Refactoring JavaScript

1ST EDITION

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://safaribooksonline.com). For more information, contact our corporate/institutional sales department: 800-998-9938 or corporate@oreilly.com .

Editor: Allyson MacDonald

Production Editor: FILL IN PRODUCTION EDITOR

Copyeditor: FILL IN COPYEDITOR

Proofreader: FILL IN PROOFREADER

Indexer: FILL IN INDEXER

Interior Designer: David Futato

Cover Designer: Karen Montgomery

Illustrator: Rebecca Demarest

January 2017: First Edition

Revision History for the First Edition

- 2016-10-04: First Early Release

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

978-1-491-96492-7

[FILL IN]

Preface

Conventions Used in This Book

The following typographical conventions are used in this book:

Italic

Indicates new terms, URLs, email addresses, filenames, and file extensions.

`Constant width`

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

`Constant width bold`

Shows commands or other text that should be typed literally by the user.

`Constant width italic`

Shows text that should be replaced with user-supplied values or by values determined by context.

> **TIP**
>
> This element signifies a tip or suggestion.

> **NOTE**
>
> This element signifies a general note.

> **WARNING**
>
> This element indicates a warning or caution.

Using Code Examples

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

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: "*Book Title* by Some Author (O'Reilly). Copyright 2012 Some Copyright Holder, 978-0-596-xxxx-x."

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

Safari® Books Online

Safari Books Online is an on-demand digital library that delivers expert content in both book and video form from the world's leading authors in technology and business.

Technology professionals, software developers, web designers, and business and creative professionals use Safari Books Online as their primary resource for research, problem solving, learning, and certification training.

Safari Books Online offers a range of plans and pricing for enterprise, government, education, and individuals.

Members have access to thousands of books, training videos, and prepublication manuscripts in one fully searchable database from publishers like O'Reilly Media, Prentice Hall Professional, Addison-Wesley Professional, Microsoft Press, Sams, Que, Peachpit Press, Focal Press, Cisco Press, John Wiley & Sons,

Syngress, Morgan Kaufmann, IBM Redbooks, Packt, Adobe Press, FT Press, Apress, Manning, New Riders, McGraw-Hill, Jones & Bartlett, Course Technology, and hundreds more. For more information about Safari Books Online, please visit us online.

How to Contact Us

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

O’Reilly Media, Inc.

1005 Gravenstein Highway North

Sebastopol, CA 95472

800-998-9938 (in the United States or Canada)

707-829-0515 (international or local)

707-829-0104 (fax)

We have a web page for this book, where we list errata, examples, and any additional information. You can access this page at http://www.oreilly.com/catalog/<catalog page>.

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

Chapter 1. What is Refactoring?

Refactoring is not changing code.

Ok. 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:

1. How do you guarantee that behavior does not change?
2. 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. If you're curious about the history of JavaScript and refactoring, considering having a look at Appendix A: Timeline.

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 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 by removing the human element, the types of changes that can be made become limited, as the number of changes that are able

to be guaranteed as “safe” is confined to the functionality of the tools.

Writing these tools would be especially difficult to encompass the whole refactoring catalog proposed by Martin Fowler in his seminal book, “Refactoring: Improving the Design of Existing Code.” And in JavaScript, a dynamic, multi-paradigmic language, with its 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 “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 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 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 code base if you cannot easily revert it to a previous, safe version. If you don’t have versioned backups of the code base that you plan on “refactoring,” put this book down and don’t pick it up till you have your code under version control.

If you’re not using version control already, you probably want to use git, and you probably want to back your work up on GitHub.

Admittedly, our approach might seem reactive and cavalier in comparison to the earlier paths of automation and mechanics. However, the process (the red, green, 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 those 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 work humbly stands on, is that they gave us a new mindset around making code better and doing it safely. Exposure to and concern for that mindset, is the most important thing to convey.

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 use 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 `<<` operators are 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 numbers get too large. 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 code base 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. Let's say that you're testing something like this:

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

If you do this, you're testing JavaScript itself: it's numbers, `+` operator, and `===` operator. Similarly, and more frequently, you might find yourself testing libraries.

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

This is testing to see if underscore 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 better to rely on the tests that the packages themselves have (any good library would have its own tests). One caveat to this is "sanity tests," where you're ensuring 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.

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, 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 the behavior.

This situation isn’t confined to legacy code either. New code that is untested is also not possible to refactor 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 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 “non-functional” tests, and generally not the focus of refactoring, we can prioritize performance characteristics (and other “non-functional” 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 “non-functional” aspects of a codebase by being relatively easy to elevate to a similar standard of correctness. By “benchmarking” our code, we can create tests that fail when performance (eg. of a function) is too slow, and pass when performance is acceptable. This is done 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 non-functional 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: which JavaScript are you using?), it will be completely impossible to either change (unsafe) or refactor (safe) your code without changing performance. Adding a few lines of front-end 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 code base. 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?" Second, what is quality, and how does refactoring to 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” code base that doesn’t do anything of value. And on the other hand, you have a code base 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 psuedo-financial lingo that non-coders 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 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 from being written.

There is no one grand metric for quality. For the purposes of this book, quality code is that 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. The author not so humbly introduces the “EVAN” principles of code quality:

- Every high-level code path is tested
- Variables, functions, objects and other constructs are simple and well-defined
- Abstractions are applied after tests and functionality are in place
- No unnecessary side-effects or other code

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 code base, 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 code base.

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 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 “non-functional” 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 a visual language called "Trellus Diagramming," which you can learn more about at trell.us.

WHY DID I SPELL "TRELLIS" WRONG?

The .us domain was 80 cents. The .is domain was around $400. Plus there was already a "trellis" npm package. And thus, "trellus" was born.

If you're wondering what a "trellis" is anyways, it's a structure that supports hanging plants and vines. Looking at trell.us or Chapter 5 should make the similarities clear.

In Chapters 6 and 7, we look at general refactoring techniques. Following that, we Object-Oriented Pattern inspired refactoring (Chapter 8), Asyncronous Refactoring (Chapter 9), and refactoring through Functional Programming (Chapter 10).

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.

What is and isn't refactoring

Before we leave off, let's once again distinguish between refactoring and other lookalike processes. Initially for this book, it was considered that "refactoring," as it is colloquially used to mean "changing code" would be designated by a lower case "r" whereas our more specific definition (that preserves behavior) would have a capital "R." Because this is cumbersome and we *never* mean lower-case "refactoring" in the context of this book, this distinction is not used. However, when you hear someone say "refactoring," it is worth pausing to consider whether they mean "Refactoring" or "refactoring" (aka changing code).

Here are 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
- Performance optimization
- Converting code to use a different interface (eg. 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 guarantee to preserve behavior, and therefore, are not refactoring, just changing code.

Conclusion

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

If defining and achieving quality code through refactoring in JavaScript could be done in the abstract or by simply by looking at inspiring source examples from other languages (notably Java), our JavaScript code bases would not suffer from the “broken or new” dichotomy of today, where code bases 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, 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 “Framework Of The Month”, followed by more suffering in that framework, followed by rebuilding... and so on.

Chapter 2. Which JavaScript Are You Using?

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
- Pre-compiled Languages
- Frameworks
- Libraries
- What JavaScript Do You Need?

These can represent not only a different way 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, before later being 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. 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 "which language should I learn?" that is probably the biggest question from new developers. 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, the amount of frameworks, platforms and ultimately distinct types of code you might write varies so much, 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 there first. If you don't have a particular job in mind, find friends and mentors at meetups and follow what they do. If neither of those is the case, and you're concerned with learning something that's interesting to you, just pick one that seems cool* and go as deep as you want, move on, and at some point, consider getting very adept with a handful of technologies.

You can apply this same process (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 solves 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 things rather than spending to 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." Although features of JavaScript can bubble up from libraries and specific implementations, if you are looking for root for features that are likely to stick around, watch what ECMAScript is doing.

Specifically, (as of this writing) the committee responsible for tracking and adopting new features to the spec is called TC39. Spec proposals go through a multi-staged process for adoption. You can track Proposals in all 5 stages (0-4) at this URL:

http://github.com/tc39/proposals

ABOUT 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 are “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 might either: enjoy describing the features in a more accessible way through a blog post, 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 (ie. 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 (eg. 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 are not yet supported by your choice implementation, three words you’ll want to know are: “shims,” “polyfills,” and “transpilers.” Searching “how do I support ‘whatever feature’ in ’some platform’ (eg. node, firefox, chrome, etc.)” is will likely give you the answer you’re looking for.

Platforms and Implementations

When node.js hit the scene, some web developers were some combination of relieved and enthusiastic about the prospect of writing the same language on both the back-end and the front-end. Others lamented the fact that JavaScript could be the language to have such a prominent role.

This promise has seen mixed results. The JavaScript ecosystem has flourished with back end pushing the front end to be treated more like real code (organizationally and paradigmatically), while the pure mass of front end developers available ensured that the back end 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 front end and back end 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 front-end and back-end 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 which 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 to not only JavaScript APIs, but also HTML and CSS. For considering both front-end and back-end implementations, you can find broader feature tracking at kangax.github.io/compat-table/

If you're specifically interested in what new ECMA/TC39 proposals are implemented on a given platform, you can filter that table like this: kangax.github.io/compat-table/esnext

Implementations of a programming language can be called "run-times" or "installs" as well. Particular versions of JavaScript, especially when implementations exist in a browser, are sometimes referred to as a "JavaScript Engine." 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 term "build" or "release" used to describe where it is in the process. You could see things like "Nightly Build," "Weekly Build," "Stable Release," or "Long Term Support Release."

Experimental features (not from the ECMAScript spec), non-normative 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 whither 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" vs. "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 pre-compiled 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.js as a whole is out of scope for this section, but the language often used with it, JSX, is also an precompiled language that reads like a mixture of JavaScript and HTML. As evidence to 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 pre-compilations take a similar approach: write code the way you want to (either a new or 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 as of this writing. If one 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 website for whatever browsers we want to target by using supported features. Or we could choose to build a program outside of the browser using node. We could even use an precompiled language to backfill or extend the code we want to write and avoid writing *actual* JavaScript altogether if we wanted. 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, non-standard, and duplicated functionality (looking at you, `prototype`, `__proto__`, and `Object.getPrototypeOf`) guarantee 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.js, React, Angular, and so on 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) that 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 different enough in syntax and purpose that they still ask which they should learn. This is an evergreen question that any framework (front-end or back-end) 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.js 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 (eg. "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 how 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"`, 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 (ala `"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.

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, only bringing in tools when they justify themselves. This process is covered when we gradually introduce a testing framework in chapter 4.

Conclusion

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 "knowing JavaScript" being something of a moving target isn't without its frustrations, the outlook is great for finding new interests and work within JavaScript(s).

Chapter 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.”

“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.”

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 by mandate 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) with no interest in quality, the most likely outcome is 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 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, "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 judgement to write quality software. No reasonable boss or client would refuse *any* verification that 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 internal (from you) 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 why testing is useless/costly/not your job. Here, you're acting as Aesop's fox who, after grasping for grapes and failing, consoles itself by saying that they must have been sour anyways. If 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 the "Regression Testing for Bugs" section of the next chapter, 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 code base. 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 as a percentage of lines of code, that are covered by tests.
- "high-level" and "low-level" – Just like ordinary code, tests can be more broad (high-level) or 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 with "end-to-end tests," whereas "low-level" will correspond with "unit tests"
- "complexity" – This is a measurement of the pathways through the code, and 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 the non-functional and TDD sections.
- "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" – The situation where a lack of confidence and trust (via complexity and a lack of test coverage) in the code base results in more guesswork and slower development overall.

- “feedback loop” – How long the gap is between writing code and knowing if it is correct. A “tight” or “small” feedback loop (vs. 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 swapping it out 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 the “Manual Testing” section 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. Working with a team is easier

If your coworker writes some broken code, the test suite should let you both know that there’s a potential problem.

4. Demonstrating (not documenting) functionality

Making and maintaining documentation is a whole different discussion. But assuming you don’t have docs 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 is a step up from only having 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 your code is broken. 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 their tests then?

6. Big upgrades

You really want to start using using the latest version of big framework/new runtime. How can you upgrade responsibly and quickly? Just quickly? YOLO. Deploy it. It's probably ok, 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. To catch bugs early

The earlier bugs are caught in the development cycle, the easier they are to fix. Before you write them is ideal. If QA or product find it, that means more people have put hours into it. Once it hits customers, you're talking about another production cycle and potential loss of business or trust.

8. To 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. To have a tighter feedback loop

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 complexity of the program grows), how often will you do it? Odds are the tighter your feedback loop, 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. On each, one important thing to note is the testing methods we are looking at have three stages, the setup, the assertion, and the "tear down."

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 are 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," etc. 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 quality of the software itself, rather than the experience 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 visa-versa. In any case, 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 expected values. Whether you call it "monkey testing" or "manual testing," "making sure it works," or "QA'ing" this testing strategy is useful for exploration and debugging.

If you're faced with an undertested code-base, 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 towards 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. In this form, you automate the test set up and teardown, but the assertion (or “approval”) is left to human input.

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 when it comes to asserting that image created by the cropping tool worked well, it could be difficult to write a simple assertion to do so. 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. Do you skip testing altogether?

Instead of a simple “failure,” this test suite has a memory for what was alright the last time around. 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/db entries on the development machine or 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, and 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, and thus 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 regex parser (most of the time) for HTML/CSS, those are better tested through unit tests or end-to-end tests.

Just as manual tests are an organic individual 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 the output of something, an ad hoc approval test system is in place. Depending on the technical ability of the approver, they may be responsible for the setup programmatically, through a documented manual test, or asking the developer for a demo.

The weaknesses of the ad-hoc approval test system is that setup may be onerous for the approver, and such system has a dubious capability to remember what outputs were approved/rejected. This is not an insult the memory of anyone involved. Even if they are singularly focused on this process, if the team using this process changes, all of the knowledge of the old approvals changes as well.

But recognize that any attempt to framework-itize process will produce something that looks different to an approver than what they might be used to.

Keeping things simple for them probably means a list of urls to be reviewed in the queue.

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

Even though this process is popularly practiced in an ad hoc way by teams, 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 this 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 above 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, “CucumberJS” 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 flexibility in process 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 non-developer 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 webpage, downloading a file, etc.

For tests like these, code should be exercised in collaboration with other pieces of the code base. “Mocking” and “stubbing” should be avoided except when absolutely necessary (usually involving the file system 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. If you’re not familiar with those terms, “high-level” means to take a broader view and have your code be more concerned with an integration of parts vs. “low-level” which means 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, object, 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 of that class, along with creation of objects from the class.

ABOUT “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 doing so necessarily involves extra setup and/or awkward testing scenarios.

In any case, a popular recommendation is to only test “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 by running the public methods that make use of them. This leaves private methods with a bit of

flexibility to change their "implementation details," without having to rewrite the tests (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 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 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 fake the calls to remote services and possibly the file system and database as well, 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 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 sign up 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," etc. 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 at 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 potential 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 write to begin with. Although at some point this approach may be undesirable, having a 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 the performance must be addressed. This can be done through rented test running architecture online, parallelizing tests, splitting into slow and fast suites, and mocking/stubbing calls to systems that are particularly slow.

Non-functional Testing

In the context of refactoring, non-functional testing does not directly contribute to accomplishing our code quality goals. Nor does it contribute to "confidence" that the code works. Non-function testing techniques include:

- Performance Testing
- Usability Testing

- Playtesting
- Security Testing
- Accessibility Testing
- Localization Testing

Results of tests of these types contribute to new features 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 it 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, a terribly unwelcoming and frustrating process.

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

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

Non-functional 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 non-functional 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 diverse team, functionally, demographically, and in life experiences, can help to route out the

most egregious problems quickly, and make the nuances more clear.

All this said, these testing techniques will 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 main line 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, as will be demonstrated in the next chapter.
- Regression Tests – These tests are intended initially to reproduce a bug, followed by changes to the implementation code in order to fix the bug. This guarantees coverage so that the 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 are 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 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, we'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 our 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 backwards 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-a-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 reproduce the bug." One document may cover all the systems and languages that a team uses, or they could be broken out into several (eg. "css style guide," "front-end style guide," "testing style guide," etc.).

Developing this document 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 document in place provides a good foundation for the other processes covered here. Also, in an ideal world, this 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 with "*coding* style guides” is not recommended.

Developer Happiness Meeting

Another lightweight quality-focused process worth considering is a weekly, short meeting sometimes referred to as “developer happiness" (this name may not be well received by non-coders in the organization), “engineering quality,” or “technical debt.” 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 type of tasks.

However the tasks are given priority, this process allows for things like “speeding up the test suite,” “adding test coverage to the 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, aka “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 small bugs, even just typos, can be caught 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, ranging from algorithms to keyboard shortcuts, to spread through an organization.

Knowledge sharing, focus, high quality code, and company. What's the downside? Well, like 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 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, along 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 members. 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 team-wide 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 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."

ABOUT "RESOURCES"

When people (usually project/product managers) refer to 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 a quizzical look and saying "What? Oh... You mean *people*?"

> 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 face) strictly by their function is somewhere between unprofessional and mechanistically dehumanizing.

Three (not mutually exclusive) variations on pairing worth noting are "TDD (test-driven development) 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 ensuring 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 to address, or at least 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 (aka 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 *only* written 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 task is understood well enough to write tests and implement them a la the normal “red, green, refactor” cycle.

A term that you’ll often see near to 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 things of importance. First, the tests tend to be high-level, end-to-end tests. Second, inside of 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.

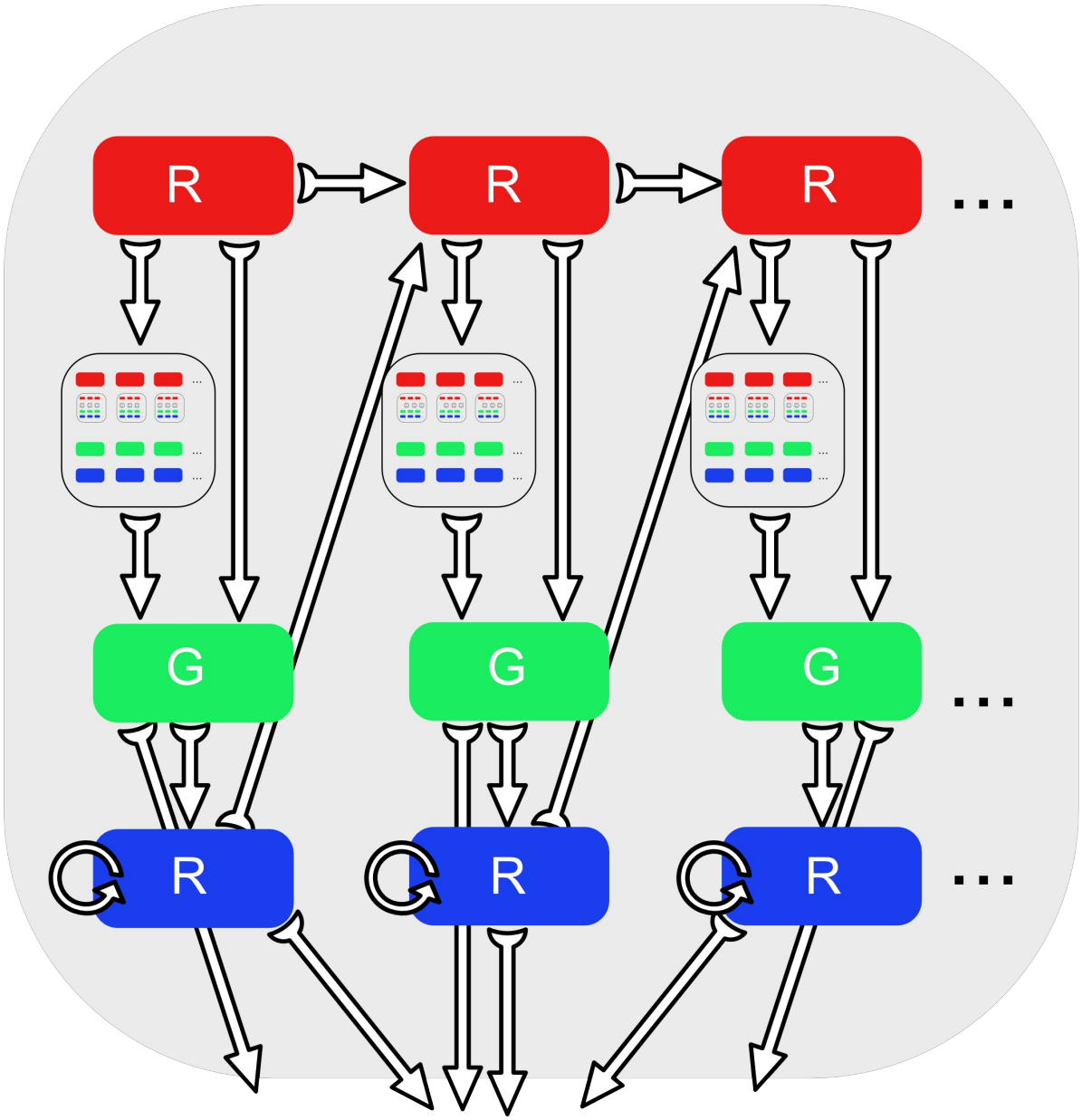

Figure 3-1.

red_green_refactor_.png

Woah. Ok. 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 3 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). If possible, we could follow the long arrow down to the green state (implement the test), or the most complicated possibility exists for when we can't implement the test and 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 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, you can do one of two things. Either, refactor, or call it done and consider this test and code complete (leave the cycle). When you start refactoring, you 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 you're in an inner cycle, that means you move into the green phase of the outer cycle. If you're in the outer cycle, and you can't think of any more tests to write (no more reds to tack on to the top row, then you're done.

One subtlety process 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.

- (Outer Cycle) Failing ("red"), high-level test is written: "A customer can login."
- (Inner Cycle) Red, low-level test is written: "Route '/login' returns 200 response."
- (Inner Cycle) Routing code is written to pass the low-level test. You can refactor this code written on the inner cycle now.
- (Inner Cycle) 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."

- (Inner Cycle) Code is written to handle successfully posting email/password and returning the logged in home page. This inner cycle test is passing, so you can refactor this code written on the inner cycle now.
- (Outer Cycle) Now the second low-level test, as well as the high-level test are both passing.
- Once everything 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 resumes, hiring pages display them on job listings, and top consultancies build their brand 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 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 “js [tooltype]” (eg. “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 (git + GitHub as of this writing 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 below 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 run the test suite. A test framework may include many of the tools listed below as well. Some frameworks are specific to the front-end or back-end 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 provide 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. They are also likely 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 afterwards (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 lighter weight 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.

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 which extends the testing framework.

Factories and Fixtures

These tools are responsible for creating db objects either on demand (factories) or from a file specifying data (fixtures). These libraries are sometimes associated with the word “fake” or “faker.”

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 are 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 of your test suite can slow you down significantly. Loaders 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 (aka "CI")

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 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 coverage is able to be determined without actually running the test suite (and consequently exercising the code) is an interesting academic question of "dynamic" vs. "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 continuous integration systems.

MUTATION TESTING

If you're interested in other possibilities using dynamic analysis, as 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 code base (most easily by changing test inputs, eg. a string input where a boolean was expected) and *fails* when your tests *pass* in spite of running 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 here. First is that mutation testing relies on multiple variations on your normal test suite, so it will tend to be slow. Second is that it may break your code in unexpected ways and complicate the tear-down 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. Files can be inspected 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 programmers 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 an exploratory spike, a tricky test case, or while manually testing, it may be unclear what value 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 (eg. “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/VMs) 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.

Chapter 4. Testing in action

In the last chapter, we looked at some common objections to testing, and explored the benefits that hopefully overwhelmed 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, 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
- Regression Testing for Bugs

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, he 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 non-academic setting, 24 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 from nodejs.org
- Installing node should also install npm
- When you have npm, install mocha with `sudo npm -g install mocha` (may not need "sudo")

To make sure everything is alright, try running the following

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

Yes, that last "V" is a capital "V." (I don't know why either) Others are lower case. If you run into any issues, try searching for "installing node/npm on [whatever your operating system is]" (eg. "installing npm on windows").

New code from scratch

Here, 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 (eg. 'straight,' 'flush,' 'pair,' etc.)"

NEW FEATURES VS. 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 code base 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? Create this file and save it as checkHand.js. If you run it with `node checkHand.js` right now, you'll get an error, because we haven't defined any of our inner functions yet.

```
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';
  }
  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 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 eight or ten 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']));
```

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 and pumping your fist into the air saying "yes!", followed by some more swearing?

This is a sign your feedback loop is not tight enough. Getting too excited or disappointed about things working or not both reflect 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 afterwards, effectively destroying any *test-like* coverage that they had. Because we know that tests are important, we will get our coverage back after the tests are written, 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? After implementing all of these check methods, manually testing to "see if it's working" as we go, we eventually, we might end up with something like 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';
  }
  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 `getValues`

function is going to break with a value of 10. Namely, it will return a 1, aka an ace.

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

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 of implementation first?

SOUP VS. 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 know 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.

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?

```
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');
```

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 talk about the specifics of how to deal with long lines in later chapters, but for now just trust that these places are ok.

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 introduce a flow chart to introduce all of the possibilities of what code we need to write when.

_testing_in_action_.png

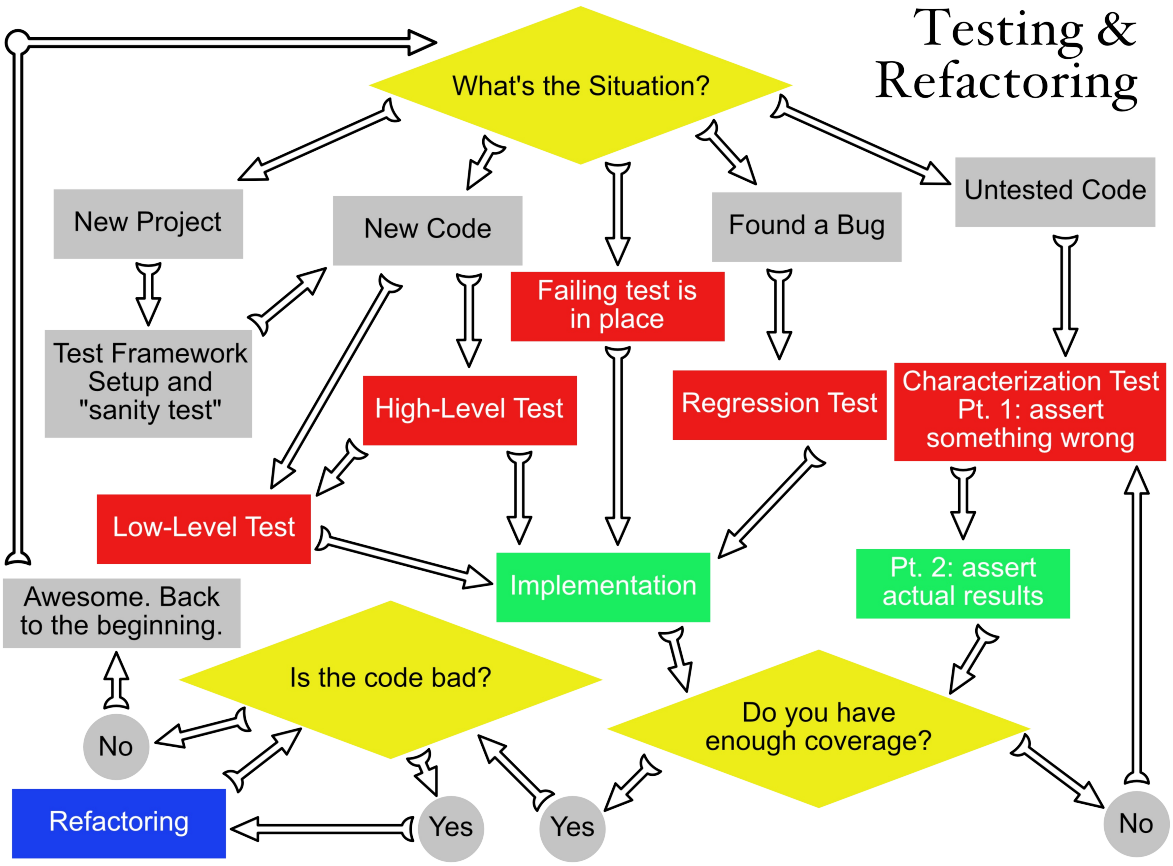

Figure 4-1. If you're ever wondering whether to write a test, refactor, or implement the code to make your test pass, this flow chart should help.

New code from scratch with TDD

Of course not all new code written without tests is going to end up in as awkward of a state as in the last section. It is certainly not reflective of an experienced coder's *best effort*. However, it is reflective of a lot of coders' *first effort*. The wonderful thing about refactoring is that given enough test coverage, your *first effort* doesn't have to be your *best effort*. Get the tests passing, and you have the flexibility to make it better afterwards.

ABOUT TDD AND RED, GREEN, REFACTOR

This section jumps between lots of tiny 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 would give you a good idea of how the pacing works. Even if TDD isn't something that you won't rely on all the time, knowing how to use tests for immediate feedback is valuable.

Ok. Back to our `checkHand` code. This is what we have so far.

```
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');
```

Let's keep those lines at the bottom and add our implementation to the top of the file. 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');
```

Ok. So we 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 “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 assert error is a little harder to understand, but basically, this is what asserts look like when they fail. But asserts can take 2 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 it means the same thing. Our assertion failed. The message doesn’t really give us more information than we had before though. The most important part of the error is this part:

```
/fs/check-hand.js:3:1
```

That means that the failure was on line 3. We know what assertion is on line 3, we won't worry about differentiating our test cases with messages just yet. Incidentally, realize that if we were using a normal *test* framework, it would give us more information, and be able to report multiple failures at once.

So there is this idea in TDD that in order to ensure moving by small steps, you should only write enough code to make your tests pass.

```
var checkHand = function(){
  return 'pair';
};
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');
```

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 something. In any case, now that we have that test "green," it's time to either refactor, or write another test. No obvious refactoring here (and we're going to mostly avoid it in this chapter), so we're onto our next test. Conveniently, we already have it written, so we can just uncomment the last line:

```
var checkHand = function(){
  return 'pair';
};
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 failure. Because we didn't supply the failure message, it's this one again:

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

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 (which you should at some point), you might be intimidated because it has a ton of features. Here, we’re using the simplest set up 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 assert = require('assert');

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

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 like this (make sure you're in the same directory as your file):

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

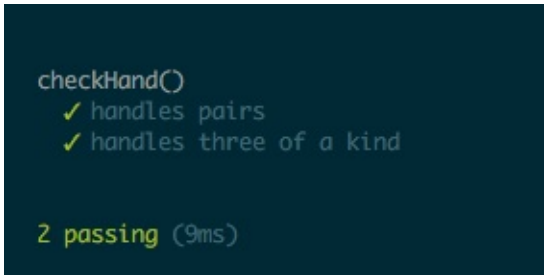

Figure 4-2. The output looks pretty good.

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 your files up like that, you can just run this: `mocha`

Ok. 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']);
    assert.equal(result, 'pair');

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

Now run mocha again. Three things to note here. First is that we can have multiple assertions inside of one `it`. Second is that 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()){
    return 'pair';
  }else{
    return 'three of a kind';
  }
}
```

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
```

Ok. Again doing just enough to shut the failures up, we write:

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

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

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

Now whenever we save the file, we'll get a new report (`ctrl-c` to exit). Ok, but 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']);
    assert(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, errors because `multiplesIn` is not defined. We want to define the method, but also, 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']);
    assert.equal(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 interface (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 = highestCount(['2-H', '3-C', '4-D', '5-H', '2-C']);
    assert.equal(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']);
      assert.equal(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?!

```
AssertionError:
[ '2', '3', '4', '5', '2' ] == [ '2', '3', '4', '5', '2' ]
```

Wait a second, I'm no equality genius, but those things look equal, don't they? Yes, but they aren't the same *object*, which is what `assert.equal` does. Our test actually needs to test the structure with `assert.deepEqual` rather than `assert.equal`.

```
describe('valuesFromHand()', function() {
  it('returns just the values from a hand', function() {
    var result = highestCount(['2-H', '3-C', '4-D', '5-H', '2-C']);
    assert.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 it was the first thing that came to mind. 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 list comprehension library like lodash or underscore to handle array manipulation, or use `reduce` (refactoring using reduce is covered in Chapter 6), 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';
  }
}
```

Undefined Error 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']);
    assert.equal(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)){ //new code
    return 'four of a kind'; //new code
  }
}

//new code
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']);
    assert.equal(result, 'high card');
  });
```

Failure. Red. 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']);
    assert.equal(result, 'flush');
  });
```

Failure. It doesn't meet any of the conditions, so it reports as high card. 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';
  }
}
```

`UndefinedError`. So we want:

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

Which 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.

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

Undefined errors for those new functions. We could write the boilerplate, but the `allTheSame` 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 `allTheSame` is working as expected.

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

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

Passing. But undefined error for `suitsFor`. It's pretty similar to our values function, so between that and the high-level test in place, we should be able to trust it.

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

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

```
  1) checkHand() handles high card:

      AssertionError: 'flush' == 'high card'
      + expected - actual

      -flush
      +high card
```

That must mean that `allTheSame` 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 the `allTheSame` function would actually return false when they aren't

all the same. Let's add that test now.

```
describe('allTheSame()', function() {
...
  it('reports false if elements are not the same', function() {
    var result = allTheSame(['D', 'H', 'D', 'D', 'D']);
    assert(!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 allTheSame(suits){
  var toReturn = true;
  suits.forEach(function(suit){
    if(suit !== suits[0]){
      toReturn = false;
    }
  })
  return toReturn;
}
```

Again, there's a better way to do this using lodash 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:

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

It's 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. Now 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']);
    assert(result);
  });
});
```

Note that these + signs in line 2 are turning strings into numbers.

Onto `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 if all elements are different', function() {
    var result = noMultiples(['2', '6']);
    assert(result);
  });
  it('reports true if all elements are different', function() {
    var result = noMultiples(['2', '2']);
    assert(!result);
  });
});
```

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

```
  it('handles straight flush', function() {
    var result = checkHand(['1-H', '2-H', '3-H', '4-H', '5-H']);
    assert.equal(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. This one is easy to implement, so we won't worry about a test.

```
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.

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

It's catching on the "three of a kind" branch of the `checkHand` conditional, so we want this 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, you 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});
}
```

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 break 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 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.

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);
}
```

Works. Great. Two pair and we’re done.

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

This is catching on the "`pair`" condition. That means it will have to go before the pair check.

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

And then an implementation that looks a terrific amount like full house.

```
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 how to write lots and lots of tests. There is a ton of duplication in the code and the tests. The amount 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

The code for randomly generating a hand of cards was written by a coworker. He's taken 2 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 3 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 one the difference between "changing code" and "refactoring." Also not recommended. Your third and best option is to add tests.

Here is his code:

```
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());
};

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 also, the confusing code won't seem to work without it. Those situations are tougher, but here, we can get this in a test harness 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.

First, we just assume that every function does nothing. You can add this code to the bottom of the file and run it with `mocha untested_code.js`

```
var assert = require('assert');
describe('doIt()', function() {
  it('returns nothing', function() {
    var result = doIt();
    assert.equal(result, null);
  });
});
describe('rC()', function() {
  it('returns nothing', function() {
    var result = rC();
    assert.equal(result, null);
  });
});
describe('rV()', function() {
  it('returns nothing', function() {
    var result = rV();
    assert.equal(result, null);
  });
});
describe('rS()', function() {
  it('returns nothing', function() {
    var result = rS();
    assert.equal(result, null);
  });
});
```

When you assume nothing from the code, it protests through the tests, “I beg your pardon, I’m actually returning *this* when I’m called with no arguments, thank you very much.” This is called a *characterization test*.

So once you know what is returned. You can test against *that* instead of *null*. So where we started by claiming that functions did nothing, we actually are just listening for them to tell us about themselves. Their behavior as reported by the test output becomes the basis for our coverage.

Our failures tell us what the code did, mostly. However, the `doIt` function returns *undefined* (if our test was `assert(result === null);`, then it fails because under the hood, our equality test was double equals, `==`). That 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).

ABOUT SIDE-EFFECTS

A side-effect means something like printing, altering a non-local 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 which JavaScript you're writing (chapter 2) as well as your personal style. Aiming to *minimize* side-effects is in line with the goals of this book. *Eliminating* them is an altogether different thing.

Anyways, 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, and it, combined with the first step of just asserting `null`, makes testing untested code way easier. If there aren't side effects, you never even have to look at the implementation! It's just inputs and outputs.

Because randomness is involved in these functions, it's a little trickier, but we can just use a regex to cover the variations in outputs. Here are the new tests.

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

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

We have options here. If this is part of a larger function program, first, we need to make sure that the variable `c` isn't accessed anywhere. Hard to do with a one letter variable name, but if you're sure it's confined to this, 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 all of our tests. 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());
```

Now our `doIt` test is broken because it returns something, 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 a high level test like that. If we decide to change the format of `rC`, we don't want two tests two become invalid. So what's a good high-level test here? Well, what's unique about the `doIt` function is that it returns 5 of something. Let's test that.

```
describe('doIt()', function() {
  it('returns 5 rCs', function() {
    var result = doIt();
    assert.equal(result, null);
  });
});
```

For smaller code bases that you want to get "in a test harness" or "under test," this process works well. For larger code bases, 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% 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, 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 code base. 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 code bases, 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 on 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 by writing regression tests.

To recap, if you find yourself with a larger legacy code base with poor coverage, identify small sections to get under test and use the process from this "Untested Code" section, adopt a policy of complete or majority coverage for new work and use the process for "New features" or "New Code from Scratch (with or without TDD)" sections, and write "regression tests" (detailed in the next section) 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” or “in a test harness,” 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 “just changing code,” and 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 is 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());
```

First thing to notice is that yes, 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 assert = require('assert');
describe('randomHand()', function() {
  it('returns 5 randomCards', function() {
    var result = randomHand();
    assert.equal(result.length, 5);
  });
});
describe('randomCard()', function() {
  it('returns nothing', function() {
    var result = randomCard();
    assert(result.match(/\w{1,2}-[HDSC]/));
  });
});
describe('randomValue()', function() {
  it('returns nothing', function() {
    var result = randomValue();
    assert(result.match(/\w{1,2}/));
  });
});
describe('randomSuit()', function() {
  it('returns nothing', function() {
    var result = randomSuit();
    assert(result.match(/[HDSC]/));
  });
});
```

First, we want to reproduce the problem somehow. Let’s try running this code (without the tests or mocha, so comment the tests lines out for now) using just node. Did you see the same card twice? Try running it again. How many times did it take to reproduce the bug?

With the “manual testing” 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();
      assert(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 will slow our system down further.

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.

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 last 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’s answer to 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.

```
describe('buildCardArray()', function() {
  it('returns a full deck', function() {
    var cardArrary = buildCardArray();
    assert.equal(cardArray.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.

```
it('returns nothing', function() {
  var cardArray = buildCardArray();
  assert.equal(cardArray, null);
});
```

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 our test case.

```
it('returns nothing', function() {
  var cardArray = buildCardArray();
  console.log(cardArray);
  assert.equal(cardArray, null);
});
```

Ok. So now we have the full array printed in the test runner. A bit of reformatting (now is a good time to learn your editor's "join lines" function if you don't already know it), and 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 was thousands of elements, we'd need another way to derive confidence, but since it's only fifty-two, by visual inspection, we can affirm that this array looks good. Let's change our "returns nothing" characterization 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('returns a good deck of cards', function() {
    var cardArray = buildCardArray();
    assert.deepEqual(cardArray, [ '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. Ok, 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 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 1 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. Perfect for our situation here, but the terms "destructive" and "impure" both apply to this function (see Chapter 10 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 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 1 iteration, our odds are 100% - 98.077%, or about 1.923%. Not 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%. 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 1 out of every 7 times, our test will not fail.

Back to confidence, ten-thousand 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() {
    var result = spliceCard(buildCardArray());
    assert.equal(result.length, 2);
  });
  it('returns the selected card', function() {
    var result = spliceCard(buildCardArray());
    assert(result[0].match(/\w{1,2}-[HDSC]/));
  });
  it('returns an array with one card gone', function() {
    var result = spliceCard(buildCardArray());
    assert.equal(result[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), 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? Extremely high-level tests aligned with corporate objectives? (How much money or did 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 section, 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 remove "dead code." Specifically, in addition to getting rid of the slow test, or isolating it in another file, we can remove the `randomSuit`, `randomValue`, and `randomCard` functions, as well as their tests.

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" as was 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 as a regex rather than a string:

```
/(red )*green (refactor ?)*/
```

Make a failing test. Then write the code to get that test passing. Then refactor the code if need be, and refactor as many times as you want. We could stop at one refactoring, but let's refactor our `randomHand` function one more time, taking advantage of destructuring (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.
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.

Chapter 5. Basic Refactoring Goals

In chapter one, we discussed refactoring as a process of changing code safely and without changing behavior in order to improve quality. In chapter two we looked at the complexity of the JavaScript ecosystem and therefore the difficulty in pinning down what style and quality means for us. In chapters 3 and 4, we laid the ground work 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 of refactoring and quality. We had a diagram in the last chapter that described the relationship between Testing & Refactoring. Recall that the three questions of our diagram were:

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

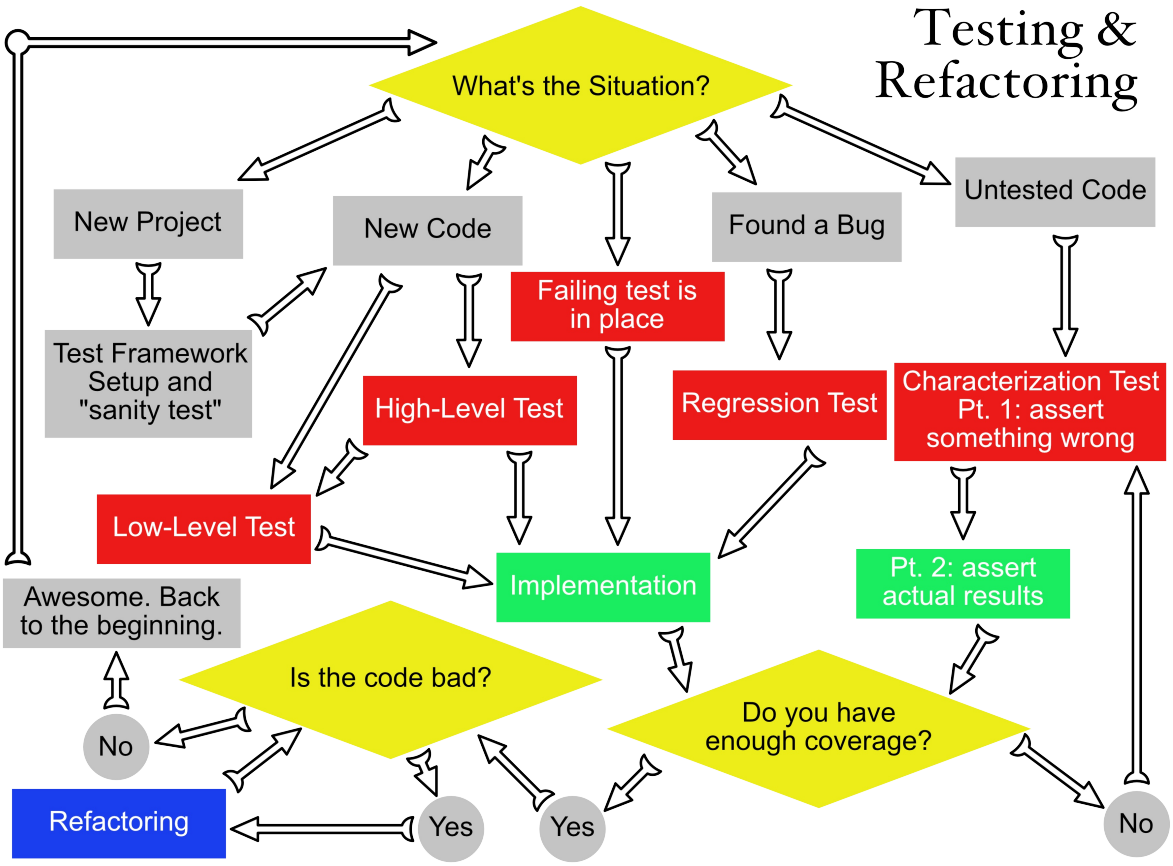

Figure 5-1. Testing and refactoring flow chart

_testing_in_action_.png

The first question should hopefully be very easy to answer. The second one can be addressed either by a coverage tool that runs with your test suite and outputs places specifically missing coverage. There are times when coverage tools may miss complex cases, so a line being covered does not necessarily mean we are confident in it. 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?

Due to JavaScript's multi-paradigmatic 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.

Before we address what refactoring in those styles entails, we must deal with a

paradigm that the worst code bases tend to employ: *unstructured imperative programming*. If those italics didn't scare you, perhaps a description of this type of code base in a front-end context will. Time for a spooky JavaScript story:

> *The JavaScript in the file, called "main.js," runs on page load. It's about 2000 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. They break often, as the team member who was initially excited about that framework has moved on to a startup 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, frameworks 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. Through this chapter, we will be exploring six basic components of functions:

- bulk
- inputs
- outputs (return values)
- side effects
- Context part 1: The Implicit Input
- Context Part 2: Privacy

"JAVASCRIPT JENGA"

If the front-end JavaScript code base 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 a changes to the front-end. Every time, they make as small of 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 available to ensure the code base are 1) performing as many manual checks of critical functionality as is allowed by time and patience and 2) hope. The code base 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 the user of the site. The only saving grace about this system is that version control allows the unstable (but mostly ok... 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 building a visual language for diagramming JavaScript. The full specification of this visual language can be found at trell.us.

Here's how it starts: Functions are a circle.

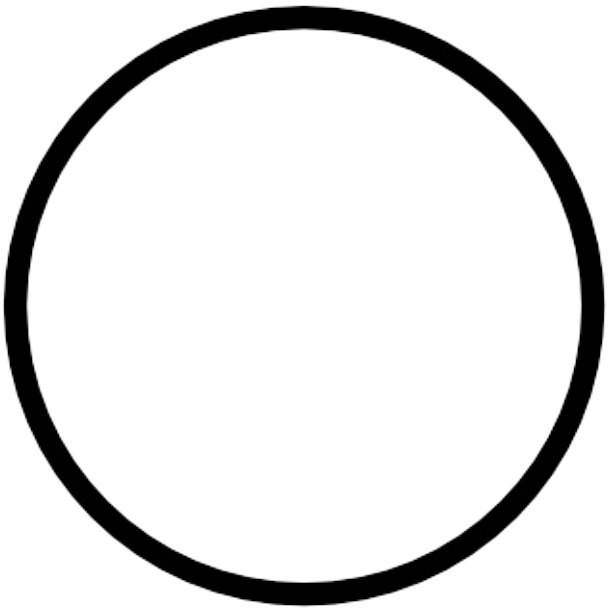

Figure 5-2. A diagram of a function.

Not much to it yet, but to reiterate what we said earlier, these are the basic

components of quality:

- bulk
- inputs
- outputs (return values)
- side effects
- context
- privacy

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

Function Bulk

The term “bulk” is not in common use to describe this, but is apt for two different characteristics of the function body:

- 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.

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”), limits tend to be around six.

Too much of one type of bulk will probably indicate the other type. A one-hundred 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.

IN DEFENSE OF BULK

Although classical refactoring techniques and trends in composable structures within frameworks and JavaScript packages point toward small functions being a preferable approach, there are detractors.

Although no one is rushing to defend 200 line functions on principle, you may occasionally encounter some that 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, there is usually 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 skil*l to learn for refactoring.

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 face objections both on stylistic preferences and (real or perceived) importance with respect to other development objectives.

To represent the “Bulk” of a function, we add two pieces to our diagram. The first is a flag on the right with the lines of code our function contains. To represent the paths of code (complexity) of our function, we can add pie slices. Slices filled in as grey are tested, an white ones are untested. Here are two examples:

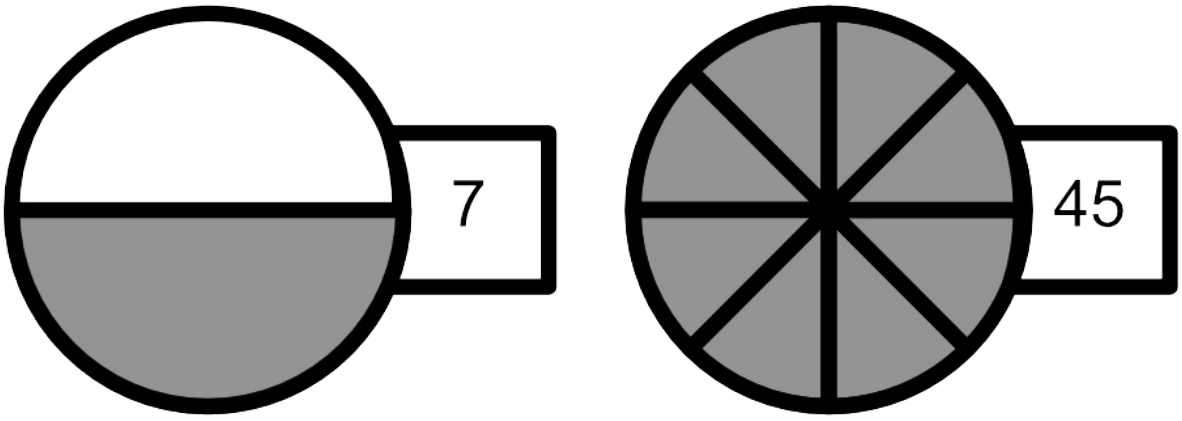

Figure 5-3. In this figure, we have two functions diagrammed. The one on the left has two code paths and 7 lines of code. One of the code paths is tested (grey) and one is not (white). The one on the right is bulkier, with 45 lines of code and 8 code paths (eight pie slices). Because all of the pie slices are filled in with grey, we know that each of the eight code paths is tested.

Inputs

For our purposes, we will consider three types of inputs to a function: explicit, implicit, and non-local. In the following function, we would consider `a` and `b` to be *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," because they are used in the *function call*, as opposed to "formal parameters," as they are referred to as when they occur in the *function definition*.

Secondly, although we're not diving deeply into objects here, the *implicit input* or *implicit parameter* in the add function here is the calculator object that the function is defined as:

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

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, which we'll talk about a bit more in the "Context"

section of this chapter.

The third type of inputs, *non-local*, can be especially tricky, especially in their ultimate form, the dreaded global variable. A more innocent-looking example can be seen below:

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

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

In testing, figuring out what inputs you need to set up takes more time than any other task involved. When explicit inputs are complex objects, this can be difficult enough (although it is often helped by “factories” as discussed in Chapter 3), but if your function relies on implicit input heavily (or worse, non-local/global inputs), then you have that much more set up to do. Implicit state isn’t so bad. Actually, Object-Oriented Programming relies on using it intelligently.

So the recommendation here is to have your functions, as much as possible rely on explicit inputs, followed by implicit input, followed in a distant third by non-local/global state. An easy way to guard against non-local state is to wrap as much of the code up in functions or classes (which are actually functions in disguise) as possible.

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 specific explicit inputs) require the types (eg. 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 which 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 passed in. You can even supply less arguments, as in this: `add(2)`. This will return `NaN` because 2 is being added to `undefined`.

As for extra arguments supplied to a function call, it is possible to recover and use them in the function body, but using 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 is allowing 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 are passed in at run-time, you have potentially bloated your test cases without realizing how. By hiding the values needed inside of a object called something generic 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, it's definitely preferable to used 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 Chapter 6.

This chapter is not meant to go too deep into the details, but 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 (`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. It 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 this 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 use 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.

ABOUT "SAD PATHS"

We discussed earlier about 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 (eg. 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 non-local 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 earlier, 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 realizing the trade offs between simplicity in testing and flexibility is a mistake.

The overall recommendation for this section is to have simple inputs whenever possible, both in the function definitions and calls. This makes testing a great deal easier.

To illustrate these types of inputs in our diagramming system, let's add some antennas.

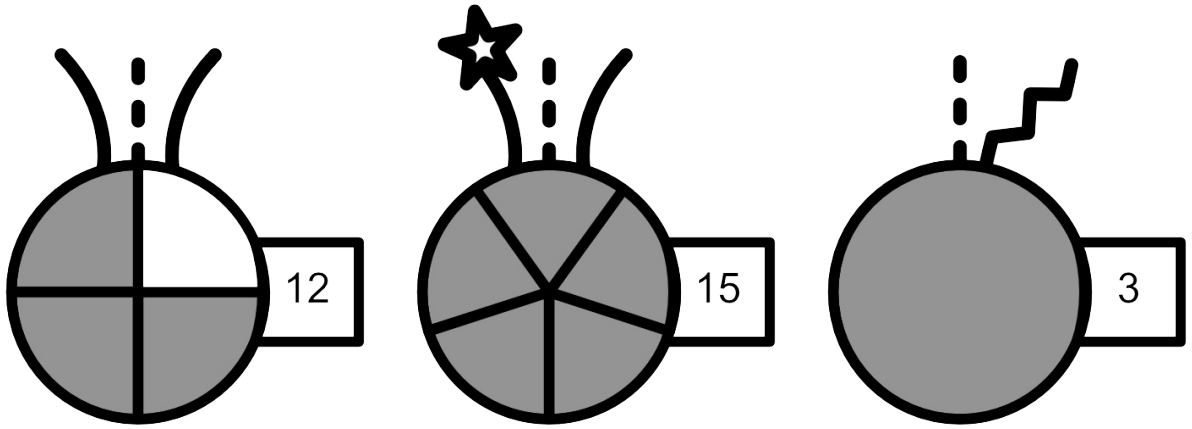

Figure 5-4. Here we have 3 more functions with some new symbols for inputs. But first, from left to right, let's look at their bulk. The first has 12 lines of code and 4 code paths, and 3 of them are tested. The second has 15 lines of code, and 5 tested code paths. The third has three lines of code and one tested code path. As far as inputs, the first one has two explicit inputs (aka. formal parameters), designated with solid lines. It also has one dotted line to represent the implicit input (aka. its "this"). The second function also has one implicit input (its "this") as indicated by the dotted line. It also accepts two explicit inputs, but one has a star on it. This star indicates that the input is complex. It could be an object or array (especially those highly variable ones that tend to be called things like `params` *or* `options`*) or function (a function passed into this function as a parameter) that can radically change how this function behaves. The function on the right has an implicit parameter (its "this") as notated by the dotted line. It does not have any explicit parameters (curved solid lines). It has one jagged line to indicate that it makes use of an non-local input. An example of this type of function would be: var sidesOfDie = 6; function dieRoll(){ Math.floor(Math.random() * (sidesOfDie) + 1); } Of course, there are a potentially huge number of non-local inputs (jagged lines) that are possible to be used in our function. However, we only add one jagged line, because we only make use of one of the* `sidesOfDie` *non-local variable. Of our three kinds of inputs, these are the most difficult to track down. Put another way, it might occur to you that we could also consider the* `Math` *object as a non-local input just for being available, and add a jagged line (or even two: one for the* `random` *function and one for the* `floor` *function). However, the jagged lines are meant to represent non-local inputs that put the function at risk for errors, not ubiquitous utility functions that themselves carry no relevance to the function. A jagged line for math functions is not needed. However, if you make use of a utility function that relies on data outside the function (like the current date), this necessitates a jagged line. If your non-local input is of a complex variety (such as a mystery object called "params" or "options"), feel free to add a star to the end of the jagged antenna.*

with_inputs.png

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 starred antenna (complex inputs) you have, the easier your code will be to understand and test.
- The fewer jagged line antenna (non-local 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 some cases (see the Context and Privacy sections later in this chapter). If `this` is unused, the dashed line is optional.

Outputs

Ignoring the complexity of asynchronous code for the time being, let's discuss outputs. 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`. 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.

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 just if it is just the result or simply the success of the side-effect. 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 or implicitly returning `undefined`.

Secondly, if the style of most of the code base consists of side-effects (covered next), then the return value is less important than that effect. This is especially common in jQuery supported code bases 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.

Generally speaking, we want a decent return value (not `null` or `undefined`), and we want the types from various code paths to match (ie. returning a string sometimes and a number other times means you will probably have a few `if/else` statements in your future. Also returning an array with mixed types can be similarly awkward when compared with returning a simple value or an array of like types.

Some languages prescribe that return values be of a particular type (or explicitly return nothing). As we saw with how JavaScript handles inputs, we should not be surprised to find a similar flexibility for return values.

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 what effect took place. Sometimes that just means returning `this`. Returning something informative, even when nothing was explicitly asked for is a good habit to be in, and helps in testing and debugging.

Now that we have some idea of how return statements work, let's extend our function diagrams to reflect them.

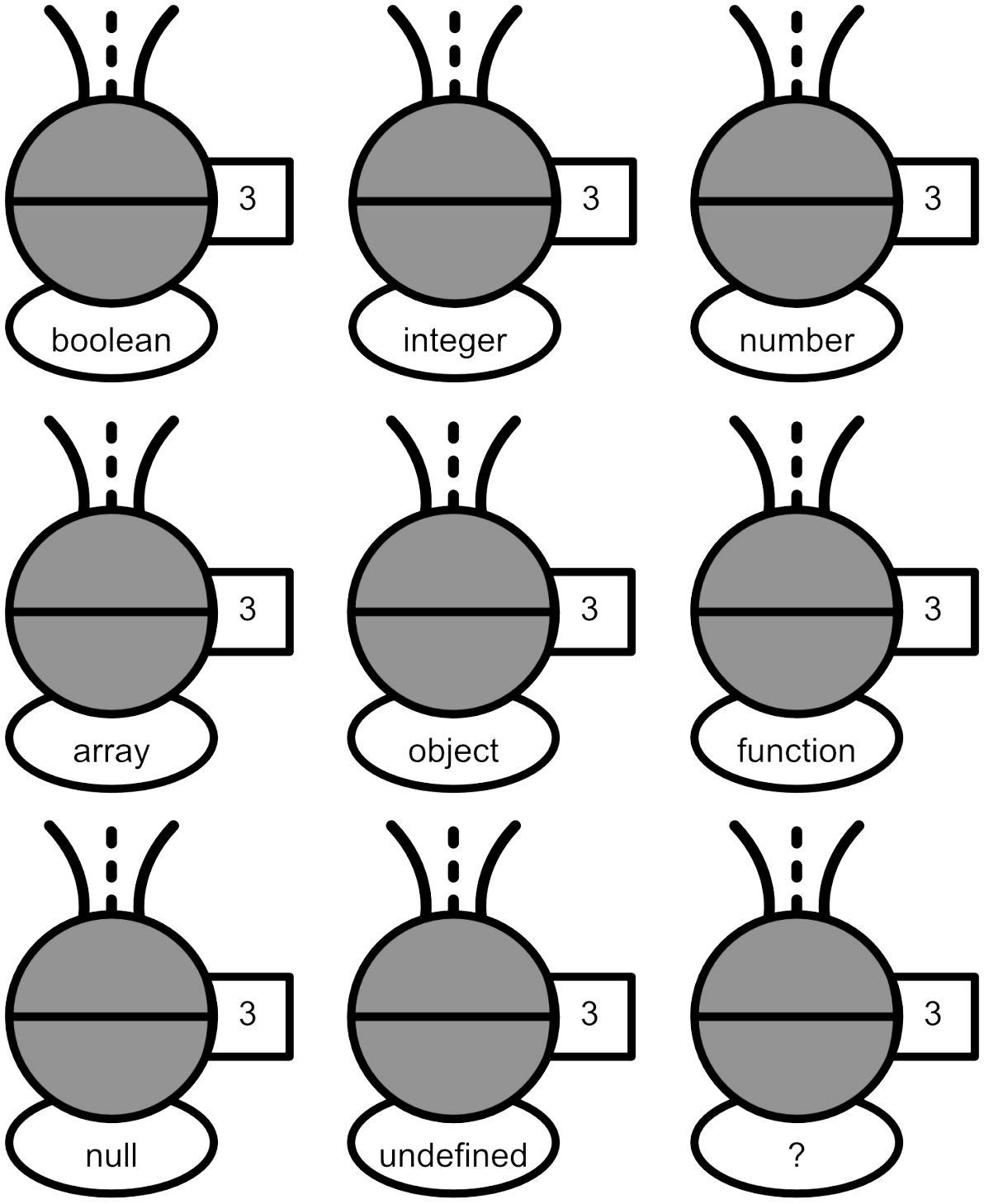

Figure 5-5. Examining the inputs of these 9 functions, we can see that they all take two explicit inputs as well as their "this," the implicit input. As far as bulk, they all have 3 lines of code and 2 code paths (both halves of each circle are grey, indicating that both paths are tested). The new element we have added is the return type in the ellipse at the bottom. Starting with the top row, we can see that these three functions return a boolean, an integer, and a number. Boolean and number are both primitive types in JavaScript. Although "integer" is not a primitive in JavaScript and we could have described this as "number," in our diagrams, we can make our label more specific as to our expectations of the return type, independently of

whether we enforce that specificity in our code or tests. In the middle row, these three functions are returning an array, an object, and a function. As with "integer" being denoted in the top row, we may want to get more specific in all three of these cases. In fact, we're already being more specific than JavaScript's core types, because arrays and functions are actually special types of objects in JavaScript. As for the last row, these are three that we should stay away from in most of our functions. `null` *and* `undefined` *should both be avoided as return types when possible, preferring instead Null Objects (see Chapter 8), raised errors, caught errors, or side-effect result descriptions. What about the question mark in the bottom right? The question mark indicates a function that returns disparate types depending on its inputs. The most common case is returning null or undefined in a failure or "early return" case, and returning a useful value in a successful case. Besides mixed return values being more complicated to think about and test, they also have the likely effect of requiring a conditional to check the type directly following the function call. Note that this idea of "disparate" types does not apply if the return values with different types are still capable of supporting a common interface that is useful to the function-calling code. We'll discuss this more with the pattern refactorings concerning null objects and the state pattern (Chapter 8).*

As is the theme so far for this chapter, simpler is better when it comes to outputs (return types). 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, regardless of which code paths are run in the function.

Side Effects

Some languages find side-effects to be so *dangerous* that they make it 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 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? Two reasons. One is that functions with side effects are harder to test. Second is that they necessarily have some effect on (and/or rely on) state that complicates our design.

We will discuss side-effects more in specific refactoring contexts, but for now, lets update our symbolic diagramming of side-effects to add two types of side-effects.

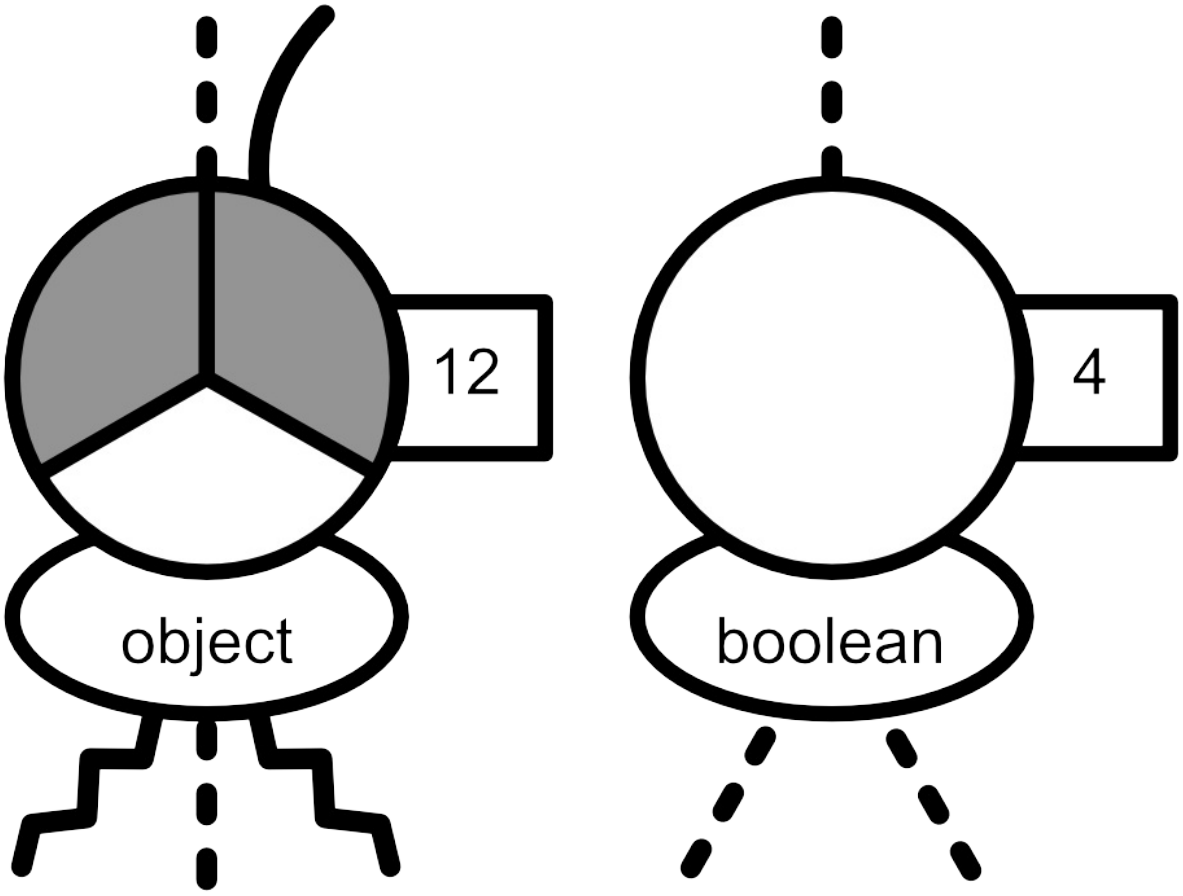

Figure 5-6. On the left, we have a function with 12 lines of code, and two out of three code paths tested. It takes one explicit input, as well as the implicit (this) input, and returns an object. On the right, we have a four line function with only one (untested) code path. It has no explicit parameters, and returns a boolean. As for what's new, the bottoms of these functions now have legs. The dotted lines refer again to the "this" context object, and the jagged lines again refer to some non-local state. The function on the left has one update (or create or delete) to the the context object and two updates (or creation or deletion) of non-local state. On the right, this function has two side-effects, both affecting the "this" object.

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 multiple database rows.

Context Part 1: The Implicit Input

In explaining inputs and outputs, we skipped over something somewhat complex but very important, that we will cover now. The “implicit input,” appearing as a dotted line in our diagrams so far, actually attaches to something. That thing is the "`this`" that we’ve been talking about.

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 return) in a browser’s interpreter. 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 all 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` is equal to 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. This is related to how the node module system works, which we’ll get into later. For now, know that the top-level scope in most contexts 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 name space anyways.

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 the context.

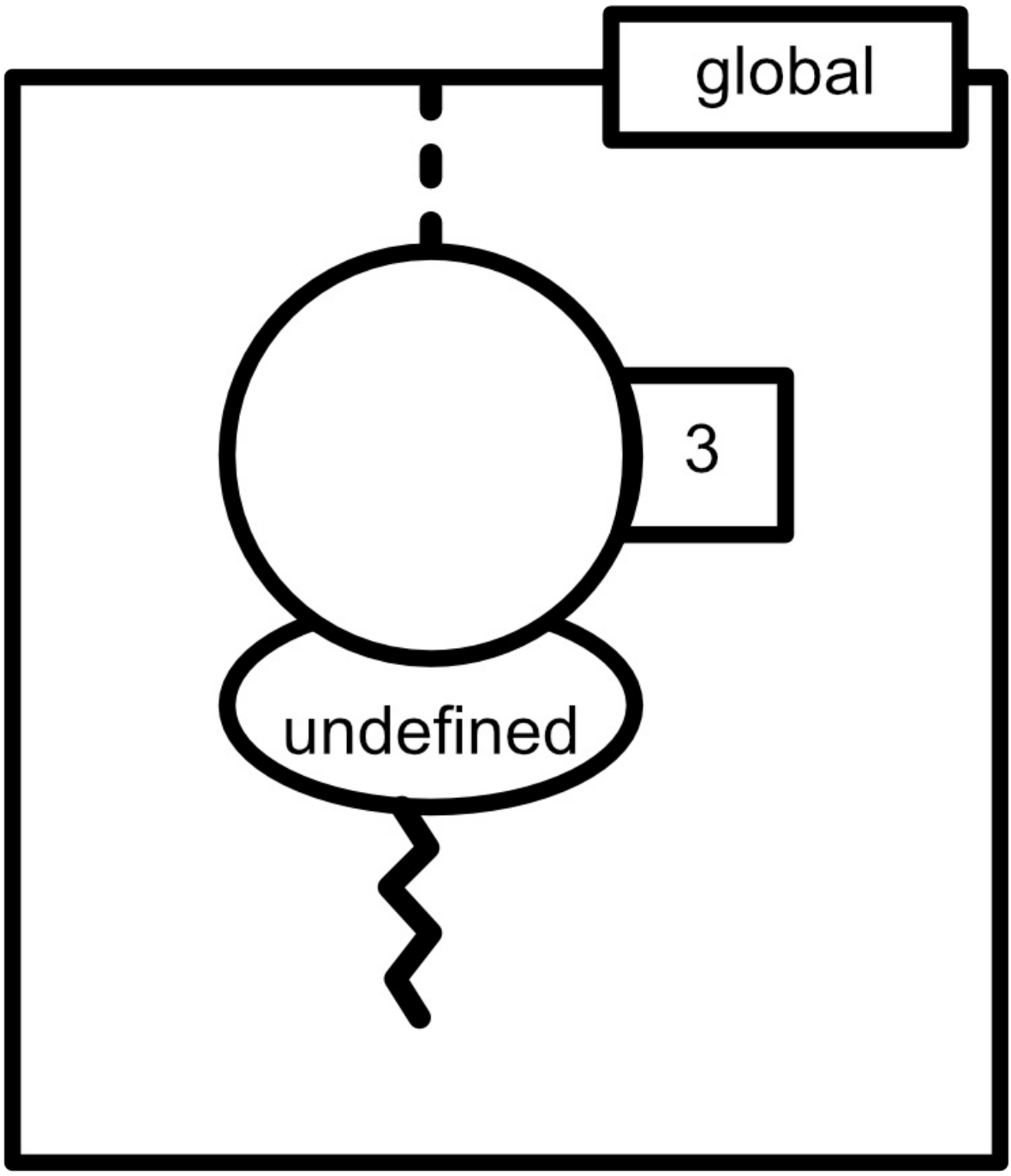

Figure 5-7. Here is our “x” function. Three lines of code with one untested code path. It returns undefined and has a non-local side-effect of printing with console.log. What’s new here is that the implicit input is attached to the global object: The first `this` *that we’ve diagrammed.*

THIS IS 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` the first snippet looks like this:

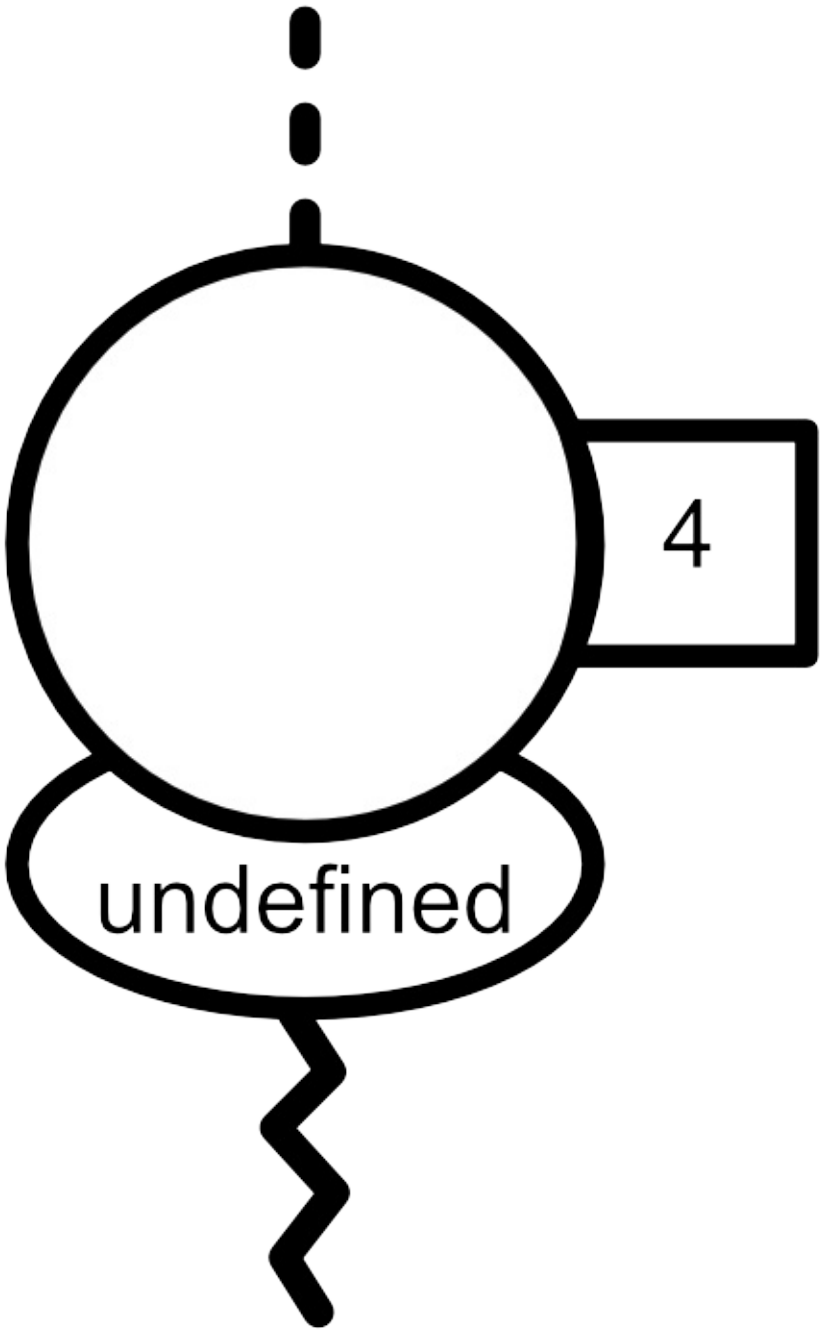

FIGURE 5-8. `'USE STRICT'` ***APPLIED TO X.***

Now the function is 4 lines long, still returns undefined as it did before. The side-effect (which now logs `undefined` instead of the global object) is still present with the jagged line. The most crucial difference is that this no longer attaches to any `this` object other than `undefined`, so there is no containing box for the dotted line to attach to.

The reason the dashed line is there at all is to note that we are referencing `this`, within the function. If it wasn't there, we might think that this function could be attached to an object

(either as an attribute or through `call`/`apply`/`bind`) without changing how the function behaves. This is not the case, so the dashed line actually contains important information, even if it doesn't attach to anything.

Whether you're writing a node module or any decently sized program on the front-end, 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 access anything from the outermost context).

One way to create a new context is by using an object.

```
var anObject = {
  number: 5,
  getNumber: function(){ return this.number }
}

console.log(anObject.getNumber());
```

Here, `this` is not the global, but `anObject`. There are other ways to set a context, but this is the simplest. Because you're creating a literal object, 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.

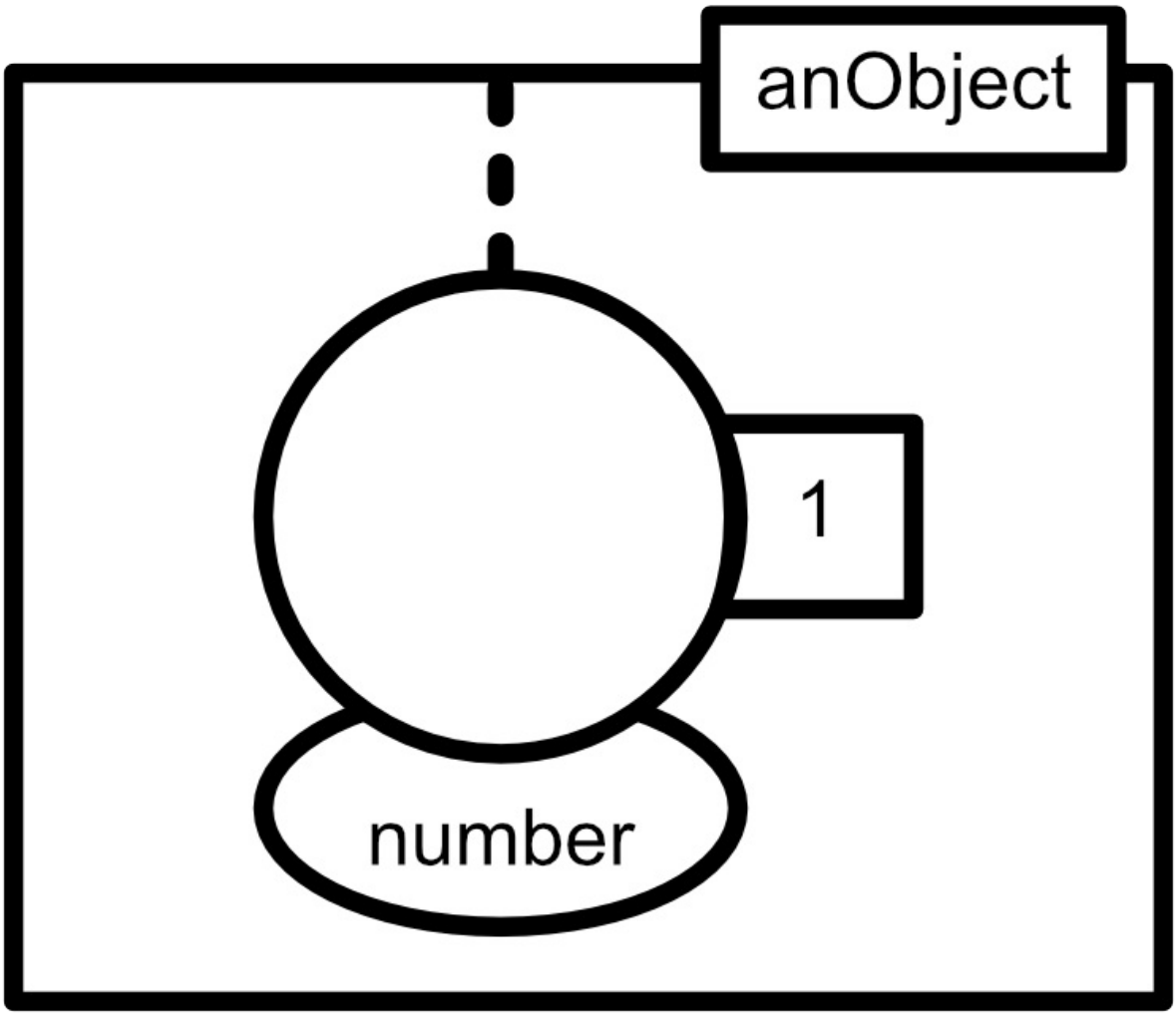

Figure 5-9. One line defines this function. It has a single, untested code path, returns a number, and has one implicit input. The implicit input (`this`*) is the* `anObject` *object.*

We have a few more ways to write code that will follow the diagram. 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 that they obscure 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 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 this book, we use object literals for the simple stuff and classes for the complex stuff, occasionally demonstrating alternate approaches.

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 = {
  getIt: function(){ return this.number }
}
console.log(anotherObject.getIt.call(anObject));
console.log(anotherObject.getIt.apply(anObject));
var callForTheFirstObject = anotherObject.getIt.bind(anObject);
console.log(callForTheFirstObject());
```

Note that neither object has both the number and the function. They need each other. Whether it's `bind`, `call`, or `apply` we're talking about, we'll diagram this a little bit differently.

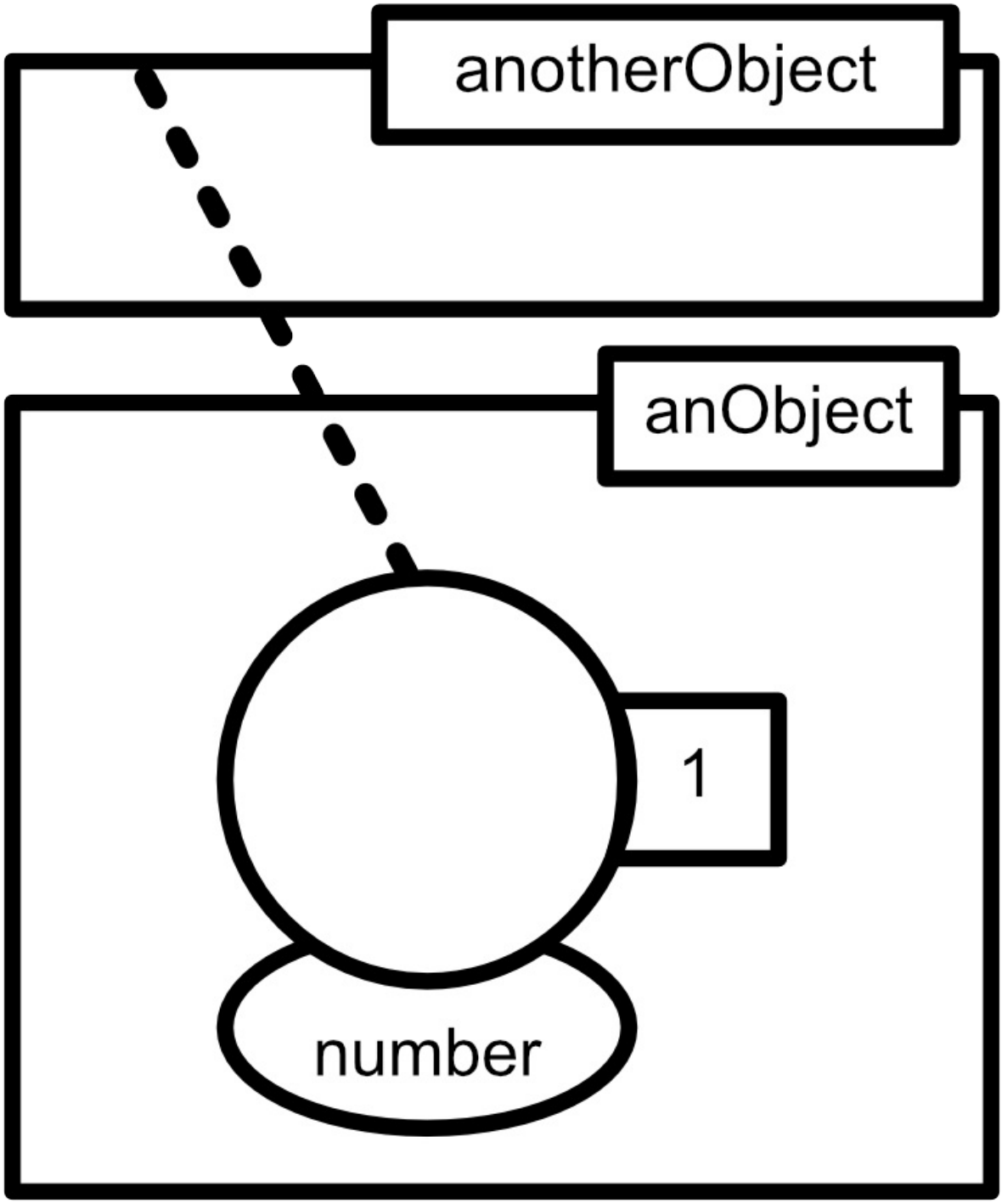

Figure 5-10. A function with one line, that returns a number. One untested code path. What's new here is that, although the function lives in `anObject`*, the use of* `bind`*,* `call` *and* `apply` *are used to assign the "implicit" input in an "explicit" way. So the dotted line no longer attaches to the place where the function lives. It attaches to* `anotherObject`*.*

If a function updates its own `this`, then the dotted side-effect line would simply attach to the bottom of its context box.

Here, we'll demonstrate what happens when you update a different `this`.

```
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 this code returns itself, 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 and no clear confirmation of what happened. This makes testing (manually or automatically) much easier than just returning `undefined` from side-effect causing methods.

Let's look at the diagram for this.

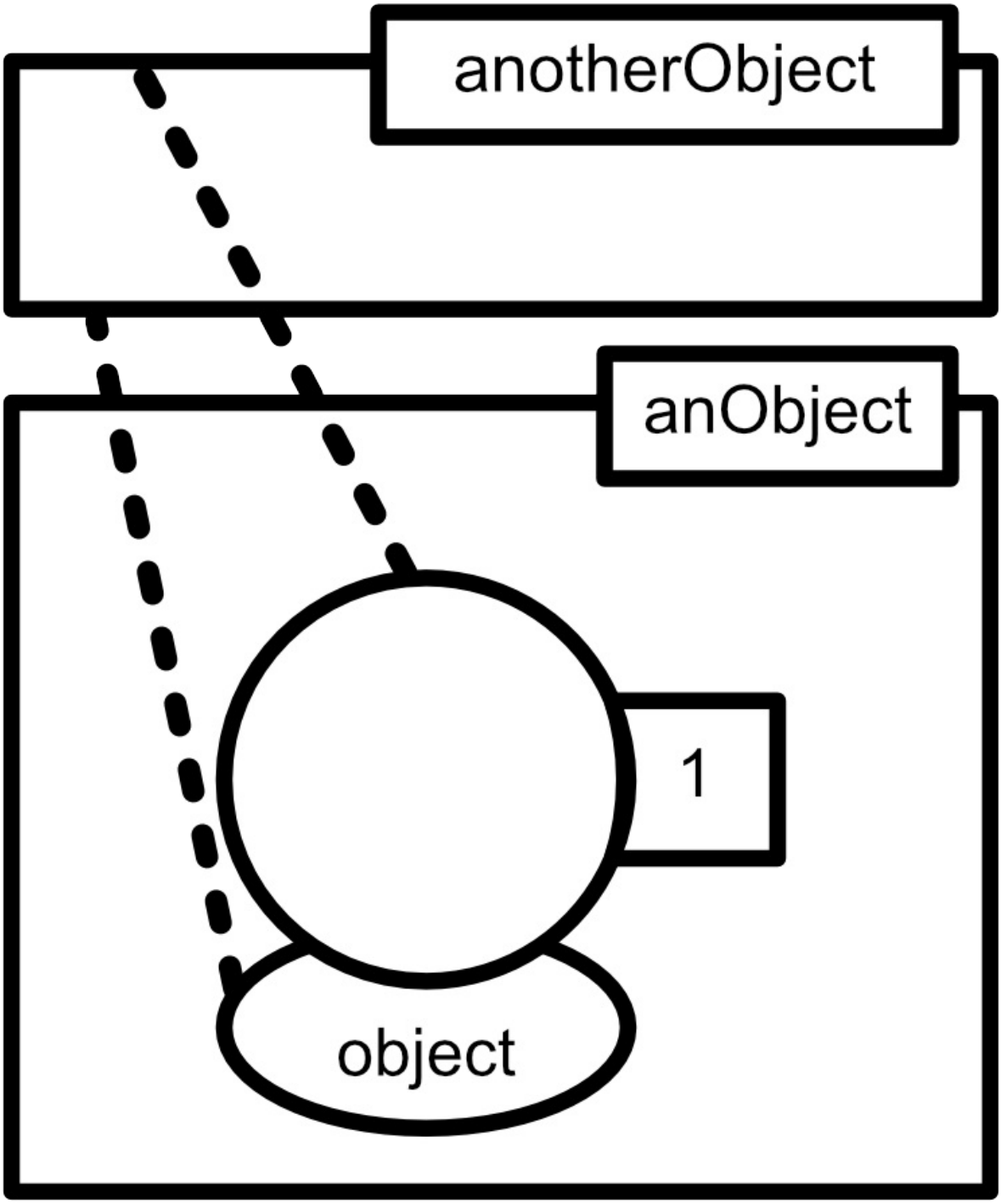

Figure 5-11. Here we still have a function defined inside of the `anObject` *context, with one line and one untested code path. It receives its* `this` *(implicit input) from* `anotherObject` *explicitly by using* `call`*, and causes an update to that* `anotherObject`*'s* `this`*. Finally, note that even though it is a side-effect producing method, we have returned our* `this` *context from* `anotherObject` *to simplify verification and testing, rather than simply returning* `undefined`*.*

side_effect_to_other_this.png

As far as context goes, that's about as complicated as it gets without delving into prototypes (and the 3 or 4 things that means), inheritance, mixins, modules, constructors, factory methods, properties, descriptors, getters, and setters.

After this chapter, it's all about *better code* through *better interfaces*. We won't shy away from those 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.

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. Less tests 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`'s we put inside would 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, because they don't have a name and can't be addressed.

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. Both diagrams for these functions are the same (except for the lines of code, which you might argue is 5 rather than 3 for the second one).

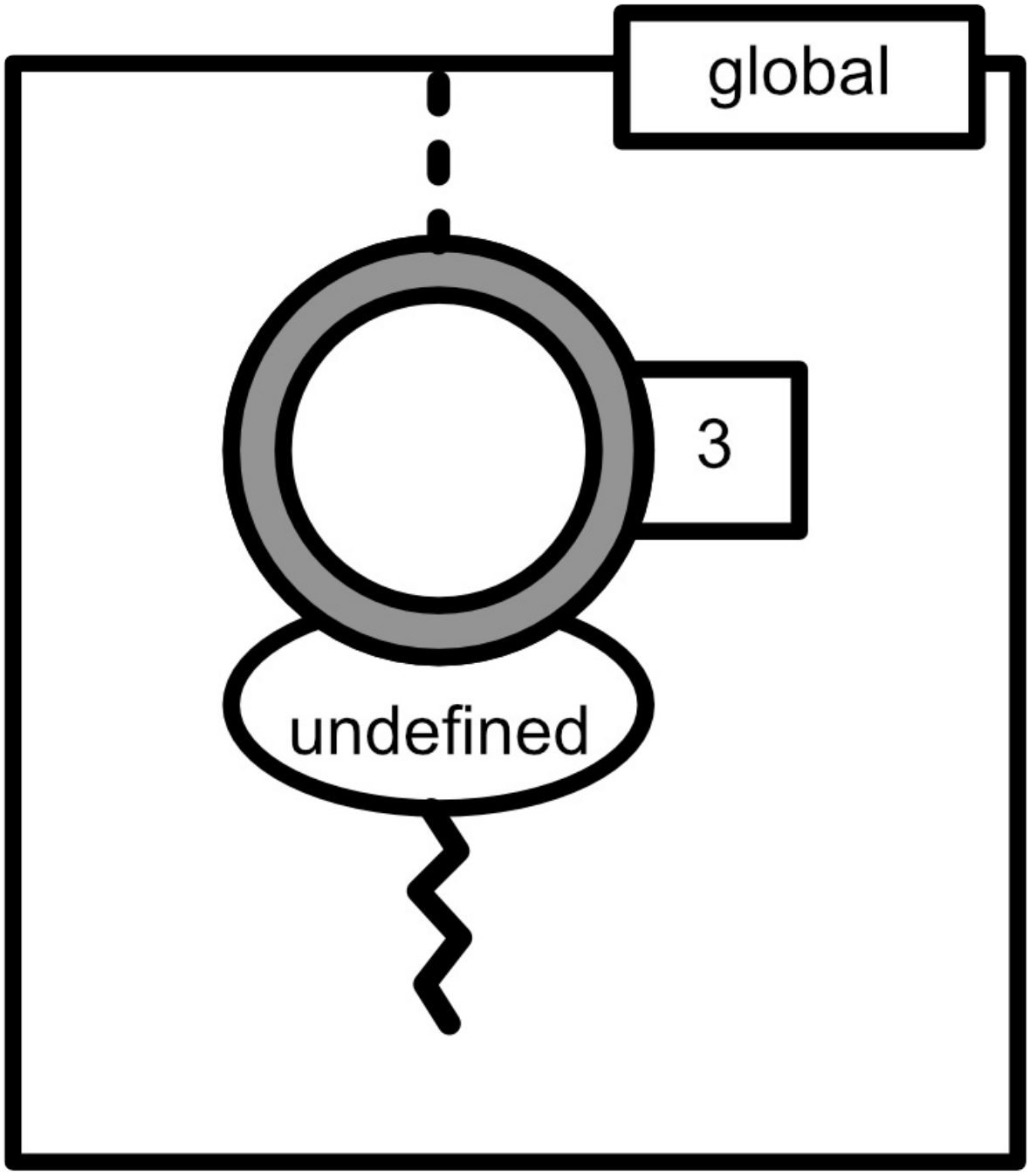

Figure 5-12. A three line function with one untested code path and one non-local side effect. It returns undefined and gets its `this` *from the global object. What's new here is that we have a grey ring around the main circle to denote that this is a private function, and you may or may not want to test it. Any pie slices would still be apparent in a function with more code paths, and as with public functions, they would be greyed out when tested.*

private_function.png

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`. You happen to be surrounding it with 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.

The diary's first property is `open`, which is a boolean initially set to false. Then it has two other properties which map to function definitions provided above. The interesting part is that we have some things hidden in here. Neither the `key` and `secrets` variables, nor the `privateUnlock` function, have any way to access them 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 the global object inside of `privateUnlock`.

What would this look like as a Trellus Diagram? What we end up with is a `diary` object with three properties attached: two functions and one boolean:

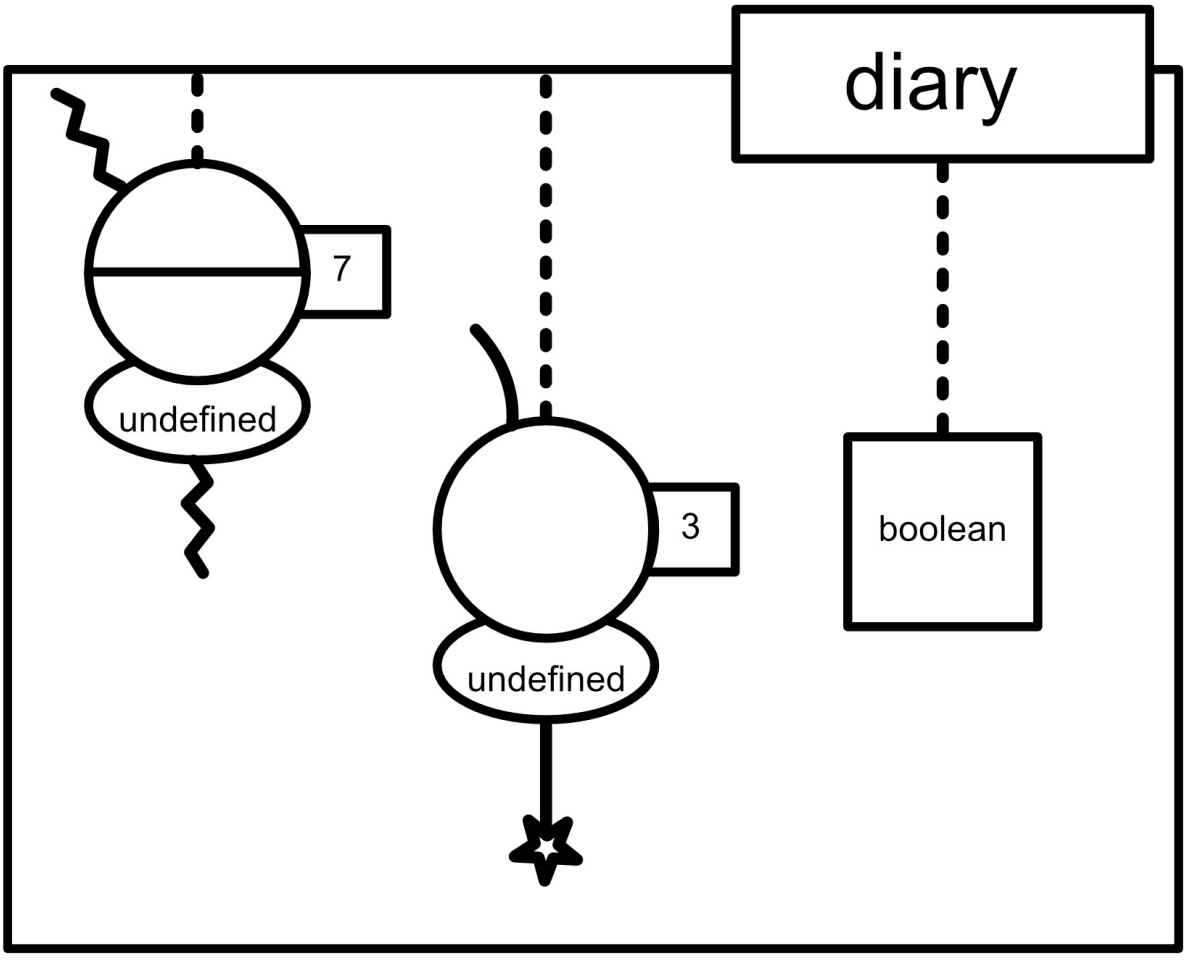

Figure 5-13. This is an image caption

diary7.png

The attribute in the diagram on the right is a simple boolean. It is attached by a dashed line to the `diary` object.

On the left, the attribute diagrammed is the function `read`, which has 7 lines, returns `undefined`, and has a side-effect of printing with console. It has 2 code paths. For parameters, it has one non-local (`secrets`) and the implicit parameter, `this`, which is attached to the `diary` object.

The middle attribute of `diary` diagrammed is the function `tryLock`. It has three lines of code, one code path, and two parameters: `this`, and `keyAttempt` (an explicit parameter). It does not return anything (return value is `undefined`), and its side-effect is a star, which we haven't seen before. This means that the side-effect is to call some other function. If we want to be more explicit about what happens, as a result of that, we need to diagram that inner function call as well:

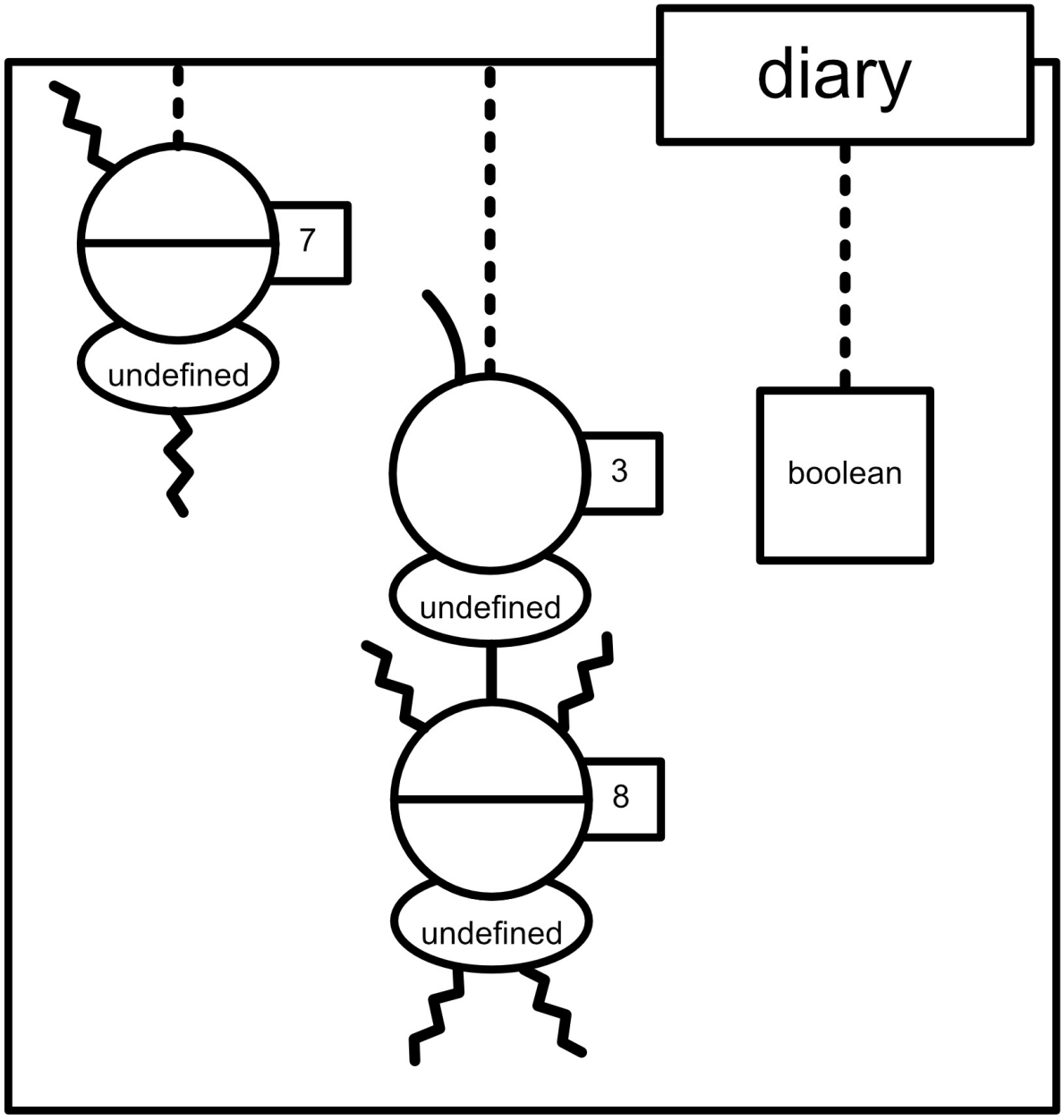

Figure 5-14. With the private function diagrammed

diary8.png

Here we can see that three lines are coming into that function. The straight line is the explicit parameter (`keyAttempt`) passed to `privateUnlock` from `privateTryLock`. It also pulls in two non-local variables: `key` and `diary` (we'll deal with that soon). It's implicit parameter is bound to either the global object in non-strict mode, or `undefined` in strict mode. Because we don't make use of this in `privateUnlock`, we have no dashed line needed as an input to mark the

implicit parameter.

As for side-effects, the first jagged line represents a non-local side-effect (the `console.log`) and the second one is a change `diary`, which to this function, looks like any other non-local variable. In other words, it doesn't know what box it's inside of.

Writing `diary` instead of `this` inside of `privateUnlock` is a little awkward and unnatural. Although it is tempting to just expose the `privateUnlock` function to the object (adding another attribute to the returned object), we won't be able to keep it "private" 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:

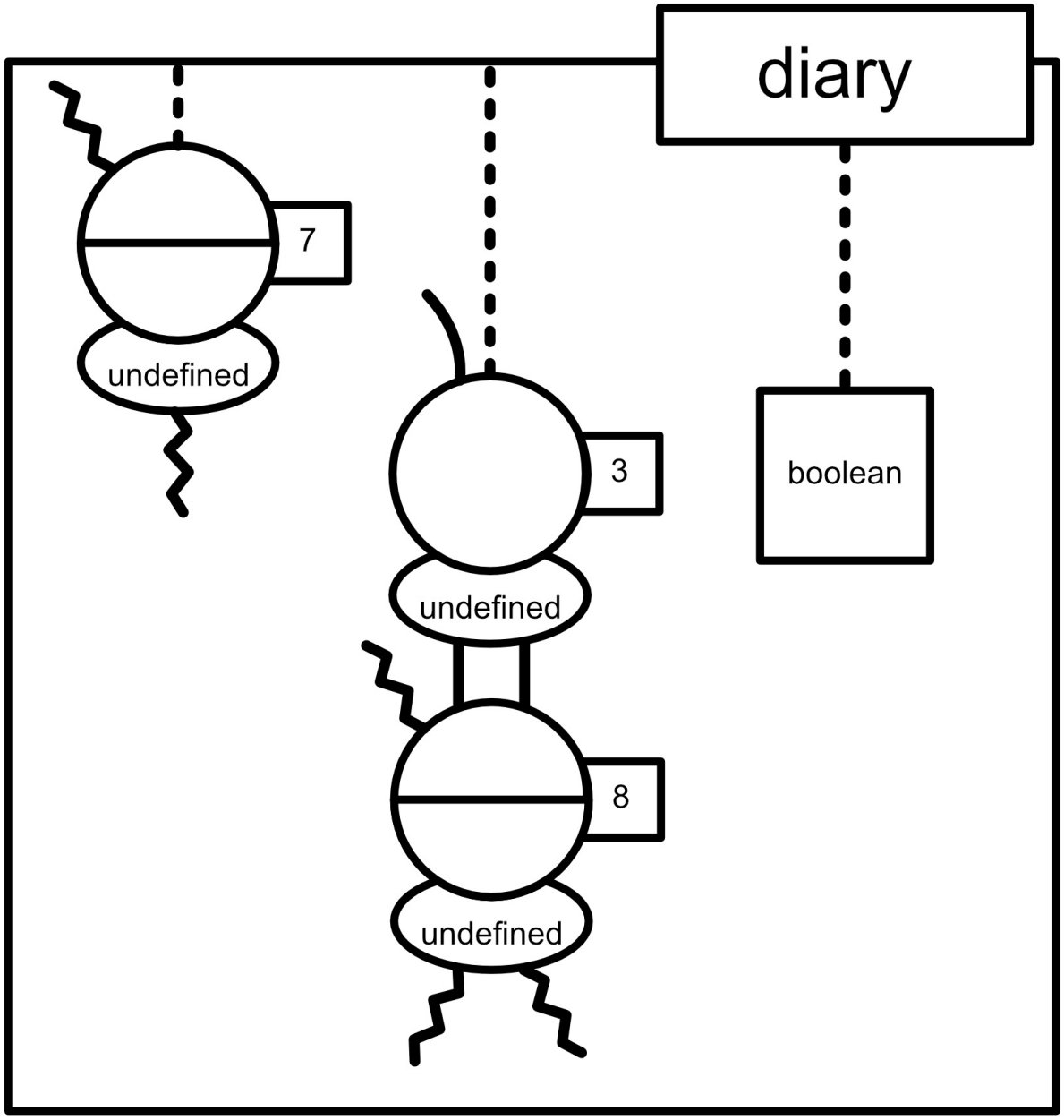

Figure 5-15. This is an image caption

diary9.png

Nothing much has changed here. The only difference is that now we have two explicit inputs to the `privateUnlock` function and one non-local input, which is an improvement from before, but we can do better.

ISN'T `CONSOLE` A NON-LOCAL INPUT?

In other words: Shouldn't it have a jagged line too?

When it is used within the function, yes it does act as a non-local input. We've omitted it in these diagrams to simplify them, but by all means, when you're writing your own functions, add jagged lines for `console` and anything else that's not `this` or passed in as an explicit parameter.

Alternatively, you could 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, how would we diagram our functions with this last bind variant in place?

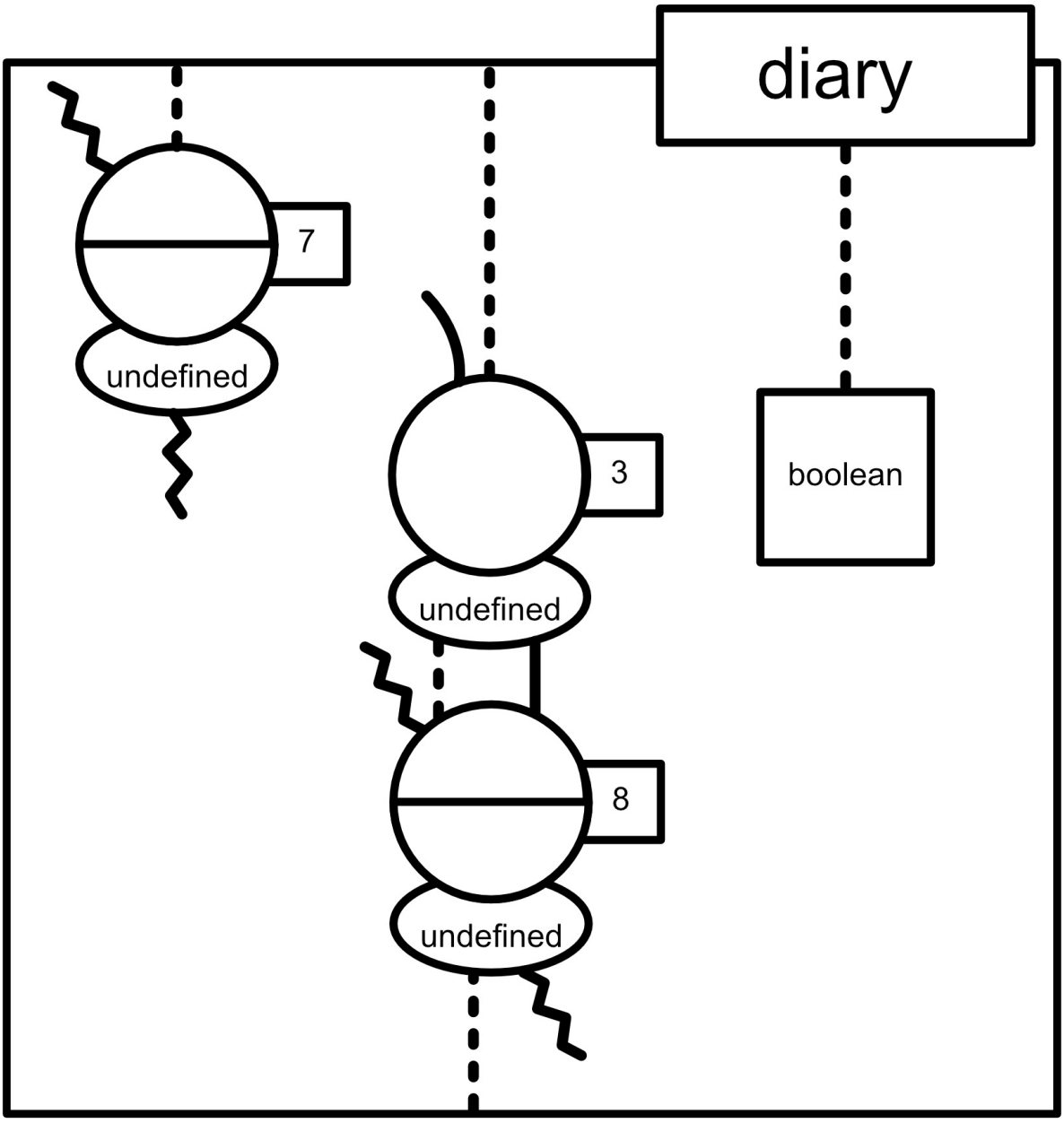

Figure 5-16. This is an image caption

diary10.png

Now instead of two solid lines for parameters, our “private” function (`privateUnlock`) still takes one explicit parameter (`keyAttempt`), but now accepts an implicit parameter (`this`). But notice that there is no direct, dashed line input from `diary` to the second function, so `diary.privateUnlock` is not defined, and only available as a utility to the function `privateTryLock` (which itself only provides the definition to `diary.tryLock`).

Something else has changed. Whereas before, we had two jagged lines representing non-local side-effects, we now have attached a dashed line to `diary`, indicating that we are changing that object, and the `this` of the function. From a diagramming perspective, we could have considered this dashed line before. However, without the `this` chain in place from `diary`, to the top function to the bottom function as input, it was less obvious that we are talking about the same `this` throughout.

You might still be confused about why (without `bind` and friends) `privateUnlock`'s `this` is initially bound to the global context, despite it, `key` and `secrets` all *not* being available from the global context. The answer to the small *why* is that functions create a *closure* where variables can live, and this is independent of the `this` context.

Some variables (including functions) have a useful `this` that they are attached to. Others just have a scope where they are available and addressable.

SO THERE'S NO "PRIVACY" IN JAVASCRIPT?

In the most useful way, not really. Unfortunately, a desire to have an object, and simply declare some attributes as public and other 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, we have 2 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 sub-conventions of doing it in a wrapping anonymous function (a 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 giving visual cues (prefixing the function name with an _ is the most common) to indicate when something is *intended* to be private.

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 things like this, but as of this writing, they're not going anywhere.

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(_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 an implementation detail and thus not need to be tested, 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 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, we've created global variables! What good is that?

Well, this is actually very close to something great that solves our problem.

```
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 to import the module. Here's what that file looks like:

```
const Diary = require('./diary.js');
let d = new Diary();
d.tryLock(12345);
d.read();
```

In *this* file, between the `Diary` "class" or `d`, the "instance," 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 putting our "private" functions in their own modules somehow, or conditionally including them in for tests (and excluding them otherwise).

Conclusion

In this chapter, we covered a lot of detail about how JavaScript works. We centered our conversation around 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
- Prefer implicit inputs to explicit inputs
- Prefer explicit inputs to non-local inputs
- 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 non-local 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 all of this advice helps in making the code simpler, as well as the trellus diagrams, if you should choose to draw them.

Chapter 6. Single Object Refactoring

Over the next two chapters, we will be dealing with code that is messy, undertested, and does something cool.

The project is a Naive Bayes Classifier (NBC). The first part that is cool is that if you have interest, but lack experience, in machine learning, this particular algorithm is fairly simple and still very powerful.

The second cool thing 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.

> **BUT I DON'T KNOW ANYTHING ABOUT MUSIC!**
>
> That's ok. 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.

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.

There are a few things we won't be covering to a great deal. JSHint and JSCS (now ESLint) checker criteria (of which there are hundreds) will not all be

specifically addressed. It is recommended that you try these tools out in your editor. More specifically, we won't be covering single quotes vs. double quotes, ASI (automatic semicolon insertion) and "unnecessary" semicolons, or the amount 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 "living style guide," but covering all those here would neither be interesting or 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 trite as the Uncle Ben quote in the tech book, 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, and 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 and 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 implicit inputs to explicit inputs
- Prefer explicit inputs to non-local inputs

- 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 non-local 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 we go:

```
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 about 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 develop confidence through testing and refactoring.

That means we need to get this 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 decide on a testing strategy.

Our Testing Strategy

Assuming you're all set up with the code (---link to code) in a file called "naive-bayes.js" and version control, let's get started.

Back to our model of what types of tests to write when (diagram in chapter 4), we know that we need characterization tests. 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, and we should see the following output:

```
(from command line)
>node naive-bayes.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 }
```

The downside, as mentioned, is that print statements are our only output. 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. 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.

WHERE SHOULD WE START?

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 the techniques presented in this chapter.

We delay extracting functions in this chapter for three reasons:

- This chapter is roughly arranged from simple to complex techniques, and other refactoring approaches are simpler.
- 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.

However, features and training sets will need to be added, and if you'd rather discover what movies you should watch next, instead of what song you should learn to play next, the changes would be more significant.

Back to the question of confidence (the result of our manual test), we need one more step: run it again (maybe twice). And the result is still the same. That's great. Our algorithm is *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.

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
- non-descriptive/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 with a number and bucking the camel convention: `song_11`, which is just an empty array. To avoid numbers and snake case (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 is following the snake_convention, rather

than the camelConvention: `somewhere_over_the_rainbow`. The fix is the same, search/replace/save/verify/commit.

Next is a tougher case, and some may disagree, but using single letter variable names doesn't make sense unless you have some insane memory requirements (although these are normal in compiled js, as build processes may shorten labels). `i` and `j` appear throughout the program, with their one 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 into a manual search and replace process. 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 won't help you if they are reused in various places.

Second, although these variables happen to be in the expected scopes, having variables that are not unique creates 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 that by changing one of these variable names, we aren't introducing a bug.

The third and worst thing about them is they give no information about what is inside. It is somewhat conventional (a bad one as described above) 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. Save, check, commit.

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`, and each instance of `j` in that function should be `chord`.

`setProbabilityOfChordsInLabels` has `i` and `j` variables as well. In this case, `i` is more appropriately called `difficulty` and `j` is better named as `chord`.

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. In either case, 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 arrary indexes, the name `index` (and `innerIndex` where necessary) are still preferable to single letter variable names. If what you discover are string keys to an object (eg. `"easy"`, `"medium"`, `"hard"` in our case for the `i` we renamed as `difficulty`), you should rely on domain knowledge to chose an appropriate name. In those cases, don't worry if your first guess is imprecise. As you gain domain knowledge and gain confidence in how the program works, you can always rename things again.

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 out 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 totals = labelProbabilities;
  console.log(totals);
  var classified = {}
  Object.keys(totals).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);
};
```

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, we'll see that "easy," "medium," and "hard" are what is printed when we run the program. We've been calling these "labels" in most other parts of the program, but have just renamed a similar concept as `difficulty`.

It's not always this easy to find appropriate names for things without the 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`. Keep in mind that changing all instances of `obj` is easy in this case because `obj` is confined not only to this file, but also this function.

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), feel free to do so, but for now, we'll proceed assuming that those names have not been changed.

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 one, this 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
- comments
- whitespace (including EOL, EOF)
- do nothing code (reachable, but has no effect, eg. `$($('.someClass'))` in jQuery or if(!!booleans), empty files)
- debugging/logging statements

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. 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.

Can you find any instances of dead code in our NBC?

There are actually three variables that we can elimate. Neither `army` nor `blankSong` are 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).

Additionally, we can delete our first line: `var fs = require('fs')`

Apparently, there was an intention to include and make use of the file system module at some point, but it was never realized. If you are not actively working on some file system based feature, then this should go.

But don't just comment it out. Why not?

Sometimes you'll see comments that are intended as future code (a stub, psuedocode, 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 ToDos (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" (Ya Ain't Gonna Need It) 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.

Speaking of which, that `total` variable that we spent a bit of time earlier renaming is due for deletion. Why? All it does is receive an assignment and log. 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 the program.

While on the topic of comments, there are two useful approaches. 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 variable 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 preceed 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 is 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.

The whitespace in our code seems alright for the most part. Before the `classify` function, there are three lines, two of which can and should be deleted. Beyond extra blank lines, you may see trailing white space at the end of a line. 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. Generally, this seems like a good idea (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).

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 non-empty strings are “truthy” in JavaScript.

```
if("print hi"){ console.log('hi')}
```

There are six “falsey” values in JavaScript: `undefined`, `null`, `0`, `""` (the empty string), `NaN`, and `false`. Applying `!!` of these will produce a `false`, whereas other strings, numbers, objects, functions, arrays, etc. 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 way to use it appropriately 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 Ruby, with a calculation that involves an integer values of `10` and `3`, at least one value has to be converted into a float first. This

can be done by multiplying one of the terms by `1.0`.

In JavaScript, *floats* and *integers* are both just *numbers*, so both of the above 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 them:

```
probabilityOfChordsInLabels[difficulty][chord] =
probabilityOfChordsInLabels[difficulty][chord] / songs.length;
```

Where else does unnecessary code like this pop up? We don't have an example of this in our classifier, but another way code can double wrap itself is through 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 to a boolean, a float, a jQuery object, or something else, you'll see these "multiple/uneeded casting" efforts every once in a while.

We'll look at unnecessary variables more in the next section (we actually already had one ealier when we "inlined" the `labelProbabilities` function call and removed `total`), but notice that here, `elementToHide` is not actually needed with the following change:

```
$('input').on('click', function(){
  $(this).hide();
});
```

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're safe to simplify this code to be 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 a way to eliminate this conditional altogether (the state pattern), which we'll look at in Chapter 8.

The last type of useless code is debugging/logging statements. Usually these show up as things we forgot to delete. Currently, we're relying on these for our manual testing, but once we get into refactoring our functions, we'll take care of them properly.

Variables

Now that 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.

- making magic numbers (and strings) disappear

- Fixing long lines with variables or special operators (part 1)
- inlining function calls
- Removing unnecessary variables (again)
- caching variable instead of multiple calls
- declaring variables where they are hoisted to

“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 too magical 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 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 it’s still better than having a 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 here and admit that this variable should be called “smoothing.” Basically, it helps to keep zeros from blowing up our algorithm. 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 to 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 being not changing it, and missing a cool feature in an otherwise awesome game (looking at you DuckTales).

On a related note, magic *strings* can be just as bad, or worse. When user-facing strings are hard-coded, 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 in them that they are not prone to be reusable. 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.

Next up: fixing long lines by adding variables, part 1 (we'll cover other ways later).

```
probabilityOfChordsInLabels[difficulty][chord] =
probabilityOfChordsInLabels[difficulty][chord] / songs.length;
```

This line is too long. To shorten it, we *could* introduce a new variable with a descriptive name:

```
var chordInstances = probabilityOfChordsInLabels[difficulty][chord];
probabilityOfChordsInLabels[difficulty][chord] = chordInstances / songs.length;
```

However, we can make use of a shorthand function. If `/` was a more complicated operation, extracting a variable as above would make sense, but for this, we can make use of JavaScript's `/=` operator.

```
probabilityOfChordsInLabels[difficulty][chord] /=  songs.length;
```

This is equivalent, 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, in the more familiar `+=` operator:

```
chordCountsInLabels[song[0]][chord] += 1;
```

Then we save, run, check, and commit.

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 it's 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 explicit parameters to `getNumberOfSongs` would need to somehow act as inputs for `setLabelProperties` as well. Since in this case, it only relies on the non-local variable `songs`, no additional changes are required to make it available. If `getNumberOfSongs` took explicit parameters or made use of `this`, our situation would be slightly different.

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;
  })
};
```

Oftentimes, when you inline a function call, you'll find that it is just being set as a variable (`numberOfSongs` in this case) and used somewhere else. *Especially if it is only used once*, then you have a 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?

Now let's look at the opposite case of *inlining* a variable, which is *extracting* a variable. Hopefully, you don't see much code like this to print out an array:

```
console.log(someArrayReturningGetterFunction()[0]);
console.log(someArrayReturningGetterFunction()[1]);
console.log(someArrayReturningGetterFunction()[2]);
console.log(someArrayReturningGetterFunction()[3]);
```

But a shocking amount of people will find themselves doing something similar in jQuery.

```
$('#someDomElement').css('width', 5);
$('#someDomElement').css('background-color', 'red');
$('#someDomElement').show();
```

Some jQuery DOM selections are expensive. For that reason, the following is preferable:

```
var domElement = $('#someDomElement');
domElement.css('width', 5);
domElement.css('background-color', 'red');
domElement.show();
```

This is called introducing a "caching" variable, because the code only has to perform the query one time. But 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 let's 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 9.

ENOUGH JQUERY ALREADY!

It's dead! No one uses it. React and Ember and Meteor and Angular and Vanilla all make it obsolete.

Although personally, I think jQuery is still relevant and not that bad for certain things, there are three practical reasons for considering it.

- 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.
- You can make these mistakes in other frameworks, or even on the backend (repeating the same SQL query for instance)

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 Scoping section of 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. In actuality, each variable is declared (and assigned) right at the top of their functional scope. 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]
...
```

Since anonymous functions *also* create scopes, `first` is declared at the top of its

function scope, and so is `probabilityOfChordInLabel`.

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){
  ...
};
```

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 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.

Strings

In this section, we're looking at refactorings that we can use for strings. Here are the things we'll cover:

- "adding" strings
- template strings
- regex
- long lines part 2: long strings

Go to the previous section for the discussion of "magic strings."

We've decided that when we run our code, we want to output "Welcome to " plus the name of our file.

Let's start by adding this code to the top of our file:

```
console.log('Welcome to naive-bayes.js!');
```

This will work fine, but the file name does remind us of a Magic String, doesn't it? We can separate the parts out with the + operator.

```
console.log('Welcome to ' + 'naive-bayes.js' + '!');
```

Then we can move our Magic String into a filename variable.

```
var fileName = 'naive-bayes.js';
console.log('Welcome to ' + fileName + '!');
```

Note that if you want to use this for our current file, that this additional output breaks our manual test in that it adds output that was not previously there.

And one last tweak we can make here is to use "template strings," instead of concatenating with the + operator.

```
var fileName = 'naive-bayes.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 'naive-bayes.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__` that is 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/mine.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
```

By the magic of:

```
typeof theError === 'object';
Object.getOwnPropertyNames(theError)
typeof theError.message === 'string'
typeof theError.stack === 'string'
```

you'll find that `theError` has two properties: a `stack`, and a `message`. Both properties are strings. `stack` is what we're interested in, specifically the filename in it.

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 `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");
  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()}!`);
var easy = 'easy';
```

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 vs. 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 and dashes (`[\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 (known as the "capture" in regex terms) vs. what we use to match the pattern overall.

Save/Run/Check/Commit.

There's not too much more to say about Regex for refactoring. 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 you feel queasy, and then use a regex.

One thing 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];
};
```

On to our last topic, we 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 limit? 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 hold 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 later sections, we'll talk about extracting functions and sensible places to break up arrays and objects.

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.

```
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."
```

And that was 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.

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 get the title "solution 5."

As for the first three, your options are all some combination of awkward and brittle. Using option 1 with a text editor that wraps clearly is the best you can hope for in some cases. As for two and three, two comes with a performance hit, has more syntax, but is slightly less brittle. If you want your string to have new line characters, go with template strings (solution 4), even if you don't need the capabilities of template strings to allow for interpolation of variables.

ABOUT 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, 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 white space 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. 80 characters is pretty aggressive, especially for codebases that use 4 (or 8!) spaces to indent.

Containers and Iteration

Containers (arrays, objects, sets, and maps) and iterating through them are fundamental concepts in programming JavaScript. In this section, we’ll explore more nuanced options JavaScript has available for those who might be a little too dependent on arrays and for loops. Here are the topics that we’ll cover:

- Line too long, part three
- Can’t find the right loop
- Eliminate loops
- Did you really want an Array? Sets
- Did you really want an Array? Objects
- Did you really want an Object? Maps
- Weak Stuff
- bit fields

Getting back to our classifier, we actually have a song with enough chords that discussing long lines is relevant again.

```
paperBag = ['bm7', 'e', 'c', 'g', 'b7', 'f', 'em', 'a', 'cmaj7', 'em7', 'a7', 'f7',
'b'];
```

Our “Solution 1: just let it wrap” from the last section would still work here, but 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, 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 do 3 * 4 + 1 instead of just counting.

But this isn’t a book about style. It’s about refactoring. We’ll make that change, but leave other arrays alone. Feel free to play around with different approaches.

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 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 an 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" or "for index in danger of being wrong." (see following warning on for...in)

FOR...IN CAVEATS

1. Unlike a normal for loop with explicit indices, indices in a for...in loop are not guaranteed to be in order.

2. Any properties that are enumerable will be enumerated. This sounds tautological, but arrays could inherit “enumerable” properties from other places (eg. `Array.prototype.customFunction`), or have them directly set a la `myArray.cool = true`

3. 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`.

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);
  }
});
```

All that work, and although they’re shorter, we only were able to get rid of one line altogether (`var index;` at the top). 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. We will talk about this more in the functional section, loops are driven by the idea of doing something a bunch of times. But what we *really* want to do is create new sets of values to work with and then apply functions to them. There are a lot of functions that look like `forEach`, but are latent with side-effects. We’ll look at those next.

Leaving the context of our classifier for a bit, here are two ways to assign elements to arrays.

```
//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 the functional chapter.

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 nuclear holocaust, two things would survive: cockroaches and people arguing 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 is slowing down your code, but if it is, fix it.

Not only are there many types of loops to chose from, Arrays have some alternatives as well.

First up, "Sets". A `Set` is like an array, but can only hold *one* of a certain value. Although they are iterable, like arrays, their interface is very different. Because they can only hold one of something, we can just try adding to them without checking conditionally.

So with a `Set` instead of an array we can turn this code:

```
var allChords = []; //this is outside 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
chords.forEach(chord => {
  allChords.add(chord);
});
```

It saves 2 lines and a conditional check. Win win. Make that change now.

Another thing you might want instead of 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.

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, like an object. One could argue that we're now changing behavior and beyond the scope of refactoring by making these changes. On the other hand, our output is only for visual inspection at this point, and still looks correct, and if anything, is improved a bit.

We have two arrays left, `songs` and `labels`. Looking 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 two things are 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 thing to notice about `songs` is that "arrays" are pushed onto it, but these "arrays" just have two elements, neither of which are 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 on it should be objects. This involves a few changes.

Inside the `train` function, what we push will be different:

```
// get rid of this
songs.push([label, chords]);

// do this instead
songs.push({label: label, chords: chords});
```

Now we're using an object instead of an array. This will break a lot of things. Fortunately, since we haven't changed `songs` itself, the calls from it (`length` and `forEach`) will still work fine. Inside of 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`. This involves the following lines:

```
-    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 `-` means a line to be deleted and replaced with the lines with the `+`. Lines with no plus or minus are just provided for context.

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 `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 heavier weight 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") class system (along with modules) are taking over the larger architectural duties, whereas maps are intended to fulfil the lighter weight "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, eg. 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.

On the other hand, the `.get` and `.set` notation may seem foreign (to you or other members of your team) or inconvenient when you're dealing with especially deep nesting of maps based on objects. For those reasons, we will end up leaving some internal objects as they are. Also for those reasons, when you get data from some remote API, you're likely to get back JSON ("JavaScript Object Notation"), which means you're dealing with objects there. Converting those to maps is probably not worth the trouble most of the time.
The easiest object to convert to a map is `classified` inside of 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);
```

And 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:

```
// change 1
-var labelCounts = {};
+var labelCounts = new Map();

// change 2
-  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);

// change 3
-    labelCounts[label] = 1;
```

```
+    labelCounts.set(label, 1);

// change 4
-  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 first our `git diff` notation where `+` is an added line and `-` is a deleted one.

So 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 get the array of labels with our old object, the `Object.keys` function is used, taking a parameter of the `labelCounts` object. 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 needing it need to look up the function definition to realize what the first parameter means or use `console.log` to find its value? Or should they just delete the underscore that prepends a perfectly useful variable name that is descriptive? I’d argue the latter.

WHY DID THEY DO THAT?

The `forEach` function of `Map` ordering the params is backwards to how

people usually think of hashes/dictionaries: “key/value pairs”

It might be nicer if it was `(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)` vs. `[thing]`. And setting attributes: `.set(thing, newValue)` vs. `[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;
```

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.

After applying the same approach as before, albeit 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(function(chord){
-    probabilityOfChordsInLabels[difficulty][chord] /=  songs.length;
+probabilityOfChordsInLabels.forEach(function(_chords, difficulty){
+  Object.keys(probabilityOfChordsInLabels.get(difficulty)).forEach(function(chord){
+    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”, ie. available for garbage collection when they don’t have any references

Basically, with the weak forms, you give up 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.

One more candidate for replacing arrays deserves a mention: bitfields. If you have an array that stores booleans, it could be a candidate. In a way, there is no native implementation of bitfields in JavaScript. In another way, you have access

to numbers and bitwise 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 peformance optimization that refactoring, because bit-wise arithmetic is super fast, but 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.

Extracting Functions

Finally! As stated towards the beginning of the chapter, this section is about what is likely the most useful and underutilized refactoring technique: extracting functions.
For a lot of our changes, we could limp along without needing to pull in a testing framework. And that is a huge dependency, so knowing what you can do without one is nice. We've made some improvements to our code, and maybe picked up some understanding of it along the way. However, it is still an unstructured mess. We'll tackle that with functions, but first, we need to get our tests in place.

As a first step, add the following code to the bottom of your file:

```
var assert = require('assert');
describe('the file', function() {
  it('works', function(){
    assert(true);
  })
})
```

Now if we run with `mocha naive-bayes.js` (don't forget about using `mocha -w naive-bayes.js` to run the watcher in a separate terminal window) we should see a passing test as well as our logging information.

The next step is to have our functions that use console.log at the end actually return something that we can test. Add this inside the describe block (the classify line is the same as from 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 as well as log:

```
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 make the assumption that all three elements of the map that is returned are `null` and have mocha correct us. Make the "classifies" test look like this:

```
  it('classifies', function(){
    var classified = classify(['f#m7', 'a', 'dadd9', 'dmaj7', 'bm', 'bm7', 'd',
'f#m']);
    assert.equal(classified.get('easy'), null);
    assert.equal(classified.get('medium'), null);
    assert.equal(classified.get('hard'), null);
  });
```

After running mocha, this leads to:

```
AssertionError: 1.3433333333333333 == null
```

And then we can just put that value into our test:

```
assert.equal(classified.get('easy'), 1.3433333333333333);
```

Run mocha again and:

```
AssertionError: 1.5060259259259259 == null
```

Perfect. Again, we replace the null with the number in our test:

```
assert.equal(classified.get('medium'), 1.5060259259259259);
```

Save, run mocha and, one more time:

```
AssertionError: 1.6884223991769547 == null
```

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']);
  assert.equal(classified.get('easy'), 1.3433333333333333);
  assert.equal(classified.get('medium'), 1.5060259259259259);
  assert.equal(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']);
  assert.equal(classified.get('easy'), 2.023094827160494);
  assert.equal(classified.get('medium'), 1.855758613168724);
  assert.equal(classified.get('hard'), 1.855758613168724);
});
```

Now we can remove our logging statement (`console.log(classified);`) from the `classify` function.

You might be wondering why we didn't just copy the values from the console.log and put them in the test. The reason is so that 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 null in the assertion to the value being returned from the function, and that turned the test from red to green."

While we're getting rid of logging statements, let's add a new test for the welcome message:

```
it('sets welcome message', function(){
  console.log(`Welcome to ${fileName()}!`);
});
```

Right now, this passes, as there is no assertion made. Let's add the assertion now:

```
it('sets welcome message', function(){
  console.log(`Welcome to ${fileName()}!`);
  assert.equal(welcomeMessage(), 'Welcome to naive-bayes.js!')
});
```

Note that we're intending to add a function here. This is not a characterization test. This is a unit test for a new function.

We get an error:

```
ReferenceError: welcomeMessage is not defined
```

Great. Let's define that function:

```
function welcomeMessage(){
  return `Welcome to ${fileName()}!`
}
```

And the test passes, which means we no longer need to rely on the welcome message logging statement anywhere, including our test:

```
it('sets welcome message', function(){
  assert.equal(welcomeMessage(), 'Welcome to naive-bayes.js!')
});
```

To take care of the remaining two "sanity check" logging statements, let's first move them into the tests:

```
it('number of chords', function(){
  console.log(allChords.size);
});

it('label probabilities', function(){
  console.log(labelProbabilities);
});
```

This is a characterization test, so assert something implausible for both:

```
it('number of chords', function(){
  assert.equal(allChords.size, null);
  console.log(allChords.size);
});

it('label probabilities', function(){
  assert.equal(labelProbabilities, null);
  console.log(labelProbabilities);
});
```

Our assertion errors point the way toward the fixes:

```
  1) the file number of chords:
     AssertionError: 37 == null
      at Context.<anonymous> (bayes/naive-bayes.js:145:12)

  2) the file label probabilities:
     AssertionError: Map {
'easy' => 0.3333333333333333,
'medium' => 0.3333333333333333,
'hard' => 0.3333333333333333 } == null
      at Context.<anonymous> (bayes/naive-bayes.js:150:12)
```

And we have the info to fix our tests and get rid of the logging statements:

```
it('number of chords', function(){
  assert.equal(allChords.size, 37);
});

it('label probabilities', function(){
  assert.equal(labelProbabilities.get('easy'), 0.3333333333333333);
  assert.equal(labelProbabilities.get('medium'), 0.3333333333333333);
  assert.equal(labelProbabilities.get('hard'), 0.3333333333333333);
});
```

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:

- set up data (songs and labels)
- set up objects, sets and maps

- train our classifier on the set of songs
- set counts and probabilities
- 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. First, we can move our count and probability setup to be right after our training code:

```
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();
```

We want to get away from *running the file*, and start to think about *executing functions*. These last three functions are always run together, so we can wrap them in another function, and run that immediately. So we can wrap them in a function and call it like this:

```
function setLabelsAndProbabilities(){
  setLabelProbabilities();
  setChordCountsInLabels();
  setProbabilityOfChordsInLabels();
};
setLabelsAndProbabilities();
```

Note that we could also run this as an anonymous IIFE (“immediately invoked function expression”), 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'll have to name it or find a way to run the code containing it each time.

Anyways, 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`. We can then extract a new function for training and add this to the end:

```
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();
};
```

Now we need `trainAll` to be run once, after our difficulties, songs, and other variables are declared, but before we run `classsify`. Since we want `trainAll` to be part of our public interface (a *function* that we run, rather than *part of a file* that we run), we should move it to happen before our test as setup code.

```
describe('the file', function() {
  trainAll();
```

Next, we can extract a function around setting our songs and 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 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 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 ommitting the `var` keyword.

```
function setDifficulties(){
  easy = 'easy';
  medium = 'medium';
  hard = 'hard';
};
```

```
setDifficulties();
```

And we can do the same for our other global variables.

```
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();
...
```

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 not pretending they're not global by using the `var` keyword anymore.

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: we run `trainAll` and `classify` rather than the file.

Great. So now everything is inside of a function and our interface for the tests is well defined. We can 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:

```
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 group some lines inside:

```
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 it to group things where it makes sense. But these changes don't actually help our code be more clear.

The opposite of *extracting* functions is *inlining* them. When an extracted function (just like with variables as we saw earlier) doesn't really do anything, it's sensible to inline them. Let's inline these functions to put them back the way they were.

```
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 setting (extracting) and inlining (deleting) variables.

Another area where extracting functions can be useful is in 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, 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!

```
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 VS. FUNCTION LITERALS AND REFERENCES

For programmers who are new to JavaScript or those who mostly work in some other language on the backend, there is 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, not 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.” It is a function call.

The confusion comes when people don’t realize that an anonymous function declaration, when replaced with a named function for reuse, is as we did it with a function *reference*:

```
$('.my-button').on('click', visitSite);
```

And NOT with a function call like this:

```
$('.my-button').on('click', visitSite());
```

You are passing a reference to the `visit` 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 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 that takes 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 keep in mind 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 call it and make use of a parameter that was not defined by the signature. And any parameters that are unfulfilled by the function call will be set as `undefined`.

Scoping and Objects

Now we have everything wrapped in functions, but we have many global variables floating around. Let's solve that by creating an object called classifier to store those. Put this at 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(){
  setDifficulties();
  classifier.setup(); //this line is new
```

Now we need to move our `setup` function into our `classifier` object. Let's add this function to that object that was on top. 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. Everywhere reference to any of those variables: `songs`, `allChords`, `labelCounts`, `labelProbabilities`, `chordCountsInLabels`, and `probabilityOfChordsInLabels` has to be changed to add `this.` in front of them. Make those changes now.

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 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()
};
```

And we can also delete the call to it in trainAll:

```
function trainAll(){
  setDifficulties();
  classifier.setup(); //delete this line
  setSongs();
```

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 and everything looks good.

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 song list, we have to do it in two places. But worse than that, we are not prepared for a case where we might have a database of songs to train, rather than a handful that we hardcode in the 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});
  }
}
```

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);
}
```

At this point, we would get errors. 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 save/test/commit should reveal that all is well, but we still have 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`.

Following that, it might seem a waste to complicate our `setSongs` function with these variables. What if we just used numbers as a shorthand, 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)
...
```

Save/Test/Commit

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 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 add the function body to the tests and remove it from the program:

```
describe('the file', function() {
  songList.addSong('imagine', ['c', 'cmaj7', 'f', 'am', 'dm', 'g', 'e7'], 0)
...
  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);
  trainAll();
```

Now we're finally 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.

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 code bases or 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 10 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 the sad news. It does prevent a variable from being reassigned (ie. using the = operator), but the contents of the variable can still change. Array indexes can be updated. Same goes for object attributes and members of sets and maps.

To *ensure* immutability, you'll need to use something like immutable.js, which provides 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.

```
TypeError: Assignment to constant variable.
```

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;
```

But this variable doesn't have to be reassigned. The variable `first` initially is meant to reflect the likelihood of a difficulty appearing as is proportional 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 we multiply 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 haven't used the `reduce` function before. 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, our `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 onto the next element.

If that's confusing, think of it this way: Say you had a `reduce` function that summed values, but only returned the total, starting with an initial value of 10.

```
[2, 3, 4].reduce(function(total, element){ return total }, 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`.

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;
```

It's true that 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 priviledges the less likely possibilities at the top of a function. Moreover, it less strongly signals two code paths, which if only slightly, disguises the complexity of the code. For someone who is interested in refactoring and consequently, eliminating complexity, it can mask places that may be good candidates for changes.

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
}
```

We can pick on this function a bit more. First of all, we should have no issue with tying it more closely with the `classifier` object. If there's one obvious thing that a classifier does, it's classifying.

To that end, let's update our `classifier` to have the `classify` function as a property.

```
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 brake 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 this:

```
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 3rd person in a sense. Rather than using `this`, it says it's own name: `classifier`. There are three instances of this. 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 these statements are within. 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`. We will look at a better way (arrow functions) later in this chapter.

```
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);
```

Notice the third line from the bottom is where the `bind` function is called. It is what allows, on the fifth line from the top, for us to now use `this`, instead of `self`. Now our classify function looks like this:

```
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;
}
```

To simplify this function a bit, `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(function(_probabilities,
                                           difficulty){
    const totalLikelihood = chords.reduce(function(total, chord){
      const probabilityOfChordInLabel =
        this.probabilityOfChordsInLabels.get(difficulty)[chord]
      if(probabilityOfChordInLabel){
        return total * (probabilityOfChordInLabel + this.smoothing)
      }else{
        return total;
      }
    }.bind(this), this.labelProbabilities.get(difficulty) +
      this.smoothing)
    classified.set(difficulty, totalLikelihood);
  }, this);
  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. Unfotunately, to get 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, but `defineProperty` is a good tool to look into if you want a lot of control over how properties are used.

Normally, this 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, 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), use a loop of some kind to change that variable, and then return 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(function(labelWithProbability){
    const difficulty = labelWithProbability[0];
...
  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, so we set a new const: `difficulty`.

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(
    function(labelWithProbability){
      const difficulty = labelWithProbability[0];
      const totalLikelihood = chords.reduce(function(total, chord){
        const probabilityOfChordInLabel =
          this.probabilityOfChordsInLabels.get(difficulty)[chord]
        if(probabilityOfChordInLabel){
          return total * (probabilityOfChordInLabel + this.smoothing)
        }else{
          return total;
        }
    }.bind(this), this.labelProbabilities.get(difficulty)
      + this.smoothing)
    return [difficulty, totalLikelihood];
  }, this));
  return classified;
}
```

The second line now includes directly assigning a `new Map` object to `classified`. Additionally, our `return`, 5 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 parentheses in the line:

```
  }, this));
```

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(
      function(labelWithProbability){
...
  }, this));
}
```

We now have a similar unnecessary variable in `totalLikelihood`:

```
const totalLikelihood = chords.reduce(function(total, chord){
  const probabilityOfChordInLabel =
    this.probabilityOfChordsInLabels.get(difficulty)[chord]
  if(probabilityOfChordInLabel){
    return total * (probabilityOfChordInLabel + this.smoothing)
  }else{
    return total;
  }
}.bind(this), 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:

```
return [difficulty,
  chords.reduce(function(total, chord){
    const probabilityOfChordInLabel =
      this.probabilityOfChordsInLabels.get(difficulty)[chord]
    if(probabilityOfChordInLabel){
      return total * (probabilityOfChordInLabel + this.smoothing)
    }else{
      return total;
    }
  }.bind(this), this.labelProbabilities.get(difficulty) +
    this.smoothing)
]
```

This might look confusing, but you'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;
}
```

First, simply delete that first line, and replace the other two calls with the right hand 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: function(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: function(difficulty, chord){
  const value =
    this.probabilityOfChordsInLabels.get(difficulty)[chord];
  return value ? value + this.smoothing : 1;
},
```

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 more dense. We also have 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. 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 means more potential places that you can test your code.

For both variables and functions, the important part is to be aware of inlining and extracting as choices.

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.

Moving on from our `classify` function, we have a problem in the code that needs to be addressed. 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`.

The confusion happens in the `setProbabilityOfChordsInLabels` function: specifically, the second line.

```
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` on the right hand side of the assignment into the `Set` on the left hand side. Both `classifier.probabilityOfChordsInLabels` and `classifier.chordCountsInLabels` are just fingers pointing at the same object. If you change the values by referencing either name, both will be affected.

Here is a short example:

```
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 it still ponts at the original object
```

Back to our code, if `probabilityOfChordsInLabels` and `chordCountsInLabels` do the same thing, we can just get replace references of 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`:

```
const classifier = {
...
chordCountsInLabels: new Map(),
probabilityOfChordsInLabels: new Map(),
...
const value =
this.probabilityOfChordsInLabels.get(difficulty)[chord];
```

The last two lines of that snippet should be deleted. The last line should be updated to use `chordCountsInLabels` instead:

```
const classifier = {
...
chordCountsInLabels: new Map(),
...
const value = this.chordCountsInLabels.get(difficulty)[chord];
```

In some cases, the correct refactoring here would be truly *copy* the `Set` into a new one. Then, both sets would have appropriate names and values. But we'll

try a different approach here.

"COPYING" OBJECTS

If you're interested in copying objects in JavaScript, look into the terms "deep copy" vs. "shallow copy," as well as "cloning" and the `Object` functions for `freeze`/`assign`/`seal`. They might do what you want. Or maybe you want 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 a constructor function or a class? Maybe you just want a factory function (see Chapter 7). Maybe you want a new *and* immutable container for your data.

By the way, 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. We'll cover some popular and useful options in Chapter 7. Just 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, 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: function(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.

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.

Also, in this case, 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. In its place, we'll be using a simple function to check a given chord and difficulty combo. Update your classifier object so that it looks like this:

```
const classifier = {
...
  likelihoodFromChord: function(difficulty, chord){
    return this.chordCountsInLabels
      .get(difficulty)[chord] / this.songs.length;
  },
  valueForChordDifficulty: function(difficulty, chord){
    const value = this.likelihoodFromChord(difficulty, chord);
```

The `likelihoodFromChord` function and its call on the last line are both new. Make sure to delete the call to `setProbabilityOfChordsInLabels` from setLabelsAndProbabilities function as well. It should look like this:

```
function setLabelsAndProbabilities(){
  setLabelProbabilities();
  setChordCountsInLabels();
};
```

With these changes, we've simplified the code and stopped reassignments to `likelihoodFromChord`.

ABOUT REASSIGNING VARIABLES

Among the most important developments recently happening in JavaScript and beyond is an increased importance in functional programming. We'll explore the benefits more in a later chapter, 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 very long functions that assign and reassign to variables throughout.

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 behind a function and assigning to a new variable when possible. Overall, making the scope of *most* variables small should be a priority.

While we're on the topic of objects that seem too similar, `classifier.songs` and `songList.songs` seem to have very almost identical data. Really, the only difference is the extra property (`name`) in the `songList.songs`. Let's get rid of `classifier`'s `song` property.

First, we need to make two changes to classifier. After deleting the `songs` attribute and changing `this` to `songList` in the `likelihoodFromChord` function, we should have the following:

```
const classifier = {
...
likelihoodFromChord: function(difficulty, chord){
  return this.chordCountsInLabels
    .get(difficulty)[chord] / songList.songs.length;
},
```

Next, we have to delete the second line of our `train` function:

```
function train(chords, label){
  classifier.songs.push({label: label, chords: chords});
```

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

Next, let's see if we can get `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 changing the call in `setLabelsAndProbabilities` to reference it through the `classifier`:

```
function setLabelsAndProbabilities(){
  setLabelProbabilities();
  classifier.setChordCountsInLabels();
};
```

Next, we "thisify" 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 retreiving the values from the `probabilityOfChordsInLabels` map, we can do the same thing to eliminate the `chordCountsInLabels` map.

Taking a look again at the `setChordCountsInLabels` function, it loops through each chord in each song, and adds 1 for every instance of the chord:

```
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);
},
```

If all we have to do is determine the count that a chord appears in a given difficulty, we can set up similar loops, a counter, and just add 1 when there is a match. Add this as just following `setChordCountsInLabels` in the `classify` object:

```
chordCountForDifficulty: function(difficulty, test_chord){
  let counter = 0;
  songList.songs.forEach(function(song){
    if(song.difficulty === difficulty){
      song.chords.forEach(function(chord){
        if(chord===test_chord){
          counter = counter + 1;
        }
      }, this);
    }
```

```
    }, this);
    return counter;
  },
```

Now we can use that function instead of setting and retreiving the chord counts. This means a few 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` :

```
  function setLabelsAndProbabilities(){
    setLabelProbabilities();
    classifier.setChordCountsInLabels();
  };
```

After getting rid of the third line in that function, we're left with:

```
  function setLabelsAndProbabilities(){
    setLabelProbabilities();
  };
```

This function now only exists to call another function. That means we can delete `setLabelsAndProbabilities` and change its call to 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 to another function, 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;
          }
        }, this);
      }
    }, this);
    return counter;
  },
```

As we did in the `classify` function, when you 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.

```
  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
          }
        }, this);
      }
      return counter;
    }.bind(this), 0);
  },
```

So here, we have a few changes.

- We return the result of reduce directly.

- We replace our former `counter` variable with a parameter to `reduce`'s callback function.
- Our thisArg changes to be a `.bind(this)` because `reduce` doesn't accept a thisArg like `forEach` does, and we want to keep our `this` in the inner scope.
- 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 matching a 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
      }, this).length
    }
    return counter;
  }.bind(this), 0);
},
```

The `filter` function returns a new array consisting of elements that matched the conditional. We get the `length` of this array and add it to the `counter`. This means fewer updates to the `counter` and we also shaved two lines off of the length of the `chordCountForDifficulty` 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 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 is only for refactoring purposes of 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 often will probably see some performance benefit.

We'll discuss "memoization" along with functional programming. It, as well as other caching techniques might overwhelm any performance hit you'll take from the kind of refactoring we did here.

The approach of this book is to write for humans first, and worry about performance after the fact.

Ignoring the welcome message code (mostly included to explore string-based refactorings), which you can delete along with its test now if you want (we won't be discussing it further), 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 needs a new `classifier.` also.

```
classifier.trainAll();
```

Now, let's "thisify" our functions. That means changing instances of `classifier` to `this` as well as adding the "thisArgs" as second arguments to the callbacks of the `forEach` functions:

```
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)
}
```

Now, everything is part of either `classifier` or `songList`. We're down to two gloabal 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 two references to it (function calls) so that they are prepended with `songList`.

One is inside of `train`:

```
songList.allChords.add(chord);
```

The other is in the tests:

```
it('number of chords', function(){
  assert.equal(songList.allChords.size, 37);
});
```

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, commit.

Next up, let's address some inconsistencies that we have in our anonymous function syntax. Previously, we used an arrow function in our train function:

```
trainAll: function(){
  chords.forEach(chord => {
```

```
      this.songList.allChords.add(chord);
    });
  ...
```

You might notice something interesting about this, namely `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 back to its old form:

```
trainAll: function(){
  chords.forEach(function(chord){
    this.songList.allChords.add(chord);
  });
...
```

And it will be broken! If you want to use that form, you need to include the `this`.

```
trainAll: function(){
  chords.forEach(function(chord){
    this.songList.allChords.add(chord);
  }, this);
...
```

Or use `bind` on the function as we did with reduce:

```
trainAll: function(){
  chords.forEach(function(chord){
    this.songList.allChords.add(chord);
  }.bind(this));
...
```

The arrow syntax 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), they are very similar to function declaration syntax, so they are easy to extract:

```
chords.forEach(myFunction, this);

function myFunction(){
  this.songList.allChords.add(chord);
}
```

But for all of our functions used with `forEach`, `map`, and `reduce`, arrow functions are the way to go. With them, we shave off about ten lines of code. By the way, with our refactoring so far, what was once about 110 lines is now down to 60. Anyways, it's small enough to just paste the whole thing (minus the tests).

```
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]})
    }
  },
  labelCounts: new Map(),
  labelProbabilities: new Map(),
  smoothing: 1.01,
  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);
  },

  likelihoodFromChord: function(difficulty, chord){
    return this.chordCountForDifficulty(difficulty, chord) /
this.songList.songs.length;
  },
```

```
  valueForChordDifficulty: function(difficulty, chord){
    const value = this.likelihoodFromChord(difficulty, chord);
    return value ? value + this.smoothing : 1;
  },
  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)]
    }));
  },

  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);
    })
  }
};
```

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:

```
setLabelProbabilities: function(){
  this.labelCounts.forEach((_count, label) =>{
```

One somewhat strange thing, is that we also need parens if we have zero arguments. At this point, we can change all of our tests to use this syntax:

```
describe('the file', () => {
...
  it('works', () => {
...
  it('classifies', () => {
...
  it('classifies again', () => {
...
  it('number of chords', () => {
...
  it('label probabilities', () => {
...
```

We could also declare our `trainAll` function like this:

```
trainAll: () => {
  this.songList.songs.forEach(song =>{ this.train(song.chords, song.difficulty)});
  this.setLabelProbabilities();
},
```

Unfortunately, this is a bad move, because 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 in our case, assuming non-strict mode). In any case, there is a better shorthand for function declarations, which we'll get to shortly.

One other weird thing about the syntax is that sometimes we use braces `{}`, and other times, we don't. We can see this in the code:

```
song.chords.filter(chord => chord === testChord)
```

The lack of braces might seem a little jarring because it makes the statement more dense. There is an additional consideration with the no brace version though: The no brace version implitly returns the result of its execution. In the case of our last code snippet, that means it returns `true` or `false`.

Also to note is 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
```

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 functions:

```
const classifier = {
  songList: {
...
    addSong: function(name, chords, difficulty){
...
  chordCountForDifficulty: function(difficulty, testChord){
...
```

But we 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!

ABOUT "COMPUTED PROPERTIES"

While we're discussing function declaration shorthand, it's worth noting that you could do it dynamically, like this:

```
songs = {
```

```
    ['first' + 'Song']: {},
    ['second' + 'Song']: {},
    ['third' + 'Song']: {}
  }
```

Basically, you can run JavaScript inside those 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 code base.

But this shorthand only works inside of object literals and classes. Outside of those contexts (in a normal function or constructor function scope for example), the interpreter will throw an error on the `{`.

Speaking of classes, right now, we're just dealing with object literals. But it's time for us to explore other options.

First up is instance objects. 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: name,
                       chords: chords,
                       difficulty: this.difficulties[difficulty]})
    }
  };
  this.labelCounts = new Map();
  this.labelProbabilities = 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.songList = new SongList();

  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.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)]
    }));
  };

  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);
    })
  }
};
const assert = require('assert');
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();
```

DON'T FORGET THE `NEW` IN CONSTRUCTOR FUNCTIONS

If you forget the `new` when calling a constructor function, the 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 considered as the main motivation behind prefering using the `Object.create` to `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` promotes.

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. Chapter 7 has more details on your options when dealing with multiple objects.

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," which is what the capitalized functions are called. 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," aka "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 funtion and not constrained by the JSON syntax, we're free to write any statements we want. That means we could have private 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). In the following snippet, notice both of these aspects (private functions/variables as well as globals) in action:

```
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";
```

```
}
let 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 program, let's just observe the differences between this construct vs. `new` as in the last snippet.

```
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;
})();
let 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 "Immediately Invoked Function Expression," aka. an "IFFE." We could happily use a normal function expression as well:

```
var secretTemplate = function(){
...
}
let s = Object.create(secretTemplate());
```

This gives us the same object, but seem a bit less clear. Also, in this construction, the function will have to run every time we create a new object. The IFFE only has to run once, and creating new objects will just reference the template object that was created.

Reordering our code a bit, we could express our object returning code more consisely like this:

```
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;
          }}
})();
let 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 below:

```
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: normalInfo,
          notSecret: notSecret}
})();
let 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.

Getting back to our classifier, how would we use the `Object.create` pattern? 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`.

```
classifierTemplate = {
  songList: {
    allChords: new Set(),
...
const assert = require('assert');
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, and not a function or IFFE because we don't have anything that we care about being private at the moment.

In case you forgot what advantage `Object.create` and `new` have over just using the object literal directly, the point 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 will be created for each `classifier`, even though it uses the object literal syntax. As we will explore a bit more in the chapter about "Design Patterns," `Object.create` and `new` (along with classes and modules) are also your gateways to inheritance.

Next, we'll convert our code into a class. Note that neither of our objects is a good fit for `Map` because they contain various of 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: function(name, chords, difficulty){
        this.songs.push({name: name,
                         chords: 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, but it's a special kind of function. And like 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 with when we used a constructor function) having semicolons instead of commas. Otherwise our function definitions are the same. Properties that aren't functions, however, must 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;
```

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 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 (appparent 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! It's *true nature* is disguised by them!"

Or "JavaScript was only built in ten 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."

Now that our class is looking to be in pretty good shape, it's time to consider our API itself. In other words, if someone was 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 (private either means obscured or inaccessible), but nonetheless, we can determine the ideal now, even before realizing the distinction fully in a module.

There are three functions from `classify` that we need to be public:

- `constructor`
- `trainAll`
- `classify`

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 10). 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` was tied to `train`, instead of `trainAll`, our tests would break.

This is a problem we will not fix in this chapter, but we will be looking at idempotency as part of the functional programming chapter.

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)
  };
...
```

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 the “spread operator.” If you come up with a good mnemonic for remembering which is which, please feel free to email the author.

Rest 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 delegatation functions and letting the tests (or “client code” that uses our module) is worth considering. Having just one object that the client interacts with is nice for them, but if our delgation is adding sufficient bulk, we should rethink the design.

Anyways, as we saw in the last chapter, 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. These are the labels you should change to having an underscore in front of them:

- `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, make a new file called bayes.js including all of the non-testing code:

```
module.exports = class Classifier  {
  constructor(){
    this._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]})
      }
    };
    this._labelCounts = new Map();
    this._labelProbabilities = new Map();
    this._smoothing = 1.01;
  };
  addSong(...songParams){ //rest
...
  _setLabelProbabilities(){
    this._labelCounts.forEach((_count, label) =>{
      this._labelProbabilities.set(label, this._labelCounts.get(label) /
this._songList.songs.length);
    })
  }
};
```

Notice that there is only one change that we need to make. It's right up top: `module.exports =`

Then create a file in the same directory called bayes_music_test.js, and add your testing code:

```
Classifier = require('./bayes_music.js');
const assert = require('assert');
describe('the file', () => {
  const classifier = new Classifier;
  classifier.addSong('imagine', ['c', 'cmaj7', 'f', 'am', 'dm', 'g', 'e7'], 0)
...
  it('label probabilities', () => {
    assert.equal(classifier._labelProbabilities.get('easy'), 0.3333333333333333);
    assert.equal(classifier._labelProbabilities.get('medium'), 0.3333333333333333);
    assert.equal(classifier._labelProbabilities.get('hard'), 0.3333333333333333);
  });
})
```

Again, we only have one change, and it's right up top. If you run `mocha bayes_music_test.js`, you should have no failures. Awesome.

But do you want actual privacy? If so, 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 "Symbols" (creating privacy through obscurity) or WeakMaps, or using an initialization function (which is basically designing your own form of the revealing module pattern).

If we honestly think about the tradeoffs here though, what are you gaining by doing something complex and non-standard? You're making more work for yourself and others that might work on your module. The code itself and the tests become more complex. Losing your ability to call tests for *truly* private, unaddressable functions seems like a non-starter. On the pro-side, you might discover a unique way of going about things that will become adopted as the standard, and then you're covered, but that's pretty unlikely.

What do you give up by going with the _ route? A few extra characters here and there 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 unitiated, 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.

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 coders and managers, 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 vs. the code now.

Personally, I recommend building 2 or 3 kinds of something before insisting that they’re similar and attempting to abstract.

But now that our code is in better shape, and our program is fairly small, it’s easy enough to attempt for this one.

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: function(name, words, comprehension){
        this.texts.push({name: name,
                         words: words,
                         comprehension: this.understood[comprehension]})
      }
    };
    this._labelCounts = new Map();
    this._labelProbabilities = new Map();
    this._smoothing = 1.01;
  };
  addText(...textParams){ //rest
    this._textList.addText(...textParams); //spread
  };
  _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;
  };
  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)]
    }));
  };

  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);
    })
  }
};
```

And the tests:

```
Classifier = require('./bayes.js');
const assert = require('assert');
describe('the file', () => {
  const classifier = new Classifier;
  classifier.addText('japanese text',
                     ['あ',     'い',    'う',     'え',     'お',
                       'か',     'き',    'く',    'け',    'こ'],
                       0)
  classifier.addText('english text',
                     ['a', 'b', 'c', 'd', 'e', 'f', 'g', 'h', 'i',
                      'j', 'k', 'l', 'm', 'n', 'o', 'p', 'q'],
                      1)

  classifier.trainAll();
  it('works', () => {
    assert(true);
  })
  it('classifies', () =>{
    const classified = classifier.classify(['f#m7', 'a', 'dadd9', 'dmaj7', 'bm',
'bm7', 'd', 'f#m']);
  classifier.addText(['お', 'は', 'よ', 'う', 'ご', 'ざ', 'い', 'ま', 'す'])
    assert.equal(classified.get('yes'), 1.51 );
    assert.equal(classified.get('no'), 3.442951);
  });
  it('number of words', ()=>{
    assert.equal(classifier._textList.allWords.size, 27);
  });

  it('label probabilities', ()=>{
    assert.equal(classifier._labelProbabilities.get('yes'), 0.5);
    assert.equal(classifier._labelProbabilities.get('no'), 0.5);
  });
})
```

Based on the earlier input, the tests show that the Japanese is likely to be incomprehensible. If we want 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).

Those are all features to add, and we’re all about refactoring here. We’re paying off technical debt and improving quality.

Conclusion

In this chapter, 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.