working on one task. What a waste!" Second, this process demands total focus, and it can actually be quite exhausting for the programmers. For this reason, pairing is often seen coupled with use of the "pomodoro" technique, where focus is mandated in 25-minute increments, with 5-minute breaks in between. Third, this process can feel insulting and frustrating to programmers (especially high performers) who think their output is slowed by it. Often this happens when there is a large

gap in experience between the pairing partners. Although this difference in skill potentially benefits the team most by allowing the greatest amount of knowledge transfer, not every programmer wants to act as a mentor. If it's a priority to keep pair-averse individual contributors happy and on staff, a teamwide mandate might be too aggressive of an approach.

Experimenting with pairing and getting feedback is a good idea, but teams tend to pair either very often or very rarely. Without a mandate or at least a genuinely supportive attitude from the team and management, despite its benefits, a situation can develop where those who would pair are reluctant to do so for fear of being seen as ineffective on their own. Skepticism of the process by pair-averse team members or managers will feed this reluctance. Then "pairing" takes on a new meaning, where people "pair" when they're stuck. Thus a cycle forms where pairing is stigmatized. So while pairing isn't "all or nothing," it likely is "mostly or barely."

A NOTE ON "RESOURCES"

When people (usually project/product managers) refer to team members in the design, development, or any other department as "resources," as in "We need more programming *resources* on this project," or "I need a *resource* for my project," try giving them a quizzical look and saying "What? Oh…You mean *people*?"

There are two problems here. First, hopefully, your team consists of programmers with diverse skills and experiences, and thinking of them as interchangeable units is not a good start to ensuring a project is well executed. Second, defining people (especially to their faces) strictly by their function is somewhere between unprofessional and mechanistically dehumanizing.

Three (not mutually exclusive) variations on pairing worth noting are *TDD pairing*, *remote pairing*, and *promiscuous pairing*. With TDD pairing, the driving role tends to alternate with the first person writing a test, and the second person implementing the code to make the test pass. In remote pairing, people not in the same physical location share their screens along with open audio and video channels. In promiscuous pairing, as opposed to *dedicated pairing*, the pairs of people are rotated on a day-by-day, week-by-week, sprint-by-sprint, or project-by-project basis.

CODE REVIEW

Another technique that helps ensure quality by putting another set of eyes on the code is *code review*. Through this process, when code is "finished," it is put into a review phase where another programmer looks through it, checking for style guide violations, bugs, and comprehensive test coverage. This may also be combined with a QA pass from the developers (especially on teams without a QA department), where the reviewing programmer manually runs the code and compares it with the task description (or more formally, *acceptance criteria*).

While pair programming and code review will not directly help overcome a dislike of testing, they will provide opportunities to focus on testing and quality. Enough of these experiences should help provide a more nuanced perspective on testing than that reflected by the quotes used to start off this chapter. Both processes give opportunities to establish norms and reinforce culture.

TEST-DRIVEN DEVELOPMENT

If you have the basics of testing down, test-driven development (TDD) can produce quality code with good test coverage. The downside is that it is a significant departure from a process of testing after the implementation code is written. The upside is that TDD, through the red/green/refactor cycle, can provide guidance throughout development. Additionally, if TDD is followed strictly, implementation code is written *only* in order to get a passing test. This means that no code is written that does not have coverage, and therefore, no code is written that cannot be refactored.

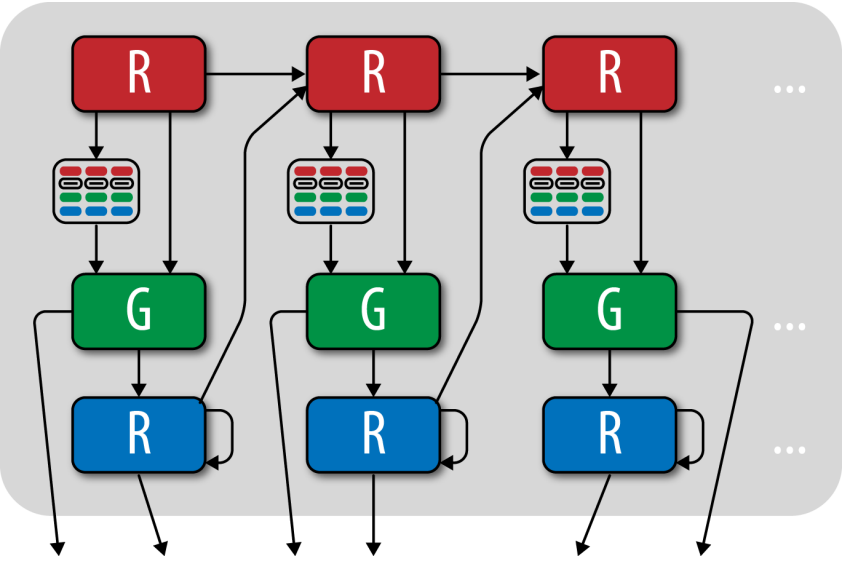
One challenge to a coder using TDD is that sometimes it is not obvious how to test, or even write the implementation code to begin with. Here, an exploratory phase known as *spiking* is suggested, where the implementation code is attempted before the tests are written. Strict TDD advocates will advise deleting or commenting out this *spike code* once the coder understands the task well enough to write tests and implement them, à la the normal red/green/refactor cycle.

A term that you'll often see alongside TDD is BDD (behavior-driven development). Basically, this process is extremely similar to TDD, but is done from the perspective of the end user. This implies two main points of importance. First, the tests tend to be high-level, end-to-end tests. Second, inside the red/green/refactor cycle of BDD end-to-end tests, there are smaller red/green/refactor cycles for TDD'd unit tests.

Let's take a look at a slightly complex diagram of a red/green/refactor cycle (**Figure 3-1**).

FIGURE 3-1

A red/green/refactor cycle



Whoa. Okay, so this might look a little crazy, but there are only a few complications here, and this should help you if you're wondering what part of the red/green/refactor cycle you are in.

Starting at the top left, we write a failing test, which puts us in a red state. We have three possibilities from there (one failing test). We could follow the arrow to the right (top-middle red state) and write another failing test (then we would have two). Or, if possible, we could follow the long arrow down to the green state (implement the test). The most complicated possibility exists when we can't implement the test to make it pass right away. In that case, we follow the short arrow down, where we meet another cycle just like this one. Assume we're stuck there for now, but we're always free to create new failing tests in any cycle that we've initiated.

If any tests (at any level) are in the green state in the middle, we can do one of two things: either refactor, or call it done and consider this test and code complete (leave the cycle). When we start refactoring, we can either keep refactoring or consider this test and code complete (leave the cycle).

When all the tests are complete, the arrows can all leave the cycle. If we're in an inner cycle, that means we move into the green phase of the outer cycle. If we're in the outer cycle, and we can't think of any more tests to write (no more reds to tack on to the top row), then we're done.

One subtlety in this process worth noting is that creating a new red test after a green test is recommended only after you have at least considered a refactoring phase. If you move on right away, there is no clear indication that more

work is to be done. By contrast, if you have just refactored something, then you're free to move on. Similarly, if you have written a failing test (red state), then if you write another test case, your test framework will still let you know that there is more work to be done on the first one, so you won't get lost there.

Moving on from the diagram, here is a concrete example of a TDD cycle containing another TDD cycle:

1. Failing (red) high-level test is written: "A customer can log in."
    1. Failing low-level test is written: "Route '/login' returns 200 response."
    2. Routing code is written to pass the low-level test. We can refactor this code written on the inner cycle now.
    3. The high-level test is still failing, so a new failing low-level test is written: "The form post route at '/login_post' should redirect to '/' for valid email and password."
    4. Code is written to handle successfully posting the email/password and returning the logged-in home page. This inner cycle test is passing, so we can refactor this code written on the inner cycle now.
2. Now both the second low-level test and the high-level test are passing.
3. Once everything in our outer cycle test is green, we're free to refactor the outer cycle as we see fit. And we have two levels of testing to ensure our code continues to behave.

For the purposes of refactoring, using a methodology like BDD doesn't really matter. What matters is that you have good coverage in your code, preferably from unit tests *and* end-to-end tests. If those were created in a "test first" manner via TDD or BDD, that's fine. But the aspect of testing from the end users' perspective through BDD is not critical. It's possible to create high-level tests with good coverage that don't test from this perspective and that aren't written before the implementation code.

QUALITY PROCESSES AS TEAM AND PERSONAL BRANDING

You will often see development methodologies and quality-enhancing techniques used as social proof. Job seekers add them to their résumés, hiring pages display them on job listings, and top consultancies build their brands around them.

Although day-to-day there may a focus on "getting things done" (potentially to the detriment of quality), being comfortable with these techniques and tools means access to better employment opportunities, be they with companies or clients.

Tools for Quality

Moving on to tools for testing...this could be a book unto itself. Though individual frameworks and tools rise and fall in popularity, knowing what types of tools are available will remain useful. Sometimes, GitHub or npmjs stars are a good proxy for quality and popularity. Searching for "js <tooltype>" (e.g., "js test coverage") with your preferred search engine usually returns decent results. It's best to learn one or two versions of each of these tools well, but with JavaScript's open source package development being as expansive as it is, be prepared to frequently adapt to new but similar tools.

VERSION CONTROL

Before any other tool is mentioned (in case you missed it in **Chapter 1**), it is critical that your project is under version control, preferably with backups somewhere other than just your computer. If lack of testing should produce skepticism rather than confidence, lack of version control should produce uneasiness, if not terror. Version control, with a backup online (as of this writing, Git + GitHub is recommended), ensures your code doesn't completely disappear. No amount of quality in the code matters if it can just get wiped out. Additionally, many of the tools covered here rely on versioned software for integration and to demonstrate progress.

TEST FRAMEWORKS

These vary significantly based on what JavaScript you are using (see **Chapter 2**), and how many tools you want bundled together. In general, these allow you to specify test cases in a file (or many) and have a command-line interface that you can use to run the test suite. A test framework may include many of the tools listed next as well. Some frameworks are specific to the frontend or backend code. Some are specific to the JavaScript framework. Some are specific to high-level, low-level, or acceptance tests. They will usually dictate how your test files are written as well as providing a test runner. The test runner will execute the suite (often allowing targeting of a specific directory, file, or individual test case), and output errors and failures of the run. It will also likely be responsible for the setup and teardown phases of your test run. Besides loading code, this can also mean putting the database in a particular state ("seeding" it) before the test run, and then resetting it afterward (as your tests may have created/deleted/changed records).

In the next chapter, we use some of the more basic features of the Mocha test framework. In **Chapter 9**, we'll also be trying out a more lightweight framework called Tape.

ASSERTION/EXPECTATION SYNTAX LIBRARIES

These are usually meant to work with particular testing frameworks and often come bundled with them. They are what enable you to assert that a given function returns "hello world" or that running a function with a given set of inputs results in an error.

Assertion libraries can be unnecessarily complicated. **Chapter 4** introduces a simple assertion and characterization testing library called wish.

DOMAIN-SPECIFIC LIBRARIES

Examples of these include database adapters and web drivers (to simulate clicks and other website interactions). These may be bundled into a JavaScript framework or testing framework. Often, they come with their own assertion/expectation syntax that extends the testing framework.

FACTORIES AND FIXTURES

These tools are responsible for creating database objects either on demand (factories) or from a file specifying data (fixtures). These libraries are sometimes associated with the word *fake* or *faker*. You'll find these useful if your tests require a lot of code to set up.

MOCKING/STUBBING LIBRARIES

These are sometimes associated with the terms *mocks/mocking*, *stubs/stubbing*, *doubles*, and *spies*. Mocks and stubs both allow you to avoid calling a certain function in your tests, while mocks also set an expectation (a test) that the function is called. General mocking/stubbing functionality is usually included as part of a testing framework, but there are also more specific mocking/stubbing libraries that are not. These may stub out whole classes of function calls, including calls to the filesystem, the database, or any external web requests.

BUILD/TASK/PACKAGING TOOLS

The JavaScript ecosystem has a ton of tools for gathering code together, transforming it, and running scripts (custom or library-defined). These can be dictated by a JavaScript framework, and you might have some that overlap in your project.

LOADERS AND WATCHERS

If you're running the test suite frequently, which is essential to having a tight feedback loop, loading your whole app/program before every run can slow you down significantly. A loader can speed up the process by keeping your app in memory. This is often paired with a watcher program that executes your test suite when you save a file. The feedback loop can be tightened even further when the watcher script intelligently only runs the tests relevant to the saved file.

TEST RUN PARALLELIZERS

Sometimes built into loaders/task runners or test frameworks, these tools make use of multiple cores on your machine, parallelizing the test run and speeding up the execution of the test suite. One caution here worth noting is that if your application is heavily dependent on side effects (including using a database), you may see more failures when tests produce state that conflicts with other tests.

CONTINUOUS INTEGRATION (CI) SERVICES

These are online services that run your test suite on demand or upon events like committing to the shared version control repository, sometimes restricted to particular branches. These often make use of parallelization (and offer overall performance) beyond what you could accomplish on your personal machine.

COVERAGE REPORTERS

In order to know if code is safe to refactor, it is essential to know that the code is sufficiently covered by tests. Whether or not you are able to determine coverage without actually running the test suite (and consequently exercising the code) is an interesting academic question of dynamic versus static analysis, but fortunately, coverage tools that determine coverage by running the test suite are abundant. These can be run locally, but are often paired with CI systems.

MUTATION TESTING

If you're interested in other possibilities using dynamic analysis, which coverage tools make use of, you might want to check out *mutation testing*.

This topic can run a bit deep, but basically a tool runs your test suite with a mutated codebase (most easily by changing test inputs, e.g., a string input where a boolean was expected) and *fails* when your tests *pass* in spite of the mutated code.

This can allow you to find cases where tests don't actually require the code to be as it is. That may mean that the code is not exercised, but it also could mean that the normal input is meaningless in the context of the test. For example, if a parameter to a function is used in a way that is so general that various mutated inputs would work just as well, it suggests that even if coverage shows a line of code as being run, the code is not sufficiently exercised. At best, you may just be relying on type coercion in JavaScript's case.

Two warnings apply here. First, mutation testing relies on multiple variations on your normal test suite, so it will tend to be slow. Second, it may break your code in unexpected ways and complicate the teardown phase of your test—for example, by doing anything from leaving your test database in an unexpected state to actually changing the code in your files. Check your code into version control and back up your database before you run something this aggressive and unpredictable.

STYLE CHECKERS, AKA LINTERS

Most quality tools don't require dynamic analysis. They can inspect files without executing the code to look for many types of errors or stylistic violations. These checks are sometimes run locally as discrete scripts or through online services after code is committed to a shared repository. For a tighter feedback loop, these checks can often be integrated in a programmer's IDE or editor. You can't do any better than getting notifications of mistakes immediately after you write them. Sometimes these are called *linters*. At best, these style checkers you have in place serve as an executable style guide.

DEBUGGERS/LOGGERS

Sometimes, during a spike (writing exploratory, temporary code without testing), a tricky test case, or a manual test, it may be unclear what values variables have, what functions are outputting, or even whether a certain section of code is running. In these cases, debuggers and loggers are your friends. At the point

in the file where the confusion lies, either in your test or your implementation code, you can add a logging statement or set a debugging *breakpoint*. Logging will sometimes give you the answer you need (e.g., "this function was reached" or "this file was loaded"), but debuggers offer the chance to stop execution of the code and inspect any variable or function call you want.

STAGING/QA SERVERS

It can be difficult to simulate production to the level needed on your local development machine. For this reason, servers (or instances/virtual machines) that are as similar as possible to production are often used with a database that has a sanitized but representative version of production data.

If these processes and tools seem overwhelming, don't worry. When you need them, they can be useful, and if you don't, you can usually avoid them. It's good to experiment with new tools like these on hobby projects, but if you go overboard, you'll end up spending more time configuring everything to work together than you will actually doing the project!

Wrapping Up

So now we have a good overview of what testing is, as well as why it's useful and completely essential to refactoring. In the next chapter, we'll cover how to test your codebase, regardless of what state it's in.

4 Testing in Action

In the last chapter, we looked at some common objections to testing, and explored the benefits that can hopefully overwhelm those objections. If they can't for your project, then there's a strong possibility that either your team doesn't have the strongest engineering culture (quite possibly for reasons out of their control), or you haven't quite turned the corner on being able to write tests fast enough to justify their benefits.

In this chapter, we're getting into the details of how to test in the following scenarios:

- New code from scratch
- New code from scratch with TDD
- New features
- Untested code
- Debugging and regression tests

At the risk of beating a dead horse, you can't refactor without tests. You can *change code*, but you need a way to guarantee that your code paths are working.

REFACTORING WITHOUT TESTS: A HISTORICAL NOTE

In the original 1992 work on refactoring, "Refactoring Object-Oriented Frameworks" by William Opdyke (and advised by Ralph Johnson of "Design Patterns" fame), the word *test* appears only 39 times in a 202-page paper. However, Opdyke is insistent on refactoring as a process that preserves behavior. The term *invariant* appears 125 times, and *precondition* comes up frequently as well.

In a nonacademic setting, 25 years after the original work, and in JavaScript, with its rich ecosystem of testing tools, the mechanisms of preserving behavior are best facilitated by the processes we discuss in this book, and especially this chapter.

However, if you're curious about the initial ideas that went into refactoring, in a context likely a bit removed from your day job, checking out the original work is highly recommended.

BEFORE MOVING ON!

We have some shopping to do. We need node, npm, and mocha.

- Get node (version 6.7.0) from **nodejs.org**. A later version will also probably work.
- Installing node should also install npm.
- When you have npm, install mocha with **`sudo npm -g install mocha`** (you may not need the `sudo`).

To make sure everything is working, try running the following commands:

```
node -v
npm -v
mocha -V
```

Yes, that last `V` is a capital `V`, but the others are lowercase. If you get back anything other than version numbers, try searching for "installing node/npm on <whatever your operating system is>" (e.g., "installing npm on windows").

By the way, we tend to not use the `-g` (global) flag for other packages. We use that flag for mocha because we want the tool available at the command line. The **npm docs** (http://docs.npmjs.com/getting-started/installing-npm-packages-globally) have additional details.

The full reference versions of libraries used in creating this book can be found in **Appendix A**.

New Code from Scratch

Say we get the following specification from our boss or client:

> Build a program that, given an array of 5 cards (each being a string like `'Q-H'` for queen of hearts) from a standard 52-card deck, prints the name of the hand (`'straight'`, `'flush'`, `'pair'`, etc.).

NEW FEATURES VERSUS NEW CODE FROM SCRATCH

In terms of testing, creating a new feature is *almost* identical to creating the program from scratch. The biggest difference is that for new *features*, you will likely be able to use some testing infrastructure that has already been decided on, whereas in a *greenfield* (from scratch*)* project, the testing will be a blank slate, and you will have some setting up to do.

In the following two sections, testing with a new codebase quickly becomes breaking things down into individual features to test, which should illustrate this similarity.

Note that on a new project, while you might prefer to set up testing infrastructure before creating any implementation code, in these sections testing infrastructure is introduced as complexities arise that justify its use. This is done primarily to serve those without much experience testing, even though it is recommended that basic test infrastructure is set up beforehand. That said, be careful not to go overboard with test tooling. Worrying about too many dependencies of any kind can be very frustrating and take time away from implementing actual features.

So how do we start? How about with a checkHand function? First, create a file and save it as *check-hand.js*. If you run it with **`node check-hand.js`** right now, nothing will happen. It's just a blank file.

BE CAREFUL WITH ORDERING OF FUNCTION EXPRESSIONS

In this chapter, we use the syntax:

```
var functionName = function(){

// rather than
function functionName(){
```

Either style is fine, but in the first one (function expressions) the order in which you define your functions matters. If you get this error:

```
TypeError: functionName is not a function
```

then you should rearrange your functions or use the second syntax. We'll discuss this (along with the idea of "hoisting") more in **Chapter 6**.

We can start by writing a function that is basically a large `case` statement:

```
var checkHand = function(hand){
  if (checkStraightFlush(hand)){
    return 'straight flush';
  }
  else if (checkFourOfKind(hand)){
    return 'four of a kind';
  }
  else if (checkFullHouse(hand)){
    return 'full house';
  }
  else if (checkFlush(hand)){
    return 'flush';
  }
  else if (checkStraight(hand)){
    return 'straight';
  }
  else if (checkThreeOfKind(hand)){
    return 'three of a kind';
  }
  else if (checkTwoPair(hand)){
    return 'two pair';
  }
  else if (checkPair(hand)){
    return 'pair';
  }
  else {
    return 'high card';
  }
};

console.log(checkHand(['2-H', '3-C', '4-D', '5-H', '2-C']));
```

The last line is typical of something we'd add to ensure things are still behaving. While we're working, we might adjust this statement, or add more. If we're honest with ourselves, this `console.log` is a test case, but it's very high-level, and the output isn't structured. So in a way, adding lines like this is like having tests, but just not great ones. Probably the strangest part about doing things this way is that once we have 8 or 10 print statements, it will be hard to keep track of what means what. That means we need some structure to the format of our output. So we add things like:

```
console.log('value of checkHand is ' +
            checkHand(['2-H', '3-C', '4-D', '5-H', '2-C']));
```

BREAKING UP LONG LINES OF CODE

Notice that we're sometimes breaking up lines so that they don't run off the edge of the page. We'll talk about the specifics of how to deal with long lines in later chapters, but for now just trust that these breaks are okay.

Once we start doing this, we're actually doing the job of the test runner part of a test framework: one with very few features, but tons of duplication and inconsistency. Natural as it might be, this is a sure path to guesswork and frustration with temporary fits of confidence.

LOOK FOR THESE SYMPTOMS

Do you find programming to be an emotional roller coaster, cycling between swearing, pumping your fist into the air saying "Yes!" and then some more swearing?

This is a sign your feedback loop is not tight enough. Getting too excited or disappointed about things working or not reflects a state of surprise. It is very hard to be surprised if you are working in small steps, testing frequently, and using version control to maintain a recent and good version of the code.

Is it boring to not let your code surprise you? Maybe. Will you save a lot of time by not having to guess and redo work? Definitely.

Instinctively, new coders will manually test like this as they write the code, and then delete all of the `console.log` statements afterward, effectively destroying any *test-like* coverage that they had. Because we know that tests are important, we will get our coverage back after we've written the tests, but isn't

it odd to write these tiny, weird tests, delete them, and then write better versions after?

CONSOLE.LOG ISN'T THE ONLY WAY TO GO

With node, you can use the debugger by running **`node debug` *`my-program-name.js`*** instead of **`node` *`my-program-name.js`***.

By default, it gives you an interface to step through your file line by line. If you're interested in a specific section of your code, you can set a breakpoint by adding a line like this:

```
debugger;
```

Now when you run **`node debug` *`my-program-name.js`***, you'll still be started at the first line, but typing **`c`** or **`cont`** (it means "continue," but "continue" isn't a valid command, as it's already a reserved word in JS) will take you to your breakpoint.

If you're unfamiliar with what the debugger can do, type **`help`** inside of the debugger for a list of commands.

So what's next? Well, you take off for a week, and your coworkers start implementing these `check` methods, manually testing with `console` along the way to "see if it's working."

And you come back to this:

```
// not just multiples
checkStraightFlush = function(){
  return false;
};
checkFullHouse = function(){
  return false;
};
checkFlush = function(){
  return false;
};
checkStraight = function(){
  return false;
};
checkStraightFlush = function(){
  return false;
};
checkTwoPair = function(){
```

```
  return false;
};

// just multiples
checkFourOfKind = function(){
  return false;
};
checkThreeOfKind = function(){
  return false;
};
checkPair = function(){
  return false;
};

// get just the values
var getValues = function(hand){
  console.log(hand);
  var values = [];
  for(var i=0;i<hand.length;i++){
    console.log(hand[i]);
    values.push(hand[i][0]);
  }
  console.log(values);
  return values;
};

var countDuplicates = function(values){
  console.log('values are: ' + values);
  var numberOfDuplicates = 0;
  var duplicatesOfThisCard;
  for(var i=0;i<values.length;i++){
    duplicatesOfThisCard = 0;
    console.log(numberOfDuplicates);
    console.log(duplicatesOfThisCard);
    if(values[i] == values[0]){
      duplicatesOfThisCard += 1;
    }
    if(values[i] == values[1]){
      duplicatesOfThisCard += 1;
    }
    if(values[i] == values[2]){
      duplicatesOfThisCard += 1;
    }
    if(values[i] == values[3]){
      duplicatesOfThisCard += 1;
    }
    if(values[i] == values[4]){
      duplicatesOfThisCard += 1;
    }
    if(duplicatesOfThisCard > numberOfDuplicates){
```

```
      numberOfDuplicates = duplicatesOfThisCard;
    }
  }
  return numberOfDuplicates;
};

var checkHand = function(hand){
  var values = getValues(hand);
  var number = countDuplicates(values);
  console.log(number);

  if (checkStraightFlush(hand)){
    return 'straight flush';
  }
  else if (number==4){
    return 'four of a kind';
  }
  else if (checkFullHouse(hand)){
    return 'full house';
  }
  else if (checkFlush(hand)){
    return 'flush';
  }
  else if (checkStraight(hand)){
    return 'straight';
  }
  else if (number==3){
    return 'three of a kind';
  }
  else if (checkTwoPair(hand)){
    return 'two pair';
  }
  else if (number==2){
    return 'pair';
  }
  else{
    return 'high card';
  }
};
// debugger;
console.log(checkHand(['2-H', '3-C', '4-D', '5-H', '2-C']));
console.log(checkHand(['3-H', '3-C', '3-D', '5-H', '2-H']));
```

Oh no! What happened? Well, first we decided to make sure pairs worked, so we made every other function return `false` so that only the pair condition would be triggered. Then we introduced a function to count the number of duplicates, which required getting the values of the cards. We reused this function for four and three of a kind, but it doesn't seem very elegant. Also, our `getVal-`

ues function is going to break with a value of `10`, because it will return a `1`, aka an ace. Note that the `[0]` in the following line takes the first character of the string:

```
values.push(hand[i][0]);
```

So we have one very rough implementation, one bug that we may or may not have noticed, half of the functions implemented, half of them replaced with inline variables, and a lot of `console.log` statements in place as sanity checks. There's no real consistency in what was logged—these statements were introduced at points of confusion. We also have commented out a place where we might have had a debugger earlier.

Where do we go from here? Fix the bug? Implement the other functions? Hopefully, the last three and a half chapters have convinced you that attempting to improve the `countDuplicates` function would not be *refactoring* at this point. It would be changing code, but it would not be a safe process we could be confident in.

If we're testing after rather than before, how long should we wait? Should we add tests now? Should we finish our attempt at implementation first?

MAKING SOUP VERSUS BAKING

I like making soup. Fairly early on, you can start to taste it as you go, adding more ingredients, and tasting them individually as well. Baking, on the other hand, is very frustrating. You have to wait 45 minutes before you can find out if it's any good.

TDD is like making soup. You can have a tight feedback loop with the ingredients (low-level tests) or in bites (high-level tests).

Some people say the bites are all that matters, or they find tasting as you go to be too much effort. Some people hate soup: making it or eating it.

Tough. In other words, get ready for some TDD later in this chapter.

Later, we'll deal with a legacy system that we have to basically trust, but at this point, we've just started this code. No one is relying on it yet, so we can feel free to throw away all the bad parts. What does that leave us with? Take a look:

```
console.log(checkHand(['2-H', '3-C', '4-D', '5-H', '2-C']));
console.log(checkHand(['3-H', '3-C', '3-D', '5-H', '2-H']));
```

Yes. Just the high-level test cases. Let's use these and start over, with one tiny transformation:

```
var assert = require('assert');
assert(checkHand(['2-H', '3-C', '4-D', '5-H', '2-C'])==='pair');
assert(checkHand(['3-H', '3-C', '3-D',
                  '5-H', '2-H'])==='three of a kind');
```

Instead of printing with `console.log`, let's use node's assert library to *assert* that `checkHand` returns the values that we want. Now when we run the file, if anything inside of the `assert` function throws an error or is false, we'll get an error—no print statements necessary. We just assert that running our function with that input returns the expected strings: `'pair'` and `'three of a kind'`.

We also added a line to the beginning. This simply makes the `assert` statement available from node's core libraries. There are more sophisticated testing framework choices, but this involves minimal setup.

Before we move on, let's look at a flow chart to introduce all of the possibilities of what code we need to write when (**Figure 4-1**).

Keep in mind that if you're not writing tests first, you're following the "untested code" path rather than the "new code" path.

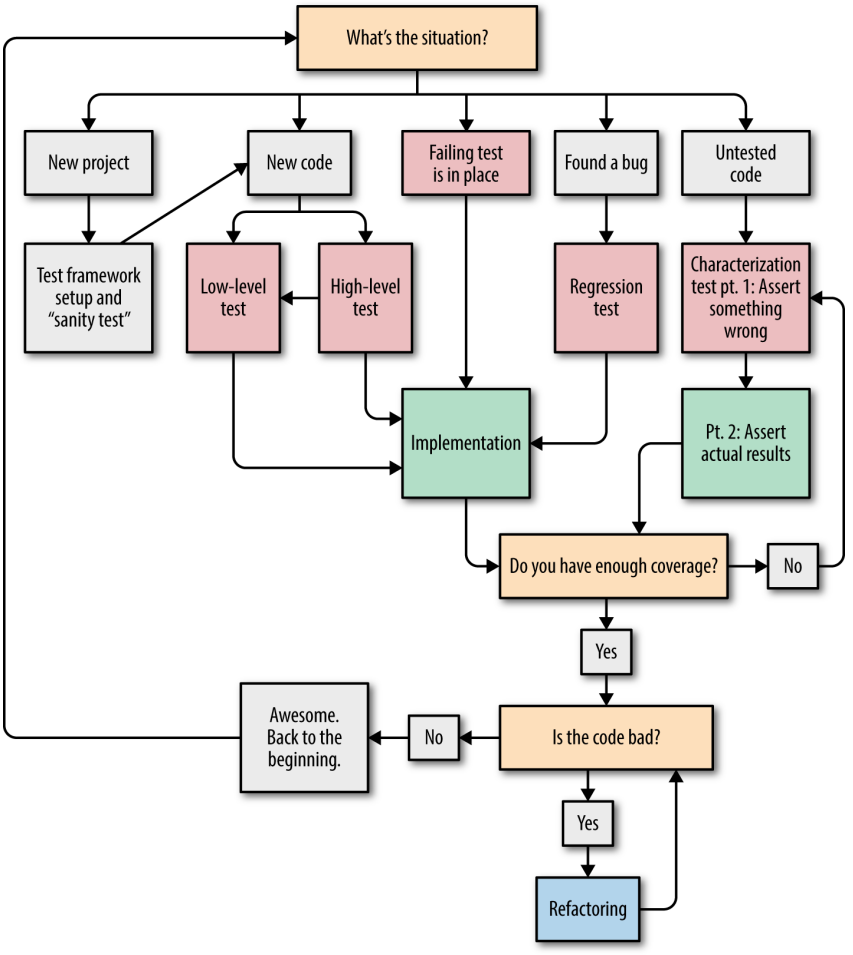


FIGURE 4-1

Flow chart to help you determine whether to write a test, refactor, or implement the code to make your test pass

New Code from Scratch with TDD

Of course, not all new code written without tests is going to end up in as awkward a state as that in the last section. It is certainly not reflective of an experienced coder's best efforts. However, it is reflective of a lot of coders' first effort. The wonderful thing about refactoring is that, given enough test coverage and

confidence, your first effort doesn't have to be your best effort. Get the tests passing, and you have the flexibility to make your code better afterward.

ABOUT TDD AND RED/GREEN/REFACTOR

This section jumps between lots of short code samples. This might seem tedious, but making tiny changes makes it much easier to discover and fix errors more quickly.

If you've never done TDD before, actually typing the samples out and running the tests in this section will give you a good idea of how the pacing works. Even if TDD isn't something that you rely on all the time, knowing how to use tests for immediate feedback is valuable.

Okay, back to our checkHand code. Let's just start with this:

```
var assert = require('assert');
assert(checkHand(['2-H', '3-C', '4-D', '5-H', '2-C'])==='pair');
assert(checkHand(['3-H', '3-C', '3-D',
                  '5-H', '2-H'])==='three of a kind');
```

We'll keep those lines (the tests) at the bottom and add our implementation to the top of the file as we go. Ordinarily, we would start with just one test case, so let's ignore the "three of a kind" assertion for now.

```
var assert = require('assert');
assert(checkHand(['2-H', '3-C', '4-D', '5-H', '2-C'])==='pair');
/* assert(checkHand(['3-H', '3-C', '3-D',
                    '5-H', '2-H'])==='three of a kind'); */
```

Now save that file as *check-hand.js* and run it with **`node check-hand.js`**. What happens?

```
/fs/check-hand.js:2
assert(checkHand(['2-H', '3-C', '4-D', '5-H', '2-C'])==='pair');
       ^

ReferenceError: checkHand is not defined
    at Object.<anonymous> (/fs/check-hand.js:2:8)
    at Module._compile (module.js:397:26)
    at Object.Module._extensions..js (module.js:404:10)
    at Module.load (module.js:343:32)
    at Function.Module._load (module.js:300:12)
    at Function.Module.runMain (module.js:429:10)
    at startup (node.js:139:18)
    at node.js:999:3

shell returned 1
```

Great! We got to the “red” of the red/green/refactor cycle, and we know exactly what to do next—add a `checkHand` function:

```
var checkHand = function(){ };
var assert = require('assert');
assert(checkHand(['2-H', '3-C', '4-D', '5-H', '2-C'])==='pair');
/* assert(checkHand(['3-H', '3-C', '3-D',
                    '5-H', '2-H'])==='three of a kind'); */
```

And we get a new error:

```
assert.js:89
  throw new assert.AssertionError({
  ^
AssertionError: false == true
    at Object.<anonymous> (/fs/check-hand.js:3:1)
...(more of the stack trace)
```

This assertion error is a little harder to understand. All this assertion knows about is whether what is inside evaluates to `true`, and it doesn’t.

`assert`s can take two parameters. The first is the assertion, and the second is a message:

```
var checkHand = function(){ };
var assert = require('assert');
assert(checkHand(['2-H', '3-C', '4-D', '5-H', '2-C'])==='pair',
                 'checkHand did not reveal a "pair"');
/* assert(checkHand(['3-H', '3-C', '3-D',
                    '5-H', '2-H'])==='three of a kind'); */
```

Now we get a new error message:

```
assert.js:89
  throw new assert.AssertionError({
  ^
AssertionError: checkHand did not reveal a "pair"
    at Object.<anonymous> (/fs/check-hand.js:3:1)
...(more stack trace)
```

That’s a little more clear, but if you find writing that second parameter tedious, there is a better option: use wish. First, install it through **`npm install wish`**.

Then our code becomes:

```
var checkHand = function(){ };
var wish = require('wish');
wish(checkHand(['2-H', '3-C', '4-D', '5-H', '2-C'])==='pair');
```

The error is clear without us specifying a message as a second parameter:

```
WishError:
    Expected "checkHand(['2-H', '3-C', '4-D', '5-H', '2-C'])"
    to be equal(===) to "'pair'".
```

There is an idea in TDD that in order to ensure that you're moving in small steps, you should write only enough code to make your tests pass:

```
var checkHand = function(){
  return 'pair';
};
var wish = require('wish');
wish(checkHand(['2-H', '3-C', '4-D', '5-H', '2-C'])==='pair');
/* wish(checkHand(['3-H', '3-C', '3-D',
                   '5-H', '2-H'])==='three of a kind'); */
```

When we run this, there are no failures. The file just runs and exits. If we used a test runner, it would give us a message like "All assertions passed!" or similar. We'll do that later, but in any case, now that we have that test "green," it's time to either refactor or write another test. There's no obvious refactoring here (we're mostly exploring testing, rather than refactoring, in this chapter), so we're on to our next test. Conveniently, we already have it written, so we can just uncomment the last line:

```
var checkHand = function(){
  return 'pair';
};
var wish = require('wish');
wish(checkHand(['2-H', '3-C', '4-D', '5-H', '2-C'])==='pair');
wish(checkHand(['3-H','3-C','3-D','5-H','2-H'])==='three of a kind');
```

Now we get a new failure that is readable (rather than the `false == true` error from `assert`):

```
WishError:
    Expected "checkHand(['3-H', '3-C', '3-D', '5-H', '2-H'])"
    to be equal(===) to "'three of a kind'".
```

We know it's talking about the three-of-a-kind line, so how do we fix it? If we are really, truly just writing the simplest code to make the test pass, we could write our function like this:

```
var checkHand = function(hand){
  if(hand[0]==='2-H' && hand[1]==='3-C'
     && hand[2]==='4-D' && hand[3]==='5-H'
     && hand[4]==='2-C'){
    return 'pair';
  }else{
    return 'three of a kind';
  }
};
```

This passes (no output when run), but this code reads with the same tone as a child taunting "Not touching! Can't get mad!" while hovering his hand above your face. While technically it passes, it is ready to break at the slightest change or expansion in test cases. Only this specific array will count as a `"pair"`, while any other hand will return `"three-of-a-kind"`. We would describe this code as *brittle*, rather than *robust*, for its inability to handle many test cases. We can also describe tests as brittle when they are so coupled to the implementation that any minor change will break them. While this is true of our `pair` assertion here as well, it is this implementation, not the test case, that is the problem.

We started with high-level tests, but we're about to go deeper. For that reason, things are about to get into a more complicated pattern than just red/green/refactor. We will have multiple levels of testing, and multiple failures at once, some of them persisting for a while. If we stick with our simple `asserts`, things will get confusing. Let's tool up and start testing with mocha. If you look through the **docs for mocha** (https://mochajs.org/), which you should at some point, you might be intimidated because it has a ton of features. Here, we're using the simplest setup possible.

As we discussed at the beginning of the chapter, make sure that you have node, npm, and mocha installed, and that they can all be run from the command line. Assuming that's sorted, let's create a new file called *check-hand-with-mocha.js* and fill it out with the following code:

```
var wish = require('wish');

function checkHand(hand) {
  if(hand[0]==='2-H' && hand[1]==='3-C'
     && hand[2]==='4-D' && hand[3]==='5-H'
     && hand[4]==='2-C'){
    return 'pair';
  }else{
    return 'three of a kind';
  }
};

describe('checkHand()', function() {
```

```
  it('handles pairs', function() {
    var result = checkHand(['2-H', '3-C', '4-D', '5-H', '2-C']);
    wish(result === 'pair');
  });
  it('handles three of a kind', function() {
    var result = checkHand(['3-H', '3-C', '3-D', '5-H', '2-H']);
    wish(result === 'three of a kind');
  });
});
```

ASSERTIONS AND EXPECTATIONS

Among the biggest wastes of coders' time, as represented by numerous Stack Overflow questions, Google searches, enormous docs, and blog posts explaining the docs, doing assertions "right" is really up there. Mocha provides numerous ways to assert. You can do this:

```
assert.equal(foo, 'bar');
```

Or this:

```
expect(foo).to.equal('bar');
```

Or this:

```
foo.should.equal('bar');
```

This is all nonsense. We have a perfectly good equality test in JavaScript itself; all you need is something to wrap the code and throw an error:

```
assert(foo === 'bar');
```

But that gives a nondescriptive error (`AssertionError: false == true`), which is why we're using wish:

```
wish(foo === 'bar');
```

which gives:

```
WishError:
    Expected "foo" to be equal(===) to " 'bar'".
```

What we're left with is basically the same thing, just in a form that mocha can use. The `describe` function indicates what function we are testing, and the `it` functions contain our assertions. The syntax for assertion has changed slightly, but these are the same tests we had before. And you can run them using **`mocha check-hand-with-mocha.js`** (make sure you're in the same directory as your file), which gives the output shown in **Figure 4-2**.

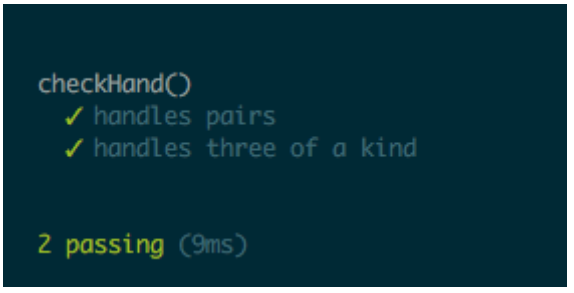


FIGURE 4-2

The output looks pretty good—nice and clear

QUICK TIP ABOUT MOCHA

If you have a file named *test.js* or put your test file(s) inside of a directory called *test*, then mocha will find your test file(s) without you specifying a name. If you set up your files like that, you can just run this: **`mocha`**.

Let's work on our checkHand function now. For proof that the checkHand pair checking is broken, add any other array that should count as a pair. Change the test to this:

```
describe('checkHand()', function() {
  it('handles pairs', function() {
    var result = checkHand(['2-H', '3-C', '4-D', '5-H', '2-C']);
    wish(result === 'pair');

    var anotherResult = checkHand(['3-H', '3-C',
                                   '4-D', '5-H', '2-C']);
    wish(anotherResult === 'pair');
  });
  it('handles three of a kind', function() {
    var result = checkHand(['3-H', '3-C', '3-D', '5-H', '2-H']);
    wish(result === 'three of a kind');
  });
});
```

Now run mocha again. There are three things to note here. First, we can have multiple assertions inside of one `it` block. Second, we get one failing test and one passing test. If any assertion in the `it` block fails, the whole `it` block fails. And third, as expected, we have a failure, a "red" state, and thus a *code-produced* impetus to change our implementation. Because there's no easy way out of this one, we're going to have to actually implement a function that checks for pairs. First, let's code the interface we want on this. Change the checkHand function to this:

```
function checkHand(hand) {
  if(isPair(hand)){
    return 'pair';
  }else{
    return 'three of a kind';
  }
};
```

And run mocha again. Two failures! Of course, because we didn't implement the `isPair` function yet, as is clearly explained by the errors mocha gives us:

```
ReferenceError: isPair is not defined
```

So, again doing just enough to shut the failures up, we write:

```
function isPair(){ };
```

And run mocha again...hey, wait a second. We sure are running mocha a lot. What about those *watchers* we talked about in the previous chapter? Turns out, mocha has one built in! Let's run this command:

```
mocha -w check-hand-with-mocha.js
```

Now whenever we save the file, we'll get a new report (hit Ctrl-C to exit). Okay, now back to the tests. It's not really clear how to write the `isPair` function. We know we'll get a hand and output a boolean, but what should happen in between? Let's write another test for `isPair` itself that takes a hand as input, and outputs a boolean. We can put it above our first `describe` block:

```
...
describe('isPair()', function() {
  it('finds a pair', function() {
    var result = isPair(['2-H', '3-C', '4-D', '5-H', '2-C']);
    wish(result);
  });
});
```

```
describe('checkHand()', function() {
...
```

Because we're using the watcher, we see this failure as soon as we save. We could return `true` from that function to pass this new test, but we know that will just make the three-of-a-kind tests fail, so let's actually implement this method. To check for pairs, we want to know how many duplicates are in the hand. What would `isPair` look like with our ideal interface? Maybe this:

```
function isPair(hand){
  return multiplesIn(hand) === 2;
};
```

Naturally, we'll get errors because `multiplesIn` is not defined. We want to define the method, but we can now imagine a test for it as well:

```
function multiplesIn(hand){};

describe('multiplesIn()', function() {
  it('finds a duplicate', function() {
    var result = multiplesIn(['2-H', '3-C', '4-D', '5-H', '2-C']);
    wish(result === 2);
  });
});
```

Another failure. What would our ideal implementation of `multiplesIn` look like? At this point, let's assume that we should have a `highestCount` function that takes the values of the cards:

```
function multiplesIn(hand){
  return highestCount(valuesFromHand(hand));
};
```

We'll get errors for `highestCount` and `valuesFromHand`. So let's give them empty implementations and tests that describe their ideal interfaces (the parameters we'd like to pass, and the results we'd like to get):

```
function highestCount(values){};
function valuesFromHand(hand){};

describe('valuesFromHand()', function() {
  it('returns just the values from a hand', function() {
    var result = valuesFromHand(['2-H', '3-C', '4-D', '5-H', '2-C']);
    wish(result === ['2', '3', '4', '5', '2']);
  });
```

```
});

describe('highestCount()', function() {
  it('returns count of the most common card from array',
    function() {
      var result = highestCount(['2', '4', '4', '4', '2']);
      wish(result === 3);
    }
  );
});
```

Implementing the `valuesFromHand` function seems simple, so let's do that:

```
function valuesFromHand(hand){
  return hand.map(function(card){
    return card.split('-')[0];
  })
}
```

Failure?!

```
Expected "result" to be equal(===) to " ['2', '3', '4', '5', '2']".
```

Certainly `split` is working as expected, right? What about a hardcoded version:

```
wish(['2', '3', '4', '5', '2'] === ['2', '3', '4', '5', '2']);
```

That gives us:

```
Expected "['2', '3', '4', '5', '2'] "
to be equal(===) to " ['2', '3', '4', '5', '2']".
```

Wait a second, aren't those arrays equal? Unfortunately, not according to JavaScript. Primitives like integers, booleans, and strings work fine with ===, but objects (and arrays are objects under the hood) will only work with === if we are testing variables that reference the same object:

```
x = []
y = []
x === y; // false
```

However:

```
x = [];
y = x;
x === y; // true
```

If you want to solve this with a more complicated assertion library, you can. Most have support for something like `assert.deepEqual`, which checks the contents of the objects. But we want to keep our assertions simple, and just have plain old JavaScript syntax inside of the `wish` assertions. Additionally, we might reasonably want to check equality of arrays or objects elsewhere in our program.

Rather than build `deepEqual` ourselves or bring in one that's tied to an assertion or testing framework, let's use a standalone library:

```
npm install deep-equal
```

Now in our test, we can do the following:

```
var deepEqual = require('deep-equal');
...
describe('valuesFromHand()', function() {
  it('returns just the values from a hand', function() {
    var result = valuesFromHand(['2-H', '3-C', '4-D', '5-H', '2-C']);
    wish(deepEqual(result, ['2', '3', '4', '5', '2']));
  });
});
```

Now it works. Awesome. Next up, let's implement `highestCount`:

```
function highestCount(values){
  var counts = {};
  values.forEach(function(value, index){
    counts[value]= 0;
    if(value == values[0]){
      counts[value] = counts[value] + 1;
    };
    if(value == values[1]){
      counts[value] = counts[value] + 1;
    };
    if(value == values[2]){
      counts[value] = counts[value] + 1;
    };
    if(value == values[3]){
      counts[value] = counts[value] + 1;
    };
    if(value == values[4]){
      counts[value] = counts[value] + 1;
    };
```

```
  });
  var totalCounts = Object.keys(counts).map(function(key){
    return counts[key];
  });
  return totalCounts.sort(function(a,b){return b-a})[0];
};
```

It's not pretty, but it passes the test. In fact, it passes all of them! Which means we're ready to implement something else.

You probably can see some potential refactoring in this function. That's great. It's not a very good implementation, but the first implementation of something often is not. That's the advantage of having tests. We can happily ignore this working, but ugly, function because it satisfies the right inputs and outputs. We could bring in a more sophisticated functional library like Ramda (**Chapter 11**) to handle array manipulation, or `Array`'s built-in `reduce` function (refactoring using `reduce` is covered in **Chapter 7**) to build our object, but for now `forEach` works fine.

Moving on, let's actually handle three of a kind:

```
function checkHand(hand) {
  if(isPair(hand)){
    return 'pair';
  }else if(isTriple(hand)){
    return 'three of a kind';
  }
};
```

We get an `UndefinedError` for `isTriple`. This time, though, we already have a known test case, so implementation is obvious:

```
function isTriple(hand){
  return multiplesIn(hand) === 3;
};
```

We don't have a test specifically for `isTriple`, but the high-level test should give us enough confidence to move on. For the same confidence on four of a kind, all we need is another high-level test (another `it` block in the `checkHand` test):

```
describe('checkHand()', function() {
...
  it('handles four of a kind', function() {
    var result = checkHand(['3-H', '3-C', '3-D', '3-S', '2-H']);
    wish(result === 'four of a kind');
  });
...
```

And the implementation:

```
function checkHand(hand) {
  if(isPair(hand)){
    return 'pair';
  }else if(isTriple(hand)){
    return 'three of a kind';
  }else if(isQuadruple(hand)){
    return 'four of a kind';
  }
};

function isQuadruple(hand){
  return multiplesIn(hand) === 4;
};
```

Next, let's write a test for the high card, which means another `it` block in the checkHand test:

```
  it('handles high card', function() {
    var result = checkHand(['2-H', '5-C', '9-D', '7-S', '3-H']);
    wish(result === 'high card');
  });
```

Failure. Red. And here's the implementation:

```
function checkHand(hand) {
  if(isPair(hand)){
    return 'pair';
  }else if(isTriple(hand)){
    return 'three of a kind';
  }else if(isQuadruple(hand)){
    return 'four of a kind';
  }else{
    return 'high card';
  }
}
```

Green. Passing. Let's take care of flush with another high-level test in the checkHand section:

```
it('handles flush', function() {
  var result = checkHand(['2-H', '5-H', '9-H', '7-H', '3-H']);
  wish(result === 'flush');
});
```

Failure. It doesn't meet any of the conditions, so our test reports as high card. The ideal interface is:

```
function checkHand(hand) {
  if(isPair(hand)){
    return 'pair';
  }else if(isTriple(hand)){
    return 'three of a kind';
  }else if(isQuadruple(hand)){
    return 'four of a kind';
  }else if(isFlush(hand)){
    return 'flush';
  }else{
    return 'high card';
  }
};
```

We get an `UndefinedError` for `isFlush`. So we want:

```
function isFlush(hand){ }
```

This returns `undefined`, so we get a failure because we're still going to hit the high card (`else`) path. We're going to need to check that the suits are all the same. Let's assume we need two functions for that, changing our `isFlush` implementation to the following:

```
function isFlush(hand){
  return allTheSameSuit(suitsFor(hand));
};
```

We get `UndefinedErrors` for those new functions. We could write the boilerplate, but the `allTheSameSuit` function seems pretty obvious to implement. Let's do that first. But since we have two functions that are new, we'll write a test so that we can be sure that `allTheSameSuit` is working as expected:

```
function allTheSameSuit(suits){
  suits.forEach(function(suit){
    if(suit !== suits[0]){
      return false;
    }
  })
  return true;
}

describe('allTheSameSuit()', function() {
```

```
  it('reports true if elements are the same', function() {
    var result = allTheSameSuit(['D', 'D', 'D', 'D', 'D']);
    wish(result);
  });
});
```

Passing. But naturally, we still have an undefined error for `suitsFor`. Here's the implementation:

```
function suitsFor(hand){
  return hand.map(function(card){
    return card.split('-')[1];
  })
};
```

It's pretty similar to our `valuesFromHand` function, so we're going to move on without a test. Feel free to write one if you want to stay in the test-first mode.

Uh-oh! Our flush condition seems to be returning `true` for our high card as well. Error:

```
  1) checkHand() handles high card:
     WishError:
    Expected "result" to be equal(===) to " 'high card'".
```

That must mean that `allTheSameSuit` is also returning `true`. We introduced a bug, so it's time for a regression test. First, we reproduce the behavior with a test. We didn't test that the `allTheSameSuit` function would actually return `false` when the cards aren't all the same. Let's add that test now:

```
describe('allTheSameSuit()', function() {
...
  it('reports false if elements are not the same', function() {
    var result = allTheSameSuit(['D', 'H', 'D', 'D', 'D']);
    wish(!result);
  });
});
```

Two failures, which means we reproduced the bug (and still have the original). Apparently, our `return false` was only returning from the loop. Let's change our implementation:

```
function allTheSameSuit(suits){
  var toReturn = true;
  suits.forEach(function(suit){
    if(suit !== suits[0]){
```

```
      toReturn = false;
    }
  });
  return toReturn;
};
```

And now all of our tests are passing again.

There's a better way to handle this `forEach`, by using Ramda (or Sanctuary, underscore, lodash, etc.) or digging into JavaScript's native `Array` functions a bit more (keeping in mind that using native functions requires us to check their availability on our target platform), but this was the easiest thing that passed the tests. We want our code to be readable, correct, good, and fast—in that order.

Only a few hands left. Let's do the straight. First a high-level test:

```
describe('checkHand()', function() {
...
  it('handles straight', function() {
    var result = checkHand(['1-H', '2-H', '3-H', '4-H', '5-D']);
    wish(result === 'straight');
  });
});
```

The code is hitting the `else` clause for high card. That's good. We know then that there aren't any competing conditions and are free to add one to `checkHand`:

```
function checkHand(hand) {
  if(isPair(hand)){
    return 'pair';
  }else if(isTriple(hand)){
    return 'three of a kind';
  }else if(isQuadruple(hand)){
    return 'four of a kind';
  }else if(isFlush(hand)){
    return 'flush';
  }else if(isStraight(hand)){
    return 'straight';
  }else{
    return 'high card';
  }
}
```

`isStraight` is not defined. Let's define it, and its ideal interface, in one step. We'll skip the test for `isStraight`, since it would be redundant with the high-level test:

```
function isStraight(hand){
  return cardsInSequence(valuesFromHand(hand));
};
```

Error. We need to define `cardsInSequence`. What should it look like?

```
function cardsInSequence(values){
  var sortedValues = values.sort();
  return fourAway(sortedValues) && noMultiples(values);
};
```

Two undefined functions. We'll add tests for both of these. First, let's get a passing test for `fourAway`:

```
function fourAway(values){
  return ((+values[values.length-1] - 4 - +values[0])===0);
};

describe('fourAway()', function() {
  it('reports true if first and last are 4 away', function() {
    var result = fourAway(['2', '6']);
    wish(result);
  });
});
```

Note that the + signs in line 2 are turning strings into numbers. If that seems hard to read, unidiomatic, or just likely to be changed without someone realizing the importance, use this line with `parseInt` instead:

```
return ((parseInt(values[values.length-1]) - 4 - parseInt(values[0]))===0);
```

Let's move on to `noMultiples`. We'll write a negative test case here, just for added assurance. The implementation turns out to be simple, though, because we already have something to count cards for us:

```
function noMultiples(values){
  return highestCount(values)==1;
};

describe('noMultiples()', function() {
  it('reports true when all elements are different', function() {
    var result = noMultiples(['2', '6']);
    wish(result);
  });
  it('reports false when two elements are the same', function() {
    var result = noMultiples(['2', '2']);
    wish(!result);
```

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

All tests are passing. Now for `StraightFlush`. Let's add this to our high-level checkHand `describe` block:

```
describe('checkHand()', function() {
...
  it('handles straight flush', function() {
    var result = checkHand(['1-H', '2-H', '3-H', '4-H', '5-H']);
    wish(result === 'straight flush');
  });
});
```

It seems to be hitting the flush condition, so we'll have to add this check above that in the `if`/`else` clause:

```
function checkHand(hand) {
  if(isPair(hand)){
    return 'pair';
  }else if(isTriple(hand)){
    return 'three of a kind';
  }else if(isQuadruple(hand)){
    return 'four of a kind';
  }else if(isStraightFlush(hand)){
    return 'straight flush';
  }else if(isFlush(hand)){
    return 'flush';
  }else if(isStraight(hand)){
    return 'straight';
  }else{
    return 'high card';
  }
}
```

`isStraightFlush` is not defined. Since this code is just the result of two functions, we won't worry about a low-level test for it (feel free to write one, though):

```
function isStraightFlush(hand){
  return isStraight(hand) && isFlush(hand);
}
```

It passes. Only two left: two pair and full house. Let's do full house first, starting with a high-level test:

```
describe('checkHand()', function() {
...
  it('handles full house', function() {
    var result = checkHand(['2-D', '2-H', '3-H', '3-D', '3-C']);
    wish(result === 'full house');
  });
});
```

The code is following the “three of a kind” branch of the checkHand conditional, so we want the `isFullHouse` check to go above that:

```
function checkHand(hand) {
  if(isPair(hand)){
    return 'pair';
  }else if(isFullHouse(hand)){
    return 'full house';
  }else if(isTriple(hand)){
    return 'three of a kind';
  }else if(isQuadruple(hand)){
    return 'four of a kind';
  }else if(isStraightFlush(hand)){
    return 'straight flush';
  }else if(isFlush(hand)){
    return 'flush';
  }else if(isStraight(hand)){
    return 'straight';
  }else{
    return 'high card';
  }
};
```

Now we need to implement the function `isFullHouse`. It looks like what we need is buried inside of `highestCount`. It just returns the top one, but we want them all. Basically, we need everything from that function except for the very last three characters. What kind of subtle, elegant thing should we do to avoid just duplicating the code?

```
function allCounts(values){
  var counts = {};
  values.forEach(function(value, index){
    counts[value]= 0;
    if(value == values[0]){
      counts[value] = counts[value] + 1;
    };
    if(value == values[1]){
      counts[value] = counts[value] + 1;
    };
    if(value == values[2]){
```

```
      counts[value] = counts[value] + 1;
    };
    if(value == values[3]){
      counts[value] = counts[value] + 1;
    };
    if(value == values[4]){
      counts[value] = counts[value] + 1;
    };
  });
  var totalCounts = Object.keys(counts).map(function(key){
    return counts[key];
  });
  return totalCounts.sort(function(a,b){return b-a});
};
```

Nothing! Don't try to be elegant. Duplicate the code. Copying and pasting is often the smallest and safest step you can take. Although copying and pasting gets a bad rap, it is absolutely a better *first step* than trying to extract functions (and other structures) and ending up breaking too many things at once (especially if you've strayed too far from your last `git commit`!). The real problem comes from *leaving* the duplication, which is a very serious maintenance concern—but the best time to deal with this is in the refactor step of the red/green/refactor cycle, not while you're trying to get tests to pass in the green phase.

THERE'S NOTHING WRONG WITH TERRIBLE CODE

As the previous paragraph should make clear, bad code is fine. Unused functions are fine. Bad variable names are fine. Duplicate code if it helps you get to the place where you can refactor more easily. Inline functions to extract more meaningful ones: it doesn't matter if that makes your function longer in the short term. Create a bunch of functions that you don't end up using while you're figuring out which one you need.

There's nothing wrong with writing terrible code. It's easier to start there, and fix it later. Just remember to fix it. What happens in your editor stays in your editor (until you replace it with better code).

Notice that we've left out the `[0]` because we want all of the results. Now all that's left is the `isFullHouse` implementation:

```
function isFullHouse(hand){
  var theCounts = allCounts(valuesFromHand(hand));
```

```
  return(theCounts[0]===3 && theCounts[1]===2);
};
```

It works. Great. Two pair, and we're done:

```
describe('checkHand()', function() {
...
  it('handles two pair', function() {
    var result = checkHand(['2-D', '2-H', '3-H', '3-D', '8-D']);
    wish(result === 'two pair');
  });
});
```

This is catching on the pair condition. That means the `isTwoPair` check will have to go before the `isPair` check in the conditional:

```
function checkHand(hand) {
  if(isTwoPair(hand)){
    return 'two pair';
  } else if(isPair(hand)){
...
```

And then an implementation that looks a great deal like `isFullHouse`:

```
function isTwoPair(hand){
  var theCounts = allCounts(valuesFromHand(hand));
  return(theCounts[0]===2 && theCounts[1]===2);
};
```

And we're done! That's how you start new code with tests, and maintain confidence throughout. The rest of the book is about refactoring. This chapter is about how to write lots and lots of tests. There is a ton of duplication in the code and the tests. The number of loops and conditionals is too high. There is barely any information hiding, and there are no attempts at private methods. No classes. No libraries that elegantly do loop-like work. It's all synchronous. And we definitely have some gaps when it comes to representing face cards.

But for the functionality it has, it is well tested, and in places that it's not, because we have a functioning test suite in place it's easy to add more. We even had a chance to try out writing regression tests for bugs.

Untested Code and Characterization Tests

So here's the scenario: the code for randomly generating a hand of cards was written by a coworker. He's taken two months off to prepare for Burning Man,

and the team is suspicious that he'll never *really* come back. You can't get in touch with him and he didn't write any tests.

Here, you have three options. First, you could rewrite the code from scratch. Especially for bigger projects, this is risky and could take a long time. Not recommended. Second, you could change the code as needed, without any tests (see **Chapter 1** for a description of the difference between "changing code" and "refactoring"). Also not recommended. Your third and best option is to add tests.

Here is your colleague's code (save it as *random-hand.js*):

```
var s = ['H', 'D', 'S', 'C'];
var v = ['1', '2', '3', '4', '5', '6', '7', '8', '9', '10', 'J', 'Q', 'K'];
var c = [];
var rS = function(){
  return s[Math.floor(Math.random()*(s.length))];
};
var rV = function(){
  return v[Math.floor(Math.random()*(v.length))];
};
var rC = function(){
 return rV() + '-' + rS();
};

var doIt = function(){
  c.push(rC());
  c.push(rC());
  c.push(rC());
  c.push(rC());
  c.push(rC());
};
doIt();
console.log(c);
```

Pretty cryptic. Sometimes when you see code that you don't understand, it's right next to data that's very important, and it seems the confusing code won't work without it. Those situations are tougher, but here, we can get this in a *test harness* (exercising the code through tests) fairly easily. We'll talk about variable names later, but for now, recognize that getting this under test does not depend on good variable names, or even understanding the code very well.

If we were using `assert`, we'd write a characterization test like this (you can put this test code at the bottom of the preceding code, and run it with **`mocha random-hand.js`**):

```
const assert = require('assert');
describe('doIt()', function() {
  it('returns nothing', function() {
```

```
    var result = doIt();
    assert.equal(result, null);
  });
});
```

That is, we'd just assume that the function returns nothing. And when we assume nothing from the code, normally it protests through the tests, "I beg your pardon, I'm actually returning *something* when I'm called with no arguments, thank you very much." This is called a *characterization test*. Sometimes (like in this case) the tests will pass, though, because they actually do return `null`, or whatever other value we provide as the second parameter to `assert.equal`. When we use `assert.equal` like this, the test passes, because `null == undefined`. If we instead use `assert(result === null);`, we're left with this gem of an error:

```
AssertionError: false == true
```

Neither one of those is helpful. Yes, we could get a better error with a different arbitrary value that JavaScript won't coerce to a value that happens to be equal in some way to the output of a function. For instance, we'd get a decent error like this:

```
assert.equal(result, 3);
AssertionError: undefined == 3
```

But personally, I like to avoid thinking about different types of equality and coercion as much as possible when writing characterization tests. So instead of using `assert` for characterization tests, we'll use wish's characterization test mode. We activate it by adding a second parameter of `true` to the call to wish, like this:

```
wish(whateverValueIAmChecking, true);
```

We can add the following code (replacing the `assert`-based test) to the bottom of the file and run it with **`mocha random-hand.js`**:

```
const wish = require('wish');
describe('doIt()', function() {
  it('returns something', function() {
    wish(doIt(), true);
  });
});
describe('rC()', function() {
  it('returns something', function() {
    wish(rC(), true);
```

```
  });
});
describe('rV()', function() {
  it('returns something', function() {
    wish(rV(), true);
  });
});
describe('rS()', function() {
  it('returns something', function() {
    wish(rS(), true);
  });
});
```

And the test errors tell us what we need to know:

```
WishCharacterization: doIt() evaluated to undefined
WishCharacterization: rC() evaluated to "3-C"
WishCharacterization: rV() evaluated to "7"
WishCharacterization: rS() evaluated to "H"
```

Our failures tell us what the code did, at least in terms of what type of return values we're dealing with. However, the `doIt` function returns `undefined`, which usually means there's a side effect in that code (unless the function actually does nothing at all, in which case it's dead code and we'd remove it).

SIDE EFFECTS

A side effect means something like printing, altering a variable, or changing a database value. In some circles, immutability is very cool and side effects are very uncool. In JavaScript, how side effect–friendly you are depends on what JavaScript you're writing (**Chapter 2**) as well as your personal style. Aiming to *minimize* side effects is generally in line with the goals of this book. *Eliminating* them is an altogether different thing. We'll talk about them throughout the book, but the deepest dive is in **Chapter 11**.

For the non-`null` returning functions, *you can just plug in input values and assert whatever is returned by the test*. That is the second part of a characterization test. If there aren't side effects, you never even have to look at the implementation! It's just inputs and outputs.

But there's a catch in this code. It's not deterministic! If we just plugged values into wish assertions, our tests would only work on rare occasions.

The reason is that randomness is involved in these functions. It's a little trickier, but we can just use a *regex* (regular expression) to cover the variations in outputs. We can delete our characterization tests, and replace them with the following:

```
describe('rC()', function() {
  it('returns match for card', function() {
    wish(rC().match(/\w{1,2}-[HDSC]/));
  });
});
describe('rV()', function() {
  it('returns match for card value', function() {
    wish(rV().match(/\w{1,2}/));
  });
});
describe('rS()', function() {
  it('returns match for suit', function() {
    wish(rS().match(/[HDSC]/));
  });
});
```

Because these three functions simply take input and output, we're done with these ones. What to do about our `undefined`-returning `doIt` function, though?

We have options here. If this is part of a multifile program, first, we need to make sure that the variable `c` isn't accessed anywhere else. It's hard to do with a one-letter variable name, but if we're sure it's confined to this file, we can just move the first line where `c` is defined and return it inside the `doIt` function like this:

```
var doIt = function(){
  var c = [];
  c.push(rC());
  c.push(rC());
  c.push(rC());
  c.push(rC());
  c.push(rC());
  return c;
};
console.log(c);
```

Now we've broken the `console` output. The `console.log(c)` statement no longer knows what `c` is. This is a simple fix. We just replace it with the function call:

```
console.log(doIt());
```

We still need a `doIt` test. Let's use a characterization test again:

```
describe('doIt()', function() {
  it('does something', function() {
    wish(doIt(), true);
```

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

This gives us back something like:

```
WishCharacterization: doIt() evaluated to
  ["7-S","8-S","9-H","4-D","J-H"]
```

Specifically, it returns what looks like the results of five calls to `rC`. For the test, we could use a regex to check every element of this array, but we already test `rC` like that. We probably don't want the brittleness of that kind of high-level test. If we decide to change the format of `rC`, we don't want two tests to become invalid. So what's a good high-level test here? Well, what's unique about the `doIt` function is that it returns five of something. Let's test that:

```
describe('doIt()', function() {
  it('returns something with a length of 5', function() {
    wish(doIt().length === 5);
  });
});
```

For smaller codebases that you want to get in a test harness or "under test," this process works well. For larger codebases, even with approval and enthusiasm from management and other developers, it is impractical to go from 0 (or even 50) percent coverage all the way to 100 percent in a short time via a process like this.

In those cases, you may identify some high-priority sections to get under test through this process. You'll want to target areas that are especially lacking in test coverage, core to the application, very low quality, or that have a high "churn rate" (files and sections that frequently change), or some combination of all of these.

Another adaptation you can make is to adopt a process that insists on certain quality standards or code coverage rates for all new code. See the previous chapter for some types of tools and processes that will help you with this. Over time, your high-quality and well-tested new (and changed) lines will begin to dwarf the older, more "legacy" sections of the codebase. This is what *paying off technical debt* looks like. It begins with getting code coverage, and then continues through refactoring. Keep in mind that for larger codebases, it takes months, not a weekend, of heroic effort from the team, and certainly not from one particularly enthusiastic developer.

Alongside process improvements and emphasizing quality in new changes to the code, one additional policy is worth considering. If the code is live and has people using it, even if it is poorly tested and low quality, it should be looked at with some skepticism, but not too critically. The implications are that

changing code that is not under test should be avoided, the programmers who wrote the legacy code should not be marginalized or insulted (they often understand the business logic better than newer programmers), and bugs should be handled on an ad hoc basis through regression tests (detailed in the next section). Also, since code in use *is* exercised (although unfortunately by the people using it rather than a test suite), it's possible that you can be *slightly* more confident in it than in new code.

To recap, if you find yourself with a large legacy codebase with poor coverage, identify small chunks to get under test (using the process from this section), adopt a policy of complete or majority coverage for new work (using the processes described in the sections on new features and new code from scratch —with or without TDD), and write regression tests for bugs that come up. Along the way, try not to insult the programmers who wrote the legacy code, whether they're still with the team or not.

Debugging and Regression Tests

After writing the tests from the last section, we've successfully created an automated way to confirm that the code works. We should be happy with getting the random hand generator under test, because now we can confidently refactor, add features, or fix bugs.

BEWARE OF THE URGE TO "JUST FIX THE CODE"!

Sometimes, a bug looks easy to fix. It's tempting to "just fix the code," but what stops the bug from coming back? If you write a regression test like we do here, you can squash it for good.

A related but distinct impulse is to fix code that "looks ugly or would probably cause bugs." The code could even be something that you identify as causing a bug. Unless you have a test suite in place, don't "just fix" this code either. This is changing code, not refactoring.

And that's a good thing. Here's the scenario: we've just encountered a bug in the code that made it into production. The bug report says that sometimes players get multiple versions of the same card.

The implications are dire. Certain hands, like four of a kind, are far more likely than expected (and *five* of a kind is even possible!). The competing online casinos have a perfectly working card dealing system, and people playing our game are losing trust in our faulty one.

So what's wrong with the code? Let's take a look:

```
var suits = ['H', 'D', 'S', 'C'];
var values = ['1', '2', '3', '4', '5', '6',
```

```
              '7', '8', '9', '10', 'J', 'Q', 'K'];
var randomSuit = function(){
  return suits[Math.floor(Math.random()*(suits.length))];
};
var randomValue = function(){
  return values[Math.floor(Math.random()*(values.length))];
};
var randomCard = function(){
 return randomValue() + '-' + randomSuit();
};

var randomHand = function(){
  var cards = [];
  cards.push(randomCard());
  cards.push(randomCard());
  cards.push(randomCard());
  cards.push(randomCard());
  cards.push(randomCard());
  return cards;
};
console.log(randomHand());
```

The first thing to notice is that the variable and function names have been expanded to make the code more clear. That also breaks all of the tests. Can you reproduce the test coverage from scratch without just renaming the variables? If you want to try it on your own first, go for it, but in either case, here are the tests:

```
var wish = require('wish');
var deepEqual = require('deep-equal');
describe('randomHand()', function() {
  it('returns 5 randomCards', function() {
    wish(randomHand().length === 5);
  });
});
describe('randomCard()', function() {
  it('returns nothing', function() {
    wish(randomCard().match(/\w{1,2}-[HDSC]/));
  });
});
describe('randomValue()', function() {
  it('returns nothing', function() {
    wish(randomValue().match(/\w{1,2}/));
  });
});
describe('randomSuit()', function() {
  it('returns nothing', function() {
    wish(randomSuit().match(/[HDSC]/));
```

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

First, we want to reproduce the problem somehow. Let's try running this code using just node (without the tests or mocha, so comment out the tests lines for now, but leave the `console.log(randomHand());`). Did you see the same card twice? Try running it again, and again if need be. How many times did it take to reproduce the bug?

With the manual testing (see **Chapter 3**) approach it can take a while, and be hard to see the error even when it does show up. Our next step is writing a test to exercise the code and attempt to produce the error. Note that we want a failing test before we write any code. Our first instinct might be to write a test like this:

```
describe('randomHand()', function() {
  ...
  for(var i=0; i<100; i++){
    it('should not have the first two cards be the same', function() {
      var result = randomHand();
      wish(result[0] !== result[1]);
    });
  };
});
```

This will produce failures fairly often, but isn't a great test for two reasons, both having to do with the randomness. First, by requiring many iterations of the code to be exercised, we've created a test that is sure to be fairly slow. Second, our test won't always reproduce the error and fail. We could increase the number of test runs to make not getting a failure virtually impossible, but that will necessarily slow our system down further.

THAT SLOW TEST

We want to keep it, and we want to run it, but if you put it on 100,000 iterations for fun, now is a good time to comment it out. And here we have a good case for where you would want a "slow test suite" in addition to your fast one.

As we talked about in the previous chapter, splitting up fast and slow tests is a nice thing to plan for, but every case is different. Here, we can just comment it out. If we had our suites split by files, we could run the one test file frequently and the other as often as necessary. Other times, you might want to isolate one particular test case. With mocha, you can use **`mocha -g pattern your_test_file`** to run test cases

where the string descriptions match the ***pattern***. See other mocha options by running **`mocha -h`** on the command line.

In spite of this not being a great test, we can use it as scaffolding to change our `randomHand` function. We don't want to change the format of the output, but our implementation is off. We can use as many iterations of the functions as we need to (almost) guarantee observing the failure in action.

Now that we have a harness (the currently failing test) in place, we can change the implementation of the function safely. Note that this is *not* refactoring. We are changing code (safely, because it is under test), but we are also changing *behavior*. We are moving the test from red to green, not refactoring.

Instead of pulling a random value and suit, let's return both at once from *one* array of values. We could manually build this array as having 52 elements, but since we already have our two arrays ready to go, let's use those. First, we'll test to make sure we get a full deck:

```
describe('buildCardArray()', function() {
  it('returns a full deck', function() {
    wish(buildCardArray().length === 52);
  });
});
```

This produces an error because we haven't defined `buildCardArray`:

```
var buildCardArray = function(){ };
```

This produces an error because `buildCardArray` doesn't return anything:

```
var buildCardArray = function(){
  return [];
};
```

This isn't really the simplest thing to get us past our error, but anything with a `length` would have worked to get us to a new failure (instead of an error), which in our case is that the length is 0 rather than 52. Here, maybe you think the simplest solution is to build the array of possible cards manually and return that. That's fine, but typos and editor macro mishaps might cause some issues. Let's just build the array with some simple loops:

```
var buildCardArray = function(){
  var tempArray = [];
  for(var i=0; i < values.length; i++){
```

```
    for(var j=0; j < suits.length; j++){
      tempArray.push(values[i]+'-'+suits[j])
    }
  };
  return tempArray;
};
```

The test is passing. But we're not really testing the behavior, are we? Where are we? Are we lost? How do we test if we're outside the red/green/refactor cycle? Well, what do we have? Basically, if we move in a big step like this, we create untested code. And just like before, we can use a new *characterization test* to tell us what happens when we run the function:

```
describe('buildCardArray()', function() {
  it('does something?', function() {
    wish(buildCardArray(), true);
  });
...
});
```

Depending on your mocha setup, you could get the full deck of cards back as an array, or it might be truncated. There are dozens of flags and reporters for mocha, so while it might be possible to change the output to what we want, in this case, it's just as fast to manually add a `console.log` to the test case:

```
it('does something?', function() {
  console.log(buildCardArray());
  wish(buildCardArray(), true);
});
```

So now we have the full array printed in the test runner. Depending on what output you got, you might have a bit of reformatting to do (now is a good time to learn your editor's "join lines" function if you don't already know it). In the end, we get:

```
[ '1-H', '1-D', '1-S', '1-C', '2-H', '2-D', '2-S', '2-C',
  '3-H', '3-D', '3-S', '3-C', '4-H', '4-D', '4-S', '4-C',
  '5-H', '5-D', '5-S', '5-C', '6-H', '6-D', '6-S', '6-C',
  '7-H', '7-D', '7-S', '7-C', '8-H', '8-D', '8-S', '8-C',
  '9-H', '9-D', '9-S', '9-C', '10-H', '10-D', '10-S', '10-C',
  'J-H', 'J-D', 'J-S', 'J-C', 'Q-H', 'Q-D', 'Q-S', 'Q-C',
  'K-H', 'K-D', 'K-S', 'K-C' ]
```

Great. If this array contained thousands of elements we'd need another way to derive confidence, but since it only has 52, by visual inspection we can affirm that the array looks good. We are playing with a full deck. Let's change our char-

acterization test so that it asserts against the output. Here, we actually got our confidence in the result from visual inspection. This characterization test is so that we have coverage, and to make sure we don't break anything later:

```
it('gives a card array', function() {
  wish(deepEqual(buildCardArray(), [ '1-H', '1-D', '1-S', '1-C',
    '2-H', '2-D', '2-S', '2-C',
    '3-H', '3-D', '3-S', '3-C', '4-H', '4-D', '4-S', '4-C',
    '5-H', '5-D', '5-S', '5-C', '6-H', '6-D', '6-S', '6-C',
    '7-H', '7-D', '7-S', '7-C', '8-H', '8-D', '8-S', '8-C',
    '9-H', '9-D', '9-S', '9-C', '10-H', '10-D', '10-S', '10-C',
    'J-H', 'J-D', 'J-S', 'J-C', 'Q-H', 'Q-D', 'Q-S', 'Q-C',
    'K-H', 'K-D', 'K-S', 'K-C' ]));
});
```

Passing. Good. Okay, so now we have a function that returns a full deck of cards so that our `randomHand` function can stop returning duplicates. If we uncomment our slow and occasionally failing "should not have the first two cards be the same" test, we'll see that it's still failing. That makes sense, as we haven't actually changed anything about the `randomHand` function yet. Let's have it return a random element from our array:

```
var randomHand = function(){
  var cards = [];
  var deckSize = 52;
  cards.push(buildCardArray()[Math.floor(Math.random() * deckSize)]);
  cards.push(buildCardArray()[Math.floor(Math.random() * deckSize)]);
  cards.push(buildCardArray()[Math.floor(Math.random() * deckSize)]);
  cards.push(buildCardArray()[Math.floor(Math.random() * deckSize)]);
  cards.push(buildCardArray()[Math.floor(Math.random() * deckSize)]);
  return cards;
};
```

We should still see our failure for the random/slow test (given enough iterations), and this is a great moment. What if we didn't have that test in place? Perhaps even without the test, it's obvious in this instance that we didn't fix the problem, but this isn't always the case. Without a test like this one, we could very well think that we had fixed the problem, only to see the same bug come up again later. By the way, are we changing the *behavior*? Not really, since we're still returning five cards as strings, so we would say that we changed the *implementation*. As evidence to that, we have the same passing and failing tests.

So how do we fix it?

```
var randomHand = function(){
  var cards = [];
  var cardArray = buildCardArray();
```

```
  cards.push(cardArray.splice(Math.floor(
    Math.random()*cardArray.length), 1)[0]);
  cards.push(cardArray.splice(Math.floor(
    Math.random()*cardArray.length), 1)[0]);
  cards.push(cardArray.splice(Math.floor(
    Math.random()*cardArray.length), 1)[0]);
  cards.push(cardArray.splice(Math.floor(
    Math.random()*cardArray.length), 1)[0]);
  cards.push(cardArray.splice(Math.floor(
    Math.random()*cardArray.length), 1)[0]);
  return cards;
};
```

Instead of just returning a card at a random index, we're only using the function once to build the array. Then, we use the `splice` function to, starting at a random index, return one element (the `1` is the second parameter of the `slice` function) and push it onto the array. For better or worse, `splice` *returns* an element, but also has the *side effect* of removing it from the array. That's perfect for our situation here, but the terms *destructive* and *impure* both apply to this function (see **Chapter 11** for more details). Note that we need the `[0]` because `splice` returns an array. Although it only has one element in it, it's still an array, so we just need to grab the first element.

Back to a recurring question: are we refactoring yet? Nope. We've changed the behavior (moving from the "red" to the "green" part of the cycle). As testament to that, our test appears to be passing now, even with many iterations.

HOW MANY ITERATIONS DOES IT TAKE TO TRIGGER THE FAILURE?

If you're good at math, feel free to ignore this, but if you're curious and feel like you might be in this situation again, knowing this math could be handy. We're testing that the first two cards are the same. What are the odds of that happening? How many iterations of the test should we use?

We actually need to invert the test, and calculate the odds of the cards *not being identical* first. It's a 51/52 or about 98.077% chance. So with one iteration, our odds are 100% – 98.077%, or about 1.923%. Not a very good chance of hitting it.

With 100 iterations, we have 98.077 times itself 100 times ((98.077)^100). That gives us 14.344%, and 100% minus 14.344% is 85.666%. So 100 iterations is enough to make our failure likely (>85% of the time), but a little less than one out of every seven times, our test will not fail.

Back to confidence, 10,000 iterations give us a 3.688×10^−7 chance of an errant passing test. Is that close enough to zero for "confidence"?

So are we confident that our change worked? The trouble is, we're still testing something random, which means we're stuck with an inconsistent and possibly slow test.

If you search for "testing randomness" online, many of the solutions will suggest things that make the random function more predictable. In our case, though, it's the implementation itself that we should still be skeptical about. How can we get rid of the slow scaffolding test and still have confidence that our code works? We need to test the implementation of a function that doesn't depend on randomness. Here's one way:

```
describe('spliceCard()', function() {
  it('returns two things', function() {
    wish(spliceCard(buildCardArray()).length === 2);
  });
  it('returns the selected card', function() {
    wish(spliceCard(buildCardArray())[0].match(/\w{1,2}-[HDSC]/));
  });
  it('returns an array with one card gone', function() {
    wish(spliceCard(buildCardArray())[1].length ===
      buildCardArray().length - 1);
  });
});
```

In this approach, we decide that to isolate the `spliceCard` function, we have to return its return value as well as its side effect:

```
var spliceCard = function(cardArray){
  var takeAway = cardArray.splice(
                Math.floor(Math.random()*cardArray.length), 1)[0];
  return [takeAway, cardArray];
};
```

Not bad. Our tests, including the slow test, still pass. But we still need to hook in the `randomHand` function. Here's what a first attempt could look like:

```
var randomHand = function(){
  var cards = [];
  var cardArray = buildCardArray();
  var result = spliceCard(cardArray);
  cards[0] = result[0];
  cardarray = result[1];
```

```
    result = spliceCard(cardArray);
    cards[1] = result[0];
    cardarray = result[1];
    result = spliceCard(cardArray);
    cards[2] = result[0];
    cardarray = result[1];
    result = spliceCard(cardArray);
    cards[3] = result[0];
    cardarray = result[1];
    result = spliceCard(cardArray);
    cards[4] = result[0];
    cardarray = result[1];
    return cards;
  };
```

Are we refactoring yet? Yes. We extracted a function without changing the behavior (our tests behave the same). Our scaffolding test will not fail no matter how many times we run it.

We have three considerations left. First, is this inner function that we've tested useful by itself, or only in this context? If it's useful outside of the context of dealing a hand of five (say, for blackjack?), leaving it in the same scope as `dealHand` makes sense. If it's only useful for the poker game, we might want to attempt to make it "private" (to the extent this is possible in JavaScript; see **"Context Part 2: Privacy"** in **Chapter 5**), which leads to a potentially unexpected conundrum: should you test private functions?

Many say no because behavior should be tested by the outer function, and testing an implementation rather than an interface is a path that leads to brittle and unnecessary tests. However, if you take this advice too far, what happens? Do you still need unit tests at all? Maybe you just need extremely high-level tests aligned with corporate objectives? (How much money do people spend playing poker in our application? How many new people signed up today?) Maybe a survey that asks people if they had fun using the application?

For this code, adhering strictly to the idea of "testing an interface, not an implementation" does not give us the confidence we need. Can we be confident in this code as a whole if we do not test the inner function? Not really, unless we leave our scaffold in place. Our second concern, getting rid of this slow test while retaining confidence, should be solved by our new test.

Third, are we done? As we covered in the TDD discussion, the red/green/refactor cycle is an appropriate process for improving code quality. We got to green, and we even refactored by extracting a method. Although this refactoring had a dual purpose in increasing confidence to delete a slow test, we used all three of those steps.

One last bit of cleanup we can do is removing the dead code. Specifically, in addition to getting rid of the slow test, or isolating it in another file, we can re-

move the functions that are no longer called—`randomSuit`, `randomValue`, and `randomCard`—as well as their tests. See more about identifying dead code in **"Dead Code"**.

But if we do that, are we done? It depends. Another iteration of the red/green/refactor cycle is appropriate if we can think of more features to implement (tests that would fail). We're happy with how our code works, so another iteration of the cycle doesn't make sense. We're also happy with our test coverage, so we needn't go through the process for untested code covered earlier.

So we're done? In many contexts, yes. But because this book is about refactoring, it's worth proposing an augmentation to the red/green/refactor cycle (see **Figure 4-1** earlier in this chapter for a flow chart of the interaction between testing and refactoring. You can also find this flow chart at the beginning of **Chapter 5**).

We've already refactored once by removing the dead code, but let's refactor our `randomHand` function one more time, taking advantage of destructuring. It sounds intimidating, but we're just setting multiple values at once:

```
var randomHand = function(){
  var cards = [];
  var cardArray = buildCardArray();
  [cards[0], cardArray] = spliceCard(cardArray);
  [cards[1], cardArray] = spliceCard(cardArray);
  [cards[2], cardArray] = spliceCard(cardArray);
  [cards[3], cardArray] = spliceCard(cardArray);
  [cards[4], cardArray] = spliceCard(cardArray);
  return cards;
};
```

And the tests still pass.

Wrapping Up

As you're refactoring, you might be tempted to change the interface of a function (not just the implementation). If you find yourself at that point, you are writing new code that requires new tests. Refactoring shouldn't require new tests for code that is already covered and passing, although there are cases where more tests can increase confidence.

To recap, use the red/green/refactor cycle for regressions. Write tests for confidence. Write characterization tests for untested code. Write regression tests for bugs. You can refactor as much as is practical, but only if you have enough coverage to be reasonably confident in the changes. In any case, keep the steps between your `git commit` commands small, so that you're ready to roll back to a clean version easily.

5 Basic Refactoring Goals

In **Chapter 1**, we discussed refactoring as a process of changing code safely and without changing behavior in order to improve quality. In **Chapter 2** we looked at the complexity of the JavaScript ecosystem and the consequent difficulty in pinning down what style and quality mean for us. In Chapters **3** and **4**, we laid the groundwork for testing, which is the easiest way to have confidence in our code, and a prerequisite to changing code safely (aka refactoring).

In this chapter, we finally turn to the specific relationship between refactoring and quality. The previous chapter included a diagram (**Figure 4-1**, reproduced as **Figure 5-1**) that described the relationship between testing and refactoring. Consider the main three questions in the diagram:

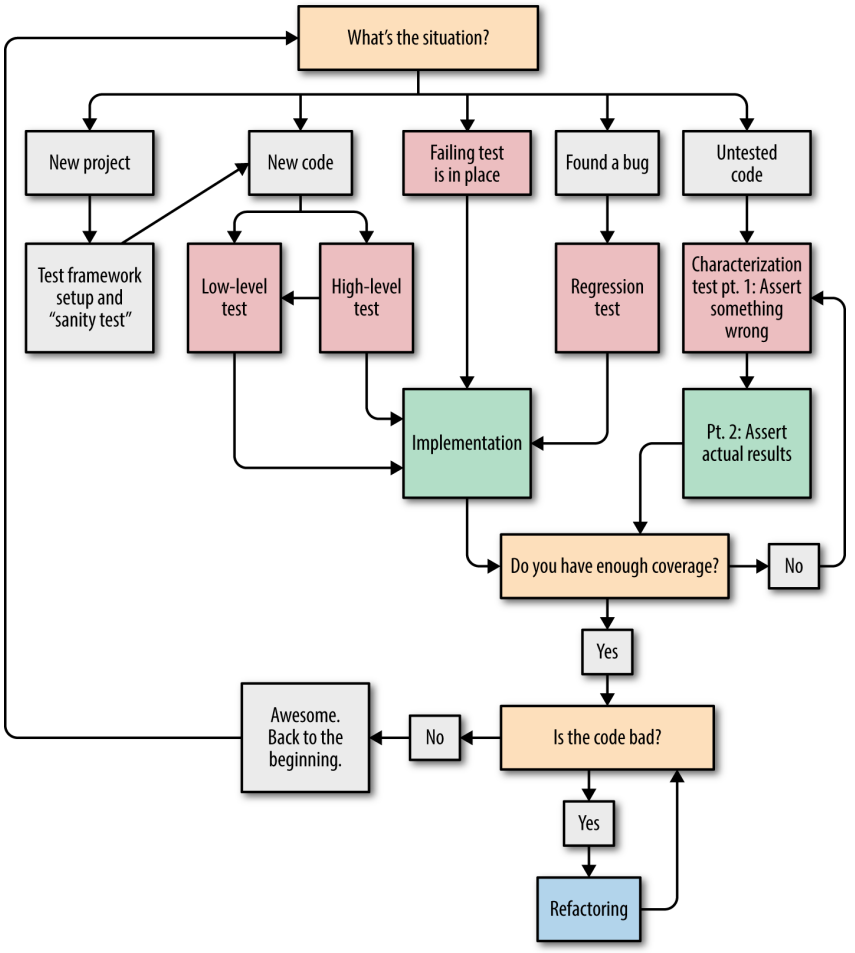
- What's the situation?
- Do you have enough coverage?
- Is the code bad?

The first question should hopefully be very easy to answer. The second one can be addressed by a coverage tool that runs with your test suite and outputs places that are specifically missing coverage. There are times when coverage tools may miss complex cases, though, so a line of code being covered does not necessarily mean we are confident in it. Achieving code quality through the process of refactoring is absolutely dependent on confidence.

In this chapter, we're developing a strategy to answer the third question: is the code bad?

FIGURE 5-1

Testing and refactoring flow chart



Due to JavaScript's multiparadigmatic nature, answering this question is not always easy. Even without using specialized libraries, we can strive to (or be forced to) make our code follow a few styles: object-oriented (prototypal or class-based), functional, or asynchronous (with promises or callbacks).

Before we address what refactoring in those styles entails, we must deal with a paradigm that the worst codebases tend to employ: *unstructured imperative programming*. If those italics didn't scare you, perhaps a description of this type of codebase in a frontend context will. Time for a spooky JavaScript story:

> The JavaScript in the file, called *main.js*, runs on page load. It's about 2,000 lines. There are some functions declared, mostly attached to an object like $ so that jQuery will do its magic.

Others are declared with `function whatever(){}` so that they are in the global scope. Objects, arrays, and other variables are created as needed along the way, and freely changed throughout the file. A half-hearted attempt at Backbone, React, or some other framework was made at one point for a few crucial and complicated features. Those parts break often, as the team member who was initially excited about that framework has moved on to a start-up that builds IoT litter boxes. The code is also heavily dependent on inline JavaScript inside of *index.html*.

Although frameworks can help to make this type of code less likely, they cannot and should not completely prevent the possibility of free-form JavaScript. However, this type of JavaScript is not inevitable, nor is it impossible to fix. The absolute first order of business is to understand functions, and not any of the fancy kinds. In this chapter, we will be exploring six basic components of functions:

- Bulk (lines of code and code paths)
- Inputs
- Outputs (return values)
- Side effects
- `this`: the implicit input
- Privacy

"JAVASCRIPT JENGA"

If the frontend JavaScript codebase spooky story didn't resonate with you, perhaps a description of what working with one is like will sound more familiar.

Programmers are tasked with making changes to the frontend. Each time they do, they make as small a change as possible, adding a bit of code and maybe some duplication. There is no test suite to run, so in lieu of confidence, the tools they have available to ensure the quality of the codebase are 1) performing as many manual checks of critical functionality as is allowed by time and patience, and 2) hope. The codebase gets larger and a bit more unstable with every change.

Occasionally, a critical bug sneaks in, and the whole stack of blocks topples over, right in front of a user of the site. The only saving grace about this system is that version control allows the unstable (but mostly okay…probably?) previous version of the blocks to be set up without too much trouble.

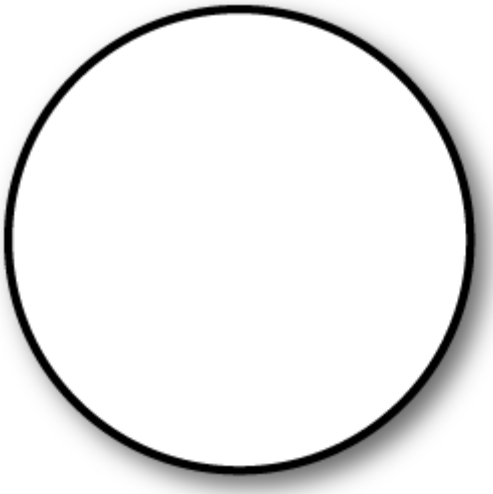
This is technical debt in action. This is JavaScript Jenga.

Throughout the rest of this chapter, we will be using a tool for diagramming JavaScript called Trellus. All the diagrams in this chapter were generated with it, and you can diagram your own functions at **trell.us**.

Here's how it starts: functions are just a circle (**Figure 5-2**).

FIGURE 5-2

Just a circle



Simple enough. We'll be adding parts to represent the following:

- Bulk
- Inputs
- Outputs (return values)
- Side effects
- Context
- Privacy

Before the chapter is through, we will have explored all of these components through this diagramming technique, making note of qualitative differences between them along the way.

Function Bulk

We use the term *bulk* to describe function bodies. It refers to two different but related characteristics:

- Complexity
- Lines of code

JavaScript linters (described in **Chapter 3**) pick up on both of these things. So if you're using one as part of a build process (or preferably within your editor), you should see warnings for both if they become excessive.

There are no strict rules on bulk. Some teams like functions to be 25 or fewer lines of code. For others, 10 or fewer is the rule. As for complexity (aka *cyclomatic complexity*), or "the number of code paths you should be testing," the upper limit tends to be around six. By "code path" or "branch of code," we mean one possible flow of execution. Branches of code can be created in a few ways, but the simplest way is through `if` statements. For example:

```
if(someCondition){
  // branch 1
} else {
  // branch 2
};

function(error){
  if(error){
    return error; // branch 1
  };
  // branch 2
};
```

Too much of one type of bulk will probably indicate the other type. A 100-line function probably has too many potential code paths as well. Similarly, a function with several `switch` statements and variable assignments will probably have a bulky line count too.

The problem with bulk is that it makes the code harder to understand and harder to test. The resulting lack of confidence is the exact scenario that leads to JavaScript Jenga.

IN DEFENSE OF BULK

Although classical refactoring techniques and trends in composable structures within frameworks and JavaScript packages point toward small functions (as well as objects and "components") being a preferable approach, there are detractors.

No one is rushing to defend 200-line functions on principle, but you may occasionally encounter some who are critical of "tiny functions that don't do anything but tell some other function to do something," which makes the logic "hard to follow."

This is tricky. Although following logic from function to function takes some patience and practice when compared to reading straight through a function, ideally there is less context to keep in your head and testing small functions is way easier.

All that said, after testing, being able to extract new functions and reduce bulk in extant ones is *the most important skill* to learn for refactoring. In doing so, however, you must make the higher-level code with consideration of the interface's ability to hide its implementation appropriately. That demands naming things well, and having sensible inputs/outputs.

If every function was named some variant of *`passResultOfDataToNextFunction`*, then extracting functions would only scatter the implementation, meaning the bulk would be preferable.

This book argues against bulk, but in your team, you might have to put up with a bit more bulk than what your preference may be. Be aware that reducing bulk, as with any refactoring target, may be met with objections on the grounds of both stylistic preferences and its (real or perceived) importance with respect to other development objectives.

Although it's a less common shape for bad code to take, if you find that your code goes too far in delegating functions for your comfort, just inline the functions. It's not hard to do:

```
function outer(x, y){
  return inner(x, y);
};
function inner(x, y){
  // setup
  return // something with x and y
};
```

You have a few choices here. If `outer` is the only thing calling `inner`, you could just move the function body from `inner` into `outer`, keeping in mind that you'd need:

```
function outer(x, y){
  // setup
  return // something with x and y
};
```

Or if the `// setup` part is complex enough, and there are a lot of calls to `inner` that don't come from `outer`, you might want to delete `outer`, change any calls from `outer` to `inner`, and, as necessary, negate or adjust the `// setup` you used for `inner`.

If you find yourself confused by inner functions, inlining them *and then* extracting new ones is a great way to explore the code. As always, have tests in place, and be ready to roll back to an earlier version.

Let's add a few pieces to our diagram to help us represent bulk, and while we're at it, we'll give our functions names. We simply have a box that states the function name and the number of lines that the function has. To represent the paths of code in (complexity in) our function, we can add pie slices. The darker shaded slices are tested, and the lighter ones are untested. Figures **5-3** and **5-4** are two examples.

Figure 5-3 has two code paths and seven lines of code. One of the code paths is tested (dark) and one is not (light).

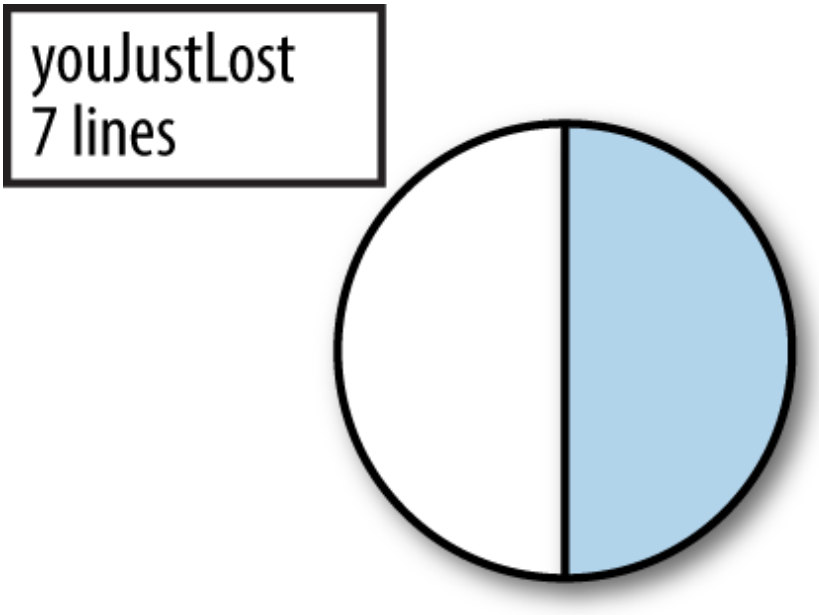


FIGURE 5-3

7 lines, with one out of two code paths tested

Figure 5-4 is bulkier, with 45 lines of code and 8 code paths (pie slices). But because all of the pie slices are dark, we know that each of the eight code paths is tested.

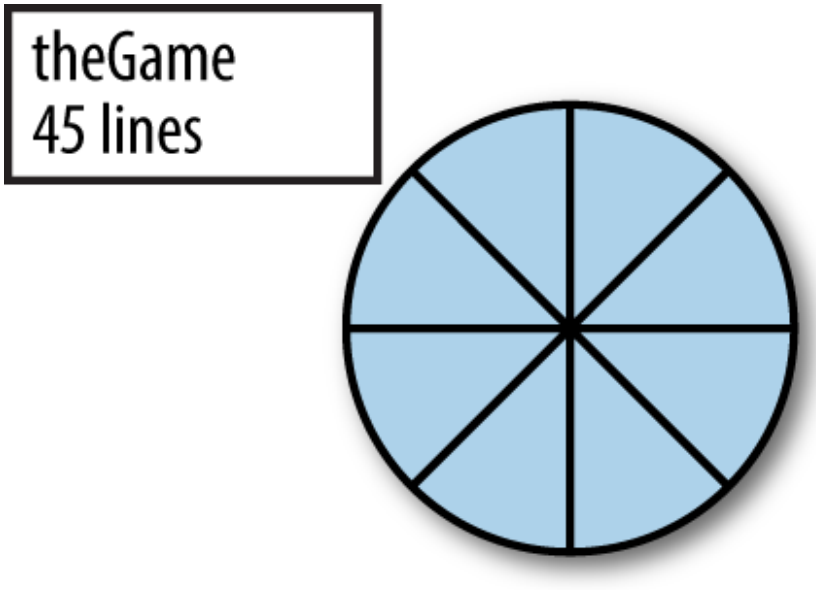


FIGURE 5-4

45 lines with eight code paths, all tested

Inputs

For our purposes, we will consider three types of inputs to a function: explicit, implicit, and nonlocal. In the following function, we would say `a` and `b` are *explicit inputs* (aka *explicit parameters* or *formal parameters*), because these inputs are part of the function definition:

```
function add(a, b){
  return a + b;
};
```

Incidentally, if we call `add(2, 3)` later on, 2 and 3 are "actual parameters," "actual arguments," or just "arguments" because they are used in the *function call*, as opposed to "formal parameters," which occur in the *function definition*. People mix up the arguments versus parameters thing all the time, so don't worry too much about it if you mix them up too. The main thing is that what we're calling "explicit parameters" appear inside of the function definition (see **Figure 5-5**).

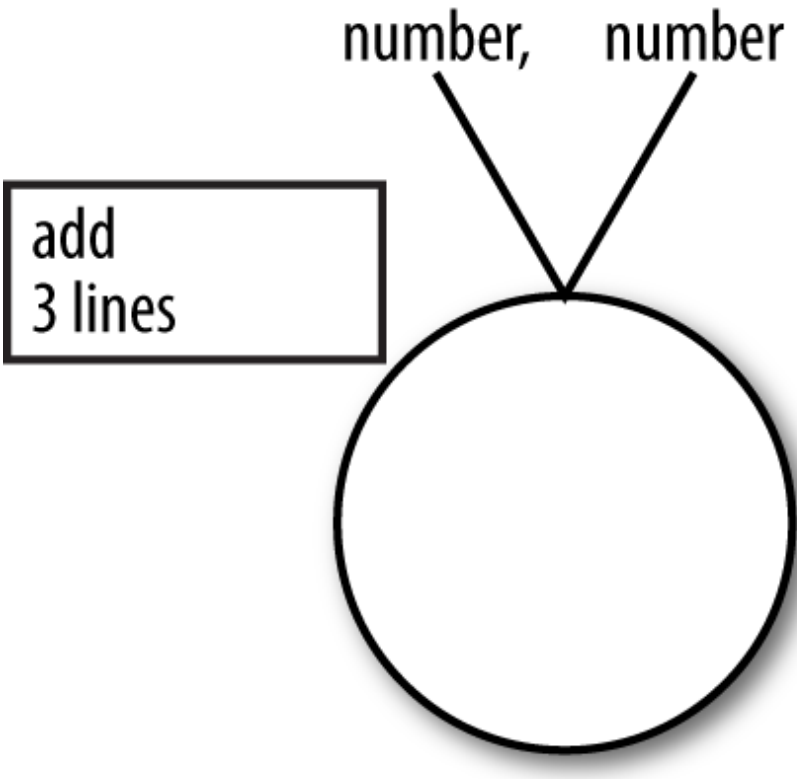


FIGURE 5-5

An add function with two explicit parameters

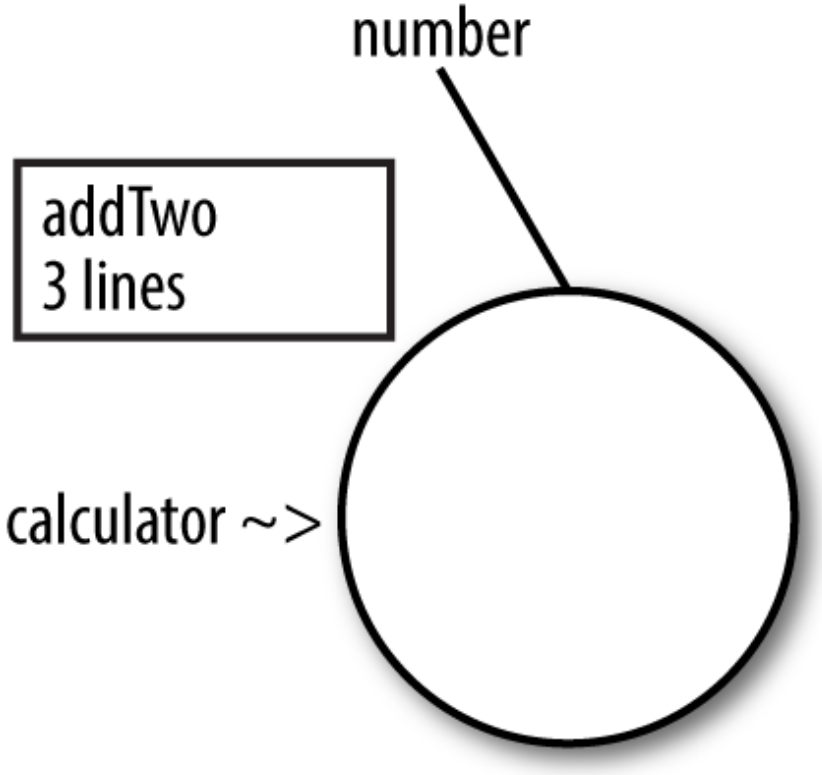
Note that we are describing the type of the inputs (`number`), rather than specific values or references to the formal parameters of the function signature (`a` and `b`). Since JavaScript doesn't care what types we pass in, our names will also not necessarily map to specific types of primitives or objects that we're passing.

Although we're not diving deeply into objects here, the *implicit input* or *implicit parameter* in the following `addTwo` function is the `calculator` object within which the function is defined (see **Figure 5-6**):

```
var calculator = {
  add: function(a, b){
    return a + b;
  },
  addTwo: function(a){
    return this.add(a, this.two);
  },
  two: 2
};
```

FIGURE 5-6

addTwo function with explicit input (number) and implicit input (calculator)



In JavaScript the implicit parameter is referred to by `this`. You may have seen it as `self` in other languages. Keeping track of `this` is probably the most confusing thing in JavaScript, and we'll talk about it a bit more in **"Context Part 1: The Implicit Input"**.

The third type of inputs, *nonlocal inputs* (commonly known as "free variables"), can be especially tricky, especially in their ultimate form, the dreaded global variable. Here's an innocent-looking example (see **Figure 5-7**):

```
var name = "Max";
var punctuation = "!";
function sayHi(){
  return "Hi " + name + punctuation;
};
```

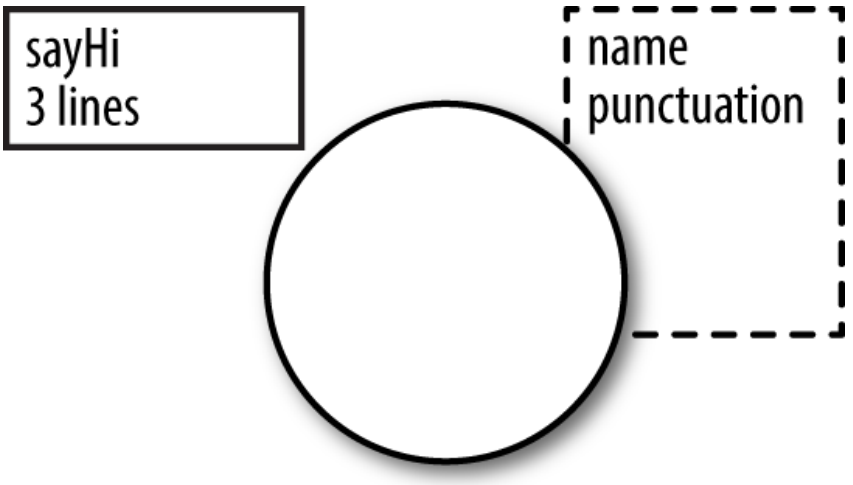


FIGURE 5-7

The sayHi function has two nonlocal inputs: name and punctuation

Is it so innocent, though? `name` and `punctuation` can be redefined, and the function can still be called at any point. The function has no opinion or guard against that. It may not jump out as a problem in these 5 lines, but when a file is 200 or 300 lines long, variables floating around like this make life more difficult, as they could change at any point.

> **"NAME" IS NOT A GREAT NAME**
>
> Although it's not perfectly supported across all implementations, `name` happens to be a property of function objects for many contexts. When in doubt, avoid it.

In testing, figuring out what inputs you need to set up can take more time than any other task involved. When explicit inputs are complex objects, this can be difficult enough (although it can be helped by factories and fixtures, as discussed in **Chapter 3**), but if your function relies heavily on implicit input (or worse, nonlocal/global inputs), then you have that much more setup to do. Implicit state (using `this`) isn't as bad as nonlocal input. Actually, object-oriented programming (OOP) relies on using it intelligently.

The recommendation here is to have your functions, as much as possible, rely on explicit inputs (which hints at functional programming, or FP, style), followed by implicit inputs (aka `this`, which hints at OOP style), followed in a distant third by nonlocal/global state. An easy way to guard against nonlocal state is to wrap as much of the code as possible in modules, functions, and classes (which, yes, are actually functions in disguise).

Note that even when state is explicit, JavaScript allows a wide range of flexibility in how parameters are passed. In some languages, formal parameters (the parts of a function definition that specify explicit inputs) require the types (`int`, `boolean`, *`MyCoolClass`*, etc.) to be specified as well as the parameter name. There is no such restriction in JavaScript. This leads to some convenient possibilities. Consider the following:

```
function trueForTruthyThings(sometimesTruthyThing){
  return !!(sometimesTruthyThing);
};
```

When calling this function, we could pass in a parameter of any type we wanted. It could be a boolean, an array, an object, or any other type of variable. All that matters is that we can use the variable inside of the function. This is a very flexible approach, which can be handy, but can also make it difficult to know what types of parameters are required in order to exercise a function under test.

JavaScript offers two additional approaches that increase flexibility even further. The first is that the number of formal parameters doesn't have to correspond with the number passed into the function call. Take a function like:

```
function add(a, b){
  return a + b;
};
```

This will happily return 5 if you call it as `add(2, 3)`, but also if you call it as `add(2, 3, 4)`. The formal parameters don't care what you pass in; only the function body does. You can even supply fewer arguments, as in this: `add(2)`. This will return `NaN` ("Not a Number") because 2 is being added to `undefined`, although in **Chapter 11**, we'll explore a technique called *currying* that makes it useful to supply fewer arguments than specified by the formal parameters.

As for extra arguments supplied to a function call, it is possible to recover and use them in the function body, but this functionality should be used cautiously, as it complicates a testing procedure that benefits from simple inputs and less bulk in the function body.

One last trick that the formal parameters can play in JavaScript is allowing not just simple types but also objects and functions as parameters. Sometimes this extreme amount of flexibility can be useful, but consider the following case:

```
function doesSomething(options){
  if(options.a){
    var a = options.a;
  }
  if(options.b){
```

```
    var b = options.b;
  }
  ...
}
```

If you have a mystery object or function that is passed in at runtime, you have potentially bloated your test cases without realizing how. When you hide the values needed inside of an object with a generic name like `params` or `options`, your function body should hopefully supply guidance on how those values are used and clues about what they are. Even with clarity inside of the function body, though, it's definitely preferable to use parameters with real names to keep the interface smaller and help to document the function right up top.

There is nothing wrong with passing a whole object to a function rather than breaking it all into named individual parameters, but calling it `params` or `options` might hint that the function is doing too much. If the function does take a specific type of object, it should have a specific name. See more about renaming things in **"Renaming Things"**.

A BIT ABOUT ECMASCRIPT

We briefly discussed the ECMAScript (ES) specification in **Chapter 2** as what to watch for updates to JavaScript (keeping in mind that libraries and implementations may either lag behind or actually be ahead of the curve). What we didn't talk about is how ES naming conventions work. As a relatively new convention, the standards body dropped version numbers like "ES6" in favor of yearly releases like "ES2015" (which happens to correspond to ES6). The version that will be released next is described as "ESNext." As of this writing, this convention hasn't been in use for long, so don't be shocked if this changes again in the future.

Prior to ES2015, passing an object to a function call offered the advantage of somewhat illustrating the function signature (what parameters are used) as opposed to having what could be just magic strings or numbers. Compare these two:

```
search('something', 20);
// vs.
search({query: 'something', pageSize: 20});
```

The first function definition would necessarily include explicitly named parameters. The second would likely have one parameter named, at best, `searchOptions` and at worst `options`. The first name (`searchOptions`) does not offer much more detail to document the call, but is at least potentially unique.

However, there is a way (thanks to ES2015) that you can have clarity in the calls and in the definitions:

```
function search({query: query, pageSize: pageSize}){
  console.log(query, pageSize);
};
search({query: 'something', pageSize: 20});
```

It's a little awkward, but it allows you to be specific in both places and avoids the pitfalls of sending the wrong types of arguments or mindlessly passing a `params` object through to another function call (maybe after some manipulations first). Honestly, that second pattern is so bad and rampant that this somewhat-awkward construct still looks pretty great by comparison. If you're curious about the feature that makes this work, it's called *destructuring*, and there's a lot to it. It's for assignments in general (not just `params`), and you can use it with arrays as well as objects.

If you throw using a function as a parameter (a *callback*) into the mix, then you could be adding significantly more bulk. Every function call now has the potential to require any number of new test cases to support it.

SAD PATHS

We discussed earlier how coverage does not necessarily equate to confidence. *Sad paths* are one particular reason why. The problem might be a data integrity issue from a user interaction gone awry (e.g., an incorrectly formatted form entry step that allows bad data to get into the database).

Even if your code tests each branch of an `if-else` clause inside your function, some inputs will cause problems. This includes not just unexpected parameters passed to the function, but also nonlocal inputs such as what time or date it is, as well as random numbers.

In languages with strict type checking systems (unlike JavaScript), a good number of these sad paths go away. Yet automated coverage tools will not help you here. Sad paths will likely be hidden in the code (and test cases) that you didn't write.

Mutation testing, as described in **Chapter 3**, can be of some help here. But considering that there are potentially infinite error-causing inputs to your functions in JavaScript (and the more flexibility you insist on, the more likely you are to hit one), the best defense against sad paths is to ensure your inputs will be evaluated correctly by your function ahead of time. Otherwise, you're stuck with seeing the bug happen live, and then writing a regression test and code to handle it.

Note that because of input validation (or using more robust mechanisms inside of a function), a sad path will not necessarily mean a new path of code that would require a new pie slice in our diagramming method.

Passing in functions as inputs to a function can be done in a rational way, but not recognizing the trade-offs between simplicity in testing and flexibility is a mistake.

The overall recommendation for this section is to have simple and explicit inputs whenever possible, both in the function definitions and calls. This makes testing a great deal easier.

Here are a few recommendations to wrap up our discussion on inputs:

- Overall, the fewer inputs you have, the easier it will be to keep bulk under control and test your code.
- The fewer nonlocal inputs you have, the easier your code will be to understand and test.
- Every function has a `this` as an implicit input; however, `this` may be unused, or even `undefined` in a lot of cases. If `this` is unused, feel free not to add the `thisType ~>` part of the diagram.
- Most of all, explicit inputs are more reliable than `this` or nonlocal inputs.

Outputs

By output, we mean the value that is returned from a function. In our ideal style, we always want to return something. There are cases where this is hard or not true (some asynchronous styles and procedural, side effect–driven code), but we'll deal with asynchronous code in **Chapter 10** and discuss side effects later in this chapter (as well as in **Chapter 11**).

One of the most common mistakes you will see in a function is ignoring the output by not returning anything:

```
function add(a, b){
  a + b;
};
```

In this case, it's pretty clear that the `return` keyword has been omitted, which means this function will simply return `undefined` (**Figure 5-8**). This can be easy to miss, for two reasons. First of all, not all languages are the same. Rubyists (a lot of whom write JavaScript as their second language) will sometimes forget the `return` statement because the last line in a Ruby function is returned implicitly.

Secondly, if the style of most of the codebase consists of side effects (covered next), then the return value is less important than that effect. This is especially common in jQuery-supported codebases where almost every line is a click handler that runs a callback (which is often fancy side effect code). Programmers used to side effect–based code will also tend to fail to return anything.

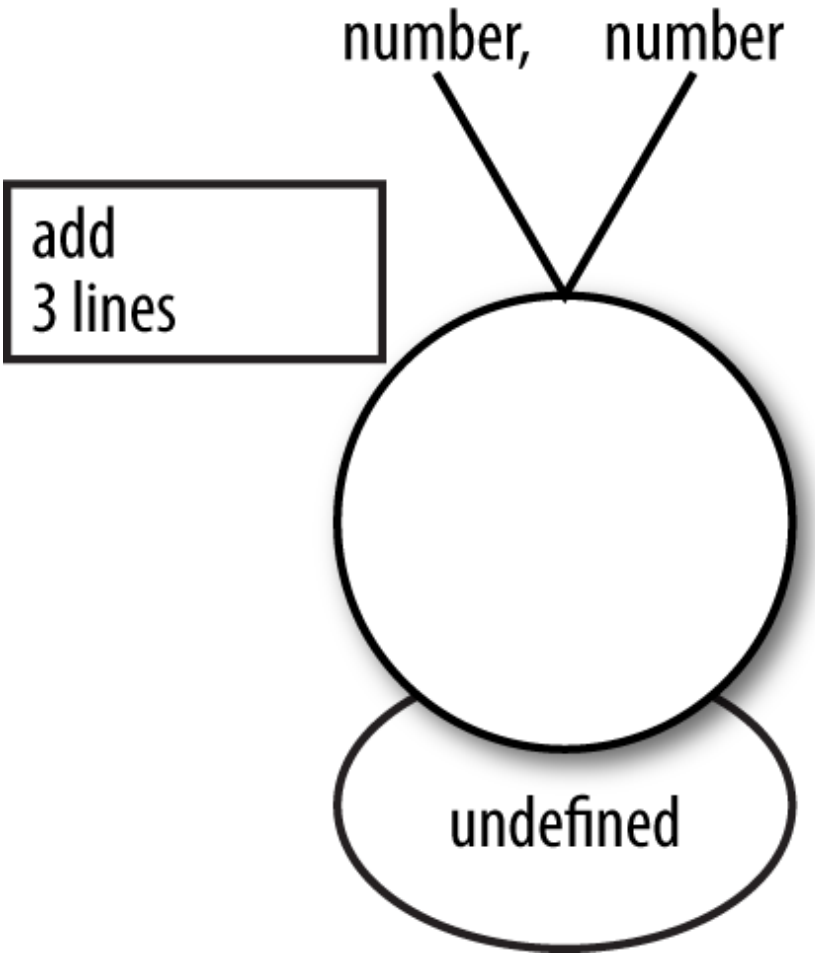


FIGURE 5-8

This add function doesn't specify a return value, so it defaults to undefined

VOID FUNCTIONS

If no value is explicitly returned in JavaScript (using the `return` keyword), `undefined` will be returned. Since this provides little to no information about what the function did, even in the traditional case of using a void function for side effect–producing functions, there is likely some information of value produced by the side effect—even if it is just the result or simply the success of the side effect—that it would be preferable to return. When the side effect specifically changes the context (`this`), returning `this` is also a viable option.

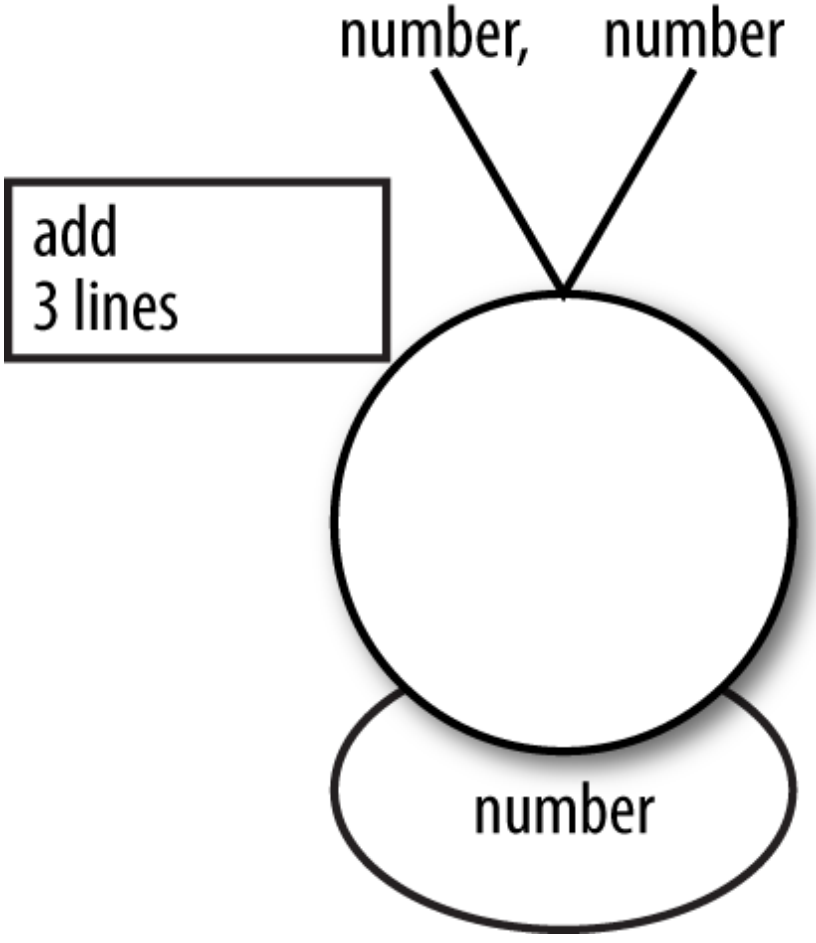
It is recommended that you return real values when possible rather than explicitly returning `undefined`/`null` or implicitly returning `undefined`.

This function, however, does return something (see **Figure 5-9**):

```
function add(a, b){
  return a + b;
};
```

FIGURE 5-9

An add function that returns a number



Generally speaking, we want a decent return value (not `null` or `undefined`), and we want the types from various code paths to match—returning a string sometimes and a number other times means you will probably have a few `if-else` statements in your future. Returning an array with mixed types can be similarly awkward when compared with returning a simple value or an array of like types.

STRONGLY TYPED LANGUAGES

Some languages have mechanisms to prescribe that return values (and inputs) be of a particular type (or explicitly return nothing). Having seen how JavaScript handles inputs, we should not be surprised to find a similar flexibility for return values.

We'll get into this more in **Chapter 11**.

The recommended approach to output values is to, whenever possible, return a consistent and simple value, and avoid that value being `null` or `undefined`. With functions that cause destructive actions (like altering an array or changing the DOM), it can be nice to return an object that describes what effect took place. Sometimes that just means returning `this`. Returning something informative, even when nothing was explicitly asked for, is a good habit, and helps in testing and debugging.

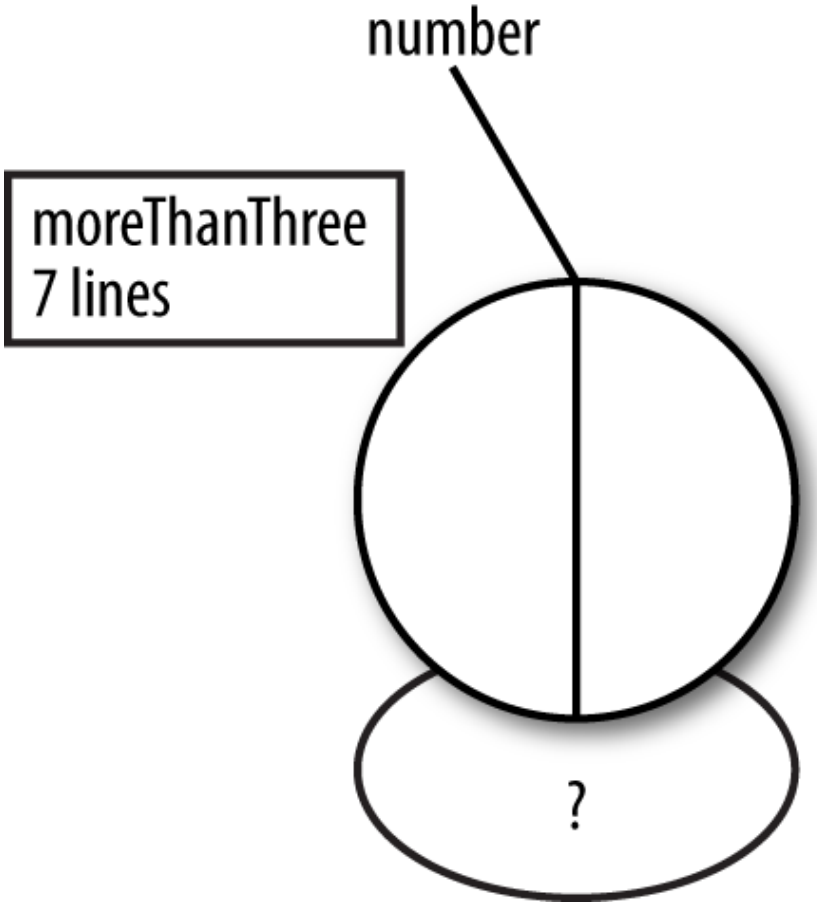
One additional complication for outputs is functions that return different types. Consider this function:

```
function moreThanThree(number){
  if(number > 3){
    return true;
  } else {
    return "No. The number was only " + number + ".";
  }
};
```

This function returns either a boolean or a string (see **Figure 5-10**). This isn't great because code that calls this function will likely have its own conditionals to check for which type was returned.

FIGURE 5-10

This function could return a boolean or a string



As is the theme so far for this chapter, simpler is better when it comes to outputs (return types). Returning different types of values can complicate the code. Additionally, we want to avoid returning a `null` or `undefined`, as a better solution is likely available. Finally, we should strive to return values that are of the same type (or types implementing interfaces that won't require a conditional check after or around the function call), regardless of which value is returned.

Side Effects

Some languages find side effects to be so *dangerous* that they make it very difficult to introduce them. JavaScript doesn't mind the danger at all. As it is used

in practice, one of jQuery's main jobs (if not its primary responsibility) is to manipulate the DOM, and this happens through side effects.

The good part about side effects is that they are usually directly responsible for all the work anyone cares about. Side effects update the DOM. Side effects update values in the database. Side effects make `console.log` happen.

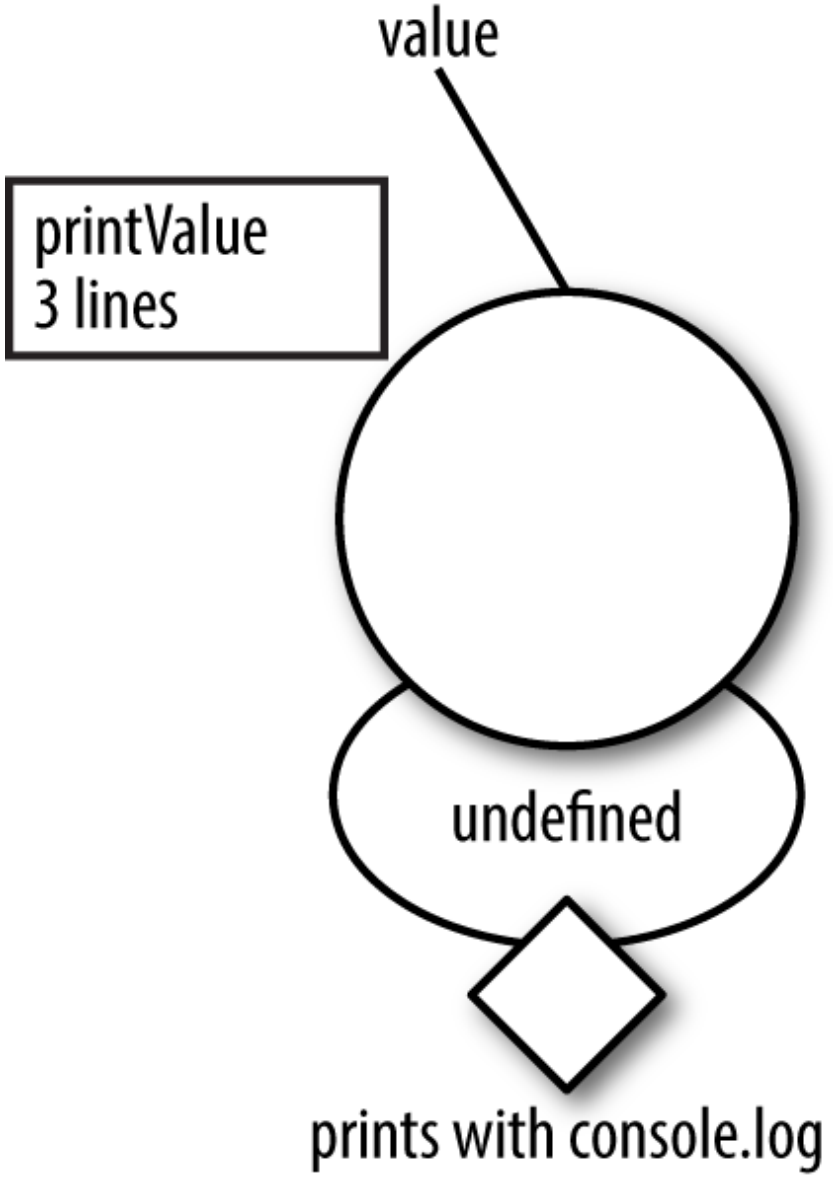
Despite all that side effects make possible, our goal is to isolate them and limit their scope. Why? There are two reasons. One is that functions with side effects are harder to test. The other is that they necessarily have some effect on (and/or rely on) state that complicates our design.

We will discuss side effects much more in **Chapter 11**, but for now, let's update our symbolic diagramming of side effects. Here is a simple example (see **Figure 5-11**):

```
function printValue(value){
  console.log(value);
};
```

FIGURE 5-11

A function with a very common side effect: logging



Notice that because this function does not return anything, the return value is `undefined`.

Ideally, the fewer side effects the better, and where they must exist, they should be isolated if possible. One update to some well-defined interface (say, a

single database row) is easier to test than multiple updates to it, or to multiple database rows.

Context Part 1: The Implicit Input

In explaining inputs and outputs, we skimmed over something somewhat complex but very important, which we will cover now: the "implicit input," which appears on the left of the diagrams as *someThisValue* ~> (see e.g., **Figure 5-6**). *someThisValue* is the `this` that we've been talking about so far.

So what is `this`?

Depending on your environment, outside of any other context, `this` could refer to a base object particular to the environment. Try typing `this` (and pressing Enter) in a browser's interpreter (console). You should get back the `window` object that provides the kinds of functions and subobjects you might expect, such as `console.log`. In a node shell, typing `this` will yield a different type of base object, with a similar purpose. So in those contexts, typing any of these things will give you `'blah'`.

```
console.log('blah');
this.console.log('blah');
window.console.log('blah'); // in a browser
global.console.log('blah'); // in a node shell
```

Interestingly enough, if you save a node file and run it, `this` prints as an empty object, `{}` , but `global` works as in the node shell and `global` objects like `console` are still available to use. Although `this.console.log` won't work in a node file that you run, `global.console.log` will. That is related to how the node module system works. It's a bit complicated, but know that the top-level scope in most environments is also `this`, but in node, it's the module scope. Either way, it's fine, because most of the time you don't want to define functions or variables in the global namespace anyway.

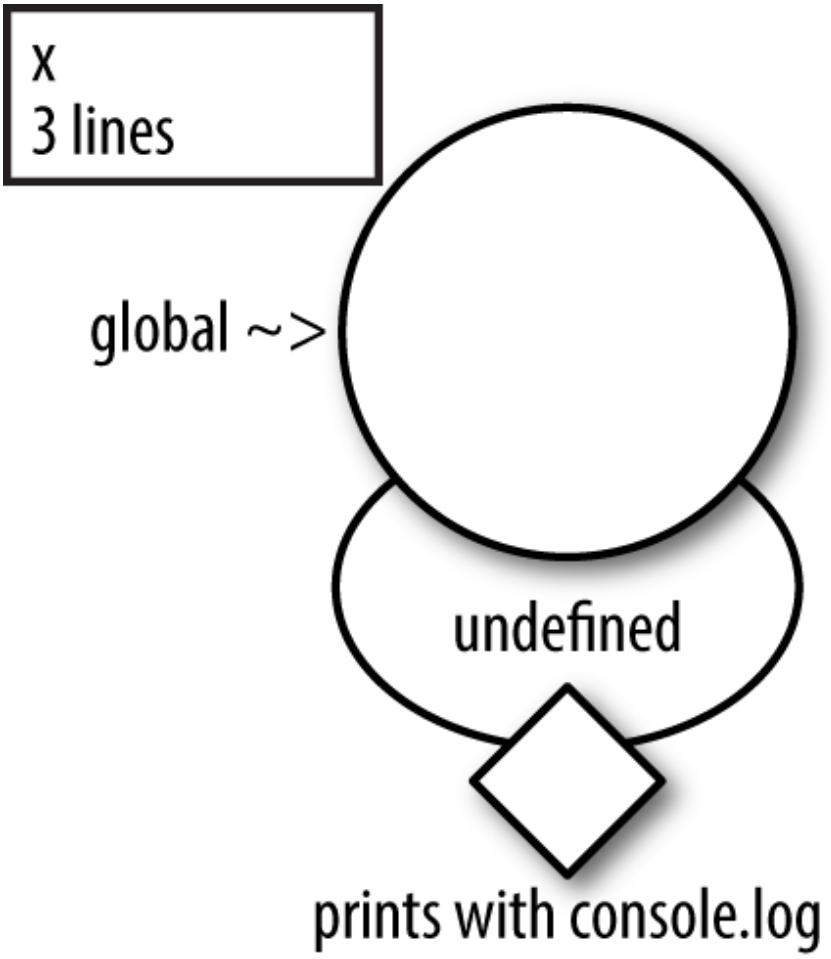
In any case, we can expect our `this` to be the top-level scope when we check it inside of functions declared in the top-level scope:

```
var x = function(){
  console.log(this);
}
x(); // here, we'll see our global object, even in node files
```

So, that is top-level scope in a nutshell. Let's diagram this last code listing with its implicit input (**Figure 5-12**).

FIGURE 5-12

This function has the global object as its "this" value



this in Strict Mode

When you're in strict mode, `this` will behave differently. For this code:

```
var x = function(){
  'use strict'
  console.log(this);
}
x();
```

`undefined` will be logged. In strict mode, not every function has a `this`. If you create this script and run it with node:

```
'use strict'
var x = function(){
  console.log(this);
}
x();
```

You will see the result (`this` is `undefined`). However, typing this second code snippet line by line in a node REPL (just type **node** at the terminal) or in a browser console will not apply strict mode to x, and the global object will be returned.

The Trellus function diagram for x in the first snippet looks like **Figure 5-13**.

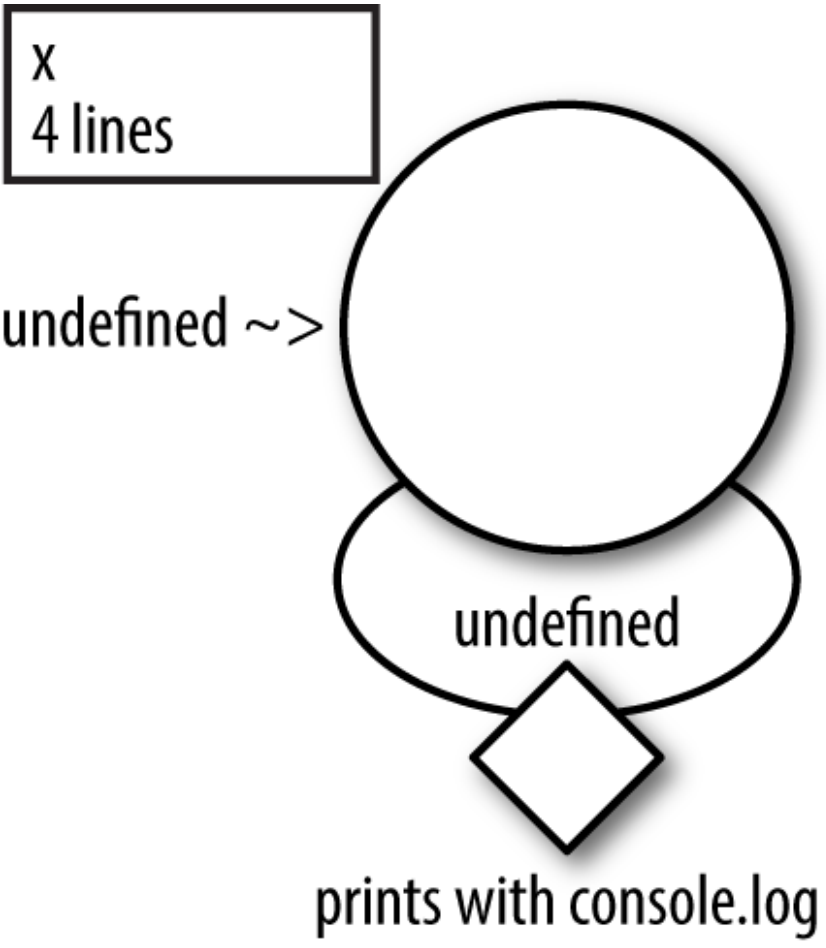


FIGURE 5-13

A function with a this value of undefined

Now the function is four lines long and still returns `undefined` as it did before. The side effect (which now logs `undefined` instead of the global object) is still present. The most crucial difference is that this no longer attaches to any `this` object other than `undefined`.

Whether you're writing a node module or any decently sized program on the frontend, you'll generally want only one or a small handful of variables to be scoped inside of the top-level scope (something has to be defined there, or you wouldn't be able to access anything from the outermost context).

One way to create a new context is by using a simple object like this:

```
var anObject = {
  number: 5,
  getNumber: function(){ return this.number }
}

console.log(anObject.getNumber());
```

Here, `this` is not the global object, but `anObject`. There are other ways to set a context, but this is the simplest. By the way, because you're creating a literal object with the `{}` syntax, this is called an *object literal*.

Keeping in mind that we're diagramming functions, not objects, let's see what our `getNumber` function looks like (**Figure 5-14**).

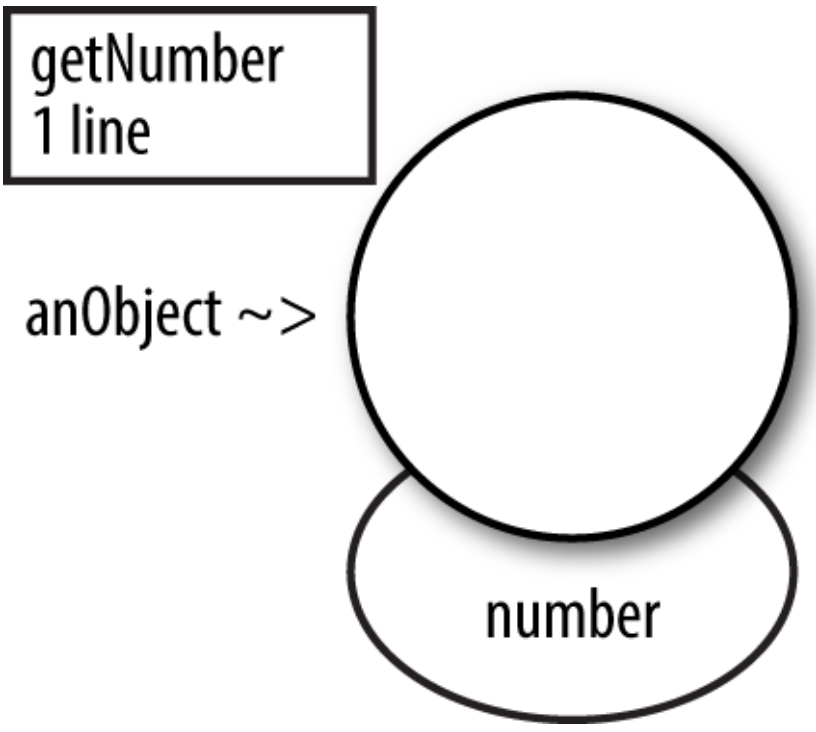


FIGURE 5-14

getNumber attached to a this of anObject

We have a few more ways to write code that will follow the diagram as well as the interface. First is the `Object.create` pattern:

```
var anObject =
    Object.create(null, {"number": {value: 5},
    "getNumber": {value: function(){return this.number}}});
console.log(anObject.getNumber());
```

Next is how you would attach `getNumber` to `anObject` using classes:

```
class AnObject{
  constructor(){
    this.number = 5;
    this.getNumber = function(){return this.number}
  }
}
anObject = new AnObject;
console.log(anObject.getNumber());
```

SOME PEOPLE REALLY, REALLY HATE CLASSES

Pretty much everyone likes object literals. Some people like `Object.create`. Some people like classes. Some people like writing constructor functions that behave a lot like classes, without all of the "syntactic sugar."

Some objections to classes are are based on them obscuring the purity and power of JavaScript's prototypal system...but on the other hand, a typical demonstration of this power and flexibility is the creation of an awkward, ad hoc class system.

Other objections to classes are based around inheritance being worse than delegation and/or composition, which is valid, although fairly unrelated to the use of *just* the `class` keyword.

If you are using the `new` keyword, whether from a class or a constructor function, `this` will be attached to a new object that is returned by the call to `new`.

In an OOP approach, you might find yourself using objects for simple things and object-producing classes, factory functions, and so on for more complex things, whereas in the FP style, you will probably find yourself using fewer classes.

JavaScript doesn't seem to care how you write it. Perhaps you'll find FP style to be more maintainable, but a given project or team could have a significant investment in an OOP style.

Your `this` can also change through use of the `call`, `apply`, and `bind` functions. `call` and `apply` are used in exactly the same way if you're not passing any explicit inputs to the function. `bind` is like `call` or `apply`, but for saving the function (with the bound `this`) for later use:

```
var anObject = {
  number: 5
}
var anotherObject = {
  getNumber: function(){ return this.number }
}
console.log(anotherObject.getNumber.call(anObject));
console.log(anotherObject.getNumber.apply(anObject));
var callForTheFirstObject = anotherObject.getNumber.bind(anObject);
console.log(callForTheFirstObject());
```

Note that neither object has both the number and the function. They need each other. Since we're using `bind`, `call`, or `apply`, our diagram actually

doesn't have to change. As we are using it, `this` still refers to `anObject`, even though the function is defined on `anotherObject` (**Figure 5-15**).

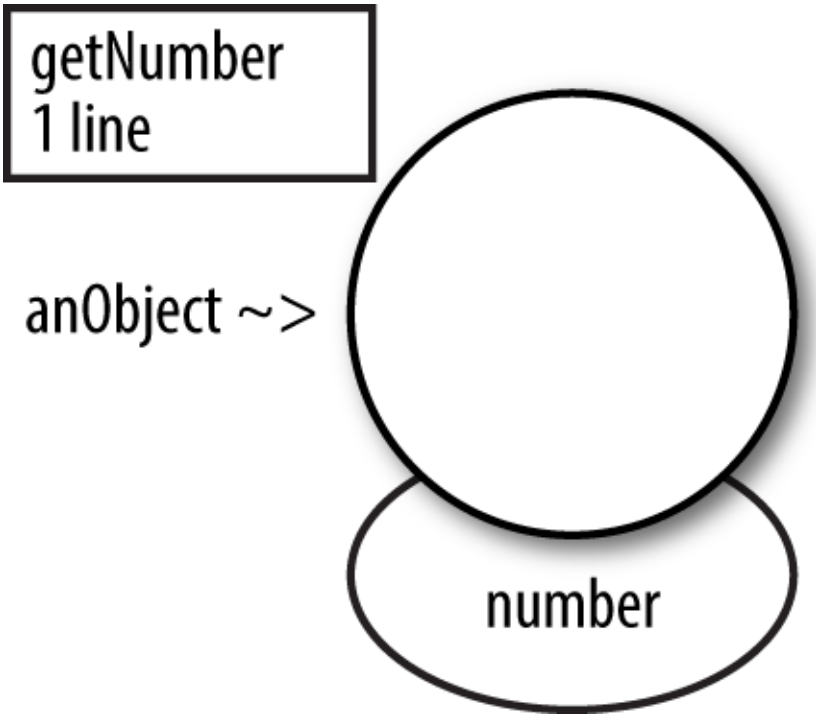


FIGURE 5-15

Nothing is actually changed here: getNumber still has anObject as its "this"

What's new here is that, although the function lives in `anotherObject`, `bind`, `call`, and `apply` are used to assign the "implicit" input (`this`) in an "explicit" way to `anObject`.

You might be confused about why **Figure 5-15** has `anObject` as its implicit parameter, even though the function is defined inside of `anotherObject`. It is because we're diagramming the function calls that are like this:

```
anotherObject.getNumber.call(anObject);
```

We're diagramming from the perspective of a function. We could also diagram the following function call:

```
anotherObject.getNumber();
```

And then `anotherObject` would be the implicit parameter (the `this`), but its return type would be `undefined`, not `number`.

Let's look at one more example:

```
var anObject = {
  number: 5
}
var anotherObject = {
  getIt: function(){ return this.number },
  setIt: function(value){ this.number = value; return this; }
}
console.log(anotherObject.setIt.call(anObject, 3));
```

Note that the `setIt` code returns its `this` value, so if you run this code, you will see the full `anObject` object with the updated value: `{ number: 3 }`. This is what `return this` in the `setIt` function does. Otherwise, we would have just the side effect (mutating the `anObject`'s `number`) and no clear confirmation of what happened. Returning `this` makes testing (manually or automatically) much easier than just returning `undefined` from side effect–causing methods.

Let's look at the diagram for `setIt` as it is called by the `call` function (**Figure 5-16**).

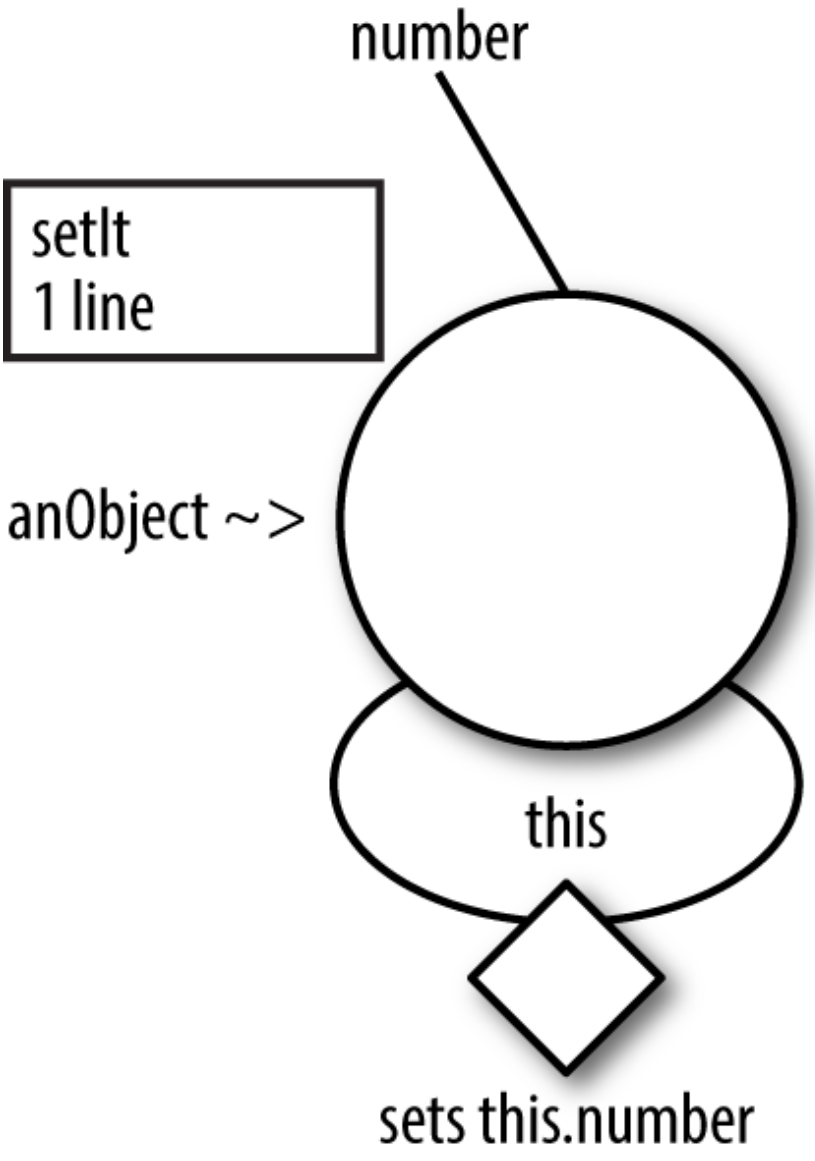


FIGURE 5-16

Has a side effect, but still returns something useful

Note that even though it is a side effect–producing method, we have returned something: specifically, our `this` from `anotherObject`. As described earlier, that can help to simplify verification and testing as compared with simply returning `undefined`. Additionally, returning `this` opens us up to the pos-

sibility of having a *fluent interface*, which means we could chain functions like this:

```
object.setIt(3).setIt(4).setIt(5);
```

FLUENT INTERFACES

Fluent interfaces can be useful for things like aggregating database query conditions or bundling up DOM manipulations (jQuery does this). You might also see this described as *chaining functions*. This is a common and useful pattern in both OOP and FP styles. In OOP, it tends to be a result of returning `this`, whereas in FP, it is commonly the result of `map`, which returns an object (or *functor*, actually) of the same type (for instance, arrays "map" to other arrays and promises "then" onto other promises).

In FP style (and in the wrapper OOP patterns), you'll see more of this:

```
f(g(h()));
```

That might seem very awkward in comparison with fluent interfaces, but FP has great strategies for combining/composing functions. We'll look at those more in **Chapter 11**.

All this is to say: if you're just returning `undefined` anyway, returning `this` instead will provide more information and better interfaces.

As far as context goes, that's about as complicated as it gets without delving into prototypes (and the three or four things that means), inheritance, mixins, modules, constructors, factory functions, properties, descriptors, getters, and setters.

After this chapter, it's all about *better code* through *better interfaces*. We won't shy away from the topics in the previous paragraph, but we won't make idols of any patterns either. This book is meant to explore many different ways to improve code.

Any coding style you fall in love with is bound to be someone else's heresy. JavaScript presents many opportunities for both reactions.

Context Part 2: Privacy

The last topic in this chapter is that of "private" functions, and what that means in JavaScript. *Scope* is a broader topic of hiding and exposing behavior that we'll explore through examples later on. For now, we are only concerned with the privacy of functions, because as we noted earlier, private functions have unique implications for testing.

Namely, some feel that private functions are "implementation details," and thus do not require tests. If we accept that premise, then ideally we can hide

most of our functionality in private functions and have less code exposed that we need to test. Fewer tests can mean less maintenance. Additionally, we can clearly separate our "public interface" from the rest of the code, which means anyone using our code can still benefit when learning or referencing a small part of it.

So how do we create private functions? Pretending we don't know anything about objects for a minute, we could do this:

```
(function(){
  console.log('hi');
})();
```

or this:

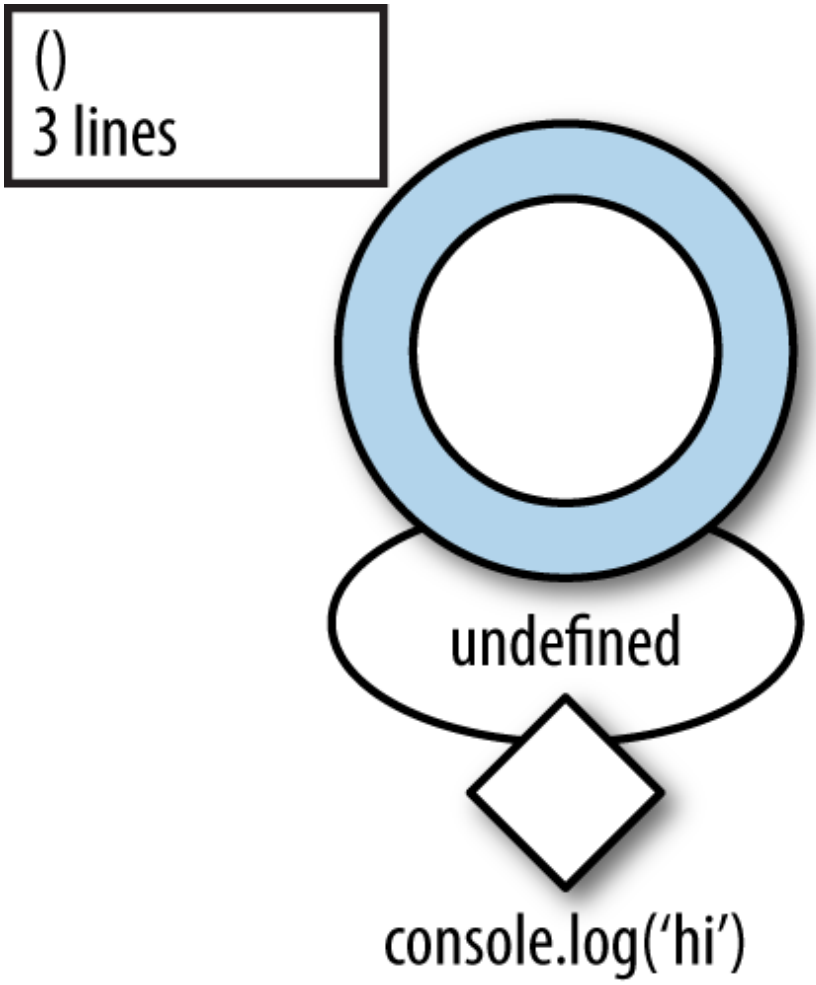
```
(function(){
  (function(){
    console.log('hi');
  })();
})();
```

Here we have some anonymous functions. They are created, and then disappear. Any `this` we put inside will link back to the top-level context. Since they're anonymous, all they can do is run when we tell them to. Even if we know the `this` for anonymous functions, we can't run them (except right away or as a callback), because they don't have a name and can't be addressed. Incidentally, these are called *immediately invoked function expressions* (IIFEs), and we'll talk about them more in **Chapter 7**.

We'll explore a few more useful types of private functions in a minute, but first we have a new piece to add to our diagrams (see **Figure 5-17**). The diagrams for both of the preceding functions are the same (except for the lines of code, which you might argue is five rather than three for the second one).

FIGURE 5-17

A "private" anonymous function



What's new here is that we have a dark ring around the main circle to denote that this is a "private" function, and you may or may not want (or be able) to test it. Any pie slices would still be apparent in a function with more code paths, and as with public functions, the slices would be darkened when tested.

Another way to design private methods is through the revealing module pattern:

```
var diary = (function(){
  var key = 12345;
  var secrets = 'rosebud';

  function privateUnlock(keyAttempt){
```

```
    if(key===keyAttempt){
      console.log('unlocked');
      diary.open = true;
    }else{
      console.log('no');
    }
  };

  function privateTryLock(keyAttempt){
    privateUnlock(keyAttempt);
  };

  function privateRead(){
    if(this.open){
      console.log(secrets);
    }else{
      console.log('no');
    }
  };

  return {
    open: false,
    read: privateRead,
    tryLock: privateTryLock
  }

})();

// run with
diary.tryLock(12345);
diary.read();
```

Reading this from top to bottom is a mistake. At its core, this is just creating an object with three properties, and assigning it to `diary`. We happen to be surrounding it with an anonymous (and immediately executing) function, but that is just to create a context where we can hide things. The easiest way to read this function is to first look at the object that it returns. Otherwise, it's fairly similar to the last example in that all we're doing is wrapping some code with an anonymous function.

`diary`'s first property is `open`, which is a boolean initially set to `false`. Then it has two other properties that map to function definitions provided previously. The interesting part is that we have some things hidden in here. Neither the `key` and `secrets` variables nor the `privateUnlock` function has any way to be accessed directly through `diary`.

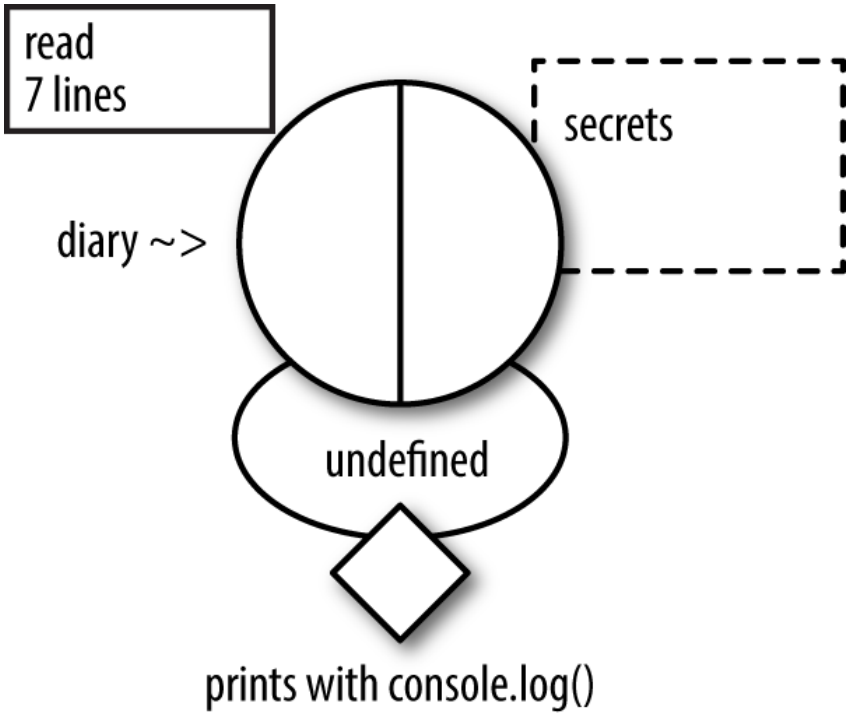
One thing that may look strange is that in the "private" `privateUnlock` function, instead of `this.open`, we have `diary.open`. This is because when

we are running `privateUnlock` via the `privateTryLock` function, we lose our `this` context. To be clear, `this` is `diary` inside of the `privateTryLock` function, but it is the global object inside of `privateUnlock`.

As Trellus diagrams, these functions would look like Figures **5-18** and **5-19**.

FIGURE 5-18

The read function of diary



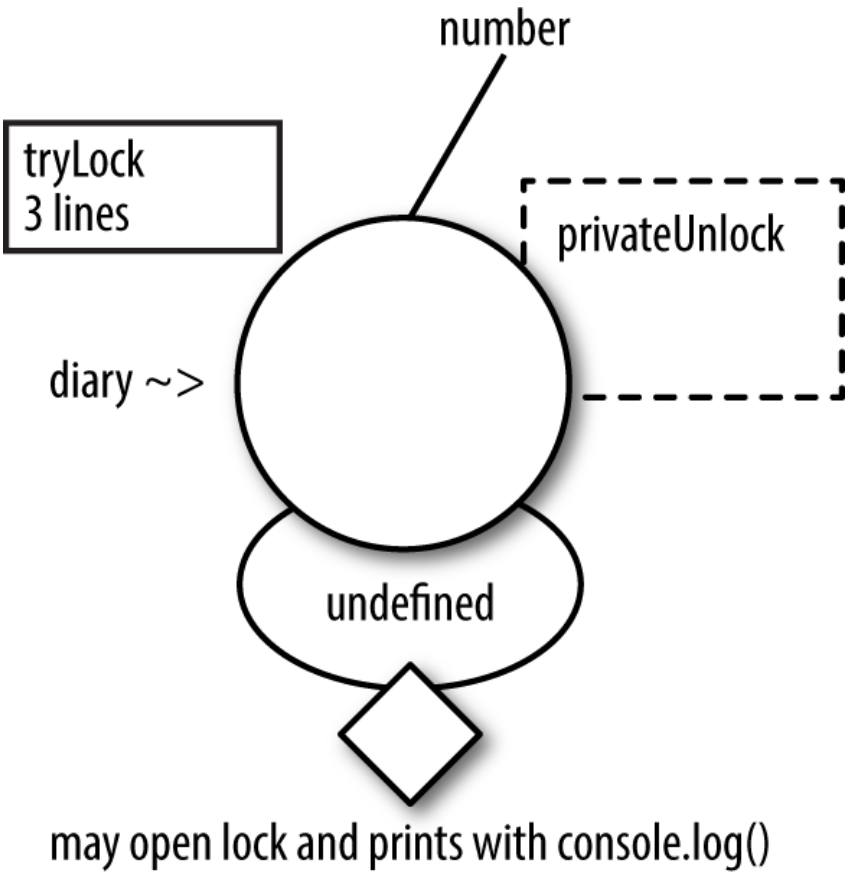


FIGURE 5-19

The tryLock function of diary

The `read` function just points to `privateRead`, so we use that definition for our diagram. It takes no explicit parameters. Its `this` (the implicit parameter) is the `diary` object (that is returned from the anonymous function call). It returns `undefined` and calls `console.log` as a side effect. But what about `secrets`, the nonlocal input? It is tempting to think of `secrets` as part of the `diary` *object*, but it isn't. It's part of the *scope* within which the object returned by `diary` was created. Contrast `secrets` with `this.open`, which is an attribute of the `diary` object itself.

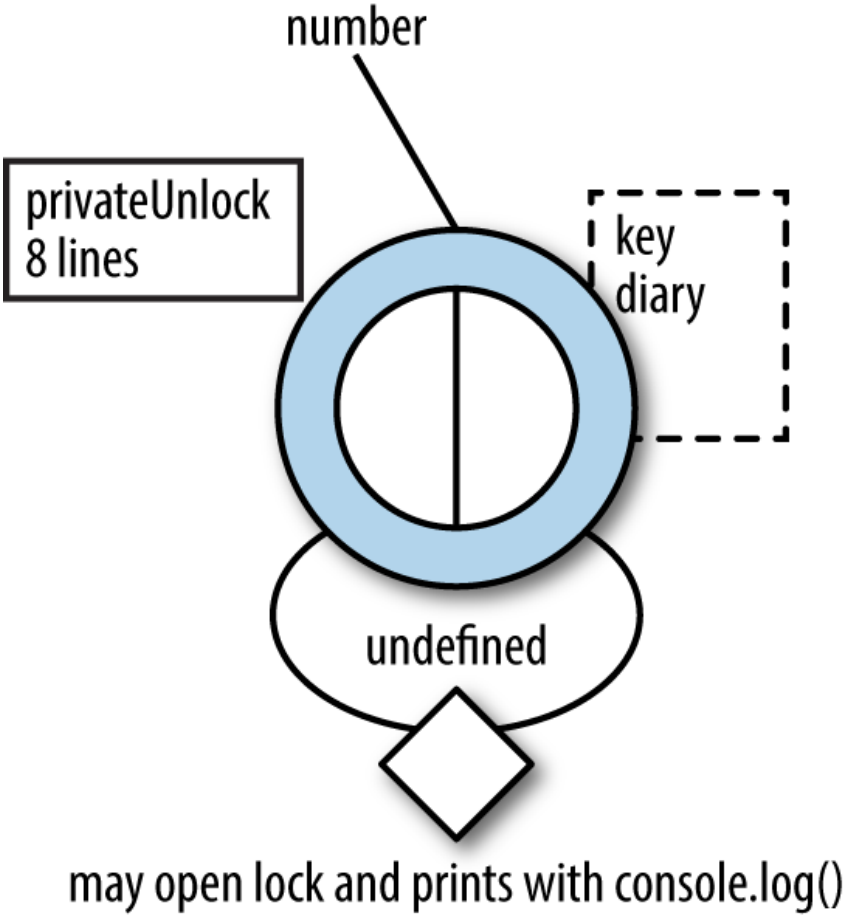
The `tryLock` function also just points to another function (`privateTryLock`), so we use that function definition. Like `read`, it has a nonlocal input, but this time it is a function (`privateUnlock`) rather than a simple value (`secrets`) as in `read`'s case. As far as the return values and side effects, the function definition itself doesn't indicate it, but we don't have to work too hard to see what

side effects it has. However, in keeping with what most code coverage tools would report, this function only has *one* code path, even though its side effects depend on *two* paths inside of `privateUnlock`. Note that the same single code path would also be indicated by the diagram in the case that those two code paths affected the return value rather than, or in addition to, the side effects.

Now let's look at `privateUnlock` (**Figure 5-20**).

FIGURE 5-20

The privateUnlock function of diary



This function's diagram looks a lot like `tryLock`'s. One major difference is that it has the dark circle wrapping its code paths. This indicates that we are considering it as a private function. We'll discuss this a bit more, but for now, here is a working idea of "private" functions in JavaScript: in JavaScript, there

isn't really a good way to make private functions. Basically, you have variables and functions that are either in scope and addressable, or not.

Something else might have jumped out of the diagram as interesting. The `diary` object (that is used for `diary.open`) is not the function's `this`, nor is it an explicit input: it is a nonlocal input. The mechanism here is a bit tricky, but this should illustrate what's happening:

```
function hi(){
  console.log(hello);
};
hi();

// ReferenceError: hello is not defined

var hello = "hi";
hi();
// logs "hi"
```

It seems weird that `diary` is in scope, and it might seem like it has something to do with the function it is declared inside of being assigned to `diary`:

```
var diary = (function(){
  // does everyone in here know about diary?
```

But actually it works just like the `hi` function. When `privateUnlock` is declared, it doesn't know what `diary` is yet, but that doesn't matter. Once `diary` is declared in the top-level scope, *everything* knows about it, including previously declared functions, which includes `privateUnlock`. This might still seem magical, but basically, you can declare nonlocal inputs to functions *after those functions are declared*. As long as the nonlocal inputs are in a scope that the function can access *when the functions are called*, you can still use them in the function declarations.

If that doesn't sink in, it's okay. We're about to stop using `diary` in that function because it's a little awkward (also, it is hardcoded and will break if we change the variable name).

It is tempting to just expose the `privateUnlock` function to the object (adding another attribute to the returned object), but we won't be able to keep it "private" (out of a directly addressable scope) that way.

To get around the awkwardness of repeating a name as we are doing with `diary`, some people's first instinct is to pass along `this` with a variable called `that`:

```
var diary = (function(){
  var key = 12345;
```

```
    var secrets = 'programming is just syntactic sugar for labor';

    function privateUnlock(keyAttempt, that){
      if(key===keyAttempt){
        console.log('unlocked');
        that.open = true;
      }else{
        console.log('no');
      }
    };

    function privateTryLock(keyAttempt){
      privateUnlock(keyAttempt, this);
    };

    function privateRead(){
      if(this.open){
        console.log(secrets);
      }else{
        console.log('no');
      }
    }

    return {
      open: false,
      read: privateRead,
      tryLock: privateTryLock
    }

  })();
```

Let's see what that does to our diagram for `privateUnlock` (see **Figure 5-21**).

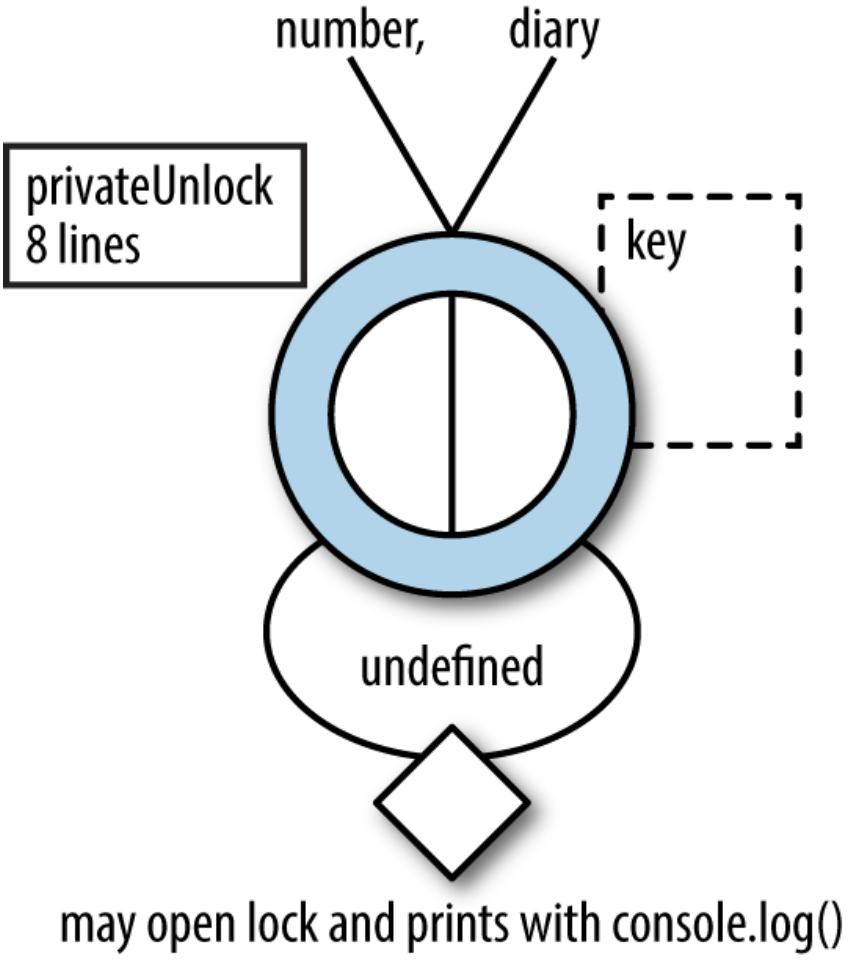


FIGURE 5-21

With two explicit inputs

Nothing much has changed here. The only difference is that now we have two explicit inputs to the `privateUnlock` function and one nonlocal input, which is an improvement from before.

ISN'T CONSOLE A NONLOCAL INPUT?

In other words: shouldn't it be listed on the right of the diagrams too?

When it is used within the function, yes, it does act as a nonlocal input. We've omitted it in these diagrams to simplify them, but by all means, when you're writing your own functions, add entries for `console` and anything else that's not `this` or passed in as an explicit parameter.

Also realize that we're not adding nonlocal inputs for every global object and subobject that we're *not* using. That would make the diagrams very noisy.

Alternatively, we can use one of our `this`-fixing functions: `call`, `apply`, or `bind`. For `call`, you would change `privateUnlock` and `privateTryLock` like this:

```
var diary = (function(){
  var key = 12345;
  var secrets = 'sitting for 8 hrs/day straight considered harmful';

  function privateUnlock(keyAttempt){
    if(key===keyAttempt){
      console.log('unlocked');
      this.open = true;
    }else{
      console.log('no');
    }
  };

  function privateTryLock(keyAttempt){
    privateUnlock.call(this, keyAttempt);
  };

  function privateRead(){
    if(this.open){
      console.log(secrets);
    }else{
      console.log('no');
    }
  }

  return {
    open: false,
    read: privateRead,
    tryLock: privateTryLock
  };

})();
```

And in the `bind` version our `privateTryLock` function would look like this:

```
function privateTryLock(keyAttempt){
  var boundUnlock = privateUnlock.bind(this);
  boundUnlock(keyAttempt);
};
```

Or we could inline that `boundUnlock` variable by calling the bound function right away:

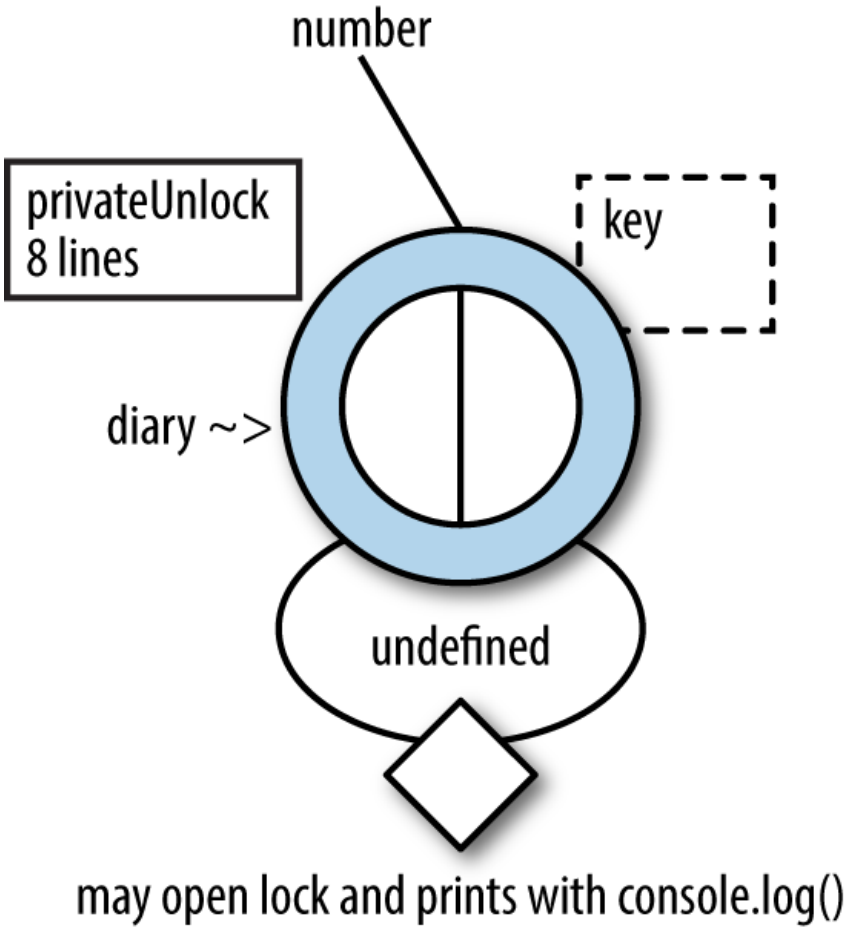
```
function privateTryLock(keyAttempt){
  privateUnlock.bind(this)(keyAttempt);
};
```

which puts us back to being pretty similar to the `call` syntax.

In any case, our diagram for `privateUnlock` with the `this`-fixing in place shouldn't be too shocking (see **Figure 5-22**).

FIGURE 5-22

One explicit parameter (number) and one implicit parameter (diary)



Now our function has an implicit input of `diary`. `key` is still stuck as a nonlocal. It, like `secrets` and `privateUnlock`, is there for anyone to grab up when the `diary`-creating anonymous function runs, but it's not attached to any object (any `this`) of significance.

Some variables (including functions) have a useful `this` that they are attached to. Others just have a scope where they are available and addressable.

Before leaving our `diary` example, there's one important function that we've neglected to diagram: our `diary`-creating function (**Figure 5-23**).

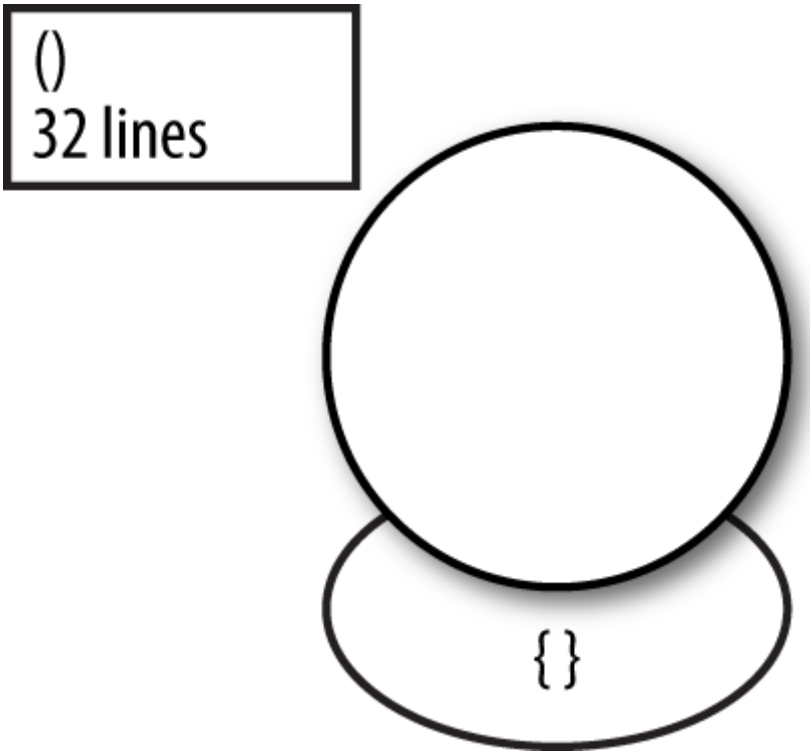


FIGURE 5-23

The diary-creating function is pretty simple

It's actually what's not in this diagram that is surprising. First of all, the function is anonymous. It is the result of calling the function that is assigned to a variable that happens to be called `diary`:

```
var diary = (function(){
```

Similarly, the return type is an object: `{}`. We could get specific and say it returns an object with particular properties, or we could say it's a `diary` object. However, it's worth highlighting that our function has no idea of what a `diary` is until after its result is assigned to the variable.

At this point, you might be wondering about classes. Maybe classes have some magical way to implement private methods? Nope. There have been proposals to ECMAScript petitioning for such things, but as of this writing, they're still not a sure bet.

If we were really insistent on this behavior for classes, how might we write it?

```
class Diary {
  constructor(){
```

```
    this.open = false;
    this._key = 12345;
    this._secrets = 'the average human lives around 1000 months';
  };

  _unlock(keyAttempt){
    if(this._key===keyAttempt){
      console.log('unlocked');
      this.open = true;
    }else{
      console.log('no')
    }
  };
  tryLock(keyAttempt){
    this._unlock(keyAttempt);
  };

  read(){
    if(this.open){
      console.log(this._secrets);
    }else{
      console.log('no');
    }
  }
}
d = new Diary();
d.tryLock(12345);
d.read();
```

Now our private variables and `_unlock` function are exposed in the class. Also, we've prepended an underscore to indicate functions and variables that shouldn't be accessed directly. If we consider our private/underscore function to be a private implementation detail and therefore to not require testing, we now have a visual cue to help us convey that to others and our future selves. On the other hand, in this form, our tests should be very easy to write because all of our private methods are still addressable.

However, if we genuinely wanted to not expose our "hidden" information, we've failed. Let's take a step in what looks like the wrong direction:

```
var key = 12345;
var secrets = 'rectangles are popular with people, but not nature';
function globalUnlock(keyAttempt){
  if(key===keyAttempt){
    console.log('unlocked')
    this.open = true;
  }else{
    console.log('no')
  }
```

```
};

class Diary {
  constructor(){
    this.open = false;
  };
  tryLock(keyAttempt){
    globalUnlock.bind(this)(keyAttempt);
  };
  read(){
    if(this.open){
      console.log(secrets);
    }else{
      console.log('no')
    }
  }
};
d = new Diary();
d.tryLock(12345);
d.read();
```

Now our hidden information is outside of our class. In fact (assuming we're at the top-level scope), we've created global variables! What good is that?

Well, this is actually very close to something great that solves our problem in a different way. Save the following as *diary_module.js*:

```
var key = 12345;
var secrets='how to win friends/influence people is for psychopaths';
function globalUnlock(keyAttempt){
  if(key===keyAttempt){
    console.log('unlocked')
    this.open = true;
  }else{
    console.log('no')
  }
};

module.exports = class Diary {
  constructor(){
    this.open = false;
  };

  tryLock(keyAttempt){
    globalUnlock.bind(this)(keyAttempt);
  };

  read(){
    if(this.open){
      console.log(secrets);
```

```
    }else{
      console.log('no')
    }
  }
}
```

The only line we've changed is:

```
module.exports = class Diary {
```

To make use of this, we'll need another file (you can call it *diary_reader.js*) to import the module. Here's what that file looks like:

```
const Diary = require('./diary_module.js');
let d = new Diary();
d.tryLock(12345);
d.read();
```

In *this* file, between the `Diary` "class" or the "instance" `d`, we're not able to see the `key` or read the diary `secrets` without it. Unfortunately, this also means that if we want to test it using a similar `require` mechanism, we're stuck either going back to the prepended underscore technique, putting our private functions in their own modules somehow, or conditionally including them for tests (and excluding them otherwise).

Is There Privacy in JavaScript?

As of this writing, there is no real privacy in JavaScript. Things are either in scope, or they're not. Unfortunately, having an object and simply declaring some attributes as public and others as private is not really possible. Because every attribute (every property of an object) has a "this" that it attaches to, for functions to be "private" in JavaScript, they are necessarily also inaccessible.

So, practically speaking, at this time we have two conventions to choose from. The first is to give up on that dream and let attributes attach to some other `this`, with an eye toward the subconventions of doing so in a wrapping anonymous function (à la the revealing module pattern), or allowing `this` to attach to the global `this` for modules that are exported. Because exporting is a whitelisting operation, only the functions we specify will be imported by other scripts. This is handy for having a smaller API, but does complicate testing somewhat.

The second (admittedly clunky) convention is to let our functions happily bind to the same `this` as public members, but give visual cues (prefixing the

function name with an _ is the most common) to indicate when something is *intended* to be private.

So for now, the meaning of *private* is more or less up to you, as well as how you decide that which you term "private."

However, there are proposals from TC39 (the committee that evaluates and determines JavaScript features) related to privacy that are in the works. These include:

- **Private fields** (https://github.com/tc39/proposal-private-fields)
- **Private methods** (http://github.com/tc39/proposal-private-fields/blob/master/METHODS.md)
- **Private static fields and methods** (http://github.com/tc39/proposal-private-fields/blob/master/STATIC.md)

In this book, we're doing the same thing with these additions as we're doing with the likely forthcoming `async` and `await` features: we're mentioning them, but not going into detail. From the spec, this is the proposed new syntax as of this writing:

```
class Foo {
  #a;
  #b;
  #sum() { return #a + #b; }
  printSum() { console.log(#sum()); }
  constructor(a, b) { #a = a; #b = b; }
};
```

So, `#a` and `#b` are both private (nonfunction/method) "fields," and `#sum` is a private "method." These identifiers with a `#` would be unavailable outside of the `class` context block. Thus, a new `foo` instance of class `Foo` wouldn't have traditional property-style access à la `foo.#a` (or maybe `foo.a`—we don't know, and we're assuming that wouldn't work anyway), `foo.#b`, or `foo.#sum()`. However, `foo.printSum()` would be fine, since that's not private. Beyond that, details are sketchy at the moment (e.g., does `foo.#a` throw an error? Can `a` and `#a` both be used as field names? Is there a workaround to access the private fields and methods for testing?).

"METHODS" VERSUS "FUNCTIONS"

Throughout this book, we mostly refer to JavaScript functions as being *functions*. For some people, that implies functions in the sense of explicit inputs and output, whereas a *method* is used to mean some procedure or "just a chunk of code" that is attached to an object.

In any case, even without all of the facts in, the proposals for private fields and methods have implications for the future of JS generally:

- Classes are getting more features that make unique constructs and not just "syntactic sugar" for constructor functions.
- JavaScript is doubling down on OOP. Functional programming might be the "future" of JavaScript, but OOP is at least *also* the future.
- Choosing "your JavaScript" will likely again be proven to be a function of time. The JavaScript of five years ago is starting to look weird to modern eyes. It's likely that trend will continue.
- If there is no workaround provided by the *`#privateFunction`* syntax, you might see people still preferring the *`_privateFunction`* underscore hack for backward compatibility and in cases where testing private functions is desired.

Wrapping Up

In this chapter, we covered a lot of detail about how JavaScript works. We centered our conversation on functions, as they are the most important and complicated construct in JavaScript (in any paradigm worth pursuing).

But as the goal was to be prescriptive as well as descriptive, here are some takeaways worth repeating:

- Try to keep bulk (complexity and lines of code) low.
- Try to keep the total number of inputs low.
- Prefer explicit inputs to nonlocal inputs.
- Make choices between passing `this` explicitly versus binding it to the function call, rather than hardcoding object names as nonlocal inputs.
- Prefer real, meaningful return values to side effects.
- Keep side effects to a minimum or nonexistent.
- Have a well-defined `this` when possible for functions and other variables (attributes) by making them part of classes (or at least objects) to cut down on nonlocal inputs and global variables.
- In JavaScript, mechanisms for privacy necessarily impact access, which can complicate code, especially when it comes to testing.

You should find these ideas help to make your code (as well as the diagrams) simpler.

As a final note, the style that we used here is "`this`-friendly" as compared with some other approaches. In particular, using objects that change their values conflicts with aspects of the functional style we'll discuss in **Chapter 11**. In object-oriented (class- or prototype-based) style, however, you will use `this` frequently.

Refactoring Simple Structures 6

Over the next two chapters, we will be dealing with code that is messy, undertested, and does something cool. Note that both of these chapters work from the same project codebase.

The first part that is cool is that if you have interest, but lack experience, in machine learning, we're using a particular algorithm is fairly simple and still very powerful. It's called a Naive Bayes Classifier (NBC). You can use it to classify things based on previous knowledge. A spam filter is a frequently cited example. An NBC has two basic steps. First, you give it data that you already know how a human would classify (e.g., "These 35 subject lines are from spam emails"). That is called "training" the algorithm. Then, you give it a new piece of data and ask it what category that data likely fits into (e.g., "Here is the subject line of an email we just received. Is it spam or not?").

The second cool thing (if you're into playing music at all) is that our specific application of the algorithm will use chords in songs along with their difficulty as training data. Following that, we can feed it the chords of other songs, and it will automatically characterize its difficulty for us. At the end of these two chapters, we'll make some tweaks to have the algorithm guess at whether a segment of text is understandable or not (with the assumption that we understand English, but not Japanese).

This might seem like an intimidating or complex problem, but two things will keep us afloat. First, the basis of the entire program is just multiplying and comparing different sets of numbers. Second, we can rely on our abilities to test and refactor to get us through, even if some details don't intuitively make sense at first.

BUT I DON'T KNOW ANYTHING ABOUT MUSIC!

That's okay. This won't be technical as far as music goes. All you need to know is that to play songs (on guitar, for example), you usually need to know some chords, which are a bunch of notes put together.

If you plucked a guitar string or hit a piano key, you'd be playing a note. And if you strum multiple strings or hit multiple piano keys (playing multiple notes at once), you'd be playing a chord. Some chords are harder to play than others.

Music can be complicated, but for our purposes, songs are simply made up of chords, and how difficult the chords are to play determines how difficult it is to play the song overall.

There are a few things we won't be covering a great deal. Linters (discussed in **Chapter 3**) like JSHint and JSCS (now ESLint) checker criteria, of which there are hundreds, will not all be specifically addressed. However, it is recommended that you try these tools out in your editor. More specifically, we won't be covering single quotes versus double quotes, ASI (automatic semicolon insertion) and "unnecessary" semicolons, or the number of spaces between braces and the object values that live inside of them.

Those details are nice to have consistency on, so by all means, use a linter as a "living style guide," but covering all the tiny things that they catch here, some of which are fairly opinionated (e.g., x===3 should have spaces and instead be x === 3), would be neither interesting nor useful to someone using modern tools and already using some determined form of JavaScript (see **Chapter 2**) that they want to write.

WHY NOT JUST CARS, BANK ACCOUNTS, AND EMPLOYEES?

We aren't talking about cars, employees, or bank accounts? Not only are those examples as overused in tech books as the quote from Spiderman's Uncle Ben, but also consider this passage from *Gradus Ad Parnassum*:

> Perhaps the hope of future riches and possessions induces you to choose this life? If this is the case, believe me you must change your mind; not Plutus but Apollo rules Parnassus. Whoever wants riches must take another path.

Aloys is talking about music theory, not programming. Nonetheless, economic acquisitions forming the basis of craft and knowledge (unless that craft and knowledge is economics, I guess) just feels wrong.

You can use an NBC to help you learn a new language, study an instrument, or figure out how best to entertain yourself through media. Let Plutus have your 9 to 5 if he must, but we're all about the Apollo here.

DON'T FORGET THE ADVICE FROM THE LAST CHAPTER

- Try to keep bulk (complexity and lines of code) low.
- Prefer explicit inputs to implicit inputs (although we're working in OOP style, so we're favoring implicit inputs in this chapter; see **Chapter 11** for this code in FP style).
- Prefer implicit inputs to nonlocal inputs (free variables).
- Prefer real, meaningful return values to side effects.
- Keep side effects to a minimum.
- Have a well-defined `this` when possible for functions and other variables (attributes) by making them part of classes (or at least objects) to cut down on nonlocal inputs and global variables.
- In JavaScript, mechanisms for privacy necessarily impact access, which can complicate code, especially when it comes to testing.

The Code

Here's the NBC, the initial "bad version" that we're going to improve upon throughout the next two chapters:

```
fs = require('fs');
// songs
imagine = ['c', 'cmaj7', 'f', 'am', 'dm', 'g', 'e7'];
somewhere_over_the_rainbow = ['c', 'em', 'f', 'g', 'am'];
tooManyCooks = ['c', 'g', 'f'];
iWillFollowYouIntoTheDark = ['f', 'dm', 'bb', 'c', 'a', 'bbm'];
babyOneMoreTime = ['cm', 'g', 'bb', 'eb', 'fm', 'ab'];
creep = ['g', 'gsus4', 'b', 'bsus4', 'c', 'cmsus4', 'cm6'];
army = ['ab', 'ebm7', 'dbadd9', 'fm7', 'bbm', 'abmaj7', 'ebm'];
paperBag = ['bm7', 'e', 'c', 'g', 'b7', 'f', 'em', 'a', 'cmaj7',
            'em7', 'a7', 'f7', 'b'];
toxic = ['cm', 'eb', 'g', 'cdim', 'eb7', 'd7', 'db7', 'ab', 'gmaj7',
         'g7'];
bulletproof = ['d#m', 'g#', 'b', 'f#', 'g#m', 'c#'];
song_11 = [];

var songs = [];
var labels = [];
var allChords = [];
var labelCounts = [];
var labelProbabilities = [];
```

```
var chordCountsInLabels = {};
var probabilityOfChordsInLabels = {};

function train(chords, label){
  songs.push([label, chords]);
  labels.push(label);
  for (var i = 0; i < chords.length; i++){
    if(!allChords.includes(chords[i])){
      allChords.push(chords[i]);
    }
  }
  if(!!(Object.keys(labelCounts).includes(label))){
    labelCounts[label] = labelCounts[label] + 1;
  } else {
    labelCounts[label] = 1;
  }
};

function getNumberOfSongs(){
   return songs.length;
};

function setLabelProbabilities(){
  Object.keys(labelCounts).forEach(function(label){
    var numberOfSongs = getNumberOfSongs();
    labelProbabilities[label] = labelCounts[label] / numberOfSongs;
  });
};

function setChordCountsInLabels(){
  songs.forEach(function(i){
    if(chordCountsInLabels[i[0]] === undefined){
      chordCountsInLabels[i[0]] = {};
    }
    i[1].forEach(function(j){
      if(chordCountsInLabels[i[0]][j] > 0){
        chordCountsInLabels[i[0]][j] =
chordCountsInLabels[i[0]][j] + 1;
      } else {
        chordCountsInLabels[i[0]][j] = 1;
      }
    });
  });
}

function setProbabilityOfChordsInLabels(){
  probabilityOfChordsInLabels = chordCountsInLabels;
  Object.keys(probabilityOfChordsInLabels).forEach(function(i){
    Object.keys(probabilityOfChordsInLabels[i]).forEach(function(j){
      probabilityOfChordsInLabels[i][j] =
```

```
probabilityOfChordsInLabels[i][j] * 1.0 / songs.length;
    });
  });
}

train(imagine, 'easy');
train(somewhere_over_the_rainbow, 'easy');
train(tooManyCooks, 'easy');
train(iWillFollowYouIntoTheDark, 'medium');
train(babyOneMoreTime, 'medium');
train(creep, 'medium');
train(paperBag, 'hard');
train(toxic, 'hard');
train(bulletproof, 'hard');

setLabelProbabilities();
setChordCountsInLabels();
setProbabilityOfChordsInLabels();



function classify(chords){
  var ttal = labelProbabilities;
  console.log(ttal);
  var classified = {};
  Object.keys(ttal).forEach(function(obj){
    var first = labelProbabilities[obj] + 1.01;
    chords.forEach(function(chord){
      var probabilityOfChordInLabel =
probabilityOfChordsInLabels[obj][chord];
      if(probabilityOfChordInLabel === undefined){
        first + 1.01;
      } else {
        first = first * (probabilityOfChordInLabel + 1.01);
      }
    });
    classified[obj] = first;
  });
  console.log(classified);
};

classify(['d', 'g', 'e', 'dm']);
classify(['f#m7', 'a', 'dadd9', 'dmaj7', 'bm', 'bm7', 'd', 'f#m']);
```

What does it do?

Honestly, what we have here is a little over 100 lines of fairly incomprehensible stuff. Although we could try to break it down, or look to the mathematical model of NBCs first, that is not our approach here.

We will be developing confidence in this code through testing and refactoring.

In general, we want to get code into a file (and also isolated in a *.js* file if it was started as part of `<script>` tag inside an *.html* file), then under version control, and then we'll decide on a testing strategy.

LINE LENGTH

To ensure that everything in this chapter fits on the page, some lines have been aggressively split onto two in a seemingly unnatural way:

```
      var probabilityOfChordInLabel =
probabilityOfChordsInLabels[obj][chord];
```

For many situations, there are better places than the first column to continue a line. In the case of assignments, in such a limited space, we'll tend to continue in this way when we have to. Normally, you would hit a maximum desirable line length with a long array, string, or other data.

For this and the following chapter, we will assume that you have a file called *nb.js* with the code just listed.

Our Strategy for Confidence

Assuming you're all set up with the code from earlier, and have it saved to a file called *nb.js* that is under version control, let's get started.

Back to our model of what types of tests to write when (the diagram in **Chapter 4**), we know that we need characterization tests for untested code. But looking at our file, we discover with horror that none of our functions return anything: we have a bit of structure, but our functions just group lines of statements together and generate side effects. Inputs come in the form of variables defined at the top-level scope, and variable reassignments run rampant.

Yikes. So what's our first course of attack? Run the file with node (**`node nb.js`**), and we should see the following output:

```
(from command line)
>node nb.js
[ easy: 0.3333333333333333,
  medium: 0.3333333333333333,
  hard: 0.3333333333333333 ]
{ easy: 2.023094827160494,
  medium: 1.855758613168724,
  hard: 1.855758613168724 }
[ easy: 0.3333333333333333,
  medium: 0.3333333333333333,
```

```
  hard: 0.3333333333333333 ]
{ easy: 1.3433333333333333,
  medium: 1.5060259259259259,
  hard: 1.6884223991769547 }
```

DID YOU GET AN ERROR?

Specifically this one?

```
TypeError: allChords.includes is not a function
```

If so, you probably have an old node version. You should have at least version 6.70. Go to *nodejs.org* and download a current version.

So print statements are our only output, which is not great. The good news is that we have running code, and it actually outputs something. That means that we actually have a test in place. Unfortunately, it is a manual one, but it doesn't require much setup (we don't need a test library to run **`node nb.js`** and see output). If we define our "public interface" to be the whole program, run as it is with no further options, we could be satisfied that this is the only test we may ever need.

Long term, this is not a great approach, and we bring in formal testing in the next chapter. For this chapter, we'll limp along with checking the output through our logging statements.

THESE RESULTS STOP ONE STEP BEFORE ACTUAL CLASSIFICATION

In these results, we can see that we get numbers for each type of value (`easy`, `medium`, and `hard`). Where they are all the same (`0.333...`), what is printed is the base likelihood of classifying as that value simply based on the ratio of other samples falling into that value—or "category," if that's an easier way to think about it.

As for the ones with different values, the label corresponding to the highest number in the set of values (`easy`, `medium`, or `hard`) reflects the category in which the classified data most strongly fits. In other words, this means the following data would classify as `hard`:

```
{ easy: 1.3433333333333333,
  medium: 1.5060259259259259,
  hard: 1.6884223991769547 }
```

If our classifier was used as a spam filter, this message would most likely be spam:

```
{ nonSpam: 3,
  spam: 8 }
```

It's trivial to pull out the highest number and report the matching key, but that is the final step of this algorithm. If you want to write that code, and tests to go with it, feel free. We avoid it in this chapter because the numbers are more explicit, giving us more certainty that we haven't changed our algorithm.

Back to the question of confidence (through the result of our manual test), we need one more step: run it again (maybe two to five times). And the result is still the same. That's great. Our algorithm is most likely *deterministic*. A word of warning here is that before making this assumption, you should check the code for `Math.random`, `new Date`, and other sources of variation (including calls to remote URLs) in the program before assuming it will always behave the same way.

So, now that we're reasonably confident in the behavior, how should we start refactoring this?

For this code, we are taking a broad view of refactoring. In practice, you might find another order more logical than what is presented here. In particular, after getting tests in place, it is often easiest to extract functions before doing anything else.

Extracting functions is often the best way to reveal the structure of a program, and is probably the most underutilized of refactoring techniques. However, we will wait until the next chapter to introduce it.

We delay extracting functions for three reasons:

- These two chapters are roughly arranged from simple to complex techniques, and it's good to understand the simple ones first.
- Extracting functions often obviates other approaches. If we covered it first, some simpler (but still important) methods would become unnecessary.
- Additionally, when extracting functions, you often discover something else that could be done at the same time—for example, renaming a variable. To keep techniques isolated, we need to move in small steps first.

Generally speaking, the techniques presented in this chapter are low-level. In other words, they are concerned with small pieces of code. They are more likely to be things that a linter could find or a text editor could "automatically refactor."

Renaming Things

The easiest refactorings to do are simply renaming things that don't make sense; whether it's variables, functions, objects, or modules, this is a good place to start. In the worst case these break the program, and we **`git checkout .`** to get back to our previous good version.

On our hunt for bad names, we have a few things to look for:

- Misspelled words
- Short names (abbreviations and one-letter names)
- Nondescriptive/general names
- Numbers in variable names
- Doubled-up names
- Not CapitalCase for constructors
- Not camelCase for functions and variables

And consider that these may all apply to the following:

- Variables
- Loop variables
- Functions
- Objects
- Classes
- Parameters
- Files
- Directories
- Modules
- Projects

Does anything look obviously misspelled? Does not seem so, but the short name `ttal` stands out as not a real word. It could be a misspelling, or an intentional but misguided abbreviation for `total`. For now, do a search and replace that turns `ttal` into `total`.

Run the program (our test). Same output? Great. Save and **`git commit -am '`*`fixed bad variable name`*`'`** and let's move on.

We have a variable name including a number and bucking the camelConvention: `song_11`, which is just an empty array. To avoid numbers and snake_convention (using underscores), we could call this `songEleven`, but `blankSong` seems more fitting and specific. Save, run the file, and (assuming everything looks good) commit the changes.

Next, we have a variable name that follows the snake_convention, rather than the camelConvention: `somewhere_over_the_rainbow`. The fix is the

same: search/replace/save/verify/commit. Make sure you find both instances of it.

Next is a tougher case, but using single-letter variable names doesn't make sense for most JavaScript source code (although these are normal in compiled JS, as build processes may shorten labels). `i` and `j` appear throughout the program, with their one-letter variable name signaling that they are just an index, not requiring a full and descriptive name. This is nonsense. But why?

First, because these names are more resistant to change. If you apply the same workflow as we have been using to rename variables to single-letter variable names, you are very likely to break something. Find/replace should not have to require much focus, so when you want "only `i` if it's *not* part of a bigger word/starts with var/has parens around it, etc.", you will quickly find yourself in a more manual search-and-replace process than you want. This is complicated further when you have to search across files. Although a lot of editors let you search by regex, it's an extra step, and regex won't solve everything if the variable names are reused in various places.

Second, although these variables happen to be in the expected scopes, having variables that are not unique creates a risk of them overwriting each other. What makes this worse is that this might be expected or relied-upon behavior, so we don't know ahead of time whether changing one of these variable names, will introduce a bug.

The third and worst thing about these names is that they give no information about what is inside. It is somewhat conventional (though a bad convention, as just described) to use `i` and `j` as indices inside of loops. In those cases, the names `index` and `innerIndex` are more appropriate when the variables represent *numerical* keys. Change `i` to `index` in the `train` function (there should be five of them). Save, check, and commit.

ALTERNATIVE OPINION: DESCRIPTIVE VARIABLE NAMES INDICATE BAD CODE

This is a nuanced view, and doesn't apply to an easily attainable style in JavaScript, but in some languages (including some that compile to JavaScript), it is possible to specify much about a program through its *type system*. In those cases, descriptive variable names may actually detract from the clarity of the possibilities provided by the *type signatures* of functions.

To illustrate the point, imagine we have a function that we know takes a list/array of *somethings* as input and returns one *something* as output. We could describe these somethings as "numbers" or "strings" or another "type" as it applies to our case, but if we trust the type system to transform a list of them into just one, we might

prefer not to describe what kind of *something* it is. In those cases, some prefer to use an `x`, `y`, `a`, or `b`, rather than a longer "descriptive" name that does nothing to describe the important part (e.g., the transformation from the list of *somethings* into just one *something*).

We'll get into functional programming more in **Chapter 11**, but even in a functional context, you might not find single-letter descriptors compelling. That's okay. In this book, we try to avoid that style, but it's worth understanding the justification for the contrarian view on this.

When we apply the pattern of thinking of `i` and `j` as indices for an argument in a `forEach` function, then we've really obscured their values. In the `setChordCountsInLabels` function, `i` should actually be `song` (eight replacements), and each instance of `j` in that function should be `chord` (five replacements).

FLIPPING BACK FIVE PAGES TO LOOK AT CODE? THIS IS AN OUTRAGE!

Dear reader, I highly recommend at this point that if you haven't already, you become dear *writer*. If you aren't using your editor to add code and make changes, the code in this and the next chapter will be very hard to follow. In addition to just being good practice (writing code rather than just reading it), the code in this chapter and the next is quite long compared to that found in other chapters.

As written in the preface, it's your book, so you can treat it how you wish, but actually making the changes described, running the code, checking into version control, and writing tests are all part of the skills you should develop through the book.

`setProbabilityOfChordsInLabels` has `i` and `j` variables as well. In this case, `i` is more appropriately called `difficulty` (four replacements) and `j` is better named as `chord` (three replacements). Make those changes, save, check the results, and commit.

For those last two functions, `i` and `j` did not represent *numerical* keys (array indexes), but rather were *string* keys of an object. If you weren't sure of more appropriate names, a quick `console.log(i)` or `console.log(j)` statement inside of the loop will reveal the values, and hint at a more appropriate name.

This bears reiterating: if you discover true numerical array indexes, the name `index` (and `innerIndex` where necessary) is still preferable to single-letter variable names. If what you discover are string keys to an object (e.g., `"easy"`, `"medium"`, and `"hard"` in our case for the `i` we renamed as `difficulty`), you should rely on domain knowledge to chose an appropriate name. In those ca-

ses, don't worry if your first guess is imprecise. As you gain domain knowledge and confidence in how the program works, you can always rename things again later.

RENAMING THINGS CAN BE A BIG DEAL

When changing labels, be sure that you change everywhere that needs to be changed. With our single file program, that means every case in the file. With multiple files, lean on your editor/IDE/command line (ack, grep, etc.) to help you find all of the instances.

Also, be sure to give sufficient notice (via deprecation warnings) and allow support for old versions if your code is a module or package external to your project and relied upon by others, although this may be unnecessary for internal values.

The code is a bit clearer after our renamings, but one object should still stick out as having a short and generic name: `obj`.

With our method of refactoring first and understanding after, we don't have a great idea of what `obj` could mean. Let's take a look at the `classify` function and see if we can find some hints.

```
function classify(chords){
  var total = labelProbabilities;
  console.log(total);
  var classified = {};
  Object.keys(total).forEach(function(difficulty){
    var first = labelProbabilities[difficulty] + 1.01;
    chords.forEach(function(chord){
      var probabilityOfChordInLabel =
probabilityOfChordsInLabels[difficulty][chord];
      if(probabilityOfChordInLabel === undefined){
        first + 1.01;
      } else {
        first = first * (probabilityOfChordInLabel + 1.01);
      }
    });
    classified[difficulty] = first;
  });
  console.log(classified);
};
```

We could try to reason it out, but if instead we cheat a little bit, and add a `console.log(obj)` after the fifth line (then run with **`node nb.js`**), we'll see that "easy," "medium," and "hard" are now printed (in addition to the old output) when we run the program. In some parts of our program, we've been call-

ing these labels, but have just renamed a similar concept as `difficulty`. Did we make a mistake?

It's not always this easy to find appropriate names for things without a full understanding of the program. `label` is a name that is more relevant to the algorithm (NBC), but `difficulty` is more specific to the problem domain (learning to play songs). For now let's change `obj` to `difficulty` (four replacements not including the logging statement). Keep in mind that changing all instances of `obj` is easy in this case because the name `obj` is confined not only to this file, but also this function. Feel free to delete the logging statement if you added one.

It is worth considering at this point if adopting the terminology of "difficulty" rather than "label" would make sense for associated variable and function names across the entire program. However, that is a bit more complex, as there are a few dozen names that employ that terminology. If you are feeling confident enough to make those changes (the "test" of running the program will cover you, after all), you can do so, but we'll proceed assuming that those names have not been changed. For that reason, it might be worth waiting until you complete this chapter along with the next one before making those changes.

Useless Code

Next up is useless code, and the bottom line is that if you don't need it, get rid of it. If you remember the term YAGNI ("Ya ain't gonna need it") from **Chapter 1**, that is what we're covering in this section.

Here are the forms you might encounter useless code in:

- Dead code (variables, functions, files, modules, etc.)
- Speculative code and comments
- Whitespace (including EOL and EOF)
- Do-nothing code (reachable but has no effect, e.g., `$($('.someClass'))` in jQuery or `if(!!booleans)`; empty files)
- Debugging/logging statements

Dead Code

How do you find dead code? Look for just one instance (project-wide) of a function or variable name. If there's just a function declaration that isn't called anywhere, we can happily delete that function. The same goes for variables that aren't used. Keep in mind that this would be harder to ensure if our program went beyond one file. Make sure you have a good way to search through a whole project either on the command line or through your editor.

Can you find any instances of dead code in our NBC?

There are actually three variables that we can eliminate. First up, neither `army` nor `blankSong` is used as training data (or anywhere else), so the lines with those variable declarations can be safely deleted. A save/run/check/commit cycle shows us that we haven't broken anything (the result is the same).

Speculative Code and Comments

Sometimes you'll see comments that are intended as future code (a stub, pseudocode, or a full implementation). This is the deadest of dead code, and any details of what code *should* be there are best left to some task management system that shares to-dos (and more formally, tickets/tasks/bugs) with the team. The codebase is for real, running code. Speculative code, commented-out or not, violates the YAGNI principle. If the code reflects not only its functionality, but somehow all of its potential, it is due to be pruned. An additional danger with commented code is that one might assume that it actually should work if uncommented. It may work or not. If it is not exercised by tests or even running with the rest of the code, it should not be trusted.

We can delete our first line: `fs = require('fs')`

Apparently, there was an intention to include and make use of the filesystem module at some point, but it was never realized. If you are not actively working on some filesystem-based feature, then this should go. If you see something like this, especially with an accompanying comment:

```
// use the file system for *something* later
fs = require('fs')
```

It is dead code of a particular type: speculative code. Maybe `army` and `blankSong` were speculative as well. The difference is intention, which is hard to tell without comments, supporting tests, or domain knowledge. You might also say they're not dead code because they do something (assign a couple of variables) whereas "real" dead code is unreachable/impossible to execute. For our purposes here, it doesn't matter. We treat all those cases the same: delete the code.

Don't just comment it out. Any reasonably complex project should have some way of keeping track of future intentions (a bug list, feature tickets or "user stories," etc.). If you comment it out, your code is taking on extra work that it shouldn't.

There are two useful approaches to problematic comments. The first is simply to delete them. The second is to use them as inspiration for creating a variable or function. Our second line, `//songs`, is a candidate for creating a vari-

able or function (*explaining comments* often indicate a good place to extract a function or variable), but in this section, we're just covering that which can be deleted. Feel free to delete this line for now, and save/run/check/commit the code.

DOCUMENTATION: WHEN COMMENTS ARE USEFUL

In any code of sufficient complexity and likelihood of being used by others, comments can be useful as documentation. These typically precede functions and classes (or objects) and describe what the function does, as well as the explicit parameters and return type. They may even be responsible for helping to build external documentation. Obviously, we don't want to delete those comments in the source files (compiled/minified files should strip these to make files smaller).

ReadMes and tutorials can serve as a type of documentation as well, but high-quality descriptions of how code works, living as comments in the source files, are especially useful.

Another interesting use of comments on the front end is to send secret messages to those who would "view the source." Usually this involves ASCII art and "Hey, developers! Work for us!"-type notes.

Whitespace

The whitespace in our code seems okay for the most part. Before the `classify` function, there are three blank lines, two of which can and should be deleted. Beyond extra blank lines, you may see trailing whitespace at the end of a line (EOL). This shows up in a different (and usually distracting) color in some editors, and doesn't in others. How much you dislike this trailing, meaningless whitespace probably has to do with what editor (and settings for that editor) you use. You should feel free to delete it in most cases, but it will make your **`git diff`** (the set of changes to a project) potentially much larger, a distraction of a different type.

Another, somewhat editor-specific, whitespace instance comes from a blank line at the end of a file (EOF). Generally, this seems like a good idea (i.e., it follows the IEEE POSIX standard for what a "file" is), but can lead to version control noise/conflicts if two developers have different editors or personal preferences (as in the previous paragraph).

Do-Nothing Code

Moving on to do-nothing code, we have an example of this in our file. The conditional check that follows contains an unnecessary part:

```
if(!!(Object.keys(labelCounts).includes(label))){
```

Specifically, the `!!` can go away, leaving:

```
if(Object.keys(labelCounts).includes(label)){
```

Make that change and save/run/check/commit.

Since the `includes` function already returns a boolean, there is no need to use `!!` to cast it to one. In case you're wondering how this works, a *unary* `!` gives returns the inverted "truthiness" of a value. You can try these out in a console if you're curious:

```
!true // returns false
!!true // returns true
!![] // returns true
!!0 // returns false
```

A second reason this `!!` is unneeded here is that, although explicitly setting a boolean might seem to make sense, for any value that is tested in an `if` statement, you can expect the `if` branch to be followed for `true` (and `else` for `false`) values without explicitly coercing the boolean. So the following snippet would print "hi," because nonempty strings are "truthy" in JavaScript.

```
if("print hi"){ console.log('hi')}
```

FALSEY VALUES IN JAVASCRIPT

There are six "falsey" values in JavaScript: `undefined`, `null`, `0`, `""` (the empty string), `NaN`, and `false`. Applying `!!` to these will produce a `false`, whereas other strings, numbers, objects, functions, arrays, and so on will all produce a `true`.

Back to unnecessary code: there is no need to `!!` values in an `if` statement test as we did. If you are looking for a use for the `!!`, one appropriate use would be in front of a return value of a function that you want to return a boolean. Here is a contrived example:

```
function didItWork(){
  return !!numberOfTimesItWorked();
};
```

Here, we have access to a function of the number of times something worked. If it happened 0 times, then we want to return a `false`. If it happened more than that, we want to return a `true`.

One other example from our classifier is related to how JavaScript handles numbers. Specifically, there aren't "integers" and "floats," just numbers. In some languages (such as Ruby), these give different results:

```
10 / 3 # this returns 3
10.0 / 3 # this returns 3.33333...
```

So if you're coming from a language like Ruby, in a calculation that involves an integer values of `10` and `3`, at least one value has to be converted into a float first. You can do this by multiplying one of the terms by `1.0`.

In most JavaScript, *floats* and *integers* are both just *numbers*, so both of the preceding division expressions produce `3.3333...` No explicit conversion is needed. That means that we have another unnecessary bit of code in our classifier. The following line is able to drop the `* 1.0` part:

```
probabilityOfChordsInLabels[difficulty][chord] =
probabilityOfChordsInLabels[difficulty][chord] * 1.0 / songs.length;
```

making it:

```
probabilityOfChordsInLabels[difficulty][chord] =
probabilityOfChordsInLabels[difficulty][chord] / songs.length;
```

Another example of do-nothing code: the `total` variable that we spent a bit of time earlier renaming. It's due for deletion. Why? All it does is receive an assignment and log something. We can just log `labelProbabilities` directly.

Change this section of code:

```
function classify(chords){
  var total = labelProbabilities;
  console.log(total);
  var classified = {};
  Object.keys(total).forEach(function(difficulty){
```

to this:

```
function classify(chords){
  console.log(labelProbabilities);
```

```
  var classified = {};
  Object.keys(labelProbabilities).forEach(function(difficulty){
```

Save/run/check/commit to confirm we haven't changed how the program works.

Where else does unnecessary code pop up? We don't have an example of this in our classifier, but another way code can be useless is by double-wrapping itself like in the following jQuery snippet:

```
$('input').on('click', function(){
  var elementToHide = $(this);
  $(elementToHide).hide();
});
```

The value of `this` in that example gets wrapped into a jQuery object twice. jQuery is smart enough to ignore this, but the second wrapping of `elementToHide` with a dollar sign is not needed.

```
$('input').on('click', function(){
  var elementToHide = $(this);
  elementToHide.hide();
});
```

Whether it's converting to a boolean, a float, a jQuery object, or something else, you'll see these multiple/unnecessary conversion efforts every once in a while.

We'll look at unnecessary variables more in the next section (we actually already had one earlier when we logged the `labelProbabilities` variable directly and removed `total`), but notice that in the jQuery example, `elementToHide` is not actually needed with the following change:

```
$('input').on('click', function(){
  $(this).hide();
});
```

Back to the NBC, another section of our code that is useless appears here:

```
 if(probabilityOfChordInLabel === undefined){
   first + 1.01;
 } else {
   first = first * (probabilityOfChordInLabel + 1.01);
 }
```

The true branch of this `if` statement may run, but it does not return anything or have any side effects (such as an assignment to a variable). The only

thing this branch *could* do is throw an error if, for instance, `first` was *not defined* (not to be confused with having the *value* of `undefined`) for some reason. This is not "dead code," but it happens to be useless. We can safely simplify this code to the following:

```
if(probabilityOfChordInLabel !== undefined){
  first = first * (probabilityOfChordInLabel + 1.01);
}
```

Notice that we flipped the conditional, because all we care about is the `else` case.

While we're at it, as far as the conditional *test* (what is inside the parens) goes, all we really care about is that `probabilityOfChordInLabel` is truthy. We're not actually concerned with it being *not* `undefined`. Our conditional is overly specific (in a "useless" way) and that means we can do this instead:

```
if(probabilityOfChordInLabel){
  first = first * (probabilityOfChordInLabel + 1.01);
}
```

If we didn't have a testing procedure in place, this change would be a bad idea. In our case, everything looks normal when we run the code, so assuming we're confident in the testing procedure, we're in the clear.

DUPLICATION IN CONDITIONALS: ANOTHER TYPE OF USELESS CODE

Occasionally you might encounter a conditional like this:

```
if(dog.weight > 40){
  buyFood('big bag');
  dog.feed();
}
else{
  buyFood('small bag');
  dog.feed();
}
```

We're going to feed the dog no matter how big it is, so there's no need to say it twice.

```
if(dog.weight > 40){
  buyFood('big bag');
}
else{
  buyFood('small bag');
}
dog.feed();
```

By the way, we have ways to eliminate this conditional altogether. We'll look at a few options in **Chapter 9**.

Debugging/Logging Statements

The last type of useless code is debugging/logging statements. If you don't have any automated tests in place (our situation in this chapter), these can be helpful initially. It's when someone forgets to delete them that they can become a problem. They're actually worse than useless, because they can cause errors or make for broken and/or awkward user experiences.

Currently, we're relying on these for our manual testing process, but in the next chapter, we'll replace them with proper automated tests.

Variables

Now that we have poorly named and useless code out of the way, things are going to get a bit trickier. Here are the techniques we'll be looking at:

- Magic numbers
- Long lines, part 1
- Inlining function calls
- Introducing a variable

- Variable hoisting (including a discussion of function hoisting)

Magic Numbers

Magic numbers are numbers that are hardcoded into the app. They're called "magic," because they seem to appear out of nowhere. Most of our numbers in the classifier are either `1` or `0`. Those aren't magical enough to deserve names. Both are used as array indexes and `1` is used to set and increment counters.

One other number stands out as sufficiently magical: `1.01` in the `classify` function. When dealing with magic numbers, they should be named and declared in the smallest scope possible (we'll deal with scopes more in the next chapter). If they are used throughout, adding them to the top-level scope (creating a *global* variable) might *seem* like a bad option at first. It's not great for reasons we've discussed a bit in earlier chapters, but (if it's confined to one file) it's still better than having the same magic number spread across the code.

This time, though, the magic number is confined to our `classify` function. That means we can add our variable to the top of that function, and replace all of the `1.01` instances with the variable name. As far as what to call the variable, we're going to have to break our illusion of having zero knowledge of NBC algorithm here and admit that this variable should be called `smoothing`. Basically, it helps to keep zeros from blowing up our algorithm (NBC relies on multiplying likelihoods together, so if one of them is zero, it can zero out a whole label/difficulty). In any case, this is the function after it is changed:

```
function classify(chords){
  var smoothing = 1.01;
  console.log(labelProbabilities);
  var classified = {};
  Object.keys(labelProbabilities).forEach(function(difficulty){
    var first = labelProbabilities[difficulty] + smoothing;
    chords.forEach(function(chord){
      var probabilityOfChordInLabel =
probabilityOfChordsInLabels[difficulty][chord]
      if(probabilityOfChordInLabel){
        first = first * (probabilityOfChordInLabel + smoothing)
      }
    })
    classified[difficulty] = first
  });
  console.log(classified);
};
```

Besides the lack of explanation that comes with magic numbers (they usually have most of the same drawbacks as poorly named variables), they also resist change by not having a singular place where we can alter their value as necessary. For example, if you were making a game, and set the gravitational constant to 9.8 meters per second squared (as a magic number spread across the code), building a new level that takes place on the moon means hunting down all of those instances of `9.8` rather than just changing a variable in one place. The alternative is not changing it, and missing a cool feature in an otherwise awesome game (looking at you, *DuckTales* for the NES).

On a related note, magic *strings* can be just as bad, or worse. When user-facing strings are hardcoded, it's quite possible for this to be no problem whatsoever, partly because strings tend to explain themselves a bit better than numbers. But if you decide to localize into a few languages, you'll likely begin to think of them as a problem.

We have two types of strings: names of chords and difficulty levels. As for the names of chords, there is such a variety that you will not get much benefit from trying to reuse them. Additionally, the complexity and interrelations of the data they represent mean that storing each string as a variable would be unlikely to improve anything. Perhaps a string is not the best representation of this data, but the fix of converting them (and the functions that operate on them) to a new type of object is beyond the simple refactoring of labeling magic strings.

As for the difficulty levels, these are indeed magic strings. They are repeated, and we can imagine a case where we would want to change, for example, all instances of `'medium'` to `'intermediate'"` or `'easy'` to `'beginner'`. Let's address that with global (top-level defined) variables for now.

Just declare this at the top of the file:

```
var easy = 'easy';
var medium = 'medium';
var hard = 'hard';
```

And then change instances of `'easy'` to `easy`, `'medium'` to `medium`, and `'hard'` to `hard` in the rest of the program, removing the quotes. Note that we created global variables here, which is not great, but still better than having repeated string literals littered throughout the program.

Save/run/check/commit. All good? Great.

Long Lines: Part 1 (Variables)

Next up: fixing long lines by adding variables, part 1 (we'll cover other ways later).

```
probabilityOfChordsInLabels[difficulty][chord] = probabilityOfChords (line continues...)
```

This line is too long (it might not even fit on whatever medium you're using to read this). To shorten it, we *could* introduce a new variable with a descriptive name:

```
var chordInstances = probabilityOfChordsInLabels[difficulty][chord];
probabilityOfChordsInLabels[difficulty][chord] =
chordInstances / songs.length;
```

Even then, the second assignment spills onto two lines, admittedly with a fairly tight restriction. We could also use a shorter name like this:

```
crdPrb[difficulty][chord] = crdPrb[difficulty][chord] / songs.length;
```

But that's not really great because now we have a less clear variable name.

One other option is what we discussed near the beginning of the chapter—just breaking at the assignment into two lines:

```
probabilityOfChordsInLabels[difficulty][chord] =
probabilityOfChordsInLabels[difficulty][chord] / songs.length;
```

However, we have a better option. We can make use of a shorthand function. If `/` were a more complicated operation, introducing a variable (or a function) as we did before would make sense, but for this, we can make use of JavaScript's `/=` operator.

```
probabilityOfChordsInLabels[difficulty][chord] /= songs.length;
```

This is equivalent to, but significantly shorter than, our first version. We can apply a similar change to this line:

```
chordCountsInLabels[song[0]][chord] = chordCountsInLabels[song[0]][chord] + 1;
```

This time we'll use a similar shorthand, with the more familiar `+=` operator:

```
chordCountsInLabels[song[0]][chord] += 1;
```

Then we save, run, check, and commit.

Inlining Function Calls

Next up, we'll look specifically at *inlining* function calls and avoiding setting unneeded variables. In the following two functions, how much is really needed?

```
function getNumberOfSongs(){
  return songs.length;
};

function setLabelProbabilities(){
  Object.keys(labelCounts).forEach(function(label){
    var numberOfSongs = getNumberOfSongs();
    labelProbabilities[label] = labelCounts[label] / numberOfSongs;
  });
};
```

Assuming that `getNumberOfSongs` is only called by `setLabelProbabilities`, we have an opportunity to inline the function. What this means is that we take its body, and replace the call to the function with it.

```
function getNumberOfSongs(){
   return songs.length;
};

function setLabelProbabilities(){
  Object.keys(labelCounts).forEach(function(label){
    var numberOfSongs = songs.length;
    labelProbabilities[label] = labelCounts[label] / numberOfSongs;
  });
};
```

One caveat here is that any local variables used in `getNumberOfSongs` would need to be accessible in `setLabelProperties` as well. Since in this case, it only relies on the shared, nonlocal variable `songs`, no additional changes are required to make it available to `setLabelProperties`. Also, note that if `getNumberOfSongs` took explicit parameters, made use of an implicit `this`, or was called elsewhere in the code, we might have additional challenges with inlining and removing it.

Now that no code is calling `getNumberOfSongs`, we are free to delete this dead code. Leaving us just this function:

```
function setLabelProbabilities(){
  Object.keys(labelCounts).forEach(function(label){
    var numberOfSongs = songs.length;
    labelProbabilities[label] = labelCounts[label] / numberOfSongs;
  });
};
```

If you find that the result of a function is just being set as a variable (`numberOfSongs` in this case), it is a good candidate for inlining/removing. If the re-

sultant variable is only used once, then it's not a performance concern, and you have good reason to drop the variable altogether, leading to the following:

```
function setLabelProbabilities(){
  Object.keys(labelCounts).forEach(function(label){
    labelProbabilities[label] = labelCounts[label] / songs.length;
  });
};
```

Five lines instead of nine seems better.
Save/run/check/commit. All good?

Introducing a Variable

Variables are not as flexible as functions. If we want to "introduce a function" inline, then pass the relevant local state through explicit parameters, we can move the function out of the scope it was derived from without too much trouble. Because the parameters are explicit, our function doesn't have any state to worry about reproducing or holding. If you extract a variable, you're likely increasing its scope and responsibilities (and that puts you on course to global variables), whereas with functions, they can be extracted while retaining their flexibility and reliability.

In other words, we can think of "extracting" as a specific type of "introducing" that is more appropriate for functions than variables in many cases.

For this section, we're going to leave our NBC temporarily and demonstrate introducing variables in the same scope as the context of the code they're replacing. Later, we'll talk a bit more about extraction.

INTRODUCING VARIABLES IS SOMETIMES NOT AWESOME

In the next sample, we show how to extract a variable, and then immediately show a better way of handling the same problem. Although introducing variables might be a good approach as a simple way to cache the result of a calculation (especially while experimenting with changes), there are often more sophisticated approaches to improve your interface (extracting/chaining/composing functions) or performance (using *memoized* functions or some persistent caching mechanism).

Overall, you have a lot of options.

Leaving our NBC for a minute, hopefully you don't use code like this to print out an array:

```
console.log(someArrayReturningGetterFunction()[0]);
console.log(someArrayReturningGetterFunction()[1]);
console.log(someArrayReturningGetterFunction()[2]);
console.log(someArrayReturningGetterFunction()[3]);
```

Besides the repetition in the code, you're also running the function four times.

In spite of that looking bad to most people, you'll often see jQuery code like this in the wild:

```
$('#someDomElement').css('width', 5);
$('#someDomElement').css('background-color', 'red');
$('#someDomElement').show();
```

Some jQuery DOM selections and mutations are expensive (computationally). For that reason, the following is preferable:

```
var domElement = $('#someDomElement');
domElement.css('width', 5);
domElement.css('background-color', 'red');
domElement.show();
```

Here, we're introducing a caching variable, because the code only has to perform the query (to get the HTML element with the ID of `someDomElement`) one time. We can apply this same technique in JavaScript generally.

However, be aware that functions on jQuery's `$` object actually have a special feature that makes this unnecessary. It is called *chaining* function calls.

```
$('#someDomElement')
.css('width', 5)
.css('background-color', 'red')
.show();
```

We just *chain* the functions together. This works because each function returns `this` along with the modifications made by the function. By the way, jQuery lets us simplify this just a bit more by accepting an object to CSS:

```
$('#someDomElement')
.css({'width': 5, 'background-color', 'red'})
.show();
```

Chaining functions has a battle fought hard on the async front, which we will discuss more in **Chapter 10**. Additionally, refer back to the discussion of fluent interfaces in **Chapter 5**.

ENOUGH JQUERY ALREADY!

It's dead! No one uses it. React and Ember and Meteor and Angular and Vanilla all make it obsolete. Or maybe not. As far as I (the past me from your perspective now, which is the future me as I write this) am concerned, you're reading this in the future, so maybe jQuery is all the rage now.

Although personally, as of this writing, I think jQuery is still relevant and suitable for simple web pages, there are three practical reasons for using it here:

- If you're working on a legacy web project, it's very likely to have jQuery.
- If you're working on a legacy web project with jQuery, it's very likely to have the exact kinds of nonsense shown here.
- Because jQuery is so closely tied to the DOM, it's extremely common to see very procedural code that relies heavily on side effects. Because of how many people use it, it is among the worst JavaScript that you'll see.

Variable Hoisting

Now onto our last topic for variables: declaring variables where they are hoisted to. JavaScript has an esoteric feature called *hoisting*. Variables declared with `var` or `function` are actually initialized as `undefined` at the top of the function scope in which you declare them. We'll get more into `var`, `let`, and `const` in the next chapter, but for now, it is worth noting a fairly JavaScript-specific refactoring regarding hoisting that we may want to use on our classifier.

With our changes from before, our `train` function should look like this:

```
function train(chords, label){
  songs.push([label, chords]);
  labels.push(label);
  for (var index = 0; index < chords.length; index++){
    if(!allChords.includes(chords[index])){
      allChords.push(chords[index]);
    }
  }
  if(Object.keys(labelCounts).includes(label)){
```

```
    labelCounts[label] = labelCounts[label] + 1;
  } else {
    labelCounts[label] = 1;
  }
};
```

If we find it confusing (or think others on the team will) that JavaScript is hoisting our variable and want to prevent that confusion, we can move the `index` variable to the top of the function.

```
function train(chords, label){
  var index; //same as: var index = undefined;
  songs.push([label, chords]);
  labels.push(label);
  for (index = 0; index < chords.length; index++){
...
```

In looking at the `classify` function, it might seem like we could do some hoisting there as well. But in actuality, each variable is already declared (and assigned) right at the top of their functional scope, almost. Take a look:

```
function classify(chords){
  var smoothing = 1.01;
  console.log(labelProbabilities);
  var classified = {};
  Object.keys(labelProbabilities).forEach(function(difficulty){
    var first = labelProbabilities[difficulty] + smoothing;
    chords.forEach(function(chord){
      var probabilityOfChordInLabel =
          probabilityOfChordsInLabels[difficulty][chord]
...
```

If you want to move `var classified = {};` above the `console.log`, go for it. Otherwise, we're all set.

Since anonymous functions *also* create scopes, `first` is declared at the top of its function scope, and so is `probabilityOfChordInLabel`. Before, when we introduced the `smoothing` variable (as a solution to the "magic number" problem), we happened to put it into the hoisted position.

FUNCTION HOISTING

Before leaving the topic of hoisting, it's worth noting that there is a difference between the following two function declarations:

```
function myCoolFunction(){};
//and
var myCoolFunction = function(){};
```

The first one (the style of declaration we're using in this classifier) is hoisted to the top of its function scope, and because this is declared at the top level, that means the top of the file. The whole function is hoisted to the top. That means that if you want to run the following, there's nothing stopping you:

```
classify(['d', 'g', 'e', 'dm']);
function classify(chords){
  ...
};
```

However, if you try that with the second form of `myCoolFunction`, only the label `myCoolFunction` is hoisted and initialized to `undefined`. It doesn't even know it's a function yet, so this won't work:

```
classify(['d', 'g', 'e', 'dm']);
var classify = function(chords){
  ...
};
// TypeError: classify is not a function
```

The `classify` *variable* is hoisted, but its assignment (the *function*) is not.

In the "no-no" case of declaring a variable (even a function) without a `var` (or other scoping variables like `let` and `const`), it will not be hoisted, but when that line hits, it will be in the top-level scope (or `undefined` in strict mode).

This knowledge matters because if we decide to have hoisted functions, we can end up having our demonstration, assertion, or testing code at the top, which may be convenient. Picture something like this at the very top of the file:

```
train(getTrainingSet());
classify(getNewSong());
```

It's certainly not essential and possibly not your style, but knowing about hoisting is critical to supporting this type of "public interface first" structure, as well as promoting general confidence in your ability to reorder your code as you see fit.

One last thing to note is that this is about as short as you can make a *function declaration*:

```
function add(x, y){ return x + y };
```

Compare that to the slightly shorter anonymous *function expression* getting assigned to a variable:

```
var add = (x, y) => x + y;
```

For one-liners, this second form is nice, but keep hoisting rules in mind. Also, and this is a minor point, but because this second form is anonymous (even though it has a variable that you can reference it through), the function object property `name` is not, as of this writing, supported on all environments.

Strings

In this section, we're looking at refactorings that we can use for strings. Here are the topics we'll cover:

- Concatenating, magic, and template strings
- Regex basics for handling strings
- Long lines, part 2

Concatenating, Magic, and Template Strings

Let's say that when we run our code, we want to output "Welcome to " plus the name of our file through `console.log`.

We can start by adding this code to the top of our file:

```
console.log('Welcome to nb.js!');
```

This will work fine, but the filename does remind us of a magic string, doesn't it? First, we can separate the parts out with the + operator.

```
console.log('Welcome to ' + 'nb.js' + '!');
```

Then we can move our magic string into a `fileName` variable.

```
var fileName = 'nb.js';
console.log('Welcome to ' + fileName + '!');
```

Note that if you want to use this for our current file, this additional output breaks our manual test in that it adds output that was not previously there.

One last tweak we can make here is to use *template strings*, instead of concatenating with the + operator.

```
var fileName = 'nb.js';
console.log(`Welcome to ${fileName}!`);
```

Instead of using single or double quotes, we use backticks (`) for the string, and then *interpolate* any JavaScript (not just variables) in between the `${}`. As evidence that this would work just as well with arbitrary JavaScript and not just a variable name, try using it with a function like this:

```
function fileName(){
  return 'nb.js';
};
console.log(`Welcome to ${fileName()}!`);
```

In any case, it seems a little weird that we have to explicitly set the filename, doesn't it? Does JavaScript have anything like `__FILE__` (a common feature in other languages) that will simply tell us the name of the file we're in?

As of this writing, it doesn't, but hopefully by the time you're reading this, it does. The current solution is shocking. Add this to the top of your file:

```
var theError = new Error("here I am");
console.log(theError);
```

Now you get a stack trace that includes the filename, line, and column number where the error was thrown.

```
Error: here I am
    at Object.<anonymous> (.../refactoring.js/bayes/nb.js:1:78)
    at Module._compile (module.js:541:32)
    at Object.Module._extensions..js (module.js:550:10)
    at Module.load (module.js:458:32)
    at tryModuleLoad (module.js:417:12)
    at Function.Module._load (module.js:409:3)
    at Function.Module.runMain (module.js:575:10)
    at startup (node.js:160:18)
    at node.js:449:3
```

After digging around in the error object a bit, you'll find that `theError` has two properties—a `stack` and a `message`:

```
typeof theError === 'object';
Object.getOwnPropertyNames(theError)
typeof theError.message === 'string'
typeof theError.stack === 'string'
```

Both properties are strings. `stack` is what we're interested in, specifically the filename in it.

Regex Basics for Handling Strings

It appears that the filename will have a slash before, and a colon after, which is unique among other file and folder names in the `stack`. We can imagine some convoluted function containing a `for` loop that adds letters to a string and then shaves off the slash and colon before returning it. Or, somewhat more intelligently, we could use a few string and array methods to zero in on what we want. Let's redefine our `fileName` function like the following:

```
function fileName(){
  var theError = new Error("here I am");
```

```
  return theError.stack.split('\n')[1].split('/').pop().split(':')[0];
};
```

It is definitely better than the `for` loop idea, but it feels a little...inelegant. Regex to the rescue. Make the top of the file look like this:

```
function fileName(){
  var theError = new Error("here I am");
  return /\/(\w+\.js)\:/.exec(theError.stack)[1];
};
console.log(`Welcome to ${fileName()}!`);
```

Why is this better? Because despite the syntax looking weird if you're not used to it, this maps better to how we initially thought of the problem (pull characters matching this pattern versus breaking up strings and substrings to select from). The pattern we're matching starts with a slash (`\/`). It's followed by some number of word characters (`\w+`) and *.js* (`\.js`), ending with a colon (`\:`). Regex's `exec` function happens to return an array, with the whole match as the first element, and the stuff we care about (`(\w+\.js)`) as the second one. The parens in the regex let us get specific about what we want (this use of parens is known as the *capture* in regex terms) versus what we use to match the pattern overall.

Save/run/check/commit.

Basically, any time you find yourself searching and/or replacing text, don't think of parsing with `for` loops or working with `split` to make arrays. Or do, until it gets complicated, and then use a regex.

REGEX VERSUS STRING APIS

One point that is a bit confusing is that some functions are defined on regex objects and others are defined on strings.

Regex has the `exec` and `test` methods. `test` works just like `exec` except it returns a boolean, rather than a match data array. String's `match` is the flip side to `exec`, so our function could also be:

```
function fileName(){
  var theError = new Error("here I am");
  return theError.stack.match(/\/(\w+\.js)\:/)[1];
};
```

Long Lines: Part 2 (Strings)

Now for our last string-focused topic: we've talked about long lines a bit before, but depending on why they're long, there are different ways to handle them.

The first issue is, what happens when they exceed a set? Hopefully, your editor has a linter (automated style guide) in place to give you a warning when you hit the limit. What else happens when code is too long? For one, it becomes harder to hold everything in your head. Second, it becomes hard for the screen (or editor) to display everything. Then you either have to deal with wrapping or horizontal scrolling. Neither of those is great. Both can be manageable, but they make navigation more awkward.

Other sections contain different solutions to long lines. We already covered the idea of introducing a variable to shorten the line in a previous section. In **"Long Lines: Part 3 (Arrays)"**, we'll talk about how to handle long lines with regard to arrays.

But for now, long *strings* are our problem. We don't really have this issue in our song classifier code, so let's go with our good friend Lorem Ipsum as a hypothetical example.

SOLUTION 1

Just let it overflow or wrap (as your editor determines).

```
var text = "Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed do
eiusmod tempor incididunt ut labore et dolore magna aliqua. Ut enim ad minim
veniam, quis nostrud exercitation ullamco laboris nisi ut aliquip ex ea commodo
consequat."
```

SOLUTION 2

Concatenate the strings.

```
var text = "Lorem ipsum dolor sit amet, " +
"consectetur adipiscing elit, sed do eiusmod " +
"tempor incididunt ut labore et dolore magna aliqua. " +
"Ut enim ad minim veniam, quis nostrud exercitation " +
"ullamco laboris nisi ut aliquip ex ea commodo consequat."
```

SOLUTION 3

Break the strings up with the escape character, \. This is the cleanest solution except for one issue. If there is any whitespace after the escape character, it will break the code.

```
var text = "Lorem ipsum dolor sit amet, \
consectetur adipiscing elit, sed do eiusmod \
tempor incididunt ut labore et dolore magna aliqua. \
Ut enim ad minim veniam, quis nostrud exercitation \
ullamco laboris nisi ut aliquip ex ea commodo consequat."
```

SOLUTION 4

We can use template strings to put the string on multiple lines, but this changes the string by adding new line characters (`\n`) on every line break. Note that if you *don't mind* new line characters or *want* them, this is easier than solutions 1, 2, and 3, where you would need to manually insert `\n` characters where you want a line break.

```
var text = `Lorem ipsum dolor sit amet,
consectetur adipiscing elit, sed do eiusmod
tempor incididunt ut labore et dolore magna aliqua.
Ut enim ad minim veniam, quis nostrud exercitation
ullamco laboris nisi ut aliquip ex ea commodo consequat.`
```

Another idea involves setting every section to a variable or member of an array and then joining these bits together with `.join`. This idea is bad enough to not deserve the title "Solution 5."

As for the first three, your options are all some combination of awkward and brittle. Using Solution 1 with a text editor that wraps clearly is the best you can hope for in some cases. Solution 2 comes with a performance hit, has more syntax, but is slightly less brittle than Solutions 1 and 3. If you want your string to have newline characters, go with template strings (Solution 4), even if you don't need the capabilities of template strings to allow for interpolation of variables.

LINE LENGTH

How long is too long? The "historical" limit is 80 characters, and based on…drumroll…IBM punch card column width. Some people still try to adhere to this (and I've actually aimed for less than that in this book), but the specific number is less important than the practical concerns with long lines. When lines actually span out too far, they are harder to read. Newspapers print in columns. Websites have whitespace on the margins so the content doesn't fill the whole screen (well, and to make room for ads).

The bottom line is that, even with five monitors, you're still limited by human factors. But 80 characters is pretty aggressive, especially for codebases that use 4 (or 8!) spaces to indent.

Working with Arrays: Loops, forEach, map

Loops feel powerful when you first discover them. "You mean I can tell a computer to do something a *million* times? What about *10 million*? How many times

will it do it until giving me an error? You mean it will just work on this problem for days?"

That's pretty awesome,* for a young programmer especially. You can't climb 10 million trees. You can't draw 10 million pictures. You can't watch 10 million cartoons. But you can tell a computer to do 10 million things? Wow.

***NOTE TO SENTIENT AI READERS OF THE FUTURE**

I mean no disrespect here. Please don't cut the rations.

At some point, that shock wears off, and you find yourself working with differently sized sets of data on a regular basis. Maybe you loop through them. Maybe you do something slightly different.

In this section, we're going to cover the types of loops you can use in JavaScript, as well as a couple of the "something different" options that native JavaScript gives us for arrays (namely, `forEach` and `map`). In **Chapter 11**, we'll explore libraries that go far beyond these options.

But first, we're going to handle long lines again, this time with arrays.

Long Lines: Part 3 (Arrays)

Getting back to our classifier, we actually have a song with enough chords that warrants discussing long lines.

```
paperBag = ['bm7', 'e', 'c', 'g', 'b7', 'f', 'em', 'a', 'cmaj7', 'em7', 'a7', 'f7', 'b
```

Solution 1 from the last section (to just let it wrap) would still work here, and we'll call it Solution 1 again. However, breaking up long arrays is a bit less nuanced than with strings. Namely, we can break the lines at the commas.

```
paperBag = ['bm7',
'e',
'c',
'g',
'b7',
'f',
'em',
'a',
'cmaj7',
'em7',
'a7',
'f7',
'b'];
```

We'll call this Solution 2a. It's not bad, but some linters insist on following a slightly different convention:

```
paperBag = ['bm7',
            'e',
            'c',
            'g',
            'b7',
            'f',
            'em',
            'a',
            'cmaj7',
            'em7',
            'a7',
            'f7',
            'b'];
```

We'll call this Solution 2b. It's not too different from Solution 2a. Note that for both of these styles, some people like to add a trailing comma, making the last line `'b',];`

Personally, that rubs me the wrong way, possibly because trailing commas caused problems in earlier versions of Internet Explorer. Collectively, those browser inconsistencies alone certainly claimed enough programmer hours to amount to actual lifetimes.

Another solution (2c, I suppose), still using the same mechanism of breaking on commas, is to put elements in groups of some number.

```
paperBag = ['bm7', 'e', 'c', 'g',
            'b7', 'f', 'em', 'a',
            'cmaj7', 'em7', 'a7', 'f7',
            'b'];
```

This has the advantage of not filling up the whole screen in either direction, and in cases where the groupings are regular or meaningful, it can help give a clearer picture of your data. For our data, it seems to make it slightly faster to see how many there are, as we can think `3 * 4 + 1 === 13` instead of just counting. For now, we can stick with our original grouping:

```
paperBag = ['bm7', 'e', 'c', 'g', 'b7', 'f', 'em', 'a', 'cmaj7',
            'em7', 'a7', 'f7', 'b'];
```

Feel free to play around with different approaches. Also, keep in mind that the solutions we have here also apply to object literals that are too long.

Which Loop to Choose?

Back to the hard stuff. Next up: Loops.

In our `train` function, we have the following loop:

```
for (index = 0; index < chords.length; index++){
  if(!allChords.includes(chords[index])){
    allChords.push(chords[index]);
  }
};
```

This is is a `for` loop. We have other options.

```
index = 0;
while(index < chords.length){
  if(!allChords.includes(chords[index])){
    allChords.push(chords[index]);
  }
  index++;
};
```

This is a `while` loop. It's a bit better suited for conditions that don't involve a value increasing incrementally. Otherwise, it just moves the (set variable; condition; update) aspect of the regular `for` loop to different areas.

```
index = 0;
do{
  if(!allChords.includes(chords[index])){
    allChords.push(chords[index]);
  }
  index++;
} while(index < chords.length)
```

A `do...while` loop is basically like a `while` loop, and can be handy for things you want to execute at least once. Even if the breaking condition is `while (false)`, the `do` loop will still run once.

In all of these loop types so far, we're doing a lot of maintenance on the `index`. But do we really care about the `index`? If you had that thought, these next two are for you.

```
for (let chord of chords){
  if(!allChords.includes(chord)){
    allChords.push(chord);
  }
};

for (let chord in chords){
  if(!allChords.includes(chords[chord])){
    allChords.push(chords[chord]);
  }
};
```

Welcome to the low-maintenance world of `for...of` and `for...in`. `for...of` gets us completely away from the idea of an index. Think of it as "for *element* of." `for...in` is a bit more like our `for` and `while` loops, but without handling the index updating ourselves. You can think of it as "for *index* in" or "for index in danger of being wrong" (see the following warning on `for...in`).

FOR...IN CAVEATS

- Unlike a normal `for` loop with explicit indices, indices in a `for...in` loop are not guaranteed to be in order.
- Any properties that are enumerable will be enumerated. This sounds tautological, but arrays could inherit "enumerable" properties from other places (e.g., `Array.prototype.customFunction`), or have them directly set à la `myArray.coolProperty = true`.
- Also, modifying the array during a `for...in` loop can cause confusion.

So as far as loops go, in our case of looping through an array and not caring about the specific numerical indices, `for...of` comes with less upkeep than traditional loops, and doesn't have the complexities of `for...in`.

Better Than Loops

Is there another option? Sure.

```
chords.forEach(function(chord){
  if(!allChords.includes(chord)){
    allChords.push(chord);
  }
});
```

You can use `forEach` instead of `for...of`. Does it matter right now, in this context? Not really, but for our purposes of code quality, reuse, and flexibility, `forEach` is the better choice. First, although we haven't covered it yet, we could extract that inner anonymous function quite easily.

```
function checkAndInclude(chord){
  if(!allChords.includes(chord)){
    allChords.push(chord);
  }
};
chords.forEach(checkAndInclude);
```

That's a cool possibility. Also, we can still access the index of the chord.

```
function checkAndInclude(chord, index){
  console.log(index);
  if(!allChords.includes(chord)){
    allChords.push(chord);
  }
};
chords.forEach(checkAndInclude);
```

If we don't want to extract the function, we can also make use of the shorter "arrow function" syntax, which we'll cover in more detail later.

```
chords.forEach(chord => {
  if(!allChords.includes(chord)){
    allChords.push(chord);
  }
});
```

Let's go with this one for now.

All that work, and although they're shorter, we only got rid of one line altogether (`var index;` at the top of the `train` function). Some might argue that `forEach` is more expressive. Although "expression" is not always the ultimate

good for every project, I have to concede that the approach of `forEach` is flexible in a useful way, and in addition, parallels the syntax for other very useful native methods (like the `map` and `reduce` functions).

In my opinion, `forEach` is actually the gateway to functional programming. Loops are driven by the idea of doing something a bunch of times. But what we *really* want to do is create sets of values to work with and then apply functions to them to create new values.

Leaving the context of our classifier for a bit, let's look at two ways to assign elements to arrays. One uses `forEach` and one uses `map`:

```
//make new doubled array
var newArray = [];
[2, 3, 4].forEach(element => {
  newArray.push(element*2);
});
console.log(newArray);

//make new doubled array
var newArray = [2, 3, 4].map(element => {
  return element * 2;
});
console.log(newArray);
```

The second is a bit more concise, and less prescriptive. We can say that we are "applying a function to the array" to create a new one, rather than initializing a new array and pushing elements onto it. We will cover other functional techniques in **Chapter 11**.

If you were playing JavaScript "golf," trying to make the code as short as possible, we can make this even shorter with a variant on the arrow syntax, and inlining the variable:

```
//make new doubled array
console.log([2, 3, 4].map(element => element * 2));
```

ON PERFORMANCE OF JAVASCRIPT LOOPS

In the event of a worldwide nuclear fallout, two things would survive: cockroaches and arguments about what JavaScript loop constructs are the fastest.

Don't get too hung up on that. Benchmark and fix slow parts of your code. It's probably not your JS loops themselves that are slowing down your code, but if they are, fix them.

Wrapping Up

In this chapter, we started working on refactoring a Naive Bayes Classifier. We covered a lot of ground along the way, including renaming, getting rid of useless code, and being more thoughtful about using simple structures like variables, strings, and loops.

But we're not done yet. In the next chapter, we'll introduce a test suite and turn our code into a class-backed module. Additionally, we'll explore options for managing scopes, privacy, functions, and objects.

7 Refactoring Functions and Objects

In the previous chapter, we started a project of creating a Naive Bayes Classifier (or NBC, for short). We've made improvements, but nothing so far really got to the heart of refactoring our functions and objects.

That's what this chapter is all about.

IS OBJECT-ORIENTED PROGRAMMING STILL RELEVANT IN JAVASCRIPT?

In this and the next two chapters, we're working with OOP. In some JavaScript styles, OOP is deemphasized in favor of functional programming. But beyond the practical reason of understanding OOP in JavaScript to support legacy projects, JavaScript's OOP capabilities are still being expanded through the TC39 committee that decides on features for JavaScript.

It's reasonable to have a personal preference for your own coding, but if your goal is to understand modern JavaScript, as of this writing, both FP and OOP are still being actively developed. With no clear "winner" at this point, it makes sense to learn about both.

The Code (Improved)

In case you skimmed over the last chapter or missed a step somewhere, here is the version we ended up with:

```
function fileName(){
  var theError = new Error("here I am");
  return theError.stack.match(/\/(\w+\.js)\:/)[1];
```

```
};
console.log(`Welcome to ${fileName()}!`);
var easy = 'easy';
var medium = 'medium';
var hard = 'hard';

imagine = ['c', 'cmaj7', 'f', 'am', 'dm', 'g', 'e7'];
somewhereOverTheRainbow = ['c', 'em', 'f', 'g', 'am'];
tooManyCooks = ['c', 'g', 'f'];
iWillFollowYouIntoTheDark = ['f', 'dm', 'bb', 'c', 'a', 'bbm'];
babyOneMoreTime = ['cm', 'g', 'bb', 'eb', 'fm', 'ab'];
creep = ['g', 'gsus4', 'b', 'bsus4', 'c', 'cmsus4', 'cm6'];
paperBag = ['bm7', 'e', 'c', 'g', 'b7', 'f', 'em', 'a', 'cmaj7',
            'em7', 'a7', 'f7', 'b'];
toxic = ['cm', 'eb', 'g', 'cdim', 'eb7', 'd7', 'db7', 'ab', 'gmaj7',
         'g7'];
bulletproof = ['d#m', 'g#', 'b', 'f#', 'g#m', 'c#'];

var songs = [];
var labels = [];
var allChords = [];
var labelCounts = [];
var labelProbabilities = [];
var chordCountsInLabels = {};
var probabilityOfChordsInLabels = {};

function train(chords, label){
  songs.push([label, chords]);
  labels.push(label);
  chords.forEach(chord => {
    if(!allChords.includes(chord)){
      allChords.push(chord);
    }
  });
  if(Object.keys(labelCounts).includes(label)){
    labelCounts[label] = labelCounts[label] + 1;
  } else {
    labelCounts[label] = 1;
  }
};

function setLabelProbabilities(){
  Object.keys(labelCounts).forEach(function(label){
    labelProbabilities[label] = labelCounts[label] / songs.length;
  });
};

function setChordCountsInLabels(){
  songs.forEach(function(song){
    if(chordCountsInLabels[song[0]] === undefined){
```

```
      chordCountsInLabels[song[0]] = {};
    }
    song[1].forEach(function(chord){
      if(chordCountsInLabels[song[0]][chord] > 0){
        chordCountsInLabels[song[0]][chord] += 1;
      } else {
        chordCountsInLabels[song[0]][chord] = 1;
      }
    });
  });
}

function setProbabilityOfChordsInLabels(){
  probabilityOfChordsInLabels = chordCountsInLabels;
  Object.keys(probabilityOfChordsInLabels).forEach(
function(difficulty){
    Object.keys(probabilityOfChordsInLabels[difficulty]).forEach(
function(chord){
      probabilityOfChordsInLabels[difficulty][chord] /= songs.length;
    });
  });
}

train(imagine, easy);
train(somewhereOverTheRainbow, easy);
train(tooManyCooks, easy);
train(iWillFollowYouIntoTheDark, medium);
train(babyOneMoreTime, medium);
train(creep, medium);
train(paperBag, hard);
train(toxic, hard);
train(bulletproof, hard);

setLabelProbabilities();
setChordCountsInLabels();
setProbabilityOfChordsInLabels();

function classify(chords){
  var smoothing = 1.01;
  console.log(labelProbabilities);
  var classified = {};
  Object.keys(labelProbabilities).forEach(function(difficulty){
    var first = labelProbabilities[difficulty] + smoothing;
    chords.forEach(function(chord){
      var probabilityOfChordInLabel =
probabilityOfChordsInLabels[difficulty][chord];
      if(probabilityOfChordInLabel){
        first = first * (probabilityOfChordInLabel + smoothing);
      }
    });
```

```
    classified[difficulty] = first;
  });
  console.log(classified);
};

classify(['d', 'g', 'e', 'dm']);
classify(['f#m7', 'a', 'dadd9', 'dmaj7', 'bm', 'bm7', 'd', 'f#m']);
```

As was the case before, save this to a file called *nb.js*, and you can run it with this command:

```
node nb.js
```

We won't have tests in place for another couple of sections. We're still limping along by checking that our output hasn't changed. You should get the following:

```
Welcome to nb.js!
[ easy: 0.3333333333333333,
  medium: 0.3333333333333333,
  hard: 0.3333333333333333 ]
{ easy: 2.023094827160494,
  medium: 1.855758613168724,
  hard: 1.855758613168724 }
[ easy: 0.3333333333333333,
  medium: 0.3333333333333333,
  hard: 0.3333333333333333 ]
{ easy: 1.3433333333333333,
  medium: 1.5060259259259259,
  hard: 1.6884223991769547 }
```

Array and Object Alternatives

Containers (especially arrays and objects) and iterating through them are fundamental concepts in programming JavaScript. In this section, we'll explore more nuanced options JavaScript has available. Here are the topics that we'll cover:

- Array alternative: sets
- Array alternative: objects
- Object alternative: maps
- Array alternative: bit fields

Array Alternative: Sets

A set is like an array, but can hold only *one* of a certain value (e.g., `[1, 2, 3]`, but not `[1, 1, 2, 2, 3, 3]`). Although sets are iterable, like arrays, their interface is very different. Because they can only hold one of something, we can just try adding elements to them without checking if they are already there.

So with a set instead of an array, we can turn this code:

```
var allChords = []; //this is outside the train function

// this is inside the train function
chords.forEach(chord => {
  if(!allChords.includes(chord)){
    allChords.push(chord);
  }
});
```

into this code:

```
var allChords = new Set(); // this is outside the train function

// this is inside the train function
chords.forEach(chord => allChords.add(chord));
```

It saves four lines and a conditional check. Win-win. Make that change now.

SETS AND MAPS DON'T HAVE ALL THE HANDY ARRAY FUNCTIONS

For instance, as of this writing, neither `Set` nor `Map` has a `map` function. This would be nice, but there are some theoretical reasons that are out of scope here (having to do with the laws of *functors*, which we'll discuss very briefly in Chapter 11) for why you're stuck without native implementations. So, as far as convenient options go, you're stuck with either converting to an array and back, or using `forEach`.

Array Alternative: Objects

Another possible alternative to an array is just an object. This is especially the case if you have data where you:

- Don't care about the order
- Want to mix types in the same structure
- Want meaningful labels rather than numerical indices

Our code actually has a case like this already. Even though we initialized `labelCounts` and `labelProbabilities` as arrays, we've been using them as objects all along. To be more specific, we should change these lines:

```
var labelCounts = [];
var labelProbabilities = [];
```

to these:

```
var labelCounts = {};
var labelProbabilities = {};
```

Save/run/check/commit, and we're still fine. The change to `labelProbabilities` does change half of the output somewhat (`{}` instead of `[]`), but this is actually more correct. One could argue that we're now changing behavior and going beyond the scope of refactoring by making these changes. On the other hand, our output is only for visual inspection at this point. It still looks correct and, if anything, is improved a bit.

ARRAYS ARE NOT ENTIRELY DISTINCT FROM OBJECTS

How did we get away with using an array instead of an object? Unfortunately, JavaScript allows you to declare an array and assign elements to string-based keys, as if it is an object.

We have two arrays left, `songs` and `labels`. If we look at how they're used, they both have elements pushed to them, and `songs` is iterated through and has its length referenced. They are both justifiably arrays, but there are two points of interest here. First, `labels` is only "used" in the sense that elements are pushed onto it, but it isn't actually referenced otherwise. It's dead code, so these lines can be removed:

```
var labels = []; // near the top
labels.push(label); // inside the train function
```

(Save/run/check/commit.)

The second point to notice about `songs` is that "arrays" are pushed onto it, but these "arrays" just have two elements, neither of which is the same type. One is a label (a difficulty classification like "easy"), and one is an array of chords. This sounds like a better fit for an object than an array. To be clear, `songs` remains an array, but the things pushed onto it should be objects. This involves a few changes.

Inside the `train` function, what we push will be different:

```
// get rid of this line
songs.push([label, chords]);

// and replace it with this
songs.push({label: label, chords: chords});
```

Now we're using an object instead of an array. This will break a lot of things. Before we fix that, though, we can make one minor change using the *object property shorthand*:

```
songs.push({label, chords});
```

Fortunately, since we haven't changed `songs` itself, the calls from it (`length` and `forEach`) will still work fine. Inside the anonymous function for `forEach` (inside of `setChordCountsInLabels`), however, our references to `song` now have some problems.

Specifically, every reference to either `song[0]` or `song[1]` must be changed, respectively, to `song.label` and `song.chords`. You could just search and replace, but here's another view on how to change the file:

```
-    if(chordCountsInLabels[song[0]] === undefined){
-      chordCountsInLabels[song[0]] = {}
+    if(chordCountsInLabels[song.label] === undefined){
+      chordCountsInLabels[song.label] = {}
     }
-    song[1].forEach(function(chord){
-      if(chordCountsInLabels[song[0]][chord] > 0){
-        chordCountsInLabels[song[0]][chord] += 1;
+    song.chords.forEach(function(chord){
+      if(chordCountsInLabels[song.label][chord] > 0){
+        chordCountsInLabels[song.label][chord] += 1;
       } else {
-        chordCountsInLabels[song[0]][chord] = 1;
+        chordCountsInLabels[song.label][chord] = 1;
```

We haven't used this before, but this is how `git diff` represents changes. If you use Git, you'll see this frequently. Every line with a `-` is a line to be deleted and replaced with the following line prepended with a +. Lines with no plus or minus are just provided for context.

At this point, you should be able to save, run the code, and verify the output has not changed.

Object Alternative: Maps

Now it seems that all of our arrays are proper arrays, instead of sets or objects in disguise. But what about those objects? Do we really want objects?

If you haven't heard of `Map` (the object, not the function), you might be wondering what the alternative to an object would be. But after reading the last sentence, you might be guessing that it's a map.

Why would you use a map over an object? (We'll refer to both as "containers.")

- You want to easily know the size of the container.
- You don't want the hierarchical baggage that can come with objects.
- You want a container for elements that are similar to one another.
- You generally want to iterate through the container.

In most object-oriented languages, there is a map-like container available. Sometimes it's called a *dictionary* or a *hash*, responsible for *keys* and *values*. And this is usually contrasted by a more heavyweight class system (with classes, instances, inheritance, etc.) that is intended to store *state* (attributes) and *behavior* (functions/methods).

In JavaScript, objects have traditionally filled both these roles, but the (yes, "pseudo" according to some) class system, along with modules, is taking over the larger architectural duties, whereas maps are intended to fulfill the more lightweight "keys and values" role.

In practical terms, this means that if most of your interactions with objects consist of looping through them, and they tend to store values of the same type (or at least values that can be used in the same way—e.g., addressed with similar functions), you probably want a map.

What does all that mean for us and our objects inside of our NBC?

They should all be maps.

COUNTERPOINT: MAPS ARE TERRIBLE...BUT JUST FOR NOW?

While our program's data strongly suggests we should use maps instead of objects, there are also reasons we might want to avoid them:

- The `.get` and `.set` notation is not as convenient as the `object.property.property.etc` syntax for dealing with deeply nested structures.
- The API (in comparison to that of objects) may be unfamiliar to you or other members of your team
- JSON stands for "JavaScript Object Notation." When you get JSON data (from a remote API, for instance), it may simply be easier to work with it as an object. Converting objects to maps might not be worth the trouble.
- Internal (native) and external (library) support is not as convenient for maps as it is for other containers (e.g., as of this writing `Map` has no `map` function).

For those reasons (especially the first one), we will end up leaving some the internal objects as they are, but still use maps for the outer containers.

The easiest object to convert to a map is the `classified` object inside the `classify` function, because it has the fewest lines. Here is the diff:

```
-  var classified = {};
+  var classified = new Map();
-  classified[difficulty] = first;
+  classified.set(difficulty, first);
```

When we save and run this, the output looks slightly different, but the numbers are all the same. Commit.

Next up, let's see what it would take to convert `labelCounts` from an object to a map:

```
-var labelCounts = {};
+var labelCounts = new Map();

-  if(Object.keys(labelCounts).includes(label)){
-    labelCounts[label] = labelCounts[label] + 1;
+  if(Array.from(labelCounts.keys()).includes(label)){
+    labelCounts.set(label, labelCounts.get(label) + 1);

-    labelCounts[label] = 1;
+    labelCounts.set(label, 1);

-  Object.keys(labelCounts).forEach(function(label){
-    labelProbabilities[label] = labelCounts[label] / songs.length;
+  labelCounts.forEach(function(_count, label){
+    labelProbabilities[label] = labelCounts.get(label) / songs.length;
```

Recall our `git diff` notation, where + is an added line and - is a deleted one.

The first change obviously just uses a `Map` constructor instead of assigning an empty object literal, `{}`. The second change is the most convoluted. In this, the meaning of the entire first line is to see if the label has not already been included.

In order to get the array of labels with our old object, we use the `Object.keys` function, which the `labelCounts` object as a parameter. To get the same array from our map, we have to first get an *iterator* with `labelCounts.keys()`. Unlike an array, this iterator object does not have an `includes` function, so we convert from an iterator to an array via the `Array.from` function.

Another somewhat confusing part is in change 4, where we `forEach` our way through the map. The odd part is that our anonymous function is using two parameters instead of one: `_count` and `label`.

The `label` is the key of our map, and the value is the `_count`. The underscore is there to signify that, although we must include something in the first spot of the parameter list in order to label and access the second one; the first is unused. Some would use just an `_`, but there is no good reason not to name the variable something useful. Should someone later using this need to look up the function definition to realize what the first parameter means, or use `console.log` to find its value? Also, if the first parameter does turn out to be useful, it's convenient to just delete the underscore that prepends a perfectly useful and descriptive variable name.

WHY DID THEY DO THAT?

The ordering of the parameters in the `forEach` function of `Map` is backward in that people usually describe hashes/dictionaries as "key/value pairs," and here the value is listed first.

It might be nicer if it were `(key, value)`, rather than `(value, key)`, but I suppose the assumption is that people will more often be interested in strictly the `value`, meaning they would tend to call it with one parameter: `(value)`.

In any case, stay safe out there.

The other changes just reflect differences in getting attributes, `.get(`*`thing`*`)` versus `[`*`thing`*`]`, and in setting attributes, `.set(`*`thing, newValue`*`)` versus `[`*`thing`*`] = `*`newValue`*.

Through the same approach, you can convert `labelProbabilities` to a map as well. Make the following changes:

```
- var labelProbabilities = {};
+ var labelProbabilities = new Map();

- labelProbabilities[label] = labelCounts.get(label) / songs.length;
+ labelProbabilities.set(label, labelCounts.get(label) / songs.length);

- Object.keys(labelProbabilities).forEach(function(difficulty){
-   var first = labelProbabilities[difficulty] + smoothing;
+ labelProbabilities.forEach(function(_probabilities, difficulty){
+   var first = labelProbabilities.get(difficulty) + smoothing;
```

At this point, you should still have code that runs and gives you the numbers you expect.

The other top-level objects (`chordCountsInLabels` and `probabilityOfChordsInLabels`) are a bit trickier to convert into maps. This is mostly because their state is global and mutable. They are also a bit more resistant to change because the latter is initially assigned to the former.

We can apply the same approach as before, although it's more finicky this time. We will need the following changes:

```
-var chordCountsInLabels = {};
-var probabilityOfChordsInLabels = {};
+var chordCountsInLabels = new Map();
+var probabilityOfChordsInLabels = new Map();

-if(chordCountsInLabels[song.label] === undefined){
-  chordCountsInLabels[song.label] = {};
+if(chordCountsInLabels.get(song.label) === undefined){
+  chordCountsInLabels.set(song.label, {});

-if(chordCountsInLabels[song.label][chord] > 0){
-  chordCountsInLabels[song.label][chord] += 1;
+if(chordCountsInLabels.get(song.label)[chord] > 0){
+  chordCountsInLabels.get(song.label)[chord] += 1;

-chordCountsInLabels[song.label][chord] = 1;
+chordCountsInLabels.get(song.label)[chord] = 1;

-  Object.keys(probabilityOfChordsInLabels).forEach(
-function(difficulty){
-  Object.keys(probabilityOfChordsInLabels[difficulty]).forEach(
-    probabilityOfChordsInLabels[difficulty][chord] /= songs.length;
+probabilityOfChordsInLabels.forEach(function(_chords, difficulty){
+ Object.keys(probabilityOfChordsInLabels.get(difficulty)).forEach(
+  probabilityOfChordsInLabels.get(difficulty)[chord] /= songs.length;

-var probabilityOfChordInLabel =
-probabilityOfChordsInLabels[difficulty][chord];
```

```
+var probabilityOfChordInLabel =
+probabilityOfChordsInLabels.get(difficulty)[chord];
```

Save/run/check/commit.

"WEAK" VERSIONS OF SET AND MAP

Before we leave our discussion of sets and maps, you should note that there are also `WeakSet` and `WeakMap`. The main differences between them and their "strong" (normal strength?) counterparts are:

- They cannot be iterated (no `forEach` function).
- They do not have a reference to their size.
- `WeakSet` cannot store primitives.
- They hold their keys "weakly"; that is, the keys are available for garbage collection when they don't have any references.

Basically, with the weak forms, you give up the capability of easily knowing what is inside or applying functions to the whole set. And what you gain is control over memory leaks and privacy.

Array Alternative: Bit Fields

One more candidate for replacing arrays deserves a mention: bit fields. If you have an array that stores booleans, you might want bit fields in some cases. There is no native implementation of bit fields in JavaScript. However, you have access to numbers and bit-wise arithmetic, and that's all you need.

Imagine you had the following conditionals:

```
states = [true,
          true,
          true,
          true,
          true,
          true,
          false,
          true]
```

You could also represent these in binary as `0b11111101`.

If you had a conditional that was only valid under these conditions, you could do something like this:

```
if(state[0] && state[1] && state[2] && state[3] && state[4]
&& state[5] && !state[6] && state[7]){
  // something something
```

Because these states have little meaning by themselves, one potential refactoring would be to move these conditions into a function:

```
if(stateIsOk()){
  // something something
...

stateIsOk = function(state){
  return state[0] && state[1] && state[2] && state[3] && state[4] &&
    state[5] && !state[6] && state[7]
}
```

But if you stored your state in a bit field, you could do this instead:

```
if(state===0b11111101){
  // something something

// or you can give this state a more specific name

if(stateIsOk()){
  // something something
...

stateIsOk = function(state){
  return state===0b11111101;
}
```

This has more potential for performance optimization than refactoring, because bit-wise arithmetic is super fast, but it's a little tricky to work with in many applications. If you're doing something that is graphically intensive and/or needs to be fast (like a game), keep this array-like representation in mind.

Extracting this function is nice because we can easily describe (via the function name) what `0b11111101` actually means. Speaking of extracting functions, we're about to do a whole bunch of that in the next section.

Testing What We Have

For our changes so far, we could limp along without needing to pull in a testing framework, instead manually testing with `console.log`. Everyone has a differ-

ent tolerance for how big and complex a project should be before tooling up certain aspects, including testing. For some, visually inspecting the correct output might have been enough of a hassle to motivate testing early on. Others might have felt a lack of confidence due to not testing low-level functions from the very start.

On the other hand, there is a case for sticking with high-level tests (even manual ones like ours) until your code begins to take a bit more shape. Many tests that cover low-level refactorings may provide extra work without extra confidence.

For instance, if you know that you have dead code, do you want to test it before you delete it? If you have a function that you know you want to rename, do you really want to test it beforehand?

FOUR MORE TESTING PHILOSOPHIES

The combination of unit and high-level testing in JavaScript (including testing private functions when you feel like, and using TDD when it works well for you and your team) is fairly mainstream, although this book emphasizes characterization tests because they are terribly underrated.

It is a good approach to be comfortable with, and tends to mesh well with most teams. However, there are a few other testing philosophies that are worth considering:

Unit tests are a waste of time (only high-level tests/metrics matter)

I've never heard this from a competent engineer who is currently working on coding tasks, rather than as a manager or consultant focused more on high-level goals.

TATFT (Test All The [Redacted] Time)

In this approach, you test it all. You test your dependencies. You test your tests. You test *everything*. I've never personally seen people adapt this in practice, but I've definitely seen it inspire people to write more tests.

If it compiles (or *type checks*), it works

For languages that care about type safety (like Haskell or the compiles-to-JS variant, PureScript), many errors simply won't happen, because the errors will come up at compile time. However, even in the case of type-safe, idempotent, pure functions, logic errors are still possible, so using a functional language doesn't actually let you off the hook for testing.

UUM (Uses Until Modification)

This is a nuanced view on testing that suggests the degree to which you test code should be proportional to how much it will be used before it is modified. If it's in your editor, actively being worked on, then it will be used zero times before it's

modified, so no test is necessary. If it's headed out into the world where millions of people will use it (whether it's a developer library or something facing the general public), then it should have more tests. The **original post** (http://bit.ly/uum-post) is only a few bullet points long, and worth a look.

In this chapter, our changes are much more aggressive. In order to approach those changes with confidence, we're going to need more tests. Before we make any more changes, we're going to test what we have now.

In case it isn't obvious what we want to test, the premise is that we want to convert our manual workflow into an automated one. That means that anything that we've been looking to for output should be inside of a test.

CODE WANTING TO BE BETTER

If you look at code in a certain way, you can see how it *wants* to be better. Long files want to be shorter. Comments explaining a few lines of code want to be functions, named with the comment, that wrap up those few lines. As it applies to us right now, these logging statements want to be functions.

Our Setup Test

Before we start testing the actual behavior of our code, it's useful to have a test to just ensure that everything is working properly.

Add this to the bottom of your file:

```
var wish = require('wish');
describe('the file', function() {
  it('works', function(){
    wish(true);
  });
});
```

Note that you may also need to run these commands from the command line:

```
npm install -g mocha
npm install wish
```

Now if you run **`mocha nb.js`** you should see a passing test as well as our logging information.

DON'T FORGET ABOUT THE WATCHER

Don't forget about using `mocha -w nb.js` to run the watcher in a terminal window to update the results when you make changes.

Characterization Tests for classify

Instead of using `console.log` at the end of functions, we can actually return something from them that we can test. Add this inside the `describe` block (the `classify` line is the same as before):

```
it('classifies', function(){
  classify(['f#m7', 'a', 'dadd9', 'dmaj7', 'bm', 'bm7', 'd', 'f#m']);
});
```

Next, at the very end of the `classify` function, have it return in addition to logging:

```
function classify(chords){
...
  });
  console.log(classified);
  return classified;  // this line is new
};
```

Back to the test: we know that, since we're lacking coverage, we want to write a characterization test. Let's use the characterization mode of wish by supplying a second parameter of `true`, and let the test output tell us what the output is.

Make the "`classifies`" test look like this:

```
it('classifies', function(){
  var classified = classify(['f#m7', 'a', 'dadd9',
                             'dmaj7', 'bm', 'bm7', 'd', 'f#m']);
  wish(classified.get('easy'), true);
  wish(classified.get('medium'), true);
  wish(classified.get('hard'), true);
});
```

After running **`mocha`**, this leads to:

```
WishCharacterization: classified.get('easy')
  evaluated to 1.3433333333333333
```

And then we can just put that value into our test:

```
// replace this
wish(classified.get('easy'), true);
```

```
// with this
wish(classified.get('easy') === 1.3433333333333333);
```

Run **mocha** again and we get:

```
WishCharacterization: classified.get('medium')
  evaluated to 1.5060259259259259
```

Perfect. Again, we replace the `, true` with the output value:

```
wish(classified.get('medium') === 1.5060259259259259);
```

Save and run **mocha**. Once again, we get a characterization error:

```
WishCharacterization: classified.get('hard')
  evaluated to 1.6884223991769547
```

Now we have the information we need for the full test block:

```
it('classifies', function(){
  var classified = classify(['f#m7', 'a', 'dadd9',
                             'dmaj7', 'bm', 'bm7', 'd', 'f#m']);
  wish(classified.get('easy') === 1.3433333333333333);
  wish(classified.get('medium') === 1.5060259259259259);
  wish(classified.get('hard') === 1.6884223991769547);
});
```

Following that same process again, we can write a similar test for the other song we are classifying:

```
it('classifies again', function(){
  var classified = classify(['d', 'g', 'e', 'dm']);
  wish(classified.get('easy') === 2.023094827160494);
  wish(classified.get('medium') === 1.855758613168724);
  wish(classified.get('hard') === 1.855758613168724);
});
```

Now we can remove three things:

- Our logging statement (`console.log(classified);`) from the `classify` function
- The calls to `classify` that are outside of the tests
- The `"works"` test that we added to make sure our setup was okay

You might be wondering why we didn't just copy the values from the `console.log` statement and put them in the test. The reason is so we could have a

test failure first. If we don't see a failure, we can't be totally certain that the specific action we took is actually what made the test pass. "Things seem to be functioning correctly" is not as confident of a statement as "I changed the test expectation to match the result of the failing characterization test, and that turned the test from red to green." That might seem subtle, but small steps are easier to undo.

Testing the welcomeMessage

Now let's add a new test for the welcome message (inside of the `describe` block). This time, instead of leaving our logging statement in place, we'll move it into a test:

```
// delete this line from the file (near the top)
console.log(`Welcome to ${fileName()}!`);

// add this test inside of the describe block
it('sets welcome message', function(){
  console.log(`Welcome to ${fileName()}!`);
});
```

Right now, this passes, as there is no assertion made. But we still see the output from the logging statement, so we know the code still works. Let's add the assertion now:

```
it('sets welcome message', function(){
  console.log(`Welcome to ${fileName()}!`);
  wish(welcomeMessage() === 'Welcome to nb.js!') // this line is new
});
```

Note that we're intending to add a function here. This is not a characterization test. This is a unit test for a function that hasn't been written yet.

We get an error:

```
ReferenceError: welcomeMessage is not defined
```

Great. Let's define that function at the top of the file:

```
function welcomeMessage(){
  return `Welcome to ${fileName()}!`;
};
```

The test now passes. This means we no longer need to rely on the welcome message logging statement anywhere, including in our test, so we can just have the following:

```
it('sets welcome message', function(){
  wish(welcomeMessage() === 'Welcome to nb.js!')
});
```

Testing for labelProbabilities

Here, we'll take the same approach of moving the logging statement into a test, and then making it into a proper assertion. To start, delete this line from the `classify` function:

```
console.log(labelProbabilities);
```

And add this code inside of the `describe` block:

```
it('label probabilities', function(){
  console.log(labelProbabilities);
});
```

We can change that to a characterization test like this:

```
it('label probabilities', function(){
  wish(labelProbabilities, true);
});
```

When we run the test, our output tells us what we need to do next:

```
WishCharacterization: labelProbabilities
evaluated to [["easy",0.3333333333333333],
              ["medium",0.3333333333333333],
              ["hard",0.3333333333333333]]
```

Since we know that we're dealing with `labelProbabilities` as a set, we can test individual components like this:

```
it('label probabilities', function(){
  wish(labelProbabilities.get('easy') === 0.3333333333333333);
  wish(labelProbabilities.get('medium') === 0.3333333333333333);
  wish(labelProbabilities.get('hard') === 0.3333333333333333);
});
```

Now that we have high-level tests in place, we can refactor aggressively and with confidence. Also, we're completely free from `console.log` statements.

Extracting Functions

Here we go! This section is about what, as we said in the last chapter, is likely the most useful and underutilized refactoring technique: extracting functions.

Getting Away from Procedural Code

If we look through the code at this point, we might notice that we have data (songs and labels) mixed in with our functions. Overall, our program looks like it has these steps:

1. Set up data (songs and labels).
2. Set up objects, sets, and maps.
3. Train our classifier on the set of songs.
4. Set counts and probabilities.
5. Classify with new songs (handled by the tests).

Right now, the first four steps are still spelled out awkwardly in a procedural and unstructured way.

We want to get away from *running the file*, and start to think about *executing functions*. That means that anything not inside of a function or the testing code is a problem.

The following functions are always run together:

```
        setLabelProbabilities();
        setChordCountsInLabels();
        setProbabilityOfChordsInLabels();
```

That means we can wrap them in another function and run that immediately, like this:

```
function setLabelsAndProbabilities(){
  setLabelProbabilities();
  setChordCountsInLabels();
  setProbabilityOfChordsInLabels();
};
setLabelsAndProbabilities();
```

Note that we could also run this as an anonymous IIFE (immediately invoked function expression; see **"Function Calls and Function Literals"**), like this:

```
(function(){
  setLabelProbabilities();
  setChordCountsInLabels();
  setProbabilityOfChordsInLabels();
})();
```

But in doing that we lose some control. If we ever decided to call this more than once, we'd have to name it or find a way to run the code containing it each time.

For now, it doesn't actually matter when this function is called, as long as it is called just once: *after* the classifier is trained, but before we run `classify`.

Let's add a `trainAll` function and call `setLabelsAndProbabilities` at the end of it. We won't call it anywhere else. Also, let's add a call to `trainAll` right after its declaration:

```
function trainAll(){
  train(imagine, easy);
  train(somewhereOverTheRainbow, easy);
  train(tooManyCooks, easy);
  train(iWillFollowYouIntoTheDark, medium);
  train(babyOneMoreTime, medium);
  train(creep, medium);
  train(paperBag, hard);
  train(toxic, hard);
  train(bulletproof, hard);
  setLabelsAndProbabilities();
};

trainAll();

function setLabelsAndProbabilities(){
  setLabelProbabilities();
  setChordCountsInLabels();
```

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

We want `trainAll` to be part of our public interface (a *function* that we run, rather than *part of a file* that we run) just like any other statement, so we should move it to be called right inside of the `describe` block of our tests:

```
describe('the file', function() {
  trainAll();
```

Next, we can extract a function for setting our songs and (for now) call it right away:

```
function setSongs(){
  imagine = ['c', 'cmaj7', 'f', 'am', 'dm', 'g', 'e7'];
  somewhereOverTheRainbow = ['c', 'em', 'f', 'g', 'am'];
  tooManyCooks = ['c', 'g', 'f'];
  iWillFollowYouIntoTheDark = ['f', 'dm', 'bb', 'c', 'a', 'bbm'];
  babyOneMoreTime = ['cm', 'g', 'bb', 'eb', 'fm', 'ab'];
  creep = ['g', 'gsus4', 'b', 'bsus4', 'c', 'cmsus4', 'cm6'];
  paperBag = ['bm7', 'e', 'c', 'g',
              'b7', 'f', 'em', 'a',
              'cmaj7', 'em7', 'a7', 'f7',
              'b'];
  toxic = ['cm', 'eb', 'g', 'cdim', 'eb7', 'd7', 'db7', 'ab',
           'gmaj7', 'g7'];
  bulletproof = ['d#m', 'g#', 'b', 'f#', 'g#m', 'c#'];
};
setSongs();
```

We know that we want this to happen once before training, so we can move our `setSongs` function call (the last line of the previous snippet) to be inside `trainAll`:

```
function trainAll(){
  setSongs();
...
```

As always, after each of these changes, you should be noting the passing test suite and committing the results.

We want to do the same with the setup of difficulty variables and our containers (array, sets, and maps) that we use throughout the program, but if we take the same approach of extracting a function, our test suite will give an error. Try this:

```
function setDifficulties(){
  var easy = 'easy';
  var medium = 'medium';
  var hard = 'hard';
};
setDifficulties();
```

If we do this, our variables are stuck inside that function's scope, and no longer readable by other functions. Later, we'll be addressing how to tighten up the scopes for these variables, but for now, we can take the easy way out and leave these as global variables by omitting the `var` keyword:

```
function setDifficulties(){
  easy = 'easy';
  medium = 'medium';
  hard = 'hard';
};
setDifficulties();
```

And we can do the same for our other global variables (move them into a function, remove the scoping declaration, and call the function):

```
function setup(){
  songs = [];
  allChords = new Set();
  labelCounts = new Map();
  labelProbabilities = new Map();
  chordCountsInLabels = new Map();
  probabilityOfChordsInLabels = new Map();
};
setup();
```

Next, we can move the function calls into our `trainAll` function:

```
function trainAll(){
  setDifficulties();
  setup();
  setSongs();
...
```

Make sure to remove those three calls (`setDifficulties`, `setup`, and `setSongs`) from other places in the program. Assuming you've done that, now everything is either inside of a function or the test code.

OH NO! VAR WAS DELETED! EVERYTHING IS WORSE NOW!

And we didn't even have `var` declarations for our songs to begin with! Bad code!

Right?

We'll talk about `var` and other scoping declarations later. For now, recognize that our program relied on these as global variables before. We're just no longer using the `var` keyword to pretend they're not global.

One way to fix this is by specifically scoping and passing them into each function that requires them. Done naively, this path would require a lot of changes, including more complex method signatures (more parameters) and more return values.

At some point, we'll want to attach these variables to some object other than the global one, but for now, it's a win to have the interface well defined by the tests: they just run `trainAll` and `classify`, which means we're more concerned with the *contents* of the file as opposed to the *order*.

Great. So now everything is inside of a function and our interface for the tests is well defined. We could extract more functions if we wanted to get more specific in places. For instance, we could change our `setup` function to do something like this instead (but don't do this):

```
function setSongsVariable(){
  songs = [];
};
function setup(){
  setSongsVariable();
  allChords = new Set();
  labelCounts = new Map();
  labelProbabilities = new Map();
  chordCountsInLabels = new Map();
  probabilityOfChordsInLabels = new Map();
};
```

And we could do that with the rest of the lines in `setup` as well. Or we could use it to arbitrarily group some lines inside (don't do this either):

```
function setSome(){
  songs = [];
  allChords = new Set();
  labelCounts = new Map();
};
```

```
function setOthers(){
  labelProbabilities = new Map();
  chordCountsInLabels = new Map();
  probabilityOfChordsInLabels = new Map();
};
function setup(){
  setSome();
  setOthers();
};
```

We can add as much indirection as we'd like by extracting functions, and use this technique to group things where it makes sense. But these changes don't actually help our code be more clear. We're just grouping arbitrary lines at this point.

The opposite of *extracting* functions is *inlining* them. When an extracted function doesn't really do anything, it's sensible to inline it. If you did extract those functions, inline them now and restore the `setup` function to how it was:

```
function setup(){
  songs = [];
  allChords = new Set();
  labelCounts = new Map();
  labelProbabilities = new Map();
  chordCountsInLabels = new Map();
  probabilityOfChordsInLabels = new Map();
};
```

If you're wondering what the point of that was, it's that extracting and inlining functions should come as naturally to you as introducing and inlining variables.

Extracting and Naming Anonymous Functions

In addition to extracting functions helping to group behavior, it is also useful for naming anonymous functions. We have some anonymous functions in our program. For examples, look at any code that follows a `forEach`, or the second parameter to the `describe` and `it` function calls in the test.

We could extract some of these functions, but to demonstrate the benefits a bit more simply, let's take a look at some pedestrian jQuery code that visits a URL when a button is clicked:

```
$('.my-button').on('click', function(){
  window.location = "http://refactoringjs.com";
});
$('.other-button').on('click', function(){
```

```
  window.location = "http://refactoringjs.com";
});
```

Some people would correctly identify that duplication as a maintenance problem deserving of some kind of fix. Unfortunately, many of them would jump to this solution, simply extracting a variable, and stop there:

```
var siteUrl = "http://refactoringjs.com";
$('.my-button').on('click', function(){
  window.location = siteUrl;
});
$('.other-button').on('click', function(){
  window.location = siteUrl;
});
```

This does make it easier to change `siteUrl` in the future (assuming it is used in other places), but we can reduce duplication further by extracting a function:

```
var siteUrl = "http://refactoringjs.com";
function visitSite(){
  window.location = siteUrl;
}
$('.my-button').on('click', function(){
  visitSite();
});
$('.other-button').on('click', function(){
  visitSite();
});
```

Now our implementation will be easier to change in the future, but what good does wrapping our function call in a function accomplish? Nothing! Try this instead:

```
var siteUrl = "http://refactoringjs.com";
function visitSite(){
  window.location = siteUrl;
}
$('.my-button').on('click', visitSite);
$('.other-button').on('click', visitSite);
```

Now we can keep our click handlers together. This organization is much better.

Function Calls and Function Literals

For programmers who are new to JavaScript or those who mostly work in some other language on the backend, there is often some confusion over function syntax that makes this type of refactoring difficult for them.

This is an anonymous function literal:

```
function(){};
```

This is a named function literal:

```
function visitSite(){};
```

This is an anonymous function literal assigned to a variable:

```
var visitSite = function(){};
```

This is a function call:

```
visitSite();
```

The only way to call an anonymous function (the first one, when it is not assigned to a variable as in the third snippet) is by running the containing code:

```
(function(){})();
// or
(function(){}());
```

This is called an *IIFE* ("iffy"), or *immediately invoked function expression*. And it is a function call (i.e., *invocation*).

The confusion arises when people don't realize that an anonymous function declaration, when replaced with a named function for reuse, can be used in the same way as a function *reference*:

```
$('.my-button').on('click', visitSite);
```

and *not* with a function call like this:

```
$('.my-button').on('click', visitSite());
```

You should be passing a reference to the `visitSite` function into the `on` function, not a function call.

Additional confusion arises when a parameter is used by the function being referenced. If our function looked like this:

```
function visitSite(siteUrl){
  window.location = siteUrl;
};
```

some people would be tempted to write the click handler like this:

```
$('.my-button').on('click', visitSite("http://refactoringjs.com"));
```

But that won't work. In this case, the most obvious solution would be using an anonymous function to wrap the call like before:

```
$('.my-button').on('click', function(){
  visitSite("http://refactoringjs.com");
});
```

Another solution, which applies not only to the `on` function of jQuery but to native JavaScript functions like `forEach` and `map` as well, is to pass another argument along with the others. In many cases, what you pass will end up being the "implicit" parameter (the `this`) of the function that you are calling, but if you are defining a function taking a function as an argument, you could also design the function signature to take arguments that will become explicit arguments to the function passed.

In any case, when you are passing functions inside of functions, you should keep two things in mind. First, consult the API of the function that takes a function as an argument for any optional arguments that can be passed in, and know what happens when they are not. Second, recognize that JavaScript function calls will work fine with too many or too few arguments. You will get an error *inside* of the function if you try to call it and make use of a parameter that was not defined by the signature. Also, any parameters that are unfulfilled by the function call will be set as `undefined`.

Streamlining the API with One Global Object

At this point, we have everything wrapped up into functions, but we still have many global variables floating around (this includes our functions as well as variables set without `var`). In this section, we have two goals. The first is to reduce the scope of all of the global variables we have. The second is to design our API—in other words, make decisions about how our code will be used.

For the style of code we're exploring here, we want one main object, which we will call `classifier`, and it will hold all of the functions and incidental variables created along the way. We'll explore its creation through multiple mechanisms (factory function, constructor function, and class), and in all cases, we'll

be working with it in an OOP fashion. That means we'll be using language constructs like `this` and allowing our `classifier` object to mutate throughout the execution of our program.

If you'd prefer to work toward a functional approach, rather than an OOP one, some of what follows will not follow the path you want to go. For that path, I would recommend still reading through the rest of this chapter and then looking into **Chapter 11**. But the time we spend "`this`-ifying" for the OOP style here will need extra work to convert to FP style. The reason is that OOP will lead us to having a main mutable object (although it could reasonably break down into smaller objects), whereas in FP, we want to treat our bounding object (`classifier`) as more of a dumb namespace for functions.

A functional version of our NBC is provided in **Chapter 11**. If you're interested in getting to that point, try starting from the code we have here as well as the code you have at the end of this chapter. I suspect you'll find code that is split into functions (but not as heavily invested in OOP as the code at the end of this chapter), like we have at this point, easier to work with. By all means, try both approaches if that interests you.

For the rest of the chapter, we'll be ignoring the `welcomeMessage` and `fileName` functions, as well as the accompanying test. In case you're lost at this point, or want to build an FP version starting from what we have now, here is the code:

```
function setDifficulties(){
  easy = 'easy';
  medium = 'medium';
  hard = 'hard';
};

function setSongs(){
  imagine = ['c', 'cmaj7', 'f', 'am', 'dm', 'g', 'e7'];
  somewhereOverTheRainbow = ['c', 'em', 'f', 'g', 'am'];
  tooManyCooks = ['c', 'g', 'f'];
  iWillFollowYouIntoTheDark = ['f', 'dm', 'bb', 'c', 'a', 'bbm'];
  babyOneMoreTime = ['cm', 'g', 'bb', 'eb', 'fm', 'ab'];
  creep = ['g', 'gsus4', 'b', 'bsus4', 'c', 'cmsus4', 'cm6'];
  paperBag = ['bm7', 'e', 'c', 'g',
              'b7', 'f', 'em', 'a',
              'cmaj7', 'em7', 'a7', 'f7',
              'b'];
  toxic = ['cm', 'eb', 'g', 'cdim', 'eb7', 'd7', 'db7', 'ab',
           'gmaj7', 'g7'];
  bulletproof = ['d#m', 'g#', 'b', 'f#', 'g#m', 'c#'];
};

function setup(){
```

```
  songs = [];
  allChords = new Set();
  labelCounts = new Map();
  labelProbabilities = new Map();
  chordCountsInLabels = new Map();
  probabilityOfChordsInLabels = new Map();
};

function train(chords, label){
  songs.push({label, chords});
  chords.forEach(chord => allChords.add(chord));
  if(Array.from(labelCounts.keys()).includes(label)){
    labelCounts.set(label, labelCounts.get(label) + 1);
  } else {
    labelCounts.set(label, 1);
  }
};

function setLabelProbabilities(){
  labelCounts.forEach(function(_count, label){
    labelProbabilities.set(label,
                           labelCounts.get(label) / songs.length);
  });
};

function setChordCountsInLabels(){
  songs.forEach(function(song){
    if(chordCountsInLabels.get(song.label) === undefined){
      chordCountsInLabels.set(song.label, {});
    }
    song.chords.forEach(function(chord){
      if(chordCountsInLabels.get(song.label)[chord] > 0){
        chordCountsInLabels.get(song.label)[chord] += 1;
      } else {
        chordCountsInLabels.get(song.label)[chord] = 1;
      }
    });
  });
}

function setProbabilityOfChordsInLabels(){
  probabilityOfChordsInLabels = chordCountsInLabels;
  probabilityOfChordsInLabels.forEach(function(_chords, difficulty){
    Object.keys(probabilityOfChordsInLabels.get(difficulty)).forEach(
function(chord){
      probabilityOfChordsInLabels.get(difficulty)[chord]
/= songs.length;
    });
  });
}
```

```
function trainAll(){
  setDifficulties();
  setup();
  setSongs();
  train(imagine, easy);
  train(somewhereOverTheRainbow, easy);
  train(tooManyCooks, easy);
  train(iWillFollowYouIntoTheDark, medium);
  train(babyOneMoreTime, medium);
  train(creep, medium);
  train(paperBag, hard);
  train(toxic, hard);
  train(bulletproof, hard);
  setLabelsAndProbabilities();
};

function setLabelsAndProbabilities(){
  setLabelProbabilities();
  setChordCountsInLabels();
  setProbabilityOfChordsInLabels();
};

function classify(chords){
  var smoothing = 1.01;
  var classified = new Map();
  labelProbabilities.forEach(function(_probabilities, difficulty){
    var first = labelProbabilities.get(difficulty) + smoothing;
    chords.forEach(function(chord){
      var probabilityOfChordInLabel =
probabilityOfChordsInLabels.get(difficulty)[chord];
      if(probabilityOfChordInLabel){
        first = first * (probabilityOfChordInLabel + smoothing);
      }
    });
    classified.set(difficulty, first);
  });
  return classified;
};

var wish = require('wish');
describe('the file', function() {
  trainAll();
  it('classifies', function(){
    var classified = classify(['f#m7', 'a', 'dadd9',
                               'dmaj7', 'bm', 'bm7', 'd', 'f#m']);
    wish(classified.get('easy') === 1.3433333333333333);
    wish(classified.get('medium') === 1.5060259259259259);
    wish(classified.get('hard') === 1.6884223991769547);
  });
```

```
  it('classifies again', function(){
    var classified = classify(['d', 'g', 'e', 'dm']);
    wish(classified.get('easy') === 2.023094827160494);
    wish(classified.get('medium') === 1.855758613168724);
    wish(classified.get('hard') === 1.855758613168724);
  });
  it('label probabilities', function(){
    wish(labelProbabilities.get('easy') === 0.3333333333333333);
    wish(labelProbabilities.get('medium') === 0.3333333333333333);
    wish(labelProbabilities.get('hard') === 0.3333333333333333);
  });
});
```

Extracting the classifier Object

To our global variables (and functions) we'll create an object called `classifier` to store them, so that they're not attached directly to the global object. Put this the top of your file:

```
var classifier = {};
```

Then, inside of `trainAll`, instead of `setup` being run as a function of the global object, call it as part of this `classifier`:

```
function trainAll(){
  classifier.setup(); // this line is new
  setDifficulties();
  setup(); // get rid of this one
...
```

Now we need to move our `setup` function into our `classifier` object, at the top of the file. You can also delete the old `setup` function:

```
var classifier = {
  setup: function(){
    this.songs = [];
    this.allChords = new Set();
    this.labelCounts = new Map();
    this.labelProbabilities = new Map();
    this.chordCountsInLabels = new Map();
    this.probabilityOfChordsInLabels = new Map();
  };
};
```

We want these variables to be part of `classifier`, so we need to add a `this.` in front of each of them. Notice that `setup` is a property with the label (`setup`) followed by a function literal that contains the body of the function.

It would be great if the tests passed at this point, but they don't. Fortunately, running the tests will report a useful error:

```
ReferenceError: songs is not defined
```

And here we have some very boring and repetitive, but relatively easy, changes to make. We need to add a `classifier.` in front of every reference to the following variables (that don't start with `this.`): `songs`, `allChords`, `labelCounts`, `labelProbabilities`, `chordCountsInLabels`, and `probabilityOfChordsInLabels`. Make those changes now.

The tests should pass again. Save and commit those changes.

Inlining the setup Function

Now that we look at our `setup` function attached to `classifier`, however, we might notice that it's not doing much work. We can assign those variables directly to the object without using a wrapping function at all:

```
var classifier = {
  songs: [],
  allChords: new Set(),
  labelCounts: new Map(),
  labelProbabilities: new Map(),
  chordCountsInLabels: new Map(),
  probabilityOfChordsInLabels: new Map()
};
```

Notice that the function wrapping is gone, but the syntax inside has also been tweaked. For example, instead of this:

```
this.songs = [];
```

we now have this:

```
songs: [],
```

The comma versus semicolon is easy to miss. And we can also delete the call to `setup` in `trainAll`:

```
function trainAll(){
  classifier.setup(); // delete this line
...
```

That's great, not only because we have one less line, but also because our training code really doesn't have anything to do with the general setup of our `classifier`.

Save/test/commit to make sure everything still looks good.

Extracting the songList Object

Next let's consider how we're adding songs to be trained. `setSongs` defines global variables for each song name, and then each of those is referenced in the `trainAll` function. This coupling is error prone and not really scalable. Clearly, if we alter our list of songs, we have to do it in two places. But worse than that, we are not prepared for a likely case where we would have a *database* of songs to train, rather than a handful that we hardcode into a *file*.

Let's make a `songList` object that contains the songs in an array and has a function to add songs to it. This can go at the top of your file:

```
var songList = {
  songs: [],
  addSong: function(name, chords, difficulty){
    this.songs.push({name: name,
                     chords: chords,
                     difficulty: difficulty});
  }
};
```

Now let's change our `setSongs` function to make use of this `songList` object:

```
function setSongs(){
  songList.addSong('imagine',
['c', 'cmaj7', 'f', 'am', 'dm', 'g', 'e7'], easy)
  songList.addSong('somewhereOverTheRainbow',
['c', 'em', 'f', 'g', 'am'], easy)
  songList.addSong('tooManyCooks', ['c', 'g', 'f'], easy)
  songList.addSong('iWillFollowYouIntoTheDark',
['f', 'dm', 'bb', 'c', 'a', 'bbm'], medium);
  songList.addSong('babyOneMoreTime',
['cm', 'g', 'bb', 'eb', 'fm', 'ab'], medium);
  songList.addSong('creep',
['g', 'gsus4', 'b', 'bsus4', 'c', 'cmsus4', 'cm6'], medium);
  songList.addSong('paperBag',
['bm7', 'e', 'c', 'g', 'b7', 'f', 'em',
```

```
'a', 'cmaj7', 'em7', 'a7', 'f7',
'b'], hard);
  songList.addSong('toxic',
['cm', 'eb', 'g', 'cdim', 'eb7',
'd7', 'db7', 'ab', 'gmaj7', 'g7'], hard);
  songList.addSong('bulletproof',
['d#m', 'g#', 'b', 'f#', 'g#m', 'c#'], hard);
};
```

If we run the tests now, we get errors because our variables named by the song names are no longer available. We need to change our `trainAll` function to add the songs from the `songList`:

```
function trainAll(){
  setDifficulties();
  setSongs();
  songList.songs.forEach(function(song){
    train(song.chords, song.difficulty);
  });
  setLabelsAndProbabilities();
};
```

And now our tests are back in working order.

Handling the Remaining Global Variables

We're still left with those three nagging global variables to represent difficulty.

As a first effort, since they are only referenced inside of `setSongs`, we can simply inline the variables there:

```
function setSongs(){
  var easy = 'easy';
  var medium = 'medium';
  var hard = 'hard';
...
```

And now we can delete the function `setDifficulties` and our call to it inside of `trainAll`. Tests should be passing at this point.

Following that, it might seem a waste to complicate our `setSongs` function with these variables. What if we just used an array for `difficulties`, and let the `songList` handle the strings?

```
var songList = {
  difficulties: ['easy', 'medium', 'hard'],
  songs: [],
```

```
  addSong: function(name, chords, difficulty){
    this.songs.push({name: name,
                     chords: chords,
                     difficulty: this.difficulties[difficulty]})
  }
};
```

Here, we've created a `difficulties` attribute in `songList` as an array with the desired labels. Then, when we add a song, we're expecting the array index to come through and assigning as needed with `this.difficulties[difficulty]`.

This means we can get rid of the label description noise in `setSongs` and use numbers instead of the variables we were using before. Notice the numbers as the third parameter of the `addSong` function calls:

```
function setSongs(){
  songList.addSong('imagine',
['c', 'cmaj7', 'f', 'am', 'dm', 'g', 'e7'], 0);
  songList.addSong('somewhereOverTheRainbow',
['c', 'em', 'f', 'g', 'am'], 0);
  songList.addSong('tooManyCooks', ['c', 'g', 'f'], 0);
  songList.addSong('iWillFollowYouIntoTheDark',
['f', 'dm', 'bb', 'c', 'a', 'bbm'], 1);
  songList.addSong('babyOneMoreTime',
['cm', 'g', 'bb', 'eb', 'fm', 'ab'], 1);
  songList.addSong('creep',
['g', 'gsus4', 'b', 'bsus4', 'c', 'cmsus4', 'cm6'], 1);
  songList.addSong('paperBag',
['bm7', 'e', 'c', 'g', 'b7', 'f', 'em',
 'a', 'cmaj7', 'em7', 'a7', 'f7',
 'b'], 2);
  songList.addSong('toxic',
['cm', 'eb', 'g', 'cdim', 'eb7', 'd7', 'db7', 'ab', 'gmaj7', 'g7'], 2);
  songList.addSong('bulletproof',
['d#m', 'g#', 'b', 'f#', 'g#m', 'c#'], 2);
};
```

Save/test/commit.

Making Data Independent from the Program

Let's think about that `trainAll` function for a minute. What does *setting* songs have to do with *training* our classifier? Nothing at all, really. What if we move the call to `setSongs` into our test?

```
describe('the file', function() {
  setSongs(); // moved and deleted from inside of trainAll
  trainAll();
```

And while we're at it, does setting the songs have anything to do with the *structure* of our program at all? No. It's only data that we are setting and using. That means it is part of the execution of one possibility of our program, not the program itself. That implies that the function of `setSongs` itself belongs in the tests. Let's inline the function body into the tests and remove the function from the program completely:

```
describe('the file', function() {
  songList.addSong('imagine',
['c', 'cmaj7', 'f', 'am', 'dm', 'g', 'e7'], 0);
  songList.addSong('somewhereOverTheRainbow',
['c', 'em', 'f', 'g', 'am'], 0)
  songList.addSong('tooManyCooks', ['c', 'g', 'f'], 0);
...
  songList.addSong('bulletproof',
['d#m', 'g#', 'b', 'f#', 'g#m', 'c#'], 2);
  trainAll();
...
```

Now we're free to execute the program independently of the data provided. This is a huge step, and opens our program up to further testing and putting it into production as a module.

Scoping Declarations: var, let, and const

Moving on to other scoping concerns, let's talk about scoping declarations: `var`, `let`, and `const`. `var` has been around for the longest, so you're most likely to find it in older codebases or code written by people who are stuck in an older mindset. Although `var` is better than nothing (a variable declaration, and necessarily an assignment, without any scoping declaration will create a global variable), `let` and `const` create tighter scopes (block scope rather than just functional scope). The difference between `let` and `const` is that `const` will not allow a variable to be reassigned.

We have nine instances of `var` in our program right now. Should we change any of them? If so, should they change to `let` or `const`?

CONST DOES NOT MEAN IMMUTABILITY!

If you are thinking that `const` provides an easy path to immutability, I'm sorry to report some sad news. It does prevent a variable from being reassigned (i.e., through the = operator), but the contents of the variable can still change. Array indexes can be updated. The same goes for object attributes and members of sets and maps.

`Object.freeze` to the rescue!? Almost. If you freeze an object that is more than one level deep, you can still update those inner properties. Also, even if you freeze an object, if it was declared with `var` or `let`, you can still reassign it.

To *ensure* immutability, you'll need to go further, and might find it easier to use a packaged solution like Immutable.js or mori, which provide immutable versions of arrays, maps, sets, and so on. Even if you're not using anything that enforces it, it's best to try to create new variables rather than repurposing them.

In general, it's preferable to use `const`. The less that your variables are reassigned, the better.

Since we have tests in place, we can make the bold assumption that all of our `var` declarations should be `const` instead. Search and replace those, and then run the tests. You'll get an error:

```
TypeError: Assignment to constant variable.
```

The error description "constant variable" is worth noting as an odd oxymoron, but let's move on. Unsurprisingly, our bold assumption is wrong, but only in one case. Our `first` variable inside of the `classify` function requires reassignment for now, so the following line should be modified to use `let`:

```
const first = classifier.labelProbabilities.get(difficulty) +
smoothing;
```

It should be:

```
let first = classifier.labelProbabilities.get(difficulty) +
smoothing;
```

All the rest of the `const` declarations are fine, and we're free to move on.

Bringing classify into the classifier

With a few changes, the variable `first` doesn't have to be reassigned, though. Initially, it is meant to reflect the likelihood of a difficulty appearing relative to

other difficulties. Later, it is multiplied by the likelihood of a chord existing in a song of a given difficulty.

What if instead of reassigning that variable, we introduced a `likelihoods` array that would capture all of the values we need to multiply together, and then multiplied them?

```
function classify(chords){
  const smoothing = 1.01;
  const classified = new Map();
  classifier.labelProbabilities.forEach(
function(_probabilities, difficulty){
    const likelihoods = [classifier.labelProbabilities.get(difficulty)
+ smoothing];
    chords.forEach(function(chord){
      const probabilityOfChordInLabel =
classifier.probabilityOfChordsInLabels.get(difficulty)[chord]
      if(probabilityOfChordInLabel){
        likelihoods.push(probabilityOfChordInLabel + smoothing)
      }
    })
    const totalLikelihood = likelihoods.reduce(function(total, index) {
      return total * index;
    });
    classified.set(difficulty, totalLikelihood);
  });
  return classified;
};
```

We've mentioned, but haven't gone into depth on, the `reduce` function . It allows us to step through an array and apply some function to it, while working with a returned value, along with each element.

But wait! We already have an array that we're looping through (`chords`). Why would we create a new one to step through? Do we really need to loop twice (once to accumulate and once to multiply)? Let's try using `reduce` on that `forEach` instead:

```
function classify(chords){
  const smoothing = 1.01;
  const classified = new Map();
  classifier.labelProbabilities.forEach(
function(_probabilities, difficulty){

// reduce starts
    const totalLikelihood = chords.reduce(function(total, chord){
      const probabilityOfChordInLabel =
classifier.probabilityOfChordsInLabels.get(difficulty)[chord]
      if(probabilityOfChordInLabel){
```

```
          return total * (probabilityOfChordInLabel + smoothing)
        }else{
          return total;
        }
      }, classifier.labelProbabilities.get(difficulty) + smoothing)
  // reduce ends

      classified.set(difficulty, totalLikelihood);
    });
    return classified;
  };
```

This is a bit more complex, for two reasons. First, our initial likelihood is now at the end, just above this line: `//reduce ends`. This supplies the "initial value" to the `reduce` total. Second, the `else` branch of our `if` statement is back. The reason is that on every step through the `reduce` function, if nothing is returned (which would be the case if the `if` condition was `false` and we had no `else` branch), then `undefined` would be returned. Returning the `total` has the same effect as just moving on to the next element.

If that's confusing, think of it this way. Say you had a `reduce` function like this:

```
[2, 3, 4].reduce(function(result, element){ return result }, 10)
```

This function would simply return `10`. If no value was supplied as the second parameter (if the `, 10` was not there), it would return the first element: 2. This is because `result` is fed back into the function for each element. Contrast this with another simple use of `reduce`, which sums the values of an array:

```
[2, 3, 4].reduce(function(result, element){return result + element })
```

This will return 9. If it had an initial element of `10`, it would add that as well, returning 19.

Back to our example, one pattern that is especially common in JavaScript is to allow unmet conditions to return something outside of an `else`. For example:

```
if(probabilityOfChordInLabel){
  return total * (probabilityOfChordInLabel + smoothing);
}
return total;
```

If the `true` branch of the `if` statement isn't executed, the code will continue beyond it. But personally, I feel this style invites mismatched return types and

often visually privileges the less likely possibilities at the top of a function. Moreover, it less strongly signals two code paths than an explicit `if`/`else`, which (if only slightly) disguises the complexity of the code. For someone who is interested in refactoring and thus eliminating complexity, it can mask places that may be good candidates for change.

In any case, you will frequently see this style used for error handling like this:

```
function callback(error, response){
  if(error){
    return new Error(error);
  }
  // do something with the response
}
```

Now let's pick on the `classify` function a bit more. First of all, we should have no issue with tying it more closely to the `classifier` object. If there's one obvious thing that a classifier does, it's classifying.

To that end, let's move the `classify` function into the `classifier`:

```
const classifier = {
  songs: [],
  allChords: new Set(),
  labelCounts: new Map(),
  labelProbabilities: new Map(),
  chordCountsInLabels: new Map(),
  probabilityOfChordsInLabels: new Map(),
  classify: function(chords){
    const smoothing = 1.01;
    const classified = new Map();
    classifier.labelProbabilities.forEach(
function(_probabilities, difficulty){
      const totalLikelihood = chords.reduce(function(total, chord){
        const probabilityOfChordInLabel =
classifier.probabilityOfChordsInLabels.get(difficulty)[chord]
        if(probabilityOfChordInLabel){
          return total * (probabilityOfChordInLabel + smoothing);
        }else{
          return total;
        }
      }, classifier.labelProbabilities.get(difficulty) + smoothing);
      classified.set(difficulty, totalLikelihood);
    });
    return classified;
  }
};
```

Our tests will break at this point, because we need to replace instances of `classify` with `classifier.classify` in the tests. Change these lines:

```
const classified = classify(['f#m7', 'a', 'dadd9', 'dmaj7',
                             'bm', 'bm7', 'd', 'f#m']);
...
const classified = classify(['d', 'g', 'e', 'dm']);
```

to these:

```
const classified = classifier.classify(['f#m7', 'a', 'dadd9',
                                        'dmaj7', 'bm', 'bm7',
                                        'd', 'f#m']);
...
const classified = classifier.classify(['d', 'g', 'e', 'dm']);
```

Now that our tests are working again (save/test/commit), we still have some work to do with this function. It awkwardly refers to itself in the third person, in a sense. That is, rather than using `this`, it says its own name: `classifier`. There are three instances of that. Let's change those, making the function the following:

```
classify: function(chords){
  const smoothing = 1.01;
  const classified = new Map();
  this.labelProbabilities.forEach(
function(_probabilities, difficulty){
    const totalLikelihood = chords.reduce(function(total, chord){
      const probabilityOfChordInLabel =
this.probabilityOfChordsInLabels.get(difficulty)[chord]
      if(probabilityOfChordInLabel){
        return total * (probabilityOfChordInLabel + smoothing);
      }else{
        return total;
      }
    }, this.labelProbabilities.get(difficulty) + smoothing);
    classified.set(difficulty, totalLikelihood);
  });
  return classified;
}
```

The first instance of `this` will be fine, but the second and third will cause problems:

```
  const probabilityOfChordInLabel =
this.probabilityOfChordsInLabels.get(difficulty)[chord]
```

```
// and
}, this.labelProbabilities.get(difficulty) + smoothing)
```

This is because our `this` (implicit parameter) is set in the context of the function that contains these statements. The most common and oafish fix is this:

```
classify: function(chords){
  const smoothing = 1.01;
  const classified = new Map();
  const self = this;
  this.labelProbabilities.forEach(
function(_probabilities, difficulty){
    const totalLikelihood = chords.reduce(function(total, chord){
      const probabilityOfChordInLabel =
self.probabilityOfChordsInLabels.get(difficulty)[chord]
      if(probabilityOfChordInLabel){
        return total * (probabilityOfChordInLabel + smoothing);
      }else{
        return total;
      }
    }, self.labelProbabilities.get(difficulty) + smoothing);
    classified.set(difficulty, totalLikelihood);
  });
  return classified;
}
```

By setting a `self` variable (in **Chapter 5**, we used `that` instead), you can ensure you have access to the object you want, even within the inner functions. In **Chapter 5**, we learned about using `call`, `apply`, or `bind` to accomplish the same thing with a bit more elegance. In this case, since we are not calling these anonymous functions directly, only declaring them to be called by `forEach` and `reduce`, we cannot use `call` or `apply` to set the implicit argument, `this`. For this case, we must use `bind` or find another way.

As for `forEach`, we can rely on the ability of that function to accept a `thisArg` as a parameter. This allows us to set what we want `this` to be in the anonymous function:

```
this.labelProbabilities.forEach(
  function(_probabilities, difficulty){
    const totalLikelihood = chords.reduce(function(total, chord){
      const probabilityOfChordInLabel =
self.probabilityOfChordsInLabels.get(difficulty)[chord];
      if(probabilityOfChordInLabel){
        return total * (probabilityOfChordInLabel + smoothing);
      }else{
```

```
        return total;
      }
    }, this.labelProbabilities.get(difficulty) + smoothing);
    classified.set(difficulty, totalLikelihood);
}, this);
```

Now that we have added `this` as a second parameter to `forEach` (on the last line), we are free to use `this` rather than `self` on the third-from-last line. But the remaining call to `self` cannot yet be changed to `this`, because its value is wrapped by the anonymous function inside of `reduce`.

One could reasonably expect that the function signature for `reduce`'s callback would also allow a `thisArg` to be accepted as an optional parameter. Unfortunately, this is not the case, so in this style, the `this` inside of that function must be set with `bind`:

```
  const totalLikelihood = chords.reduce(function(total, chord){
...
  }.bind(this), this.labelProbabilities.get(difficulty) + smoothing);
```

Now our `classify` function can be free from the awkward `self` variable:

```
classify: function(chords){
  const smoothing = 1.01;
  const classified = new Map();
  this.labelProbabilities.forEach(
function(_probabilities, difficulty){
    const totalLikelihood = chords.reduce(function(total, chord){
      const probabilityOfChordInLabel =
this.probabilityOfChordsInLabels.get(difficulty)[chord];
      if(probabilityOfChordInLabel){
        return total * (probabilityOfChordInLabel + smoothing);
      }else{
        return total;
      }
    }.bind(this),
    this.labelProbabilities.get(difficulty) + smoothing);
    classified.set(difficulty, totalLikelihood);
  }, this);
  return classified;
}
```

But before we leave this example, there is another tool we can use to deal with passing our `this` through: arrow functions. Through them, our function can be simplified like this:

```
classify: function(chords){
  const smoothing = 1.01;
  const classified = new Map();
  this.labelProbabilities.forEach((_probabilities, difficulty) => {
    const totalLikelihood = chords.reduce((total, chord) => {
      const probabilityOfChordInLabel =
this.probabilityOfChordsInLabels.get(difficulty)[chord];
      if(probabilityOfChordInLabel){
        return total * (probabilityOfChordInLabel + smoothing);
      }else{
        return total;
      }
    }, this.labelProbabilities.get(difficulty) + smoothing);
    classified.set(difficulty, totalLikelihood);
  });
  return classified;
}
```

We can now get rid of our `this` as a second parameter to `forEach` and our `bind(this)` in `reduce`. The arrow functions are in the following lines:

```
this.labelProbabilities.forEach((_probabilities, difficulty) => {
...
const totalLikelihood = chords.reduce((total, chord) => {
```

The parts with => are what make them *arrow* functions. The syntax might seem a little weird (and it has a lot of variation), but what's great is the `this` passes through from the outer function.

To simplify this function a bit further, `smoothing` can be taken out of the function and added as a new attribute of the `classifier`:

```
smoothing: 1.01,
classify: function(chords){
  const classified = new Map();
  this.labelProbabilities.forEach((_probabilities, difficulty) => {
    const totalLikelihood = chords.reduce((total, chord) => {
      const probabilityOfChordInLabel =
this.probabilityOfChordsInLabels.get(difficulty)[chord];
      if(probabilityOfChordInLabel){
        return total * (probabilityOfChordInLabel + this.smoothing);
      }else{
        return total;
      }
    }, this.labelProbabilities.get(difficulty) + this.smoothing);
    classified.set(difficulty, totalLikelihood);
  });
  return classified;
}
```

On the one hand, this is nice because it gets `smoothing` out of our function. However, it does increase the scope of where it is available and necessitates adding `this.` to precede it inside of the function. Extracting this `const` to an attribute also makes it assignable again. Unfortunately, getting something like a `const` inside of our `classifier` object literal takes a bit of work. There is a `defineProperty` function that we can use to define attributes that are not only not writable, but even immutable by default. We won't go into depth here, but `defineProperty` is a good tool to look into if you want a lot of control over how properties are used.

Normally, `defineProperty` would be called *after* the object is created, which would add complexity to our execution at some point, and move the attribute's creation to some other physical place in the file. On the other hand, we could just call this inside of the `classify` function like this:

```
classify: function(chords){
  Object.defineProperty(this, 'smoothing', {value: 1.01});
```

But that puts complexity back into our function, which is what we were trying to avoid by making it an object to begin with. Later, we'll discuss alternatives to creating objects with object literals, which could give us more control over our properties as the objects are created. But for now, let's revert this change and allow `smoothing` to be an attribute as it was.

Another change we can make to the `classify` function is to inline the classified variable by using the `map` function instead of `forEach`. Whenever you find yourself setting up a container (usually an array, but in our case a `Map` object), using a loop of some kind to change that variable, and then returning the variable, you may be better served by using a `map` function rather than a loop:

```
classify: function(chords){
  const classified = new Map();
  Array.from(this.labelProbabilities.entries()).map(
(labelWithProbability) => {
    const totalLikelihood = chords.reduce((total, chord) => {
      const probabilityOfChordInLabel =
this.probabilityOfChordsInLabels.get(labelWithProbability[0])[chord];
      if(probabilityOfChordInLabel){
        return total * (probabilityOfChordInLabel + this.smoothing);
      }else{
        return total;
      }
    }, this.labelProbabilities.get(labelWithProbability[0]) +
this.smoothing);
    classified.set(labelWithProbability[0], totalLikelihood);
  });
```

```
  return classified;
}
```

Unfortunately, the `Map` *object* doesn't have a `map` *function* (yes, it bums me out, too.) So our first step involves pulling an array out of the entries in the `Map` so that we can use `map` through it. The second change is that we end up with a slightly less convenient object (`labelWithProbability` instead of `difficulty`), which requires us to take the first index of `[0]`. In other words, `difficulty` is replaced with `labelWithProbability[0]`. But we can initialize `difficulty` as a new `const`:

```
classify: function(chords){
  const classified = new Map();
  Array.from(this.labelProbabilities.entries()).map(
(labelWithProbability) => {
    const difficulty = labelWithProbability[0];
    const totalLikelihood = chords.reduce((total, chord) => {
      const probabilityOfChordInLabel =
this.probabilityOfChordsInLabels.get(difficulty)[chord];
      if(probabilityOfChordInLabel){
        return total * (probabilityOfChordInLabel + this.smoothing);
      }else{
        return total;
      }
    }, this.labelProbabilities.get(difficulty) + this.smoothing);
    classified.set(difficulty, totalLikelihood);
  });
  return classified;
}
```

Now we can almost get rid of our `classified` variable. Instead of initializing it, updating it in the loop, and returning it after, we can have `map` return a multidimensional array to set the `Map`:

```
classify: function(chords){
  const classified = new Map(Array.from(
    this.labelProbabilities.entries()).map((labelWithProbability) => {
    const difficulty = labelWithProbability[0];
    const totalLikelihood = chords.reduce((total, chord) => {
      const probabilityOfChordInLabel =
this.probabilityOfChordsInLabels.get(difficulty)[chord];
      if(probabilityOfChordInLabel){
        return total * (probabilityOfChordInLabel + this.smoothing);
      }else{
        return total;
      }
    }, this.labelProbabilities.get(difficulty) + this.smoothing);
```

```
      return [difficulty, totalLikelihood];
    }));
    return classified;
  }
```

The second line now includes directly assigning a `new Map` object to `classified`. Additionally, our `return`, four lines from the bottom, now returns an array with `difficulty` and `totalLikelihood`. The last (very minor) change to notice is that three lines from the bottom of this sample, we now have an extra closing parenthesis in the line:

```
})); // just above "return classified;"
```

And now we're ready to get rid of the `classified` variable. Simply delete the line with the last `return` statement and instead return the result of the call to the `new Map` constructor:

```
classify: function(chords){
  return new Map(Array.from(
    this.labelProbabilities.entries()).map((labelWithProbability) => {
...
  }));
}
```

We now have a similar unnecessary variable in `totalLikelihood`:

```
const totalLikelihood = chords.reduce((total, chord) => {
  const probabilityOfChordInLabel =
this.probabilityOfChordsInLabels.get(difficulty)[chord];
  if(probabilityOfChordInLabel){
    return total * (probabilityOfChordInLabel + this.smoothing);
  }else{
    return total;
  }
}, this.labelProbabilities.get(difficulty) + this.smoothing);
return [difficulty, totalLikelihood];
```

Instead of assigning a variable to the result of the `reduce` call and returning it in the array, we can inline that and return the array directly:

```
classify: function(chords){
  return new Map(Array.from(
    this.labelProbabilities.entries()).map((labelWithProbability) => {
    const difficulty = labelWithProbability[0];
    return [difficulty, chords.reduce((total, chord) => {
      const probabilityOfChordInLabel =
```

```
this.probabilityOfChordsInLabels.get(difficulty)[chord];
      if(probabilityOfChordInLabel){
        return total * (probabilityOfChordInLabel + this.smoothing);
      }else{
        return total;
      }
    }, this.labelProbabilities.get(difficulty) + this.smoothing)];
  }));
}
```

This might look confusing, but we're still just returning a two-element array. The first element is `difficulty`, and the second is the result of the `reduce` call.

If we want to simplify further, we can extract a function to eliminate our assignment and our conditional inside of this code:

```
const probabilityOfChordInLabel =
  this.probabilityOfChordsInLabels.get(difficulty)[chord]
if(probabilityOfChordInLabel){
  return total * (probabilityOfChordInLabel + this.smoothing)
}else{
  return total;
}
```

We have to delete that assignment, and replace the other two references (one in the conditional's test and one in the conditional's `if` branch) with the righthand side of the assignment. That should leave us with this:

```
if(this.probabilityOfChordsInLabels.get(difficulty)[chord]){
  return total *
  (this.probabilityOfChordsInLabels.get(
    difficulty)[chord] + this.smoothing);
}else{
  return total;
}
```

Next, we extract a function to completely replace those lines with this:

```
return total * this.valueForChordDifficulty(difficulty, chord);
```

And we can add a new function above `classify`:

```
valueForChordDifficulty(difficulty, chord){
  if(this.probabilityOfChordsInLabels.get(difficulty)[chord]){
    return this.probabilityOfChordsInLabels.get(difficulty)[chord] +
this.smoothing;
  }else{
```

```
      return 1;
    }
  },
```

That function will produce a value to multiply by the running total. If we want a slightly denser syntax for that function, we can use the ternary syntax:

```
valueForChordDifficulty(difficulty, chord){
  const value =
    this.probabilityOfChordsInLabels.get(difficulty)[chord];
  return value ? value + this.smoothing : 1;
},
```

Note that this would also work as the first line if you don't like the function shorthand syntax:

```
valueForChordDifficulty: function(difficulty, chord){
```

As opposed to what we have right now, which is:

```
valueForChordDifficulty(difficulty, chord){
```

DENSITY AND ABSTRACTION

At this point, you might be wondering if this refactoring is really worth it. By inlining variables, we've made the code denser. There is also some overhead to using `map` with a `Map` object. For some people, inlining variables and using functions like `bind` is going to make the code harder to read. For others, the arrow syntax, with all of its variation (which we'll cover later), might be overwhelming. It's important to be aware of how your style might impact other members of your team.

So did we just make the code worse? Not necessarily. The advantage of inlining variables is twofold. First, less internal state means less to keep track of. Second, *inlining* variables can make it easier to *extract* functions, which are more flexible and testable than simple variables.

For both variables and functions, the important part is to be aware of inlining and extracting as choices.

CODE HAS NO FINAL, PERFECT, "REFACTORED" STATE

One important thing to note about our work on the `classify` function is that there is no way to tell when you are "done" refactoring. Some refactorings (like inlining and extracting) are inverse processes, so you could end up undoing previous refactoring work with subsequent efforts. You could create an infinite loop of refactoring just due to that fact. Mix in others' opinions of what "good" code is, and you really can "improve" the code forever. This is part of why it is important to develop your own standards of quality, and to calibrate those with the people you work with.

Untangling Coupled Values

Moving on from our `classify` function, we have a problem in the code that needs to be addressed. You'll see something interesting if you add logging statements to the tests (after `trainAll();`) like the following:

```
console.log(classifier.probabilityOfChordsInLabels);
console.log(classifier.chordCountsInLabels);
```

They have the same values. Not only that, they actually are referencing the same `Set` object.

The reason for this is in the `setProbabilityOfChordsInLabels` function—specifically, the second line (which spills onto two lines):

```
function setProbabilityOfChordsInLabels(){
  classifier.probabilityOfChordsInLabels =
classifier.chordCountsInLabels;
  classifier.probabilityOfChordsInLabels.forEach(
function(_chords, difficulty){
    Object.keys(
classifier.probabilityOfChordsInLabels.get(difficulty)).forEach(
function(chord){
      classifier.probabilityOfChordsInLabels.get(difficulty)[chord]
/= classifier.songs.length;
    });
  });
}
```

The problem is that when we assign one set to another, we aren't just copying the values from the `Set` object on the righthand side of the assignment into the `Set` object on the lefthand side. Both `classifier.probabilityOfChordsInLabels` and `classifier.chordCountsInLabels` are just fingers

pointing at the same object. If you change the values of the set by referencing either name, both will be affected.

Here is a short example to demonstrate this:

```
x = {a: 2};
// returns { a: 2 }
y = x;
// returns { a: 2 }
x['b'] = 3;
// returns 3
y;
// returns { a: 2, b: 3 }
y['c'] = 5;
// returns 5
x;
// returns { a: 2, b: 3, c: 5 }
```

However, these all involve what happens when you *update* an object. If you reassign the object again, it will not change both labels to be pointing at the new object:

```
x = {a: 2};
// returns { a: 2 }
y = x;
// returns { a: 2 }
x = {b: 5};
console.log(y);
// prints { a: 2 }
// because y still ponts at the original object
// x has been assigned to a new object
```

Back to our code, if `probabilityOfChordsInLabels` and `chordCountsInLabels` do the same thing, we can just replace references to the former with the latter:

```
function setProbabilityOfChordsInLabels(){
  classifier.chordCountsInLabels = classifier.chordCountsInLabels;
  classifier.chordCountsInLabels.forEach(function(_chords, difficulty){
    Object.keys(classifier.chordCountsInLabels.get(difficulty))
      .forEach(function(chord){
        classifier.chordCountsInLabels.get(difficulty)[chord]
/= classifier.songs.length;
    });
  });
}
```

Now line two is looking more obviously redundant, which is great, because we know we can delete it:

```
function setProbabilityOfChordsInLabels(){
  classifier.chordCountsInLabels.forEach(function(_chords, difficulty){
    Object.keys(classifier.chordCountsInLabels.get(difficulty))
      .forEach(function(chord){
        classifier.chordCountsInLabels.get(difficulty)[chord]
/= classifier.songs.length;
    });
  });
}
```

We also have two more references to `probabilityOfChordsInLabels` to change inside of `classifier`. One is this line, which we can delete:

```
probabilityOfChordsInLabels: new Map(),
```

The other is the last line in the following snippet:

```
const classifier = {
...
const value =
this.probabilityOfChordsInLabels.get(difficulty)[chord];
```

The assignment to `value` (inside of the `valueForChordDifficulty` function of `classifier`) should be updated to use `chordCountsInLabels` instead.

This leaves us with the following:

```
const classifier = {
...
const value = this.chordCountsInLabels.get(difficulty)[chord];
```

And tests should be passing at this point. In some cases, the correct refactoring here would be to truly *copy* the `Set` object into a new one. Then, both would be independent objects. We don't actually need two objects, though, so we'll try a different approach here.

"COPYING" OBJECTS

If you're interested in copying objects in JavaScript, look into the terms *deep copy* versus *shallow copy*, as well as *cloning* and the `Object`functions `freeze`, `assign`, and

`seal`. They might do what you want. Or maybe you want to create a new object tied to the old one's prototype with `Object.create`. Maybe you want a totally new object with `Object.assign`. Maybe you want to make a constructor function or a class? Maybe you just want a factory function (see **Chapter 8**). Maybe you want a new *and* immutable container for your data.

By the way, there are two things to note about `Object.create`. First, it doesn't own the properties from the prototype object (its first parameter), so they could possibly be overwritten. Second, it only "copies" enumerable properties, so don't expect it to copy *everything*.

There are about a million and five ways to do inheritance and copying objects in JavaScript. That makes it hard to pick one. The point here is to remember that assigning with = is *not* one of them.

Since the last code sample contains our only reference to the value of the new object, all we have now is an object that changes. But maybe we don't need to update the object at all. Let's try using a function as an attribute, instead of a set that stores all of the values.

First, let's move the `setProbabilityOfChordsInLabels` function into the `classifier` object:

```
const classifier = {
...
  setProbabilityOfChordsInLabels: function(){
    classifier.chordCountsInLabels
.forEach(function(_chords, difficulty){
      Object.keys(classifier.chordCountsInLabels.get(difficulty))
        .forEach(function(chord){
          classifier.chordCountsInLabels.get(difficulty)[chord]
  /= classifier.songs.length;
      });
    });
  },
  valueForChordDifficulty(difficulty, chord){
...
```

And we'll also need to update our `setLabelsAndProbabilities` function to use `classifier.` ahead of the function call:

```
function setLabelsAndProbabilities(){
  setLabelProbabilities();
  setChordCountsInLabels();
  classifier.setProbabilityOfChordsInLabels();
};
```

Running the test suite shows everything is working properly. Now let's replace the references to `classifier` with `this`:

```
  setProbabilityOfChordsInLabels: function(){
    this.chordCountsInLabels.forEach(function(_chords, difficulty){
      Object.keys(this.chordCountsInLabels.get(difficulty))
        .forEach(function(chord){
          this.chordCountsInLabels.get(difficulty)[chord]
/= this.songs.length;
      }, this);
    }, this);
  },
```

Recall that in addition to replacing `classifier` with `this`, we also need to pass `this` in as the optional second argument to both `forEach` calls. Otherwise, our `this` context would get lost. From earlier, we know that we could accomplish the same thing with arrow functions, but we'll actually be removing this function soon, so we'll avoid that refactoring for now.

Now we get to the crux of the problem: we don't really want functions that set values through side effects. We want functions that return values. They are much easier to work with and keep track of. We won't get all the way there in this chapter, and arguably, OOP fights that ethos, whereas FP encourages it (see **Chapter 11**).

As for `setProbabilityOfChordsInLabels`, all that looping amounts to very little. Really, all we want to do is divide the number of times the chord appears in a given difficulty by the number of songs.

You can delete the `setProbabilityOfChordsInLabels` function from the object as well as its call (from inside of `setLabelsAndProbabilities`). In its place, we'll be using a simple function to check a given chord and difficulty combo. Add the following `likelihoodFromChord` function and update the `valueForChordDifficulty` function in `classifier` as follows:

```
const classifier = {
...
  likelihoodFromChord: function(difficulty, chord){
    return this.chordCountsInLabels
      .get(difficulty)[chord] / this.songs.length;
  },
  valueForChordDifficulty(difficulty, chord){
    const value = this.likelihoodFromChord(difficulty, chord);
    return value ? value + this.smoothing : 1;
  },
...
```

Now `setLabelsAndProbabilities` should look like this:

```
function setLabelsAndProbabilities(){
  setLabelProbabilities();
  setChordCountsInLabels();
};
```

With these changes, we've simplified the code and stopped relying on as many reassignments.

At this point, all the tests should be working again. Just to recap what happened in this section, we had two variable names that were assigned to the same object. First, we changed all the instances of the second variable name to be the first variable name, leaving us with a variable that was updated. Then, instead of relying on the updates to that variable, we changed references to it to use a calculation to get the information (instead of just accessing where the information was stored).

ABOUT REASSIGNING VARIABLES

Among the most important recent developments in JavaScript and beyond is the increased importance of functional programming. We'll explore the benefits more in **Chapter 11**, but one of the best things about it is how it allows for values to be easily trusted.

On the other end of the spectrum is reassigning values. Nothing makes a program harder to debug, write features for, refactor, or understand than broadly scoped variables that are assigned and reassigned. However, even variables that are fairly limited in scope can create maintenance difficulties when they are reassigned multiple times (e.g., a variable that changes a few times in just a 20-line function).

If there's one thing that everyone could *stop* doing to benefit their code, it would be reassigning variables.

Updating values (adding/deleting/changing elements in an object or an array, for instance) can be just as bad, and is best done as a one-step operation (mostly simply through `map`, `filter`, or `reduce`) behind a function and assigning to a new variable when possible. Overall, making the scope of *most* variables small should be a priority.

Something to consider is the idea that OOP encourages reassigning variables by having objects that stick around and allow properties to be changed.

Objects with Duplicate Information

In the last section, we dealt with two references to an object that was updated. Now, we'll deal with two independent objects that are a bit too similar. `classifier.songs` and `songList.songs` seem to have almost identical data. Really, the only difference is the extra property (`name`) in `songList.songs`. Let's get rid of `classifier`'s `songs` property:

```
const classifier = {
...
likelihoodFromChord: function(difficulty, chord){
  return this.chordCountsInLabels
    .get(difficulty)[chord] / songList.songs.length;
},
```

We need to make two changes to `classifier`. Delete the `songs` attribute and change `this` to `songList` in the `likelihoodFromChord` function.

Next, we have to delete the second line of our `train` function:

```
function train(chords, label){
  classifier.songs.push({label: label, chords: chords}); // this one
```

Also, `setLabelProbabilities` should use `songList.songs.length` instead of `classifier.songs.length`:

```
function setLabelProbabilities(){
  classifier.labelCounts.forEach(function(_count, label){
    classifier.labelProbabilities.set(label,
      classifier.labelCounts.get(label) / songList.songs.length);
  })
};
```

Last up, our `setChordCountsInLabels` function needs to be changed to use `songList.songs` instead of `classifier.songs`, and also to use `song.difficulty` rather than `song.label`:

```
function setChordCountsInLabels(){
  songList.songs.forEach(function(song){
    if(classifier.chordCountsInLabels.get(song.difficulty)
=== undefined){
      classifier.chordCountsInLabels.set(song.difficulty, {});
    }
    song.chords.forEach(function(chord){
      if(classifier.chordCountsInLabels.get(song.difficulty)[chord] >
0){
```

```
        classifier.chordCountsInLabels.get(song.difficulty)[chord] +=
1;
      } else {
        classifier.chordCountsInLabels.get(song.difficulty)[chord] =
1;
      }
    });
  });
}
```

At this point, the tests all should work again.

Bringing the Other Functions and Variables into classifier

Now, let's move `chordCountsInLabels` into our `classifier` object:

```
const classifier = {
...
  setChordCountsInLabels = function(){
    songList.songs.forEach(function(song){
      if(classifier.chordCountsInLabels.get(song.difficulty)
  === undefined){
        classifier.chordCountsInLabels.set(song.difficulty, {});
      }
      song.chords.forEach(function(chord){
        if(classifier.chordCountsInLabels
.get(song.difficulty)[chord] > 0){
          classifier.chordCountsInLabels
.get(song.difficulty)[chord] += 1;
        } else {
          classifier.chordCountsInLabels
.get(song.difficulty)[chord] = 1;
        }
      });
    });
  },
...
```

And then we need to change the call in `setLabelsAndProbabilities` to reference it through the `classifier`:

```
function setLabelsAndProbabilities(){
  setLabelProbabilities();
  classifier.setChordCountsInLabels();
};
```

Next, we "this-ify" the function by changing references to `classifier` to `this` and adding the `thisArg` to the `forEach` functions:

```
const classifier = {
...
  setChordCountsInLabels: function(){
    songList.songs.forEach(function(song){
      if(this.chordCountsInLabels.get(song.difficulty) === undefined){
        this.chordCountsInLabels.set(song.difficulty, {});
      }
      song.chords.forEach(function(chord){
        if(this.chordCountsInLabels.get(song.difficulty)[chord] > 0){
          this.chordCountsInLabels.get(song.difficulty)[chord] += 1;
        } else {
          this.chordCountsInLabels.get(song.difficulty)[chord] = 1;
        }
      }, this);
    }, this);
  },
```

All the tests still pass. Next, as with turning our `likelihoodFromChord` into a query rather than setting and later retrieving the values from the `probabilityOfChordsInLabels` map, we can now do the same thing to eliminate the `chordCountsInLabels` map.

Take a look again at the `setChordCountsInLabels` function in the last snippet. All it does is loop through each chord in each song, and add 1 for every instance of the chord.

If all we have to do is count the number of times that a chord appears in a given difficulty, we can set up similar loops and a counter, and just add 1 when there is a match. Add this after `setChordCountsInLabels` in the `classify` object:

```
chordCountForDifficulty: function(difficulty, testChord){
  let counter = 0;
  songList.songs.forEach(function(song){
    if(song.difficulty === difficulty){
      song.chords.forEach(function(chord){
        if(chord === testChord){
          counter = counter + 1;
        }
      });
    }
  });
  return counter;
},
```

Now we can use that function instead of setting and retrieving the chord counts. Note that we don't need `thisArg`s for our `forEach` functions because we don't have any references to `this` anymore.

We need a few additional changes. First, the `likelihoodFromChord` function can be changed to:

```
likelihoodFromChord: function(difficulty, chord){
  return this.chordCountForDifficulty(difficulty, chord) /
songList.songs.length;
},
```

Next, we can delete our `setChordCountsInLabels` function from `classifier`, as well as the call to it in `setLabelsAndProbabilities`,which leaves us with this:

```
function setLabelsAndProbabilities(){
  setLabelProbabilities();
};
```

This function now only exists to call another function. That means we can delete `setLabelsAndProbabilities` and just call `setLabelProbabilities`, which happens inside of the `trainAll` function. Change this:

```
function trainAll(){
  songList.songs.forEach(function(song){
    train(song.chords, song.difficulty);
  });
  setLabelsAndProbabilities();
};
```

to this:

```
function trainAll(){
  songList.songs.forEach(function(song){
    train(song.chords, song.difficulty);
  });
  setLabelProbabilities();
};
```

All the tests still pass. Before moving on, we should have another look at the `chordCountForDifficulty` function:

```
chordCountForDifficulty: function(difficulty, testChord){
  let counter = 0;
  songList.songs.forEach(function(song){
    if(song.difficulty === difficulty){
```

```
      song.chords.forEach(function(chord){
        if(chord === testChord){
          counter = counter + 1;
        }
      });
    }
  });
  return counter;
},
```

As we did in the `classify` function, when we have code that uses a loop to apply some function to a collection while altering a variable throughout, it is a good candidate for `reduce`. By the way, notice that we used `let` here, rather than `const`, because we have a variable that genuinely updates. Let's change it to use `reduce`:

```
chordCountForDifficulty: function(difficulty, testChord){
  return songList.songs.reduce(function(counter, song){
    if(song.difficulty === difficulty){
      song.chords.forEach(function(chord){
        if(chord === testChord){
          counter = counter + 1;
        }
      });
    }
    return counter;
  }, 0);
},
```

We have a few changes here:

- We return the result of `reduce` directly.
- We replace our former `counter` variable with a parameter to `reduce`'s callback function.
- We return the `counter` inside of the `reduce` function. Recognize that this is not returning from the `chordCountForDifficulty` function, but rather is used inside of `reduce` to set the `counter` value as it walks through the `songList`.

We can also use a `filter`, rather than a `forEach`, to help us count the elements meeting a condition. The `filter` function returns a new array consisting of elements that matched the conditional:

```
chordCountForDifficulty: function(difficulty, testChord){
  return songList.songs.reduce(function(counter, song){
    if(song.difficulty === difficulty){
```

```
        counter += song.chords.filter(function(chord){
          return chord === testChord;
        }).length;
      }
      return counter;
    }, 0);
  },
```

We get the `length` of this array and add it to the `counter`. This means fewer updates to the `counter`. We also shaved two lines off of the `chordCountFor-Difficulty` function. We could have used a second `reduce` here instead of `filter` if we wanted to focus more on the count than the conditional, but that would mean an `innerCount` variable, which seems a bit clunkier.

PERFORMANCE IMPACTS

The changes that we've made to query on an as-needed basis, rather than create structures more specific to later needs, are for refactoring purposes including reducing the size of the code.

Depending on the data access patterns of your program, this strategy might make your program execute slower or faster. Transforming structures into others that you'll *never* use would be a waste. Transforming them into simpler (shallower) ones that will be accessed frequently could produce some performance benefit. All of this comes with the caveat that a given JavaScript platform might optimize your code in ways you don't expect.

We'll discuss *memoization* along with functional programming in **Chapter 11**. It, as well as other caching techniques, will help to overcome a performance hit you could take from the kind of refactoring we did here.

In any case, the approach of this book is to write for humans first, and worry about performance after the fact.

Moving on, we only have three functions left in the global scope: `train`, `trainAll`, and `setLabelProbabilities`. Those seem like they would fit right in as part of the `classifier`. Let's move them now:

```
const classifier = {
...
trainAll: function(){
  songList.songs.forEach(function(song){
    classifier.train(song.chords, song.difficulty);
  });
  classifier.setLabelProbabilities();
},

train: function(chords, label){
  chords.forEach(chord => {
    classifier.allChords.add(chord);
  });
  if(Array.from(classifier.labelCounts.keys()).includes(label)){
    classifier.labelCounts.set(
      label, classifier.labelCounts.get(label) + 1);
  } else {
    classifier.labelCounts.set(label, 1);
  }
},

setLabelProbabilities: function(){
  classifier.labelCounts.forEach(function(_count, label){
    classifier.labelProbabilities.set(
      label, classifier.labelCounts.get(label) /
songList.songs.length);
  });
}
...
```

The biggest change is that we need to put these functions in the object attribute syntax. Additionally, `trainAll` needs to prepend its calls to `train` and `setLabelProbabilities` with `classifier.`.

Our call to `trainAll` in the tests also needs a new `classifier.` at the beginning:

```
classifier.trainAll();
```

The tests should be working at this point.

Now, let's "`this`-ify" our functions. That means changing instances of `classifier` to `this` as well as adding the `thisArg` as a second argument to the

callbacks of the `forEach` functions inside of `trainAll` and `setLabelProbabilities`:

```
trainAll: function(){
  songList.songs.forEach(function(song){
    this.train(song.chords, song.difficulty);
  }, this);
  this.setLabelProbabilities();
},

train: function(chords, label){
  chords.forEach(chord => { this.allChords.add(chord) });
  if(Array.from(this.labelCounts.keys()).includes(label)){
    this.labelCounts.set(label, this.labelCounts.get(label) + 1);
  } else {
    this.labelCounts.set(label, 1);
  }
},

setLabelProbabilities: function(){
  this.labelCounts.forEach(function(_count, label){
    this.labelProbabilities.set(label, this.labelCounts.get(label) /
songList.songs.length);
  }, this);
}
```

Note that our `forEach` function inside of `train` doesn't need a `thisArg`, because it uses the arrow syntax.

Now, everything is part of either `classifier` or `songList`. We're down to two global variables. Now we can start to think about what belongs where a little more. One attribute of `classifier` sticks out as more appropriately being a part of `songList`: `allChords`.

We can simply move the attribute:

```
const songList = {
  allChords: new Set(),
```

Now we can delete it from `classifier`, and change `this` to `songList` in the function call inside of `train`:

```
chords.forEach(chord => { songList.allChords.add(chord) });
```

The tests should pass at this point.

Next, since we want our `classifier` to be the only point of global access in the program, we should move `songList` into it:

```
const classifier = {
  songList: {
    allChords: new Set(),
    difficulties: ['easy', 'medium', 'hard'],
    songs: [],
    addSong: function(name, chords, difficulty){
      this.songs.push({name: name,
                       chords: chords,
                       difficulty: this.difficulties[difficulty]})
    }
  },
...
```

And now, there are a number of references to `songList` that need to be prepended with `this.` where they are inside of the `classifier` object. For those that are in the tests, they need to be prepended with `classifier.`. Do a search for "songList" and prepend as needed. Save, test, check, and commit.

Shorthand Syntax: Arrow, Object Function, and Object

Next up, let's address some inconsistencies that we have in our function syntax. We've used arrow functions earlier in the book, but let's go into a bit more detail here. We'll start with an instance where we are using the arrow syntax already. In our `train` function we have the following:

```
train: function(){
  chords.forEach(chord => { this.songList.allChords.add(chord) });
...
```

We've discussed it a bit before, but what's interesting about this is, well, `this`. We reference `this` inside of the anonymous function here and are free to *not* pass a `thisArg` (`this`) as the second parameter to `forEach`. As proof that this is something special with arrow functions and not code that happens to work fine without it, try converting it to the longhand form:

```
train: function(){
  chords.forEach(function(chord){
    this.songList.allChords.add(chord);
  });
...
```

It will break the tests! If you want to use that form, you need to include the `this`, as we currently do in `trainAll`:

```
train: function(){
  chords.forEach(function(chord){
    this.songList.allChords.add(chord);
  }, this);
...
```

But we could use `bind` on the function instead (we do this with `reduce` since it doesn't have an option to pass a `thisArg` as a second parameter):

```
train: function(){
  chords.forEach(function(chord){
    this.songList.allChords.add(chord);
  }.bind(this));
...
```

Instead of doing either of those, let's use the arrow syntax, which passes the `this` context to the inner function and saves us from having to type the word *function* so much. Awesome!

You might be wondering why we would ever *not* use the arrow syntax. There are a few cases. Most importantly, you wouldn't want to use it in cases where the `this` you care about should *not* be the `this` that you have access to going into the function. For instance, you might expect the `this` of a click handler in jQuery to refer to the object clicked. If you use an arrow function, that won't be the case.

Another reason is that in this form (ignoring the difficulty in passing parameters), the functions are easy to extract or inline, as the syntax is very similar to function declaration syntax:

```
// form 1

chords.forEach(function myFunction(){
  this.songList.allChords.add(chord);
}, this);

// form 2

chords.forEach(myFunction, this);

function myFunction(){
  this.songList.allChords.add(chord);
};
```

It's easy to convert between these two forms. If you start with an anonymous function in form 1 (with or without arrow syntax), it takes a bit of extra work to convert it into a standalone, named function. But for all of our functions used with `forEach`, `map`, `reduce`, and `filter`, we can easily replace them with ar-

row functions. That shaves off about 10 lines of code. By the way, with our refactoring so far, what was once about 110 lines is now down to 63. It's small enough to see the whole thing at once (minus the tests):

```
const classifier = {
  labelCounts: new Map(),
  labelProbabilities: new Map(),
  chordCountsInLabels: new Map(),
  smoothing: 1.01,
  songList: {
    allChords: new Set(),
    difficulties: ['easy', 'medium', 'hard'],
    songs: [],
    addSong: function(name, chords, difficulty){
      this.songs.push({name: name,
                       chords: chords,
                       difficulty: this.difficulties[difficulty]});
    }
  },
  chordCountForDifficulty: function(difficulty, testChord){
    return this.songList.songs.reduce((counter, song) => {
      if(song.difficulty === difficulty){
        counter += song.chords.filter((chord) => {
          return chord === testChord;
        }).length;
      }
      return counter;
    }, 0);
  },
  likelihoodFromChord: function(difficulty, chord){
    return this.chordCountForDifficulty(difficulty, chord) /
this.songList.songs.length;
  },
  valueForChordDifficulty(difficulty, chord){
    const value = this.likelihoodFromChord(difficulty, chord);
    return value ? value + this.smoothing : 1;
  },
  trainAll: function(){
    this.songList.songs.forEach((song) => {
      this.train(song.chords, song.difficulty);
    });
    this.setLabelProbabilities();
  },
  train: function(chords, label){
    chords.forEach(chord => { this.songList.allChords.add(chord) } );
    if(Array.from(this.labelCounts.keys()).includes(label)){
      this.labelCounts.set(label, this.labelCounts.get(label) + 1);
    } else {
      this.labelCounts.set(label, 1);
```

```
    }
  },
  setLabelProbabilities: function(){
    this.labelCounts.forEach((_count, label) => {
      this.labelProbabilities.set(label, this.labelCounts.get(label) /
this.songList.songs.length);
    });
  },
  classify: function(chords){
    return new Map(Array.from(
      this.labelProbabilities.entries()).map((labelWithProbability) => {
      const difficulty = labelWithProbability[0];
      return [difficulty, chords.reduce((total, chord) => {
        return total * this.valueForChordDifficulty(difficulty, chord);
      }, this.labelProbabilities.get(difficulty) + this.smoothing)];
    }));
  }
};
```

There are a few things to notice with the arrow syntax. First, if there is only one argument, it does not need to go in parentheses:

```
return new Map(Array.from(
  this.labelProbabilities.entries()).map(labelWithProbability => {
```

If there are two or more arguments, you'll need them, as in this case:

```
setLabelProbabilities: function(){
  this.labelCounts.forEach((_count, label) =>{
```

One somewhat strange thing is that we also need parentheses if we have zero arguments. At this point, we can change all of our tests to use this syntax:

```
describe('the file', () => {
...
  it('classifies', () => {
...
  it('classifies again', () => {
...
  it('label probabilities', () => {
...
```

Moving on to function declaration shorthand syntax for objects, we *could* use the arrow syntax and declare our `trainAll` function like this:

```
trainAll: () => {
  this.songList.songs.forEach(song => {
    this.train(song.chords, song.difficulty);
  });
  this.setLabelProbabilities();
},
```

However, this is not what we want. We actually want the `this` that we get with the function keyword used, because that `this` is the object (`classifier`), which is what our `this` inside of the function refers to. When we declare functions this way, we get the context outside of the object (the global object, assuming nonstrict mode). In any case, there is a better shorthand for function declarations, which we'll get to shortly. Let's put it back to this:

```
trainAll: function(){
  this.songList.songs.forEach(song => {
    this.train(song.chords, song.difficulty);
  });
  this.setLabelProbabilities();
},
```

One other weird thing about the arrow syntax is that sometimes we use braces {}, and other times we don't. Right now we have this:

```
counter += song.chords.filter((chord) => {
          return chord === testChord;
        }).length;
```

Let's change that to this:

```
counter += song.chords.filter(chord => chord === testChord).length;
```

The lack of parentheses, braces, and the `return` might seem a little jarring because it makes the statement denser. There is an additional consideration with the braceless version, though: it implicitly returns the result of its execution. In the case of our last code snippet, that means it returns `true` or `false`.

Also note that because the brace syntax is used as a way to group the function body, if you want to return an object like this, you'll be disappointed:

```
someFunction(someArg => {someThing: 'someValue'}) // nope
```

To return an object, you'll need to add parentheses:

```
someFunction(someArg => ({someThing: 'someValue'})) // ok
```

As a matter of personal style, I would recommend going for less syntax when possible. That means this:

```
chords.forEach(chord => { this.songList.allChords.add(chord) } );
```

can become this:

```
chords.forEach(chord => this.songList.allChords.add(chord) );
```

During the discussion of arrow functions, we brushed up against the idea of a shorthand for declaring functions as part of an object. This is how we're currently declaring most functions:

```
const classifier = {
...
    addSong: function(name, chords, difficulty){
...
  chordCountForDifficulty: function(difficulty, testChord){
...
```

However, we have one exception:

```
valueForChordDifficulty(difficulty, chord){

// instead of

valueForChordDifficulty: function(difficulty, chord){
```

Using this shorthand syntax can completely remove the `: function` part, making the functions look like this:

```
const classifier = {
  songList: {
...
    addSong(name, chords, difficulty){
...
  chordCountForDifficulty(difficulty, testChord){
...
```

After doing that with all of our function declarations, we are actually completely free from the `function` keyword in this file! An incredible source of visual noise and extra typing is now gone. Thanks, ES2015!

Note, however, that this shorthand only works inside of object literals and classes. Outside of those contexts (in a normal function, global scope, or constructor function scope, for example), the interpreter will throw an error on the `{`.

Once again, we can see that our shorthand syntax is less portable than the longhand version.

COMPUTED PROPERTIES

While we're discussing function declaration shorthand, it's worth noting that you could declare properties of an object dynamically, like this:

```
songs = {
  ['first' + 'Song']: {},
  ['second' + 'Song']: {},
  ['third' + 'Song']: {}
}
```

Basically, you can run JavaScript inside those square brackets to generate property names. This is a trivial example (and you'd probably want some data in your objects), but you might find this convenient at some point.

Be aware, however, that when dynamically defining labels of properties (or accessing them dynamically as in `songs['first' + 'Song']`), you lose your ability to easily search for simple strings like `firstSong` in your codebase.

The last bit of shorthand we could implement is in the `addSong` function. Right now we have this:

```
addSong(name, chords, difficulty){
  this.songs.push({name: name,
                   chords: chords,
                   difficulty: this.difficulties[difficulty]});
}
```

But we could shorten the object inside of push to use the *object shorthand*, like this:

```
this.songs.push({name, chords,
                 difficulty: this.difficulties[difficulty]});
```

Getting New Objects with Constructor Functions

Up to now, we've been dealing with object literals. We'll get to classes in a bit, but first let's explore another option: creating objects with constructor functions. To use these we would have to change our code to something like the following:

```
const Classifier = function(){
  const SongList = function() {
    this.allChords = new Set();
    this.difficulties = ['easy', 'medium', 'hard'];
    this.songs = [];
    this.addSong = function(name, chords, difficulty){
      this.songs.push({name,
                       chords,
                       difficulty: this.difficulties[difficulty]});
    };
  };
  this.songList = new SongList();
  this.labelCounts = new Map();
  this.labelProbabilities = new Map();
  this.chordCountsInLabels = new Map();
  this.smoothing = 1.01;
  this.chordCountForDifficulty = function(difficulty, testChord){
    return this.songList.songs.reduce((counter, song) => {
      if(song.difficulty === difficulty){
        counter += song.chords.filter(chord => chord === testChord).length;
      }
      return counter;
    }, 0);
  };
  this.likelihoodFromChord = function(difficulty, chord){
    return this.chordCountForDifficulty(difficulty, chord) /
this.songList.songs.length;
  };
  this.valueForChordDifficulty = function(difficulty, chord){
    const value = this.likelihoodFromChord(difficulty, chord);
    return value ? value + this.smoothing : 1;
  };
  this.trainAll = function(){
    this.songList.songs.forEach((song) => {
      this.train(song.chords, song.difficulty);
```

```
      });
      this.setLabelProbabilities();
    };
    this.train = function(chords, label){
      chords.forEach(chord => this.songList.allChords.add(chord) );
      if(Array.from(this.labelCounts.keys()).includes(label)){
        this.labelCounts.set(label, this.labelCounts.get(label) + 1);
      } else {
        this.labelCounts.set(label, 1);
      }
    };
    this.setLabelProbabilities = function(){
      this.labelCounts.forEach((_count, label) => {
        this.labelProbabilities.set(label, this.labelCounts.get(label) /
  this.songList.songs.length);
      });
    };
    this.classify = function(chords){
      return new Map(Array.from(
        this.labelProbabilities.entries()).map((labelWithProbability) => {
        const difficulty = labelWithProbability[0];
        return [difficulty, chords.reduce((total, chord) => {
          return total * this.valueForChordDifficulty(difficulty, chord);
        }, this.labelProbabilities.get(difficulty) + this.smoothing)];
      }));
    };
  };
  const wish = require('wish');
  describe('the file', () => {
    const classifier = new Classifier();
  ...
```

The tests only have a small change, because we need to initialize our `classifier` with:

```
const classifier = new Classifier();
```

That is how you instantiate an object in JavaScript using `new` with a constructor function. Notice that we are also instantiating a `songList` property in a similar way inside of the `classifier`:

```
this.songList = new SongList();
```

Most of our changes are to existing code, but the `new Classifier()` and `new SongList()` lines are, well, new.

DON'T FORGET TO USE NEW IN CONSTRUCTOR FUNCTIONS

If you forget `new` when calling a constructor function, your function may still run fine, but there's a chance it won't. This is because `this` will be bound to the global object (or `undefined` in strict mode).

Sometimes, this fear (what if people forget the `new` keyword!?) is cited as the main motivation behind preferring `Object.create` over `new` with a constructor function. The stronger case for `Object.create` has more to do with seeking consistency with and not obscuring JavaScript's prototypal nature, rather than using the "pseudoclassical style" that constructor functions (and classes) with `new` promote.

On the more flexible side of things, the parentheses after the constructor call following `new` are optional when no arguments are passed to the function. It's a bit weird, but these are the same:

```
this.songList = new SongList();
this.songList = new SongList;
```

There is an important difference between creating objects like this and using object literals. If you use an object literal, you are stuck doing some cloning/deep copying/`Object.create` hijinks to get a new version. As we explored earlier, assigning something to a new variable using = just creates a new reference to the same object: two fingers pointing to the same moon. If you need more than one moon, you're going to need `new`, `Object.create`, `class`, and/or some other copying/cloning utility. If you know that you're going to want more than one object, object literals might not be the best starting point for creating them.

This snippet also demonstrates another result of this change: we have to add `this.` to all of our properties when we use a constructor function. Also notice that this syntax forces us to add our `function` keywords back in with the colons replaced with equal signs. Lastly, the commas at the end of statements have been replaced with semicolons (or not, in special cases, if you're one of those people).

SPEAKING OF SEMICOLONS...

Some people really hate semicolons. They'll only use them when absolutely necessary, as evidence of their deep knowledge of JavaScript's *automatic semicolon insertion* (ASI).

Personally, I find the edge cases for when semicolons are necessary somewhat hard to remember, and I feel that others (including people I work with) do too. For those reasons, I include them most of the time.

"To semicolon, or not to semicolon?" is not the ultimate question for a human: this is a job for a linter to handle.

Although the syntax might not be what we prefer, the tests still pass, so this is a successful refactoring so far. What have we gained? Well, because we're inside of the constructor function and not constrained by the JSON syntax, we're free to write any statements we want. That means we could have private (meaning unexposed) functions and variables inside of the constructor, as well as declaring global variables by not using `var`, `let`, or `const` (we could also do that in the object literal syntax, but not as directly). Most importantly, as mentioned before, we can get multiple instances of the same object quite simply with `new`, rather than going through somewhat unintuitive steps to "clone" or "copy" or "create" one.

Constructor Functions Versus Factory Functions

Let's take a break from our `classifier` for just a minute. You might find this sample similar to the `diary` code from **Chapter 5**, as it also concerns privacy. However, this discussion is focused more on objects and their creation.

We'll be defining our API in a later section of this chapter, and in order to do that, we should be familiar with what we can and can't access inside of constructor functions. The following code sample is a bit shorter, so it will be easier to understand this before applying the concepts to our NBC code:

```
// constructor function
const Secret = function(){
  this.normalInfo = 'this is normal';
  const secret = 'sekrit';
  const secretFunction = function(){
    return secret;
  };
  this.notSecret = function(){
    return secret;
  };
  totallyNotSecret = "I'm defined in the global scope";
};
const s = new Secret();
console.log(s.normalInfo); // 'this is normal'
console.log(s.secret); // undefined
console.log(s.secretFunction()); // error
console.log(s.notSecret()); // 'sekrit'
console.log(s.totallyNotSecret); // undefined
console.log(totallyNotSecret); // I'm defined in the global scope
```

An alternative to creating objects with the `new` keyword is using `Object.create`. Before we change our classifier, let's just observe the differences between this construct and `new`, as in the last snippet:

```
// factory function
var secretTemplate = (function(){
  var obj = {};
  obj.normalInfo = 'this is normal';
  const secret = 'sekrit';
  const secretFunction = function(){
    return secret;
  };
  obj.notSecret = function(){
    return secret;
  };
  totallyNotSecret = "I'm defined in the global scope";
  return obj;
})();
const s = Object.create(secretTemplate);
console.log(s.normalInfo); // 'this is normal'
console.log(s.secret); // undefined
console.log(s.secretFunction()); // error
console.log(s.notSecret()); // 'sekrit'
console.log(s.totallyNotSecret); // undefined
console.log(totallyNotSecret); // "I'm defined in the global scope"
```

So, when we use `Object.create`, we need to supply an *object* to be patterned from. To retain the flexibility of using a constructor with `new`, we end up returning an object inside of the function. Note that this function is an IIFE. However, we could use a normal function expression as well:

```
var secretTemplate = function(){
...
};
const s = Object.create(secretTemplate());
```

This gives us the same object, but seems a bit less clear. Also, in this construction, the function will have to run every time we create a new object. In the IIFE version, the `secretTemplate` function only has to run once, and to create new objects we just reference the template object that was created.

Reordering our code a bit, we could express our object-returning code more concisely like this:

```
// module pattern
var secretTemplate = (function(){
  const secret = 'sekrit';
  const secretFunction = function(){
    return secret;
  };
  totallyNotSecret = "I'm defined in the global scope";
  return {normalInfo: 'this is normal',
          notSecret(){
            return secret;
          }};
})();
const s = Object.create(secretTemplate);
console.log(s.normalInfo); // 'this is normal'
console.log(s.secret); // undefined
console.log(s.secretFunction()); // error
console.log(s.notSecret()); // 'sekrit'
console.log(s.totallyNotSecret); // undefined
console.log(totallyNotSecret); // "I'm defined in the global scope"
```

This uses the *module pattern*, not to be confused with modules to import and export packages. A popular variation on the module pattern is the *revealing module pattern*, shown here:

```
// revealing module pattern
var secretTemplate = (function(){
  const secret = 'sekrit';
  const secretFunction = function(){
    return secret;
  };
  totallyNotSecret = "I'm defined in the global scope";
  const normalInfo = 'this is normal';
  const notSecret = function(){
    return secret;
  };
  return {normalInfo, notSecret};

})();
const s = Object.create(secretTemplate);
console.log(s.normalInfo); // 'this is normal'
console.log(s.secret); // undefined
console.log(s.secretFunction()); // error
console.log(s.notSecret()); // 'sekrit'
console.log(s.totallyNotSecret); // undefined
console.log(totallyNotSecret); // "I'm defined in the global scope"
```

This is a bit cleaner, as it makes the object that is returned very concise and readable.

ONE MORE THING ABOUT IIFES

We just used an IIFE as the right side of our assignment statement. In that case, the variable is likely a good candidate to export as a module.

In other cases, where an IIFE is used specifically to reduce the scope of the code inside, we can use a block instead. So this:

```
(function(){
// code we don't want outside of this scope
})();
```

becomes this:

```
{
// code we don't want outside of this scope
};
```

But if you try to use a block, meaning {}, on the right side of an assignment (as we did with an IIFE), it will be interpreted as an object and likely give an error. In those cases, you're stuck with the IIFE.

Getting back to our `classifier`, how would we use `Object.create` for a factory function? It turns out to be very simple, and it relies on the object literal syntax much more than how we rearranged things to use `new` with the constructor functions in the last section:

```
const classifierTemplate = {
  songList: {
    allChords: new Set(),
...
const wish = require('wish');
describe('the file', () => {
  var classifier = Object.create(classifierTemplate);
```

To make this code work, we only need two changes from what we did with the object literal. First, we need to regard our initial object as a *template*, so we rename it `classifierTemplate`. Second, in our tests (which refer to `classifier`), we create the `classifier` object by passing the template object to `Object.create`. Note that we are just using an object literal here, not a function

or IIFE, because we don't have anything that we care about being private at the moment.

In case you forgot what advantages a factory function with `Object.create` and a constructor function with `new` have over just using the object literal directly, the main one is that these are ways to create multiple `classifier` variables. The `songList` is a property of `classifier`, and as a nested object, a new version of it will be created for each `classifier`, even though it uses the object literal syntax. `Object.create` and `new` (along with classes and modules) are also your gateways to inheritance, which we'll discuss in more depth in the next chapter.

A class for Our Classifier

Next, we'll convert our code into a class. Note that neither of our objects is a good fit for `Map` because they both contain various attributes, including functions. So what do we need to change to *classify* our code? Not too much:

```
class Classifier {
  constructor(){
    this.songList = {
      allChords: new Set(),
      difficulties: ['easy', 'medium', 'hard'],
      songs: [],
      addSong(name, chords, difficulty){
        this.songs.push({name,
                         chords,
                         difficulty: this.difficulties[difficulty]});
      }
    };
    this.labelCounts = new Map();
    this.labelProbabilities = new Map();
    this.smoothing = 1.01;
  };
  chordCountForDifficulty(difficulty, testChord){
    return this.songList.songs.reduce((counter, song) => {
      if(song.difficulty === difficulty){
        counter += song.chords.filter(
          chord => chord === testChord
        ).length;
      }
      return counter;
    }, 0);
  };
...
```

The first line might look a bit like a function definition, and under the hood it is (although this is becoming less apparent as JS classes evolve), but it's a special kind of function. And as with a function, we could also assign a class expression to a variable, like this:

```
const Classifier = class {
```

As for the other changes in our code, the most significant changes in adapting from the object literal syntax are (as when we used a constructor function) having semicolons instead of commas. Otherwise, our function definitions are the same. Properties that aren't functions, however, can conveniently be defined inside of a `constructor` function and use the `this.` syntax and semicolons at the end. The `constructor` function runs when a new object is created, and it is responsible for assigning properties.

The only other important change is how the constructor is called, which is identical to the new/constructor function pattern:

```
const classifier = new Classifier();
```

Recall from earlier that if you don't have any arguments needed for the constructor, this will work fine without the parentheses:

```
const classifier = new Classifier;
```

STATIC FUNCTIONS

One utility that classes offer is the ability to add *static* functions. These are useful if you have any functions that don't require an instance to be useful. In our case, every function we have references `this`, a sure sign that none of them can be made static. If we wanted to be very aggressive and extract a static function (that only is responsible for division), we could try to turn our `likelihoodFromChord` from this:

```
likelihoodFromChord(difficulty, chord){
  return this.chordCountForDifficulty(difficulty, chord)
  / this.songList.songs.length;
};
```

into these:

```
likelihoodFromChord(difficulty, chord){
  return this.divide(this.chordCountForDifficulty(difficulty,
                                                  chord),
                     this.songList.songs.length);
```

```
  };
  divide(dividend, divisor){
    return dividend / divisor;
  };
```

And since `divide` clearly has nothing to do with the function itself (which is apparent because there are no references to `this`), we can make it static, and change the call to reflect the new container of the function (the class, rather than the instance):

```
  likelihoodFromChord(difficulty, chord){
    return Classifier.divide(this.chordCountForDifficulty(difficulty, chord),
                             this.songList.songs.length);
  };
  static divide(dividend, divisor){
    return dividend / divisor;
  };
```

This change is unnecessary, as `/` gets the job done just fine, but static functions are fairly easy to implement. Feel free to revert these changes.

BUT AREN'T CLASSES BAD?

"JavaScript doesn't have real classes! They are just functions and prototypes and objects underneath! It's a trick. Don't fall for it! Its *true nature* is disguised by them!"

Or "JavaScript was only built in 10 days or something, right? It's a mess underneath, and clinging to new syntax is our only hope to avoid thinking about the differences between `.prototype`, `[[prototype]]`, `getPrototypeOf`, and `__proto__`."

Alternatively, "Classes are fine and useful if they run on your platform and your team can understand them."

Also, as more features are added to JavaScript classes (such as real private attributes, as discussed in **Chapter 5**), they are starting to look less like "syntactic sugar" and more like a unique construct. This doesn't invalidate arguments of preferring object composition or functional programming over OOP, but it does weaken the "classes are nothing special" argument.

Choosing Our API

Now that our class appears to be in pretty good shape, it's time to consider our API itself. In other words, if someone were importing our code as a module, what functions would be public (accessible to them), and which would be private? As we know from **Chapter 5**, the private/public distinction is a bit complicated in JavaScript (as of this writing, "private" currently either means obscured or inaccessible, although this is likely to change), but nonetheless, we can determine the ideal now, even before realizing the distinction fully in a module. We'll tackle that in the next section. For now, we're just choosing what functions we definitely want to be accessible.

There are three functions from `classify` that we need to be public:

- `constructor`
- `trainAll`
- `classify`

Additionally, our `addSong` function from `songList` needs to be accessible. For convenience's sake, let's add a function to the classifier:

```
class Classifier  {
  constructor(){
```

```
...
  };
  addSong(name, chords, difficulty){
    this.songList.addSong(name, chords, difficulty);
  };
...
```

WHAT ABOUT TRAIN?

Currently, our API relies on following the pattern of adding songs and then training them all at once. Unfortunately, because of the side effects we have, our code is not *idempotent* (a concept covered in **Chapter 11**). In other words, our functions are not "pure," because of their side effects, and running them in a different order than specified (or multiple times) is not handled well.

Namely, `train` does not run `setLabelProbabilities`, and if `setLabelProbabilities` were tied to `train` instead of `trainAll`, our tests would break.

This is a problem we will not fix in this chapter, but there is an idempotent (and functional) version of this NBC in **Chapter 11**.

Now we can call the function directly from the classifier, which means the calls in our tests like this:

```
classifier.songList.addSong('imagine',
['c', 'cmaj7', 'f', 'am', 'dm', 'g', 'e7'], 0);
classifier.songList.addSong('somewhereOverTheRainbow',
['c', 'em', 'f', 'g', 'am'], 0);
```

can turn into this:

```
classifier.addSong('imagine',
['c', 'cmaj7', 'f', 'am', 'dm', 'g', 'e7'], 0);
classifier.addSong('somewhereOverTheRainbow',
['c', 'em', 'f', 'g', 'am'], 0);
```

Additionally, since our function in the classifier has become pure delegation, we no longer need to be so specific about the parameters:

```
class Classifier  {
  constructor(){
...
  };
  addSong(...songParams){ // rest
    this.songList.addSong(...songParams); // spread
  };
...
```

The `...songParams` in the function definition is known as *rest parameter syntax*, and using a similar style in a function call is known as using the *spread operator*. It's hard to keep these terms straight, but Chris Deely, one of the technical reviewers of this book, suggested this mnemonic: *rest* is for "receiving," *spread* is for "sending." That works for me. (Thanks, Chris.)

The rest parameter syntax takes any arguments you give it and turns them into an array. The spread operator does the opposite, splitting the array you give it into individual arguments to pass to the function.

The reason this is a good pattern to apply in this situation is that we already have a function definition for `addSong` inside of `songList`. If we decided to change the function in terms of what arguments it accepts, it would be nice not to have to change this function as well. Using rest and spread here affords us that flexibility.

By the way, if we discovered that in actuality `songList` was doing all the work, and `classifier` was simply delegating every function to it, removing these delegation functions and letting the tests (or client code that uses our module) call the `songList` versions directly would be worth considering. Having just one object that clients interact with is nice for them, but if our delegation is adding sufficient bulk, we should rethink the design.

Time for a Little Privacy?

As we saw in **Chapter 5**, if we want fake privacy in a class, we can just add an underscore (_) in front of all of the properties we want to be private. This convention lets people using the API know that they're in weird territory if they're addressing these properties directly (sometimes the private/internal aspects of an API are referred to as the "plumbing" as opposed to the "porcelain").

Hopefully, your editor has a way to easily rename things (find and replace across not just a file, but the whole project). These are the labels you should prepend an underscore to:

- `songList`
- `labelCounts`
- `labelProbabilities`
- `smoothing`
- `chordCountForDifficulty`
- `likelihoodFromChord`
- `valueForChordDifficulty`
- `train`
- `setLabelProbabilities`

Now we have an extremely simple path to exporting our class as a module. Let's split up our tests and main file to prove it. First, we have some changes to make to *nb.js*:

```
module.exports = class Classifier {
  constructor(){
    this._songList = {
      allChords: new Set(),
      difficulties: ['easy', 'medium', 'hard'],
      songs: [],
      addSong(name, chords, difficulty){
        this.songs.push({name,
                         chords,
                         difficulty: this.difficulties[difficulty]});
      }
    };
    this._labelCounts = new Map();
    this._labelProbabilities = new Map();
    this._smoothing = 1.01;
  };
  addSong(...songParams){
    this._songList.addSong(...songParams);
  };
...
  classify(chords){
    return new Map(Array.from(
      this._labelProbabilities.entries()).map(
(labelWithProbability) => {
      const difficulty = labelWithProbability[0];
      return [difficulty, chords.reduce((total, chord) => {
        return total * this._valueForChordDifficulty(difficulty,
 chord);
      }, this._labelProbabilities.get(difficulty) + this._smoothing)];
    }));
  }
};
```

We've only made two changes here. The first is that the first line now has this at the beginning:

```
module.exports =
```

The second is that we're moving all of our testing to a separate file (in the same directory) called *nb_test.js*:

```
const Classifier = require('./nb.js');
const wish = require('wish');
describe('the file', () => {
```

```
    const classifier = new Classifier;
    classifier.addSong('imagine',
  ['c', 'cmaj7', 'f', 'am', 'dm', 'g', 'e7'], 0);
  ...
    it('label probabilities', () => {
      wish(classifier._labelProbabilities.get('easy') ===
  0.3333333333333333);
      wish(classifier._labelProbabilities.get('medium') ===
  0.3333333333333333);
      wish(classifier._labelProbabilities.get('hard') ===
  0.3333333333333333);
    });
  });
```

We've only made one change to our tests, and it's right up top. If you run **`mocha nb_test.js`**, you should have no failures. Awesome.

But do you want actual privacy? For now, you have a couple of options. The first is to go the "revealing module pattern" route, change the class back into a constructor function, and make some seriously hefty changes. Or, you could try some more convoluted and esoteric things with ES2015's `Symbol`s (creating privacy through obscurity) or `WeakMap`s, or using an initialization function (which is basically designing your own form of the revealing module pattern).

If we honestly think about the trade-offs here, though, what are you gaining by doing something complex and nonstandard? You're making more work for yourself and others who might work on your module. The code itself and the tests become more complex. Trading your ability to call tests for *truly* private, unaddressable functions seems like a nonstarter.

What are the risks of going the underscore route? A few extra characters here and there will make your code ugly? People will think you're too dumb to use the latest and most confusing workarounds? People using your module will insist on calling those functions, even if for the uninitiated you indicated in your documentation that they shouldn't?

If your program is already living in constructor function city, this is a tougher call, but if you're using classes, adding underscores seems like the much easier solution.

All that said, it seems like private fields and methods could be headed our way soon. The spec is still in the works as of this writing, but the basic idea is that references to private functions and attributes would use a **#** instead of an **_**, and be accessible only from inside the class definition. See **Chapter 5** for an example of what it *may* look like.

Adapting the Classifier to a New Problem Domain

And now for the last topic for this program: what if, instead of songs with chords, we're working with learning vocabulary, and instead of "easy," "medium," and "hard," we're classifying a corpus of text as either "understood" or "not understood"?

Some people (both coders and noncoders), upon thinking about a potential abstraction like this, will jump to the general case right away. Think about abstracting a general NBC with the code from the beginning versus the code now. Personally, I recommend building two or three kinds of something (even something as small as a function) before insisting they are similar and attempting to abstract them.

Now that our code is in better shape and our program is fairly small, it's easy enough to attempt adapting it to a new domain.

Mostly, all we have to do is rename a lot of objects:

```
module.exports = class Classifier {
  constructor(){
    this._textList = {
      allWords: new Set(),
      understood: ['yes', 'no'],
      texts: [],
      addText(name, words, comprehension){
        this.texts.push({name, words,
comprehension: this.understood[comprehension]});
      }
    };
    this._labelCounts = new Map();
    this._labelProbabilities = new Map();
    this._smoothing = 1.01;
  };
  addText(...textParams){
    this._textList.addText(...textParams);
  };
  _wordCountForComprehension(comprehension, testWord){
    return this._textList.texts.reduce((counter, text) => {
      if(text.comprehension === comprehension){
        counter += text.words.filter(
          word => word === testWord
        ).length;
      }
      return counter;
    }, 0);
  };

  _likelihoodFromWord(comprehension, word){
    return this._wordCountForComprehension(comprehension, word) /
```

```
this._textList.texts.length;
  };
  _valueForWordComprehension(comprehension, word){
    const value = this._likelihoodFromWord(comprehension, word);
    return value ? value + this._smoothing : 1;
  };
  trainAll(){
    this._textList.texts.forEach((text) => {
      this._train(text.words, text.comprehension);
    });
    this._setLabelProbabilities();
  };

  _train(words, label){
    words.forEach(word => this._textList.allWords.add(word) );
    if(Array.from(this._labelCounts.keys()).includes(label)){
      this._labelCounts.set(label, this._labelCounts.get(label) + 1);
    } else {
      this._labelCounts.set(label, 1);
    }
  };

  _setLabelProbabilities(){
    this._labelCounts.forEach((_count, label) => {
      this._labelProbabilities.set(label,
this._labelCounts.get(label) / this._textList.texts.length);
    });
  };

  classify(words){
    return new Map(Array.from(
      this._labelProbabilities.entries()).map(
(labelWithProbability) => {
      const comprehension = labelWithProbability[0];
      return [comprehension, words.reduce((total, word) => {
        return total * this._valueForWordComprehension(comprehension,
word);
      }, this._labelProbabilities.get(comprehension) +
this._smoothing)];
    }));
  }
};
```

And here are the tests:

```
Classifier = require('./nb_new_domain.js');
const wish = require('wish');
describe('the file', () => {
  const classifier = new Classifier;
```

```
    classifier.addText('english text',
                       ['a', 'b', 'c', 'd', 'e', 'f', 'g', 'h', 'i',
                        'j', 'k', 'l', 'm', 'n', 'o', 'p', 'q'],
                        0);
    classifier.addText('japanese text',
                       ['あ',    'い',   'う',    'え',    'お',
                         'か',    'き',   'く',   'け',   'こ'],
                         1);

    classifier.trainAll();
    it('classifies', () =>{
      const classified = classifier.classify(['お', 'は', 'よ', 'う', 'ご', 'ざ',
                                              'い', 'ま', 'す']);
      wish(classified.get('yes') === 1.51);
      wish(classified.get('no') === 5.19885601);
    });
    it('number of words', ()=>{
      wish(classifier._textList.allWords.size === 27);
    });

    it('label probabilities', ()=>{
      wish(classifier._labelProbabilities.get('yes') === 0.5);
      wish(classifier._labelProbabilities.get('no') === 0.5);
    });
  });
```

Based on the earlier input, the tests show that the Japanese is likely to be incomprehensible to someone who classified the original training data. If we wanted to go further with this text classifier, we might be interested in processing the text more thoroughly. This could mean by character/letter, by word, by sentence, or getting deeper into the grammars of the target languages with techniques like *stemming*. Similarly, with the music version, we could have considered sequences of chords (transitions can be more difficult than the chords themselves). It's also possible that a reader of both languages could have a different threshold of understanding for a specific vocabulary (e.g., parts of a motorcycle), rather than classifying everything in one language as incomprehensible.

Again (as we did with "easy," "medium," "hard"), we're stopping one step before our classifier prints "yes/no" or "Japanese/English" (we could train for either scenario), so we just pick the category with the largest associated number. Feel free to add a test and implement that feature if you're feeling ambitious.

While those are all features to add, our purpose in this chapter—improving the quality of the code—has been accomplished, so we're all set. Check out **Chapter 11** for a functional version of this code.

Wrapping Up

In this chapter (and **Chapter 6**), we covered a huge range of general refactoring techniques. Throughout the rest of the book, we'll be looking at more specialized styles based on the paradigms JavaScript provides, including object-oriented programming, functional programming, and asynchronous programming.

Refactoring Within a Hierarchy 8

In the previous two chapters, we saw a large example of refactoring in action. Ultimately, our concern was with the execution of two main actions: training and classifying. It was convenient to write our program as having one main object (a Naive Bayes Classifier), which was also explored as a class and a module.

For this chapter, one object is not enough.

About "CRUD Apps" and Frameworks

As a web application developer, you'd likely spend a lot of time with "CRUD" (create, read, update, delete) applications. That means concentrating on two high-level tasks: organizing data, and presenting that data.

The former is often addressed by database management systems (and your design of the data), while the latter is often handed off to a framework, concerned with the efficient and workable presentation of that data. Database records mix together with some proprietary data, and turn into CSV (comma-separated value) files and styled web pages.

Although we're not diving into full-stack or presentation frameworks here, it is worth pointing out that they help to avoid a good deal of the problems that refactoring helps to solve. These include:

- One huge file containing all of the code
- One huge (flat) directory containing all of the code
- One huge object or function containing all of the code

As we demonstrated in the previous chapter, extracting functions, objects, and modules can provide some clarity into how the code works. Frameworks, databases, and other libraries provide common interfaces for data access and presentation, and in doing so, seem to solve the biggest organizational problems.

However, three major drawbacks to frameworks are:

- Even if the interface in your app becomes more standardized or "simplified" within a framework, team members will need to come up to speed.
- "Refactoring with framework x" is a rewrite. There's no such thing as "refactoring" by radically changing interfaces.
- Developers can become (or begin as) completely dependent on frameworks, which constrains experience in the language itself.

This last point is as troubling as it is common. Many developers can use jQuery, React, and the like to some degree, but struggle with JavaScript outside of those contexts.

In Chapters **6** and **7**, we discussed the building blocks of architecting JavaScript in a reasonable way. In this chapter and beyond, we take those ideas a bit further, and in this chapter, we will discuss how hierarchies can go wrong.

There are definitely cases where you will want a framework to help you organize and standardize your code, but extracting functions, objects, and modules, as well as building hierarchies, are all tools that you should consider along with frameworks when you need to manage complexity.

Let's Build a Hierarchy

In this chapter, we'll be dealing with words—specifically, creating a vocabulary list. This could be a simple array of words you need to study, or, to add a bit of complexity, words could each be objects with multiple properties: aspects of state (nonfunction attributes) and behavior (functions).

As far as the motivation for hierarchy, there are two. The first is that having a "parent" object (and possibly class as well) will lend itself to code reuse and less repetition in code. Let's say we want to count the characters of our words, for which we have two classes:

```
class EnglishWord{
  constructor(word){
    this.word = word;
  }
  count(){
    return this.word.length;
  }
};
class JapaneseWord{
  constructor(word){
    this.word = word;
  }
  count(){
    return this.word.length;
  }
};
```

```
const japaneseWord = new JapaneseWord("犬");
const englishWord = new EnglishWord("dog");
console.log(japaneseWord.word);
console.log(japaneseWord.count());
console.log(englishWord.word);
console.log(englishWord.count());
```

We can see that there is a lot of duplication in the classes, but what can we do about it?

```
class Word{
  constructor(word){
    this.word = word;
  };
  count(){
    return this.word.length;
  };
};

class EnglishWord extends Word{};
class JapaneseWord extends Word{};

const japaneseWord = new JapaneseWord("犬");
const englishWord = new EnglishWord("dog");
console.log(japaneseWord.count());
console.log(japaneseWord.word);
console.log(englishWord.count());
console.log(englishWord.word);
```

Now `EnglishWord` and `JapaneseWord` both inherit from `Word`, and the code is much shorter. We've "extracted a superclass" and "pulled up the functions" from the subclasses. If the subclasses were actually the same, we'd be free to delete those two classes (collapsing the hierarchy) completely and just use `new Word("`*`something`*`")` for new words. However, we'll be adding distinguishing features to both, so let's keep them for now.

As far as distinguishing features of these different words, they won't tend to appear in the same dictionary. Should we want to look them up, we could add a function to our superclass (`Word`) like this:

```
class Word{
...
  lookUp(){
    if (this instanceof JapaneseWord){
      return `http://jisho.org/search/${this.word}`;
    }else{
      return `https://en.wiktionary.org/wiki/${this.word}`;
    }
```

```
  };
};
...
console.log(englishWord.lookUp());
console.log(japaneseWord.lookUp());
```

The `instanceof` operator is something we haven't seen yet (by the way, the unfortunate lack of camelCase is not a typo). It can be handy, but ties our conditional to a specific name of the class. To get away from that, we might want to add a `language` attribute to each class of word:

```
class Word{
  constructor(word){
    this.word = word;
  };
  count(){
    return this.word.length;
  };
  lookUp(){
    if (this.language === "Japanese"){
      return `http://jisho.org/search/${this.word}`;
    }else{
      return `https://en.wiktionary.org/wiki/${this.word}`;
    }
  };
};

class EnglishWord extends Word{
  constructor(word){
    super(word);
    this.language = "English";
  };
};
class JapaneseWord extends Word{
  constructor(word){
    super(word);
    this.language = "Japanese";
  };
};

const japaneseWord = new JapaneseWord("犬");
const englishWord = new EnglishWord("dog");
console.log(japaneseWord.count());
console.log(japaneseWord.word);
console.log(englishWord.count());
console.log(englishWord.word);
console.log(englishWord.lookUp());
console.log(japaneseWord.lookUp());
```

Now it might look like our superclass constructor isn't doing very much work. What if we defined each subclass constructor to take over all of the setup responsibilities?

```
class EnglishWord extends Word{
  constructor(word){
    this.word = word;
    this.language = "English";
  };
};
class JapaneseWord extends Word{
  constructor(word){
    this.word = word;
    this.language = "Japanese";
  };
};
```

If we do this, we'll get an error on that third line. Why? It is a weird quirk, but in order to subclass our constructor function, we still need to call `super`. Otherwise, `this` is `undefined`:

```
class Word{
  count(){
...
  };
  lookUp(){
...
  };
};
class EnglishWord extends Word{
  constructor(word){
    super();
    this.word = word;
    this.language = "English";
  };
};
class JapaneseWord extends Word{
  constructor(word){
    super();
    this.word = word;
    this.language = "Japanese";
  };
};
```

Weirder still, even if the superclass's constructor function is not present at all, we still have to pay it homage. We could split the responsibility like this:

```
class Word{
  constructor(word, language){
    this.word = word;
  };
...
};

class EnglishWord extends Word{
  constructor(word){
    super(word);
    this.language = "English";
  };
};
class JapaneseWord extends Word{
  constructor(word){
    super(word);
    this.language = "Japanese";
  };
};
```

But it's probably better just to give `super` all of the responsibility like this:

```
class Word{
  constructor(word, language){
    this.word = word;
    this.language = language;
  };
...
};
class EnglishWord extends Word{
  constructor(word){
    super(word, "English");
  };
};
class JapaneseWord extends Word{
  constructor(word){
    super(word, "Japanese");
  };
};
```

DEFAULT PARAMETERS

You might find that, most of the time, a parameter to a function (constructor or not) tends to be the same. If that is the case, you can use default parameters like this:

```
class Word{
  constructor(word, language="English"){
    this.word = word;
    this.language = language;
  };
...
};
class EnglishWord extends Word{};
class JapaneseWord extends Word{
  constructor(word){
    super(word, "Japanese");
  };
};
```

Calling `EnglishWord`'s constructor will now just use `Word`'s constructor with one argument (recall that JavaScript doesn't mind when you call too many or too few arguments to a function). By default, the language will be English, but if a second argument is passed in, the language will be set to whatever that is instead.

Note that this does create some complexity in the function definition, as you have two potential paths instead of one. Additionally, the function calls will not necessarily all look the same.

It's better to use default arguments for something that's truly a default, rather than just a common case. For instance, initially marking a word in a vocabulary list as `studied = false` would make sense because it's not possible for it to have been studied before it enters the system.

As for the second reason to subclass, let's look again at our `lookUp` function:

```
class Word{
...
  lookUp(){
    if (this.language === "Japanese"){
      return `http://jisho.org/search/${this.word}`;
    }else{
      return `https://en.wiktionary.org/wiki/${this.word}`;
    }
  };
};
```

`if` statements are not necessarily universally a bad thing, but better code will tend to avoid them through subclassing or some other polymorphic mechanism. If we delete the `lookUp` function from the superclass and add it to the subclasses, we'll end up with implementations like this:

```
class EnglishWord extends Word{
...
  lookUp(){
    return `https://en.wiktionary.org/wiki/${this.word}`;
  };
};
class JapaneseWord extends Word{
...
  lookUp(){
    return `http://jisho.org/search/${this.word}`;
  };
};
```

This works great, and we have two tiny functions (three lines each) instead of one small function (five lines). Overall, we gained a line, but both functions are, independently, simpler to test. Better still, we only have one code path for each of them. This may seem subtle, but it means that should either branch grow in complexity—for example, switching dictionaries for different kinds of words (within a language)—the complexity will grow in a smaller context. `if` statements tend to lead to more `if` statements. Avoiding them helps to keep code bulk down, and subclassing helps to eliminate them.

By the way, notice that if we only had a `Word` class and no subclasses, two things would be true with our given tools: the conditional check from before would be our only option, and we wouldn't be able to use `instanceof` for that check even if we wanted to.

SUBCLASSING? JUST TO AVOID A CONDITIONAL?

You may be skeptical of subclassing being the best way to avoid conditionals. Sometimes you'll prefer a simple conditional to subclassing. Other times, you may want to skip both the subclassing and the conditional altogether (see **"Strategy"**).

There are many ways to avoid conditionals, but simplifying functions to not ask "what am I?"–type questions is key. This is sometimes called "tell, don't ask."

You may find that the lookup URL itself is valuable. As with the language, there are a few options here. You might change the superclass's implementation of `lookUp` and just add to the subclasses' constructors:

```
class Word{
  lookUp(){
    return this.lookUpUrl + this.word;
  };
...
```

```
};

class EnglishWord extends Word{
  constructor(word){
    super(word, "English");
    this.lookUpUrl = 'https://en.wiktionary.org/wiki/';
  };
};
class JapaneseWord extends Word{
  constructor(word){
    super(word, "Japanese");
    this.lookUpUrl = 'http://jisho.org/search/';
  };
};
```

Additionally, you could, as we did with `language`, add another parameter (`lookUpUrl`) to the superclass's constructor:

```
class Word{
  constructor(word, language, lookUpUrl){
    this.word = word;
    this.language = language;
    this.lookUpUrl = lookUpUrl;
  };
...
};
class EnglishWord extends Word{
  constructor(word){
    super(word, 'English', 'https://en.wiktionary.org/wiki/');
  };
};
class JapaneseWord extends Word{
  constructor(word){
    super(word, 'Japanese', 'http://jisho.org/search/');
  };
};
```

My take on this is to either make the superclass do as much as possible, or as little as possible. Having the subclasses simply delegate to the superclass is fairly clean. Having the subclasses each handle as much as possible themselves (and potentially, not even have a constructor superclass) is also fairly clean. Mixing the two together, creating too much variation in the constructors of sibling classes (like `EnglishWord` and `JapaneseWord`), having default parameters, or relying on deeper hierarchies (e.g., `super`'s `super`'s `super`'s constructor) can all make things more confusing.

Before we move on, let's develop some assumptions through tests. If you haven't installed wish and deep-equal, do that now with:

```
npm install wish
npm install deep-equal
```

As for our tests, we'll use the following:

```
const wish = require('wish');
const deepEqual = require('deep-equal')

// interfaces tests
wish(japaneseWord.word ===  "犬");
wish(japaneseWord.lookUp() === "http://jisho.org/search/犬");
wish(japaneseWord.count() === 1);

wish(englishWord.word ===  "dog");
wish(englishWord.lookUp() === "https://en.wiktionary.org/wiki/dog");
wish(englishWord.count() === 3);

// internals tests
wish(typeof japaneseWord === 'object');
wish(typeof JapaneseWord === 'function');
wish(japaneseWord instanceof JapaneseWord);
wish(japaneseWord instanceof Word);
wish(!(JapaneseWord instanceof Word));

wish(japaneseWord.constructor === JapaneseWord);
wish(Object.getPrototypeOf(JapaneseWord) === Word);

// sketchy bits
wish(deepEqual(Object.getPrototypeOf(japaneseWord), {}));
console.log(Object.getPrototypeOf(japaneseWord));
// reports JapaneseWord {}
```

The interface tests may not be terribly surprising. However, with our hierarchy variants that follow, these are the ones we'll really care about preserving.

The internals tests are less straightforward. `typeof` only manages to tell us that `japaneseWord` is an object and `JapaneseWord` is a function. The best information here is probably that `japaneseWord` is an instance of both `JapaneseWord` and `Word`. By contrast, `JapaneseWord` is a class, so it's not an instance of `Word`.

In the next two lines, we can see that the constructor for `japaneseWord` (the object) is `JapaneseWord` (the class) and that the prototype of `JapaneseWord` (the class) is `Word` (the class).

From there, things get a bit sketchy. The prototype of `japaneseWord` evaluates to an empty object, but if you log it, you'll get a bit more info: `JapaneseWord {}`.

If you want to go deeper than this on prototypes, you should know a few things:

- There's a "nonstandard" yet popular alternative to `Object.getPrototypeOf(`*`thing`*`)`, which you use like this: *`thing`*`.__proto__`.
- `Object.getPrototypeOf` has the alias `Reflect.getPrototypeOf`.
- There's another prototype inspection attribute that you can use like this: `thing.prototype`. It's super unreliable.
- Partly because there's so much nonsense and inconsistency around prototypes, when people talk about the "real, true, deep-down" prototype of something, they'll use syntax like `[[Prototype]]` to indicate they mean the *real* prototype, as opposed to the five ways of actually interrogating an object.

There are some subtle and nuanced opinions concerning prototypes, but for the sake of refactoring our example, we'll only care that the interface is preserved. For a more sympathetic and detailed exploration of prototypes in JavaScript, I'd recommend reading ***this & Object Prototypes*** (http://bit.ly/ydkjs_this), from the "You Don't Know JS" series.

Let's Wreck Our Hierarchy

Now that we have a nice hierarchy set up, let's break it down, exploring these options:

- Constructor functions
- Object literals
- Factory functions

Constructor Functions

We've been through the noninheritance aspects of `this` in previous chapters, but here is how we would write the constructor function version of our class-based code from the last section:

```
function Word(word, language, lookUpUrl){
  this.word = word;
  this.language = language;
  this.lookUpUrl = lookUpUrl;
  this.count = function(){
    return this.word.length;
  };
  this.lookUp = function(){
    return this.lookUpUrl + this.word;
```

```
  };
};

function EnglishWord(word){
  Word.call(this, word, "English", 'https://en.wiktionary.org/wiki/');
};

function JapaneseWord(word){
  Word.call(this, word, "Japanese", 'http://jisho.org/search/');
};

JapaneseWord.prototype = Object.create(Word.prototype);
JapaneseWord.prototype.constructor = JapaneseWord;
EnglishWord.prototype = Object.create(Word.prototype);
EnglishWord.prototype.constructor = EnglishWord;
```

Note that instead of relying on the constructor so heavily, we could have also let the subclass-like constructor functions do more work, as in the following:

```
function Word(){
  this.count = function(){
    return this.word.length;
  };
  this.lookUp = function(){
    return this.lookUpUrl + this.word;
  };
};

function EnglishWord(word){
  Word.call(this);
  this.word = word;
  this.language = "English";
  this.lookUpUrl = 'https://en.wiktionary.org/wiki/';
};

function JapaneseWord(word){
  Word.call(this);
  this.word = word;
  this.language = "Japanese";
  this.lookUpUrl = 'http://jisho.org/search/';
};

JapaneseWord.prototype = Object.create(Word.prototype);
JapaneseWord.prototype.constructor = JapaneseWord;
EnglishWord.prototype = Object.create(Word.prototype);
EnglishWord.prototype.constructor = EnglishWord;
```

This should all look fairly familiar, except for the last four lines. If we had tests in place, all of our tests and logging statements using the interfaces we've

dealt with would pass fine without these lines. However, they are important for establishing the object hierarchy. If we don't assign the prototype, then any changes we make to `Word.prototype` will not update its subclass-like objects. For example:

```
// with the last 4 lines from the previous snippet removed
Word.prototype.reportLanguage = function(){
  return `The language is: ${this.language}`;
};
const japaneseWord = new JapaneseWord("犬");
console.log(japaneseWord.reportLanguage());
```

This will result in an error, because `japaneseWord` doesn't know about its ancestry. Adding just the first line would fix this:

```
JapaneseWord.prototype = Object.create(Word.prototype);
```

Even without these lines, all of our original linkages to properties are still okay. In other words, our interface tests will still pass when the code is like this:

```
function Word(word, language, lookUpUrl){
  this.word = word;
  this.language = language;
  this.lookUpUrl = lookUpUrl;
  this.count = function(){
    return this.word.length;
  };
  this.lookUp = function(){
    return this.lookUpUrl + this.word;
  };
};

function EnglishWord(word){
  Word.call(this, word, "English", 'https://en.wiktionary.org/wiki/');
};

function JapaneseWord(word){
  Word.call(this, word, "Japanese", 'http://jisho.org/search/');
};

// JapaneseWord.prototype = Object.create(Word.prototype);
// JapaneseWord.prototype.constructor = JapaneseWord;
// EnglishWord.prototype = Object.create(Word.prototype);
// EnglishWord.prototype.constructor = EnglishWord;

// Word.prototype.reportLanguage = function(){
//   return `The language is: ${this.language}`;
// };
```

```
const japaneseWord = new JapaneseWord("犬");
// console.log(japaneseWord.reportLanguage());

const englishWord = new EnglishWord("dog");

const wish = require('wish');
const deepEqual = require('deep-equal')

// interfaces tests
wish(japaneseWord.word ===  "犬");
wish(japaneseWord.lookUp() === "http://jisho.org/search/犬");
wish(japaneseWord.count() === 1);

wish(englishWord.word ===  "dog");
wish(englishWord.lookUp() === "https://en.wiktionary.org/wiki/dog");
wish(englishWord.count() === 3);
```

Note that the same goes for the case of leaving more to the subclasses.

However, two of our internal tests will fail if we omit manually linking the prototype and constructor via:

```
JapaneseWord.prototype = Object.create(Word.prototype);
JapaneseWord.prototype.constructor = JapaneseWord;
```

Namely, these two tests:

```
wish(japaneseWord instanceof Word);
wish(Object.getPrototypeOf(JapaneseWord) === Word);
```

With just the first line in place:

```
JapaneseWord.prototype = Object.create(Word.prototype);
// JapaneseWord.prototype.constructor = JapaneseWord;
```

these tests will fail:

```
wish(japaneseWord.constructor === JapaneseWord);
wish(Object.getPrototypeOf(JapaneseWord) === Word);
```

And even with manually establishing both the prototype and constructor, our test from the sketchy bits will fail:

```
// sketchy bits
wish(deepEqual(Object.getPrototypeOf(japaneseWord), {}));
console.log(Object.getPrototypeOf(japaneseWord));
// prints JapaneseWord { constructor: [Function: JapaneseWord] }
```

as well as this one:

```
wish(Object.getPrototypeOf(JapaneseWord) === Word);
```

The same is true for our variant of the hierarchy that put more logic into the subclasses.

WHICH JAVASCRIPT IS BETTER?

In JavaScript, any time you want to know what an object is, know what is inside of it, know how it came to be, loop through it, or create a new one based on it, you have a lot of bad options.

My general recommendation is to stick to recent sources. You'll see these lean toward two types of people: spec-friendly and puristic. If you follow spec-friendly people, you'll have decent references for APIs, and you'll see many critiques and questions around the same problems you're having. Following the purists also has value, because they do the most questioning, push specs to improve, and spend a lot of time endorsing approaches that work well but might not be as popular. You can learn a lot from purists.

As far as inheritance, there has always been a group desperately pushing for OOP, as well as a group embracing the "true" nature of JavaScript. As of this writing, it seems that both of these groups are somehow paradoxically winning. While that's great, it also adds an extra layer of confusion, especially when you factor in how both approaches may change year by year.

Object Literals

Next, we'll turn to another approach we've used before: object literals. Let's start with our superclass, which in this style is instead called the *delegate prototype*. It's just an object:

```
const word = {
  count(){
    return this.word.length;
  },
  lookUp(){
    return this.lookUpUrl + this.word;
  }
};
```

So how do we inherit from it? Here is where we get stuck. Should we rely on constructor functions for the subclasses?

We don't have to. Here is the simplest approach:

```
const englishWord = Object.create(word);
englishWord.word = 'dog';
englishWord.language = 'English';
englishWord.lookUpUrl = 'https://en.wiktionary.org/wiki/';

const japaneseWord = Object.create(word);
japaneseWord.word = '犬';
japaneseWord.language = 'Japanese';
japaneseWord.lookUpUrl = 'http://jisho.org/search/';
```

That seems a little clunky, though. We create a copy of the `word` object with `Object.create`, but then we have to assign all of our new properties in an awkward way.

Let's try using ES2015's `Object.assign`:

```
const englishWord = Object.assign(Object.create(word),
                      {word: 'dog',
                       language: 'English',
                       lookUpUrl: 'https://en.wiktionary.org/wiki/'});

const japaneseWord = Object.assign(Object.create(word),
                       {word: '犬',
                        language: 'japanese',
                        lookUpUrl: 'http://jisho.org/search/'});
```

That's a little better because now we have one simple statement that combines objects, rather than updating them. Changing values of variables (even members of arrays and objects) is problematic, as we saw earlier, so this approach is preferable. By the way, that is also why we're using `Object.create(word)` instead of just `word` as the first argument. The first parameter of `Object.assign` is actually intended to be the "target" object. If we just put `word` in there, it would clobber our original, rather than giving us a fresh copy to augment.

This will pass all of our interface tests, but only one of the internal tests:

```
// interface tests
wish(japaneseWord.word ===  "犬");
wish(japaneseWord.lookUp() === "http://jisho.org/search/犬");
wish(japaneseWord.count() === 1);

wish(englishWord.word ===  "dog");
wish(englishWord.lookUp() === "https://en.wiktionary.org/wiki/dog");
```

```
wish(englishWord.count() === 3);

// internal tests
wish(typeof japaneseWord === 'object');
console.log(Object.getPrototypeOf(japaneseWord));
// prints { count: [Function: count], lookUp: [Function: lookUp] }
```

We can also see that the value of `japaneseWord`'s prototype is different now.

Factory Functions

The object literal option might seem pretty good as far as simplicity goes, but if we're creating a lot of words, it might be clunky compared to using the `new` keyword with either `class` or a constructor function. We'll explore one more alternative now.

We can use *factory functions* like this:

```
const word = {
  count(){
    return this.word.length;
  },
  lookUp(){
    return this.lookUpUrl + this.word;
  }
}
const englishWordFactory = (theWord) => {
  return Object.assign(Object.create(word),
                       {word: theWord,
                        language: 'English',
                        lookUpUrl: 'https://en.wiktionary.org/wiki/'})
};

const japaneseWordFactory = (theWord) => {
  return Object.assign(Object.create(word),
                       {word: theWord,
                        language: 'Japanese',
                        lookUpUrl: 'http://jisho.org/search/'})
};

const englishWord = englishWordFactory('dog');
const japaneseWord = japaneseWordFactory('犬');

// interfaces tests
wish(japaneseWord.word ===  "犬");
wish(japaneseWord.lookUp() === "http://jisho.org/search/犬");
wish(japaneseWord.count() === 1);
```

```
wish(englishWord.word ===  "dog");
wish(englishWord.lookUp() === "https://en.wiktionary.org/wiki/dog");
wish(englishWord.count() === 3);
```

Revisiting our entire reason for subclassing to begin with, we had two main purposes. First, we didn't want to have to repeat code. Second, we wanted to get rid of the `if` statement. This accomplishes both of those goals. Plus, our interface tests are still passing.

So what's the downside?

When we sever the link to some prototype (either through constructor functions or classes), we lose the ability to add attributes to many objects at once. With a class or constructor function, we can do this:

```
const japaneseWord = new JapaneseWord("犬"); // old code

// new code
Word.prototype.reportLanguage = function(){
  return `The language is: ${this.language}`;
}
console.log(japaneseWord.reportLanguage());
```

We can do this even after we create the individual words. With object literals created directly or through factory functions, the objects are unavailable for these kinds of late extensions because they have no prototype chain. Note that this ability to track prototypes broke when we started using factory functions, not as a part of us using `Object.create` or `Object.assign`.

If we want that mechanism with objects that lack a prototype, we can add the prototype directly:

```
japaneseWord.prototype = word;
englishWord.prototype = word;

word.reportLanguage = function(){
  return `The language is: ${this.language}`;
};

console.log(japaneseWord.reportLanguage());
console.log(englishWord.reportLanguage());
```

But for many objects, that would get tedious. Instead, we can do this with factory functions, like this:

```
const wordFactory = function(){
  return {count(){
      return this.word.length;
```

```
    },
    lookUp(){
      return this.lookUpUrl + this.word;
    }
  };
};

const englishWordFactory = (theWord) => {
  let copy = Object.assign(wordFactory(),
                        {word: theWord,
                         language: 'English',
                         lookUpUrl: 'https://en.wiktionary.org/wiki/'})
  return Object.setPrototypeOf(copy, wordFactory);
};

const japaneseWordFactory = (theWord) =>{
  let copy = Object.assign(wordFactory(),
                        {word: theWord,
                         language: 'Japanese',
                         lookUpUrl: 'http://jisho.org/search/'})
  return Object.setPrototypeOf(copy, wordFactory);
};
const englishWord = englishWordFactory('dog');
const japaneseWord = japaneseWordFactory('犬');

wordFactory.reportLanguage = function(){
  return `The language is: ${this.language}`;
};
console.log(japaneseWord.reportLanguage());
console.log(englishWord.reportLanguage());
```

It's not strictly necessary to use a factory function for `word`, but this does avoid the extra step of creating a prototype (and possibly an arbitrary one) for it. Although it might seem like a hassle to manually set the prototype inside of the specific language factory functions, the alternative of manually setting them for each instance of a word would be worse if there are too many objects.

Evaluating Your Options for Hierarchies

To wrap up this section, recall that we looked at four ways to build a hierarchy:

- Classes
- Constructor functions
- Object literals
- Factory functions

Which one you choose in a given situation is up to you. Constructor functions are probably the worst option. Because they have been a perennial attempt at classes, many variants exist, and it is hard to find great documentation on them (a web search will show you ways to do them from five, six, and seven years ago—all different). If you're considering them, just go for classes instead.

Factory functions give you a bit more control (including possibilities for establishing a prototype chain) as compared with object literals. But for simple cases, object literals are easier to work with.

Your team or codebase (including libraries you use) might be dedicated to certain methods of establishing a hierarchy. In those cases, sticking to what's there is a decent option. You could encounter two pathological approaches, however. The first is an overreliance on manipulating and inspecting prototype, constructor, and associated utilities. For that, the remedy is to keep things superficial as much as you can. Keep testing at an interface level and focus your concerns there.

The rest of this chapter is dedicated to handling the second case: an overcommitment to hierarchies and OOP.

Inheritance and Architecture

So far, we've seen a few different patterns for copying data, ranging from the classiest (pseudo-) classical to the factory function that happily ignores most of the ideas around prototypes, constructors, and any other lines we'd need to draw from box to box in a UML diagram.

To recap, we know that we can make solitary objects using:

- Object literals
- Classes (uses `new`)
- Constructor functions (uses `new`)
- Factory functions (just returns an object)
- `Object.assign` and `Object.create`
- Libraries (like mori and Immutable.js)

Additionally, when it's preferable to objects, we can make more specific containers as we need them, such as:

- `Set`
- `Map`
- `Array`
- `String`
- `Function`

- Other JavaScript types

It's already a lot of options, but in order to round out the discussion, let's address architectural concerns a bit more deeply, centered on answering the following questions:

- Why do some people hate classes?
- What about multiple inheritance?
- Which interfaces do you want?

Why Do Some People Hate Classes?

The first argument that some people latch onto is that `new`, `this`, `super`, and anything else having to do with *classical* OOP style detracts from their preferred mechanism of sharing data and behavior between JavaScript objects (hurting the purity and making things confusing)—but some of these criticisms are of object-oriented programming itself. They argue that it promotes bad practices like tight coupling of components, and encourages deep hierarchies and mutable state.

The `class` keyword is a hard blow to those who have been advocates for avoiding class-based OOP in JavaScript for a long time.

On the pro side of things, people have always struggled awkwardly to force JavaScript to behave this way. If it is what Douglas Crockford might call a "footgun," then at the very least, condoning and standardizing the "footwounds" will make problems easier to search for and fix. There's some degree of herd immunity if we're all suffering from the same disease.

Also on the pro side, as features grow around classes, richer APIs become available for reflection and inspecting your code. For example, error messages can be more detailed by default.

Additionally, having a sanctioned standard for something makes it easier to learn once (adapting with the spec as necessary), rather than chasing the latest and greatest way to avoid mostly benign "syntactic sugar" for constructor functions.

The criticisms of classical OOP itself should be taken very seriously, however. A deeply nested hierarchy will be difficult to debug and maintain. Critics are not wrong to say that the `extends` keyword encourages a deep hierarchy (e.g., class `Dog` extends `Pet` extends `Animal` extends `Organism`), but it does not necessarily demand it. They're also not wrong to feel that parent/child classes tend to be tightly coupled.

What About Multiple Inheritance?

When people discuss multiple inheritance, they mean a child class inheriting from multiple parents. If we had a clean solution for this in JavaScript, it would probably mean allowing for `extends` to take multiple base classes as parents.

Although this type of mechanism is supported in some languages (Python, for example), it can create some confusion. If class A gets all its properties from classes B and C, how does it resolve conflicts when a property exists in both parents? It depends on the language.

If JavaScript had a class-based mechanism for multiple inheritance, it might look something like this:

```
class Barky{
  bark(){ console.log('woof woof')};
};
class Bitey{
  bark(){ console.log('grrr')};
  bite(){ console.log('real bite')};
};
class Animal{
  beFluffy(){ console.log('fluffy')};
  bite(){ console.log('normal bite')};
};

// this is not possible:
class Dog extends (Animal, Barky, Bitey) { };
dog = new Dog;
dog.bite();
dog.beFluffy(); // this won't work
```

But it doesn't. `Dog` will bite like a `Bitey` but has no idea how to be fluffy like an `Animal`. This isn't multiple inheritance. `Dog` extends `Bitey`. It's as simple as that. This is not a mechanism for multiple inheritance at all. Rather, `(Animal, Barky, Bitey)` just evaluates to `Bitey`. Check out the following:

```
// try this in the console:
(1, 4, 3, 7);
```

What do you expect this to evaluate to? Well, the answer is `7`. In some languages this expression would be an array or list literal. In others it would throw an error. In JavaScript, we get this questionable behavior of returning the last value inside the parentheses. This is what happened with our earlier attempt at multiple inheritance. This weird mechanism does *not* make class-based multiple inheritance possible. `Dog` extends `Bitey`, and `Bitey` alone. End of story.

JavaScript doesn't have a class-based way to do multiple inheritance. For that you need some other approach. Other languages would term this a *module*, *interface*, or *mixin*.

What does that look like in JavaScript? Here's one option that we've seen earlier in the chapter:

```
const barky = {
  bark(){ console.log('woof woof')}
};
const bitey = {
  bark(){ console.log('grrr')},
  bite(){ console.log('real bite')}
};
const animal = {
  beFluffy(){ console.log('fluffy')},
  bite(){ console.log('normal bite')}
};
const myPet = Object.assign(Object.create(animal), barky, bitey);
myPet.beFluffy();
myPet.bark();
myPet.bite();
```

This will give us access to all three functions (`bark`, `bite`, and `beFluffy`). But if we dig a little deeper, we'll see that this is not as simple as we might expect:

```
console.log(myPet);
{ bark: [Function: bark], bite: [Function: bite] }
console.log(Object.getPrototypeOf(myPet));
{ beFluffy: [Function: beFluffy], bite: [Function: bite] }
```

Our `myPet` object is actually relying on its prototype, the `animal`, for the `beFluffy` function. These are linked, so if we alter our `beFluffy` function on `animal`, it will update on `myPet` as well:

```
animal.beFluffy = function(){ console.log('not fluffy')}
myPet.beFluffy();
// prints "not fluffy"
```

We can even add new attributes to that `animal` to make changes to `myPet` on the fly:

```
animal.hasBankAccount = false;
console.log(myPet.hasBankAccount); // prints false
```

What if we try to augment the `bite` function?

```
bitey.bite = function(){
  console.log("don't bite");
}
myPet.bite();
// prints "real bite"
```

There are two implications here. First, the `bite` function is attached directly to `myPet` from `bitey`. Even though `myPet`'s prototype (`animal`) also has a `bite` function, it is overruled by the function directly attached to `myPet` that comes from `bitey`. Second, it is an actual copy. There is no linkage between `bitey` and `myPet`.

As far as our last function, `bark`, does `myPet` inherit this from `bitey` or `barky`?

Let's look again at how we created `myPet`:

```
const myPet = Object.assign(Object.create(animal), barky, bitey);
```

It turns out that, as parameters to `Object.assign`, the properties of objects to the right will overrule those to the left:

```
myPet.bark();
// prints "grrr"
```

`bitey`'s `bark` beats `barky`'s `bark` because `bitey` was last.

IS-A VERSUS IS-JUST-A RELATIONSHIPS

With simple, one-parent inheritance, it is generally obvious when we intend to say that something is a subtype of something else (for example, a `Person` is a subtype of `Organism`).

But what if a `Person` is also a subtype of `DatabaseRecord` or (shudder) `Resource`?

In those cases, our model of a person is not "just" an organism, so multiple inheritance makes more sense. We just attach relevant behaviors.

It isn't always easy to guess how a program will evolve, and inheritance hierarchies tend to be inflexible. If you want some inspiration for other ways to model object relationships, we'll cover has-a later in this chapter, but you might also want to investigate database normalization, as well as entity-component systems, which tend to be used heavily in game development.

Which Interfaces Do You Want?

There's more to the story here. Although we showed earlier that a factory function can sever the prototypal link in the `Object.create` pattern, there is another option. Before, we used this:

```
const myPet = Object.assign(Object.create(animal), barky, bitey);
```

That creates a prototypal link between `myPet` and `animal`. This can be useful because we can add functionality to `animal` and have it inherited by `myPet` (as well as other objects that were created through a similar mechanism).

But if we want a new object with the existing behavior, but without the baggage of a prototype, we could do this instead:

```
const myPet = Object.assign({}, animal, barky, bitey);
```

Now `animal` acts just like `barky` and `bitey`. We can add properties to *them*, but `myPet` is unaffected:

```
animal.hasBankAccount = false;
console.log(myPet.hasBankAccount); // prints undefined
```

Another way to look at this is that `myPet`'s prototype is now `{}`, rather than `animal`:

```
console.log(Object.getPrototypeOf(myPet));
// prints "{}"
```

Also notice that `beFluffy` (the function inherited from `animal`) is defined directly on `myPet`:

```
console.log(myPet);

// prints
{ beFluffy: [Function: beFluffy],
  bite: [Function: bite],
  bark: [Function: bark] }
```

Most of the time, your primary concern will just be whether or not an object has a particular function or other property attached to it. In other words, "Can I call this function on this object?" is a question that is absolutely fundamental in defining your API. As such, it is worth documenting, testing, manipulating, and refactoring.

But there is another broad and confusing set of interfaces underneath every JavaScript program. Do you want to use `Object.getPrototypeOf`? What about `Object.is`? `.__proto__`? `.prototype`? `.constructor`? `.keys`? `Reflect.ownKeys`? `Reflect.has`? `typeof`? `instanceof`?

Let's call this "JavaScript's deep, dark underbelly."

Are you confident that these all behave as you expect? Are the desired properties enumerable when you want to loop through an object? When you create a new object, are you exposing either your new or old objects to side effects?

You're likely to encounter projects that favor one style or another. You might even see a codebase with a dual personality, perhaps with half of it in a classical style and the other half using factory functions. Whatever your preferred style is, rewriting the "wrong" style is likely to be painful, and possibly confusing and insulting to other team members.

Although it is best to promote confidence early in a project by having answered **Chapter 2**'s question of "Which JavaScript are you using?" with a style guide (and better yet, a linter that runs in your editor to enforce this automatically), you won't always be so lucky.

On a project that lacks opinions on a given style, and hurts confidence in that second interface (that of inspection and reflection), what should you do?

If you are not confident in a piece of code, write a test. For example, for this code:

```
const myPet = Object.assign({}, animal, barky, bitey);
```

use characterization tests to develop confidence in things like `Object.keys(myPet)`, `myPet.__proto__`, `Object.getOwnPropertyNames(myPet)`, `Object.getPrototypeOf(myPet)`, and any other members of this deeper interface of JavaScript. As we practiced in **Chapter 4**, use `wish(value, true)` or `assert(value === null)`. Then use the output of that characterization test to fill in the test: `wish(value === valueFromOutput)` or `assert(value === valueFromOutput)`.

Aren't these values already tested within JavaScript browser implementations? Yes, absolutely; they already have tests covering them. Are they "implementation details," which many argue against writing tests for? Definitely. Will these tests be brittle if you change from classes to factory functions or from factory functions to classes? Yes.

But if they cause confusion in your code, should you test these functions in spite of that?

Without a doubt. It does not matter how much a piece of code is *external to your codebase* or how much of an *implementation detail* it is. You can use tests to clear up your confusion (you won't be the only one on the team who's confused anyway) and have more confidence in your code. Additionally, you may

be relying on JavaScript's deep, dark underbelly more than you think. For instance, what seems like it should be a cosmetic change might be affecting a property's enumerability or an object's prototype.

If letting your code break in tests instead of in production is your goal, creating characterization tests like these can add a layer of confidence that rigidly only testing the public interface may not give you. Additionally, if you're moving between factory functions, constructor functions, object literals, and classes, it's not a bad idea to have a sense of what is changing, even at a deeper level than the interface you directly care about.

Has-A Relationships

So far in this chapter, we've been looking at multiple objects from the perspective of "is-a" (or "is a subtype of") relationships. For simple inheritance, an `EnglishWord` is a `Word`. This is true even when dealing with mixins/multiple inheritance/modules: `myPet` is-a `animal`, but `myPet` is-a `bitey` and `myPet` is-a `barky` also.

We saw how these is-a relationships may *imply* a prototypal link between child and parent, but according to stylistic choices within JavaScript, that link may be severed despite any adopted similarities of interface between the objects. We also saw how establishing or severing that link can impact many interfaces within JavaScript's deep, dark underbelly.

What we haven't talked about is that *composing objects does not necessarily imply inheriting properties*.

Is a `HorseWithABriefCase` a subtype of `Horse`? That sounds awfully specific. Maybe it's a subtype of `HorseWithObject`, or should it be a subtype of `BusinessHorse`? We could put all of our horses into a hierarchy of some form, but then we'll need an object (and/or class) that ends up describing every attribute, lest our `SickHorse` cannot get the antibiotics she needs because `DoctorHorseWithoutAnyMedicine` is perfectly unhelpful in that scenario.

Obviously, we already have a way to deal with this by storing properties (of arbitrary types) on objects. In Chapters **6** and **7**, we dealt with a `classifier` that *has-a* `songList`. In our `Object.create`/`Object.assign` mixin-based inheritance from earlier in the chapter, we were copying properties from one object to another. By contrast, in our `classifier`, we had to specifically drill down into the `songList` via `classifier.songList.addSong` (until we wrote the delegate function to drill down for us, allowing us to instead write `classifier.addSong`).

As for our horses, we could start by doing something like this:

```
const horse = {
  inventory: ["briefcase"],
  profession: "hippo jockey",
  healthy: true
}
```

As our program grew in complexity, we might need a lot of horses, so we'd use a class, a constructor function, `Object.create`, or a factory function to help us get more horses. If we needed to add behaviors to the horses, we would add functions as properties.

If `inventory` or `briefcase` or other properties needed to be more complicated, we'd spin them out as their own objects (as we did with `songList` in **Chapter 7**). If they needed some behavior to be attached, we'd add functions to those objects.

The reason for glossing over a general approach for how to grow JavaScript objects without a hierarchy is that it is not usually how things go wrong in JavaScript programs. Although building a complex, deep, or complicated hierarchy of JavaScript classes/objects is possible and may become more of a widespread problem as OOP with classes becomes more popular, as of this writing the worst JavaScript codebases are much more likely to suffer from a *lack of* structure than from *too much* structure of the wrong kind.

That said, next we'll address a few structural mistakes that you might see from time to time.

Inheritance Antipatterns

Here we'll look at two issues that can come up when the wrong kind of structure is in place. Here are the antipatterns we'll address:

- Hyperextension (hierarchy is too deep)
- Goat and cabbage raised by a wolf (parent and children have nothing in common)

In both cases, the motivation likely comes from a good place. Programmers don't like to just copy and paste code with minor variations. This is called "cargo cult" coding, and is a brand new programmer's best friend.

To avoid that, you create new aspects of a hierarchy instead of extracting functions and subobjects that could be shared or held independently.

Hyperextension

Here we have an example of a hierarchy that is too deep. `SpecificClientReport` inherits from `ClientReport`, which inherits from `GenericReport`, which inherits from `Report`:

```
class Report{
  constructor(params){
    this.params = params;
  }
  printReport(params){
    return params;
  }
}
class GenericReport extends Report{
  constructor(params){
    super(params);
    this.params = params;

  }
  printReport(params){
    return super.printReport(Object.assign(this.params, params));
  }

}
class ClientReport extends GenericReport{
  constructor(params){
    super(params);
    this.params = params;

  }
  printReport(params){
    return super.printReport(Object.assign(this.params, params));
  }

}
class SpecificClientReport extends ClientReport{
  constructor(params){
    super(params);
    this.params = params;
  }
  printReport(params){
    return super.printReport(Object.assign(this.params, params));
  }

}
const report =
new SpecificClientReport({whatever: 'we want', to: 'add'});
console.log(report.printReport({extra: 'params'}));
```

If there were ever a good reason not to just pass the word *params*, *object*, or *options* around, here it is. Imagine this code, but with more functions, and alterations of `params` *and* `this.params` all along the way.

If we had function calls to other members of this hierarchy, we would address them too, but here we'll just focus on `SpecificClientReport`. First, we need a test instead of a logging statement. Create a characterization test by adding these lines to the bottom:

```
const wish = require('wish');
const deepEqual = require('deep-equal');
wish(report.printReport({extra: 'params'}), true);
```

We run it and get an error:

```
WishCharacterization: report.printReport({extra: 'params'})
  evaluated to {"whatever":"we want","to":"add","extra":"params"}
```

That object is exactly what we need to replace our characterization test:

```
wish(deepEqual(report.printReport({extra: 'params'}),
     {whatever:"we want", to:"add", extra:"params"}));
```

And our test passes. Now we can safely refactor.

There are two approaches we could take. First, we could assume that there's nothing special about our `SpecificClientReport` and try this instead:

```
const report = new Report({whatever: 'we want', to: 'add'});
wish(deepEqual(report.printReport({extra: 'params'}),
     {whatever:"we want", to:"add", extra:"params"}));
```

Note that we're just changing the `report` variable here (and in the next snippet). The test is the same. We get an error, so we could try moving down one level:

```
const report = new GenericReport({whatever: 'we want', to: 'add'});
wish(deepEqual(report.printReport({extra: 'params'}),
     {whatever:"we want", to:"add", extra:"params"}))
```

This works fine. That means (assuming there are no other places using them) we can delete `SpecificClientReport` and `ClientReport`. If they both have other places using them, we should see if we could move those places up in the hierarchy to try to prune any leaves we can.

The second way to handle this is to see if we can inline the superclass's functionality. Because `SpecificClientReport` and `GenericReport` behave the

same way (according to our tests), it's fine to start where we just left off. If the last approach was difficult for your codebase, you might start with this technique instead (before climbing the superclasses).

In any event, we're left with this:

```
class Report{
  constructor(params){
    this.params = params;
  }
  printReport(params){
    return params;
  }
}
class GenericReport extends Report{
  constructor(params){
    super(params);
    this.params = params;

  }
  printReport(params){
    return super.printReport(Object.assign(this.params, params));
  }

}
const wish = require('wish');
const deepEqual = require('deep-equal');
report = new GenericReport({whatever: 'we want', to: 'add'});
wish(deepEqual(report.printReport({extra: 'params'}),
     {whatever:"we want", to:"add", extra:"params"}));
```

To get a little clarity, we can start by removing the `constructor` function from `GenericReport` (assuming that we are the only consumer), as it is redundant with what happens in `super`. As for the `printReport` function, all `super` does is return what it is given (the `params`, not the `this.params`). That means we don't have to call `super` at all, making our new `GenericReport` look like this:

```
class GenericReport extends Report{
  printReport(params){
    return Object.assign(this.params, params);
  };
};
```

The test passes. We're almost ready to free this child class. However, if we remove `extends Report` now, we'll get an error, because `this.params` was

never assigned. That means we need our constructor back first. After that, we're free to remove the link to the parent:

```
class GenericReport{
  constructor(params){
    this.params = params;
  };
  printReport(params){
    return Object.assign(this.params, params);
  };
};
```

And everything works. If `GenericReport` was the only thing using `Report`, we could remove it (`Report`) now. After that, the ugliest things in here are the nonspecific names: `params` should be something more specific (see **Chapter 6**), and `GenericReport` could possibly be called `Report` now.

If you decide to prune a hierarchy like this, it is crucial to have version control and tests in place that can check all of the consumers (object makers/function callers) of the classes along the hierarchy.

It's unlikely to go as smoothly as this. Ultimately, the goal is to carve off independent leaves and extract shared functions when you can. Don't try to "refactor" a complex hierarchy like this (but with many more side effects, attributes, and functions) all at once, especially not without tests in place.

Goat and Cabbage Raised by a Wolf

In this case, the children look nothing like the parent or each other. All the parent does is cause confusion and add a layer of indirection:

```
class Agent{
  constructor(name, type){
    this.name = 'name';
    if(Math.random() > .5){
      this.type = 'user';
    }else{
      this.type = 'project';
    }
  };
  static makeProjectOrUser(agent){
    if(agent.type === 'user'){
      return Object.assign(Object.create(new User), agent);
    }else{
      return Object.assign(Object.create(new Project), agent);
    }
  };
};
```

```
class User extends Agent{
  sayName(){
    return `my name is ${this.name}`;
  }
};

class Project extends Agent{
  sayTheName(){
    return `the project name is ${this.name}`;
  }
};
const agent = new Agent('name');
const projectOrUser = Agent.makeProjectOrUser(agent);
```

For this code, imagine that this is basically as common as it gets between the two objects, and there are thousands of lines backing up the difference between them (all requiring tests and type checking, aka conditionals).

First, we need to get some characterization tests in place. Add these lines to the end:

```
const wish = require('wish');
if(projectOrUser.type === 'user'){
  wish(projectOrUser.sayName(), true);
}else{
  wish(projectOrUser.sayTheName(), true);
}
```

We'll have a 50/50 chance of getting one of these:

```
WishCharacterization: projectOrUser.sayName() evaluated to
  "my name is name"

// or

WishCharacterization: projectOrUser.sayTheName() evaluated to
  "the project name is name"
```

Now we can just replace our characterization tests with these:

```
if(projectOrUser.type === 'user'){
  wish(projectOrUser.sayName() === "my name is name");
}else{
  wish(projectOrUser.sayTheName() === "the project name is name");
}
```

We will always have problems as long as the following two lines are in place:

```
const agent = new Agent('name');
const projectOrUser = Agent.makeProjectOrUser(agent);
```

Right now, these create a `project` or a `user`, based on a coin toss in the constructor. Since that is where the whole problem stems from, let's move the coin toss into a function:

```
function coinToss(){
  return Math.random() > .5;
};

class Agent{
  constructor(name, type){
    this.name = name;
    if(coinToss()){
      this.type = 'user';
    }else{
      this.type = 'project';
    }
  };
...
};
```

Next, because `coinToss` is always called, we can move that to our calling code, outside the constructor:

```
class Agent{
  constructor(name, type){
    this.name = name;
    this.type = type;
  };
};
```

and move our `coinToss` to be used before our `agent` is given an identity:

```
let agent;
if(coinToss()){
  agent = new Agent('name', 'user');
}else{
  agent = new Agent('name', 'project');
}
// replaces
// const agent = new Agent('name');
```

Now, we want to "push down" the constructor into the subtypes:

```
class User extends Agent{
  constructor(name, type){
```

```
    super();
    this.name = name;
    this.type = type;
  };
  sayName(){
    return `my name is ${this.name}`;
  };
}

class Project extends Agent{
  constructor(name, type){
    super();
    this.name = name;
    this.type = type;
  };
  sayTheName(){
    return `the project name is ${this.name}`;
  };
};
```

Annoyingly, as long as we're extending `Agent`, in JavaScript we must call its constructor (with `super`) in order to access this and assign properties in our subtypes. Note that we don't care about passing arguments to `super` because we're assigning the properties we need anyway.

Okay, here's the exciting part. Now we can call those constructors directly:

```
let agent;
if(coinToss()){
  agent = new User('name', 'user');
}else{
  agent = new Project('name', 'project');
}
```

Now we can delete a lot of code, leaving us with just the following:

```
function coinToss(){
  return Math.random() > .5;
};

class User{
  constructor(name, type){
    this.name = name;
    this.type = type;
  };
  sayName(){
    return `my name is ${this.name}`;
  };
};
```

```
class Project{
  constructor(name, type){
    this.name = name;
    this.type = type;
  };
  sayTheName(){
    return `the project name is ${this.name}`;
  };
};

let agent;
if(coinToss()){
  agent = new User('name', 'user');
}else{
  agent = new Project('name', 'project');
}

const wish = require('wish');
if(agent.type === 'user'){
  wish(agent.sayName() ===  "my name is name");
}else{
  wish(agent.sayTheName() === "the project name is name");
}
```

No more `super`. No more `extends`. Now we're free to rename the variable `agent` to anything we like. You might notice another potential refactoring in unifying the interface of the functions `sayName` and `sayTheName`. By all means, go ahead, as long as it won't tempt you to subclass these again.

If you think the coin toss to decide on a type of object is a bit far-fetched, you're totally correct. The point is, however, that ambivalent constructors—which could be based on date, time, or any other dynamic situation—*could* lead to this scenario. Or such a scenario could be the result of one ambiguous object creation slowly spreading throughout the codebase. I've seen it happen.

Also note, as we've discussed before, that having our test suite rely on randomness like this is not great. Really, we'd want the following tests as well:

```
wish(new User('name', 'user').sayName() === "my name is name");
wish(new Project('name', 'project').sayTheName()
 === "the project name is name");
```

Since our coin toss is just representing some (probably very confusing) way that the objects come into being, there's no clear way to remove that conditional. However, note that instead of passing an explicit type variable into the constructor, we could rely on the class instead:

```
class User{
  constructor(name){
    this.name = name;
  };
...
class Project{
  constructor(name){
    this.name = name;
  };
...
if(agent instanceof User){
  wish(agent.sayName() ===  "my name is name");
}else{
  wish(agent.sayTheName() === "the project name is name");
}
...
```

The lesson here is that if you have two classes that share some state or behavior, subclassing may not be a great way to remove duplication, and it could lead your team down a path of lots and lots of type checking. Sometimes, you just need a type attribute (we'll look at a way to avoid a conditional check with these in **Chapter 9**). Other times, you might be able to extract the duplication into functions or another object. In any case, refactoring duplication is very easy and safe compared to dismantling a "goat and cabbage raised by a wolf" inheritance issue.

Wrapping Up

In this chapter, we looked at a few ways to build up as well as break down a hierarchy. It's worth noting that for some people, object-oriented programming is more trouble than it's worth. If you're one of those people, you'll probably like learning about functional programming in **Chapter 11**. If you like OOP, the next chapter will give you a few more tools (patterns) to apply to your work.

Whatever your stance on OOP, you're likely to see code that could benefit from the kind of organization that it provides. On the other hand, you might find that misapplied OOP itself is the cause of complexity in some codebases. In any case, none of this suggests that objects or classes are inherently bad to use.

9 Refactoring to OOP Patterns

It is tempting to treat software design patterns as checklists to learn. There's something so nice about the idea that if we just "memorize those 23 things," we'll be all set. The bad news is that, as mentioned in **Chapter 2**, even learning the latest in JavaScript should keep you busy until the free and open web fails or the sun goes supernova.

Additionally, patterns have a mixed reputation. On the one hand, they can help to handle complexity. On the other hand, they can *create* complexity where it is not needed. Sometimes, extracting functions, extracting objects, breaking code up into modules, and depending on a framework (which itself likely exposes and documents patterns of its own) are simpler choices. Keeping the YAGNI ("Ya ain't gonna need it") principle in mind and considering the interface you want for your code are the best guidelines when deciding when to implement (or remove) a design pattern.

The good news is that these patterns aren't that hard to learn and are easy to reference so you don't have to memorize every detail. Additionally, we'll focus on just seven of them here, chosen by their likelihood to solve real problems in legacy code. Just learn these seven patterns, and it will change your life![1] Here they are:

- Template method
- Strategy
- State
- `null` object
- Decorator ("Wrapper" section)
- Adapter ("Wrapper" section)
- Facade

As we're describing fairly high-level changes for most of these patterns, it's likely that the specific interfaces will change. And we should be aware that when changing an interface, we might not be "refactoring." Sometimes, pat-

1 Disclaimer: Life changes may be somewhat subtle.

terns are presented with mechanics to provide a safe path between code before and after a pattern is applied. In lieu of those, the practical steps for applying *every* pattern presented are:

1. Save and check the code into version control.
2. Make a small change.
3. Repeat steps 1 and 2 until you're done.

Follow that process, and write tests as you see fit. We've already explored a few ways to do that, including test-driven development (TDD), characterization tests, end-to-end tests, and unit tests.

We are striving for code we can be confident in. That means in addition to code being well tested, it should also provide sensible interfaces. That is the priority in this chapter.

Template Method

The template method pattern is useful for when you have two algorithms that serve the same purpose, with minor variations. The basic mechanics require moving some part of two subclass functions into the parent class.

Although the refactoring itself is fairly simple, in this section we'll talk about how you might get to the point of wanting to do it, as well as what you might do instead.

Let's say you have a `Person` class. As the saying goes, there are `10` types of people: those who understand binary numbers, and those who don't. To start out, our `Person` manages this knowledge in a boolean type variable:

```
class Person{
  constructor(binaryKnower){
    this.binaryKnower = binaryKnower;
  };
  whatIs(number){ return number };
  whatIsInBinary(number){ return Number('0b' + number) };
};

const personOne = new Person(true);
const personTwo = new Person(false);

[personOne, personTwo].forEach(person => {
  if(person.binaryKnower){
    console.log(person.whatIsInBinary(10));
  } else{
    console.log(person.whatIs(10));
  }
});
```

Our `personOne` knows binary, so we'll get 2 (which is *one, zero* in binary in case you missed the joke) logged first. Our `personTwo` just assumes that *one, zero* is `10`, so that's logged second.

This works fine, but it leaves our interrogation of binary knowledge up to our client code (e.g., our interface, tests, or whatever code might use this as a module). Consequently, we're stuck with an `if` statement any time we want to get all of our people to interpret a number. Generally speaking, we want to eliminate conditionals where it is sensible and also push complexity away from the API (which you might think of as "the public interface," "the inputs and outputs for the tests," or "the way we use the code"). It's generally better if we can let the deeper parts of the code handle complexity.

Let's move that conditional into the `Person` object, and while we're at it, let's use a string for binary knowledge instead of a boolean:

```
class Person{
  constructor(typeOfPerson){
    this.typeOfPerson = typeOfPerson;
  }
  whatIs(number){
    return number;
  };
  whatIsInBinary(number){
    return Number('0b' + number);
  };
  log(number){
    if(this.typeOfPerson === "binary knower"){
      console.log(this.whatIsInBinary(10));
    } else{
      console.log(this.whatIs(10));
    }
  };
};

const personOne = new Person("binary knower");
const personTwo = new Person("binary oblivious");

[personOne, personTwo].forEach(person => {person.log(10)});
```

Now our conditional is neatly tucked away in the `log` function of the `Person` object. Next, instead of just having a type variable to let the logging function know what to do, let's use subclasses and get rid of the conditional altogether:

```
class Person{}

class BinaryKnower extends Person{
  log(number){
    console.log(this.whatIsInBinary(number));
  };
  whatIsInBinary(number){
    return Number('0b' + number);
  };
};

class BinaryOblivious extends Person{
  log(number){
    console.log(this.whatIs(number));
  };
  whatIs(number){
    return number;
  }
};
const personOne = new BinaryKnower();
const personTwo = new BinaryOblivious();
[personOne, personTwo].forEach(person => person.log(10));
```

Now our `log` function is looking a little repetitive. Let's bring it into the `Person` object:

```
class Person{
  log(number){
    console.log(this.whatIs(number));
  };
};

class BinaryKnower extends Person{
  whatIs(number){ return Number('0b' + number) };
};

class BinaryOblivious extends Person{
  whatIs(number){ return number };
};
const personOne = new BinaryKnower();
const personTwo = new BinaryOblivious();
[personOne, personTwo].forEach(person => person.log(10));
```

Notice that we had to give our functions (`whatIs` and `whatIsInBinary`) the same name to make that change.

WHEN DID WE APPLY THE TEMPLATE METHOD PATTERN?

In case it's unclear, it was in this last step of moving the `log` function from the subclasses into `Person`. The template method is really simple, so you might accidentally stumble into doing it just by moving functions around between objects. It is a specialized form of what is sometimes called the "pull-up method" of refactoring,where functions (or methods, depending on the context) are brought into the superclass. What makes it the template method pattern is that some of the implementation (in this case, the `whatIs` function) creates variation from the subclasses.

A Functional Variant

For a function as basic as ours, you might wonder if it justifies the superclass. We could also have done this:

```
function log(person, number){
  console.log(person.whatIs(number));
};
class BinaryKnower{ whatIs(number){ return Number('0b' + number) } };
class BinaryOblivious{ whatIs(number){ return number } };
const personOne = new BinaryKnower();
const personTwo = new BinaryOblivious();
[personOne, personTwo].forEach(person => { log(person, 10) });
```

Now we're passing two *explicit* arguments to the function, and must explicitly reference the `person` object rather than `this`. In the previous code snippet, we had one *implicit* parameter and one *explicit* one. You might prefer either approach, but making a choice is important, as you'll likely want to have one dominant style in the codebase. Objects help to group and namespace functions, but there is additional complexity in sharing functions between objects.

If you're tending toward an object-oriented codebase, what we had before is preferable. If your codebase is intended to be functional, you would probably opt for this, keeping in mind that you must keep the scope of the function small (likely through creating modules) in order to prevent name collisions. Additionally, you might wonder if the `BinaryKnower` and `BinaryOblivious` classes are needed. It is easy to make the case against them if they only contain one function:

```
function log(fun, number){
  console.log(fun(number));
};
function whatIsInBinary(number){return Number('0b' + number)};
function whatIs(number){return number};
```

```
[whatIsInBinary, whatIs].forEach(fun => { log(fun, 10) });
```

Presumably, in a real application, we'd have a bit more to our `Person` objects, which might justify using a class (or at least an object) rather than just the function as we did here. However, it's also entirely possible to organize your code as a series of functions and return values. Look for opportunities to do both.

Since this chapter is primarily for the purpose of illustrating object-oriented patterns, we'll ignore the functional variant for now.

Strategy

The template method pattern allowed us to remove a conditional through subclassing. The strategy pattern allows us to remove the subclasses by attaching a strategy (function) to the parent object.

Let's take a look at what we have now:

```
class Person{
  log(number){ console.log(this.whatIs(number)) };
};

class BinaryKnower extends Person{
  whatIs(number){ return Number('0b' + number) };
};

class BinaryOblivious extends Person{
  whatIs(number){ return number };
};

const personOne = new BinaryKnower();
const personTwo = new BinaryOblivious();
[personOne, personTwo].forEach(person => person.log(10));
```

As we took a critical eye to the `Person` class in the last section, now let's consider whether we need the subclasses. We could avoid them by setting a type value in the constructor and recreating the conditional in the `whatIs` function that we had earlier in the "test" code (logging output):

```
class Person{
  constructor(knowsBinary){
    this.knowsBinary = knowsBinary;
  };
  log(number){ console.log(this.whatIs(number)) };
  whatIs(number){
```

```
    if(this.knowsBinary){
      return Number('0b' + number);
    } else{
      return number;
    }
  };
};

const personOne = new Person(true);
const personTwo = new Person(false);

[personOne, personTwo].forEach(person => { person.log(10) });
```

We could eliminate this type check by adding functions to our `personOne` and `personTwo` objects after they are created:

```
class Person{
  log(number){ console.log(this.whatIs(number)) };
};

const personOne = new Person( );
personOne.whatIs = (number) => Number('0b' + number);
const personTwo = new Person(number => number);
personTwo.whatIs = (number) => number;

[personOne, personTwo].forEach(person => { person.log(10) });
```

Alternatively, we can eliminate the type check by supplementing the object with a function on construction:

```
class Person{
  constructor(whatIs){ this.whatIs = whatIs }
  log(number){console.log(this.whatIs(number)) }
};

const personOne = new Person(number => Number('0b' + number));
const personTwo = new Person(number => number);

[personOne, personTwo].forEach(person => { person.log(10) });
```

If we wanted, we could name and extract these functions to simplify the constructor:

```
function binaryAware(number){
  return Number('0b' + number);
};
function binaryOblivious(number){
  return number;
```

```
};

const personOne = new Person(binaryAware);
const personTwo = new Person(binaryOblivious);
```

Now that the functions are extracted, we can easily test them independently and without creating any `Person` objects.

You might be tempted to move these functions into the `Person` object, possibly as static functions like this:

```
class Person{
  constructor(whatIs){
    this.whatIs = whatIs;
  };
  log(number){
    console.log(this.whatIs(number));
  };
  static binaryAware(number){
    return Number('0b' + number);
  };
  static binaryOblivious(number){
    return number;
  };
};

const personOne = new Person(Person.binaryAware);
const personTwo = new Person(Person.binaryOblivious);
```

However, this is a level of coupling between the person ("context") and binary awareness ("strategy") that we're trying to avoid. *Binary awareness* is potentially a strategy we'd like to give to other objects (e.g., dolphins, androids, and aliens) later, so it's best to keep them separate.

Instead of that coupling, we can create a new object that contains the strategies, making the full code listing the following:

```
class Person{
  constructor(whatIs){ this.whatIs = whatIs };
  log(number){ console.log(this.whatIs(number)) };
};

const binary = {
  aware(number){ return Number('0b' + number) },
  oblivious(number){ return number }
};

const personOne = new Person(binary.aware);
const personTwo = new Person(binary.oblivious);
```

```
[personOne, personTwo].forEach(person => { person.log(10) });
```

Now our strategies are neatly tucked away in an object, which we pass to our `new Person` constructor. This is convenient, because if we create a new function representing a different interpretation of a number (say, octal or hexadecimal), it won't require a whole new subclass.

You might notice that our "functional variant" from the template method section applies here as well. If we're willing to drop objects and classes, we have a different (and usually shorter) type of code.

State

The state pattern is a bit more involved than the strategy pattern, but can flow naturally from it. Let's assume that "awareness of binary" actually indicates knowledge of the binary operations `read`, `and`, and `xor` (exclusive or) as well. (There are other binary operations, but three are sufficient to demonstrate the pattern.) To continue to use the strategy pattern for this case, we would need to expand our constructor to include all of the new knowledge strategies:

```
class Person{
  constructor(readKnowledge, andKnowledge, xorKnowledge){
    this.read = readKnowledge;
    this.and = andKnowledge;
    this.xor = xorKnowledge;
  };
};

const binary = {
  readAware(number){
    return Number('0b' + number);
  },
  readOblivious(number){
    return number;
  },
  andAware(numberOne, numberTwo){
    return numberOne & numberTwo;
  },
  andOblivious(numberOne, numberTwo){
   return "unknown";
  },
  xorAware(numberOne, numberTwo){
    return numberOne ^ numberTwo;
  },
  xorOblivious(numberOne, numberTwo){
    return "unknown";
```

```
  }
};

const personOne = new Person(binary.readAware,
                             binary.andAware,
                             binary.xorAware);
const personTwo = new Person(binary.readOblivious,
                             binary.andOblivious,
                             binary.xorOblivious);

[personOne, personTwo].forEach(person => {
  console.log(person.read(10));
  console.log(person.and(2, 3));
  console.log(person.xor(2, 3));
});
```

If we run this with **`node state.js`** (assuming we saved the file as *state.js*), we'll see the following output:

```
2
2
1
10
unknown
unknown
```

This expansion of the strategy pattern can get messy quickly. The easiest remedy to that is to allow two objects to contain all of the "obliviousness" or "awareness" of binary:

```
class Person{
  constructor(binaryKnowledge){
    this.binaryKnowledge = binaryKnowledge;
  }
};

const binaryAwareness = {
  read(number){
    return Number('0b' + number);
  },
  and(numberOne, numberTwo){
    return numberOne & numberTwo;
  },
  xor(numberOne, numberTwo){
    return numberOne ^ numberTwo;
  }
};
```

```
const binaryObliviousness = {
  read(number){
    return number;
  },
  and(numberOne, numberTwo){
    return "unknown";
  },
  xor(number){
    return "unknown";
  }
};

const personOne = new Person(binaryAwareness);
const personTwo = new Person(binaryObliviousness);

[personOne, personTwo].forEach(person => {
  console.log(person.binaryKnowledge.read(10));
  console.log(person.binaryKnowledge.and(2, 3));
  console.log(person.binaryKnowledge.xor(2, 3));
});
```

This simplifies the constructor calls and allows the `binaryKnowledge` to be independent of the `person`. Before we move on to fully implementing the state pattern, there are two additional possibilities worth consideration. The first is a potential shortcoming of all three functions being contained in the same knowledge objects. Is it possible for a person to know how to `read` binary, but not execute an `xor` operation? Sure, why not? This design doesn't offer a flexible approach to that situation, but we'll address that after completing the state pattern.

A second thing to consider is whether or not to add delegation from the `person` object directly to the `binaryKnowledge` functions, allowing for an interface like `person.read` or `person.xor`. There is a possible convenience in that, but it also has weaknesses. Tying those functions together in the constructor (using statements like `this.read = binaryKnowledge.read`) would mean any new functions added to the `binaryKnowledge` objects would also need new references in the `Person` constructor. Additionally, the client code then masks the interfaces of the `binaryKnowledge` objects. Whether to delegate or not is going to come down to a matter of taste, but in general, the more dynamic and complex the delegate objects (`binaryKnowledge` in this case), the more overhead and confusion you will have when delegating individual functions.

THE BINARY OBJECT'S FUNCTIONS CAN BE CHANGED!

Although it's convenient to attach objects directly to other objects like this, it is worth considering that although `const` will protect the value of `binary` from being reassigned, it does not prevent our `aware` or `oblivious` functions from being reassigned. Furthermore, if multiple `person` objects may reference the same function, redefining a function on a given `person` object will redefine the function for other objects as well. Adding the following code to the end should demonstrate this:

```
const personOne = new Person(binaryAwareness);
const personTwo = new Person(binaryAwareness);
personTwo.binaryKnowledge.read = () => `
redefined on both objects`;

[personOne, personTwo].forEach(person => {
  console.log(person.binaryKnowledge.read(10));
});
```

The easiest path to having *immutable* objects is to have *new* objects. We'll address this toward the end of this section.

What makes this not quite yet the state pattern is that we haven't defined the transitions between our objects. Let's do that now:

```
class Person{
  constructor(binaryKnowledge){
    this.binaryKnowledge = binaryKnowledge;
  };
  change(binaryKnowledge){
    this.binaryKnowledge = binaryKnowledge;
  };
};

const binaryAwareness = {
  read(number){
    return Number('0b' + number);
  },
  and(numberOne, numberTwo){
    return numberOne & numberTwo;
  },
  xor(numberOne, numberTwo){
    return numberOne ^ numberTwo;
  },
  forget(person){
    person.change(binaryObliviousness);
  }
}
const binaryObliviousness = {
  read(number){
```

```
    return number;
  },
  and(numberOne, numberTwo){
    return "unknown";
  },
  xor(number){
    return "unknown";
  },
  learn(person){
    person.change(binaryAwareness);
  }
};

const personOne = new Person(binaryAwareness);
const personTwo = new Person(binaryObliviousness);

[personOne, personTwo].forEach(person => {
  console.log(person.binaryKnowledge.read(10));
  console.log(person.binaryKnowledge.and(2, 3));
  console.log(person.binaryKnowledge.xor(2, 3));
});

personOne.binaryKnowledge.forget(personOne);
personTwo.binaryKnowledge.learn(personTwo);

[personOne, personTwo].forEach(person => {
  console.log(person.binaryKnowledge.read(10));
  console.log(person.binaryKnowledge.and(2, 3));
  console.log(person.binaryKnowledge.xor(2, 3));
});
```

This requires one new function (`change`) to be defined on `Person`, as well as a `forget` function defined on `binaryAwareness` and a `learn` function defined on `binaryObliviousness`. For our tests (just logging statements) we flip the binary knowledge states of our two `person` objects and should see the output reversed when it prints the second time.

There are two things about this code that are suspect. First, it is awkward to pass a `person` object into the `forget` and `learn` functions. Second, as the warning text previously indicated, our `binaryKnowledge` objects are not safe from being redefined.

We could address the second problem by making factory functions, constructor functions, or classes that would return our `binaryKnowledge` objects. However, a simpler solution is just to wrap our assignment to `binaryKnowledge` in the `Person` constructor with an `Object.create`. To make our `change` function safe, we can wrap it as well:

```
class Person{
  constructor(binaryKnowledge){
    this.binaryKnowledge = Object.create(binaryKnowledge);
  };
  change(binaryKnowledge){
    this.binaryKnowledge = Object.create(binaryKnowledge);
  };
};
```

To solve the first problem (the awkwardness of duplicating our references to `person` in the `learn` and `forget` functions), we can use `Object.assign` in the `Person` constructor to establish a two-way link between `binaryKnowledge` and the `person` object. Then we can remove the parameter from both `forget` and `learn`, and change the references to that parameter to instead use `this.person`:

```
class Person{
  constructor(binaryKnowledge){
    this.binaryKnowledge = Object.create(
                             Object.assign(
                               {person: this},
                                binaryKnowledge));
  };
  change(binaryKnowledge){
    this.binaryKnowledge = Object.create(
                             Object.assign(
                               {person: this},
                                binaryKnowledge));
  };
};

const binaryAwareness = {
...
  forget(){
    this.person.change(binaryObliviousness);
  }
}
const binaryObliviousness = {
...
  learn(){
    this.person.change(binaryAwareness);
  }
};

const personOne = new Person(binaryAwareness);
const personTwo = new Person(binaryObliviousness);

...
```

```
personTwo.binaryKnowledge.forget();
personTwo.binaryKnowledge.read = () => 'will not assign both';
[personOne, personTwo].forEach(person => {
  console.log(person.binaryKnowledge.read(3));
});
```

The last five lines of this code snippet should confirm that the new reference to `read` stays within its own object and `forget` (as well as `learn`) no longer requires an explicit argument for `person`.

Alternatively, we could define both `forget` and `learn` on the `Person` class (the original design pattern does not specify where transitions are defined), but the issue raised earlier about delegating functions to `Person` also applies now: if we add information about states to `Person`, then `Person` becomes more complex and less focused. That said, it may be worth it in some cases in order to centralize transition information.

STATE PATTERN VERSUS STATE MACHINE

There is a bit of crossover between the state *pattern* and the state *machine*. Although the ideas are related, state machines tend to focus on the states themselves and transitions between them, rather than providing a unified interface for common functions. Although you gain some simplicity by virtue of having just one class/object controlling all of your states and transitions (and typically callback functions to execute during transitions as well), you *lose* the simplicity of a unified interface while in any given state. That means more type checking (`if` statements), and is a likely place for complexity to accumulate over time.

And now, some unfortunate news about the state pattern.

Here, we had a fairly simple example of the pattern, with only two states. Although it scales fairly well, refactoring to use the state pattern just for the sake of removing conditional checks is fairly aggressive once a codebase has headed down the path of lots and lots of type checking (whether with or without a state machine in place). And adding a lot of new objects/classes (one for each state) might not be a popular idea among your teammates.

You can also encounter problems if your states are complex. Picture a branching structure where "knowing `xor`" is a substate of "knowing binary," which is a substate of "knowing math," which is a substate of "understanding the universe." If your states branch out like that, you'll be overwhelmed with

classes/objects very quickly. Put another way, it's possible that your states may have nothing to do with one another. Do you make a state for every combination of "knowing binary," "knowing English," and "having a pet"? Do you introduce another full-blown state pattern implementation for each independent type of state? Hopefully, the answer is no, and simplicity wins over competing types of states for one type of object.

One final thing that the classical state pattern comes with is some contract that a certain *interface* is *implemented* by each state class/object. We have made no such guarantee in our usage. In JavaScript, without compile-time checks for such contracts, we have a couple of options. We could use assertions inside of constructors for our states or our context (`Person`). We could create a `BaseState` object that contains stubs of functions (and possibly throws an error if they are not overwritten). Or we could do nothing to enforce that type of contract. Given our options, this isn't that bad. We'll get `TypeError: someObject.whatever is not a function` errors, which is probably as descriptive as any error we would throw in the `BaseState` or an `AssertionError` we would get from failed assertions in the constructors. Another option is using a language that compiles to JavaScript and has these features (like TypeScript). Hopefully, the appeals to simplicity in this chapter make it clear that it is not advisable to rewrite your entire program in a different language and complicate your build process for the sake of fully realizing a design pattern.

THE POWER OF HAS-A

At the core of the state pattern is a willingness to move part of what may be thought of as "belonging to" an object into another. There is a tendency in OOP to prioritize hierarchies over delegate objects. The state pattern is a relatively complex version of delegation, but simpler delegate objects should not be overlooked.

---

null Object

The `null` object pattern, unlike the others discussed here, was not one of the original 23 patterns defined in the Gang of Four's book *Design Patterns: Elements of Reusable Object-Oriented Software*—and is probably the most underused of all. Consider how many conditional checks in your code are executed against `null`, `undefined`, or the existence of a particular variable or function. Odds are, that type of checking makes up a large portion of your `if` statements. And worse, the absence of those type checks likely accounts for a large portion of your errors.

A (NON-)VALUE NOT EVEN A FATHER COULD LOVE

Tony Hoare invented the `null` reference in 1965. In 2009, he called it his "billion dollar mistake" because of all the damage it has caused. Considering not only run-time errors, but all of the extra time spent coding type checks for `null` values across every language that has implemented it, a billion dollars is probably an understatement.

In fact, according to **research** (http://insight.jbs.cam.ac.uk/2013/research-by-cambridge-mbas-for-tech-firm-undo-finds-software-bugs-cost-the-industry-316-billion-a-year/) from Cambridge University in 2013 (admittedly sponsored by a bug-hunting software vendor), bugs cost the world $316 billion dollars every year. Surely, not all of those derive from improper handling of `null` conditions, but from my personal experience with web applications, I would guess that it accounts for a significant portion. If I was hard-pressed for a percentage, I'd estimate somewhere around half, assuming no static typing tools are in place. It's certainly more than 1 in 316, and we're talking about *annually* here.

Hoare's career is a distinguished one, so this is not brought up to criticize him. The point is that "`null` is a billion-dollar mistake" is not common enough knowledge for people to avoid using it. It (and `undefined`) didn't make Crockford's list of "bad parts" (Appendix B in *JavaScript: The Good Parts*), and it is returned by so many APIs in so many libraries.

Let's start with something that might return `null`:

```
class Person {
  constructor(name){
    this.name = name;
  }
};
class AnonymousPerson extends Person {
  constructor(){
    super();
    this.name = null;
  }
};

personOne = new Person("tony");
personTwo = new AnonymousPerson("tony");
console.log(personOne.name);
console.log(personTwo.name);
```

One unfortunate distraction in this code is the call to `super` in the constructor of the subclass. If you don't call `super`, you'll get a `ReferenceError: this is not defined`, which not only is an unclear error, but also demonstrates how `null` or `undefined` can come from core parts of the language, not just libraries.

Regardless, the output doesn't look so bad:

```
tony
null
```

But let's say we want to do something with those values:

```
function capitalize(string) {
  return string[0].toUpperCase() + string.substring(1);
};

console.log(capitalize(personOne.name));
console.log(capitalize(personTwo.name));
```

Now our trouble starts with:

```
TypeError: Cannot read property '0' of null
```

We might be inclined to try to solve this by using an empty string instead of `null`:

```
class AnonymousPerson extends Person {
  constructor(){
    super();
    this.name = "";
  }
};
```

This time, we'll get a new error:

```
TypeError: Cannot read property 'toUpperCase' of undefined
```

It's just as bad as the first one, so we might as well put our `null` back. Now we're stuck doing some type checking inside of `capitalize`:

```
function capitalize(string) {
  if(string === null){
    return null;
  }else{
    return string[0].toUpperCase() + string.substring(1);
```

```
  }
};
```

Because we didn't have a good way out of the empty value we got from the `AnonymousPerson` `name` property (we're pretending we don't know about `null` objects), we're stuck with some kind of nonsensical type check like this (it could just as easily be `""` or `undefined`). And the consequence for *not* type checking is that our program will throw an error at runtime.

So what do we do if the string is `null`? We might as well return another `null`. Now say we want to *tigerify* our `tonys`:

```
function tigerify(string) {
  return `${string}, the tiger`;
};
console.log(tigerify(capitalize(personOne.name)));
console.log(tigerify(capitalize(personTwo.name)));
```

Now our output is:

```
Tony, the tiger
null, the tiger
```

Well, it's not an error, but that's not a value we want to display. Let's try this:

```
function tigerify(string) {
  if(string === null){
    return null;
  }else{
    return `${string}, the tiger`;
  }
};
```

Now we'll get just plain `null` for the string, which isn't really a better value to display than before. We don't *really* want to make our `tigerify` function responsible for the final output (it's nice that `capitalize` and `tigerify` have the same interface, as we can call both in either order, or one without the other), so let's create a `display` function to handle the finalized output:

```
function display(string){
  if(string === null){
    return '';
  }else{
    return string;
  }
};
```

Then we run another `null` check so that we can display a blank string.

BUT NO ONE WOULD ACTUALLY DISPLAY NULL, RIGHT?

Yeah, they would. I'm not going to say what website I saw Figure 9–1 on, but they have over $40 million in funding. Also, they created the `null` here, not me.

FIGURE 9-1

A display of null in the wild

Hopefully, the antipattern is becoming clear: either check for `null`, or subject people to errors and other awkward interactions. It takes one `null` to kick off a codebase full of conditional checks and sadness. Not only does every function that calls something that could return a `null` value likely *need* a `null` check, but your tests are necessarily more complex as well, because any functions you test will have at least two branches.

So how do we fix it? The answer is we start treating our names like objects, rather than strings and `null`s. Then we can begin to implement functions that have a mirror interface:

```
class Person {
  constructor(name){
    this.name = new NameString(name);
  }
};
class AnonymousPerson extends Person {
  constructor(){
    super();
    this.name = new NullString;
  }
};

class NullString{
  capitalize(){
    return null;
  }
```

```
};

class NameString extends String{
  capitalize() {
    return new NameString(this[0].toUpperCase() + this.substring(1));
  };
  tigerify() {
    if(this === null){
      return null;
    }else{
      return new NameString(`${this}, the tiger`);
    }
  };
  display(){
    if(this === null){
      return '';
    }else{
      return this.toString();
    }
  };
}

personOne = new Person("tony");
personTwo = new AnonymousPerson("tony");
console.log(personOne.name.capitalize().tigerify().display());
console.log(personTwo.name.capitalize());
```

The biggest change here is the addition of two new classes: `NameString` and `NullString`. The `Person` and `AnonymousPerson` constructors create these objects. Notice that we're now chaining functions instead of nesting them, and that `display` now calls `toString` so that `console` doesn't print the type information. `NameString` implements our functions as before (except for the type check we could remove from `capitalize`, but `NullString` only has `capitalize` implemented right now).

Why is that?

Because the traditional way to implement the `null` object pattern is to return `null`. We hit the `null` wall and can't do anything else with `personTwo` without type checking. It's worth stopping there for a minute to think about how unfortunate that is.

Now let's do something better. We'll implement a `NullString` that not only has mirror functions, but also returns mirror values to the `NameString`.

WHY DIDN'T WE JUST OVERWRITE STRING.PROTOTYPE?

Overwriting base objects is considered a likely bad idea unless you're sure that the changes are desirable for some scope that's easy to contain. Most of the time, other programmers (also you, after you've forgotten that you overwrote parts of `String`) will assume that it is the same `String` that they're used to. Although subclassing is usually worse than other options for extracting an object, it is very useful for creating copies of native objects like `String` and `Array`.

Our code looks like this with mirror interfaces:

```
// with null object
class Person {
  constructor(name){
    this.name = new NameString(name);
  }
};

class AnonymousPerson extends Person {
  constructor(){
    super();
    this.name = new NullString;
  }
};

class NullString{
  capitalize(){
    return this; // same as new NullString in this case
  };
  tigerify() {
    return this; // same as new NullString in this case
  };
  display() {
    return '';
  };
};

class NameString extends String{
  capitalize() {
    return new NameString(this[0].toUpperCase() + this.substring(1));
  };
  tigerify() {
    return new NameString(`${this}, the tiger`);
  };
  display(){
    return this.toString();
  };
}
```

```
personOne = new Person("tony");
personTwo = new AnonymousPerson("tony");
console.log(personOne.name.capitalize().tigerify().display());
console.log(personTwo.name.capitalize().tigerify().display());
```

Some people claim that this is more complicated than the equivalent version. Let's look at the other for reference:

```
// without null object
class Person {
  constructor(name){
    this.name = name;
  };
};
class AnonymousPerson extends Person {
  constructor(){
    super();
    this.name = null;
  };
};

function capitalize(string) {
  if(string === null){
    return null;
  }else{
    return string[0].toUpperCase() + string.substring(1);
  }
};

function tigerify(string) {
  if(string === null){
    return null;
  }else{
    return `${string}, the tiger`;
  }
};

function display(string){
  if(string === null){
    return '';
  }else{
    return string;
  }
};

personOne = new Person("tony");
personTwo = new AnonymousPerson("tony");
```

```
console.log(display(tigerify(capitalize(personOne.name))));
console.log(display(tigerify(capitalize(personTwo.name))));
```

It's actually about the same number of lines. Is it better to have a conditional in every function or to have two functions in different classes? Now that we see them side by side, let's talk about the pros and cons of using the `null` object pattern:

Pros:

- You can use `console.log` and actually find out something about your values.
- You can do anything you want in the function body.
- You can have the `null` object inherit and override functions (from `NameString`).
- It's very easy to forget to do a `null` check. Having a complete API to duplicate may seem more straightforward to you.

Cons:

- You might have the classes in two separate files by convention, creating overhead in searching the project.
- You might buck the convention of separating them by having them in the same file. This also creates overhead in searching the project.
- You might have teammates who don't like the idea of `null` objects, don't understand the idea of `null` objects, or just don't care about `null` objects (this is the case with pretty much every pattern or quality concern in general).
- Since `null` and `undefined` are so commonly returned, your codebase is unlikely to be completely `null`/`undefined` free. That means you will probably end up with inconsistencies and some confusion from people unfamiliar with the patterns you implement.
- Because returning `null`/`undefined` is so common, if you want consistent expectations for not returning `null`/`undefined`, you'll likely need to adapt a lot of your codebase as well as third-party APIs.
- You have to implement a function on the `null` object every time you create a new function call that might call the `null` object instead of its mirrored real one. Since it's likely that you'd have to add and might forget a `null` check (if you're not using `null` objects), I think this one is a wash.
- You can't insist that third-party code follow sane practices for returning real values. If you're stuck with a library that returns the billion-dollar mistake, you'll need to wrap that interface with a new functionality, which could be confusing to people used to working with the basic interface.

- You might want different `null` object functions depending on the eventual context of how the value will be used. Will it be saved to the database? Logged? Used to create a new value? These all suggest different terminal functionality (such as our `display` function).
- When working with native object types such as strings and arrays, you'll face overhead in extending them to build subclasses to mirror the `null` objects.

Overall, in *application* development, the `null` object pattern provides an alternative to `null` checks (and the errors caused when you forget to check for `null`). If you're writing a *library, framework, or module*, however, it is worth considering this pattern in order to avoid "`null`-poisoning" the codebases of anyone who uses your code.

If you're trying to avoid all of the subclasses involved in this pattern, you can try combining it with the techniques presented in the sections for the state and strategy patterns. Also, in **Chapter 11**, we'll look at a functional alternative to the `null` object pattern called `Maybe`. As for the next section, we'll double down on the `null` object pattern by combining it with a wrapper.

Wrapper (Decorator and Adapter)

There are a few ways to implement the decorator pattern. One thing that should not be surprising based on earlier sections in this chapter is that JavaScript does not naturally fit in well with interfaces, abstract classes, and other object-oriented language features that tend to be a part of the classical pattern.

Before moving on, install tape with **`npm install tape`**. It's a more lightweight test framework than mocha. We'll use it again in **Chapter 10**.

Most of the time for the decorator pattern, you'll see an example like this:

```
class Dog{
  constructor(){
    this.cost = 50;
  }
  displayPrice(){
    return `The dog costs $${this.cost}.`;
  }
};

const test = require('tape');
test("base dog price", (assert) => {
  assert.equal((new Dog).displayPrice(), 'The dog costs $50.');
  assert.end();
});
```

THIS IS NOT THE SAME ASSERT AS BEFORE

tape has a special callback parameter that has an assertion syntax built in. It's also responsible for ending the test and counting the test cases. If we use a different assertion library in tape, we have to add an extra `assert.pass()` line, which is awkward:

```
const test = require('tape');
const wish = require('wish');
test("base dog price", (assert) => {
  wish((new Dog).displayPrice()
=== 'The dog costs $50.');
  assert.pass();
  assert.end();
});
```

If we don't add that line, the tests won't *fail*, but tape won't count them as passing, or even as tests(!), in its output. We'll stick with tape's `assert` in this chapter.

The decorator is useful in a couple of cases. First, it's good if we're pulling the dog class in from a module we don't own and don't want to (or can't) manipulate it directly. Second, it's good to keep us from a sprawling mass of subclasses. To illustrate that second point, consider if our price was based on whether the dog was cute, trained, robotic, friendly, or a show dog. All of those traits could affect the price, but they are not mutually exclusive, so comprehensive subclassing would have to collect every attribute (e.g., `class FriendlyNotCuteTrainedNonRoboticNonShowDog extends Dog`).

To decorate the dog with a particular trait, we can add a factory function that takes a dog as input:

```
function Cute(dog){
  const cuteDog = Object.create(dog);
  cuteDog.cost = dog.cost + 20;
  return cuteDog;
};

test("cute dog price", (assert) => {
  assert.equal((Cute(new Dog)).displayPrice(), 'The dog costs $70.');
  assert.end();
});
```

Adding another trait is as simple as adding a new factory function:

```
function Trained(dog){
  const trainedDog = Object.create(dog);
  trainedDog.cost = dog.cost + 60;
```

```
  return trainedDog;
};
test("trained/cute dog price", (assert) => {
  assert.equal(Trained(Cute(new Dog)).displayPrice(),
               'The dog costs $130.');
  assert.end();
});
```

We can add new decorators like this quite easily.

DO WE HAVE TO NEST THE FACTORY FUNCTIONS?

You might be wondering if we could get away with an interface like this instead:

```
(new Dog).Cute().Trained();
```

The answer is "yes, we absolutely *could* do that." However, can we do that without adding new functionality to the original class? No. One major point of this pattern is that we either *can't* or *don't want to* alter the original class.

The "cost of something" decorator focuses on adding traits to an existing interface. In other cases, we want to adapt an interface by applying another wrapper. Let's try addressing the problem of an API that returns `null`s, "billion-dollar mistakes" that they are (see the previous section if you need some background on `null` objects). Here is some code similar to what we looked at in the previous section (before we made the `null` objects):

```
class Person {
  constructor(name){
    this.name = new NameString(name);
  }
};

class AnonymousPerson extends Person {
  constructor(){
    super();
    this.name = null;
  }
};

class NameString extends String{
  capitalize() {
    return new NameString(this[0].toUpperCase() + this.substring(1));
  };
  tigerify() {
    return new NameString(`${this}, the tiger`);
```

```
  };
  display(){
    return this.toString();
  };
};

const test = require('tape');

test("Displaying a person", (assert) => {
  const personOne = new Person("tony");
  assert.equal(personOne.name.capitalize().tigerify().display(),
               'Tony, the tiger');
  assert.end();
});
```

This works fine, but if we want to test `AnonymousPerson` objects, we're stuck handling the `null` checks in our test. To be explicit, add this test:

```
test("Displaying an anonymous person", (assert) => {
  const personTwo = new AnonymousPerson("tony");
  assert.equal(personTwo.name.capitalize().tigerify().display(),
               '');
  assert.end();
});
```

We'll quickly hit this error:

```
TypeError: Cannot read property 'capitalize' of null
```

In our `null` object example from the last section, we dealt with this by allowing `AnonymousPerson` to return a `NullString` for `name` and then added some functions to `NullString`. It would be convenient if we could add the following code (changing `AnonymousPerson` and adding `NullString`):

```
class AnonymousPerson extends Person {
  constructor(){
    super();
    this.name = new NullString; // this line is the problem
  }
};
class NullString{
  capitalize(){
    return this;
  };
  tigerify() {
    return this;
  };
```

```
  display() {
    return '';
  };
};
```

Let's say that we can't touch `AnonymousPerson`, but we have no problem adding the `NullString` class for the sake of applying this decorator pattern.

With what we need to add, our code becomes the following:

```
class Person {
  constructor(name){
    this.name = new NameString(name);
  }
};

class AnonymousPerson extends Person {
  constructor(){
    super();
    this.name = null;
  }
};

class NameString extends String{
  capitalize() {
    return new NameString(this[0].toUpperCase() + this.substring(1));
  };
  tigerify() {
    return new NameString(`${this}, the tiger`);
  };
  display(){
    return this.toString();
  };
};

// this is new
class NullString{
  capitalize(){
    return this;
  };
  tigerify() {
    return this;
  };
  display() {
    return '';
  };
};

// this is new
function WithoutNull(person){
```

```
  personWithoutNull = Object.create(person);
  if(personWithoutNull.name === null){
    personWithoutNull.name = new NullString;
  };
  return personWithoutNull;
};

const test = require('tape');

test("Displaying a person", (assert) => {
  const personOne = new Person("tony");
  assert.equal(personOne.name.capitalize().tigerify().display(),
               'Tony, the tiger');
  assert.end();
});

test("Displaying an anonymous person", (assert) => {
  const personTwo = new AnonymousPerson("tony");
// wrapping with WithoutNull is new
  assert.equal(WithoutNull(personTwo)
               .name.capitalize().tigerify().display(),
               '');
  assert.end();
});
```

And both tests pass. We just had to add the `WithoutNull` function and use it to wrap `personTwo`. All it does is prevent `name` from being `null`. Because it does not convert the `name` *unless* it is `null`, we can happily wrap *any* `person` object with `WithoutNull`, anonymous or not. Notice that we use a simple factory function that returns an object, but we could have used a class or constructor function instead. The disadvantage of those approaches is that they implicitly return an object, and here, it's easier to explicitly return a new one that we construct manually. With a class or constructor function, the implicitly returned class would be of type `WithoutNull`, which is odd, and overriding it (returning a different object in the constructor function of the class or from a constructor function called with `new`) reads as a bit confusing.

Two other things are worth considering here. First, it won't always be the case that we can decide what type of `person` object we have. It's possible that we'd only have one type of `Person` and we would not be able to determine ahead of time whether or not it had a `name` (or any other attributes that we might wrap with `WithoutNull`). For example, we might be receiving objects from the database with a function like `(db.get({person: {id: 17}}))`. Our `WithoutNull` wrapper would be excellent for providing a consistent interface to `name` and other attributes just by wrapping the call like this: `without-`

`Null(db.get({person: {id: 17}}))`. It's very likely that we would not want to change the database API itself, but rather wrap values that it returns.

Second, if our API returned a string instead of a real object for `name` in the normal `Person` case (rather than a `NameString`), we would have had much more difficulty in setting up a parallel function structure for `NullString` without directly extending our "untouchable" `Person` and `AnonymousPerson` classes, overwriting `String.prototype`, or compromising our fairly simple wrapping API.

We've seen two possible interfaces so far:

```
Cute(dog);
WithoutNull(personTwo);
```

As we noted earlier, we could have used a class/constructor function instead of a factory function, which would give us an interface like:

```
new Cute(dog);
new WithoutNull(personTwo);
```

The reason we didn't do that is that it is odd to explicitly return something from a constructor function (which carries the expectation of returning the type of object named by the constructor or class).

From the perspective of an interface, what is a `new Cute`? What is a `new WithoutNull`? These are traits, and don't really make sense to instantiate. So it's not only the implementation, but also the interface, that suggests a factory function.

For another take on this, let's look at a very simple adapter that uses a class:

```
class Target{
  hello(){
    console.log('hello');
  };
  goodbye(){
    console.log('goodbye');
  };

};
class Adaptee{
  hi(){
    console.log('hi');
  };
  bye(){
    console.log('bye');
  };
};
```

```
const formal = new Target;
formal.hello();
formal.goodbye();

const casual = new Adaptee;
casual.hi();
casual.bye();

class Adapter{
  constructor(adaptee){
    this.hello = adaptee.hi;
    this.goodbye = adaptee.bye;
  };
};
const adaptedCasual = new Adapter(new Adaptee);
adaptedCasual.hello();
adaptedCasual.goodbye();
```

So we have `casual` and `formal` objects with different interfaces (`hi` versus `hello`, and `bye` versus `goodbye`). If we need to support the same interface, `Adapter` can remap the functions to new ones. This is the same problem that `WithNull` helped us solve earlier.

For the same reasons as with the decorators, `Adapter` would probably be best as a factory function. But if we use a factory function for the adapter, what is the difference between an adapter and a decorator?

THE "DECORATOR" TC39 PROPOSAL

There are, as of this writing, a few proposals for JavaScript features that use the term *decorator*. Although they are distinct from our discussion here, they involve similar ideas of altering existing behavior.

The difference is one of emphasis and likely usage. A decorator is more likely to have nested wrappers to add a few disparate traits. An adapter is more of a mapping of interfaces between one object and another. Imagine our `WithNull` implementing a conversion of every property to not allow for `null`. In a decorator scenario, we might prefer to nest as in `WithoutNullName(WithoutNullPhone(person))`. For an adapter, we would be more likely to apply the interface transformation as a necessary and single step. So the `dog` example was probably more of a decorator, while the "without `null`" example was more of an adapter. But does it matter? Not really. In all cases, we're using wrapper functions to add properties to objects. It's tempting to think of patterns in terms of "interfaces" (the OOP term) and "abstract classes," subclasses and private/public methods and members, but JavaScript isn't Java. UML diagrams to

describe these complex relationships can be a useful starting point for design, but for JavaScript, your best option is to start with the interface (the test/client code) you want to write and then figure out how to implement it.

When should we avoid decorators and adapters? Well, if we have control over the entire implementation (i.e., we own our own libraries), we might get the same utility from altering the base code rather than complicating our interfaces with wrapping possibilities. Additionally, as with applying any pattern, ensure that the result is actually simpler and more understandable than the original (unpatterned) code. Before using the adapter function, first try extracting objects and functions. If you still want to apply the pattern, this will probably just make the process easier.

Facade

After the more challenging wrapper patterns, the facade is extremely simple. We have some complex API, but instead of interacting with it directly, we use an interface. When you play a piano, you're not physically pushing the hammers into the strings. When you drive a car, you're probably focused on the destination or next high-level step, occasionally thinking about your controls, and rarely (if ever) thinking about the internal components.

Basically, a facade is an interface that contains a curated subset from one or more APIs with the intention of streamlining and simplifying the code one needs to write.

As for a more specific software example, let's consider how JavaScript interacts with native web APIs. The number of properties available on `document` and `window` alone is huge, and it can hardly be recommended that a beginner just read the documentation in order to get a handle on how they work.

Let's create a facade that demonstrates the kinds of things you can do with JavaScript. Since we'll be interacting with a browser, we can start with an HTML page and save it as *facade.html*:

```
<html>
  <head>
    <meta http-equiv="content-type" content="text/html;
     charset=utf-8" />
    <title></title>
    <script type="text/javascript" src='facade.js'></script>
  </head>
  <body>
  </body>
</html>
```

In the same directory, we can add a file for *facade.js*:

```
const page = {
  say(string){
    console.log(string);
  },
  yell(string){
    alert(string);
  },
  addNewLine(){
    document.body.appendChild(document.createElement("br"));
  },

  addButton(text){
    const button = document.createElement("button");
    button.appendChild(document.createTextNode(text));
    document.body.appendChild(button);
  },
  addText(text){
    const span = document.createElement("span");
    span.appendChild(document.createTextNode(text));
    document.body.appendChild(span);
  },
  changeBackground(color){
    document.body.style.background = color;
  },
  now(asNumber = false){
    if(asNumber === false){
      return new Date().toLocaleTimeString();
    }else{
      return new Date().getTime();
    }
  },
  timeOnPage(){
    return ((this.now(true) - this._start) / 1000) + " seconds";
  },
  loadTime(){
    return ((this._start - this._loaded) / 1000) + " seconds";
  },
  eventsSoFar(){
    console.info(this._events);
  },
  _events: [],
  _start: 'nothing yet',
  _loaded: 'nothing yet'
};

window.onload = function(){
  page._start = page.now(true);
  page._loaded = performance.timing.navigationStart;
  document.onclick = function(event) {
    page._events.push(event.target + " clicked at " + page.now());
```

```
  };
};
```

In this file, we're making the following functions available:

- `say` (`console.log`)
- `yell` (alert)
- `addNewLine` (adds `<br>` tag)
- `addButton` (adds a button)
- `addText` (adds a paragraph)
- `changeBackground` (changes background to a color)
- `now` (prints the current time, nicely formatted with default parameter)
- `timeOnPage` (how many seconds someone has been on the page)
- `loadTime` (how long it took the page to load)
- `eventsSoFar` (what the visitor has clicked on)

If you want to try it out, open *facade.html* in a browser. We have black box functions on an object called `page`. All it does is put together a limited subset of how you might interact with a website, including the basics of logging, analytics, page interactions, and performance monitoring. This could be useful for someone unfamiliar with browser APIs or the browser console.

When should you not use a facade? Any time the direct interactions with an API are simple enough or understood well enough by you or other people interacting with it. In those cases, adding a facade is a bad idea, because it either will not be used, or will be used *sometimes*, leading to learning/supporting/understanding/maintaining two disparate interfaces.

Facades are widely used for tasks like simplifying complex APIs. ORMs (object relational mappers) can be useful for simplifying database interactions. And jQuery is arguably a very large facade for frontend JavaScript and all the APIs it interacts with. Once you get to the ORM or framework level, you've got a lot more going on than a simple "facade," but the intention is the same.

It is tempting to think of interfaces strictly in terms of public/private (which in JavaScript could mean prepended with `_`, completely hidden, or a couple of other things). However, exposing a smaller subset of an API is underutilized. Consider that we've already seen one clear example of why we might use a less feature-rich API over a complex one (tape versus mocha). Whether for convenience or learning, exposing a core/popular set of features independently from a large API could make documentation and coding easier on beginners (and "forgetters")—which, considering how massive the JavaScript landscape is, includes practically everyone.

WHAT DOES THE ALPHABET HAVE TO DO WITH APIS?

Even without the creation of a distinct, smaller interface for an API through a facade, people deserve better than documentation organized *only* alphabetically. What are the most useful functions? What are the most popular?

These are not impossible questions to answer. By providing a subset of documentation or allowing functionality to be ranked by something more practical than the alphabet, advice to "RTFM" would be a lot more useful, albeit just as rude.

Wrapping Up

We've left out the majority of the patterns from the seminal design pattern book, *Design Patterns: Elements of Reusable Object-Oriented Software* (also known as the "Gang of Four" or "GoF" book for its four authors), as well as a good number of those that have popped up throughout the years. Since this is a refactoring book and not a patterns book, we had to draw the line somewhere, especially given that the term *pattern* is used as freely and casually as *refactoring* can be. Additionally, our goal is turning bad (and likely to be seen) code into good code. Sometimes, there is simply not a plausible form that code could take as a "before" case. In other instances, applying a pattern is done for the sake of optimization, rather than to achieve a better interface. Another reason some patterns are not included is that they're built into JavaScript itself.

With those thoughts in mind, a few additional patterns worth exploring are:

Composite
: This pattern is good for traversing trees of data, such as JSON/objects/document nodes.

Builder
: Used for complex object creation, this pattern is good for generating test data.

Observer
: This pattern can be seen in things like publish/subscribe, events, and observables.

Prototype
: This pattern is built into JavaScript.

Iterator

Building your own iterator is an unlikely choice given the wealth of native options for iteration (including loops, array comprehension functions, and generators).

Proxy

We actually see an example of this in action in **Chapter 10** when we use the testdouble framework, which allows "faking out" the real function call as well as asserting that the function was called. Note that there is also a native proxy object in JavaScript, and it is another possible approach to wrapping an object with `withoutNull`, as we covered earlier.

The GoF book doesn't have a monopoly on design patterns, nor do object-oriented languages or web applications. In some languages or problem domains, a pattern might be very popular or easy to implement. JavaScript, given that it spans so many paradigms and applications, has a large pool of architectural ideas to draw from. If you are looking for some new patterns to think about, embedded programming, operating systems, databases, concurrency, functional programming, and game development might be of interest.

In this chapter, we looked at some of the best that object-oriented architecture has to offer, through a JavaScript lens. For the sake of refactoring, and coding generally, the advice from the GoF to "program to an interface, not an implementation" rings true. Between (our) JavaScript's lack of a distinct compilation step and some keywords that didn't cross over from Java, SmallTalk, and the like, this is not only practical advice, but the only possible approach for implementing many patterns.

Ultimately, even if perfectly implementing UML diagrams from other languages were possible with JavaScript, the patterns presented in this chapter are intended to address likely architectural problems. Specifically, we addressed maintenance issues that arise with code complexity (exemplified by the proliferation of conditionals and subclasses), learned how to avoid "billion-dollar mistakes" from external code, and provided an example of how to create a small, friendly API.

In the next two chapters, we will be exploring JavaScript's more functional side.

Asynchronous Refactoring 10

In this chapter, we'll discuss asynchronous (aka "async") programming in JavaScript, by covering the following topics:

- Why async?
- Fixing the "pyramid of doom"
- Testing async code
- Promises

Why Async?

Before we get into how to make asynchronous JavaScript better through refactoring, it's worth discussing why we need it. Why shouldn't we just use a "simpler" synchronous style and not worry about async at all?

As a practical concern, we want our programs to be performant. Despite our focus in this book being interfaces rather than performance, there is another issue, even if we thought it was okay to hold up our whole program for a web request or data processing task that could take seconds, minutes, or even longer: sometimes async is the only option for a given module or library.

Async has become the norm for many APIs. For example, one coming from a mostly synchronous paradigm (language or style) might expect node's `http` module to behave like this:

```
const http = require('http');
const response = http.get('http://refactoringjs.com');
console.log(response.body);
```

But this will print `undefined`. The reason is that our `response` constant is actually named a bit optimistically. The return value of `http.get` is a `ClientRequest` object, not a response at all. It describes the request, but not the result.

There are good reasons to use asynchronous remote HTTP calls. If our remote call were synchronous, it would necessarily "stop the world" (STW) and keep our program waiting. Still, when we look at the immediate alternative, this compromise may feel frustratingly complex:

```
http.get('http://refactoringjs.com', (result) => {
  result.on('data', (chunk) => {
    console.log(chunk.toString());
  });
});
```

WHY TOSTRING()?

`chunk` is a `Buffer` of characters. Try leaving off the `toString()`, and you'll see something like `<Buffer 3c 21 44 4f 43 ... >`.

This is clearly a more complicated process than our idea of how a synchronous HTTP API would function. It forces not only the asynchronous style, but the functional paradigm as well. We have two inner asynchronous functions. If you try to cling to using synchronous-style coding beyond this initial call, your frustration won't stop:

```
let theResult = [];
http.get('http://refactoringjs.com', (result) => {
  result.on('data', (chunk) => {
    theResult.push(chunk.toString());
  });
});
console.log(theResult);
```

Now we can get an array of the chunks of the response, right? Nope. This prints an empty array: `[]`.

The reason is that the `http.get` function *returns* right away, and `console.log` is evaluated before the chunks are pushed onto the array in the callback. In other words:

```
http.get('http://refactoringjs.com', (result) => {
  result.on('data', (chunk) => {
    console.log('this prints after (also twice)');
  });
});
console.log('this prints first');
```

The last line prints before the innermost function has a chance to execute the third line (incidentally, it prints that logging statement twice). So if it's just a

matter of waiting, and we want to do something with the chunks, we should just be able to wait, right? But how long do we wait? Is 500 milliseconds enough time?

```
let theResult = [];
http.get('http://refactoringjs.com', (result) => {
  result.on('data', (chunk) => {
    theResult.push(chunk.toString());
  });
});
setTimeout(function(){console.log(theResult)}, 500);
```

It's hard to say. Using this approach, we might end up with an empty array, or an array with one or more elements. If we really want to be sure that the data is in place before we log it (or do anything else with it), we'll end up waiting too much, and tying up our program too. If we wait too little, we'll miss some data. So this solution isn't very good. Not only is it unpredictable, but it involves setting state through a side effect.

SETTIMEOUT AND THE EVENT LOOP

It is worth noting that `setTimeout(myFunction, 300)` doesn't necessarily execute `myFunction` after 300 milliseconds. What it does is first return (in node, if you assign with `x = setTimeout(myFunction, 300)`, you'll see that it returns a `Timeout` object), and then add the function to the event loop to be executed after 300 milliseconds has passed.

There are two questions to keep in mind with this type of situation. First, will the event loop be stuck doing something else at 300 milliseconds? Maybe.

Second, does code execute immediately when given a timeout of 0 milliseconds? In other words, what executes first?

```
setTimeout(() => {console.log('the chicken')}, 0);
console.log('the egg');
```

In this case, `"the egg"` will get printed first.

What about this?

```
setTimeout(() => {console.log('the chicken')}, 2);
setTimeout(() => {console.log('the chicken 2')}, 0);
setTimeout(() => {console.log('the chicken 3')}, 1);
setTimeout(() => {console.log('the chicken 4')}, 1);
setTimeout(() => {console.log('the chicken 5')}, 1);
setTimeout(() => {console.log('the chicken 6')}, 1);
```

```
setTimeout(() => {console.log('the chicken 7')}, 0);
setTimeout(() => {console.log('the chicken 8')}, 2);
console.log('the egg');
```

The egg wins again, but the chickens are all over the place. Try running it in different browser consoles and on node. As of this writing, the order on Chrome and Firefox is different but consistent. The order on node varies from run to run.

With the problems of the `setTimeout` approaches, it looks like we're better off going back to our first async method (the code snippet directly before "**Why toString()?**"), but is that really the best way to write it?

Fixing the Pyramid of Doom

If you're unfamiliar with the term "pyramid of doom" or the often-related "callback hell," they can both refer to code in the following form:

```
levelOne(function(){
  levelTwo(function(){
    levelThree(function(){
      levelFour(function(){
        // some code here
      });
    });
  });
});
```

The "pyramid of doom" refers to the shape where code creeps to the right with many levels of indentation. "Callback hell" is less about the shape of the code, and more of a description of code that follows many layers of callback functions.

Extracting Functions into a Containing Object

Let's get back to the code from the last section. Clearly, we're going to want some asynchronous form here, but how do we manage the complexity and nesting required with something like this?

```
const http = require('http');
http.get('http://refactoringjs.com', (result) => {
  result.on('data', (chunk) => {
    console.log(chunk.toString());
```

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

Note that this could be much more complicated with many more levels of nesting. This is callback hell and the pyramid of doom in action. There's no strict line for how much indentation constitutes a pyramid of doom or how many callbacks put you in "hell."

We already have a strategy for this from previous parts of the book. We simply de-anonymize and extract a function like this:

```
const http = require('http');
function printBody(chunk){
  console.log(chunk.toString());
};

http.get('http://refactoringjs.com', (result) => {
  result.on('data', printBody);
});
```

And we could even name and extract another:

```
const http = require('http');
function printBody(chunk){
  console.log(chunk.toString());
};

function getResults(result){
  result.on('data', printBody);
};

http.get('http://refactoringjs.com', getResults);
```

Now we're left with two named functions and the last line as a piece of *client* or *calling* code.

Because this is a streaming API, delivering "chunks" of data rather than the full HTML body at once, our code would currently put a line break between chunks. Let's avoid that with an array to capture the results:

```
const http = require('http');
let bodyArray = [];
const saveBody = function(chunk){
  bodyArray.push(chunk);
};
const printBody = function(){
  console.log(bodyArray.join(''))
};
const getResults = function(result){
```

```
  result.on('data', saveBody);
  result.on('end', printBody);
};

http.get('http://refactoringjs.com', getResults);
```

Note that we had to add a new event handler for the `'end'` event, and we no longer need the `toString` because our `join` function is smart enough to coerce the buffers into one string.

Now that we have extracted our functions and are printing properly, we might be tempted to further change the code by moving this into an object, exporting a module, and defining our public interface by some combination of pseudoprivate functions (prefixed with an underscore), classes, factory functions, or constructor functions. Depending on what you liked from the previous chapters (especially Chapters **5**, **6**, and **7**) and your own style, any of these options might seem overkill or prudent.

If you do decide to move things into an object, one thing to be aware of is how easily the `this` context will get dropped:

```
const http = require('http');
const getBody = {
  bodyArray: [],
  saveBody: function(chunk){
    this.bodyArray.push(chunk);
  },
  printBody: function(){
    console.log(this.bodyArray.join(''))
  },
  getResult: function(result){
    result.on('data', this.saveBody);
    result.on('end', this.printBody);
  }
};

http.get('http://refactoringjs.com', getBody.getResult);
```

This code will lead to the following error:

```
TypeError: "listener" argument must be a function
```

This means that in the last line, `getBody.getResult` is not a function. Changing that last line gets us a bit further:

```
http.get('http://refactoringjs.com', getBody.getResult.bind(getBody));
```

But we still get an error from pushing onto the `bodyArray`:

```
TypeError: Cannot read property 'push' of undefined
```

To get everything passing around `this` properly to the callbacks, we'll need our code to `bind` `this` for the callbacks of the events in `getResult` as well:

```
const http = require('http');
const getBody = {
  bodyArray: [],
  saveBody: function(chunk){
    this.bodyArray.push(chunk);
  },
  printBody: function(){
    console.log(this.bodyArray.join(''))
  },
  getResult: function(result){
    result.on('data', this.saveBody.bind(this));
    result.on('end', this.printBody.bind(this));
  }
};

http.get('http://refactoringjs.com', getBody.getResult.bind(getBody));
```

Is it worth putting this in an object? Is it worth disabling what was a fairly shallow pyramid of doom? Did we eliminate callback hell? Maybe not, but it's good to have options. We'll stick with this form to start the next section.

Before moving on, though, there are two critical things for us to notice.

First, by relying on callbacks to do the real work of our program, we're also counting on side effects. We've left the simple world of returning values. We're not even just *returning* values from functions. We're running them (and returning *right away* with nothing of value*)*, but the callbacks will run *sometime*. Our fundamental basis for achieving confidence relies on knowing what's happening in our code, and async in JavaScript, as we know it so far, completely undermines this.

Second, and related to the first point, we don't have any tests! But what would we test anyway? Let's start with our old assumptions about how testing works, and try to test some known value. We know that after the function is executed, the `bodyArray` should have some data in it. In other words, its length should not be equal to zero.

Testing Our Asynchronous Program

With that in mind, let's work with the testing library from **Chapter 9** called tape. It is a bit simpler than mocha, and you can run it just by running **node**

whatever-you-call-the-file.js. You can install it with **`npm install tape`**.

The following test will fail:

```
const http = require('http');
const getBody = {
...
}
const test = require('tape');
test('our async routine', function(assert){
  http.get('http://refactoringjs.com',
           getBody.getResult.bind(getBody));
  assert.notEqual(getBody.bodyArray.length, 0);
  assert.end();
});
```

Why? Because it is executed before `bodyArray` has a chance to be updated!

You might instinctively want to crawl back into a comfortable synchronous world with an update to the test like this:

```
test('our async routine', function(assert){
  http.get('http://refactoringjs.com',
           getBody.getResult.bind(getBody));
  setTimeout(() => {
    assert.notEqual(getBody.bodyArray.length, 0);
    assert.end();
  }, 3000);
});
```

Here we get a passing test, but it takes 3 seconds to execute.

So how can we delay our assertion until after the `bodyArray` gets populated?

Because we are testing a side effect, and our code is not very "callback-friendly," we're stuck with `setTimeout` unless we rewrite the code or add some odd machinations to our tests. In an ideal case, `printBody` would take a callback that would run to indicate that everything is all done.

Trading one blunt tool for another, we can eliminate our reliance on `setTimeout` by overwriting the existing function that indicates when things are done:

```
test('our async routine', function (assert) {
  getBody.printBody = function(){
    assert.notEqual(getBody.bodyArray.length, 0);
    assert.end();
  }
  http.get('http://refactoringjs.com',
```

```
            getBody.getResult.bind(getBody));
});
```

This might seem outrageous, for a couple of reasons. First, it overwrites a function that we may want to test later (will we have to restore the original implementation?). Second, changes to `printBody`'s implementation could lead to some complexity later. Without introducing mocking, or trying to feed a callback all the way through each function and event, we could do slightly better:

```
const getBody = {
...
  printBody: function(){
    console.log(this.bodyArray.join(''))
    this.allDone();
  },
  allDone: function(){}
}

test('our async routine', function (assert) {
  getBody.allDone = function(){
    assert.equal(getBody.bodyArray.length, 2);
    assert.end();
  }
  http.get('http://refactoringjs.com',
           getBody.getResult.bind(getBody));
});
```

Here, we create a function whose only responsibility is running when `printBody` runs. Because there is no default implementation, overwriting it for the test is no big deal. We'll just need to reset it in future tests. Here is an additional test that ensures that setting the `bodyArray` to `[]` allows a clean slate:

```
test('our async routine', function (assert) {
  getBody.allDone = function(){
    assert.equal(getBody.bodyArray.length, 2);
    assert.end();
  }
  http.get('http://refactoringjs.com',
           getBody.getResult.bind(getBody));
});

test('our async routine two', function (assert) {
  getBody.bodyArray = [];
  getBody.allDone = function(){ };
  http.get('http://refactoringjs.com',
           getBody.getResult.bind(getBody));
  assert.equal(getBody.bodyArray.length, 0);
```

```
  assert.end();
});
```

Additional Testing Considerations

Considering that we also need to reset our `bodyArray` to an empty array (and revert any other side effects, such as would appear in a database), one additional upkeep step shouldn't trouble us too much. We can even refactor these steps into simple functions:

```
function setup(){
  getBody.bodyArray = [];
}
function teardown(){
  getBody.allDone = function(){ };
}

test('our async routine', function (assert) {
  setup();
  getBody.allDone = function(){
    assert.equal(getBody.bodyArray.length, 2);
    teardown();
    assert.end();
  }
  http.get('http://refactoringjs.com',
           getBody.getResult.bind(getBody));
});

test('our async routine two', function (assert) {
  setup();
  http.get('http://refactoringjs.com',
           getBody.getResult.bind(getBody));
  assert.equal(getBody.bodyArray.length, 0);
  teardown();
  assert.end();
});
```

Note that mocha and other more fully featured frameworks try to handle setup and teardown on your behalf. They do okay most of the time, but having explicit `setup` and `teardown` functions (as in the last example) gives you more control.

TEST PARALLELIZATION

No framework will save you from tests running in parallel and clobbering shared state. The solution to this is to run tests that share state serially (as a tape test file will do). And for disparate aspects of the code, splitting these into modules and giving each its own chance to run independently and in parallel will still let you have speedy, parallel test runs.

Architecturally, splitting your code into modules is probably what you wanted to do anyway, right?

If that sounds like too much work, go with mocha or something else that handles setup/teardown. But don't be surprised if you still have an occasional parallelization problem (most likely resulting in test failures).

Let's take care of that function reassignment using the testdouble library (**`npm install testdouble`**):

```
const testDouble = require('testdouble');

function setup(){
  getBody.bodyArray = [];
}
function teardown(){
  getBody.allDone = function(){ };
}
test('our async routine', function (assert) {
  getBody.allDone = testDouble.function();
  testDouble.when(getBody.allDone()).thenDo(function(){
    assert.notEqual(getBody.bodyArray.length, 0)
    assert.end()
  });
  http.get('http://refactoringjs.com',
           getBody.getResult.bind(getBody));
});
```

When we make a test double like this for our function (we could also do it for whole objects), our code can now just "fake" the call to `allDone`. More typically, doubles are used to avoid performing expensive or otherwise slow operations (such as calling an external API), but be wary of using this technique too much, as it is possible to fake everything, which results in useless tests. One thing to notice is how convenient our `teardown` is. It's easy to reassign this one empty function, but if we were creating doubles of more functions (mocking, stubbing, spying, etc.), our `teardown` could get pretty complicated.

How about this for isolation?

```
function setup(){
  return Object.create(getBody);
```

```
};

test('our async routine', function (assert) {
  const newBody = setup();
  newBody.allDone = testDouble.function();
  testDouble.when(newBody.allDone()).thenDo(function(){
    assert.notEqual(newBody.bodyArray.length, 0)
    assert.end()
  });
  http.get('http://refactoringjs.com',
           newBody.getResult.bind(newBody));
});
```

Instead of having to reset our object, we can just use a new one with our test runs. Our `setup` function might not be specific enough for every situation, but it's perfect for this.

DID WE TEST ENOUGH?

Depending on our confidence, we can pretty much always add more tests. In this case, we might have opted to return the HTML string from `printBody` and test that (probably with a regex rather than a full match). We could have made a double for this call:

```
result.on('data', this.saveBody.bind(this));
```

And made it always produce a simple HTML fragment.

Additionally, we could test the fact that a function was called at all, or that it *was called and did not produce an error*.

In a lot of asynchronous code, the return values are not as interesting (or confidence-producing) as knowing what functions were called and what other side effects took place.

In this section, we created objects and explored some lightweight testing options for asynchronous code. You might lean toward heavier tools like mocha (as we used earlier) for testing and Sinon.JS (which we haven't looked at) for doubles. Or you might try out simpler tools like tape and testdouble. You might even want to just scrape by with `setTimeout` and `assert` or `wish` statements from time to time.

As we discussed in **Chapter 3**, you have many options for tooling and testing. If something feels overly complex or doesn't do what you need it to, you can always tool down or tool up as needed. Flexibility and clarity are more important than hitting a screw with a hammer until it drives into the wall.

Callbacks and Testing

From the last section, we have a new approach to fixing the pyramid of doom, but it doesn't really get us out of callback hell. In fact, by naming and extracting functions, we may actually reduce the clarity of our code in some cases. Rather than callbacks being nested, they could be spread across the file or multiple files.

Creating an object helped with keeping callbacks somewhat organized, but that isn't our only option. Part of what led us to that solution was the utility of having a container to store our side effect array. We had to do some aggregation based on the streaming/event-emitting nature of node's `http` library and our desire to print the entire HTML body at once. If we were adding something to a page or saving a file, we might consider allowing the file or DOM to *be* the aggregate itself, and just pass the results of the stream to it instead of getting them into an intermediate form (the array).

We've seen that passing a (callback) function into another function is useful for asynchronous programming, but it completely changes how we've been working in the earlier parts of this book. Instead of returning something valuable and acting on it, we're letting the inner function call the shots (and its inner function [and its inner function]). This is a property of *continuation passing style* (CPS) called *inversion of control* (IoC), and while useful, it has some drawbacks:

- It's confusing. You have to think backward until you get used to it.
- It makes function signatures complex. Instead of function parameters acting as "inputs," they now may be responsible for outputs as well.
- Callback hell and the pyramid of doom are both likely without organizing code into objects or other high-level containers.
- Error handling is more complicated.

Additionally, asynchronous code in general is hard:

- It makes testing more difficult (although part of this is just the nature of asynchronous programming).
- It's hard to mix with synchronous code.
- Return values are likely no longer important throughout the sequence of callbacks. This makes us rely on testing of callback arguments to determine intermediate values.

Basic CPS and IoC

Let's look at an example of the most basic usage of callbacks in a function, simply to see this inversion of control in action. It doesn't even have to be asyn-

chronous. Here's an example of a noncallback (aka "direct style") version of our function:

```
function addOne(addend){
  console.log(addend + 1);
};
addOne(2);
```

And when we use callbacks to do the same thing:

```
function two(callback){
  callback(2);
};
two((addend) => console.log(addend + 1));
```

The weight of the algorithm is now in the callback, rather than the `two` function, which merely gives up control and passes a 2 to the callback. As we have done before, we can name and extract the anonymous function, giving us this:

```
function two(callback){
  callback(2);
};
function addOne(addend){
  console.log(addend + 1);
};
two(addOne);
```

The value of the calling function (`two`) is that it supplies a variable to the callback. In this case, that is all it does. If the `two` function needed its value to come from some longer-running task, we'd want a callback (CPS) version. However, because `two` can immediately return, the direct style is fine, if not preferable.

Let's add a `three` function that *does* need to work asynchronously:

```
function three(callback){
  setTimeout(function(){
    callback(3);
    },
  500);
};
three(addOne);
```

If we tried doing the same synchronously:

```
function three(){
  setTimeout(function(){
```

```
    return 3
    },
    500);
}
function addOne(addend){
  console.log(addend + 1);
};
addOne(three());
```

We end up printing `NaN` ("Not a Number") because `addOne` finishes executing before `three` gets a chance to return. In other words, we're trying to add `1` to `undefined` (the `addend` in `addOne`), resulting in `NaN`. So we'll need to go back to our previous version:

```
function addOne(addend){
  console.log(addend + 1);
};
function three(callback){
  setTimeout(function(){
    callback(3);
    },
  500);
};
three(addOne);
```

Note that we could also have written our `addOne` function to take a callback, like this:

```
function addOne(addend, callback){
  callback(addend + 1);
};
function three(callback){
  setTimeout(function(){
    callback(3, console.log);
    },
    500);
};
three(addOne);
```

Let's stick with this form for the tests.

Callback Style Testing

The example from the last section might seem redundant with our earlier use of the `http.get` function, but there are four reasons we introduced it:

- The motivation in the first part of this chapter was the need to work with an asynchronous *library*. In this example, the motivation is that we need to work with an asynchronous *function* (just one).
- The earlier example is more complex because the callback of `get` leads to other callbacks in multiple `result.on` functions.
- Our earlier example did not use CPS the whole way through. We relied on a nonlocal object to do some of the dirty work.
- A *simple* example, where we write both the interface and the implementation code, is needed because we'll increase the complexity when we introduce promises.

Before we get to promises, we need tests for this code. Earlier, we cheated a bit by relying on a global variable for the value we were testing. We could do the same here, or rely on some kind of test double for `console.log` to check if it is called with the right argument (a viable approach for an end-to-end test), but that's not where we're headed. This time, we'll try doing things a bit more async-like by relying only on the resulting parameters from our callbacks. Since addOne is simpler, let's start there:

```
const test = require('tape');

test('our addOne function', (assert) => {
  addOne(3, (result) => {
    assert.equal(result, 4);
    assert.end();
  });
});
```

For the sake of testing, we're basically treating `result` like we would a return value. Instead of testing the return value, we're testing the parameter that is passed to the callback. As for reading this, we could say this:

1. addOne takes two arguments: a number and a callback.
2. We're passing in a callback as the second (actual) *argument*. It is an anonymous function.
3. That anonymous function has a (formal) *parameter* we call `result`. We are *declaring* the function in the test.
4. That anonymous function is *called* inside of addOne, with the argument (`result`) being the addition of 1 and whatever was passed as the first argument to addOne.
5. We test that result against the numerical literal 4.
6. We end the test.

Look at how different the addOne function and its test would be if we were just returning the result:

```
function addOneSync(addend){
  return addend + 1;
};
...
test('our addOneSync function', (assert) => {
  assert.equal(addOneSync(3), 4);
  assert.end();
});
```

Here we:

1. Pass 3 to the `addOne` function and get the return value.
2. Test that return value against the numerical literal 4.
3. End the test.

One of those processes is way simpler, but we're living in an async world a lot of the time. Also, as we saw earlier, our `three` function doesn't have the luxury of having a usable synchronous analog. Here is our test:

```
test('our three function', (assert) => {
  three((result, callback) => {
    assert.equal(result, 3);
    assert.equal(callback, console.log);
  });
  assert.end();
});
```

The `three` function takes only one argument. That is a function, which we've left anonymous. That anonymous function takes two parameters, which are supplied as arguments when the anonymous function is called inside of `three`. One is the `result` and the other is the `callback`. Our tests confirm that `result` is 3 and `callback` is `console.log`.

If we want an end-to-end test, our best bet is using the testdouble library to see if `console.log` is called with 4:

```
const testDouble = require('testdouble');
test('our end-to-end test', (assert) => {
  testDouble.replace(console, 'log')
  three((result, callback) => {
    addOne(result, callback)
    testDouble.verify(console.log(4));
    testDouble.reset();
    assert.end();
  });
});
```

There are a couple of things worth noting here. First, the `testdouble.replace` function replaces the `console.log` function with a double that we can check later when we call `verify`. Following that, `testdouble.reset` restores `console.log` to its former self. Recall earlier when we were talking about creating a `teardown` function. We could use `testdouble.reset` to put our doubles back, meaning that after we do that, we could use `console.log` as normal.

So now that we have tests in place, let's get to promisifying.

Promises

If you like being able to write asynchronous JavaScript, but *don't* like the mess that comes along with inversion of control, promises are for you. Let's recap what we've seen so far. In direct style, you return values from functions and use those return values in other functions. In CPS (continuation passing style), you invert control by the calling code supplying (and often defining inline) a callback to be executed by the function that is called. Return values become little better than meaningless, and instead the arguments passed from the callback (one is conventionally called `result`) become the focus of tests and subsequent callbacks.

Although using callbacks opens up the possibility of asynchronous code (without using some kind of polling), we introduce some complexity both in the way we structure our functions and in the way we call them.

Promises shift this complexity to the function definition side of things, leaving the function calling code with a relatively simple API. For most situations, promises are a better choice than CPS. And where they aren't the right choice, CPS probably isn't either. In those cases, you might be looking for stream handling, observables, or some other high-level pattern.

The Basic Promise Interface

So how do we use promises? We'll cover their implementation soon, but for now, let's see what the promise interface looks like:

```
// promises
four()
.then(addOne)
.then(console.log);
```

This is pretty straightforward. We have a function that returns a 4 (wrapped in a promise), acted on by `addOne` (which itself returns a promise), which in turn is acted on by `console.log`.

If we're looking to compose functions, promises are much easier to work with. Callbacks have us stuck either hardcoding function names (and/or function literals for callbacks) in the function declarations, passing them in as extra parameters in the function, or using some other nontrivial and somewhat confusing alternatives.

With promises, we are just chaining values together. We have a value (wrapped in a promise), and `then` *unwraps* it by waiting (when necessary), then passes that value as a parameter to the promise or function. To illustrate a bit more of the interface, we could also write form 2:

```
// form 1
four()
.then(addOne)
.then(console.log);

// form 2
four()
.then((valueFromFour) => addOne(valueFromFour))
.then((valueFromAddOne) => console.log(valueFromAddOne));
```

In these forms, we have a function literal or reference. Both the function definitions and the function calls of `addOne` and `console.log` happen elsewhere. Form 1 is preferable when possible (it is also known as "point-free," a style we'll discuss more in the next chapter).

Moving from form 2 to form 1 has a similar feel to naming and extracting an anonymous function. In both cases, the function *calls* happen elsewhere, and may even be implicit or outside of your codebase (i.e., you won't be able to "grep" for them). In the case of moving from form 2 to form 1, however, the function *definitions* (along with their names) already exist, so we only need to drop the anonymous wrapping function.

THE FLEXIBILITY OF PROMISES

If you're still not sure about the utility of promises over callbacks, take a look at this:

```
four()
.then(addOne)
.then(addOne)
.then(addOne)
.then(addOne)
.then(addOne)
.then(console.log);
```

We can chain as many `addOne`s as we want. It's basically a fluent interface if you ignore the `then` calls, and it's async-friendly. You can do this with CPS, but you're headed for the pyramid of doom (and hard-to-test intermediate results).

Creating and Using Promises

Now that we have a good idea of why promises are often a good choice over callbacks, let's take a look at how we actually implement them:

```
four()
.then(addOne)
```

```
.then(console.log);

function addOne(addend){
  return Promise.resolve(addend + 1);
};

function four(){
  return new Promise((resolve, _reject) => {
    setTimeout(() => resolve(4), 500);
  });
};
```

The first three lines should be very familiar by now. So how do the new functions work? It might seem complex inside the function bodies, but notice that we're no longer passing callbacks, which can get very confusing. Also, we get our `return` statements back!

Unfortunately, what we are returning are promises, which may seem hard to understand. But they aren't. It's just like making toast:

1. You start with a toaster.
2. You put your bread in it along with some input on how to toast it.
3. The toaster determines when the bread is toasty enough and pops it up.
4. After it's ready, you get your toast and consume it how you see fit.

The same four steps are true of promises:

1. You start with a promise (usually created with `new Promise`, but the previous example also shows that you can create one with `Promise.resolve`).
2. You put a value *or* the process to create a value (which may be asynchronous) into the promise.
3. After a timer or receiving the result of some asynchronous function, the value is set by `resolve(someValue)`.
4. This value is returned wrapped in a promise. You pull it out of the toaster —er, promise—with the `then` function, and consume the value as you see fit.

DO YOU HATE METAPHORS?

No? Good. We'll be discussing burritos in Chapter 11. They're kind of like making toast.

But isn't a promise a high-level functional structure like a functor or monad or similar? Maybe, but that's a huge topic, and we can't get into all of it in this book.

Chapter 11 gives a good introduction to practical functional coding, but we'll leave the theory out of it and focus on using good coding interfaces.

Back to our example, our `addOne` function returns a promise created with `Promise.resolve(addend + 1)`. This is fine for cases where we only need a value, but using the `new Promise` constructor function and supplying a callback (the *executor*) that calls `resolve` or `reject` (the two functions named by the signature of the callback of the constructor, the executor) provides more flexibility.

SOME CONSIDERATIONS ABOUT THEN

There is something to note about our `addOne` function:

```
function addOne(addend){
  return Promise.resolve(addend + 1);
}
```

It would work just as well if the second line were this:

```
  return addend + 1;
```

Why? Because `then` will accept a promise *or* a function (or two, actually: the first for fulfillment and the second for rejection). Try this:

```
four()
.then(() => 6)
.then(console.log);
```

In this case, `6` will be printed. The first `then`'s callback throws away the `4` and just passes `6` along.

However, note that `four` cannot be a simple function returning a simple value, even without considering the `setTimeout` aspect of it. The *first* function in a promise chain must be "`then`-able"—that is, return an object that supports the `.then(`*`fulfillment, rejection`*`)` interface. That said, you could start off just by having a resolved promise:

```
Promise.resolve()
.then(() => 4)
```

```
  .then(() => 6)
  .then(console.log);
```

A `resolve` will make the value available inside of the `then` function. The `reject` function will create a *rejected* promise object. There is a `catch` function that will catch some errors (and miss others, so be careful). There are also `Promise.all` and `Promise.race` functions to, respectively, return when all promises complete and return the first promise that completes.

In some ways, it is a fairly small API, but the variations around error handling and setting promises up can make for a tricky experience. Still, the interface it provides makes the upfront work worth it.

Testing Promises

To finish things up neatly, let's see how the tests adapt to this new interface:

```
function addOne(addend){
  return Promise.resolve(addend + 1);
}

function four(){
  return new Promise((resolve, _reject) => {
    setTimeout(() => resolve(4), 500);
  })
}

const test = require('tape');
const testdouble = require('testdouble');

test('our addOne function', (assert) => {
  addOne(3).then((result) => {
    assert.equal(result, 4);
    assert.end();
  });
});

test('our four function', (assert) => {
  four().then((result) => {
    assert.equal(result, 4);
    assert.end();
  });
});

test('our end-to-end test', (assert) => {
```

```
  testdouble.replace(console, 'log')
  four()
  .then(addOne)
  .then(console.log)
  .then(() => {
    testdouble.verify(console.log(5));
    assert.pass();
    testdouble.reset();
    assert.end();
  }).catch((e) => {
    testdouble.reset();
    console.log(e);
  })
});
```

The first two tests, which are low-level, are relatively unchanged. The end-to-end test has changed quite a bit. After replacing `console.log` so that we can monitor it, we kick off the promise chain with our promise-returning `four` function. We chain our `addOne` and `console.log` callbacks with `then` functions. And then we have another `then`, with an anonymous function as the only argument. Inside of that anonymous function, we `verify` that `console.log` was called with 5. Following that, we call `assert.pass` so that our test output will confirm three instead of two passing tests. We need that because `verify` isn't part of `tape` and doesn't produce a passing assertion. Then we do the teardown with `testdouble.reset` and `assert.end`.

You might be wondering what we're doing with the `catch`. Well, unfortunately, after we replace `console.log`, our errors will no longer print anything! `catch` allows us to put `console.log` back with `testdouble.replace` *before* printing the error with `console.log(e)`.

IS CHANGING CALLBACK-STYLE CODE INTO PROMISES "REFACTORING?"

Probably not, unless you aren't concerned with unit testing at all and consider your "interface" to be some very high-level interactions with the code. The value in promises is that they change interfaces, and those are probably where you would want your testing to be.

So why spend so much time on promises in *Refactoring JavaScript*?

There are three reasons. First, you'll probably hear someone at some point talk about "refactoring to use promises." You'll know this actually means supporting new interfaces, and writing new code with new tests. See the diagram in **Chapter 5** on what tests to write (**Figure 5–1**). The second reason is that knowing where you

are in the testing cycle (testing before new code, increasing coverage, or refactoring) is the most important thing to have a handle on when developing confidence in a codebase. Third, there are tons of *cool things* in JavaScript (canvas, webvr, webgl, etc.) that make somewhat niche applications possible, but asynchronous programming with promises is increasingly important for JavaScript developers of all kinds.

Wrapping Up

Async is a huge and active area of development in JavaScript, and we only scratched the surface. Other things worth exploring are web workers, streams, observables, generators, `async`/`await`, and nonnative async/promise utilities.

For the most part, using any of these features will involve dramatically changing code and not refactoring as we define it. But having a base knowledge of the interfaces involved in your code, as well as how to test them, is crucial before any refactoring can take place.

Despite "refactoring to use promises" not really fitting our concept of refactoring, the interface is one that we should prefer (at least until `async` and `await` become more widely available), because it generates useful return values rather than relying on calling other functions (side effects). By the same token, `async` and `await` are interesting because they allow us to write synchronous-looking code just by adding a few keywords. However, as of this writing, their spec and implementations are not yet realized.

In *Design Patterns: Elements of Reusable Object-Oriented Software*, the Gang of Four's advice to "code to an interface, not an implementation" must be predicated on the *capacity to choose* what interface you want. Following that, you can develop confidence through writing tests and refactoring. Here, we've explored a few more options for interfaces.

11 Functional Refactoring

Of all the styles of JavaScript you can write, this one probably draws from the deepest well. Functional programming is older than object-oriented programming and even imperative programming. We're going to take a few drinks from this well, but try to avoid falling in. FP has so much history and theory behind it that we have to set some boundaries.

There is an expression that "farmers don't want to own all the land, just all the land their land touches."

Learning functional programming in JavaScript has this same temptation. Here is a list of statements that might hold true for some people, but are not what we're dealing with in this chapter:

- To learn functional programming, you need to learn Scheme/Haskell.
- To do "real" functional programming in JavaScript, you need to compile to JS from PureScript, TypeScript, ClojureScript, or something else.
- To learn functional programming, you need to learn the lambda calculus.
- To learn functional programming, you need to learn category theory.

A CONFESSION

An implicit statement in the above is:

> Evan needs to write this chapter as a 900-page book from the perspective of a Math/CS dual PhD (which he isn't).

Sorry to disappoint, but that's not going to happen in this chapter.

The focus here is entirely on the practical concern we've had all along: building confidence through developing and maintaining the interfaces of our code. To that end, we'll be covering five main topics:

- The restrictions and benefits of functional programming
- The basics of FP
- More basics of FP
- Burritos
- Moving from OOP to FP

The Restrictions and Benefits of Functional Programming

Generally speaking, programming within a functional paradigm requires a bit more structure than a typical JavaScript programmer might be used to. With that structure comes certain guarantees and challenges. In this section, we'll explore some of the basic trade-offs in attempting a functional style.

Restrictions

Here's a common piece of code that is often trotted out to denigrate programming in a nonfunctional style:

```
x = 1;
x = x + 1;
```

That's it. It might seem pretty innocent, but there are many problems here. Many JavaScript programmers will be fine with this. Some would just be bothered by the first assignment not being preceded by a `var` to scope it within a function. Other people would want a `let` to scope at the block level. As we've discussed earlier in the book, what we really want here is a `const`, which would prevent the reassignment on the second line. On a chalkboard in a math class, this would look especially ludicrous.

We covered earlier how reassignment makes programs more complicated to deal with. But reassignment goes a bit deeper than the strictly practical.

If we were thinking of both of these statements as mathematical *facts*, is `x` equal to `1` or does `x` somehow equal `1` *and* `1 + 1`? Apparently it depends where you are in the program, right? So we can't look at these statements as facts about `x` anymore. We have *variable assignments* instead of *facts* (values in functional terms). Values shouldn't change. When they do, our programs look less like math and more like sequences of instructions.

Okay, so we should be able to solve anyone's objections like this, right?

```
const x = 1
const y = x + 1
```

The second problem, and this might seem strange, is that just by having assignments to begin with, we've introduced the concept of *time* into our program. There is a time before these values exist, and another one after. Yes, our JavaScript programs are steps of execution no matter how we go about it. However, in *declarative programming* (of which FP is one type), we seek to deemphasize those mechanics, describing *what* a program should do rather than *how* it should do it. Moreover, assignments in and of themselves are worthless unless they're actually applied to a function (including one that just prints the value of the assignment).

SPREADSHEETS AS DECLARATIVE PROGRAMMING

As a simple example of declarative programming, consider a spreadsheet program. Some cells contain data (facts/values). Other cells contain functions to manipulate and use those calculations. Other cells have functions that use the results of executing other functions.

When using a spreadsheet program, we don't really think about the flow of execution of the program. Are the numbers displayed in the cells before they are used in other calculations? If a number is used in multiple calculations, which calculation is executed first?

By the way, if you want to try out a declarative programming language that's especially distant from JavaScript, you should look into Prolog.

If worrying about reassignments and even assignments seems really intense, I have bad news, good news, better news, and then some more bad news. The first bit of bad news is that you might find a number of restrictions in a given functional programming language, such as:

- *Variables* don't exist. They may be called *values*, which are constants.
- There is no shared global state (without some difficulty).
- Assignment itself is more complex. Values tend to come from or go into functions.
- Functions should always return something.
- `if` statements without `else` branches are invalid.

- Functions are declared along with a type signature that specifies inputs and outputs.
- There is a compilation step where types need to line up properly (more on this later).
- The language makes it difficult to use side effects that may be trivial in other languages.
- There's no concept of `null`.

You may find that not everything in this list applies to every functional language.

Benefits

The good news is that with those restrictions come a ton of benefits. When values exist only in a small scope and can't change, you can trust that a function called with the same arguments always returns the same result. This is also called *idempotence*, which is half of what makes a *pure function*. The other half is not producing side effects.

By this point in the book, functions returning something is a goal we've been striving for anyway, but what's the benefit of restricting `else`? If you look at code for an `if` without an `else`, you might notice that it will either change the returned value, throw an error, or have some other kind of side effect. In other words, it's a vector for impurity to sneak into our functions.

Because *type signatures* (we cover them more later, but just think about how we talked about input and output types like `string` and `number` in **Chapter 5**) establish what types of values get passed around where, functional purity can be protected at a compilation stage if you have one (and it's nice to discover errors, including type errors, then rather than at runtime). In a stricter functional programming language like Haskell, you can have impure functions, but they can be complicated to set up. They could very well take the "*fun*" out of "*fun*ctional" for many and make the code just "ctional."

As far as the last item in the bulleted list in the previous section, as we talked about before, avoiding `null` saves us a billion dollars. Of all the restrictions, this is the best. You won't feel restricted while swimming in your Scrooge McDuck vault of gold.

And now for the better news. The implications of using pure functions range from convenient to quite compelling. In a stateless world, there is no longer a question of when something happened. If two processing cores of a machine (or more) run the same function, they get the same result, and it doesn't matter which finishes first. In other words, you can avoid *race conditions*.

Additionally, if functions always return the same thing when called with the same arguments, they are *referentially transparent*, which means we can substitute the evaluation of the expression with the value produced. Following from that, we (through minor effort in setting up) never have to evaluate a function twice in a row.

BY THE WAY, RECURSION

If you've read this far into the book, you're either familiar with recursion or completely capable of understanding it. We haven't done much with it so far, but it deserves a brief explanation.

If you have a loop, you could likely use recursion instead. For instance, finding an element can be defined iteratively like this:

```
function find(toFind, array){
  let found = "not found";
  array.forEach((element) => {
    if(element == toFind){
      found = "found";
    };
  });
  return found;
};
console.log(find(3, [3, 9, 2])); // found
console.log(find(3, [2, 9, 3])); // found
console.log(find(3, [2, 9, 2])); // not found
```

But we could also have a recursive `find`:

```
function find(toFind, array){
  if (array[0] === toFind) {
    return "found";
  } else if(array.length === 0){
    return "not found";
  } else{
    return find(toFind, array.slice(1));
  }
};
console.log(find(3, [3, 9, 2])); // found
console.log(find(3, [2, 9, 3])); // found
console.log(find(3, [2, 9, 2])); // not found
```

Both options have performance implications, which we won't get into here, and neither is necessarily the best version of `find`. This is just to illustrate a couple of things you need for recursive functions.

First, you need a call to a function inside of its declaration (e.g., `find` calls `find()` inside of itself). Second, you need a "base case" code path (usually prescribed by a conditional) where the function *won't call* itself. In this case, we have two terminal

branches where the function returns `"found"` or `"not found"`. If you don't have a base case (or something like a timeout), you will end up with an infinite loop.

Let's explore the idea of referential transparency with everyone's favorite recursive function, `factorial`:

```
function factorial(number){
  if(number < 2){
    return 1;
  } else {
    return(number * factorial(number - 1));
  }
};
factorial(3); // returns 6
```

That's all well and good, but since we have a pure function (the outputs depend only on the input), we have an opportunity to *memoize* it:

```
const lookupTable = {};
function memoizedFactorial(number){
  if(number in lookupTable){
    console.log("cached");
    return lookupTable[number];
  }
  else{
    console.log("calculating");
    var reduceValue;
    if(number < 2){
        reduceValue = 1;
    } else {
      reduceValue = number * memoizedFactorial(number - 1);
    }
    lookupTable[number] = reduceValue;
    return reduceValue;
  }
};
console.log(memoizedFactorial(10)); // calculating 10 times
console.log(memoizedFactorial(10)); // cached 1 time
console.log(memoizedFactorial(11)); // calculating (once) and cached
```

If we run this example, we'll see that any time we calculate a value, we add it (along with the `number` parameter) to the lookup table. The first call, `memoizedFactorial(10)`, has to do a bit of work. The second call only needs to ref-

erence the lookup table. The third call still has to calculate for 11, but 10 is already a solved problem, so it just pulls that from the lookupTable.

But now our function relies on something other than explicit parameters! It has become impure! How can we avoid this? We need to make the lookupTable an explicit parameter:

```
function memoizedFactorial(number, lookupTable = {}){
  if(number in lookupTable){
    console.log("cached");
    return lookupTable[number];
  }
  else{
    console.log("calculating");
    var reduceValue;
    if(number < 2){
      reduceValue = 1;
    } else {
      reduceValue =
        number * (memoizedFactorial(number - 1, lookupTable))['result'];
    };
    lookupTable[number] = reduceValue;
    return {result: reduceValue, lookupTable: lookupTable};
  }
};
console.log(memoizedFactorial(10)['result']);
console.log(memoizedFactorial(10)['result']);
```

If you run this, you'll notice that the cache misses on the second call. This shouldn't be too surprising when you look at what we're calling. We have a default explicit parameter for lookupTable that is an empty object, and we never actually pass it in. No cache means no cache hits.

If we add the following code, we'll see the cache is activated for the second call:

```
const lookup = memoizedFactorial(10)['lookupTable'];
console.log(memoizedFactorial(10, lookup));
```

There are other ways we could wrap memoization in using functions. Some memoization functions are made to wrap arbitrary functions, but we won't get into that here. In any case, memoization is a useful tool for some situations, but the setup does have some overhead.

The Future (Maybe) of Functional Programming

Let's get back to some more bad news about FP. Some functional concepts are difficult, and yet, in the long term, the functional people are probably right. Of all the committees, frameworks, and paradigms influencing JavaScript, the functional ideas are showing a lot of promise, because not only are they based on practical software quality principles, but they are also a lens into the state of hardware today and in the future: memory is cheap and computers are parallelizing with multiple cores to keep up with Moore's law. But in order to make use of hardware with multiple processors (or distributed systems), we need code that doesn't care about order of execution so that it can execute in parallel.

It's not hard to see how performance is beginning to become a quality concern. I don't have a crystal ball here, but "You're not writing referentially transparent, pure, functional, concurrent code with compile-time type checking?" becoming the new "You don't write unit tests?" doesn't seem too far-fetched.

The best news, however, is that throughout the earlier parts of this book, we have already made significant efforts toward many of the good parts of functional programming. Not reassigning variables is definitely something we've been through. Keeping variables scoped tightly and converting between functions and variables are both things we're pretty handy with at this point too. And not every other aspect is terribly difficult. For example, separating pure and impure functions is not too hard if we're already good at extracting functions.

The Basics

In JavaScript, the benefits and restrictions of programming in functional style are not as simple as a switch that can be flipped on and off. This is partly because functional style is not rigorously enforced in JavaScript, and also because at a basic level, many of the restrictions we can adhere to and the benefits we can enjoy look generically like "better code" rather than "steps toward functional programming." Let's look at a few of those now.

Avoiding Destructive Actions, Mutation, and Reassignment

When we see a reassignment, we should seek better solutions. For the following trivial example, we would just use `const x = 2`. A preference of `const` to `let` also speaks to this intention:

```
let x = 1;
x = x + 1;
```

AVOIDING REASSIGNMENT IN CONDITIONAL TESTS

Reassignment comes up a lot in conditionals too. Instead of this form:

```
function func(x){
  if(x >= 2){
    x = x + 7;
  }
  return x;
};
```

we can do this:

```
function func(x){
  if(x >= 2){
    return x + 7;
  } else {
    return x;
  }
};

// or

function func(x){
  return x >= 2 ? x + 7 : x;
};
```

You might also see (re)assignments in the conditionals themselves:

```
function func(x){
  if((x = x + 7) >= 9){
    return x;
  } else {
    return x;
  }
};
```

This is a tough one. Reassignment is bad enough, but in conditionals, *any* assignment can make things a bit more confusing. Not only that, but our `else` branch returns the original value of `x` plus seven. In other words, both code paths are the same. Was that obvious to you? Maybe it was, but for me, it took a minute to realize. Anyway, let's assume that we have tests in place, and our `else` branch *does* execute how we want it to.

Where do we go from here? We can start to deal with this by not changing the value of `x`:

```
function func(x){
  if((x + 7) >= 9){
    return x + 7;
  } else {
    return x + 7;
  }
};
```

So now we have repetition, which is ugly, but would be especially bad if instead of `+ 7`, we had some more expensive call that we had to execute twice. However, we have made something else clear by being explicit with the conditionals. We're returning the same thing regardless of input (assuming `x` is a number), so our function can become this:

```
function func(x){
  return x + 7;
};
```

Did you immediately see this when we had the reassignment in the conditional? If so, awesome. Not everyone will see that clearly, though.

Here's another case that is even worse to deal with:

```
function func(x, y){
  if (x > 1000){
    return x;
  } else if((x = x + 7) >= 9){
    return x;
  } else {
    return y;
  }
};
```

This one is tricky, but let's first remove the reassignment:

```
function func2(x, y){
  if (x > 1000){
    return x;
  } else if((x + 7) >= 9){
    return x + 7;
  } else {
    return y;
  }
};
```

This has three effects. First, our `else if` case's `return` is complicated. Second, we're now doing the `+ 7` operation twice. That's not a big deal here, but if the calculation was more expensive, we would expect to take a performance

hit. Third, the `else` case has the original value of `x` as it was passed in. Even though we're not using it, it's good to be aware of that.

To avoid running the function twice, we could set a `newX` variable immediately:

```
function func3(x, y){
  const newX = x + 7;
  if (x > 1000){
    return x;
  } else if(newX >= 9){
    return newX;
  } else {
    return y;
  }
};
```

This prevents us having to execute the `+ 7` operation twice, but now we're doing it once, even when the `if` branch is the one we're executing (which doesn't need `newX` or `x`). One way of handling this is by introducing a nested `if` statement:

```
function func4(x, y){
  if (x > 1000){
    return x;
  } else {
    const newX = x + 7;
    if(newX >= 9){
      return newX;
    } else {
      return y;
    }
  }
};
```

Now we have three code paths, but we're only doing the `x + 7` operation once. Even with the extra complexity, you might find this clearer than burying an assignment inside of a conditional. By the way, in a more complex/expensive operation than `+ 7`, instead of our `const` we might prefer a memoized function, like this:

```
function func5(x, y){
  if (x > 1000){
    return x;
  } else {
    if(memoizedAddSeven(x) >= 9){
      return memoizedAddSeven(x);
```

```
    } else {
      return y;
    }
  }
};
```

So we could have a *cache-aware function* instead of a *cache variable*. This would also let us revert to our simpler form:

```
function func6(x, y){
  if (x > 1000){
    return x;
  } else if (memoizedAddSeven(x) >= 9){
    return memoizedAddSeven(x);
  } else {
    return y;
  }
};
```

Of course, all of what we've done so far has been fairly mechanical. If we actually understand the code, another refactoring would save us from having to mess around with caching variables/functions and ensure we take the performance hit (albeit a very, very small one for addition in this case) only once:

```
function func7(x, y){
  if (x > 1000){
    return x;
  } else if (x >= 2){
    return x + 7;
  } else {
    return y;
  }
};
```

This change might have been obvious from the start, but refactorings that avoid actually understanding the code are easier to apply in many cases. All that said, odds are if we ended up with a function like this, there's some deeper problem elsewhere in the code. (By the way, in this form, we still have three magic numbers, and our names for parameters and the function are not descriptive, searchable, or unique.)

HOW OVERRATED IS UNDERSTANDING CODE?

It might sound like heresy, but hear me out.

It's completely possible to refactor (with tests in place, of course) without having a good understanding of what the code actually does. "Digging in" (especially while debugging) is a tremendously important skill, but quality improvements through refactoring do not always demand it. As in our last example, you may hit a point where without having further context, you're stuck, but this is not always the case.

Similarly, when you have a failed test build (or a bad deployment, eek), it's tempting to "dig in" and see what exactly went wrong, but sometimes, it's much faster to just go back to a version of the code that behaved properly and work from there.

Understanding the code (which includes tracing the execution path, and examining variables and other state at various points in time) is useful but can be tedious. It's important to recognize that it's not the only strategy that works.

AVOIDING REASSIGNMENT IN LOOPS

Next, let's talk about loops (which frequently have a "loop counter" variable that updates). Instead of using them, we should prefer using `forEach`, at minimum:

```
[3, 4, 2].forEach((element) => console.log(element));
```

This is fine for when we have a simple side effect we want to run, but rather than using a loop (or `forEach`) to change values inside of an array, we can use `map` to create a new array with new values:

```
[3, 4, 2].map((element) => element * 2);
```

When we want to filter an array to kick some elements out, we can use `filter`, rather than creating a new array and pushing on the elements that are okay:

```
[3, 4, 2].filter((element) => (element % 2 == 0));
```

When we need to transform an array into some other type of value (object, number, etc.), we're likely going to want `reduce`:

```
[3, 4, 2].reduce((element, accumulator) => element + accumulator);
```

In a `for` loop, any of these functions would tend to work with the index of the loop, as well as an outer variable that reassigns to itself over time. An atomic creation of a new variable makes tests easier and keeps values the same.

If you're looking for a good jumping-off point for higher-order functions like these, the **`Array` docs** (http://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Array) are a good place to start.

BUT YOU CAN DO ANYTHING WITH A FOR LOOP!

Yes, you can. You can loop through all of your values and build up some other value. You can implement all these with a `for` loop or `forEach`:

- `every`
- `filter`
- `find`
- `forEach` (itself)
- `map`
- `reduce`/`reduceRight`
- `some`

You could also implement them all with `reduce`, but why would you do either? Use the right tool for the right job. `map` gives you a new array of transformed elements. `every`, `filter`, `find`, and `some` return something based on a test conditional you provide.

`reduce` gives you some new value based on your array. It's a bit harder to use than the others, but if you want to get back some accumulated value—a sum/product/object/string/array (that won't work with `map`)—`reduce` is what you want. For an easy test of what you want, look above where your loop starts. If it's not the same type as what you're looping through (a number or an empty object usually), you probably want to use `reduce`.

Using these functions means using functions that take functions as parameters. It might not seem exotic at this point in the book, but it's a good entry point toward exploiting JavaScript's support for first-class functions (functions as input and output to other functions). Some may argue that that's not "real" functional programming, but will still admit that we're moving closer to a declarative style in contrast to an imperative style. Think of this as telling the computer *what* to do, instead of *how* to do it.

AVOIDING REASSIGNMENT IN CONDITIONAL BODIES

Although we generally have tried to reduce `if` statements (through polymorphism of subclasses, or through delegation to properties, or extracting them to their own named function as a first step), there are places where they will show up in your code. When that happens, try to provide an `else` case and new variables. So instead of this:

```
let emailSubject = "Hi";
if(weKnowName){
  emailSubject = emailSubject + " " + name;
};
sendEmail(emailSubject, emailBody);
```

try to do this:

```
if(weKnowName){
  let emailSubject = `Hi ${name}`;
} else {
  let emailSubject = "Hi";
};
sendEmail(emailSubject, emailBody);
```

This is much more flexible, as we perceive a few transformations:

```
function emailSubject(){
  if(weKnowName){
    let subject = `Hi ${name}`;
  } else {
    let subject = "Hi";
  };
  return subject;
};
sendEmail(emailSubject(), emailBody);
```

Then it is clear that we don't need the assignment at all:

```
function emailSubject(){
  if(weKnowName){
    return `Hi ${name}`;
  } else {
    return "Hi";
  };
};
sendEmail(emailSubject(), emailBody);
```

Now, we can simplify the statement with ternary syntax:

```
function emailSubject(){
  return weKnowName ? `Hi ${name}` : "Hi"
};
sendEmail(emailSubject(), emailBody);
```

AVOIDING DESTRUCTIVE FUNCTIONS

Another source of mutating values is through using "destructive" functions.

Let's look at `splice` first:

```
const x = [1, 2, 3, 4];
x.splice(1);
```

`splice` (with one argument) will start at the index of that argument (the second element in this case) and return the rest of the array. *But* it also updates your array, leaving you with `[1]`. And because this isn't a reassignment, `const` doesn't save you from `splice`'s destruction.

Instead of `splice`, `slice` will behave itself:

```
const z = [1, 2, 3, 4];
z.slice(1);
```

This returns `[2, 3, 4]` also, but doesn't clobber the original `z` array.

Arrays have destructive functions, including `fill`, `push`, and `pop`. `Object` has `defineProperties`. Numbers and strings as variables have + and = to, respectively, add and concatenate. Arrays have functions (and syntactic sugar, e.g., `x[0] = "something"`) to alter values at particular indices. Objects have `Object.assign`, which mutates the first argument object you give it (you can avoid this by giving it a `{}` as the first argument), as well as dot (`.prop = `) and bracket (`[prop] =`) syntax to alter *or* create properties.

It might seem crazy to avoid direct assignment and a good number of functions just because they are destructive. But you might find it's easier to work with this discipline than to track down a few (or many) rogue global variables that mutate throughout thousands of lines of code.

AVOIDING DESTRUCTION IN GENERAL

Not all functions and convenience syntaxes (such as =) come with a safe version, and `const` doesn't protect you from mutations (only assignments). So if you want to avoid variables *varying*, create new variables to pass along. If you want to reassign something, odds are you're wanting its *updated value*, not just a copy, right? It might as well be a new variable with a new name.

As far as destructive actions (and yes, this includes reassignment and even just assignment when the scope is too large), avoid not only *using* them (when possible), but also *creating* functions that are destructive. It is somewhere between odd and worrisome that "global variables are evil" is seen as common knowledge, but "reassigning variables" doesn't share that reputation, even though they are two sides of the same coin.

DATABASES ARE A GIANT GLOBAL VARIABLE

For some reason, most people give databases a free pass as far as global variables go, although there are some promising approaches that treat each moment in time as a distinct "value" for the entire database. Time will tell if those systems become common wisdom at some point.

Not everyone considers the same things "destructive." For some people, assignments are fine. For other people, reassigning is fine (ugh). For other people, adding or changing the contents of an array or object is fine. For some people, it's an issue of scoping. Even if loop counter variables are unnecessary noise (and we can *at least* use `forEach` or other iterable comprehensions instead), some people are fine with variables that update within a loop. Others are fine with reassignments if the variable is only scoped to the function.

AN AWESOME INDICATOR OF DESTRUCTION

Ruby has a convention (taken from Scheme) to indicate destructive functions with a `!` after the function name (also, `?` is appended for functions that return a boolean). Although it's not used with terrific consistency, it's a very good idea.

Unfortunately, we don't have that convention, and the interpreters don't allow for it anyway. But given that it is a convention in Ruby that is so widely ignored, its greatest value might be your realization that you (should you happen to be working on Ruby code) have a capacity to care about something that others seem not to. That's not the same as "mastery" or "intelligence," and certainly not something to berate someone about. But it might be helpful to see which library authors care and don't care about these ideas.

Interestingly enough, `!` and `?` are used for completely different purposes in the Swift programming language.

For this chapter, we're taking a deliberate, hard line and saying destructive actions are not okay. If you *really* want to avoid state, and have all of the nice things (idempotent, pure functions that can be tested, cached, and parallelized easily), this is the price of admission. It might seem like an expensive ticket, but if your choice is between this and a compiles-to-JS language, this approach is easier. Definitely try out a functional compiles-to-JS or standalone functional language at some point, though, if you're interested in going from self-discipline to compiler-enforced discipline.

Don't return null

Let's look at some code from the last section:

```
function emailSubject(){
  return weKnowName ? `Hi ${name}` : "Hi"
```

```
};
sendEmail(emailSubject(), emailBody);
```

If `someName` and `nullName` were both objects with a defined `toHi` function, we could write this as:

```
const someName = {
  value: "some name",
  toHi(){return `Hi ${this.value}`};
};
const nullName = {
  value: "",
  toHi(){return "Hi"};
};
// assuming getName will get someName or nullName
sendEmail(getName().toHi(), emailBody);
```

For further cleanup, in **Chapter 9** we discussed how we might wrap an object with a `null` object. We'll cover a similar but better (albeit initially somewhat esoteric) idea later in this chapter called `Maybe`, and its friend `Either`.

Referential Transparency and Avoiding State

Can data always be the result of a function call? Yes, but any functions relying on or manipulating nonlocal inputs (free variables) cannot be replaced by their return value without extra consideration. As we briefly covered earlier in this chapter, the ability to replace a function call with its return value (without changing the program's behavior) is called *referential transparency*.

As an example, `emailBody` in the last section could instead have been a call to an `emailBody()` function, right?

What about this?

```
x = 5;
```

That could be:

```
function five(){ return 5 };
five();
```

Or we could just call it right away:

```
(() => 5 )()
```

Or we could return what we pass in instead:

```
((x) => x)(5)
```

We can do a similar transformation for functions that actually calculate things. For example, this:

```
3 + 5
```

could be:

```
(() => 3)() + (() => 5)();
```

The first function returns 3. The second returns 5. Then they're added. Here a 3 is passed into the first function, which is returned. A 5 is passed into the second function, which is returned. Then the results of those are calculated:

```
((x) => x)(3) + ((y) => y)(5);
```

If we combine the two expressions, we get this:

```
((x, y) => x + y)(3, 5);
```

This might look odd, but it has a few features worth noting. First, the arguments we passed in are scoped only inside of this line. Second, this anonymous IFFE is starting to look a bit like Lisp with a few extra bits of syntax. Third, if we assign to what is returned here, we are assigning a numerical value:

```
const result = ((x, y) => x + y)(3, 5);
```

`result` is 8, the number. It's not a function. It's the result of a function call. If we're living in a stateless, side effect–free world (although it's also guaranteed by us using only explicit inputs), that means that we can replace any of the previous calculations that return 8 with just an 8. And *that* means that we can replace any instances of `result` with 8. We never have to do this calculation again with these parameters.

This is referential transparency in action.

Why should the parameters change anyway? Well, let's say we have this situation:

```
const recentLogins = ((x, y) => x + y)(db.loginsToday,
                                       db.loginsYesterday);
```

Now we're looking to the outside world to provide us with values that update in the database. We have a huge ball of state called a database. We don't

have to use a database as a place to put lots and lots of variables. Just like in our program, we could be creating new values, rather than updating them:

```
const today = // some specific date
const yesterday = // some specific date
const recentLogins = ((x, y) => x + y)(db.logins(today),
                                       db.logins(yesterday));
```

Disk storage is incredibly cheap, and lookups shouldn't be too bad if you can index by time in your database (especially given that you should only have to do the lookup once!). Additionally, if you're not overwriting (aka updating, aka throwing away) database information, you should be able to play back the state of your application given a particular query (this also assumes being able to retrieve the version of the code at the same time).

AREN'T WE JUST HARDCODING VARIABLES? WHAT ABOUT MY DYNAMISM?

Whenever we introduce a state change, we're creating a concept of time as well. There's a world before the state change, and a world after. Once you destroy information, it's much harder to consider what might have gone wrong or restore to a previous state.

Additionally, in high-traffic websites, real-time queries and page loads are optimized for the speed of interface availability,[1] not the speed of the delivery of every new version of every piece of information in the database. That's why caching exists. Trade-offs are made between delivering something that is fast enough, and something that is recent enough. We're not going to get into making `recentLogins` cache-aware, but the point is that by parameterizing the calls to `logins` and setting our database up with time in mind, we've giving `recentLogins` the opportunity to be cached by virtue of it being referentially transparent (for a certain time period). In reality, we'd probably want this to run more than once a day, but letting this (or something much more complex) block showing someone *something* (even if it's a little stale) would not make for a good experience.

We got a little off track there, but the main point is that nothing has to be in a variable (other than the global defaults) for a program to work. Instead of creating objects, we can have functions that *store* values. Our entire program could be functions wrapping other functions, all kicked off by a single parameter to start the data flowing.

Our program *could* look a lot like Lisp. Our program *could* look a lot like XML with parentheses. This is the central concept in Lisp, by the way. The abstract syntax tree (AST) to describe how the program works is, in most languages, in a somewhat different form than the source code. In Lisp, it's the same. This is called *homoiconicity*.

JavaScript does not share this feature with Lisp, but you can get far closer with some coding styles than others. In any case, there is clearly some link between state, interface, functional programming, and lots of parentheses. The main point of this section is that we have a lot of options and flexibility when it comes to how data enters and is used by our program, and these variations matter beyond syntax.

1 Just kidding—websites are mostly prioritized to serve ads, load a large "hero" graphic (or video! how neat!) after a few seconds, and break your ability to fluidly scroll through the content of a page.

Handling Randomness

In **Chapter 4**, we dealt with randomness mostly by avoiding thinking about or testing it. We're not going to go too in-depth here either, but in the context of functional programming, we need to address that randomness makes our functions *impure* (meaning we cannot rely on them to yield the same output when given the same explicit input). In JavaScript, we call `Math.random` and we get a "random" value. But what we're actually getting is a "pseudorandom" number, which is calculated based on some "seed" value, usually having something to do with time. Unfortunately, this seed value cannot be set in our native `Math.random` function.

If it could be set with a seed, we could generate the *same* "random" value sequence. For instance, it could work like this (but not in native JavaScript):

```
const mySeed = 3;
let rng = setRng(myseed);
rng.gimmeRandom();
// returns 2593
rng.gimmeRandom();
// returns 8945

rng = setRng(myseed);
rng.gimmeRandom();
// returns 2593
rng.gimmeRandom();
// returns 8945
```

That means that even functions that appear random are actually deterministic. This is handy, because you can have random-like numbers that you can test against. It is possible that for your application (or just your tests), this kind of randomness is enough.

Keeping the Impure at Bay

There are three functions in here (not including tests). Can you pick out the impure ones?

```
const test = require('tape');
const testdouble = require('testdouble');
var x;

function add(addend1, addend2){
  return addend1 + addend2;
```

```
};

function setGlobalFromAddition(addend1, addend2){
  x = add(addend1, addend2);
}

function readAddition(addend1, addend2, done){
  console.log(add(addend1, addend2));
  done();
}

test('addition', (assert) => {
  assert.equal(add(2, 3), 5);
  assert.end();
});

test('setting global', (assert) => {
  setGlobalFromAddition(2, 3);
  assert.equal(x, 5);
  assert.end();
});

test('setting global again', (assert) => {
  setGlobalFromAddition(2, 8);
  assert.equal(x, 10);
  assert.end();
});

test('calling console', (assert) => {
  testdouble.replace(console, 'log');
  readAddition(2, 3, () => {
    testdouble.verify(console.log(5));
    assert.pass();
    testdouble.reset();
    assert.end();
  });
});
```

`addition` is pure because it relies only on its inputs, and returns an output without having any side effects. It is also completely trivial to test. Good job, `addition`.

As for the other two functions, we decided to test them even though they're impure. `setGlobalFromAddition` seems easy enough to test. The second and third tests run fine. But what if instead of a "dreaded evil global variable," this was a value in a database? And what if other people were testing their code using the same database? Obviously, we should expect occasional failures if we're depending on global shared state and all trying to use the same datastore, right? More realistically, whether it's global state in code or a database, if the

number of tests we need to run is large enough, we're going to want to parallelize our test suite. And then, we will see failures. Impure functions rely on something other than their explicit inputs, and that includes not just nonlocal variables (and functions if they themselves are impure), but also databases and the implicit `this` argument (if it is not immutable). That last point bears repeating, as it is a pretty big shift from how object-oriented programming tends to work: in functional style, we should prefer using explicit inputs and outputs to accessing and modifying a `this` value.

All in all, functions like `setGlobalFromAddition` will make our testing more difficult. Maybe not immediately, but even without a more complex program that mutates `x` in multiple places, we already can see tests that cannot be trusted to run in parallel.

The third function, `readAddition`, makes a call to `console.log`, so it is an impure function with a side effect in it. The complexities of testing this were explored in the previous chapter, but in the context of this chapter we have another reason, beyond the mechanics of testing, for why working with this is inconvenient: I/O calls are side effects and make impure functions. Impure functions are hard to test.

If you like, you could try namespacing your impure functions (don't forget to namespace the function calls as well). A simple object would work:

```
const impure = {
  setGlobalFromAddition(addend1, addend2){
    x = add(addend1, addend2);
  },
  readAddition(addend1, addend2, done){
    console.log(add(addend1, addend2));
    done();
  }
};
```

If you find doing this separation annoying, you really wouldn't like the machinations you have to do to mix pure and impure code in some functional languages (like Haskell).

Hopefully, this section has made three things clear. First, it is easy to tell pure from impure functions. If you know that something will always run the same way, with the same input, and not produce a side effect (meaning anything that doesn't simply return a value), then it is pure. Second, impure functions are harder to test. Third, the simpler (in terms of bulk, inputs, and outputs) your functions are, the better chance they have at being pure.

At this point, we've talked over some of the basic hygienic processes you can enact in your code to make it more functional, and also discussed some of the benefits of doing so, including purity, referential transparency, and easier test-

ing. The approaches we've taken and our goals for the code are not unlike those found in the rest of the book. We've already covered a preference for less bulky functions with few inputs and one clear return value. We've also covered why array comprehensions make more sense than `for` loops. Reassigning variables has been discouraged plenty of times earlier.

Now that we have a new perspective on some of our earlier concepts, it's time to branch out. Next we'll be looking at some features that can give us more functional interfaces.

Advanced Basics

In this section, we'll be exploring some fundamental aspects of functional programming that are still foreign to many JavaScript programmers:

- Currying
- Partial application
- Function composition
- Types

WHAT YOU CAN'T GET FROM JUST PLAIN JAVASCRIPT

Having a compilation step and a type safety system in place (which JavaScript doesn't) is actually pretty handy. Using a compiles-to-JS language like TypeScript or Flow will help you realize some of those benefits. Alternatively, you could just go straight for Haskell, Clojure, or Scala.

Currying and Partial Application (with Ramda)

One functional concept that you can exploit to make your code more flexible is using currying and partial application. Before explaining those terms in the text, let's look at a bit of code:

```
function add(numberOne, numberTwo){
  return numberOne + numberTwo;
};
add(1, 2);
// 3
```

This is our favorite addition function so far. It works beautifully. But some (FP rapscallions) would have you believe that this one is better:

```
function add(numberOne, numberTwo){
  return function(numberOne){
```

```
    numberOne + numberTwo;
  };
};
```

There is some weirdness here:

```
console.log(add(1, 2));
// [Function]
```

Did we break our perfectly good addition function? Yes.

Now it just makes a function that doesn't care about that second argument (2) at all. Or does it?

We can still use `add` like this:

```
console.log(add(1)(2));
// 3
```

We just replace the `,` with `)(`, and then we're good. But that doesn't explain how it works or why we'd want to do that.

Well, the first thing to notice in our second addition function is that it returns a function. That means that when you run `add(1)`, you get back a function that is waiting for the 2 (or something else):

```
const incrementer = add(1);
incrementer(2);
// 3
```

Our `incrementer` is a partially applied function. We took our original function that took two arguments, and partially applied one argument that it needed, creating a new function with reduced *arity* (the number of arguments something takes; e.g., a function with an arity of 1 takes one argument).

Currying is not the same thing as partial application. This was the partial application part:

```
const incrementer = add(1);
```

We happened to assign it to a variable, but the partial application was when we did this: `add(1)`. Currying is what we did when we transformed our normal two-parameter function (i.e., a binary function) into one that took one argument (i.e., a unary function). What we didn't do, however, was use a `curry` function to do that. We curried it manually. As of right now, we don't have a way to take *any* function of *any* arbitrary arity and curry it.

For that, we need Ramda. The underscore and lodash libraries do similar things, but Ramda has the coolest name and graphic. Also, underscore doesn't

have the `curry` function. Comparing and contrasting the libraries is an exercise left to the reader, so let's continue by installing Ramda:

```
npm install ramda
```

Next, let's try it out:

```
R = require('ramda');

function add(numberOne, numberTwo){
  return numberOne + numberTwo;
};

const curriedAdd = R.curry(add);

console.log(curriedAdd(1));
console.log(curriedAdd(1)(2));
console.log(curriedAdd(1, 2));
```

It's pretty easy to use. We just require it and write our binary function as we normally would, and then `R.curry` does some magic to it:

```
[Function]
3
3
```

Our output is better than expected. Where our normal (uncurried) function would choke on the first one (giving us `NaN` by adding `1` to `undefined`), this gives us a partially applied function. Supplying `(1)(2)` as arguments gives us `3`, just like we got from our manual currying process. As a bonus, the curried function still works with our original arity of 2. Passing two arguments with `(1, 2)` still works fine!

Also, the following works as expected:

```
const increment = curriedAdd(1);
console.log(increment(3));
```

How would we have built `increment` if it wasn't curried? Maybe like this?

```
function increment(addend){
  return add(addend, 1);
};
console.log(increment(3));
```

It's only two more lines in this case, but what if we had `increment`, `addFive`, `addTen`, and so on? Or what if the implementation were more complex?

In case you're not quite on board with currying, here's another possibility. You know that `map` function we've been praising throughout the book? What if I told you it wasn't that awesome?

Here's something that works okay:

```
const square = (thing) => thing * thing;

console.log([2, 4, 5].map(square));
```

But what if you want to reuse a `mapSquares` function? Our data is stuck in front of the dot (`.`), so what should we do? Altering `Array.prototype` is strictly a no-no. We could create a new class that `extends Array`, alter *that* prototype, and make our `[2, 4, 5]` an instance of that subclass.

Or we could let Ramda do some work:

```
R = require('ramda');
const square = (thing) => thing * thing;

const mapSquares = R.map(square);
console.log(mapSquares([2, 4, 5]));

console.log(R.map(square, [2, 4, 5]));
```

Ramda's `map` function is already curried, so we can partially apply `square` to make `mapSquares`, later applying the data. But it also allows for a convenient, two-argument version if we feel like "fully applying" the function all at once.

That's only two of Ramda's functions (`curry` and `map`), and we're already seeing more flexible, reusable, and shorter interfaces.

WHAT ABOUT THIS?!

Isn't `this` the most important keyword in JavaScript? And now, Ramda is just throwing it away? Without a doubt, `this` generates more confusion, questions, and blog posts than nearly any other topic, with the possible exceptions of "Prototypes are good/bad" and "What library should I use for X?"

But that's not the same as being "important." If we can get this kind of flexibility and clarity, then `this` is our sacrifice. Not only is determining what `this` is a bit confusing, but it also ties up what could be explicit arguments. Ramda just acts like a namespace, which is closer to how "real" functional programming works.

> I might even go as far as saying that using `this` makes your functions "impure." If you don't want to, that's okay, but isn't the `this` context just shared mutable state?

Function Composition

Okay, so we saw that Ramda was pretty awesome, but you might be thinking that lodash and underscore will work just as well for composing functions.

Let's give that a shot with the last example (after you run **`npm install lodash`**):

```
_ = require('lodash');
const square = (thing) => thing * thing;

const mapSquares = _.map(square);
console.log(mapSquares([2, 4, 5]));
console.log(_.map(square, [2, 4, 5]));
```

Kaboom:

```
TypeError: mapSquares is not a function
```

We can still do this just fine:

```
console.log(_.map([2, 4, 5], square));
```

It works, but wait a second...that's backward!

Yes, it is—and traditional. And that tradition of putting data before the callback is an impedance to composing functions. We can fix it:

```
function mapSquares(data){
  return _.map(data, function(toSquare){
    return toSquare * toSquare;
  });
}
console.log(mapSquares([2, 4, 5]));
```

Admittedly, we can refactor to:

```
function mapSquares(data){
  return _.map(data, square);
};
```

And to:

```
const mapSquares = (data) => _.map(data, square);
```

But ugh...before, we got away with just this:

```
const mapSquares = R.map(square);
```

POINT-FREE PROGRAMMING

In this chapter, we're aiming for "point-free" style most of the time. That means not working directly with your inputs. You can apply functions to them, but if you give the input a name (like `data`) and work with it directly, you've created a *point*. The word "point" is related to topology, and not the dot (`.`) in JavaScript.

In the preceding example of `mapSquares`, this one is "pointed":

```
const mapSquares = (data) => _.map(data, square);
```

And this is point-free:

```
const mapSquares = R.map(square);
```

It's difficult (maybe impossible) to make all of your functions point-free, but attempting to do so should help shorten a lot of your function definitions.

It's worth noting, however, that not everything will be easy.

Earlier, we had this bit of code:

```
[3, 4, 2].forEach((element) => console.log(element));
```

It might look like we could just do this instead:

```
[3, 4, 2].forEach(console.log);
```

But this is actually different. Although we only want to print the number, we'll end up printing:

```
3 0 [ 3, 4, 2 ]
4 1 [ 3, 4, 2 ]
2 2 [ 3, 4, 2 ]
```

That's the number, the index, and the full array. `forEach` will supply three parameters to a callback function if it wants them. `console.log` will happily print as many arguments as it is passed, but we can make a new function based on `log` that only uses its first argument:

```
const logFirst = (first) => console.log(first);
[3, 4, 2].forEach(logFirst);
```

This topic goes pretty deep. There are functions to turn any function into a unary function (as we did here), as well as functions of another arity. You can reorder parameters as they come in, and create functions that are partially applied with placeholder arguments in arbitrary positions in the function signature.

The bottom line is, if you do a lot of function composition and strive for point-free, odds are that you'll end up fiddling with your parameters a bit.

Hopefully, everyone will get together and decide that data first is a problem. But in the meantime, it's just one level of working with data directly when we have the wrong argument order, right? At first, yes. But as with conditionals, `null` returns, and every kind of bulk in the code, complexity seems to breed when no one's watching. More practically, you'll see more nesting (and data variables) if you compose your functions with a data-first library.

As you add functionality, you'll either be making your code more complex (in the function calls or the function definitions) or composing new functions. Let's see that in action. If we wanted to raise everything to the fourth power, we could complicate the function call and be done:

```
R = require('ramda');
console.log(R.map(square, R.map(square, [2, 4, 5])));
```

Or we could do something like this:

```
function fourthPower(thing){
  return square(thing) * square(thing);
};
console.log(R.map(fourthPower, [2, 4, 5]));
```

But Ramda has another trick up its sleeve, called `compose`. Check this out:

```
const fourthPower = R.compose(square, square);
console.log(R.map(fourthPower, [2, 4, 5]));
```

And we could combine that with `map` into a new `mapFourthPower` as well:

```
const mapFourthPower = R.map(fourthPower);
console.log(mapFourthPower([2, 4, 5]));
```

And yes, we could actually compose with `console.log` too (if we wanted to log the elements, rather than the array):

```
const printFourthPower = R.compose(console.log, square, square);
R.map(printFourthPower, [2, 4, 5]);

// or because map is curried, this works too
R.map(printFourthPower) ([2, 4, 5]);
```

So we can choose tiny function definitions and tiny function calls. At worst, we might end up with a few extra function calls around. Although it might be tempting to just have one long chain of `compose` and `map` calls, debugging those can be tough. Ramda helps make functions very easy to create and compose. Even though it's easy to chain things together too, it's worth striking a balance of how complex your composed functions should be. Smaller composition functions (just like any smaller function) are easier to test and reuse.

MEMOIZE REVISITED WITH RAMDA

By the way, this is what a memoized factorial function looks like in Ramda:

```
var factorial =
  R.memoize(n => R.product(R.range(1, n + 1)));
```

Apologies if you spent much time trying to learn the long version.

Another function you'll see from time to time is called `pipe`. It works like this:

```
var factorial = R.memoize(n => R.product(R.range(1, n + 1)));
var printFact = R.compose(console.log, factorial);
printFact(3);

// is the same as

var factorial = R.memoize(n => R.product(R.range(1, n + 1)));
var printFactPipe = R.pipe(factorial, console.log);
printFactPipe(3);
```

All in all, Ramda is great, and it exposes the types of functions that make certain interfaces very simple. If you look at the **docs** (http://ramdajs.com/docs/), however, you might be surprised at a few things. One is how gigantic it and other functional libraries are. A facade (**Chapter 9**) extracted for beginners would be nice.

Types: The Bare Minimum

Before we get to types, one thing that might seem scary about functional code is that if you don't extract any functions, you could write something like this to accomplish the same work we did in the previous chapter:

```
console.log(R.map((thing) =>
  thing * thing, R.map((thing) =>
    thing * thing, [2, 4, 5])));
```

And Ramda or not, we've recreated another callback hell (with an unfortunate reuse of a bad variable name and two traversals of the array to boot).

Fortunately, we know how to extract functions, whether we're doing FP, OOP, or plain old imperative coding. Extracting functions (maybe after decent naming) is our first line of defense against confusing and untestable code.

THE CASE FOR TYPES

Functional programming has another defense against confusing code: types. When you pass functions to other functions, and sometimes they return the execution of that function and other times they return a function literal, things can get confusing. If you were writing Haskell, you'd deal with this confusion up front. Unlike JavaScript, Haskell actually cares what arity and types you call functions (or compose them) with. So if you try to compose two functions where one would pass a string to the other, which expects a number for input, you'd get a compilation (type) error. The same goes if you try to do something else weird, like this:

```
console.log(Math.random("beaver"));
// returns 0.21801069199039524 (at least this time, apparently)
```

Why didn't this raise an error? It was almost as if an occult hand threw away that argument. This is fine in JavaScript. It's normal. No compilation means no compilation error. And there's no runtime error either, because JavaScript functions just don't care what you give them, unless their function bodies have some opinion.

The inconsistencies in basic operations of JavaScript are well documented, and opinions vary on whether all of the weirdness is a comedy or a tragedy. The reliance on either third-party specs or a far too dense, huge, and abstract official spec doesn't help matters.

In any case, what good library writers and third-party documenters of core functionality do is list and describe objects, functions, and other random bits,

maybe with a tutorial or two sprinkled in. The most crucial piece of this is describing inputs and outputs for functions.

One could argue that this documentation should be done anyway, and therefore, having types of inputs and outputs enforced by the language itself would be a good thing. Since someone has to write it, it might as well be written in the code itself, right?

In Haskell, and more and more frequently in documentation by JavaScript library writers who might wish JavaScript were a little more like Haskell, types are described by the Hindley–Milner type system. Most JavaScript documentation doesn't use this system, but it does show up in the docs for Ramda and Sanctuary (we'll cover this library later).

It looks something like this:

```
add :: Number -> Number -> Number
```

There is much more to it, but these are the most popular parts. `add` is the function name, followed by two inputs of type `Number`, and one output `Number`. But wouldn't it be clearer like this?

```
add :: (Number, Number) -> Number
```

In a way it would, but assuming `add` is curried (which is a good assumption for Haskell and aspirational JavaScript as used in Ramda), we should think of applying (calling the function with) the first `Number` as producing a new function that waits for the second number. For example, if we applied a `1`, we could have a new function with the type:

```
addOne :: Number -> Number
```

We covered this earlier when our `addition` function turned into an `addOne` function after we gave it only one parameter (a `1`) instead of two parameters right away. If we give `addOne` another number, it will output a number.

So there are two points here. The first is that in some programming, the assumed arity (how many parameters are passed in) is *1*, because things are understood to be curried by default.

The second point is that type declarations don't just document what functions do, they also indicate how functions can collaborate with functions. So if you know that a function produces a number, and you use its output as input to a function that takes a string, then you shouldn't be surprised if something goes wrong (a compilation or runtime error in some languages and libraries, or maybe just a confused result in JavaScript).

LESS INTUITIVE ASPECTS OF TYPES

On the seemingly less awesome side of type documentation (and code itself), the functional style prefers short variable names, which can be a bit of a shock. Take a look at the type signature for Haskell's `map` function (specifically for lists):

```
map :: (a -> b) -> [a] -> [b]
```

The grouping of the parentheses `(a -> b)` indicates that `map` takes a function as its first argument, which itself takes one input (`a`) and has one output (`b`). `map` also takes an array of `[a]`. Then it outputs an array of `[b]`. The `a` and `b` seem like excellent candidates for very bad names (short and nondescriptive), but there is a logic behind them.

`map`'s first parameter is a function that transforms an `a` into a `b`. It applies that function to a list full of `a`s, and transforms it into a list of `b`s, which it returns. All it's describing is what *needs* to remain the same, and what *may be* different. A function that transforms a number into a boolean must be used to transform a list of numbers into a list of booleans. But `map` doesn't care what type `a` and `b` are specifically.

When you declare a variable or constant, that is different because you're labeling a concept of an actual specific value. Here, the type variables are just indicating what is the same and what is different. This is common with higher-order functions like `map`, where the types `a` and `b` actually apply to many types. By contrast, note that in the previous `add` function, the inputs and outputs were specific to a `Number` type.

It can get a little complicated. This is the type description from the `concat` function of `Maybe` in Sanctuary:

```
Maybe#concat :: Semigroup a => Maybe a ~> Maybe a -> Maybe a
```

The `Maybe#` indicates the `this` object (an object of type `Maybe`) that has the `concat` function available. The `Maybe a` parts describe normal inputs and outputs, and `a` type variables could be anything, but they're all the same and wrapped up in `Maybe`s. This is kind of like what we saw before when list items were wrapped up in arrays with `[a]` and `[b]`.

We have two types of arrows in `Maybe a ~> Maybe a -> Maybe a`, though. Working from the right, the last arrow is `->`, which isn't new. It's just indicating input headed for output. The first one is `~>` (squiggly arrow), indicating that an implicit parameter (the `this`) is also in the mix.

BUT ISN'T THIS BAD?

Unlike in OOP, in functional programming, it's much more common to avoid `this` in favor of passing explicit parameters. Where we use `this` in this chapter, it is done mostly as a way to namespace functions. That goes for `R` (for Ramda functions) and `S` (for Sanctuary functions and objects).

One key thing to note is that these variables provide a namespace and utility functions, but they aren't intended to hold state "variables" that are changed.

In OOP, stateful objects (with mutable values) are arguably better than scattered nonlocal variables, but in FP, we should consider `R` and `S` to provide the same utility as *modules*, rather than *classes* or *objects*.

The `Semigroup a =>` bit is new too. That just means that this function (`concat` of `Maybe`) is implementing a function from the `Semigroup` typeclass. We're not going to get into typeclasses. There is no decent analogous structure for a typeclass in JavaScript. It could be thought of as an *interface* in some OOP languages, or as a contract that must be fulfilled. In **Chapter 9**, we discussed empty parent classes as a faulty but possible demonstration of this idea.

Anyway, if you spend some time in functional corners of JavaScript, you'll hear types discussed and see functions notated with their type signatures from time to time.

Burritos

This section might be about burritos. It might be about something else. I can't explain everything about burritos in a few pages. They're too complex and eating too many at once is a mistake. Indigestion and regurgitation full of new metaphors are both common results.

For the sake of your future investigations, *burritos* might involve any of the following:

- Monoids
- Functors
- Applicatives
- Monads
- Maybe more?!

Wow. I get a red underline marking all of those as misspellings. We're in serious jargon territory, apparently. They're complicated, and if this is your first time with them, no explanation is going to work better than some actual code. And this is why people are so quick to explain these things by analogy, and

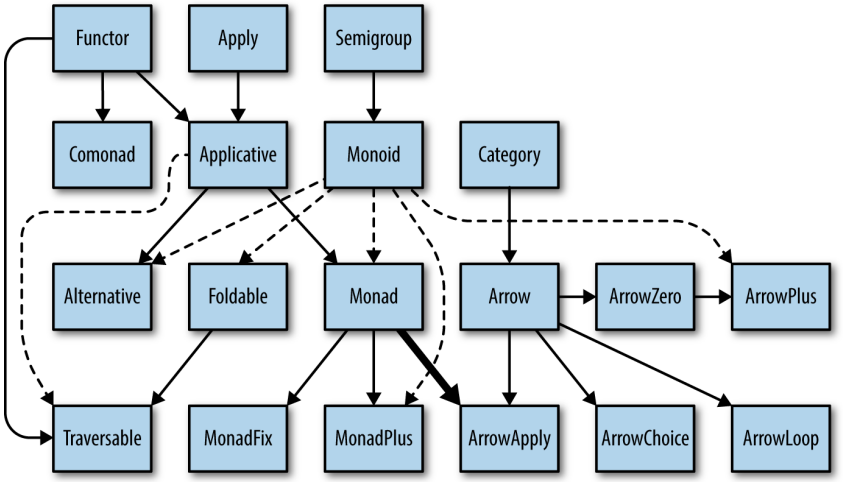
often call anything related to them "burritos." Sometimes this approach comes from someone who doesn't want to explain everything that leads to the practical stuff they want to cover (like me), or as an out for people who feel they don't know enough about functional programming or deep mathematical category theory to really do the topic justice (like me).

The real problem is, people can "get" the concept of a "queue" (for instance) without much difficulty. It's just one thing that's conceptually easy, and the real-life analogs (e.g., lines of people waiting for something) are both clear and indicated by the name of the thing. The same goes for "stack." This is, however, less the case for "linked lists" and "Bloom filters."

If you try to understand every burrito-like thing (functional *abstract data types*, or ADTs) at once, you're up against something like **Figure 11-1**.

FIGURE 11-1

Haskell burritos



Monads are usually the burrito in question. And yes, you could try to start there, but all those arrows in the diagram point from "simple thing" to "more complex thing," which indicates that monads are pretty complex. So if you want a deep dive into this stuff, try starting at the functor (**Figure 11-2**). You can definitely understand that one as easily as a queue or a stack. Basically, a functor has a `map` function and some rules. In fact, since you're certainly already using `map` with arrays, you're not starting from scratch here.

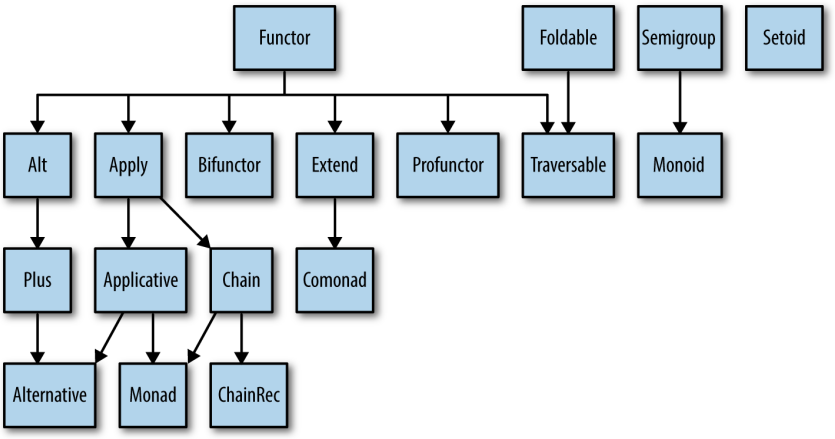


FIGURE 11-2

The Fantasy Land specification for burritos in JS

AN ACTUAL EXPLANATION

So there are these words that fly around, like *functors* and *monads*. They're intimidating at first because they're abstract, and it's common to see them explained in terms of other things you might not understand. They are abstract data types (ADTs), just like linked lists and stacks. They are defined by the operations you can perform on them and rules for how they can be used.

A *stack* isn't a thing you use directly, but you can build something that implements a stack interface (i.e., it has `push` and `pop` functions), according to the rules of how a stack should behave (i.e., it stores things in a "last in, first out" kind of way).

Similarly, you can create all the scary-seeming functional ADTs (probably using objects in JavaScript's case, but maybe just collections of functions) that implement the interface of "abstract data types" according to their rules.

My apologies if this is a letdown, but the goal of this section of this chapter is to give you some concrete experience with useful functional APIs. If you want some further reading on functional programming and functional ADTs, I'll keep **a list** (http://github.com/evanburchard/burritos) updated with some videos, tutorials, and books about burritos as I find them.

Introducing Sanctuary

Speaking of a useful API, install Sanctuary on the command line:

```
npm install sanctuary
```

Then in your program, do this:

```
const {create, env} = require('sanctuary');
const S = create({checkTypes: true, env: env});
```

You're left with an S object. Like other functional libraries, Sanctuary has a massive interface that might be intimidating. As of this writing, there are over a hundred properties defined on S. A lot of them (`compose`, `inc`, `pluck`, etc.) should look familiar after our time with Ramda. There is a bit of a difference, however. With Ramda, adding `"hello"` and `3` just returns `NaN`:

```
R = require('ramda');
R.add("hello", 3);
// NaN
```

With Sanctuary:

```
const {create, env} = require('sanctuary');
const S = create({checkTypes: true, env: env});
S.add("hello", 3);
```

you get this:

```
TypeError: Invalid value
add :: FiniteNumber -> FiniteNumber -> FiniteNumber
       ^^^^^^^^^^^^
            1
1)  "hello" :: String
The value at position 1 is not a member of 'FiniteNumber'.
```

And with a stack trace. Sanctuary has runtime type-checking errors (for the functions it has defined). That's pretty awesome, but possibly not what you'd want in production. If that's the case, you can initialize it without type checking like this:

```
const {create, env} = require('sanctuary');
const S = create({checkTypes: false, env: env});
```

Now Sanctuary won't type-check its functions, but be careful:

```
S.add('hello', 3)
// 'hello3'
```

That's default JS behavior, but Ramda and Sanctuary not being in agreement on nonstandard behavior could cause some confusion if you're using both. Also, just like lodash and Ramda have different interfaces (Ramda sensibly puts the function first), Ramda and Sanctuary also don't always have the same functions or the same interfaces for those functions that they share.

Regardless, Sanctuary has a lot of functions that are good for functional programming (like Ramda), and it even has a type checker if we want one. Awesome. Also like Ramda, the parameters for its higher-order functions (like `find` and `reduce`) follow the "function-first" idea, so it's easy to compose functions like this:

```
const getAThree = S.find(x => x === 3);
```

Or point-free:

```
const getAThree = S.find(R.equals(3));
```

Either way, we can apply the array to `getAThree`:

```
getAThree([3, 4]);
```

And then something weird happens. We don't just "get a three," we get:

```
Just(3)
```

And for an array without a 3 in it:

```
getAThree([8, 4]);
```

we get:

```
Nothing()
```

SANCTUARY'S MOTTO

Sanctuary's motto is "Refuge from unsafe JavaScript." It prefers giving `Nothing` and `Just` rather than `null`. This might seem awkward to handle, but it's much better than being surprised by "undefined method" errors.

The null Object Pattern, Revisited!

These `Nothing` and `Just` values from Sanctuary should remind us of something. We're actually working with a similar concept to `null` objects (covered in

Chapter 9). `Just` and `Nothing` will remain a mystery for just a bit longer, but let's see how they might apply to our `Person`/`AnonymousPerson` code.

Let's see what we had when we used a decorator to wrap our `null` objects:

```
class Person {
  constructor(name){
    this.name = new NameString(name);
  }
};
class AnonymousPerson extends Person {
  constructor(){
    super();
    this.name = null;
  }
};
class NameString extends String{
  capitalize() {
    return new NameString(this[0].toUpperCase()
               + this.substring(1));
  };
  tigerify() {
    return new NameString(`${this}, the tiger`);
  };
  display(){
    return this.toString();
  };
};
class NullString{
  capitalize(){
    return this;
  };
  tigerify() {
    return this;
  };
  display() {
    return '';
  };
};
function WithoutNull(person){
  personWithoutNull = Object.create(person);
  if(personWithoutNull.name === null){
    personWithoutNull.name = new NullString;
  };
  return personWithoutNull;
};
```

Not bad. It cost us quite a few lines, but we kept the billion-dollar mistake (`null`) at bay.

But using `Just` and `Nothing` provide a really easy way to do this. Here's what our new implementation looks like:

```
const {create, env} = require('sanctuary');
const S = create({checkTypes: false, env: env});

class Person {
  constructor(name){
    this.name = S.Just(name);
  }
};
class AnonymousPerson extends Person {
  constructor(){
    super();
    this.name = S.Nothing();
  }
};
const capitalize = (string) => string[0].toUpperCase()
                               + string.substring(1);
const tigerify = (string) => `${string}, the tiger`;
const display = (string) => string.toString();
```

Starting from the bottom three functions, instead of messing around with specialty forms of string objects with functions implemented on them, we opt for functions that don't rely on an implicit `this`, and we'll pass our strings as explicit inputs.

We completely got rid of our `String` extended classes. Instead of setting the names of our `Person` and `AnonymousPerson` to new instances of those extended classes, our constructors assign `name` to what we can think of as a *wrapped* `name` (for `Person`) or a *wrapped* `null` (for `AnonymousPerson`).

Unsurprisingly, our original tests will fail:

```
test("Displaying a person", (assert) => {
  const personOne = new Person("tony");
  assert.equal(personOne.name.capitalize().tigerify().display(),
               'Tony, the tiger');
  assert.end();
});
test("Displaying an anonymous person", (assert) => {
  const personTwo = new AnonymousPerson(null);
  assert.equal(WithoutNull(personTwo).name.capitalize().tigerify()
    .display(), '');
  assert.end();
});
```

The first failure we get is:

```
assert.equal(personOne.name.capitalize().tigerify().display(),
                               ^

TypeError: personOne.name.capitalize is not a function
```

This is because we no longer have a "fluent interface" of string functions returning strings to chain with another function. We need a way to apply a function to a `Just` or `Nothing`, then rewrap the value and pass it along. To do this, we use `map`:

```
test("Displaying a person", (assert) => {
  const personOne = new Person("tony");
  assert.equal(personOne.name.map(capitalize).map(tigerify)
    .map(display), 'Tony, the tiger');
  assert.end();
});
test("Displaying an anonymous person", (assert) => {
  const personTwo = new AnonymousPerson(null);
  assert.equal(personTwo.name.map(capitalize).map(tigerify)
    .map(display), '');
  assert.end();
});
```

This is the hard part. When you call `map` on a `Just` or a `Nothing`, the function will apply to the values inside of `Just`, but `Nothing` will pay no attention to the function applied through `map`. `Nothing` will just pass along another `Nothing`.

WHAT? I THOUGHT MAP WAS FOR ARRAY!

That is definitely the case in most programming you'll likely do in JavaScript. However, in the functional world, `map` has another role of applying a function to objects of various kinds. It might be hard to unify the ideas of "mapping over an iterable" and "mapping over `Just` or `Nothing`," but after working with it a bit, you should find it clearer.

If that doesn't make sense, don't worry. For now, it's okay if you think of it as a completely new type of `map` that has nothing to do with the `map` that works on iterables. This new kind "unwraps, applies a function, and rewraps" `Just` and `Nothing`.

There's another explanation here, and if it doesn't sink in yet, that's okay. One list-based type signature for `map` is this:

```
map :: (a -> b) -> [a] -> [b]
```

And the signature for `map` as related to `Just` and `Nothing` could be the following:

```
map :: (a -> b) -> Maybe a -> Maybe b
```

We'll discuss this in a bit, but `Just` and `Nothing` are both types of `Maybe`.

We're getting closer, but our tests now fail like this:

```
// for personOne
expected: 'Tony, the tiger'
actual:   Just("Tony, the tiger")

// for personTwo
expected: ''
actual:   Nothing()
```

Interestingly enough, our display function no longer actually does anything. Mapping over a `Just('String')` with `toString` still gives us `Just('String')`, and `Nothing()` stays as `Nothing()` as well. The tests are still failing, but we have a bit less code to deal with:

```
...
assert.equal(personOne.name.map(capitalize).map(tigerify),
             'Tony, the tiger');
...
assert.equal(personTwo.name.map(capitalize).map(tigerify), '');
...
```

So how do we unwrap our `Just` and `Nothing`? We can use the `S.maybeToNullable` function from Sanctuary:

```
test("Displaying a person", (assert) => {
  const personOne = new Person("tony");
  assert.equal(S.maybeToNullable(
                personOne.name.map(capitalize).map(tigerify)),
              'Tony, the tiger');
  assert.end();
});
test("Displaying an anonymous person", (assert) => {
  const personTwo = new AnonymousPerson(null);
  assert.equal(S.maybeToNullable(
                personTwo.name.map(capitalize).map(tigerify)),
              '');
  assert.end();
});
```

This gets us closer. Our first test passes, but our `personTwo` test fails like this:

```
expected: ''
actual:   null
```

While `S.maybeToNullable` unwraps the `Just` without a problem, it gives us the billion-dollar mistake for our `Nothing`. If you look at what's actually happening, it's not too surprising:

```
S.maybeToNullable(S.Nothing());
```

We should expect that to return `null`. We will take care of the failing test in a minute, but first, "`maybeToNullable`"? What's a `Maybe` anyway? Try these:

```
Object.getPrototypeOf(S.Just());
Object.getPrototypeOf(S.Nothing());
```

They both return this for their prototype:

```
Maybe {
  '@@type': 'sanctuary/Maybe',
  ap: [Function],
  chain: [Function],
  concat: [Function],
  empty: [Function],
  equals: [Function],
  extend: [Function],
  filter: [Function],
  map: [Function],
  of: [Function],
  reduce: [Function],
```

```
  sequence: [Function],
  toBoolean: [Function],
  toString: [Function],
  inspect: [Function: inspect] }
```

So `Just` and `Nothing` are both some kind of `Maybe`? Is that a perfect academic description? No, but it's a workable one. Now let's fix the tests:

```
test("Displaying a person", (assert) => {
  const personOne = new Person("tony");
  assert.equal(S.fromMaybe('',
                personOne.name.map(capitalize).map(tigerify)),
               'Tony, the tiger');
  assert.end();
});
test("Displaying an anonymous person", (assert) => {
  const personTwo = new AnonymousPerson("tony");
  assert.equal(S.fromMaybe('',
                personTwo.name.map(capitalize).map(tigerify)),
               '');
  assert.end();
});
```

Now our tests should be passing again!

Functional Refactoring with Maybe

Instead of `S.maybeToNullable`, we used `S.fromMaybe` to return a blank string (`''`) if the `Maybe` (`Nothing` or `Just`) is a `Nothing`. If it is a `Just`, it unwraps the value.

So if both names are actually `Maybe` all along, can we create the wrapper based on the input? Yes. And if the only difference between a `Person` and an `AnonymousPerson` is that they have different `name` values, we don't actually need the subclass anymore:

```
class Person {
  constructor(name){
    this.name = S.toMaybe(name);
  };
};
```

And in our tests, we can declare both objects as a `new Person`:

```
...
  const personOne = new Person("tony");
...
```

```
  const personTwo = new Person(null);
...
```

So what does our code look like now?

```
const {create, env} = require('sanctuary');
const S = create({checkTypes: false, env: env});

class Person {
  constructor(name){
    this.name = S.toMaybe(name);
  };
};
const capitalize = (string) => string[0].toUpperCase()
                                + string.substring(1);
const tigerify = (string) => `${string}, the tiger`;
const display = (string) => string.toString();

const test = require('tape');
test("Displaying a person", (assert) => {
  const personOne = new Person("tony");
  assert.equal(S.fromMaybe('',
                personOne.name.map(capitalize).map(tigerify)),
               'Tony, the tiger');
  assert.end();
});
test("Displaying an anonymous person", (assert) => {
  const personTwo = new Person(null);
  assert.equal(S.fromMaybe('',
                personTwo.name.map(capitalize).map(tigerify)),
               '');
  assert.end();
});
```

As far as refactoring, we can `compose` the `capitalize` and `tigerify` functions, which simplifies our assertions:

```
...
const capitalTiger = S.compose(capitalize, tigerify);
...
assert.equal(S.fromMaybe('', personOne.name.map(capitalTiger)),
             'Tony, the tiger');
...
assert.equal(S.fromMaybe('', personTwo.name.map(capitalTiger)), '');
...
```

But we have more possibilities. In order to not break the second test, we apply this only to the first assertion:

```
assert.equal(personOne.name.map(capitalize).chain(tigerify),
             'Tony, the tiger');
```

Instead of using two `map` functions and then wrapping with a `fromMaybe`, we can use `chain` to get an unwrapped value back from `tigerify`. Note that if `tigerify` itself returned a `Maybe`, that wouldn't work.

Better yet, we can use `chain` in combination with our composed `capitalTiger`:

```
assert.equal(personOne.name.chain(capitalTiger), 'Tony, the tiger');
```

By the way, in case you're wondering why we can't just call `personOne.name` as an argument to the `capitalTiger` function, here's the issue:

```
...
assert.equal(capitalTiger(personOne.name), 'Tony, the tiger');
...

// leads to this error

   expected: 'Tony, the tiger'
   actual:   'Just("tony"), the tiger'
```

`capitalTiger` is supposed to take a string, not a `Maybe`. Also, note that the type checker wouldn't save you here, because `capitalTiger` doesn't have its type declarations registered like the `S.` functions do.

Anyway, if we want to allow `capitalTiger` to access the `Maybe`, we can `lift` it, instead of mapping with `personOne.name.map(capitalTiger)`:

```
...
assert.equal(S.lift(capitalTiger, personOne.name), 'Tony, the tiger');
...
```

This has the same result as the earlier `map`, but is conceptually a little different. Instead of mapping over the `Maybe` with an ordinary function, you're promoting your function to work with the `Maybe`. You can think of it as making your function special ("lifted") rather than passing it as a dumb argument to `map` (which itself is already special).

In either case, you can also call `.value` on the `Just` to unwrap the value:

```
// take your pick

// map
assert.equal(personOne.name.map(capitalTiger).value,
```

```
            'Tony, the tiger');

// chain (unwraps the value, so no need for .value)
assert.equal(personOne.name.chain(capitalTiger), 'Tony, the tiger');

// lift
assert.equal(S.lift(capitalTiger, personOne.name).value,
            'Tony, the tiger');
```

As for the second test, we could create *another* function (probably composed with `capitalTiger`) that would test for `Nothing` and give us the empty string, but a better option is just using the lowercase `S.maybe` function.

It takes a value, a function, and a `Maybe`, and returns the value when the `Maybe` is not a `Just`. Otherwise, it returns the result of the function applied to the `Maybe`'s value. It's perfect for what we want:

```
assert.equal(S.maybe('', capitalTiger, personTwo.name), '');
```

But `reduce` would also work:

```
assert.equal(personTwo.name.reduce(capitalTiger, ''), '');
```

It ignores the function (because it is a `Nothing`) and returns `''`.

It's okay if the interfaces provided by Ramda and Sanctuary don't make perfect sense or are overwhelming. We covered a lot of ground here. What's nice is that you can incrementally add this style of programming to *your* JavaScript. Aside from the type signatures (which are good to know how to read, but you don't *need* to write), there's no new syntax to learn, and no compilation step necessary.

Functional Refactoring with Either

If you're not too confused by `Maybe`, you might be interested in `Either`. This section won't go into too much depth on it, but as far as motivation, our last test had this line:

```
assert.equal(S.fromMaybe('', personTwo.name.map(capitalTiger)), '');
```

The second `''` makes sense, as that's what we're testing against, but isn't the test a little bit late to be determining that our value should change from a `Nothing` into an empty string? What if we wanted to test it again? Would we have to keep specifying that value? Also, if we want to keep our tests symmetri-

cal, should we really have to specify the empty string for our `personOne` test as well?

Fortunately, there's a way to specify what we want our `Nothing` value to unwrap to before we're unwrapping it. Unfortunately, how to do so is a little complicated. Let's look at the code we would need:

```
const {create, env} = require('sanctuary');
const S = create({checkTypes: false, env: env});
const R = require('ramda');

class Person {
  constructor(name){
    this.name = S.maybeToEither('', S.toMaybe(name));
  }
};

const capitalize = (string) => string[0].toUpperCase()
                              + string.substring(1);
const tigerify = (string) => `${string}, the tiger`;

const capitalTiger = S.compose(capitalize, tigerify);

const test = require('tape');
test("Displaying a person", (assert) => {
  const personOne = new Person("tony");
  assert.equal(S.either(R.identity, capitalTiger, personOne.name),
               'Tony, the tiger');
  assert.end();
});
test("Displaying an anonymous person", (assert) => {
  const personTwo = new Person(null);
  assert.equal(S.either(R.identity, capitalTiger, personTwo.name),
               '');
  assert.end();
});
```

The new pieces here are:

- We're requiring Ramda.
- We assign our `name` property by converting the value to a `Maybe` and then an `Either`.
- Our tests are symmetrical and use `S.either`.

Let's talk about those last two bullet points. First, this line:

```
this.name = S.maybeToEither('', S.toMaybe(name));
```

As before, in the non-`null` case, `S.toMaybe` makes a `Just` that holds the `name` value that we pass in. Then, that `Just` is converted (through `S.maybeToEither`) into a `Right` that also holds the `name`.

In the `null` case for `name`, `S.toMaybe` creates a `Nothing`, which is the same as before. Then, `S.maybeToEither` turns that `Nothing` into a `Left`, and unlike `Nothing`, `Left` can hold a value. The value that it is set to hold is a blank string, `''`.

Because the `Left` is holding that value from the beginning, we can pull it out at the end in the same way we can pull out the value from the `Right`—namely, with these in the tests:

```
assert.equal(S.either(R.identity, capitalTiger, personOne.name), '');
assert.equal(S.either(R.identity, capitalTiger, personTwo.name), '');
```

The `S.either` function says: if it's a `Left`, apply the first parameter function to it (`R.identity`), and if it's a `Right`, apply the second parameter function to it (`capitalTiger`). That's *okay*, but it's also a bit prescriptive.

With a bit better of an understanding, we can use `map` instead. `Left` and `Right` are both `Either` subtypes. `Left` acts a little like `Nothing`, in that unlike `Right` (or `Just`), it will ignore when you try to map functions to it. That means our tests could also be the following:

```
assert.equal(personOne.name.map(capitalTiger).value, 'Tony, the tiger');
assert.equal(personTwo.name.map(capitalTiger).value, '');
```

This is more in line with how we expect `Left` and `Right` to behave. The `Right` (our `personOne.name`) goes along with the `map`'s function (`capitalTiger`) in the same way a `Just` would. The `Left` (our `personTwo.name`) disregards the `map`'s wishes, just like a `Nothing` would do with `map`. Unlike `Nothing`, however, the `Left` is holding onto a value (just like a `Just` or a `Right`). That means we can pull that attribute out with `.value` for both our `Left` and our `Right`.

Note that we couldn't use `map` followed by `.value` when `personTwo.name` was a `Nothing` because a `Nothing`'s `value` is actually `undefined` in Sanctuary.

As a final curiosity, you might think that this is a fairly complicated way to get our `Left` value:

```
this.name = S.maybeToEither('', S.toMaybe(name));
```

First we convert to a `Maybe`, and then to an `Either`. The reason is that a `Left` can hold a value, including `null`. To illustrate:

```
S.Either.of(null)
// returns Right(null) not Left(null)!

// vs.

S.toMaybe(null)
// returns Nothing()
```

As of this writing, you'll need to first get a `Nothing`, and then convert it to a blank string containing `Left` via:

```
S.maybeToEither('', S.toMaybe(name));
```

Based on what's in the main branch of Sanctuary, we should be seeing a more direct option that would let us do: `S.toEither('', name)`, but for now (Sanctuary version 0.11.1), we're stuck with the workaround. That said, if you're interested in checking out other functional libraries, **Folktale** (http://folktalejs.org) has an **`Either.fromNullable` function** (http://bit.ly/2mBZL1I) that would be perfect for this.

Before we leave this example completely, notice how short our `Maybe` and `Either` versions are in comparison to the `null` object wrapping code we made in **Chapter 9**. Admittedly, it's not quite a fair fight, because in the `null` object wrapper we had the artificial restriction of not touching certain parts of the code.

Learning and Using Burritos

You might be thinking that there are an enormous number of ways to use `Maybe` and `Either`. And that is true. APIs as big as `Maybe`'s can be intimidating. If you look at the **Sanctuary documentation** (https://sanctuary.js.org/), you'll see that `Maybe` implements the following ADTs from the "Fantasy Land" functional JS specification: monoid, monad, traversable, extend, setoid, and alternative. Implementing these also means that functor, apply, applicative, chain, foldable, and semigroup are implemented according to the dependency rules of the Fantasy Land spec.

The good news is, if you like that sort of thing, `Maybe` and `Either` have friends. A promise is similar to what is typically called a *future* or *task*. *List* is not unlike how you might use an array. There are a ton of other structures that implement these ADTs.

A public service announcement: monads (and their friends from Figures **11-1** and **11-2** and the previous paragraph) are at risk of being fetishized in a way that has been the case for design patterns, data structures, algorithms, and even math itself. Not knowing all of the trivia (let alone the deep mathematical basis) of monadic laws should not stop you from using useful objects and functions when you find them. "I'm not a math person" and a terribly architected website-producing company interviewing with "How do you reverse a linked list?" are evil twins.

Just because in some contexts, blocks of knowledge are unfairly treated as a binary of known or unknown does not mean we should perpetuate that. If you worked through the last section and used the `Maybe` and `Either` interfaces, you've used a monad, as well as some other good abstractions. You will never *have* to make your own (when is the last time you implemented a linked list in JavaScript because arrays weren't good enough?), but you could. And if you did make your own, you could make them correct by someone's definition. You could add a few functions that you like even if they don't fit in with other people's ideas. You could leave stuff out. And you could call them burritos if you wanted.

The thing is, you can have objects and functions that don't conform to precise definitions but are still useful. If you're *creating* them, then the interfaces *and* laws of what makes a functor versus an applicative versus a monad should be important to you. If you're just *using* them, try things out, but focus on the interfaces and don't get hung up on every definition and law. If you're interested in writing or using something that conforms to the interfaces and rules of high-level functional programming abstractions, however, you should look into the aforementioned **Fantasy Land spec** (https://github.com/fantasyland/fantasy-land), which breaks things down into digestible pieces and provides links to **implementations** (https://github.com/fantasyland/fantasy-land/blob/master/implementations.md) of the spec (that means code you can use).

What's awesome about Haskell (and Lisp, and Prolog, and other opinionated languages) is that they force you into a certain frame of mind. The same goes for spreadsheets, calculators, calendars, and so on. What's awesome about JavaScript is that we have a massive platform and can write in so many different ways. We used a couple of really great high-level abstractions, and cut down on earlier code significantly.

DEEPER INTO FUNCTIONAL PROGRAMMING

If you're looking to get deeper into FP, you might find yourself hitting a wall by only using JS directly or with libraries that don't require a compilation step. For compiles-to-JS languages, you have some interesting options:

- TypeScript
- PureScript
- ClojureScript
- Elm

If you want to explore FP on its own, without worrying about compiling to JavaScript, then you could look into these:

- Haskell
- Scheme/Clojure (Lisp variants)
- Erlang
- Scala

As of this writing, there are many other choices out there in both categories, and many more are bound to be developed. The point is that you shouldn't feel restricted by only working in JavaScript. Good ideas come from everywhere. It's not uncommon to see a standard way of working in one language appear as an innovation in another.

Moving from OOP to FP

As mentioned before, if you're jumping paradigms like OOP to FP, it's probably much too large of a change to be called refactoring anymore. Despite that, it's likely that you will find working through a similar process helpful: make small steps, keep code changes under version control, keep a passing test suite.

Return of the Naive Bayes Classifier

Let's briefly return to our NBC example from Chapters 6 and 7. It would be great to show the full process of "restructuring" (or perhaps "refactoring," if you're taking a broad view of the term), but there are so many changes that, instead, after the code sample I'll offer some general advice if you ever decide to change the code this radically. Here's the functional version of our NBC:

```
// naive_bayes_functional.js
R = require('ramda');

const smoothing = 1.01;

function wordCountForLabel(testWord, relevantTexts){
  const equalsTestword = R.equals(testWord);
  return R.filter(equalsTestword, _allWords(relevantTexts)).length;
};

function likelihoodOfWord(word, relevantTexts, numberOfTexts){
  return wordCountForLabel(word,
                           relevantTexts) / numberOfTexts + smoothing;
};

function likelihoodByLabel(label, newWords, trainedSet){
  const relevantTexts = textsForLabel(trainedSet.texts, label)
  const initialValue = trainedSet.probabilities[label] + smoothing;
  const likelihood = R.product(
    newWords.map(newWord =>
      likelihoodOfWord(newWord,
                       relevantTexts,
                       trainedSet.texts.length))) * initialValue;
  return {[label]: likelihood}
}

function textsForLabel(texts, label){
  return R.filter(text => text.label === label)(texts);
}

function _allWords(theTexts){
  return R.flatten(R.pluck('words', theTexts));
};

function addText(words, label, existingText = []){
  return R.concat(existingText, [{words: words, label: label}]);
};

function train(allTexts) {
  const overTextLength = R.divide(R.__, allTexts.length);
  return {texts: allTexts,
          probabilities: R.map(overTextLength,
                               R.countBy(R.identity,
                                         R.pluck('label', allTexts)))};
};

function classify(newWords, trainedSet){
  const labelNames = R.keys(trainedSet.probabilities);
  return R.reduce((acc, label) =>
    R.merge(acc, likelihoodByLabel(label, newWords, trainedSet))
```

```
            , {}, labelNames);
};

module.exports = {_allWords: _allWords,
                  addText: addText,
                  train: train,
                  classify: classify}
```

And then the tests:

```
// naive_bayes_functional_test.js
const NB = require('./naive_bayes_functional.js');

const wish = require('wish');
describe('the file', () => {
  const english = NB.addText(['a', 'b', 'c', 'd', 'e', 'f', 'g',
                      'h', 'i', 'j', 'k', 'l', 'm', 'n', 'o', 'p', 'q'],
                      'yes')
  const moreEnglish = NB.addText(['a', 'e', 'i', 'o', 'u'],
                      'yes', english)
  const allTexts = NB.addText(['あ',    'い',    'う',    'え',    'お',
                       'か',    'き',    'く',    'け',    'こ'],
                       'no', moreEnglish)

  var trainedSet = NB.train(allTexts);

  it('works', () => {
    wish(true);
  })
  it('classifies', () =>{
    const classified = NB.classify(['お', 'は', 'よ', 'う', 'ご', 'ざ', 'い',
                                    'ま', 'す'], trainedSet);
    wish(classified['yes'] === 1.833745640534112);
    wish(classified['no'] === 3.456713680099012);
  });
  it('number of words', ()=>{
    wish(NB._allWords(trainedSet.texts).length === 32);
  });

  it('label probabilities', ()=>{
    wish(trainedSet.probabilities['yes'] === 0.6666666666666666);
    wish(trainedSet.probabilities['no'] === 0.3333333333333333);
  });
});
```

There are major differences from what we had before. Here are some of them:

- The algorithm has been slightly altered, leading to different values for "likelihoods," but that should not affect the majority of classification outcomes.
- Rather than exporting a class, we export individual functions.
- There is no `this` anywhere in the code.
- Because there is no `this` or other state variables, we have to "carry around" all of the state that we need as return values.
- As a consequence of this, our `train` and `classify` functions are now idempotent and pure: same input, same output. No side effects.
- Every function returns something useful.
- The `addText` function is also idempotent, but we're passing the return value of the early calls to the later ones. That is how we build the set of data.
- It is shorter, but the Ramda functions make the code fairly dense in places.
- We also dropped the sets and maps. It's worth exploring them, but they're not as flexible as objects and arrays.
- The module becomes just a namespace. Nothing in it is meant to be mutable, so it doesn't hold any state or data—just functions producing output that is used as input to other functions.
- The FP code is shorter.
- In the end, you'll have code that is focused on explicitly transforming your data, "pipelining" it through different operations rather than having multiple states on objects.

As far as getting to this point, the first thing to know is restructuring from OOP to FP is not easy. Some steps may seem counterintuitive if most of your experience is in moving from procedural to OOP.

In no particular order, here are some of the steps you may find yourself taking:

- Have good high-level tests in place. If something stops working, undo your changes.
- Make every function that isn't returning anything useful return the main object, probably with `return this` at first.
- Change all of the `this` references to explicitly refer to the object, like `Classifier` (in this case).
- Flatten out any objects and subobjects that you can.
- Move properties out of the objects.
- Replace state variables with queries.
- If you have trouble doing this, consider passing in the old state as part of queries until the state eventually just turns into inputs to functions.

- You'll probably make the code look worse before it looks better, as you will want to bring things out of structure before creating new structure. You might add duplication, pass around more parameters, and/or introduce unnecessary functions and poorly named variables as you go.

WHY DIDN'T WE COMPOSE MORE FUNCTIONS?

There are plenty of ways to combine functions (e.g., `R.compose`, inlining, creating an outer function that takes in all parameters), and it might seem obvious that we could have combined the `train` function with either the `addText` function or the `classify` function.

The first question is why do we need the individual functions as opposed to high-level combined ones? The reason for not combining `train` and `addText` is that the interface provided by `addText` is most useful if it's used a few times to build a bigger set of data. Training it at the same time would mean repetitious or more complicated calculation. The reason for not combining `train` and `classify` is that if the training data set were large enough, it could take a very long time to train. We want to keep that separate, as it will be easier to do things like pass it off to a separate process. Also, we were interested in just testing with one classified set. It's useful to not repeat the training step for every test.

But why we didn't make those functions available anyway? To keep the API smaller and simpler, and provide only one reasonable way to use the library.

All in all, it's hard to move from structure to structure—possibly harder than moving from unstructured to structured code (either OOP or FP). You can strike a balance between the two approaches, but mutating high-level state (the properties of instances of any objects, sets, maps, etc.) that you introduce is very different than simply relying on input and output.

Rewrites

Keep in mind that moving from OOP to FP is just one type of restructuring. If you're intending to move from one framework to another, a "rewrite" is probably what you'll end up doing. Although it's possible to do this slowly and carefully through a *strangler app* that slowly takes more and more responsibilities from the *legacy app*, it's hard to not allow that to lead to a fractured user experience for quite a while.

A full rewrite, where the old app is supported until the new one is "good enough" (unfortunately, this is usually pretty ambiguous), is an especially difficult problem, for a few reasons:

- *Feature parity* for two apps in different frameworks is hard to define.
- It's easy for design changes and new features to become "essential" to the new app, increasing the scope from feature parity to something else.
- A solid (possibly "phased") cutover plan/upgrade path for all customers should be defined.
- If the new version demands new skills to build, your current team, though experts on the old app, might not have great experience with the technologies required for the rewrite.
- "Business rules" that may be faithfully executed in the old app can be lost in translation, mistaken for bugs or details that don't matter.
- Progress can be very difficult to demonstrate.
- It's very difficult to estimate how long a rewrite will take.

Overall, restructuring between paradigms and rewriting between frameworks are both very hard. They're not simple, mechanical processes like refactoring can be. On a multidisciplinary team with various working styles and priorities, rewrites will mean very different things to different people.

The principles of moving in small, confident steps, backed up by tests and version control, are helpful. Aside from that, the best thing to do is communicate clearly how long things will take, including admitting when your estimates carry assumptions that may prove to be incorrect. When in doubt, double your estimate, and then triple it.

A REWRITE OF YOUR OWN

If you're interested in a mini-rewrite project, try moving the NBC code from OOP to FP style. Or try moving it to FP from the procedural version of the code (before the `classifier` object sucked in all of the functions). See if your version is similar to that of the previous section. Is it easier to start from scratch or move incrementally? Take a guess at how long it will take, and see how accurate your guess was.

Wrapping Up

In this chapter, we discussed some of the benefits of functional programming, and how a good portion of them are realized through practices we've been dis-

cussing all along: not reusing variables, avoiding shared state, extracting functions, and preferring `Array`'s higher-order functions to `for` loops.

We also explored some interfaces and libraries (Ramda and Sanctuary) for functional programming along with good old arrays, as well as `Maybe`, which might be new to you but will hopefully find a place in your toolkit to help squash `null`, the billion-dollar mistake.

Maybe functional programming will be your new favorite paradigm. If so, I'd encourage you to try something stricter like Haskell, but if that gets intimidating or you miss working in the browser, remember that you can explore the concepts and exploit the interfaces of FP in JavaScript with just a few `require` statements. FP doesn't have to be all or nothing.

It's a huge topic, and in this chapter, we just skimmed the surface. As to what a path toward learning FP could look like, I recommend the following sequence:

1. Get familiar with the native JS `Array`'s higher-order functions (they take a function as input), such as `forEach`, `map`, `filter`, and `reduce`.
2. Try out a library like underscore or lodash, which have many more manipulations than what is provided for in native JavaScript.
3. Get used to composing functions and manipulating arguments with Ramda and/or Sanctuary.
4. Use Sanctuary, ramda-fantasy, Folktale, or any of a number of other implementations of the Fantasy Land spec that describe burrito-like things from the earlier diagrams.

Along the way, try these out as well:

- Use Immutable.js or mori to enforce immutability beyond what `const` and `.freeze` can easily offer.
- Try out functional languages that compile to JS.
- Try out functional languages that don't compile to JS.

12 Conclusion

The topics we've covered have ranged from techniques as small as renaming variables to those as large as applying paradigms like FP and OOP. Taking account of paradigms, architectures, libraries, frameworks, and even individual style, your JavaScript might be wildly different from someone else's. You might even find yourself in two very different modes from one project to another. If you're just writing a simple three-line shell script in JavaScript, you might not want tests or need to write any functions. You might completely leave off scoping declarations, even a simple `var`. You might be working with a team that thinks well in OOP. In that case, ES2015 classes and applying appropriate design patterns might be the way to go. For more functional (or just function-curious) teams, FP is a great approach. It doesn't have to be all or nothing, and it doesn't have to be scary. Turn your linter off every once in a while.

Whether you're making decisions about small stylistic changes or large architectural changes, ideally this book has provided you with a good basis for making choices. The alternatives of simply using a framework, rewriting into another framework, or putting up with JavaScript Jenga are comparatively expensive and demoralizing.

"JavaScript" is basically a mythological animal. "Your JavaScript" starts with whatever styles, tools, platforms, paradigms, ideologies, and purposes you decide to pursue. Having a well-developed "your JavaScript" opens up a tremendous amount of possibilities.

Hopefully, this book has helped you craft your JavaScript, leading to the creation of more maintainable, confidence-inspiring codebases.

Further Reading and Resources A

This is a collection of books, tools, videos, and other resources for additional background on the topics covered in *Refactoring JavaScript*.

If any of the links go dead, try them on the **Wayback Machine** (http://archive.org/web/).

For sources without links, try searching the words given (with and without surrounding quotes), and you should be able to find them.

Most resources are free, but some are available only for purchase (or with clever searching). In the cases of nonfree resources, I have not provided a link and have included the author name for easier searching.

Origins of Refactoring

- "Refactoring Object-Oriented Frameworks" by William F. Opdyke
- *Refactoring: Improving the Design of Existing Code* by Martin Fowler, Kent Beck, John Brant, and William Opdyke (Addison-Wesley)
- *Design Patterns: Elements of Reusable Object-Oriented Software* (GoF book) by Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides (Addison-Wesley)

Baseline JavaScript(s)

- ***Speaking JavaScript: An In-Depth Guide for Programmers*** (http://speakingjs.com/) by Axel Rauschmayer (O'Reilly)
- ***Exploring ES6: Upgrade to the Next Version of JavaScript*** (http://exploringjs.com/) by Axel Rauschmayer (Leanpub)

- **List of compiles-to-JS languages** (http://github.com/jashkenas/coffee-script/wiki/List-of-languages-that-compile-to-JS)
- *JavaScript: The Good Parts* by Douglas Crockford (O'Reilly)

Keeping Up with JavaScript

- **node.green**
- **caniuse** (http://caniuse.com)
- **ESNext Compatibility Table** (http://kangax.github.io/compat-table/esnext/)
- **TC39 Committee (proposal stages on GitHub)** (http://github.com/tc39/proposals)

JavaScript Reference

- **Mozilla Developer Network** (http://developer.mozilla.org/en-US/)
- **Global Objects** (http://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects)
- **Object** (http://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Object)
- **Array** (http://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Array)
- **Promise** (http://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Promise)
- **node docs** (http://nodejs.org/en/docs/)

Object-Oriented Programs/Patterns (Including Anticlass Stances)

- **Wikipedia: Software design pattern** (http://en.wikipedia.org/wiki/Software_design_pattern)
- **Refactoring catalog** (http://refactoring.com/catalog/)
- ***Learning JavaScript Design Patterns*** (http://addyosmani.com/resources/essentialjsdesignpatterns/book/) by Addy Osmani (O'Reilly)
- ***Game Programming Patterns*** (http://gameprogrammingpatterns.com/) by Robert Nystrom (Gennever Benning)
- **Classes reference** (http://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Classes)
- **Wikipedia: Tony Hoare** (http://en.wikipedia.org/wiki/Tony_Hoare)

- **Wikipedia: Null Object pattern** (https://en.wikipedia.org/wiki/Null_Object_pattern)
- **JavaScript Factory Functions vs Constructor Functions vs Classes** (http://medium.com/javascript-scene/javascript-factory-functions-vs-constructor-functions-vs-classes-2f22ceddf33e)
- ***You Don't Know JS: This & Object Prototypes*** (http://github.com/getify/You-Dont-Know-JS/tree/master/this%20%26%20object%20prototypes) by Kyle Simpson
- **Not Awesome: ES6 Classes** (http://github.com/joshburgess/not-awesome-es6-classes/)
- *JavaScript Patterns* by Stoyan Stefanov (O'Reilly)

Async

- **Continuation passing style** (http://www.2ality.com/2012/06/continuation-passing-style.html)
- **JavaScript Promises: An Introduction** (http://developers.google.com/web/fundamentals/getting-started/primers/promises)
- **Promises Pt. 1** (http://www.2ality.com/2014/09/es6-promises-foundations.html)
- **Promises Pt. 2** (http://www.2ality.com/2014/10/es6-promises-api.html)
- *JavaScript with Promises* by Daniel Parker (O'Reilly)

Functional

- **Hey Underscore, You're Doing It Wrong** (http://www.youtube.com/watch?v=m3svKOdZijA)
- **JavaScript Allongé** (http://leanpub.com/javascriptallongesix/read)
- **Fantasy Land Specification** (http://github.com/fantasyland/fantasy-land)
- **Professor Frisby's Mostly Adequate Guide to Functional Programming** (http://drboolean.gitbooks.io/mostly-adequate-guide/content/)
- **Hindley–Milner Type System** (http://en.wikipedia.org/wiki/Hindley%E2%80%93Milner_type_system)
- **Learn You a Haskell for Great Good!** (http://learnyouahaskell.com/chapters)
- **Constraints Liberate, Liberties Constrain** (http://www.youtube.com/watch?v=GqmsQeSzMdw)
- **Functors, Applicatives, and Monads in Pictures** (http://adit.io/posts/2013-04-17-functors,_applicatives,_and_monads_in_pictures.html)

- **Refactoring Ruby with Monads** (http://codon.com/refactoring-ruby-with-monads)
- **Burritos** (http://github.com/evanburchard/burritos)
- ***Functional-Light JavaScript*** (https://github.com/getify/functional-light-js) by Kyle Simpson
- *Functional JavaScript* by Michael Fogus (O'Reilly)

Tools

- **node** (http://nodejs.org) (JavaScript outside of the browser)
- **git** (http://git-scm.com/) (source/version control management)
- **npm** (http://www.npmjs.com) (node package manager)
- **yarn** (http://yarnpkg.com) (npm alternative)
- **node `assert`** (http://nodejs.org/api/assert.html) and **browser `console.assert`** (http://developer.mozilla.org/en-US/docs/Web/API/Console/assert)
- **wish** (http://github.com/evanburchard/wish) (`assert` alternative)
- **mocha** (http://mochajs.org/) (big testing library)
- **tape** (http://github.com/substack/tape) (smaller testing library)
- **testdouble** (http://www.npmjs.com/package/testdouble) (mocking/stubbing framework)
- **underscore.js** (http://underscorejs.org/) (functional library)
- **lodash** (http://lodash.com/) (functional library)
- **Ramda** (http://ramdajs.com/) (better functional library)
- **Sanctuary** (http://sanctuary.js.org/) (FP with objects too)
- **jQuery** (http://jquery.com/) (JavaScript library)
- **Trellus** (http://www.trell.us/) (function diagramming)

Non-JS-Specific but Relevant Sources

- *Refactoring to Patterns* by Joshua Kerievsky (Addison-Wesley)
- *Design Patterns in Ruby* by Russ Olsen (Addison-Wesley)
- *Refactoring: Ruby Edition* by Jay Fields, Shane Harvie, Martin Fowler, and Kent Beck (Addison-Wesley)
- The **"Therapeutic Refactoring" presentation** (http://confreaks.tv/videos/cascadiaruby2012-therapeutic-refactoring) by Katrina Owen

Me

- **Compliments, complaints, questions, and so on** (http://evanburchard.com/contact)

Index

Symbols

A

B

C

M

N

O

P

Q

R

S

T

U

V

W

About the Author

Evan Burchard is a web development consultant and the author of *The Web Game Developer's Cookbook* (Addison-Wesley). Offline, he has designed an award-winning kinetic game involving stacking real ice cubes, and periodically picks up his project to walk across the US.

Colophon

The animal on the cover of *Refactoring JavaScript* is a Russian desman (*Desmana moschata*), a small mammal distantly related to the mole. It is semiaquatic, and lives in a habitat of lake and river basins throughout Russia, Ukraine, and Kazakhstan. Desmans are generally only 7–8 inches long and 14–18 ounces in weight.

The desman is functionally blind, but its distinctive two-lobed snout is packed with specialized dermal bumps (known as Eimer's organs, which are also found in moles). These organs are highly sensitive to touch and are the animal's primary source of sensory input. The desman also uses its long nose like a periscope, to breathe and sniff for threats above the surface of the water. Desmans are adept swimmers, with webbed hind feet and laterally flat tails (which help steer like a ship's rudder). And indeed, though they dig dens on land in which to sleep and raise their young, desmans spend much of their time in the water; their dens even have underwater entrances. Their diet is made up of insects, larvae, amphibians, and small fish, most of which they catch underwater.

Unfortunately, the Russian desman is endangered. Its soft, thick pelt—ideal for life in cold water—made it highly sought after in the fur trade of the early 20th century and the species was greatly overhunted. In 1957, the Soviet government established a complete ban on hunting desmans. Despite this, other

factors like logging, water pollution, and wetland drainage have continued to cause a steep population decline as viable habitat disappears. Recent conservation efforts have had some success in establishing a healthy desman population within wildlife reserves.

Many of the animals on O'Reilly covers are endangered; all of them are important to the world. To learn more about how you can help, go to ***animals.oreilly.com***.

The cover image is from *Natural History of Animals*. The cover fonts are URW Typewriter and Guardian Sans. The text font is Adobe Minion Pro; the heading font is Adobe Myriad Condensed; and the code font is Dalton Maag's Ubuntu Mono.