

JavaScript

Allongé

A close-up photograph of a white ceramic cup filled with a dark liquid, likely coffee or tea, resting on a black saucer. A silver spoon lies next to the cup on the saucer. The background is a light-colored, textured surface.

the **six** edition

JavaScript Allongé, the "Six" Edition

Programming from Functions to Classes in ECMAScript
2015

Reg “raganwald” Braithwaite

This book is for sale at <http://leanpub.com/javascriptallongesix>

This version was published on 2016-02-01



This is a [Leanpub](#) book. Leanpub empowers authors and publishers with the Lean Publishing process. [Lean Publishing](#) is the act of publishing an in-progress ebook using lightweight tools and many iterations to get reader feedback, pivot until you have the right book and build traction once you do.

© 2015 - 2016 Reg “raganwald” Braithwaite

Also By Reg “raganwald” Braithwaite

Kestrels, Quirky Birds, and Hopeless Egocentricity

What I’ve Learned From Failure

How to Do What You Love & Earn What You’re Worth as a Programmer

Steal Raganwald’s Book!

CoffeeScript Ristretto

JavaScript Spessore

Contents

A Pull of the Lever: Prefaces	i
About JavaScript Allongé	ii
What JavaScript Allongé is. And isn't.	v
Foreword to the “Six” edition	viii
Forewords to the First Edition	ix
Prelude: Values and Expressions over Coffee	xi
values are expressions	xii
values and identity	xiv
A Rich Aroma: Basic Numbers	1
The first sip: Basic Functions	5
As Little As Possible About Functions, But No Less	7
Ah. I'd Like to Have an Argument, Please.	16
Closures and Scope	21
That Constant Coffee Craving	26
Naming Functions	39
Combinators and Function Decorators	45
Building Blocks	48
Magic Names	51
Summary	55
Recipes with Basic Functions	56
Partial Application	57
Unary	59
Tap	61
Maybe	63
Once	65
Left-Variadic Functions	66
Compose and Pipeline	71
Picking the Bean: Choice and Truthiness	75
Composing and Decomposing Data	81

CONTENTS

Arrays and Destructuring Arguments	82
Self-Similarity	90
Tail Calls (and Default Arguments)	98
Garbage, Garbage Everywhere	107
Plain Old JavaScript Objects	113
Mutation	122
Reassignment	129
Copy on Write	139
Tortoises, Hares, and Teleporting Turtles	145
Functional Iterators	148
Making Data Out Of Functions	158
Recipes with Data	172
mapWith	174
Flip	176
Object.assign	179
Why?	182
A Warm Cup: Basic Strings and Quasi-Literals	183
Stir the Allongé: Objects and State	186
Encapsulating State with Closures	187
Composition and Extension	194
This and That	200
What Context Applies When We Call a Function?	205
Method Decorators	211
Summary	214
Recipes with Objects, Mutations, and State	215
Memoize	216
getWith	219
pluckWith	221
Deep Mapping	223
The Coffee Factory: “Object-Oriented Programming”	225
Served by the Pot: Collections	228
Iteration and Iterables	229
Generating Iterables	247
Lazy and Eager Collections	268
Interlude: The Carpenter Interviews for a Job	283
More Iterable Mapping Operations	295
Interactive Generators	299
Basic Operations on Iterables	310

CONTENTS

A Coffeehouse: Symbols	313
Life on the Plantation: Metaobjects	317
Why Metaobjects?	318
Mixins, Forwarding, and Delegation	319
Later Binding	325
Delegation via Prototypes	331
Shared Prototypes	336
Decaffeinated: Impostors	341
Finish the Cup: Constructors and Classes	345
Constructors and <code>new</code>	346
Why Classes in JavaScript?	352
Classes with <code>class</code>	359
Object Methods	365
Why Not Classes?	368
Summary	371
Recipes with Constructors and Classes	372
Bound	375
Send	378
Invoke	379
Fluent	381
Colourful Mugs: Symmetry, Colour, and Charm	384
Con Panna: Composing Class Behaviour	398
Extending Classes with Mixins	399
Functional Mixins	403
Emulating Multiple Inheritance	411
Preventing Property Conflicts	418
Reducing Coupling	422
More Decorators	426
Stateful Method Decorators	427
Class Decorators beyond ES6/ECMAScript 2015	432
Method Decorators beyond ES6/ECMAScript 2015	440
Lightweight Traits	444
More Decorator Recipes	456
After Method Advice	457
Before Method Advice	462
Provided and Unless	466
Method Advice	469

CONTENTS

Closing Time at the Coffeeshop: Final Remarks	477
The Golden Crema: Appendices and Afterwords	482
How to run the examples	483
Thanks!	485
Reading JavaScript Allongé on Kindle	487
Copyright Notice	488
About The Author	492

A Pull of the Lever: Prefaces



Caffe Molinari

“Café Allongé, also called Espresso Lungo, is a drink midway between an Espresso and Americano in strength. There are two different ways to make it. The first, and the one I prefer, is to add a small amount of hot water to a double or quadruple Espresso Ristretto. Like adding a splash of water to whiskey, the small dilution releases more of the complex flavours in the mouth.

“The second way is to pull an extra long double shot of Espresso. This achieves approximately the same ratio of oils to water as the dilution method, but also releases a different mix of flavours due to the longer extraction. Some complain that the long pull is more bitter and detracts from the best character of the coffee, others feel it releases even more complexity.

“The important thing is that neither method of preparation should use so much water as to result in a sickly, pale ghost of Espresso. Moderation in all things.”

About JavaScript Allongé

JavaScript Allongé is a first and foremost, a book about *programming with functions*. It's written in JavaScript, because JavaScript hits the perfect sweet spot of being both widely used, and of having proper first-class functions with lexical scope. If those terms seem unfamiliar, don't worry: JavaScript Allongé takes great delight in explaining what they mean and why they matter.

JavaScript Allongé begins at the beginning, with values and expressions, and builds from there to discuss types, identity, functions, closures, scopes, collections, iterators, and many more subjects up to working with classes and instances.

It also provides recipes for using functions to write software that is simpler, cleaner, and less complicated than alternative approaches that are object-centric or code-centric. JavaScript idioms like function combinators and decorators leverage JavaScript's power to make code easier to read, modify, debug and refactor.

JavaScript Allongé teaches you how to handle complex code, and it also teaches you how to simplify code without dumbing it down. As a result, *JavaScript Allongé* is a rich read releasing many of JavaScript's subtleties, much like the Café Allongé beloved by coffee enthusiasts everywhere.

why the “six” edition?

ECMAScript 2015 (formerly called ECMAScript 6 or “ES6”), is ushering in a very large number of improvements to the way programmers can write small, powerful components and combine them into larger, fully featured programs. Features like destructuring, block-structured variables, iterables, generators, and the class keyword are poised to make JavaScript programming more expressive.

Prior to ECMAScript 2015, JavaScript did not include many features that programmers have discovered are vital to writing great software. For example, JavaScript did not include block-structured variables. Over time, programmers discovered ways to roll their own versions of important features.

For example, block-structured languages allow us to write:

```
for (int i = 0; i < array.length; ++i) {  
    // ...  
}
```

And the variable `i` is scoped locally to the code within the braces. Prior to ECMAScript 2015, JavaScript did not support block-structuring, so programmers borrowed a trick from the Scheme programming language, and would write:

```
var i;

for (i = 0; i < array.length; ++i) {
  (function (i) {
    // ...
  })(i)
}
```

To create the same scoping with an Immediately Invoked Function Expression, or “IIFE.”

Likewise, many programming languages permit functions to have a variable number of arguments, and to collect the arguments into a single variable as an array. In Ruby, we can write:

```
def foo (first, *rest)
  # ...
end
```

Prior to ECMAScript 2015, JavaScript did not support collecting a variable number of arguments into a parameter, so programmers would take advantage of an awkward work-around and write things like:

```
function foo () {
  var first = arguments[0],
  rest   = [] .slice.call(arguments, 1);

  // ...
}
```

The first edition of *JavaScript Allongé* explained these and many other patterns for writing flexible and composable programs in JavaScript, but the intention wasn’t to explain how to work around JavaScript’s missing features: The intention was to explain why the style of programming exemplified by the missing features is important.

Working around the missing features was a necessary evil.

But now, JavaScript is gaining many important features, in part because the governing body behind JavaScript has observed that programmers are constantly working around the same set of limitations. With ECMAScript 2015, we can write:

```
for (let i = 0; i < array.length; ++i) {
  // ...
}
```

And `i` is scoped to the for loop. We can also write:

```
function foo (first, ...rest) {  
  // ...  
}
```

And presto, rest collects the rest of the arguments without a lot of malarky involving slicing arguments. Not having to work around these kinds of missing features makes JavaScript Allongé a *better book*, because it can focus on the *why* to do something and *when* to do it, instead of on the how to make it work.

JavaScript Allongé, The “Six” Edition packs all the goodness of JavaScript Allongé into a new, updated package that is relevant for programmers working with (or planning to work with) the latest version of JavaScript.

that's nice. is that the only reason?

Actually, no.

If it were just a matter of updating the syntax, the original version of JavaScript Allongé could have simply iterated, slowly replacing old syntax with new. It would have continued to say much the same things, only with new syntax.

But there's more to it than that. The original JavaScript Allongé was not just written to teach JavaScript: It was written to describe certain ideas in programming: Working with small, independent entities that compose together to make bigger programs. Thus, the focus on things like writing decorators.

As noted above, JavaScript was chosen as the language for Allongé because it hit a sweet spot of having a large audience of programmers and having certain language features that happen to work well with this style of programming.

ECMAScript 2015 does more than simply update the language with some simpler syntax for a few things and help us avoid warts. It makes a number of interesting programming techniques easy to explain and easy to use. And these techniques dovetail nicely with Allongé's focus on composing entities and working with functions.

Thus, the “six” edition introduces [classes](#) and [mixins](#). It introduces the notion of implementing private properties with [symbols](#). It introduces [iterators](#) and [generators](#). But the common thread that runs through all these things is that since they are all simple objects and simple functions, we can use the same set of “programming with functions” techniques to build programs by composing small, flexible, and decoupled entities.

We just call some of those functions [constructors](#), others [decorators](#), others [functional mixins](#), and yet others, [policies](#).

Introducing so many new ideas did require a major rethink of the way the book was organized. And introducing these new ideas did add substantially to its bulk. But even so, in a way it is still explaining the exact same original idea that programs are built out of small, flexible functions composed together.

What JavaScript Allongé is. And isn't.



JavaScript Allongé is a book about thinking about programs

JavaScript Allongé is a book about programming with functions. From functions flow many ideas, from decorators to methods to delegation to mixins, and onwards in so many fruitful directions.

The focus in this book on the underlying ideas, what we might call the fundamentals, and how they combine to form new ideas. The intention is to improve the way we think about programs. That's a good thing.

But while JavaScript Allongé attempts to be provocative, it is not *prescriptive*. There is absolutely no suggestion that any of the techniques shown here are the only way to do something, the best way, or even an acceptable way to write programs that are intended to be used, read, and maintained by others.

Software development is a complex field. Choices in development are often driven by social considerations. People often say that software should be written for people to read. Doesn't that depend upon the people in question? Should code written by a small team of specialists use the same techniques and patterns as code maintained by a continuously changing cast of inexperienced interns?

Choices in software development are also often driven by requirements specific to the type of software being developed. For example, business software written in-house has a very different set of requirements than a library written to be publicly distributed as open-source.

Choices in software development must also consider the question of consistency. If a particular codebase is written with lots of helper functions that place the subject first, like this:

```
const mapIterableWith = (iterable, fn) =>
  ({
    [Symbol.iterator]: function* () {
      for (let element of iterable) {
        yield fn(element);
      }
    }
  });
});
```

Then it can be jarring to add new helpers written that place the verb first, like this:

```
const filterIterableWith = (fn, iterable) =>
  ({
    [Symbol.iterator]: function* () {
      for (let element of iterable) {
        if (!!fn(element)) yield element;
      }
    }
  });
});
```

There are reasons why the second form is more flexible, especially when used in combination with partial application, but does that outweigh the benefit of having an entire codebase do everything consistently the first way or the second way?

Finally, choices in software development cannot ignore the tooling that is used to create and maintain software. The use of source-code control systems with integrated diffing rewards making certain types of focused changes. The use of [linters¹](#) makes checking for certain types of undesirable code very cheap. Debuggers encourage the use of functions with explicit or implicit names. Continuous integration encourages the creation of software in tandem with and factored to facilitate the creation of automated test suites.

JavaScript Allongé does not attempt to address the question of JavaScript best practices in the wider context of software development, because JavaScript Allongé isn't a book about practicing, it's a book about thinking.

how this book is organized

JavaScript Allongé introduces new aspects of programming with functions in each chapter, explaining exactly how JavaScript works. Code examples within each chapter are small and emphasize exposition rather than serving as patterns for everyday use.

¹https://en.wikipedia.org/wiki/Lint_

Following some of the chapters are a series of recipes designed to show the application of the chapter's ideas in practical form. While the content of each chapter builds naturally on what was discussed in the previous chapter, the recipes may draw upon any aspect of the JavaScript programming language.

Foreword to the “Six” edition

ECMAScript 6 (short name: ES6; official name: ECMAScript 2015) was ratified as a standard on June 17. Getting there took a while – in a way, the origins of ES6 date back to the year 2000: After ECMAScript 3 was finished, TC39 (the committee evolving JavaScript) started to work on ECMAScript 4. That version was planned to have numerous new features (interfaces, namespaces, packages, multimethods, etc.), which would have turned JavaScript into a completely new language. After internal conflict, a settlement was reached in July 2008 and a new plan was made – to abandon ECMAScript 4 and to replace it with two upgrades:

- A smaller upgrade would bring a few minor enhancements to ECMAScript 3. This upgrade became ECMAScript 5.
- A larger upgrade would substantially improve JavaScript, but without being as radical as ECMAScript 4. This upgrade became ECMAScript 6 (some features that were initially discussed will show up later, in upcoming ECMAScript versions).

ECMAScript 6 has three major groups of features:

- Better syntax for features that already exist (e.g. via libraries). For example: classes and modules.
- New functionality in the standard library. For example:
 - New methods for strings and arrays
 - Promises (for asynchronous programming)
 - Maps and sets
- Completely new features. For example: Generators, proxies and WeakMaps.

With ECMAScript 6, JavaScript has become much larger as a language. *JavaScript Allongé, the “Six” Edition* is both a comprehensive tour of its features and a rich collection of techniques for making better use of them. You will learn much about functional programming and object-oriented programming. And you’ll do so via ES6 code, handed to you in small, easily digestible pieces.

– Axel Rauschmayer Blogger², trainer³ and author of “Exploring ES6⁴”

²<http://www.2ality.com>

³<http://ecmanauten.de>

⁴<http://exploringjs.com>

Forewords to the First Edition

michael fokus

As a life-long bibliophile and long-time follower of Reg's online work, I was excited when he started writing books. However, I'm very conservative about books – let's just say that if there was an aftershave scented to the essence of "Used Book Store" then I would be first in line to buy. So as you might imagine I was "skeptical" about the decision to release *JavaScript Allongé* as an ongoing ebook, with a pay-what-you-want model. However, Reg sent me a copy of his book and I was humbled. Not only was this a great book, but it was also a great way to write and distribute books. Having written books myself, I know the pain of soliciting and receiving feedback.

The act of writing is an iterative process with (very often) tight revision loops. However, the process of soliciting feedback, gathering responses, sending out copies, waiting for people to actually read it (if they ever do), receiving feedback and then ultimately making sense out of how to use it takes weeks and sometimes months. On more than one occasion I've found myself attempting to reify feedback with content that either no longer existed or was changed beyond recognition. However, with the Leanpub model the read-feedback-change process is extremely efficient, leaving in its wake a quality book that continues to get better as others likewise read and comment into infinitude.

In the case of *JavaScript Allongé*, you'll find the Leanpub model a shining example of effectiveness. Reg has crafted (and continues to craft) not only an interesting book from the perspective of a connoisseur, but also an entertaining exploration into some of the most interesting aspects of his art. No matter how much of an expert you think you are, *JavaScript Allongé* has something to teach you... about coffee. I kid.

As a staunch advocate of functional programming, much of what Reg has written rings true to me. While not exclusively a book about functional programming, *JavaScript Allongé* will provide a solid foundation for functional techniques. However, you'll not be beaten about the head and neck with dogma. Instead, every section is motivated by relevant dialog and fortified with compelling source examples. As an author of programming books I admire what Reg has managed to accomplish and I envy the fine reader who finds *JavaScript Allongé* via some darkened channel in the Internet sprawl and reads it for the first time.

Enjoy.

– Fokus, [fokus.me](http://www.fokus.me)⁵

matthew knox

A different kind of language requires a different kind of book.

JavaScript holds surprising depths—its scoping rules are neither strictly lexical nor strictly dynamic, and it supports procedural, object-oriented (in several flavors!), and functional programming. Many

⁵<http://www.fokus.me>

books try to hide most of those capabilities away, giving you recipes for writing JavaScript in a way that approximates class-centric programming in other languages. Not JavaScript Allongé. It starts with the fundamentals of values, functions, and objects, and then guides you through JavaScript from the inside with exploratory bits of code that illustrate scoping, combinators, context, state, prototypes, and constructors.

Like JavaScript itself, this book gives you a gentle start before showing you its full depth, and like a Café Allongé, it's over too soon. Enjoy!

—Matthew Knox, mattknox.com⁶

⁶<http://mattknox.com>

Prelude: Values and Expressions over Coffee

The following material is extremely basic, however like most stories, the best way to begin is to start at the very beginning.

Imagine we are visiting our favourite coffee shop. They will make for you just about any drink you desire, from a short, intense espresso ristretto through a dry cappuccino, up to those coffee-flavoured desert concoctions featuring various concentrated syrups and milks. (You tolerate the existence of sugary drinks because they provide a sufficient profit margin to the establishment to finance your hanging out there all day using their WiFi and ordering a \$3 drink every few hours.)

You express your order at one end of their counter, the folks behind the counter perform their magic, and deliver the coffee you value at the other end. This is exactly how the JavaScript environment works for the purpose of this book. We are going to dispense with web servers, browsers and other complexities and deal with this simple model: You give the computer an [expression⁷](#), and it returns a [value⁸](#), just as you express your wishes to a barista and receive a coffee in return.

⁷https://en.wikipedia.org/wiki/Expression_

⁸https://en.wikipedia.org/wiki/Value_

values are expressions

All values are expressions. Say you hand the barista a café Cubano. Yup, you hand over a cup with some coffee infused through partially caramelized sugar. You say, “I want one of these.” The barista is no fool, she gives it straight back to you, and you get exactly what you want. Thus, a café Cubano is an expression (you can use it to place an order) and a value (you get it back from the barista).

Let’s try this with something the computer understands easily:

42

Is this an expression? A value? Neither? Or both?

The answer is, this is both an expression *and* a value.⁹ The way you can tell that it’s both is very easy: When you type it into JavaScript, you get the same thing back, just like our café Cubano:

42

//=> 42

All values are expressions. That’s easy! Are there any other kinds of expressions? Sure! let’s go back to the coffee shop. Instead of handing over the finished coffee, we can hand over the ingredients. Let’s hand over some ground coffee plus some boiling water.

Astute readers will realize we’re omitting something. Congratulations! Take a sip of espresso. We’ll get to that in a moment.

Now the barista gives us back an espresso. And if we hand over the espresso, we get the espresso right back. So, boiling water plus ground coffee is an expression, but it isn’t a value.¹⁰ Boiling water is a value. Ground coffee is a value. Espresso is a value. Boiling water plus ground coffee is an expression.

Let’s try this as well with something else the computer understands easily:

```
"JavaScript" + " " + "Allonge"  
//=> "JavaScript Allonge"
```

⁹Technically, it’s a *representation* of a value using Base10 notation, but we needn’t worry about that in this book. You and I both understand that this means “42,” and so does the computer.

¹⁰In some languages, expressions are a kind of value unto themselves and can be manipulated. The grandfather of such languages is Lisp. JavaScript is not such a language, expressions in and of themselves are not values.

Now we see that “strings” are values, and you can make an expression out of strings and an operator `+`. Since strings are values, they are also expressions by themselves. But strings with operators are not values, they are expressions. Now we know what was missing with our “coffee grounds plus hot water” example. The coffee grounds were a value, the boiling hot water was a value, and the “plus” operator between them made the whole thing an expression that was not a value.

values and identity

In JavaScript, we test whether two values are identical with the `==` operator, and whether they are not identical with the `!=` operator:

```
2 === 2
//=> true

'hello' !== 'goodbye'
//=> true
```

How does `==` work, exactly? Imagine that you’re shown a cup of coffee. And then you’re shown another cup of coffee. Are the two cups “identical?” In JavaScript, there are four possibilities:

First, sometimes, the cups are of different kinds. One is a demitasse, the other a mug. This corresponds to comparing two things in JavaScript that have different *types*. For example, the string “2” is not the same thing as the number 2. Strings and numbers are different types, so strings and numbers are never identical:

```
2 === '2'
//=> false

true !== 'true'
//=> true
```

Second, sometimes, the cups are of the same type—perhaps two espresso cups—but they have different contents. One holds a single, one a double. This corresponds to comparing two JavaScript values that have the same type but different “content.” For example, the number 5 is not the same thing as the number 2.

```
true === false
//=> false

2 !== 5
//=> true

'two' === 'five'
//=> false
```

What if the cups are of the same type *and* the contents are the same? Well, JavaScript’s third and fourth possibilities cover that.

value types

Third, some types of cups have no distinguishing marks on them. If they are the same kind of cup, and they hold the same contents, we have no way to tell the difference between them. This is the case with the strings, numbers, and booleans we have seen so far.

```
2 + 2 === 4
//=> true
```

```
(2 + 2 === 4) === (2 !== 5)
//=> true
```

Note well what is happening with these examples: Even when we obtain a string, number, or boolean as the result of evaluating an expression, it is identical to another value of the same type with the same “content.” Strings, numbers, and booleans are examples of what JavaScript calls “value” or “primitive” types. We’ll use both terms interchangeably.

We haven’t encountered the fourth possibility yet. Stretching the metaphor somewhat, some types of cups have a serial number on the bottom. So even if you have two cups of the same type, and their contents are the same, you can still distinguish between them.



Cafe Macchiato is also a fine drink, especially when following up on the fortunes of the Azzurri or the standings in the Giro d’Italia

reference types

So what kinds of values might be the same type and have the same contents, but not be considered identical to JavaScript? Let’s meet a data structure that is very common in contemporary programming languages, the *Array* (other languages sometimes call it a *List* or a *Vector*).

An array looks like this: [1, 2, 3]. This is an expression, and you can combine [] with other expressions. Go wild with things like:

```
[2-1, 2, 2+1]  
[1, 1+1, 1+1+1]
```

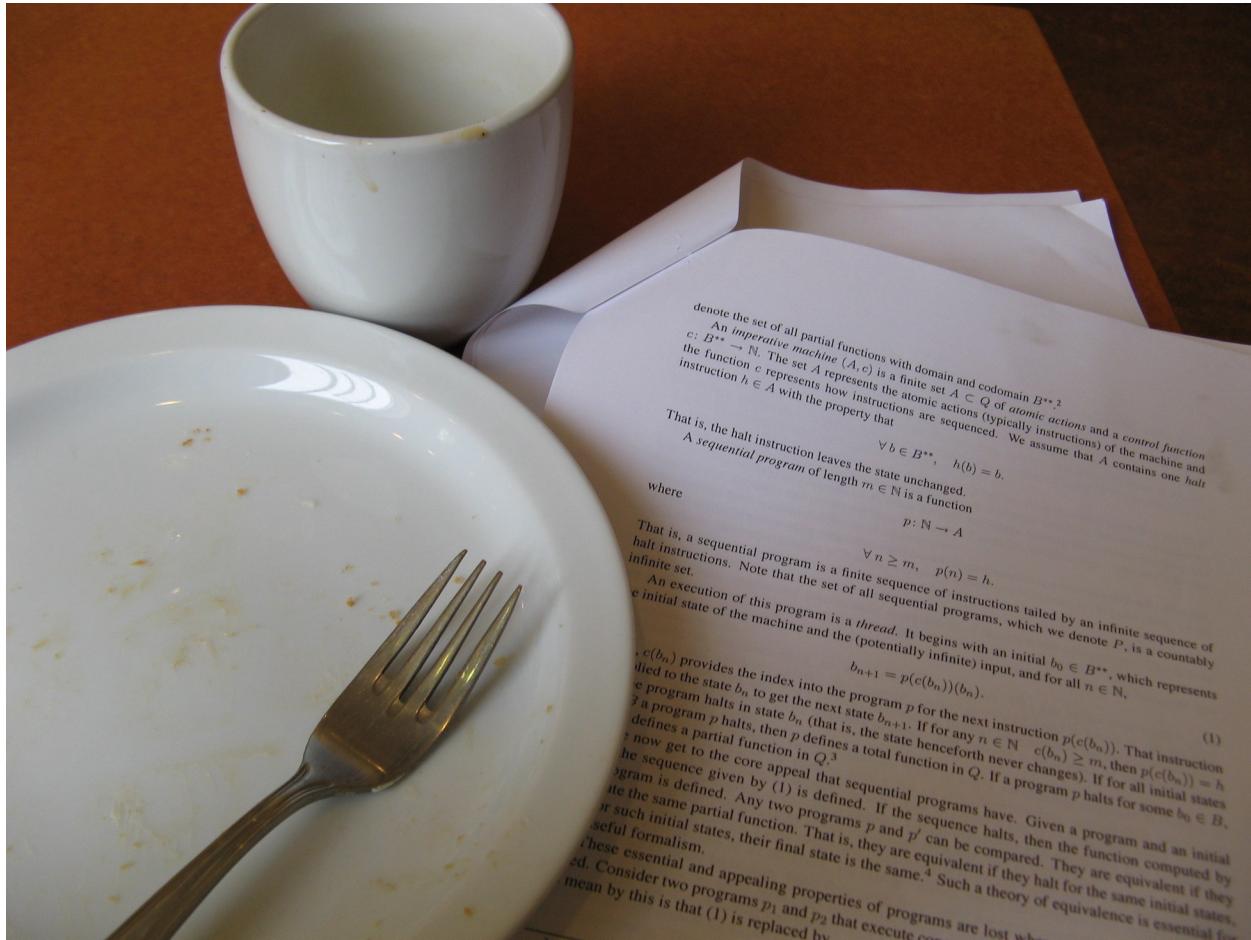
Notice that you are always generating arrays with the same contents. But are they identical the same way that every value of 42 is identical to every other value of 42? Try these for yourself:

```
[2-1, 2, 2+1] === [1,2,3]  
[1,2,3] === [1, 2, 3]  
[1, 2, 3] === [1, 2, 3]
```

How about that! When you type [1, 2, 3] or any of its variations, you are typing an expression that generates its own *unique* array that is not identical to any other array, even if that other array also looks like [1, 2, 3]. It's as if JavaScript is generating new cups of coffee with serial numbers on the bottom.

They look the same, but if you examine them with ===, you see that they are different. Every time you evaluate an expression (including typing something in) to create an array, you're creating a new, distinct value even if it *appears* to be the same as some other array value. As we'll see, this is true of many other kinds of values, including *functions*, the main subject of this book.

A Rich Aroma: Basic Numbers



Mathematics and Coffee

In computer science, a literal is a notation for representing a fixed value in source code. Almost all programming languages have notations for atomic values such as integers, floating-point numbers, and strings, and usually for booleans and characters; some also have notations for elements of enumerated types and compound values such as arrays, records, and objects. An anonymous function is a literal for the function type.—
[Wikipedia¹¹](#)

JavaScript, like most languages, has a collection of literals. We saw that an expression consisting solely of numbers, like 42, is a literal. It represents the number forty-two, which is 42 base 10. Not

¹¹[https://en.wikipedia.org/wiki/Literal_\(computer_programming\)](https://en.wikipedia.org/wiki/Literal_(computer_programming))

all numbers are base ten. If we start a literal with a zero, it is an octal literal. So the literal `042` is 42 base 8, which is actually 34 base 10.

Internally, both `042` and 34 have the same representation, as [double-precision floating point](#)¹² numbers. A computer's internal representation for numbers is important to understand. The machine's representation of a number almost never lines up perfectly with our understanding of how a number behaves, and thus there will be places where the computer's behaviour surprises us if we don't know a little about what it's doing "under the hood."

For example, the largest integer JavaScript can safely¹³ handle is 9007199254740991, or $2^{53} - 1$. Like most programming languages, JavaScript does not allow us to use commas to separate groups of digits.

floating

Most programmers never encounter the limit on the magnitude of an integer. But we mentioned that numbers are represented internally as floating point, meaning that they need not be just integers. We can, for example, write `1.5` or `33.33`, and JavaScript represents these literals as floating point numbers.

It's tempting to think we now have everything we need to do things like handle amounts of money, but as the late John Belushi would say, "Noooooooooooooo." A computer's internal representation for a floating point number is binary, while our literal number was in base ten. This makes no meaningful difference for integers, but it does for fractions, because some fractions base 10 do not have exact representations base 2.

One of the most oft-repeated examples is this:

```
1.0
//=> 1
1.0 + 1.0
//=> 2
1.0 + 1.0 + 1.0
//=> 3
```

However:

¹²http://en.wikipedia.org/wiki/Double-precision_floating-point_format

¹³Implementations of JavaScript are free to handle larger numbers. For example, if you type `9007199254740991 + 9007199254740991` into `node.js`, it will happily report that the answer is `18014398509481982`. But code that depends upon numbers larger than `9007199254740991` may not be reliable when moved to other implementations.

```

0.1
//=> 0.1
0.1 + 0.1
//=> 0.2
0.1 + 0.1 + 0.1
//=> 0.3000000000000004

```

This kind of “inexactitude” can be ignored when performing calculations that have an acceptable deviation. For example, when centering some text on a page, as long as the difference between what you might calculate longhand and JavaScript’s calculation is less than a pixel, there is no observable error.

But as a rule, if you need to work with real numbers, you should have more than a nodding acquaintance with the [IEEE Standard for Floating-Point Arithmetic](#)¹⁴. Professional programmers almost never use floating point numbers to represent monetary amounts. For example, “\$43.21” will nearly always be presented as two numbers: 43 for dollars and 21 for cents, not 43.21. In this book, we need not think about such details, but outside of this book, we must.

operations on numbers

As we’ve seen, JavaScript has many common arithmetic operators. We can create expressions that look very much like mathematical expressions, for example we can write `1 + 1` or `2 * 3` or `42 - 34` or even `6 / 2`. These can be combined to make more complex expressions, like `2 * 5 + 1`.

In JavaScript, operators have an order of precedence designed to mimic the way humans typically parse written arithmetic. So:

```

2 * 5 + 1
//=> 11
1 + 5 * 2
//=> 11

```

JavaScript treats the expressions as if we had written `(2 * 5) + 1` and `1 + (5 * 2)`, because the `*` operator has a *higher precedence* than the `+` operator. JavaScript has many more operators. In a sense, they behave like little functions. If we write `1 + 2`, this is conceptually similar to writing `plus(1, 2)` (assuming we have a function that adds two numbers bound to the name `plus`, of course).

In addition to the common `+`, `-`, `*`, and `/`, JavaScript also supports modulus, `%`, and unary negation, `-`:

¹⁴https://en.wikipedia.org/wiki/IEEE_floating_point

```
-(457 % 3)  
//=> -1
```

There are lots and lots more operators that can be used with numbers, including bitwise operators like `|` and `&` that allow you to operate directly on a number's binary representation, and a number of other operators that perform assignment or logical comparison that we will look at later.

The first sip: Basic Functions



The perfect Café Allongé begins with the right beans, properly roasted. JavaScript Allongé begins with functions.

As Little As Possible About Functions, But No Less

In JavaScript, functions are values, but they are also much more than simple numbers, strings, or even complex data structures like trees or maps. Functions represent computations to be performed. Like numbers, strings, and arrays, they have a representation. Let's start with the second simplest possible function.¹⁵ In JavaScript, it looks like this:

```
() => 0
```

This is a function that is applied to no values and returns `0`. Let's verify that our function is a value like all others:

```
((() => 0)  
 //=> [Function]
```

What!? Why didn't it type back `() => 0` for us? This *seems* to break our rule that if an expression is also a value, JavaScript will give the same value back to us. What's going on? The simplest and easiest answer is that although the JavaScript interpreter does indeed return that value, displaying it on the screen is a slightly different matter. `[Function]` is a choice made by the people who wrote Node.js, the JavaScript environment that hosts the JavaScript REPL. If you try the same thing in a browser, you may see something else.

¹⁵The simplest possible function is `() => {}`, we'll see that later.

I'd prefer something else, but I must accept that what gets typed back to us on the screen is arbitrary, and all that really counts is that it is somewhat useful for a human to read. But we must understand that whether we see [Function] or `() => 0`, internally JavaScript has a full and proper function.

functions and identities

You recall that we have two types of values with respect to identity: Value types and reference types. Value types share the same identity if they have the same contents. Reference types do not.

Which kind are functions? Let's try them out and see. For reasons of appeasing the JavaScript parser, we'll enclose our functions in parentheses:

```
((() => 0) === ((() => 0)
//=> false
```

Like arrays, every time you evaluate an expression to produce a function, you get a new function that is not identical to any other function, even if you use the same expression to generate it. "Function" is a reference type.

applying functions

Let's put functions to work. The way we use functions is to *apply* them to zero or more values called *arguments*. Just as $2 + 2$ produces a value (in this case 4), applying a function to zero or more arguments produces a value as well.

Here's how we apply a function to some values in JavaScript: Let's say that *fn_expr* is an expression that when evaluated, produces a function. Let's call the arguments *args*. Here's how to apply a function to some arguments:

```
fn_expr(args)
```

Right now, we only know about one such expression: `() => 0`, so let's use it. We'll put it in parentheses¹⁶ to keep the parser happy, like we did above: `((() => 0))`. Since we aren't giving it any arguments, we'll simply write `()` after the expression. So we write:

```
((() => 0)())
//=> 0
```

functions that return values and evaluate expressions

We've seen `() => 0`. We know that `((() => 0))` returns 0, and this is unsurprising. Likewise, the following all ought to be obvious:

¹⁶If you're used to other programming languages, you've probably internalized the idea that sometimes parentheses are used to group operations in an expression like math, and sometimes to apply a function to arguments. If not... Welcome to the ALGOL family of programming languages!

```
((() => 1)())
//=> 1
(() => "Hello, JavaScript")()
//=> "Hello, JavaScript"
(() => Infinity)()
//=> Infinity
```

Well, the last one's a doozy, but still, the general idea is this: We can make a function that returns a value by putting the value to the right of the arrow.

In the prelude, we looked at expressions. Values like `0` are expressions, as are things like `40 + 2`. Can we put an expression to the right of the arrow?

```
((() => 1 + 1)()
//=> 2
(() => "Hello, " + "JavaScript")()
//=> "Hello, JavaScript"
(() => Infinity * Infinity)()
//=> Infinity
```

Yes we can. We can put any expression to the right of the arrow. For example, `((() => 0)())` is an expression. Can we put it to the right of an arrow, like this: `() => ((() => 0)())`?

Let's try it:

```
((() => ((() => 0)())())
//=> 0
```

Yes we can! Functions can return the value of evaluating another function.

When dealing with expressions that have a lot of the same characters (like parentheses), you may find it helpful to format the code to make things stand out. So we can also write:

```
((() =>
  ((() => 0
    )())
  )()
//=> 0
```

It evaluates to the same thing, `0`.

commas

The comma operator in JavaScript is interesting. It takes two arguments, evaluates them both, and itself evaluates to the value of the right-hand argument. In other words:

```
(1, 2)  
//=> 2
```

```
(1 + 1, 2 + 2)  
//=> 4
```

We can use commas with functions to create functions that evaluate multiple expressions:

```
(( ) => (1 + 1, 2 + 2))()  
//=> 4
```

This is useful when trying to do things that might involve *side-effects*, but we'll get to that later. In most cases, JavaScript does not care whether things are separated by spaces, tabs, or line breaks. So we can also write:

```
( ) =>  
(1 + 1, 2 + 2)
```

Or even:

```
( ) => (  
 1 + 1,  
 2 + 2  
)
```

the simplest possible block

There's another thing we can put to the right of an arrow, a *block*. A block has zero or more *statements*, separated by semicolons.¹⁷

So, this is a valid function:

```
( ) => {}
```

It returns the result of evaluating a block that has no statements. What would that be? Let's try it:

¹⁷Sometimes, you will find JavaScript that has statements that are separated by newlines without semi-colons. This works because JavaScript has a feature that can infer where the semi-colons should be most of the time. We will not take advantage of this feature, but it's helpful to know it exists.

```
((() => {}))()  
//=> undefined
```

What is this `undefined`?

undefined

In JavaScript, the absence of a value is written `undefined`, and it means there is no value. It will crop up again. `undefined` is its own type of value, and it acts like a value type:

```
undefined  
//=> undefined
```

Like numbers, booleans and strings, JavaScript can print out the value `undefined`.

```
undefined === undefined  
//=> true  
(() => {})() === ((() => {}))()  
//=> true  
(() => {})() === undefined  
//=> true
```

No matter how you evaluate `undefined`, you get an identical value back. `undefined` is a value that means “I don’t have a value.” But it’s still a value :-)

You might think that `undefined` in JavaScript is equivalent to `NULL` in SQL. No. In SQL, two things that are `NULL` are not equal to nor share the same identity, because two unknowns can’t be equal. In JavaScript, every `undefined` is identical to every other `undefined`.

void

We’ve seen that JavaScript represents an `undefined` value by typing `undefined`, and we’ve generated `undefined` values in two ways:

1. By evaluating a function that doesn’t return a value `((() => {}))()`, and;
2. By writing `undefined` ourselves.

There’s a third way, with JavaScript’s `void` operator. Behold:

```
void 0
  //=> undefined
void 1
  //=> undefined
void (2 + 2)
  //=> undefined
```

`void` is an operator that takes any value and evaluates to `undefined`, always. So, when we deliberately want an `undefined` value, should we use the first, second, or third form?¹⁸ The answer is, use `void`. By convention, use `void 0`.

The first form works but it's cumbersome. The second form works most of the time, but it is possible to break it by reassigning `undefined` to a different value, something we'll discuss in [Reassignment and Mutation](#). The third form is guaranteed to always work, so that's what we will use.¹⁹

back on the block

Back to our function. We evaluated this:

```
((() => {}))()
//=> undefined
```

We said that the function returns the result of evaluating a *block*, and we said that a block is a (possibly empty) list of JavaScript *statements* separated by semicolons.²⁰

Something like: { statement¹; statement²; statement³; ... ; statementⁿ }

We haven't discussed these *statements*. What's a statement?

There are many kinds of JavaScript statements, but the first kind is one we've already met. An expression is a JavaScript statement. Although they aren't very practical, these are valid JavaScript functions, and they return `undefined` when applied:

```
() => { 2 + 2 }
() => { 1 + 1; 2 + 2 }
```

As we saw with commas above, we can rearrange these functions onto multiple lines when we feel its more readable that way:

¹⁸Experienced JavaScript programmers are aware that there's a fourth way, using a function argument. This was actually the preferred mechanism until `void` became commonplace.

¹⁹As an exercise for the reader, we suggest you ask your friendly neighbourhood programming language designer or human factors subject-matter expert to explain why a keyword called `void` is used to generate an `undefined` value, instead of calling them both `void` or both `undefined`. We have no idea.

²⁰You can also separate statements with line breaks. Readers who follow internet flame-fests may be aware of something called [automatic semi-colon insertion](#). Basically, there's a step where JavaScript looks at your code and follows some rules to guess where you meant to put semicolons in should you leave them out. This feature was originally created as a kind of helpful error-correction. Some programmers argue that since it's part of the language's definition, it's fair game to write code that exploits it, so they deliberately omit any semicolon that JavaScript will insert for them.

```
() => {  
  1 + 1;  
  2 + 2  
}
```

But no matter how we arrange them, a block with one or more expressions still evaluates to `undefined`:

```
((() => { 2 + 2 })())  
//=> undefined  
  
((() => { 1 + 1; 2 + 2 })())  
//=> undefined  
  
(() => {  
  1 + 1;  
  2 + 2  
})()  
//=> undefined
```

As you can see, a block with one expression does not behave like an expression, and a block with more than one expression does not behave like an expression constructed with the comma operator:

```
((() => 2 + 2)())  
//=> 4  
((() => { 2 + 2 })())  
//=> undefined  
  
((() => (1 + 1, 2 + 2))())  
//=> 4  
((() => { 1 + 1; 2 + 2 })())  
//=> undefined
```

So how do we get a function that evaluates a block to return a value when applied? With the `return` keyword and any expression:

```
((() => { return 0 })())
//=> 0

(() => { return 1 })()
//=> 1

(() => { return 'Hello ' + 'World' })()
// 'Hello World'
```

The `return` keyword creates a *return statement* that immediately terminates the function application and returns the result of evaluating its expression. For example:

```
((() => {
  1 + 1;
  return 2 + 2
})())
//=> 4
```

And also:

```
((() => {
  return 1 + 1;
  2 + 2
})())
//=> 2
```

The `return` statement is the first statement we've seen, and it behaves differently than an expression. For example, you can't use one as the expression in a simple function, because it isn't an expression:

```
((() => return 0))()
//=> ERROR
```

Statements belong inside blocks and only inside blocks. Some languages simplify this by making everything an expression, but JavaScript maintains this distinction, so when learning JavaScript we also learn about statements like function declarations, for loops, if statements, and so forth. We'll see a few more of these later.

functions that evaluate to functions

If an expression that evaluates to a function is, well, an expression, and if a `return` statement can have any expression on its right side... *Can we put an expression that evaluates to a function on the right side of a function expression?*

Yes:

```
() => () => 0
```

That's a function! It's a function that when applied, evaluates to a function that when applied, evaluates to `0`. So we have *a function, that returns a function, that returns zero*. Likewise:

```
() => () => true
```

That's a function, that returns a function, that returns `true`:

```
((() => () => true))()
```

//=> true

We could, of course, do the same thing with a block if we wanted:

```
() => () => { return true; }
```

But we generally don't.

Well. We've been very clever, but so far this all seems very abstract. Diffraction of a crystal is beautiful and interesting in its own right, but you can't blame us for wanting to be shown a practical use for it, like being able to determine the composition of a star millions of light years away. So... In the next chapter, “[I'd Like to Have an Argument, Please](#),” we'll see how to make functions practical.

Ah. I'd Like to Have an Argument, Please.²¹

Up to now, we've looked at functions without arguments. We haven't even said what an argument *is*, only that our functions don't have any.

Most programmers are perfectly familiar with arguments (often called "parameters"). Secondary school mathematics discusses this. So you know what they are, and I know that you know what they are, but please be patient with the explanation!

Let's make a function with an argument:

```
(room) => {}
```

This function has one argument, `room`, and an empty body. Here's a function with two arguments and an empty body:

```
(room, board) => {}
```

I'm sure you are perfectly comfortable with the idea that this function has two arguments, `room`, and `board`. What does one do with the arguments? Use them in the body, of course. What do you think this is?

```
(diameter) => diameter * 3.14159265
```

It's a function for calculating the circumference of a circle given the diameter. I read that aloud as "When applied to a value representing the diameter, this function *returns* the diameter times 3.14159265."

Remember that to apply a function with no arguments, we wrote `(() => {})()`. To apply a function with an argument (or arguments), we put the argument (or arguments) within the parentheses, like this:

```
((diameter) => diameter * 3.14159265)(2)  
//=> 6.2831853
```

You won't be surprised to see how to write and apply a function to two arguments:

²¹Abuse of this feature by extending the behaviour of built-in classes is a controversial topic.

```
((room, board) => room + board)(800, 150)
//=> 950
```



a quick summary of functions and bodies

How arguments are used in a body's expression is probably perfectly obvious to you from the examples, especially if you've used any programming language (except for the dialect of BASIC—which I recall from my secondary school—that didn't allow parameters when you called a procedure).

Expressions consist either of representations of values (like `3.14159265`, `true`, and `undefined`), operators that combine expressions (like `3 + 2`), some special forms like `[1, 2, 3]` for creating arrays out of expressions, or function `(arguments) {body-statements}` for creating functions.

One of the important possible statements is a return statement. A return statement accepts any valid JavaScript expression.

This loose definition is recursive, so we can intuit (or use our experience with other languages) that since a function can contain a return statement with an expression, we can write a function that returns a function, or an array that contains another array expression. Or a function that returns an array, an array of functions, a function that returns an array of functions, and so forth:

```
() => () => {};
() => [ 1, 2, 3];
[1, [2, 3], 4];
() => [
  () => 1,
  () => 2,
  () => 3
];
```

call by value

Like most contemporary programming languages, JavaScript uses the “call by value” [evaluation strategy²²](#). That means that when you write some code that appears to apply a function to an expression or expressions, JavaScript evaluates all of those expressions and applies the functions to the resulting value(s).

²²http://en.wikipedia.org/wiki/Evaluation_strategy

So when you write:

```
((diameter) => diameter * 3.14159265)(1 + 1)
//=> 6.2831853
```

What happened internally is that the expression `1 + 1` was evaluated first, resulting in 2. Then our circumference function was applied to 2.²³

We'll see [below](#) that while JavaScript always calls by value, the notion of a "value" has additional subtlety. But before we do, let's look at variables.

variables and bindings

Right now everything looks simple and straightforward, and we can move on to talk about arguments in more detail. And we're going to work our way up from `(diameter) => diameter * 3.14159265` to functions like:

```
(x) => (y) => x
```

`(x) => (y) => x` just looks crazy, as if we are learning English as a second language and the teacher promises us that soon we will be using words like *antidisestablishmentarianism*. Besides a desire to use long words to sound impressive, this is not going to seem attractive until we find ourselves wanting to discuss the role of the Church of England in 19th century British politics.

But there's another reason for learning the word *antidisestablishmentarianism*: We might learn how prefixes and postfixes work in English grammar. It's the same thing with `(x) => (y) => x`. It has a certain important meaning in its own right, and it's also an excellent excuse to learn about functions that make functions, environments, variables, and more.

In order to talk about how this works, we should agree on a few terms (you may already know them, but let's check-in together and "synchronize our dictionaries"). The first `x`, the one in `(x) => ...`, is an *argument*. The `y` in function `(y) ...` is another argument. The second `x`, the one in `=> x`, is not an argument, it's an *expression referring to a variable*. Arguments and variables work the same way whether we're talking about `(x) => (y) => x` or just plain `(x) => x`.

Every time a function is invoked ("invoked" means "applied to zero or more arguments"), a new *environment* is created. An environment is a (possibly empty) dictionary that maps variables to values by name. The `x` in the expression that we call a "variable" is itself an expression that is evaluated by looking up the value in the environment.

²³We said that you can't apply a function to an expression. You *can* apply a function to one or more functions. Functions are values! This has interesting applications, and they will be explored much more thoroughly in [Functions That Are Applied to Functions](#).

How does the value get put in the environment? Well for arguments, that is very simple. When you apply the function to the arguments, an entry is placed in the dictionary for each argument. So when we write:

```
((x) => x)(2)  
//=> 2
```

What happens is this:

1. JavaScript parses this whole thing as an expression made up of several sub-expressions.
2. It then starts evaluating the expression, including evaluating sub-expressions
3. One sub-expression, `(x) => x` evaluates to a function.
4. Another, `2`, evaluates to the number `2`.
5. JavaScript now evaluates applying the function to the argument `2`. Here's where it gets interesting...
6. An environment is created.
7. The value '`2`' is bound to the name '`x`' in the environment.
8. The expression '`x`' (the right side of the function) is evaluated within the environment we just created.
9. The value of a variable when evaluated in an environment is the value bound to the variable's name in that environment, which is '`2`'
10. And that's our result.

When we talk about environments, we'll use an [unsurprising syntax²⁴](#) for showing their bindings: `{x: 2, ...}`. meaning, that the environment is a dictionary, and that the value `2` is bound to the name `x`, and that there might be other stuff in that dictionary we aren't discussing right now.

call by sharing

Earlier, we distinguished JavaScript's *value types* from its *reference types*. At that time, we looked at how JavaScript distinguishes objects that are identical from objects that are not. Now it is time to take another look at the distinction between value and reference types.

There is a property that JavaScript strictly maintains: When a value—any value—is passed as an argument to a function, the value bound in the function's environment must be identical to the original.

We said that JavaScript binds names to values, but we didn't say what it means to bind a name to a value. Now we can elaborate: When JavaScript binds a value-type to a name, it makes a copy of the value and places the copy in the environment. As you recall, value types like strings and numbers

²⁴<http://json.org/>

are identical to each other if they have the same content. So JavaScript can make as many copies of strings, numbers, or booleans as it wishes.

What about reference types? JavaScript does not place copies of reference values in any environment. JavaScript places *references* to reference types in environments, and when the value needs to be used, JavaScript uses the reference to obtain the original.

Because many references can share the same value, and because JavaScript passes references as arguments, JavaScript can be said to implement “call by sharing” semantics. Call by sharing is generally understood to be a specialization of call by value, and it explains why some values are known as value types and other values are known as reference types.

And with that, we’re ready to look at *closures*. When we combine our knowledge of value types, reference types, arguments, and closures, we’ll understand why this function always evaluates to true no matter what argument²⁵ you apply it to:

```
(value) =>
  (copy) =>
    copy === value
```

²⁵Unless the argument is NaN, which isn’t equal to anything, *including itself*. NaN in JavaScript behaves a lot like NULL in SQL.

Closures and Scope

It's time to see how a function within a function works:

```
((x) => (y) => x)(1)(2)
//=> 1
```

First off, let's use what we learned above. Given (*some function*)(*some argument*), we know that we apply the function to the argument, create an environment, bind the value of the argument to the name, and evaluate the function's expression. So we do that first with this code:

```
((x) => (y) => x)(1)
//=> [Function]
```

The environment belonging to the function with signature $(x) \Rightarrow \dots$ becomes $\{x: 1, \dots\}$, and the result of applying the function is another function value. It makes sense that the result value is a function, because the expression for $(x) \Rightarrow \dots$'s body is:

```
(y) => x
```

So now we have a value representing that function. Then we're going to take the value of that function and apply it to the argument 2, something like this:

```
((y) => x)(2)
```

So we seem to get a new environment $\{y: 2, \dots\}$. How is the expression x going to be evaluated in that function's environment? There is no x in its environment, it must come from somewhere else.

This, by the way, is one of the great defining characteristics of JavaScript and languages in the same family: Whether they allow things like functions to nest inside each other, and if so, how they handle variables from "outside" of a function that are referenced inside a function. For example, here's the equivalent code in Ruby:

```
lambda { |x|
  lambda { |y| x }
}[1][2]
#=> 1
```

Now let's enjoy a relaxed Allongé before we continue!

if functions without free variables are pure, are closures impure?

The function $(y) \Rightarrow x$ is interesting. It contains a *free variable*, x .²⁶ A free variable is one that is not bound within the function. Up to now, we've only seen one way to "bind" a variable, namely by passing in an argument with the same name. Since the function $(y) \Rightarrow x$ doesn't have an argument named x , the variable x isn't bound in this function, which makes it "free."

Now that we know that variables used in a function are either bound or free, we can bifurcate functions into those with free variables and those without:

- Functions containing no free variables are called *pure functions*.
- Functions containing one or more free variables are called *closures*.

Pure functions are easiest to understand. They always mean the same thing wherever you use them. Here are some pure functions we've already seen:

$() \Rightarrow \{ \}$

$(x) \Rightarrow x$

$(x) \Rightarrow (y) \Rightarrow x$

The first function doesn't have any variables, therefore doesn't have any free variables. The second doesn't have any free variables, because its only variable is bound. The third one is actually two functions, one inside the other. $(y) \Rightarrow \dots$ has a free variable, but the entire expression refers to $(x) \Rightarrow \dots$, and it doesn't have a free variable: The only variable anywhere in its body is x , which is certainly bound within $(x) \Rightarrow \dots$.

From this, we learn something: A pure function can contain a closure.



If pure functions can contain closures, can a closure contain a pure function? Using only what we've learned so far, attempt to compose a closure that contains a pure function. If you can't, give your reasoning for why it's impossible.

Pure functions always mean the same thing because all of their "inputs" are fully defined by their arguments. Not so with a closure. If I present to you this pure function $(x, y) \Rightarrow x + y$, we know exactly what it does with $(2, 2)$. But what about this closure: $(y) \Rightarrow x + y$? We can't say what it will do with argument (2) without understanding the magic for evaluating the free variable x .

²⁶You may also hear the term "non-local variable." Both are correct.

it's always the environment

To understand how closures are evaluated, we need to revisit environments. As we've said before, all functions are associated with an environment. We also hand-waved something when describing our environment. Remember that we said the environment for $((x) \Rightarrow (y) \Rightarrow x)(1)$ is $\{x: 1, \dots\}$ and that the environment for $((y) \Rightarrow x)(2)$ is $\{y: 2, \dots\}$? Let's fill in the blanks!

The environment for $((y) \Rightarrow x)(2)$ is *actually* $\{y: 2, \dots': \{x: 1, \dots\}\}$. \dots' means something like "parent" or "enclosure" or "super-environment." It's $(x) \Rightarrow \dots$'s environment, because the function $(y) \Rightarrow x$ is within $(x) \Rightarrow \dots$'s body. So whenever a function is applied to arguments, its environment always has a reference to its parent environment.

And now you can guess how we evaluate $((y) \Rightarrow x)(2)$ in the environment $\{y: 2, \dots': \{x: 1, \dots\}\}$. The variable x isn't in $(y) \Rightarrow \dots$'s immediate environment, but it is in its parent's environment, so it evaluates to 1 and that's what $((y) \Rightarrow x)(2)$ returns even though it ended up ignoring its own argument.

$(x) \Rightarrow x$ is called the I Combinator, or the *Identity Function*. $(x) \Rightarrow (y) \Rightarrow x$ is called the K Combinator, or *Kestrel*. Some people get so excited by this that they write entire books about them, some are [great^a](#), some-how shall I put this—are [interesting^b](#) if you use Ruby.

^a<http://www.amazon.com/0192801422?tag=raganwald001-20>

^b<https://leanpub.com/combinators>

Functions can have grandparents too:

```
(x) =>
(y) =>
(z) => x + y + z
```

This function does much the same thing as:

```
(x, y, z) => x + y + z
```

Only you call it with $(1)(2)(3)$ instead of $(1, 2, 3)$. The other big difference is that you can call it with (1) and get a function back that you can later call with $(2)(3)$.

The first function is the result of [currying^a](#) the second function. Calling a curried function with only some of its arguments is sometimes called [partial application^b](#). Some programming languages automatically curry and partially evaluate functions without the need to manually nest them.

^a<https://en.wikipedia.org/wiki/Currying>

^bhttps://en.wikipedia.org/wiki/Partial_application

shadowy variables from a shadowy planet

An interesting thing happens when a variable has the same name as an ancestor environment's variable. Consider:

```
(x) =>
  (x, y) => x + y
```

The function `(x, y) => x + y` is a pure function, because its `x` is defined within its own environment. Although its parent also defines an `x`, it is ignored when evaluating `x + y`. JavaScript always searches for a binding starting with the function's own environment and then each parent in turn until it finds one. The same is true of:

```
(x) =>
  (x, y) =>
    (w, z) =>
      (w) =>
        x + y + z
```

When evaluating `x + y + z`, JavaScript will find `x` and `y` in the great-grandparent scope and `z` in the parent scope. The `x` in the great-great-grandparent scope is ignored, as are both `w`s. When a variable has the same name as an ancestor environment's binding, it is said to *shadow* the ancestor.

This is often a good thing.

which came first, the chicken or the egg?

This behaviour of pure functions and closures has many, many consequences that can be exploited to write software. We are going to explore them in some detail as well as look at some of the other mechanisms JavaScript provides for working with variables and mutable state.

But before we do so, there's one final question: Where does the ancestry start? If there's no other code in a file, what is `(x) => x`'s parent environment?

JavaScript always has the notion of at least one environment we do not control: A global environment in which many useful things are bound such as libraries full of standard functions. So when you invoke `((x) => x)(1)` in the REPL, its full environment is going to look like this: `{x: 1, ...: global environment}`.

Sometimes, programmers wish to avoid this. If you don't want your code to operate directly within the global environment, what can you do? Create an environment for them, of course. Many programmers choose to write every JavaScript file like this:

```
// top of the file
() => {

  // ... lots of JavaScript ...

})();
// bottom of the file
```

The effect is to insert a new, empty environment in between the global environment and your own functions: `{x: 1, ...: {'...': global environment}}`. As we'll see when we discuss mutable state, this helps to prevent programmers from accidentally changing the global state that is shared by all code in the program.

That Constant Coffee Craving

Up to now, all we've really seen are *anonymous functions*, functions that don't have a name. This feels very different from programming in most other languages, where the focus is on naming functions, methods, and procedures. Naming things is a critical part of programming, but all we've seen so far is how to name arguments.

There are other ways to name things in JavaScript, but before we learn some of those, let's see how to use what we already have to name things. Let's revisit a very simple example:

```
(diameter) => diameter * 3.14159265
```

What is this “3.14159265” number? [PI²⁷](#), obviously. We'd like to name it so that we can write something like:

```
(diameter) => diameter * PI
```

In order to bind 3.14159265 to the name PI, we'll need a function with a parameter of PI applied to an argument of 3.14159265. If we put our function expression in parentheses, we can apply it to the argument of 3.14159265:

```
((PI) =>
// ???
)(3.14159265)
```

What do we put inside our new function that binds 3.14159265 to the name PI when evaluated? Our circumference function, of course:

```
((PI) =>
(diameter) => diameter * PI
)(3.14159265)
```

This expression, when evaluated, returns a function that calculates circumferences. That sounds bad, but when we think about it, `(diameter) => diameter * 3.14159265` is also an expression, that when evaluated, returns a function that calculates circumferences. All of our “functions” are expressions. This one has a few more moving parts, that's all. But we can use it just like `(diameter) => diameter * 3.14159265`.

Let's test it:

²⁷<https://en.wikipedia.org/wiki/Pi>

```
((diameter) => diameter * 3.14159265)(2)
//=> 6.2831853
```

```
((PI) =>
  (diameter) => diameter * PI
)(3.14159265)(2)
//=> 6.2831853
```

That works! We can bind anything we want in an expression by wrapping it in a function that is immediately invoked with the value we want to bind.²⁸

inside-out

There's another way we can make a function that binds 3.14159265 to the name PI and then uses that in its expression. We can turn things inside-out by putting the binding inside our diameter calculating function, like this:

```
(diameter) =>
  ((PI) =>
    diameter * PI)(3.14159265)
```

It produces the same result as our previous expressions for a diameter-calculating function:

```
((diameter) => diameter * 3.14159265)(2)
//=> 6.2831853
```

```
((PI) =>
  (diameter) => diameter * PI
)(3.14159265)(2)
//=> 6.2831853
```

```
((diameter) =>
  ((PI) =>
    diameter * PI)(3.14159265))(2)
//=> 6.2831853
```

Which one is better? Well, the first one seems simplest, but a half-century of experience has taught us that names matter. A “magic literal” like 3.14159265 is anathema to sustainable software development.

The third one is easiest for most people to read. It separates concerns nicely: The “outer” function describes its parameters:

²⁸JavaScript programmers regularly use the idea of writing an expression that denotes a function and then immediately applying it to arguments. Explaining the pattern, Ben Alman coined the term [Immediately Invoked Function Expression][ife] for it, often abbreviated “IIFE.”

```
(diameter) =>
// ...
```

Everything else is encapsulated in its body. That's how it should be, naming PI is its concern, not ours. The other formulation:

```
((PI) =>
// ...
)(3.14159265)
```

“Exposes” naming PI first, and we have to look inside to find out why we care. So, should we should always write this?

```
(diameter) =>
((PI) =>
  diameter * PI)(3.14159265)
```

Well, the wrinkle with this is that typically, invoking functions is considerably more expensive than evaluating expressions. Every time we invoke the outer function, we'll invoke the inner function. We could get around this by writing

```
((PI) =>
  (diameter) => diameter * PI
)(3.14159265)
```

But then we've obfuscated our code, and we don't want to do that unless we absolutely have to.

What would be very nice is if the language gave us a way to bind names inside of blocks without incurring the cost of a function invocation. And JavaScript does.

const

Another way to write our “circumference” function would be to pass PI along with the diameter argument, something like this:

```
(diameter, PI) => diameter * PI
```

And we could use it like this:

```
((diameter, PI) => diameter * PI)(2, 3.14159265)
//=> 6.2831853
```

This differs from our example above in that there is only one environment, rather than two. We have one binding in the environment representing our regular argument, and another our “constant.” That’s more efficient, and it’s *almost* what we wanted all along: A way to bind 3.14159265 to a readable name.

JavaScript gives us a way to do that, the `const` keyword. We’ll learn a lot more about `const` in future chapters, but here’s the most important thing we can do with `const`:

```
(diameter) => {
  const PI = 3.14159265;

  return diameter * PI
}
```

The `const` keyword introduces one or more bindings in the block that encloses it. It doesn’t incur the cost of a function invocation. That’s great. Even better, it puts the symbol (like `PI`) close to the value (`3.14159265`). That’s much better than what we were writing.

We use the `const` keyword in a *const statement*. `const` statements occur inside blocks, we can’t use them when we write a fat arrow that has an expression as its body.

It works just as we want. Instead of:

```
((diameter) =>
  ((PI) =>
    diameter * PI)(3.14159265))(2)
```

Or:

```
((diameter, PI) => diameter * PI)(2, 3.14159265)
//=> 6.2831853
```

We write:

```
((diameter) => {
  const PI = 3.14159265;

  return diameter * PI
})(2)
//=> 6.2831853
```

We can bind any expression. Functions are expressions, so we can bind helper functions:

```
(d) => {
  const calc = (diameter) => {
    const PI = 3.14159265;

    return diameter * PI
  };

  return "The circumference is " + calc(d)
}
```

Notice `calc(d)`? This underscores what we've said: if we have an expression that evaluates to a function, we apply it with `()`. A name that's bound to a function is a valid expression evaluating to a function.²⁹

Amazing how such an important idea—naming functions—can be explained *en passant* in just a few words. That emphasizes one of the things JavaScript gets really, really right: Functions as “first class entities.” Functions are values that can be bound to names like any other value, passed as arguments, returned from other functions, and so forth.

We can bind more than one name-value pair by separating them with commas. For readability, most people put one binding per line:

```
(d) => {
  const PI = 3.14159265,
  calc = (diameter) => diameter * PI;

  return "The circumference is " + calc(d)
}
```

²⁹We're into the second chapter and we've finally named a function. Sheesh.

nested blocks

Up to now, we've only ever seen blocks we use as the body of functions. But there are other kinds of blocks. One of the places you can find blocks is in an `if` statement. In JavaScript, an `if` statement looks like this:

```
(n) => {
  const even = (x) => {
    if (x === 0)
      return true;
    else
      return !even(x - 1);
  }
  return even(n)
}
```

And it works for fairly small numbers:

```
((n) => {
  const even = (x) => {
    if (x === 0)
      return true;
    else
      return !even(x - 1);
  }
  return even(n)
})(13)
//=> false
```

The `if` statement is a statement, not an expression (an unfortunate design choice), and its clauses are statements or blocks. So we could also write something like:

```
(n) => {
  const even = (x) => {
    if (x === 0)
      return true;
    else {
      const odd = (y) => !even(y);

      return odd(x - 1);
    }
  }
}
```

```

    }
    return even(n)
}

```

And this also works:

```

((n) => {
  const even = (x) => {
    if (x === 0)
      return true;
    else {
      const odd = (y) => !even(y);

      return odd(x - 1);
    }
  }
  return even(n)
})(42)
//=> true

```

We've used a block as the `else` clause, and since it's a block, we've placed a `const` statement inside it.

const and lexical scope

This seems very straightforward, but alas, there are some semantics of binding names that we need to understand if we're to place `const` anywhere we like. The first thing to ask ourselves is, what happens if we use `const` to bind two different values to the "same" name?

Let's back up and reconsider how closures work. What happens if we use parameters to bind two different values to the same name?

Here's the second formulation of our diameter function, bound to a name using an IIFE:

```

((diameter_fn) =>
  // ...
)(
  ((PI) =>
    (diameter) => diameter * PI
  )(3.14159265)
)

```

It's more than a bit convoluted, but it binds `((PI) => (diameter) => diameter * PI)(3.14159265)` to `diameter_fn` and evaluates the expression that we've elided. We can use any expression in there, and that expression can invoke `diameter_fn`. For example:

```
((diameter_fn) =>
  diameter_fn(2)
)(
  ((PI) =>
    (diameter) => diameter * PI
  )(3.14159265)
)
//=> 6.2831853
```

We know this from the chapter on [closures](#), but even though PI is not bound when we invoke diameter_fn by evaluating diameter_fn(2), PI *is* bound when we evaluated (diameter) => diameter * PI, and thus the expression diameter * PI is able to access values for PI and diameter when we evaluate diameter_fn.

This is called [lexical scoping](#)³⁰, because we can discover where a name is bound by looking at the source code for the program. We can see that PI is bound in an environment surrounding (diameter) => diameter * PI, we don't need to know where diameter_fn is invoked.

We can test this by deliberately creating a “conflict:”

```
((diameter_fn) =>
  ((PI) =>
    diameter_fn(2)
  )(3)
)(
  ((PI) =>
    (diameter) => diameter * PI
  )(3.14159265)
)
//=> 6.2831853
```

Although we have bound 3 to PI in the environment surrounding diameter_fn(2), the value that counts is 3.14159265, the value we bound to PI in the environment surrounding (diameter) => diameter * PI.

That much we can carefully work out from the way closures work. Does const work the same way? Let's find out:

³⁰[https://en.wikipedia.org/wiki/Scope_\(computer_science\)#Lexical_scope_vs._dynamic_scope](https://en.wikipedia.org/wiki/Scope_(computer_science)#Lexical_scope_vs._dynamic_scope)

```
((diameter_fn) => {
  const PI = 3;

  return diameter_fn(2)
})(
  () => {
    const PI = 3.14159265;

    return (diameter) => diameter * PI
  }()
)
//=> 6.2831853
```

Yes. Binding values to names with `const` works just like binding values to names with parameter invocations, it uses lexical scope.

are consts also from a shadowy planet?

We just saw that values bound with `const` use lexical scope, just like values bound with parameters. They are looked up in the environment where they are declared. And we know that functions create environments. Parameters are declared when we create functions, so it makes sense that parameters are bound to environments created when we invoke functions.

But `const` statements can appear inside blocks, and we saw that blocks can appear inside of other blocks, including function bodies. So where are `const` variables bound? In the function environment? Or in an environment corresponding to the block?

We can test this by creating another conflict. But instead of binding two different variables to the same name in two different places, we'll bind two different values to the same name, but one environment will be completely enclosed by the other.

Let's start, as above, by doing this with parameters. We'll start with:

```
((PI) =>
  (diameter) => diameter * PI
)(3.14159265)
```

And gratuitously wrap it in another IIFE so that we can bind `PI` to something else:

```
((PI) =>
  ((PI) =>
    (diameter) => diameter * PI
  )(3.14159265)
)(3)
```

This still evaluates to a function that calculates diameters:

```
((PI) =>
  ((PI) =>
    (diameter) => diameter * PI
  )(3.14159265)
)(3)(2)
//=> 6.2831853
```

And we can see that our `diameter * PI` expression uses the binding for `PI` in the closest parent environment. but one question: Did binding `3.14159265` to `PI` somehow change the binding in the “outer” environment? Let’s rewrite things slightly differently:

```
((PI) => {
  ((PI) => {})(3);

  return (diameter) => diameter * PI;
})(3.14159265)
```

Now we bind `3` to `PI` in an otherwise empty IIFE inside of our IIFE that binds `3.14159265` to `PI`. Does that binding “overwrite” the outer one? Will our function return `6` or `6.2831853`? This is a book, you’ve already scanned ahead, so you know that the answer is **no**, the inner binding does not overwrite the outer binding:

```
((PI) => {
  ((PI) => {})(3);

  return (diameter) => diameter * PI;
})(3.14159265)(2)
//=> 6.2831853
```

We say that when we bind a variable using a parameter inside another binding, the inner binding *shadows* the outer binding. It has effect inside its own scope, but does not affect the binding in the enclosing scope.

So what about `const`. Does it work the same way?

```
((diameter) => {
  const PI = 3.14159265;

  () => {
    const PI = 3;
  })();

  return diameter * PI;
})()
//=> 6.2831853
```

Yes, names bound with `const` shadow enclosing bindings just like parameters. But wait! There's more!!!

Parameters are only bound when we invoke a function. That's why we made all these IIFEs. But `const` statements can appear inside blocks. What happens when we use a `const` inside of a block?

We'll need a gratuitous block. We've seen `if` statements, what could be more gratuitous than:

```
if (true) {
  // an immediately invoked block statement (IIBS)
}
```

Let's try it:

```
((diameter) => {
  const PI = 3;

  if (true) {
    const PI = 3.14159265;

    return diameter * PI;
  }
})()
//=> 6.2831853

((diameter) => {
  const PI = 3.14159265;

  if (true) {
    const PI = 3;
  }
  return diameter * PI;
```

```
})(2)
//=> 6.2831853
```

Ah! `const` statements don't just shadow values bound within the environments created by functions, they shadow values bound within environments created by blocks!

This is enormously important. Consider the alternative: What if `const` could be declared inside of a block, but it always bound the name in the function's scope. In that case, we'd see things like this:

```
((diameter) => {
  const PI = 3.14159265;

  if (true) {
    const PI = 3;
  }
  return diameter * PI;
})(2)
//=> would return 6 if const had function scope
```

If `const` always bound its value to the name defined in the function's environment, placing a `const` statement inside of a block would merely rebind the existing name, overwriting its old contents. That would be super-confusing. And this code would "work:"

```
((diameter) => {
  if (true) {
    const PI = 3.14159265;
  }
  return diameter * PI;
})(2)
//=> would return 6.2831853 if const had function scope
```

Again, confusing. Typically, we want to bind our names as close to where we need them as possible. This design rule is called the [Principle of Least Privilege³¹](#), and it has both quality and security implications. Being able to bind a name inside of a block means that if the name is only needed in the block, we are not "leaking" its binding to other parts of the code that do not need to interact with it.

rebinding

By default, JavaScript permits us to *rebind* new values to names bound with a parameter. For example, we can write:

³¹https://en.wikipedia.org/wiki/Principle_of_least_privilege

```
const evenStevens = (n) => {
  if (n === 0) {
    return true;
  }
  else if (n == 1) {
    return false;
  }
  else {
    n = n - 2;
    return evenStevens(n);
  }
}

evenStevens(42)
//=> true
```

The line `n = n - 2;` *rebinds* a new value to the name `n`. We will discuss this at much greater length in [Reassignment](#), but long before we do, let's try a similar thing with a name bound using `const`. We've already bound `evenStevens` using `const`, let's try rebinding it:

```
evenStevens = (n) => {
  if (n === 0) {
    return true;
  }
  else if (n == 1) {
    return false;
  }
  else {
    return evenStevens(n - 2);
  }
}
//=> ERROR, evenStevens is read-only
```

JavaScript does not permit us to rebind a name that has been bound with `const`. We can *shadow* it by using `const` to declare a new binding with a new function or block scope, but we cannot rebind a name that was bound with `const` in an existing scope.

This is valuable, as it greatly simplifies the analysis of programs to see at a glance that when something is bound with `const`, we need never worry that its value may change.

Naming Functions

Let's get right to it. This code does *not* name a function:

```
const repeat = (str) => str + str
```

It doesn't name the function "repeat" for the same reason that `const answer = 42` doesn't name the number 42. This syntax binds an anonymous function to a name in an environment, but the function itself remains anonymous.

the function keyword

JavaScript *does* have a syntax for naming a function, we use the `function` keyword. Until ECMAScript 2015 was created, `function` was the usual syntax for writing functions.

Here's our `repeat` function written using a "fat arrow"

```
(str) => str + str
```

And here's (almost) the exact same function written using the `function` keyword:

```
function (str) { return str + str }
```

Let's look at the obvious differences:

1. We introduce a function with the `function` keyword.
2. Something else we're about to discuss is optional.
3. We have arguments in parentheses, just like fat arrow functions.
4. We do not have a fat arrow, we go directly to the body.
5. We always use a block, we cannot write `function (str) str + str`. This means that if we want our functions to return a value, we always need to use the `return` keyword

If we leave out the "something optional" that comes after the `function` keyword, we can translate all of the fat arrow functions that we've seen into `function` keyword functions, e.g.

```
(n) => (1.618**n - -1.618**-n) / 2.236
```

Can be written as:

```
function (n) {
  return (1.618**n - -1.618**-n) / 2.236;
}
```

This still does not *name* a function, but as we noted above, functions written with the `function` keyword have an optional “something else.” Could that “something else” name a function? Yes, of course.³²

Here are our example functions written with names:

```
const repeat = function repeat (str) {
  return str + str;
};

const fib = function fib (n) {
  return (1.618**n - -1.618**-n) / 2.236;
};
```

Placing a name between the `function` keyword and the argument list names the function. Confusingly, the name of the function is not exactly the same thing as the name we may choose to bind to the value of the function. For example, we can write:

```
const double = function repeat (str) {
  return str + str;
}
```

In this expression, `double` is the name in the environment, but `repeat` is the function’s actual name. This is a *named function expression*. That may seem confusing, but think of the binding names as properties of the environment, not of the function. While the name of the function is a property of the function, not of the environment.

And indeed the name *is* a property:

```
double.name
//=> 'repeat'
```

In this book we are not examining JavaScript’s tooling such as debuggers baked into browsers, but we will note that when you are navigating call stacks in all modern tools, the function’s binding name is ignored but its actual name is displayed, so naming functions is very useful even if they don’t get a formal binding, e.g.

³²“Yes of course?” Well, in chapter of a book dedicated to naming functions, it is not surprising that feature we mention has something to do with naming functions.

```
someBackboneView.on('click', function clickHandler () {
  //...
});
```

Now, the function's actual name has no effect on the environment in which it is used. To whit:

```
const bindingName = function actualName () {
  //...
};

bindingName
//=> [Function: actualName]

actualName
//=> ReferenceError: actualName is not defined
```

So “actualName” isn’t bound in the environment where we use the named function expression. Is it bound anywhere else? Yes it is. Here’s a function that determines whether a positive integer is even or not. We’ll use it in an IIFE so that we don’t have to bind it to a name with const:

```
(function even (n) {
  if (n === 0) {
    return true
  }
  else return !even(n - 1)
})(5)
//=> false

(function even (n) {
  if (n === 0) {
    return true
  }
  else return !even(n - 1)
})(2)
//=> true
```

Clearly, the name even is bound to the function *within the function’s body*. Is it bound to the function outside of the function’s body?

```
even
//=> Can't find variable: even
```

even is bound within the function itself, but not outside it. This is useful for making recursive functions as we see above, and it speaks to the principle of least privilege: If you don't *need* to name it anywhere else, you needn't.

function declarations

There is another syntax for naming and/or defining a function. It's called a *function declaration statement*, and it looks a lot like a named function expression, only we use it as a statement:

```
function someName () {
  // ...
}
```

This behaves a *little* like:

```
const someName = function someName () {
  // ...
}
```

In that it binds a name in the environment to a named function. However, there are two important differences. First, function declarations are *hoisted* to the top of the function in which they occur.

Consider this example where we try to use the variable fizzbuzz as a function before we bind a function to it with const:

```
(function () {
  return fizzbuzz();

  const fizzbuzz = function fizzbuzz () {
    return "Fizz" + "Buzz";
  }
})()
//=> undefined is not a function (evaluating 'fizzbuzz()')
```

We haven't actually bound a function to the name fizzbuzz before we try to use it, so we get an error. But a function *declaration* works differently:

```
(function () {
    return fizzbuzz();

    function fizzbuzz () {
        return "Fizz" + "Buzz";
    }
})()
//=> 'FizzBuzz'
```

Although `fizzbuzz` is declared later in the function, JavaScript behaves as if we'd written:

```
(function () {
    const fizzbuzz = function fizzbuzz () {
        return "Fizz" + "Buzz";
    }

    return fizzbuzz();
})()
```

The definition of the `fizzbuzz` is “hoisted” to the top of its enclosing scope (an IIFE in this case). This behaviour is intentional on the part of JavaScript’s design to facilitate a certain style of programming where you put the main logic up front, and the “helper functions” at the bottom. It is not necessary to declare functions in this way in JavaScript, but understanding the syntax and its behaviour (especially the way it differs from `const`) is essential for working with production code.

function declaration caveats³³

Function declarations are formally only supposed to be made at what we might call the “top level” of a function. Although some JavaScript environments permit the following code, this example is technically illegal and definitely a bad idea:

³³A number of the caveats discussed here were described in Jyrly Zaytsev’s excellent article [Named function expressions demystified](#).

```
(function (camelCase) {
    return fizzbuzz();

    if (camelCase) {
        function fizzbuzz () {
            return "Fizz" + "Buzz";
        }
    }
    else {
        function fizzbuzz () {
            return "Fizz" + "Buzz";
        }
    }
})(true)
//=> 'FizzBuzz'? Or ERROR: Can't find variable: fizzbuzz?
```

Function declarations are not supposed to occur inside of blocks. The big trouble with expressions like this is that they may work just fine in your test environment but work a different way in production. Or it may work one way today and a different way when the JavaScript engine is updated, say with a new optimization.

Another caveat is that a function declaration cannot exist inside of *any* expression, otherwise it's a function expression. So this is a function declaration:

```
function trueDat () { return true }
```

But this is not:

```
(function trueDat () { return true })
```

The parentheses make this an expression, not a function declaration.

Combinators and Function Decorators

higher-order functions

As we've seen, JavaScript functions take values as arguments and return values. JavaScript functions are values, so JavaScript functions can take functions as arguments, return functions, or both. Generally speaking, a function that either takes functions as arguments, or returns a function, or both, is referred to as a "higher-order" function.

Here's a very simple higher-order function that takes a function as an argument:

```
const repeat = (num, fn) =>
  (num > 0)
    ? (repeat(num - 1, fn), fn(num))
    : undefined

repeat(3, function (n) {
  console.log(`Hello ${n}`)
})
//=>
'Hello 1'
'Hello 2'
'Hello 3'
undefined
```

Higher-order functions dominate *JavaScript Allongé*. But before we go on, we'll talk about some specific types of higher-order functions.

combinators

The word "combinator" has a precise technical meaning in mathematics:

"A combinator is a higher-order function that uses only function application and earlier defined combinators to define a result from its arguments."—[Wikipedia³⁴](#)

If we were learning Combinatorial Logic, we'd start with the most basic combinators like **S**, **K**, and **I**, and work up from there to practical combinators. We'd learn that the fundamental combinators are named after birds following the example of Raymond Smullyan's famous book [To Mock a Mockingbird³⁵](#).

³⁴https://en.wikipedia.org/wiki/Combinatory_logic

³⁵http://www.amazon.com/gp/product/B00A1P096Y/ref=as_li_ss_til?ie=UTF8&camp=1789&creative=390957&creativeASIN=B00A1P096Y&linkCode=as2&tag=raganwald001-20

In this book, we will be using a looser definition of “combinator:” Higher-order pure functions that take only functions as arguments and return a function. We won’t be strict about using only previously defined combinators in their construction.

Let’s start with a useful combinator: Most programmers call it *Compose*, although the logicians call it the B combinator or “Bluebird.” Here is the typical³⁶ programming implementation:

```
const compose = (a, b) =>
  (c) => a(b(c))
```

Let’s say we have:

```
const addOne = (number) => number + 1;

const doubleOf = (number) => number * 2;
```

With `compose`, anywhere you would write

```
const doubleOfAddOne = (number) => doubleOf(addOne(number));
```

You could also write:

```
const doubleOfAddOne = compose(doubleOf, addOne);
```

This is, of course, just one example of many. You’ll find lots more perusing the recipes in this book. While some programmers believe “There Should Only Be One Way To Do It,” having combinators available as well as explicitly writing things out with lots of symbols and keywords has some advantages when used judiciously.

a balanced statement about combinators

Code that uses a lot of combinators tends to name the verbs and adverbs (like `doubleOf`, `addOne`, and `compose`) while avoiding language keywords and the names of nouns (like `number`). So one perspective is that combinators are useful when you want to emphasize what you’re doing and how it fits together, and more explicit code is useful when you want to emphasize what you’re working with.

function decorators

A *function decorator* is a higher-order function that takes one function as an argument, returns another function, and the returned function is a variation of the argument function. Here’s a ridiculously simple decorator:³⁷

³⁶As we’ll discuss later, this implementation of the B Combinator is correct in languages like Scheme, but for truly general-purpose use in JavaScript, it needs to correctly manage the `function context`.

³⁷We’ll see later why an even more useful version would be written `(fn) => (...args) => !fn(...args)`

```
const not = (fn) => (x) => !fn(x)
```

So instead of writing `!someFunction(42)`, we can write `not(someFunction)(42)`. Hardly progress. But like `compose`, we could write either:

```
const something = (x) => x != null;
```

And elsewhere, write:

```
const nothing = (x) => !something(x);
```

Or we could write:

```
const nothing = not(something);
```

`not` is a function decorator because it modifies a function while remaining strongly related to the original function's semantics. You'll see other function decorators in the recipes, like `once` and `maybe`. Function decorators aren't strict about being pure functions, so there's more latitude for making decorators than combinators.

Building Blocks

When you look at functions within functions in JavaScript, there's a bit of a "spaghetti code" look to it. The strength of JavaScript is that you can do anything. The weakness is that you will. There are ifs, fors, returns, everything thrown higgledy piggledy together. Although you needn't restrict yourself to a small number of simple patterns, it can be helpful to understand the patterns so that you can structure your code around some basic building blocks.

composition

One of the most basic of these building blocks is *composition*:

```
const cookAndEat = (food) => eat(cook(food));
```

It's really that simple: Whenever you are chaining two or more functions together, you're composing them. You can compose them with explicit JavaScript code as we've just done. You can also generalize composition with the B Combinator or "compose" that we saw in [Combinators and Decorators](#):

```
const compose = (a, b) => (c) => a(b(c));
```

```
const cookAndEat = compose(eat, cook);
```

If that was all there was to it, composition wouldn't matter much. But like many patterns, using it when it applies is only 20% of the benefit. The other 80% comes from organizing your code such that you can use it: Writing functions that can be composed in various ways.

In the recipes, we'll look at a decorator called `once`: It ensures that a function can only be executed once. Thereafter, it does nothing. Once is useful for ensuring that certain side effects are not repeated. We'll also look at `maybe`: It ensures that a function does nothing if it is given nothing (like `null` or `undefined`) as an argument.

Of course, you needn't use combinators to implement either of these ideas, you can use if statements. But `once` and `maybe` compose, so you can chain them together as you see fit:

```
const actuallyTransfer= (from, to, amount) =>
  // do something

const invokeTransfer = once(maybe(actuallyTransfer(...)));
```

partial application

Another basic building block is *partial application*. When a function takes multiple arguments, we “apply” the function to the arguments by evaluating it with all of the arguments, producing a value. But what if we only supply some of the arguments? In that case, we can’t get the final value, but we can get a function that represents *part* of our application.

Code is easier than words for this. The [Underscore³⁸](#) library provides a higher-order function called *map*.³⁹ It applies another function to each element of an array, like this:

```
_._map([1, 2, 3], (n) => n * n)
//=> [1, 4, 9]
```

We don’t want to fool around writing `_.`, so we can use it by writing:⁴⁰

This code implements a partial application of the map function by applying the function `(n) => n * n` as its second argument:

```
const squareAll = (array) => map(array, (n) => n * n);
```

The resulting function—`squareAll`—is still the map function, it’s just that we’ve applied one of its two arguments already. `squareAll` is nice, but why write one function every time we want to partially apply a function to a map? We can abstract this one level higher. `mapWith` takes any function as an argument and returns a partially applied map function.

```
const mapWith = (fn) =>
  (array) => map(array, fn);

const squareAll = mapWith((n) => n * n);

squareAll([1, 2, 3])
//=> [1, 4, 9]
```

We’ll discuss `mapWith` again. The important thing to see is that partial application is orthogonal to composition, and that they both work together nicely:

³⁸<http://underscorejs.org>

³⁹Modern JavaScript implementations provide a `map` method for arrays, but Underscore’s implementation also works with older browsers if you are working with that headache.

⁴⁰If we don’t want to sort out [Underscore](#), we can also write the following: `const map = (a, fn) => a.map(fn);`, and trust that it works even though we haven’t discussed methods yet.

`const map = _._map;`

```
const safeSquareAll = mapWith(maybe((n) => n * n));  
  
safeSquareAll([1, null, 2, 3])  
//=> [1, null, 4, 9]
```

We generalized composition with the compose combinator. Partial application also has a combinator, which we'll see in the [partial](#) recipe.

Magic Names

When a function is applied to arguments (or “called”), JavaScript binds the values of arguments to the function’s argument names in an environment created for the function’s execution. What we haven’t discussed so far is that JavaScript also binds values to some “magic” names in addition to any you put in the argument list.⁴¹

the function keyword

There are two separate rules for these “magic” names, one for when you invoke a function using the `function` keyword, and another for functions defined with “fat arrows.” We’ll begin with how things work for functions defined with the `function` keyword.

The first magic name is `this`, and it is bound to something called the function’s `context`. We will explore `this` in more detail when we start discussing objects and classes. The second magic name is very interesting, it’s called `arguments`, and the most interesting thing about it is that it contains a list of arguments passed to a function:

```
const plus = function (a, b) {
  return arguments[0] + arguments[1];
}

plus(2,3)
//=> 5
```

Although `arguments` looks like an array, it isn’t an array: It’s more like an object⁴² that happens to bind some values to properties with names that look like integers starting with zero:

```
const args = function (a, b) {
  return arguments;
}

args(2,3)
//=> { '0': 2, '1': 3 }
```

`arguments` always contains all of the arguments passed to a function, regardless of how many are declared. Therefore, we can write `plus` like this:

⁴¹You should never attempt to define your own bindings against “magic” names that JavaScript binds for you. It is wise to treat them as read-only at all times.

⁴²We’ll look at `arrays` and plain old javascript objects in depth later.

```
const plus = function () {
  return arguments[0] + arguments[1];
}

plus(2,3)
//=> 5
```

When discussing objects, we'll discuss properties in more depth. Here's something interesting about arguments:

```
const howMany = function () {
  return arguments['length'];
}

howMany()
//=> 0

howMany('hello')
//=> 1

howMany('sharks', 'are', 'apex', 'predators')
//=> 4
```

The most common use of the arguments binding is to build functions that can take a variable number of arguments. We'll see it used in many of the recipes, starting off with [partial application](#) and [ellipses](#).

magic names and fat arrows

The magic names `this` and `arguments` have a different behaviour when you invoke a function that was defined with a fat arrow: Instead of being bound when the function is invoked, the fat arrow function always acquires the bindings for `this` and `arguments` from its enclosing scope, just like any other binding.

For example, when this expression's inner function is defined with `function`, `arguments[0]` refers to its only argument, "inner":

```
(function () {
  return (function () { return arguments[0]; })('inner');
})('outer')
//=> "inner"
```

But if we use a fat arrow, `arguments` will be defined in the outer environment, the one defined with `function`. And thus `arguments[0]` will refer to "outer", not to "inner":

```
(function () {
  return ((() => arguments[0])('inner'));
})('outer')
//=> "outer"
```

Although it seems quixotic for the two syntaxes to have different semantics, it makes sense when you consider the design goal: Fat arrow functions are designed to be very lightweight and are often used with constructs like mapping or callbacks to emulate syntax.

To give a contrived example, this function takes a number and returns an array representing a row in a hypothetical multiplication table. It uses `mapWith`, which we discussed in [Building Blocks](#).⁴³ We'll use `arguments` just to show the difference between using a fat arrow and the `function` keyword:

```
const row = function () {
  return mapWith(
    (column) => column * arguments[0],
    [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12]
  )
}

row(3)
//=> [3, 6, 9, 12, 15, 18, 21, 24, 27, 30, 33, 36]
```

This works just fine, because `arguments[0]` refers to the 3 we passed to the function `row`. Our “fat arrow” function `(column) => column * arguments[0]` doesn’t bind `arguments` when it’s invoked. But if we rewrite `row` to use the `function` keyword, it stops working:

```
const row = function () {
  return mapWith(
    function (column) { return column * arguments[0] },
    [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12]
  )
}

row(3)
//=> [1, 4, 9, 16, 25, 36, 49, 64, 81, 100, 121, 144]
```

Now our inner function binds `arguments[0]` every time it is invoked, so we get the same result as if we’d written `function (column) { return column * column }`.

⁴³If we were always `mapWith`-ing arrays, we could write `list.map(fn)`. However, there are some objects that have a `.length` property and `[]` accessors that can be `mapWith`ed but do not have a `.map` method. `mapWith` works with those objects. This points to a larger issue around the question of whether containers really ought to implement methods like `.map`. In a language like JavaScript, we are free to define objects that know about their own implementations, such as exactly how `[]` and `.length` works and then to define standalone functions that do the rest.

Although this example is clearly unrealistic, there is a general design principle that deserves attention. Sometimes, a function is meant to be used as a Big-F function. It has a name, it is called by different pieces of code, it's a first-class entity in the code.

But sometimes, a function is a small-f function. It's a simple representation of an expression to be computed. In our example above, `row` is a Big-F function, but `(column) => column * arguments[0]` is a small-f function, it exists just to give `mapWith` something to apply.

Having magic variables apply to Big-F functions but not to small-G functions makes it much easier to use small-F functions as syntax, treating them as expressions or blocks that can be passed to functions like `mapWith`.

Summary



Functions

- Functions are values that can be part of expressions, returned from other functions, and so forth.
- Functions are *reference values*.
- Functions are applied to arguments.
- The arguments are passed by sharing, which is also called “pass by value.”
- Fat arrow functions have expressions or blocks as their bodies.
- `function` keyword functions always have blocks as their bodies.
- Function bodies have zero or more statements.
- Expression bodies evaluate to the value of the expression.
- Block bodies evaluate to whatever is returned with the `return` keyword, or to `undefined`.
- JavaScript uses `const` to bind values to names within block scope.
- JavaScript uses function declarations to bind functions to names within function scope. Function declarations are “hoisted.”
- Function application creates a scope.
- Blocks also create scopes if `const` statements are within them.
- Scopes are nested and free variable references closed over.
- Variables can shadow variables in an enclosing scope.

Recipes with Basic Functions



Before combining ingredients, begin with implements so clean, they gleam.

Having looked at basic pure functions and closures, we're going to see some practical recipes that focus on the premise of functions that return functions.

Disclaimer

The recipes are written for practicality, and their implementation may introduce JavaScript features that haven't been discussed in the text to this point, such as methods and/or prototypes. The overall *use* of each recipe will fit within the spirit of the language discussed so far, even if the implementations may not.

Partial Application

In [Building Blocks](#), we discussed partial application, but we didn't write a generalized recipe for it. This is such a common tool that many libraries provide some form of partial application. You'll find examples in [Lemonad](#)⁴⁴ from Michael Fogus, [Functional JavaScript](#)⁴⁵ from Oliver Steele and the terse but handy [node-ap](#)⁴⁶ from James Halliday.

These two recipes are for quickly and simply applying a single argument, either the leftmost or rightmost.⁴⁷ If you want to bind more than one argument, or you want to leave a "hole" in the argument list, you will need to either use a [generalized partial recipe](#), or you will need to repeatedly apply arguments. They are [context](#)-agnostic.

```
const callFirst = (fn, larg) =>
  function (...rest) {
    return fn.call(this, larg, ...rest);
  }

const callLast = (fn, rarg) =>
  function (...rest) {
    return fn.call(this, ...rest, rarg);
  }

const greet = (me, you) =>
  `Hello, ${you}, my name is ${me}`;

const heliosSaysHello = callFirst(greet, 'Helios');

heliosSaysHello('Earth')
//=> 'Hello, Earth, my name is Helios'

const sayHelloToCeline = callLast(greet, 'Celine');

sayHelloToCeline('Earth')
//=> 'Hello, Celine, my name is Earth'
```

As noted above, our partial recipe allows us to create functions that are partial applications of functions that are context aware. We'd need a different recipe if we wish to create partial applications of object methods.

⁴⁴<https://github.com/fogus/lemonad>

⁴⁵<http://osteel.com/sources/javascript/functional/>

⁴⁶<https://github.com/substack/node-ap>

⁴⁷callFirst and callLast were inspired by Michael Fogus' [Lemonad](#). Thanks!

We take it a step further, and can use gathering and spreading to allow for partial application with more than one argument:

```
const callLeft = (fn, ...args) =>
  (...remainingArgs) =>
    fn(...args, ...remainingArgs);
```

```
const callRight = (fn, ...args) =>
  (...remainingArgs) =>
    fn(...remainingArgs, ...args);
```

Unary

“Unary” is a function decorator that modifies the number of arguments a function takes: Unary takes any function and turns it into a function taking exactly one argument.

The most common use case is to fix a problem. JavaScript has a `.map` method for arrays, and many libraries offer a `map` function with the same semantics. Here it is in action:

```
[ '1', '2', '3' ].map(parseFloat)
//=> [1, 2, 3]
```

In that example, it looks exactly like the mapping function you’ll find in most languages: You pass it a function, and it calls the function with one argument, the element of the array. However, that’s not the whole story. JavaScript’s `map` actually calls each function with *three* arguments: The element, the index of the element in the array, and the array itself.

Let’s try it:

```
[1, 2, 3].map(function (element, index, arr) {
  console.log({element: element, index: index, arr: arr})
})
//=> { element: 1, index: 0, arr: [ 1, 2, 3 ] }
// { element: 2, index: 1, arr: [ 1, 2, 3 ] }
// { element: 3, index: 2, arr: [ 1, 2, 3 ] }
```

If you pass in a function taking only one argument, it simply ignores the additional arguments. But some functions have optional second or even third arguments. For example:

```
[ '1', '2', '3' ].map(parseInt)
//=> [1, NaN, NaN]
```

This doesn’t work because `parseInt` is defined as `parseInt(string[, radix])`. It takes an optional `radix` argument. And when you call `parseInt` with `map`, the index is interpreted as a radix. Not good! What we want is to convert `parseInt` into a function taking only one argument.

We could write `['1', '2', '3'].map((s) => parseInt(s))`, or we could come up with a decorator to do the job for us:

```
const unary = (fn) =>
  fn.length === 1
    ? fn
    : function (something) {
      return fn.call(this, something)
    }
```

And now we can write:

```
['1', '2', '3'].map(unary(parseInt))
//=> [1, 2, 3]
```

Presto!

Tap

One of the most basic combinators is the “K Combinator,” nicknamed the “Kestrel:”

```
const K = (x) => (y) => x;
```

It has some surprising applications. One is when you want to do something with a value for side-effects, but keep the value around. Behold:

```
const tap = (value) =>
  (fn) => (
    typeof(fn) === 'function' && fn(value),
    value
  )
```

tap is a traditional name borrowed from various Unix shell commands. It takes a value and returns a function that always returns the value, but if you pass it a function, it executes the function for side-effects. Let’s see it in action as a poor-man’s debugger:

```
tap('espresso')((it) => {
  console.log(`Our drink is '${it}'`)
});
//=> Our drink is 'espresso'
'espresso'
```

It’s easy to turn off:

```
tap('espresso')();
//=> 'espresso'
```

Libraries like [Underscore](#)⁴⁸ use a version of tap that is “uncurried:”

```
_.tap('espresso', (it) =>
  console.log(`Our drink is '${it}'`)
);
//=> Our drink is 'espresso'
'espresso'
```

Let’s enhance our recipe so that it works both ways:

⁴⁸<http://underscorejs.org>

```

const tap = (value, fn) => {
  const curried = (fn) => (
    typeof(fn) === 'function' && fn(value),
    value
  );

  return fn === undefined
    ? curried
    : curried(fn);
}

```

Now we can write:

```

tap('espresso')((it) => {
  console.log(`Our drink is '${it}'`)
});
//=> Our drink is 'espresso'
'espresso'

```

Or:

```

tap('espresso', (it) => {
  console.log(`Our drink is '${it}'`)
});
//=> Our drink is 'espresso'
'espresso'

```

And if we wish it to do nothing at all, We can write either `tap('espresso')()` or `tap('espresso', null)`

p.s. `tap` can do more than just act as a debugging aid. It's also useful for working with `Object` and `instance` methods.

Maybe

A common problem in programming is checking for `null` or `undefined` (hereafter called “nothing,” while all other values including `0`, `[]` and `false` will be called “something”). Languages like JavaScript do not strongly enforce the notion that a particular variable or particular property be something, so programs are often written to account for values that may be nothing.

This recipe concerns a pattern that is very common: A function `fn` takes a value as a parameter, and its behaviour by design is to do nothing if the parameter is nothing:

```
const isSomething = (value) =>
  value !== null && value !== void 0;

const checksForSomething = (value) => {
  if (isSomething(value)) {
    // function's true logic
  }
}
```

Alternately, the function may be intended to work with any value, but the code calling the function wishes to emulate the behaviour of doing nothing by design when given nothing:

```
var something =
  isSomething(value)
  ? doesntCheckForSomething(value)
  : value;
```

Naturally, there’s a function decorator recipe for that, borrowed from Haskell’s `maybe monad`⁴⁹, Ruby’s `andand`⁵⁰, and CoffeeScript’s existential method invocation:

```
const maybe = (fn) =>
  function (...args) {
    if (args.length === 0) {
      return
    }
    else {
      for (let arg of args) {
        if (arg == null) return;
      }
    }
  }
}
```

⁴⁹[https://en.wikipedia.org/wiki/Monad_\(functional_programming\)#The_Maybe_monad](https://en.wikipedia.org/wiki/Monad_(functional_programming)#The_Maybe_monad)

⁵⁰<https://github.com/raganwald/andand>

```
    return fn.apply(this, args)
  }
}
```

maybe reduces the logic of checking for nothing to a function call:

```
maybe((a, b, c) => a + b + c)(1, 2, 3)
//=> 6
```

```
maybe((a, b, c) => a + b + c)(1, null, 3)
//=> undefined
```

As a bonus, maybe plays very nicely with instance methods, we'll discuss those [later](#):

```
function Model () {};

Model.prototype.setSomething = maybe(function (value) {
  this.something = value;
});
```

If some code ever tries to call `model.setSomething` with nothing, the operation will be skipped.

Once

once is an extremely helpful combinator. It ensures that a function can only be called, well, *once*. Here's the recipe:

```
const once = (fn) => {
  let done = false;

  return function () {
    return done ? void 0 : ((done = true), fn.apply(this, arguments))
  }
}
```

Very simple! You pass it a function, and you get a function back. That function will call your function once, and thereafter will return `undefined` whenever it is called. Let's try it:

```
const askedOnBlindDate = once(
  () => "sure, why not?"
);

askedOnBlindDate()
//=> 'sure, why not?'

askedOnBlindDate()
//=> undefined

askedOnBlindDate()
//=> undefined
```

It seems some people will only try blind dating once.

(Note: There are some subtleties with decorators like `once` that involve the intersection of state with methods. We'll look at that again in [stateful method decorators](#).)

Left-Variadic Functions

A *variadic function* is a function that is designed to accept a variable number of arguments.⁵¹ In JavaScript, you can make a variadic function by gathering parameters. For example:

```
const abccc = (a, b, ...c) => {
  console.log(a);
  console.log(b);
  console.log(c);
};

abccc(1, 2, 3, 4, 5)
1
2
[3,4,5]
```

This can be useful when writing certain kinds of destructuring algorithms. For example, we might want to have a function that builds some kind of team record. It accepts a coach, a captain, and an arbitrary number of players. Easy in ECMAScript 2015:

```
function team(coach, captain, ...players) {
  console.log(` ${captain} (captain)`);
  for (let player of players) {
    console.log(player);
  }
  console.log(` squad coached by ${coach}`);
}

team('Luis Enrique', 'Xavi Hernández', 'Marc-André ter Stegen',
      'Martín Montoya', 'Gerard Piqué')
//=>
Xavi Hernández (captain)
Marc-André ter Stegen
Martín Montoya
Gerard Piqué
squad coached by Luis Enrique
```

But we can't go the other way around:

⁵¹English is about as inconsistent as JavaScript: Functions with a fixed number of arguments can be unary, binary, ternary, and so forth. But can they be “variairy?” No! They have to be “variadic.”

```
function team2(...players, captain, coach) {
  console.log(` ${captain} (captain)`);
  for (let player of players) {
    console.log(player);
  }
  console.log(` squad coached by ${coach}`);
}
//=> Unexpected token
```

ECMAScript 2015 only permits gathering parameters from the *end* of the parameter list. Not the beginning. What to do?

a history lesson

In “Ye Olde Days,”⁵² JavaScript could not gather parameters, and we had to either do backflips with arguments and .slice, or we wrote ourselves a variadic decorator that could gather arguments into the last declared parameter. Here it is in all of its ECMAScript-5 glory:

```
var __slice = Array.prototype.slice;

function rightVariadic (fn) {
  if (fn.length < 1) return fn;

  return function () {
    var ordinaryArgs = (1 <= arguments.length ?
      __slice.call(arguments, 0, fn.length - 1) : []),
      restOfTheArgsList = __slice.call(arguments, fn.length - 1),
      args = (fn.length <= arguments.length ?
        ordinaryArgs.concat([restOfTheArgsList]) : []);

    return fn.apply(this, args);
  }
};

var firstAndButFirst = rightVariadic(function test (first, butFirst) {
  return [first, butFirst]
});

firstAndButFirst('why', 'hello', 'there', 'little', 'droid')
//=> ["why", ["hello", "there", "little", "droid"]]
```

We don't need rightVariadic any more, because instead of:

⁵²Another history lesson. “Ye” in “Ye Olde,” was not actually spelled with a “Y” in days of old, it was spelled with a *thorn*, and is pronounced “the.” Another word, “Ye” in “Ye of little programming faith,” is pronounced “ye,” but it’s a different word altogether.

```
var firstAndButFirst = rightVariadic(
  function test (first, butFirst) {
    return [first, butFirst]
  });

```

We now simply write:

```
const firstAndButFirst = (first, ...butFirst) =>
  [first, butFirst];

```

This is a *right-variadic function*, meaning that it has one or more fixed arguments, and the rest are gathered into the rightmost argument.

overcoming limitations

It's nice to have progress. But as noted above, we can't write:

```
const butLastAndLast = (...butLast, last) =>
  [butLast, last];

```

That's a *left-variadic function*. All left-variadic functions have one or more fixed arguments, and the rest are gathered into the leftmost argument. JavaScript doesn't do this. But if we wanted to write left-variadic functions, could we make ourselves a `leftVariadic` decorator to turn a function with one or more arguments into a left-variadic function?

We sure can, by using the techniques from `rightVariadic`. Mind you, we can take advantage of modern JavaScript to simplify the code:

```
const leftVariadic = (fn) => {
  if (fn.length < 1) {
    return fn;
  }
  else {
    return function (...args) {
      const gathered = args.slice(0, args.length - fn.length + 1),
            spread   = args.slice(args.length - fn.length + 1);

      return fn.apply(
        this, [gathered].concat(spread)
      );
    }
  }
}

```

```

    }
};

const butLastAndLast = leftVariadic((butLast, last) => [butLast, last]);

butLastAndLast('why', 'hello', 'there', 'little', 'droid')
//=> [[ "why", "hello", "there", "little"], "droid"]

```

Our `leftVariadic` function is a decorator that turns any function into a function that gathers parameters *from the left*, instead of from the right.

left-variadic destructuring

Gathering arguments for functions is one of the ways JavaScript can *destructure* arrays. Another way is when assigning variables, like this:

```

const [first, ...butFirst] = ['why', 'hello', 'there', 'little', 'droid'];

first
//=> 'why'
butFirst
//=> ["hello", "there", "little", "droid"]

```

As with parameters, we can't gather values from the left when destructuring an array:

```

const [...butLast, last] = ['why', 'hello', 'there', 'little', 'droid'];
//=> Unexpected token

```

We could use `leftVariadic` the hard way:

```

const [butLast, last] = leftVariadic((butLast, last) => [butLast, last])(...[ 'wh\
y', 'hello', 'there', 'little', 'droid']);

butLast
//=> ['why', 'hello', 'there', 'little']

last
//=> 'droid'

```

But we can write our own left-gathering function utility using the same principles without all the tedium:

```
const leftGather = (outputArrayLength) => {
  return function (inputArray) {
    return [inputArray.slice(0, inputArray.length - outputArrayLength + 1)].conc\
at(
  inputArray.slice(inputArray.length - outputArrayLength + 1)
)
}
};

const [butLast, last] = leftGather(2)(['why', 'hello', 'there', 'little', 'droid'])

butLast
//=> ['why', 'hello', 'there', 'little']

last
//=> 'droid'
```

With `leftGather`, we have to supply the length of the array we wish to use as the result, and it gathers excess arguments into it from the left, just like `leftVariadic` gathers excess parameters for a function.

Compose and Pipeline

Here is the B Combinator, or compose that we saw in [Combinators and Decorators](#):

```
const compose = (a, b) =>
  (c) => a(b(c))
```

As we saw before, given:

```
const addOne = (number) => number + 1;

const doubleOf = (number) => number * 2;
```

Instead of:

```
const doubleOfAddOne = (number) => doubleOf(addOne(number));
```

We could write:

```
const doubleOfAddOne = compose(doubleOf, addOne);
```

variadic compose and recursion

If we wanted to implement a compose3, we could write:

```
const compose3 = (a, b, c) => (d) =>
  a(b(c(d)))
```

Or observe that it is really:

```
const compose3 = (a, b, c) => compose(a(compose(b, c)))
```

Once we get to compose4, we ask ourselves if there is a better way. For example, if we had a *variadic* compose, we could write compose(a, b), compose(a, b, c), or compose(a, b, c, d).

We can implement a variadic compose recursively. The easiest way to reason about writing a recursive compose is to start with the smallest or *degenerate* case. If compose only took one argument, it would look like this:

```
const compose = (a) => a
```

The next thing is to have a way of breaking a piece off the problem. We can do this with a variadic function:

```
const compose = (a, ...rest) =>
  "to be determined"
```

We can test whether we have the degenerate case:

```
const compose = (a, ...rest) =>
  rest.length === 0
    ? a
    : "to be determined"
```

If it is not the degenerate case, we need to combine what we have with the solution for the rest. In other words, we need to combine `fn` with `compose(...rest)`. How do we do that? Well, consider `compose(a, b)`. We know that `compose(b)` is the degenerate case, it's just `b`. And we know that `compose(a, b)` is `(c) => a(b(c))`.

So let's substitute `compose(b)` for `b`:

```
compose(a, compose(b)) === (c) => a(compose(b)(c))
```

Now substitute `...rest` for `b`:

```
compose(a, ...rest) === (c) => a(compose(...rest)(c))
```

This is our solution:

```
const compose = (a, ...rest) =>
  rest.length === 0
    ? a
    : (c) => a(compose(...rest)(c))
```

There are others, of course. `compose` can be implemented with iteration or with `.reduce`, like this:

```
const compose = (...fns) =>
  (value) =>
    fns.reverse().reduce((acc, fn) => fn(acc), value);
```

But the principle behaviour is the same: To compose a series of functions together, creating a new one. And the value is the same: We can write smaller, single purpose functions and put them together in different ways.

the semantics of compose

With `compose`, we're usually making a new function. Although it works perfectly well, we don't need to write things like `compose(double, addOne)(3)` inline to get the result 8. It's easier and clearer to write `double(addOne(3))`.

On the other hand, when working with something like method decorators, it can help to write:

```
const setter = compose(fluent, maybe);

// ...

SomeClass.prototype.setUser = setter(function (user) {
  this.user = user;
});

SomeClass.prototype.setPrivileges = setter(function (privileges) {
  this.privileges = privileges;
});
```

This makes it clear that `setter` adds the behaviour of both `fluent` and `maybe` to each method it decorates, and it's sometimes easier to read `const setter = compose(fluent, maybe);` than:

```
const setter = (fn) => fluent(maybe(fn));
```

The take-away is that `compose` is helpful when we are defining a new function that combines the effects of existing functions.

pipeline

`compose` is extremely handy, but one thing it doesn't communicate well is the order on operations. `compose` is written that way because it matches the way explicitly composing functions works in JavaScript and most other languages: When you write `a(b(...))`, `a` happens after `b`.

Sometimes it makes more sense to compose functions in data flow order, as in “The value flows through `a` and then through `b`.” For this, we can use the `pipeline` function:

```
const pipeline = (...fns) =>
  (value) =>
    fns.reduce((acc, fn) => fn(acc), value);

const setter = pipeline(addOne, double);
```

Comparing `pipeline` to `compose`, `pipeline` says “add one to the number and then double it.” `Compose` says, “double the result of adding one to the number.” Both do the same job, but communicate their intention in opposite ways.



Saltspring Island Roasting Facility

Picking the Bean: Choice and Truthiness



Decaf and the Antidote

We've seen operators that act on numeric values, like + and %. In addition to numbers, we often need to represent a much more basic idea of truth or falsehood. Is this array empty? Does this person have a middle name? Is this user logged in?

JavaScript does have "boolean" values, they're written `true` and `false`:

```
true
//=> true

false
//=> false
```

`true` and `false` are value types. All values of `true` are === all other values of `true`. We can see that is the case by looking at some operators we can perform on boolean values, `!`, `&&`, and `||`. To begin with, `!` is a unary prefix operator that negates its argument. So:

```
!true
//=> false

!false
//=> true
```

The `&&` and `||` operators are binary infix operators that perform “logical and” and “logical or” respectively:

```
false && false //=> false
false && true //=> false
true && false //=> false
true && true //=> true

false || false //=> false
false || true //=> true
true || false //=> true
true || true //=> true
```

Now, note well: We have said what happens if you pass boolean values to `!`, `&&`, and `||`, but we’ve said nothing about expressions or about passing other values. We’ll look at those presently.

truthiness and the ternary operator

In JavaScript, there is a notion of “truthiness.” Every value is either “truthy” or “falsy.” Obviously, `false` is falsy. So are `null` and `undefined`, values that semantically represent “no value.” `Nan` is falsy, a value representing the result of a calculation that is not a number.⁵³ And there are more: `0` is falsy, a value representing “none of something.” The empty string, `''` is falsy, a value representing having no characters.

Every other value in JavaScript is “truthy” except the aforementioned `false`, `null`, `undefined`, `Nan`, `0`, and `''`. (Many other languages that have a notion of truthiness consider zero and the empty string to be truthy, not falsy, so beware of blindly transliterating code from one language to another!)

The reason why truthiness matters is that the various logical operators (as well as the `if` statement) actually operate on *truthiness*, not on boolean values. This affects the way the `!`, `&&`, and `||` operators work. We’ll look at them in a moment, but first, we’ll look at one more operator.

JavaScript inherited an operator from the C family of languages, the *ternary* operator. It’s the only operator that takes *three* arguments. It looks like this: `first ? second : third`. It evaluates `first`,

⁵³We will not discuss JavaScript’s numeric behaviour in much depth in this book, but the most important thing to know is that it implements the IEEE Standard for Floating-Point Arithmetic (IEEE 754), a technical standard for floating-point computation established in 1985 by the Institute of Electrical and Electronics Engineers (IEEE).

and if `first` is “truthy”, it evaluates `second` and that is its value. If `first` is not truthy, it evaluates `third` and that is its value.

This is a lot like the `if` statement, however it is an *expression*, not a statement, and that can be very valuable. It also doesn’t introduce braces, and that can be a help or a hindrance if we want to introduce a new scope or use statements.

Here’re some simple examples of the ternary operator:

```
true ? 'Hello' : 'Good bye'
//=> 'Hello'

0 ? 'Hello' : 'Good bye'
//=> 'Good bye'

[1, 2, 3, 4, 5].length === 5 ? 'Pentatonic' : 'Quasimodal'
//=> 'Pentatonic'
```

The fact that either the second or the third (but not both) expressions are evaluated can have important repercussions. Consider this hypothetical example:

```
const status = isAuthorized(currentUser) ? deleteRecord(currentRecord) : 'Forbidden';
```

We certainly don’t want JavaScript trying to evaluate `deleteRecord(currentRecord)` unless `isAuthorized(currentUser)` returns `true`.

truthiness and operators

Our logical operators `!`, `&&`, and `||` are a little more subtle than our examples above implied. `!` is the simplest. It always returns `false` if its argument is truthy, and `true` if its argument is not truthy:

```
!5
//=> false

!undefined
//=> true
```

Programmers often take advantage of this behaviour to observe that `!!(someExpression)` will always evaluate to `true` if `someExpression` is truthy, and to `false` if it is not. So in JavaScript (and other languages with similar semantics), when you see something like `!!currentUser()`, this

is an idiom that means “true if currentUser is truthy.” Thus, a function like currentUser() is free to return null, or undefined, or false if there is no current user.

Thus, !! is the way we write “is truthy” in JavaScript. How about && and ||? What haven’t we discussed?

First, and unlike !, && and || do not necessarily evaluate to true or false. To be precise:

- && evaluates its left-hand expression.
 - If its left-hand expression evaluates to something falsy, && returns the value of its left-hand expression without evaluating its right-hand expression.
 - If its left-hand expression evaluates to something truthy, && evaluates its right-hand expression and returns the value of the right-hand expression.
- || evaluates its left-hand expression.
 - If its left-hand expression evaluates to something truthy, || returns the value of its left-hand expression without evaluating its right-hand expression.
 - If its left-hand expression evaluates to something false, || evaluates its right-hand expression and returns the value of the right-hand expression.

If we look at our examples above, we see that when we pass true and false to && and ||, we do indeed get true or false as a result. But when we pass other values, we no longer get true or false:

```
1 || 2
//=> 1

null && undefined
//=> null

undefined && null
//=> undefined
```

In JavaScript, && and || aren’t boolean logical operators in the logical sense. They don’t operate strictly on logical values, and they don’t commute: a || b is not always equal to b || a, and the same goes for &&.

This is not a subtle distinction.

|| and && are control-flow operators

We’ve seen the ternary operator: It is a *control-flow* operator, not a logical operator. The same is true of && and ||. Consider this tail-recursive function that determines whether a positive integer is even:

For example:

```
const even = (n) =>
  n === 0 || (n !== 1 && even(n - 2))

even(42)
//=> true
```

If `n === 0`, JavaScript does not evaluate `(n !== 1 && even(n - 2))`. This is very important! Imagine that JavaScript evaluated both sides of the `||` operator before determining its value. `n === 0` would be true. What about `(n !== 1 && even(n - 2))`? Well, it would evaluate `even(n - 2)`, or `even(-2)`.

This leads us to evaluate `n === 0 || (n !== 1 && even(n - 2))` all over again, and this time we end up evaluating `even(-4)`. And then `even(-6)`. and so on and so forth until JavaScript throws up its hands and runs out of stack space.

But that's not what happens. `||` and `&&` have *short-cut semantics*. In this case, if `n === 0`, JavaScript does not evaluate `(n !== 1 && even(n - 2))`. Likewise, if `n === 1`, JavaScript evaluates `n !== 1 && even(n - 2)` as `false` without ever evaluating `even(n - 2)`.

This is more than just an optimization. It's best to think of `||` and `&&` as control-flow operators. The expression on the left is always evaluated, and its value determines whether the expression on the right is evaluated or not.

function parameters are eager

In contrast to the behaviour of the ternary operator, `||`, and `&&`, function parameters are always *eagerly evaluated*:

```
const or = (a, b) => a || b

const and = (a, b) => a && b

const even = (n) =>
  or(n === 0, and(n !== 1, even(n - 2)))

even(42)
//=> Maximum call stack size exceeded.
```

Now our expression `or(n === 0, and(n !== 1, even(n - 2)))` is calling functions, and JavaScript always evaluates the expressions for parameters before passing the values to a function to invoke. This leads to the infinite recursion we fear.

If we need to have functions with control-flow semantics, we can pass anonymous functions. We obviously don't need anything like this for `or` and `and`, but to demonstrate the technique:

```
const or = (a, b) => a() || b()

const and = (a, b) => a() && b()

const even = (n) =>
  or(() => n === 0, () => and(() => n !== 1, () => even(n - 2)))

even(7)
//=> false
```

Here we've passed functions that contain the expressions we want to evaluate, and now we can write our own functions that can delay evaluation.

summary

- Logical operators are based on truthiness and falsiness, not the strict values `true` and `false`.
- `!` is a logical operator, it always returns `true` or `false`.
- The ternary operator `(?:)`, `||`, and `&&` are control flow operators, they do not always return `true` or `false`, and they have short-cut semantics.
- Function invocation uses eager evaluation, so if we need to roll our own control-flow semantics, we pass it functions, not expressions.

Composing and Decomposing Data



Stacked Cups

Recursion is the root of computation since it trades description for time.—Alan Perlis,
Epigrams in Programming⁵⁴

⁵⁴<http://www.cs.yale.edu/homes/perlis-alan/quotes.html>

Arrays and Destructuring Arguments

While we have mentioned arrays briefly, we haven't had a close look at them. Arrays are JavaScript's "native" representation of lists. Strings are important because they represent writing. Lists are important because they represent ordered collections of things, and ordered collections are a fundamental abstraction for making sense of reality.

array literals

JavaScript has a literal syntax for creating an array: The [and] characters. We can create an empty array:

```
[]  
//=> []
```

We can create an array with one or more *elements* by placing them between the brackets and separating the items with commas. Whitespace is optional:

```
[1]  
//=> [1]  
  
[2, 3, 4]  
//=> [2,3,4]
```

Any expression will work:

```
[ 2,  
  3,  
  2 + 2  
]  
//=> [2,3,4]
```

Including an expression denoting another array:

```
[[[[[[]]]]]
```

This is an array with one element that is an empty array. Although that seems like something nobody would ever construct, many students have worked with almost the exact same thing when they explored various means of constructing arithmetic from Set Theory.

Any expression will do, including names:

```
const wrap = (something) => [something];

wrap("lunch")
//=> ["lunch"]
```

Array literals are expressions, and arrays are *reference types*. We can see that each time an array literal is evaluated, we get a new, distinct array, even if it contains the exact same elements:

```
[] === []
//=> false

[2 + 2] === [2 + 2]
//=> false

const array_of_one = () => [1];

array_of_one() === array_of_one()
//=> false
```

element references

Array elements can be extracted using [and] as postfix operators. We pass an integer as an index of the element to extract:

```
const oneTwoThree = ["one", "two", "three"];

oneTwoThree[0]
//=> 'one'

oneTwoThree[1]
//=> 'two'

oneTwoThree[2]
//=> 'three'
```

As we can see, JavaScript Arrays are [zero-based⁵⁵](#).

We know that every array is its own unique entity, with its own unique reference. What about the contents of an array? Does it store references to the things we give it? Or copies of some kind?

⁵⁵https://en.wikipedia.org/wiki/Zero-based_numbering

```
const x = [],
  a = [x];

a[0] === x
//=> true, arrays store references to the things you put in them.
```

destructuring arrays

There is another way to extract elements from arrays: *Destructuring*, a feature going back to Common Lisp, if not before. We saw how to construct an array literal using [, expressions, , and]. Here's an example of an array literal that uses a name:

```
const wrap = (something) => [something];
```

Let's expand it to use a block and an extra name:

```
const wrap = (something) => {
  const wrapped = [something];

  return wrapped;
}

wrap("package")
//=> ["package"]
```

The line `const wrapped = [something];` is interesting. On the left hand is a name to be bound, and on the right hand is an array literal, a template for constructing an array, very much like a quasi-literal string.

In JavaScript, we can actually *reverse* the statement and place the template on the left and a value on the right:

```
const unwrap = (wrapped) => {
  const [something] = wrapped;

  return something;
}

unwrap(["present"])
//=> "present"
```

The statement `const [something] = wrapped;` *destructures* the array represented by `wrapped`, binding the value of its single element to the name `something`. We can do the same thing with more than one element:

```
const surname = (name) => {
  const [first, last] = name;

  return last;
}

surname["Reginald", "Braithwaite"])
//=> "Braithwaite"
```

We could do the same thing with `(name) => name[1]`, but destructuring is code that resembles the data it consumes, a valuable coding style.

Destructuring can nest:

```
const description = (nameAndOccupation) => {
  const [[first, last], occupation] = nameAndOccupation;

  return `${first} is a ${occupation}`;
}

description([["Reginald", "Braithwaite"], "programmer"])
//=> "Reginald is a programmer"
```

gathering

Sometimes we need to extract arrays from arrays. Here is the most common pattern: Extracting the head and gathering everything but the head from an array:

```
const [car, ...cdr] = [1, 2, 3, 4, 5];

car
//=> 1
cdr
//=> [2, 3, 4, 5]
```

`car` and `cdr`⁵⁶ are archaic terms that go back to an implementation of Lisp running on the IBM 704 computer. Some other languages call them `first` and `butFirst`, or `head` and `tail`. We will use a common convention and call variables we gather `rest`, but refer to the `...` operation as a “gather,” following Kyle Simpson’s example.⁵⁷

Alas, the `...` notation does not provide a universal pattern-matching capability. For example, we cannot write

⁵⁶https://en.wikipedia.org/wiki/CAR_and_CDR

⁵⁷Kyle Simpson is the author of `You Don’t Know JS`, available [here](#)

```
const [...butLast, last] = [1, 2, 3, 4, 5];
//=> ERROR
```

```
const [first, ..., last] = [1, 2, 3, 4, 5];
//=> ERROR
```

Now, when we introduced destructuring, we saw that it is kind-of-sort-of the reverse of array literals. So if

```
const wrapped = [something];
```

Then:

```
const unwrapped = something;
```

What is the reverse of gathering? We know that:

```
const [car, ...cdr] = [1, 2, 3, 4, 5];
```

What is the reverse? It would be:

```
const cons = [car, ...cdr];
```

Let's try it:

```
const oneTwoThree = ["one", "two", "three"];
```

```
["zero", ...oneTwoThree]
//=> ["zero", "one", "two", "three"]
```

It works! We can use `...` to place the elements of an array inside another array. We say that using `...` to destructure is gathering, and using it in a literal to insert elements is called “spreading.”

destructuring is not pattern matching

Some other languages have something called *pattern matching*, where you can write something like a destructuring assignment, and the language decides whether the “patterns” matches at all. If it does, assignments are made where appropriate.

In such a language, if you wrote something like:

```
const [what] = [];
```

That match would fail because the array doesn't have an element to assign to `what`. But this is not how JavaScript works. JavaScript tries its best to assign things, and if there isn't something that fits, JavaScript binds `undefined` to the name. Therefore:

```
const [what] = [];  
  
what  
//=> undefined  
  
const [which, what, who] = ["duck feet", "tiger tail"];  
  
who  
//=> undefined
```

And if there aren't any items to assign with `...`, JavaScript assigns an empty array:

```
const [...they] = [];  
  
they  
//=> []  
  
const [which, what, ...they] = ["duck feet", "tiger tail"];  
  
they  
//=> []
```

From its very inception, JavaScript has striven to avoid catastrophic errors. As a result, it often coerces values, passes `undefined` around, or does whatever it can to keep executing without failing. This often means that we must write our own code to detect failure conditions, as we cannot rely on the language to point out when we are doing semantically meaningless things.

destructuring and return values

Some languages support multiple return values: A function can return several things at once, like a value and an error code. This can easily be emulated in JavaScript with destructuring:

```

const description = (nameAndOccupation) => {
  if (nameAndOccupation.length < 2) {
    return ["", "occupation missing"]
  }
  else {
    const [[first, last], occupation] = nameAndOccupation;

    return [`${first} is a ${occupation}`, "ok"];
  }
}

const [reg, status] = description([["Reginald", "Braithwaite"], "programmer"]);

reg
//=> "Reginald is a programmer"

status
//=> "ok"

```

destructuring parameters

Consider the way we pass arguments to parameters:

```

foo()
bar("smaug")
baz(1, 2, 3)

```

It is very much like an array literal. And consider how we bind values to parameter names:

```

const foo = () => ...
const bar = (name) => ...
const baz = (a, b, c) => ...

```

It *looks* like destructuring. It acts like destructuring. There is only one difference: We have not tried gathering. Let's do that:

```
const numbers = (...nums) => nums;  
  
numbers(1, 2, 3, 4, 5)  
//=> [1,2,3,4,5]  
  
const headAndTail = (head, ...tail) => [head, tail];  
  
headAndTail(1, 2, 3, 4, 5)  
//=> [1, [2,3,4,5]]
```

Gathering works with parameters! This is very useful indeed, and we'll see more of it in a moment.⁵⁸

⁵⁸Gathering in parameters has a long history, and the usual terms are to call gathering “pattern matching” and to call a name that is bound to gathered values a “rest parameter.” The term “rest” is perfectly compatible with gather: “Rest” is the noun, and “gather” is the verb. We *gather* the *rest* of the parameters.

Self-Similarity

Recursion is the root of computation since it trades description for time.—Alan Perlis,
*Epigrams in Programming*⁵⁹

In [Arrays and Destructuring Arguments](#), we worked with the basic idea that putting an array together with a literal array expression was the reverse or opposite of taking it apart with a destructuring assignment.

We saw that the basic idea that putting an array together with a literal array expression was the reverse or opposite of taking it apart with a destructuring assignment.

Let's be more specific. Some data structures, like lists, can obviously be seen as a collection of items. Some are empty, some have three items, some forty-two, some contain numbers, some contain strings, some a mixture of elements, there are all kinds of lists.

But we can also define a list by describing a rule for building lists. One of the simplest, and longest-standing in computer science, is to say that a list is:

1. Empty, or;
2. Consists of an element concatenated with a list .

Let's convert our rules to array literals. The first rule is simple: [] is a list. How about the second rule? We can express that using a spread. Given an element e and a list list, [e, ...list] is a list. We can test this manually by building up a list:

```
[]
//=> []

["baz", ...[]]
//=> ["baz"]

["bar", ...["baz"]]
//=> ["bar", "baz"]

["foo", ...["bar", "baz"]]
//=> ["foo", "bar", "baz"]
```

Thanks to the parallel between array literals + spreads with destructuring + rests, we can also use the same rules to decompose lists:

⁵⁹<http://www.cs.yale.edu/homes/perlis-alan/quotes.html>

```

const [first, ...rest] = [];
first
  //=> undefined
rest
  //=> []

const [first, ...rest] = ["foo"];
first
  //=> "foo"
rest
  //=> []

const [first, ...rest] = ["foo", "bar"];
first
  //=> "foo"
rest
  //=> ["bar"]

const [first, ...rest] = ["foo", "bar", "baz"];
first
  //=> "foo"
rest
  //=> ["bar", "baz"]

```

For the purpose of this exploration, we will presume the following:⁶⁰

```

const isEmpty = ([first, ...rest]) => first === undefined;

isEmpty([])
  //=> true

isEmpty([0])
  //=> false

isEmpty([[]])
  //=> false

```

Armed with our definition of an empty list and with what we've already learned, we can build a great many functions that operate on arrays. We know that we can get the length of an array using

⁶⁰Well, *actually*, the difference between prototypes and classes is like the difference between model homes and blueprints. But prototypes are not like model homes. In actual fact, the relationship between an object and its prototype is one of *delegation*. So if a model home had a kitchen, and you asked the builder to make you a home using the model as a prototype, you could customize your own kitchen. But if you didn't want to have your own custom kitchen, you would just use the model home's kitchen to do all your own cooking. The relationship between a model home and a house is sometimes described as [concatenative inheritance](#), and JavaScript lets you do that too.

its `.length`. But as an exercise, how would we write a `length` function using just what we have already?

First, we pick what we call a *terminal case*. What is the length of an empty array? 0 . So let's start our function with the observation that if an array is empty, the length is 0 :

```
const length = ([first, ...rest]) =>
  first === undefined
    ? 0
    : // ???
```

We need something for when the array isn't empty. If an array is not empty, and we break it into two pieces, `first` and `rest`, the length of our array is going to be `length(first) + length(rest)`. Well, the length of `first` is 1 , there's just one element at the front. But we don't know the length of `rest`. If only there was a function we could call... Like `length`!

```
const length = ([first, ...rest]) =>
  first === undefined
    ? 0
    : 1 + length(rest);
```

Let's try it!

```
length([])
//=> 0

length(["foo"])
//=> 1

length(["foo", "bar", "baz"])
//=> 3
```

Our `length` function is *recursive*, it calls itself. This makes sense because our definition of a list is recursive, and if a list is self-similar, it is natural to create an algorithm that is also self-similar.

linear recursion

“Recursion” sometimes seems like an elaborate party trick. There's even a joke about this:

When promising students are trying to choose between pure mathematics and applied engineering, they are given a two-part aptitude test. In the first part, they are led to a laboratory bench and told to follow the instructions printed on the card. They find a bunsen burner, a sparkler, a tap, an empty beaker, a stand, and a card with the instructions “boil water.”

Of course, all the students know what to do: They fill the beaker with water, place the stand on the burner and the beaker on the stand, then they turn the burner on and use the sparkler to ignite the flame. After a bit the water boils, and they turn off the burner and are lead to a second bench.

Once again, there is a card that reads, “boil water.” But this time, the beaker is on the stand over the burner, as left behind by the previous student. The engineers light the burner immediately. Whereas the mathematicians take the beaker off the stand and empty it, thus reducing the situation to a problem they have already solved.

There is more to recursive solutions than simply functions that invoke themselves. Recursive algorithms follow the “divide and conquer” strategy for solving a problem:

1. Divide the problem into smaller problems
2. If a smaller problem is solvable, solve the small problem
3. If a smaller problem is not solvable, divide and conquer that problem
4. When all small problems have been solved, compose the solutions into one big solution

The big elements of divide and conquer are a method for decomposing a problem into smaller problems, a test for the smallest possible problem, and a means of putting the pieces back together. Our solutions are a little simpler in that we don’t really break a problem down into multiple pieces, we break a piece off the problem that may or may not be solvable, and solve that before sticking it onto a solution for the rest of the problem.

This simpler form of “divide and conquer” is called *linear recursion*. It’s very useful and simple to understand. Let’s take another example. Sometimes we want to *flatten* an array, that is, an array of arrays needs to be turned into one array of elements that aren’t arrays.⁶¹

We already know how to divide arrays into smaller pieces. How do we decide whether a smaller problem is solvable? We need a test for the terminal case. Happily, there is something along these lines provided for us:

```
Array.isArray("foo")
//=> false

Array.isArray(["foo"])
//=> true
```

The usual “terminal case” will be that flattening an empty array will produce an empty array. The next terminal case is that if an element isn’t an array, we don’t flatten it, and can put it together with the rest of our solution directly. Whereas if an element is an array, we’ll flatten it and put it together with the rest of our solution.

So our first cut at a `flatten` function will look like this:

⁶¹`flatten` is a very simple `unfold`, a function that takes a seed value and turns it into an array. Unfolds can be thought of a “path” through a data structure, and flattening a tree is equivalent to a depth-first traverse.

```
const flatten = ([first, ...rest]) => {
  if (first === undefined) {
    return [];
  }
  else if (!Array.isArray(first)) {
    return [first, ...flatten(rest)];
  }
  else {
    return [...flatten(first), ...flatten(rest)];
  }
}

flatten(["foo", [3, 4, []]])
//=> ["foo", 3, 4]
```

Once again, the solution directly displays the important elements: Dividing a problem into subproblems, detecting terminal cases, solving the terminal cases, and composing a solution from the solved portions.

mapping

Another common problem is applying a function to every element of an array. JavaScript has a built-in function for this, but let's write our own using linear recursion.

If we want to square each number in a list, we could write:

```
const squareAll = ([first, ...rest]) => first === undefined
  ? []
  : [first * first, ...squareAll(rest)]\n];

squareAll([1, 2, 3, 4, 5])
//=> [1, 4, 9, 16, 25]
```

And if we wanted to “truthify” each element in a list, we could write:

```
const truthyAll = ([first, ...rest]) => first === undefined
  ? []
  : [!!first, ...truthyAll(rest)];

truthyAll([null, true, 25, false, "foo"])
//=> [false, true, true, false, true]
```

This specific case of linear recursion is called “mapping,” and it is not necessary to constantly write out the same pattern again and again. Functions can take functions as arguments, so let’s “extract” the thing to do to each element and separate it from the business of taking an array apart, doing the thing, and putting the array back together.

Given the signature:

```
const mapWith = (fn, array) => // ...
```

We can write it out using a ternary operator. Even in this small function, we can identify the terminal condition, the piece being broken off, and recomposing the solution.

```
const mapWith = (fn, [first, ...rest]) =>
  first === undefined
  ? []
  : [fn(first), ...mapWith(fn, rest)];

mapWith((x) => x * x, [1, 2, 3, 4, 5])
//=> [1, 4, 9, 16, 25]

mapWith((x) => !!x, [null, true, 25, false, "foo"])
//=> [false, true, true, false, true]
```

folding

With the exception of the `length` example at the beginning, our examples so far all involve rebuilding a solution using spreads. But they needn’t. A function to compute the sum of the squares of a list of numbers might look like this:

```
const sumSquares = ([first, ...rest]) => first === undefined
  ? 0
  : first * first + sumSquares(rest);

sumSquares([1, 2, 3, 4, 5])
//=> 55
```

There are two differences between `sumSquares` and our maps above:

1. Given the terminal case of an empty list, we return a `0` instead of an empty list, and;
2. We catenate the square of each element to the result of applying `sumSquares` to the rest of the elements.

Let's rewrite `mapWith` so that we can use it to sum squares.

```
const foldWith = (fn, terminalValue, [first, ...rest]) =>
  first === undefined
  ? terminalValue
  : fn(first, foldWith(fn, terminalValue, rest));
```

And now we supply a function that does slightly more than our mapping functions:

```
foldWith((number, rest) => number * number + rest, 0, [1, 2, 3, 4, 5])
//=> 55
```

Our `foldWith` function is a generalization of our `mapWith` function. We can represent a map as a fold, we just need to supply the array rebuilding code:

```
const squareAll = (array) => foldWith((first, rest) => [first * first, ...rest], [],
[], array);

squareAll([1, 2, 3, 4, 5])
//=> [1, 4, 9, 16, 25]
```

And if we like, we can write `mapWith` using `foldWith`:

```
const mapWith = (fn, array) => foldWith((first, rest) => [fn(first), ...rest], [\n], array),\n    squareAll = (array) => mapWith((x) => x * x, array);\n\nsquareAll([1, 2, 3, 4, 5])\n//=> [1, 4, 9, 16, 25]
```

And to return to our first example, our version of `length` can be written as a fold:

```
const length = (array) => foldWith((first, rest) => 1 + rest, 0, array);\n\nlength([1, 2, 3, 4, 5])\n//=> 5
```

summary

Linear recursion is a basic building block of algorithms. Its basic form parallels the way linear data structures like lists are constructed: This helps make it understandable. Its specialized cases of mapping and folding are especially useful and can be used to build other functions. And finally, while folding is a special case of linear recursion, mapping is a special case of folding.

Tail Calls (and Default Arguments)

The `mapWith` and `foldWith` functions we wrote in [Self-Similarity](#) are useful for illustrating the basic principles behind using recursion to work with self-similar data structures, but they are not “production-ready” implementations. One of the reasons they are not production-ready is that they consume memory proportional to the size of the array being folded.

Let’s look at how. Here’s our extremely simple `mapWith` function again:

```
const mapWith = (fn, [first, ...rest]) =>
  first === undefined
    ? []
    : [fn(first), ...mapWith(fn, rest)];

mapWith((x) => x * x, [1, 2, 3, 4, 5])
//=> [1, 4, 9, 16, 25]
```

Let’s step through its execution. First, `mapWith((x) => x * x, [1, 2, 3, 4, 5])` is invoked. `first` is not `undefined`, so it evaluates `[fn(first), ...mapWith(fn, rest)]`. To do that, it has to evaluate `fn(first)` and `mapWith(fn, rest)`, then evaluate `[fn(first), ...mapWith(fn, rest)]`.

This is roughly equivalent to writing:

```
const mapWith = function (fn, [first, ...rest]) {
  if (first === undefined) {
    return [];
  }
  else {
    const _temp1 = fn(first),
      _temp2 = mapWith(fn, rest),
      _temp3 = [_temp1, ..._temp2];

    return _temp3;
  }
}
```

Note that while evaluating `mapWith(fn, rest)`, JavaScript must retain the value `first` or `fn(first)`, plus some housekeeping information so it remembers what to do with `mapWith(fn, rest)` when it has a result. JavaScript cannot throw `first` away. So we know that JavaScript is going to hang on to 1.

Next, JavaScript invokes `mapWith(fn, rest)`, which is semantically equivalent to `mapWith((x) => x * x, [2, 3, 4, 5])`. And the same thing happens: JavaScript has to hang on to 2 (or 4, or both,

depending on the implementation), plus some housekeeping information so it remembers what to do with that value, while it calls the equivalent of `mapWith((x) => x * x, [3, 4, 5])`.

This keeps on happening, so that JavaScript collects the values 1, 2, 3, 4, and 5 plus housekeeping information by the time it calls `mapWith((x) => x * x, [])`. It can start assembling the resulting array and start discarding the information it is saving.

That information is saved on a *call stack*, and it is quite expensive. Furthermore, doubling the length of an array will double the amount of space we need on the stack, plus double all the work required to set up and tear down the housekeeping data for each call (these are called *call frames*, and they include the place where the function was called, an environment, and so on).

In practice, using a method like this with more than about 50 items in an array may cause some implementations to run very slow, run out of memory and freeze, or cause an error.

```
mapWith((x) => x * x, [
  0, 1, 2, 3, 4, 5, 6, 7, 8, 9,
  10, 11, 12, 13, 14, 15, 16, 17, 18, 19,
  20, 21, 22, 23, 24, 25, 26, 27, 28, 29,
  30, 31, 32, 33, 34, 35, 36, 37, 38, 39,
  40, 41, 42, 43, 44, 45, 46, 47, 48, 49,
  50, 51, 52, 53, 54, 55, 56, 57, 58, 59,
  60, 61, 62, 63, 64, 65, 66, 67, 68, 69,
  70, 71, 72, 73, 74, 75, 76, 77, 78, 79,
  80, 81, 82, 83, 84, 85, 86, 87, 88, 89,
  90, 91, 92, 93, 94, 95, 96, 97, 98, 99,
  0, 1, 2, 3, 4, 5, 6, 7, 8, 9,
  10, 11, 12, 13, 14, 15, 16, 17, 18, 19,
  20, 21, 22, 23, 24, 25, 26, 27, 28, 29,
  30, 31, 32, 33, 34, 35, 36, 37, 38, 39,
  40, 41, 42, 43, 44, 45, 46, 47, 48, 49,
  50, 51, 52, 53, 54, 55, 56, 57, 58, 59,
  60, 61, 62, 63, 64, 65, 66, 67, 68, 69,
  70, 71, 72, 73, 74, 75, 76, 77, 78, 79,
  80, 81, 82, 83, 84, 85, 86, 87, 88, 89,
  90, 91, 92, 93, 94, 95, 96, 97, 98, 99
])
//=> ???
```

Is there a better way? Several, in fact, fast algorithms is a very highly studied field of computer science. The one we're going to look at here is called *tail-call optimization*, or “TCO.”

tail-call optimization

A “tail-call” occurs when a function’s last act is to invoke another function, and then return whatever the other function returns. For example, consider the `maybe` function decorator:

```
const maybe = (fn) =>
  function (...args) {
    if (args.length === 0) {
      return;
    }
    else {
      for (let arg of args) {
        if (arg == null) return;
      }
      return fn.apply(this, args);
    }
  }
}
```

There are three places it returns. The first two don’t return anything, they don’t matter. But the third is `fn.apply(this, args)`. This is a tail-call, because it invokes another function and returns its result. This is interesting, because after sorting out what to supply as arguments (`this, args`), JavaScript can throw away everything in its current stack frame. It isn’t going to do any more work, so it can throw its existing stack frame away.

And in fact, it does exactly that: It throws the stack frame away, and does not consume extra memory when making a `maybe`-wrapped call. This is a very important characteristic of JavaScript: **If a function makes a call in tail position, JavaScript optimizes away the function call overhead and stack space.**

That is excellent, but one wrapping is not a big deal. When would we really care? Consider this implementation of `length`:

```
const length = ([first, ...rest]) =>
  first === undefined
    ? 0
    : 1 + length(rest);
```

The `length` function calls itself, but it is not a tail-call, because it returns `1 + length(rest)`, not `length(rest)`.

The problem can be stated in such a way that the answer is obvious: `length` does not call itself in tail position, because it has to do two pieces of work, and while one of them is in the recursive call to `length`, the other happens after the recursive call.

The obvious solution?

converting non-tail-calls to tail-calls

The obvious solution is push the `1 + work` into the call to `length`. Here's our first cut:

```
const lengthDelaysWork = ([first, ...rest], numberToBeAdded) =>
  first === undefined
    ? 0 + numberToBeAdded
    : lengthDelaysWork(rest, 1 + numberToBeAdded)

lengthDelaysWork(["foo", "bar", "baz"], 0)
//=> 3
```

This `lengthDelaysWork` function calls itself in tail position. The `1 + work` is done before calling itself, and by the time it reaches the terminal position, it has the answer. Now that we've seen how it works, we can clean up the `0 + numberToBeAdded` business. But while we're doing that, it's annoying to remember to call it with a zero. Let's fix that:

```
const lengthDelaysWork = ([first, ...rest], numberToBeAdded) =>
  first === undefined
    ? numberToBeAdded
    : lengthDelaysWork(rest, 1 + numberToBeAdded)

const length = (n) =>
  lengthDelaysWork(n, 0);
```

Or we could use partial application:

```
const callLast = (fn, ...args) =>
  (...remainingArgs) =>
    fn(...remainingArgs, ...args);

const length = callLast(lengthDelaysWork, 0);

length(["foo", "bar", "baz"])
//=> 3
```

This version of `length` calls uses `lengthDelaysWork`, and JavaScript optimizes that not to take up memory proportional to the length of the string. We can use this technique with `mapWith`:

```

const mapWithDelaysWork = (fn, [first, ...rest], prepend) =>
  first === undefined
    ? prepend
    : mapWithDelaysWork(fn, rest, [...prepend, fn(first)]);

const mapWith = callLast(mapWithDelaysWork, []);
mapWith((x) => x * x, [1, 2, 3, 4, 5])
  //=> [1, 4, 9, 16, 25]

```

We can use it with ridiculously large arrays:

```

mapWith((x) => x * x, [
  0, 1, 2, 3, 4, 5, 6, 7, 8, 9,
  10, 11, 12, 13, 14, 15, 16, 17, 18, 19,
  20, 21, 22, 23, 24, 25, 26, 27, 28, 29,
  30, 31, 32, 33, 34, 35, 36, 37, 38, 39,
  40, 41, 42, 43, 44, 45, 46, 47, 48, 49,
  50, 51, 52, 53, 54, 55, 56, 57, 58, 59,
  60, 61, 62, 63, 64, 65, 66, 67, 68, 69,
  70, 71, 72, 73, 74, 75, 76, 77, 78, 79,
  80, 81, 82, 83, 84, 85, 86, 87, 88, 89,
  90, 91, 92, 93, 94, 95, 96, 97, 98, 99,
  ...
  2980, 2981, 2982, 2983, 2984, 2985, 2986, 2987, 2988, 2989,
  2990, 2991, 2992, 2993, 2994, 2995, 2996, 2997, 2998, 2999 ])
  //=> [0, 1, 4, 9, 16, 25, 36, 49, 64, 81, 100, 121, 144, 169, 196, ...]

```

Brilliant! We can map over large arrays without incurring all the memory and performance overhead of non-tail-calls. And this basic transformation from a recursive function that does not make a tail call, into a recursive function that calls itself in tail position, is a bread-and-butter pattern for programmers using a language that incorporates tail-call optimization.

factorials

Introductions to recursion often mention calculating factorials:

In mathematics, the factorial of a non-negative integer n , denoted by $n!$, is the product of all positive integers less than or equal to n . For example:

$5! = 5 \times 4 \times 3 \times 2 \times 1 = 120.$

The naïve function for calculating the factorial of a positive integer follows directly from the definition:

```
const factorial = (n) =>
  n === 1
  ? n
  : n * factorial(n - 1);
```

```
factorial(1)
//=> 1
```

```
factorial(5)
//=> 120
```

While this is mathematically elegant, it is computational [filigree](#)⁶².

Once again, it is not tail-recursive, it needs to save the stack with each invocation so that it can take the result returned and compute $n * \text{factorial}(n - 1)$. We can do the same conversion, pass in the work to be done:

```
const factorialWithDelayedWork = (n, work) =>
  n === 1
  ? work
  : factorialWithDelayedWork(n - 1, n * work);

const factorial = (n) =>
  factorialWithDelayedWork(n, 1);
```

Or we could use partial application:

⁶²<https://en.wikipedia.org/wiki/Filigree>

```
const callLast = (fn, ...args) =>
  (...remainingArgs) =>
    fn(...remainingArgs, ...args);

const factorial = callLast(factorialWithDelayedWork, 1);

factorial(1)
//=> 1

factorial(5)
//=> 120
```

As before, we wrote a `factorialWithDelayedWork` function, then used partial application (`callLast`) to make a `factorial` function that took just the one argument and supplied the initial work value.

default arguments

Our problem is that we can directly write:

```
const factorial = (n, work) =>
  n === 1
  ? work
  : factorial(n - 1, n * work);

factorial(1, 1)
//=> 1

factorial(5, 1)
//=> 120
```

But it is hideous to have to always add a `1` parameter, we'd be demanding that everyone using the `factorial` function know that we are using a tail-recursive implementation.

What we really want is this: We want to write something like `factorial(6)`, and have JavaScript automatically know that we really mean `factorial(6, 1)`. But when it calls itself, it will call `factorial(5, 6)` and that will not mean `factorial(5, 1)`.

JavaScript provides this exact syntax, it's called a *default argument*, and it looks like this:

```
const factorial = (n, work = 1) =>
  n === 1
  ? work
  : factorial(n - 1, n * work);

factorial(1)
//=> 1

factorial(6)
//=> 720
```

By writing our parameter list as `(n, work = 1) =>`, we're stating that if a second parameter is not provided, `work` is to be bound to 1. We can do similar things with our other tail-recursive functions:

```
const length = ([first, ...rest], numberToBeAdded = 0) =>
  first === undefined
  ? numberToBeAdded
  : length(rest, 1 + numberToBeAdded)

length(["foo", "bar", "baz"])
//=> 3

const mapWith = (fn, [first, ...rest], prepend = []) =>
  first === undefined
  ? prepend
  : mapWith(fn, rest, [...prepend, fn(first)]);

mapWith((x) => x * x, [1, 2, 3, 4, 5])
//=> [1, 4, 9, 16, 25]
```

Now we don't need to use two functions. A default argument is concise and readable.

defaults and destructuring

We saw earlier that destructuring parameters works the same way as destructuring assignment. Now we learn that we can create a default parameter argument. Can we create a default destructuring assignment?

```
const [first, second = "two"] = ["one"];  
  
`${first} . ${second}`  
//=> "one . two"  
  
const [first, second = "two"] = ["primus", "secundus"];  
  
`${first} . ${second}`  
//=> "primus . secundus"
```

How very useful: defaults can be supplied for destructuring assignments, just like defaults for parameters.

Garbage, Garbage Everywhere



Garbage Day

We have now seen how to use [Tail Calls](#) to execute `mapWith` in constant space:

```
const mapWith = (fn, [first, ...rest], prepend = []) =>
  first === undefined
    ? prepend
    : mapWith(fn, rest, [...prepend, fn(first)]);

mapWith((x) => x * x, [1, 2, 3, 4, 5])
//=> [1, 4, 9, 16, 25]
```

But when we try it on very large arrays, we discover that it is *still* very slow. Much slower than the built-in `.map` method for arrays. The right tool to discover why it's still slow is a memory profiler, but a simple inspection of the program will reveal the following:

Every time we call `mapWith`, we're calling `[...prepend, fn(first)]`. To do that, we take the array in `prepend` and push `fn(first)` onto the end, creating a new array that will be passed to the next invocation of `mapWith`.

Worse, the JavaScript Engine actually copies the elements from `prepend` into the new array one at a time. That is very laborious.⁶³

The array we had in `prepend` is no longer used. In GC environments, it is marked as no longer being used, and eventually the garbage collector recycles the memory it is using. Lather, rinse, repeat: Every time we call `mapWith`, we're creating a new array, copying all the elements from `prepend` into the new array, and then we no longer use `prepend`.

We may not be creating 3,000 stack frames, but we are creating three thousand new arrays and copying elements into each and every one of them. Although the maximum amount of memory does not grow, the thrashing as we create short-lived arrays is very bad, and we do a lot of work copying elements from one array to another.

Key Point: Our `[first, ...rest]` approach to recursion is slow because that it creates a lot of temporary arrays, and it spends an enormous amount of time copying elements into arrays that end up being discarded.

So here's a question: If this is such a slow approach, why do some examples of "functional" algorithms work this exact way?

⁶³It needn't always be so: Programmers have developed specialized data structures that make operations like this cheap, often by arranging for structures to share common elements by default, and only making copies when changes are made. But this is not how JavaScript's built-in arrays work.



The IBM 704

some history

Once upon a time, there was a programming language called [Lisp⁶⁴](#), an acronym for LISt Processing.⁶⁵ Lisp was one of the very first high-level languages, the very first implementation was written for the [IBM 704⁶⁶](#) computer. (The very first FORTRAN implementation was also written for the 704).

The 704 had a 36-bit word, meaning that it was very fast to store and retrieve 36-bit values. The CPU's instruction set featured two important macros: CAR would fetch 15 bits representing the Contents of the Address part of the Register, while CDR would fetch the Contents of the Decrement part of the Register.

⁶⁴https://en.wikipedia.org/wiki/Lisp_

⁶⁵Lisp is still very much alive, and one of the most interesting and exciting programming languages in use today is [Clojure](#), a Lisp dialect that runs on the JVM, along with its sibling [ClojureScript](#), Clojure that transpiles to JavaScript.

⁶⁶https://en.wikipedia.org/wiki/IBM_704

In broad terms, this means that a single 36-bit word could store two separate 15-bit values and it was very fast to save and retrieve pairs of values. If you had two 15-bit values and wished to write them to the register, the CONS macro would take the values and write them to a 36-bit word.

Thus, CONS put two values together, CAR extracted one, and CDR extracted the other. Lisp's basic data type is often said to be the list, but in actuality it was the "cons cell," the term used to describe two 15-bit values stored in one word. The 15-bit values were used as pointers that could refer to a location in memory, so in effect, a cons cell was a little data structure with two pointers to other cons cells.

Lists were represented as linked lists of cons cells, with each cell's head pointing to an element and the tail pointing to another cons cell.

Having these instructions be very fast was important to those early designers: They were working on one of the first high-level languages (COBOL and FORTRAN being the others), and computers in the late 1950s were extremely small and slow by today's standards. Although the 704 used core memory, it still used vacuum tubes for its logic. Thus, the design of programming languages and algorithms was driven by what could be accomplished with limited memory and performance.

Here's the scheme in JavaScript, using two-element arrays to represent cons cells:

```
const cons = (a, d) => [a, d],  
  car  = ([a, d]) => a,  
  cdr  = ([a, d]) => d;
```

We can make a list by calling `cons` repeatedly, and terminating it with `null`:

```
const oneToFive = cons(1, cons(2, cons(3, cons(4, cons(5, null))));
```

```
oneToFive  
//=> [1, [2, [3, [4, [5, null]]]]]
```

Notice that though JavaScript displays our list as if it is composed of arrays nested within each other like Russian Dolls, in reality the arrays refer to each other with references, so `[1, [2, [3, [4, [5, null]]]]]` is actually more like:

```
const node5 = [5, null],
      node4 = [4, node5],
      node3 = [3, node4],
      node2 = [2, node3],
      node1 = [1, node2];

const oneToFive = node1;
```

This is a [Linked List⁶⁷](#), it's just that those early Lispers used the names `car` and `cdr` after the hardware instructions, whereas today we use words like `data` and `reference`. But it works the same way: If we want the head of a list, we call `car` on it:

```
car(oneToFive)
//=> 1
```

`car` is very fast, it simply extracts the first element of the cons cell.

But what about the rest of the list? `cdr` does the trick:

```
cdr(oneToFive)
//=> [2, [3, [4, [5, null]]]]
```

Again, it's just extracting a reference from a cons cell, it's very fast. In Lisp, it's blazingly fast because it happens in hardware. There's no making copies of arrays, the time to `cdr` a list with five elements is the same as the time to `cdr` a list with 5,000 elements, and no temporary arrays are needed. In JavaScript, it's still much, much, much faster to get all the elements except the head from a linked list than from an array. Getting one reference to a structure that already exists is faster than copying a bunch of elements.

So now we understand that in Lisp, a lot of things use linked lists, and they do that in part because it was what the hardware made possible.

Getting back to JavaScript now, when we write `[first, ...rest]` to gather or spread arrays, we're emulating the semantics of `car` and `cdr`, but not the implementation. We're doing something laborious and memory-inefficient compared to using a linked list as Lisp did and as we can still do if we choose.

That being said, it is easy to understand and helps us grasp how literals and destructuring works, and how recursive algorithms ought to mirror the self-similarity of the data structures they manipulate. And so it is today that languages like JavaScript have arrays that are slow to split into the equivalent of a `car/cdr` pair, but instructional examples of recursive programs still have echoes of their Lisp origins.

We'll look at linked lists again when we look at [Plain Old JavaScript Objects](#).

⁶⁷https://en.wikipedia.org/wiki/Linked_list

so why arrays

If `[first, ...rest]` is so slow, why does JavaScript use arrays instead of making everything a linked list?

Well, linked lists are fast for a few things, like taking the front element off a list, and taking the remainder of a list. But not for iterating over a list: Pointer chasing through memory is quite a bit slower than incrementing an index. In addition to the extra fetches to dereference pointers, pointer chasing suffers from cache misses. And if you want an arbitrary item from a list, you have to iterate through the list element by element, whereas with the indexed array you just fetch it.

We have avoided discussing rebinding and mutating values, but if we want to change elements of our lists, the naïve linked list implementation suffers as well: When we take the `cdr` of a linked list, we are sharing the elements. If we make any change other than `cons-ing` a new element to the front, we are changing both the new list and the old list.

Arrays avoid this problem by pessimistically copying all the references whenever we extract an element or sequence of elements from them (We'll see this explained later in [Mutation](#)).

For these and other reasons, almost all languages today make it possible to use a fast array or vector type that is optimized for iteration, and even Lisp now has a variety of data structures that are optimized for specific use cases.

summary

Although we showed how to use tail calls to map and fold over arrays with `[first, ...rest]`, in reality this is not how it ought to be done. But it is an extremely simple illustration of how recursion works when you have a self-similar means of constructing a data structure.

Plain Old JavaScript Objects

Lists are not the only way to represent collections of things, but they are the “oldest” data structure in the history of high level languages, because they map very closely to the way the hardware is organized in a computer. Lists are obviously very handy for homogeneous collections of things, like a shopping list:

```
const remember = ["the milk", "the coffee beans", "the biscotti"];
```

And they can be used to store heterogeneous things in various levels of structure:

```
const user = [[ "Reginald", "Braithwaite"], [ "author", [ "JavaScript Allongé", "Ja\\
vaScript Spessore", "CoffeeScript Ristretto"]]];
```

Remembering that the name is the first item is error-prone, and being expected to look at `user[0][1]` and know that we are talking about a surname is unreasonable. So back when lists were the only things available, programmers would introduce constants to make things easier on themselves:

```
const NAME = 0,
  FIRST = 0,
  LAST = 1,
  OCCUPATION = 1,
  TITLE = 0,
  RESPONSIBILITIES = 1;
```

```
const user = [[ "Reginald", "Braithwaite"], [ "author", [ "JavaScript Allongé", "Ja\\
vaScript Spessore", "CoffeeScript Ristretto"]]];
```

Now they could write `user[NAME][LAST]` or `user[OCCUPATION][TITLE]` instead of `user[0][1]` or `user[1][0]`. Over time, this need to build heterogeneous data structures with access to members by name evolved into the [Dictionary⁶⁸](#) data type, a mapping from a unique set of objects to another set of objects.

Dictionaries store key-value pairs, so instead of binding `NAME` to `0` and then storing a name in an array at index `0`, we can bind a name directly to `name` in a dictionary, and we let JavaScript sort out whether the implementation is a list of key-value pairs, a hashed collection, a tree of some sort, or anything else.

JavaScript has dictionaries, and it calls them “objects.” The word “object” is loaded in programming circles, due to the widespread use of the term “object-oriented programming” that was coined by Alan Kay but has since come to mean many, many things to many different people.

In JavaScript, an object is a map from string keys to values.

⁶⁸https://en.wikipedia.org/wiki/Associative_array

literal object syntax

JavaScript has a literal syntax for creating objects. This object maps values to the keys `year`, `month`, and `day`:

```
{ year: 2012, month: 6, day: 14 }
```

Two objects created with separate evaluations have differing identities, just like arrays:

```
{ year: 2012, month: 6, day: 14 } === { year: 2012, month: 6, day: 14 }
//=> false
```

Objects use `[]` to access the values by name, using a string:

```
{ year: 2012, month: 6, day: 14 }[ 'day' ]
//=> 14
```

Values contained within an object work just like values contained within an array, we access them by reference to the original:

```
const unique = () => [],
x = unique(),
y = unique(),
z = unique(),
o = { a: x, b: y, c: z };

o[ 'a' ] === x && o[ 'b' ] === y && o[ 'c' ] === z
//=> true
```

Names needn't be alphanumeric strings. For anything else, enclose the label in quotes:

```
{ 'first name': 'reginald', 'last name': 'lewis' }[ 'first name' ]
//=> 'reginald'
```

If the name is an alphanumeric string conforming to the same rules as names of variables, there's a simplified syntax for accessing the values:

```
const date = { year: 2012, month: 6, day: 14 };

date['day'] === date.day
//=> true
```

Expressions can be used for keys as well. The syntax is to enclose the key's expression in [and]:

```
{
  ["p" + "i"] : 3.14159265
}
//=> {"pi":3.14159265}
```

All containers can contain any value, including functions or other containers, like a fat arrow function:

```
const Mathematics = {
  abs: (a) => a < 0 ? -a : a
};
```

```
Mathematics.abs(-5)
//=> 5
```

Or proper functions:

```
const SecretDecoderRing = {
  encode: function (plaintext) {
    return plaintext
      .split('')
      .map( char => char.charCodeAt() )
      .map( code => code + 1 )
      .map( code => String.fromCharCode(code) )
      .join('');
  },
  decode: function (cyphertext) {
    return cyphertext
      .split('')
      .map( char => char.charCodeAt() )
      .map( code => code - 1 )
      .map( code => String.fromCharCode(code) )
      .join('');
  }
}
```

Or named function expressions:

```
const SecretDecoderRing = {
  encode: function encode (plaintext) {
    return plaintext
      .split('')
      .map( char => char.charCodeAt() )
      .map( code => code + 1 )
      .map( code => String.fromCharCode(code) )
      .join('');
  },
  decode: function decode (cyphertext) {
    return cyphertext
      .split('')
      .map( char => char.charCodeAt() )
      .map( code => code - 1 )
      .map( code => String.fromCharCode(code) )
      .join('');
  }
}
```

It is very common to associate named function expressions with keys in objects, and there is a “compact method syntax” for binding named function expressions to keywords:

```
const SecretDecoderRing = {
  encode (plaintext) {
    return plaintext
      .split('')
      .map( char => char.charCodeAt() )
      .map( code => code + 1 )
      .map( code => String.fromCharCode(code) )
      .join('');
  },
  decode (cyphertext) {
    return cyphertext
      .split('')
      .map( char => char.charCodeAt() )
      .map( code => code - 1 )
      .map( code => String.fromCharCode(code) )
      .join('');
  }
}
```

(There are some other technical differences between binding a named function expression and using

compact method syntax, but they are not relevant here. We will generally prefer compact method syntax whenever we can.)

destructuring objects

Just as we saw with arrays, we can write destructuring assignments with literal object syntax. So, we can write:

```
const user = {
  name: { first: "Reginald",
           last: "Braithwaite"
         },
  occupation: { title: "Author",
                responsibilities: [ "JavaScript Allongé",
                                      "JavaScript Spessore",
                                      "CoffeeScript Ristretto"
                                    ]
              }
};

user.name.last
//=> "Braithwaite"

user.occupation.title
//=> "Author"
```

And we can also write:

```
const {name: { first: given, last: surname}, occupation: { title: title } } = user;

surname
//=> "Braithwaite"

title
//=> "Author"
```

And of course, we destructure parameters:

```
const description = ({name: { first: given }, occupation: { title: title } }) =>
`#${given} is a ${title}`;

description(user)
//=> "Reginald is a Author"
```

Terrible grammar and capitalization, but let's move on. It is very common to write things like `title: title` when destructuring objects. When the label is a valid variable name, it's often the most obvious variable name as well. So JavaScript supports a further syntactic optimization:

```
const description = ({name: { first }, occupation: { title } }) =>
`${first} is a ${title}`;

description(user)
//=> "Reginald is a Author"
```

And that same syntax works for literals:

```
const abbrev = ({name: { first, last }, occupation: { title } }) => {
  return { first, last, title};
}

abbrev(user)
//=> {"first": "Reginald", "last": "Braithwaite", "title": "Author"}
```

revisiting linked lists

Earlier, we used two-element arrays as nodes in a linked list:

```
const cons = (a, d) => [a, d],
  car = ([a, d]) => a,
  cdr = ([a, d]) => d;
```

In essence, this simple implementation used functions to create an abstraction with named elements. But now that we've looked at objects, we can use an object instead of a two-element array. While we're at it, let's use contemporary names. So our linked list nodes will be formed from `{ first, rest }`

In that case, a linked list of the numbers 1, 2, and 3 will look like this: `{ first: 1, rest: { first: 2, rest: { first: 3, rest: EMPTY } } }`.

We can then perform the equivalent of `[first, ...rest]` with direct property accessors:

```

const EMPTY = {};
const OneTwoThree = { first: 1, rest: { first: 2, rest: { first: 3, rest: EMPTY \ } } };

OneTwoThree.first
//=> 1

OneTwoThree.rest
//=> {"first":2,"rest":{"first":3,"rest":{}}}

OneTwoThree.rest.rest.first
//=> 3

```

Taking the length of a linked list is easy:

```

const length = (node, delayed = 0) =>
  node === EMPTY
    ? delayed
    : length(node.rest, delayed + 1);

length(OneTwoThree)
//=> 3

```

What about mapping? Well, let's start with the simplest possible thing, making a *copy* of a list. As we saw above, and discussed in [Garbage, Garbage Everywhere](#), it is fast to iterate forward through a linked list. What isn't fast is naively copying a list:

```

const slowcopy = (node) =>
  node === EMPTY
    ? EMPTY
    : { first: node.first, rest: slowcopy(node.rest)};

slowcopy(OneTwoThree)
//=> {"first":1,"rest":{"first":2,"rest":{"first":3,"rest":{}}}}

```

The problem here is that linked lists are constructed back-to-front, but we iterate over them front-to-back. So to copy a list, we have to save all the bits on the call stack and then construct the list from back-to-front as all the recursive calls return.

We could follow the strategy of delaying the work. Let's write that naively:

```
const copy2 = (node, delayed = EMPTY) =>
  node === EMPTY
    ? delayed
    : copy2(node.rest, { first: node.first, rest: delayed });

copy2(OneTwoThree)
//=> {"first":3,"rest":{"first":2,"rest":{"first":1,"rest":{}}}}
```

Well, well, well. We have unwittingly *reversed* the list. This makes sense, if lists are constructed from back to front, and we make a linked list out of items as we iterate through it, we're going to get a backwards copy of the list. This isn't a bad thing by any stretch of the imagination. Let's call it what it is:

```
const reverse = (node, delayed = EMPTY) =>
  node === EMPTY
    ? delayed
    : reverse(node.rest, { first: node.first, rest: delayed });
```

And now, we can make a reversing map:

```
const reverseMapWith = (fn, node, delayed = EMPTY) =>
  node === EMPTY
    ? delayed
    : reverseMapWith(fn, node.rest, { first: fn(node.first), rest: delayed });

reverseMapWith((x) => x * x, OneTwoThree)
//=> {"first":9,"rest":{"first":4,"rest":{"first":1,"rest":{}}}}
```

And a regular `mapWith` follows:

```
const reverse = (node, delayed = EMPTY) =>
  node === EMPTY
    ? delayed
    : reverse(node.rest, { first: node.first, rest: delayed });

const mapWith = (fn, node, delayed = EMPTY) =>
  node === EMPTY
    ? reverse(delayed)
    : mapWith(fn, node.rest, { first: fn(node.first), rest: delayed });

mapWith((x) => x * x, OneTwoThree)
//=> {"first":1,"rest":{"first":4,"rest":{"first":9,"rest":{}}}}
```

Our `mapWith` function takes twice as long as a straight iteration, because it iterates over the entire list twice, once to map, and once to reverse the list. Likewise, it takes twice as much memory, because it constructs a reverse of the desired result before throwing it away.

Mind you, this is still much, much faster than making partial copies of arrays. For a list of length n , we created n superfluous nodes and copied n superfluous values. Whereas our naïve array algorithm created $2n$ superfluous arrays and copied n^2 superfluous values.

Mutation



Cupping Grinds

In JavaScript, almost every type of value can *mutate*. Their identities stay the same, but not their structure. Specifically, arrays and objects can mutate. Recall that you can access a value from within an array or an object using []. You can reassign a value using [] =:

```
const oneTwoThree = [1, 2, 3];
oneTwoThree[0] = 'one';
oneTwoThree
//=> [ 'one', 2, 3 ]
```

You can even add a value:

```
const oneTwoThree = [1, 2, 3];
oneTwoThree[3] = 'four';
oneTwoThree
//=> [ 1, 2, 3, 'four' ]
```

You can do the same thing with both syntaxes for accessing objects:

```
const name = {firstName: 'Leonard', lastName: 'Braithwaite'};
name.middleName = 'Austin'
name
//=> { firstName: 'Leonard',
#   lastName: 'Braithwaite',
#   middleName: 'Austin' }
```

We have established that JavaScript's semantics allow for two different bindings to refer to the same value. For example:

```
const allHallowsEve = [2012, 10, 31]
const halloween = allHallowsEve;
```

Both `halloween` and `allHallowsEve` are bound to the same array value within the local environment. And also:

```
const allHallowsEve = [2012, 10, 31];
(function (halloween) {
  // ...
})(allHallowsEve);
```

There are two nested environments, and each one binds a name to the exact same array value. In each of these examples, we have created two *aliases* for the same value. Before we could reassign things, the most important point about this is that the identities were the same, because they were the same value.

This is vital. Consider what we already know about shadowing:

```
const allHallowsEve = [2012, 10, 31];
(function (halloween) {
  halloween = [2013, 10, 31];
})(allHallowsEve);
allHallowsEve
//=> [2012, 10, 31]
```

The outer value of `allHallowsEve` was not changed because all we did was rebind the name `halloween` within the inner environment. However, what happens if we *mutate* the value in the inner environment?

```
const allHallowsEve = [2012, 10, 31];
(function (halloween) {
  halloween[0] = 2013;
})(allHallowsEve);
allHallowsEve
//=> [2013, 10, 31]
```

This is different. We haven't rebound the inner name to a different variable, we've mutated the value that both bindings share. Now that we've finished with mutation and aliases, let's have a look at it.



JavaScript permits the reassignment of new values to existing bindings, as well as the reassignment and assignment of new values to elements of containers such as arrays and objects. Mutating existing objects has special implications when two bindings are aliases of the same value.



Note well: Declaring a variable `const` does not prevent us from mutating its value, only from rebinding its name. This is an important distinction.

mutation and data structures

Mutation is a surprisingly complex subject. It is possible to compute anything without ever mutating an existing entity. Languages like [Haskell](#)⁶⁹ don't permit mutation at all. In general, mutation makes some algorithms shorter to write and possibly faster, but harder to reason about.

One pattern many people follow is to be liberal with mutation when constructing data, but conservative with mutation when consuming data. Let's recall linked lists from [Plain Old JavaScript Objects](#). While we're executing the `mapWith` function, we're constructing a new linked list. By this pattern, we would be happy to use mutation to construct the list while running `mapWith`.

But after returning the new list, we then become conservative about mutation. This also makes sense: Linked lists often use structure sharing. For example:

⁶⁹https://en.wikipedia.org/wiki/Haskell_

```

const EMPTY = {};
const OneToFive = { first: 1,
  rest: {
    first: 2,
    rest: {
      first: 3,
      rest: {
        first: 4,
        rest: {
          first: 5,
          rest: EMPTY } } } } };

OneToFive
//=> {"first":1,"rest":{"first":2,"rest":{"first":"three","rest":{"first":"four","rest":{"first":"five","rest":{}}}}}}}

const ThreeToFive = OneToFive.rest.rest;

ThreeToFive
//=> {"first":3,"rest":{"first":4,"rest":{"first":5,"rest":{}}}}}

ThreeToFive.first = "three";
ThreeToFive.rest.first = "four";
ThreeToFive.rest.rest.first = "five";

ThreeToFive
//=> {"first":"three","rest":{"first":"four","rest":{"first":"five","rest":{}}}\n}]

OneToFive
//=> {"first":1,"rest":{"first":2,"rest":{"first":"three","rest":{"first":"four","rest":{"first":"five","rest":{}}}}}}}

```

Changes made to `ThreeToFive` affect `OneToFive`, because they share the same structure. When we wrote `ThreeToFive = OneToFive.rest.rest;`, we weren't making a brand new copy of `{"first":3,"rest":{}}`: we were getting a reference to the same chain of nodes.

Structure sharing like this is what makes linked lists so fast for taking everything but the first item of a list: We aren't making a new list, we're using some of the old list. Whereas destructuring an array with `[first, ...rest]` does make a copy, so:

```
const OneToFive = [1, 2, 3, 4, 5];

OneToFive
//=> [1,2,3,4,5]

const [a, b, ...ThreeToFive] = OneToFive;

ThreeToFive
//=> [3, 4, 5]

ThreeToFive[0] = "three";
ThreeToFive[1] = "four";
ThreeToFive[2] = "five";

ThreeToFive
//=> ["three", "four", "five"]

OneToFive
//=> [1,2,3,4,5]
```

The gathering operation `[a, b, ...ThreeToFive]` is slower, but “safer.”

So back to avoiding mutation. In general, it’s easier to reason about data that doesn’t change. We don’t have to remember to use copying operations when we pass it as a value to a function, or extract some data from it. We just use the data, and the less we mutate it, the fewer the times we have to think about whether making changes will be “safe.”

building with mutation

As noted, one pattern is to be more liberal about mutation when building a data structure. Consider our copy algorithm. Without mutation, a copy of a linked list can be made in constant space by reversing a reverse of the list:

```
const reverse = (node, delayed = EMPTY) =>
  node === EMPTY
    ? delayed
    : reverse(node.rest, { first: node.first, rest: delayed });

const copy = (node) => reverse(reverse(node));
```

If we want to make a copy of a linked list without iterating over it twice and making a copy we discard later, we can use mutation:

```
const copy = (node, head = null, tail = null) => {
  if (node === EMPTY) {
    return head;
  }
  else if (tail === null) {
    const { first, rest } = node;
    const newNode = { first, rest };
    return copy(rest, newNode, newNode);
  }
  else {
    const { first, rest } = node;
    const newNode = { first, rest };
    tail.rest = newNode;
    return copy(node.rest, head, newNode);
  }
}
```

This algorithm makes copies of nodes as it goes, and mutates the last node in the list so that it can splice the next one on. Adding a node to an existing list is risky, as we saw when considering the fact that `OneToFive` and `ThreeToFive` share the same nodes. But when we're in the midst of creating a brand new list, we aren't sharing any nodes with any other lists, and we can afford to be more liberal about using mutation to save space and/or time.

Armed with this basic copy implementation, we can write `mapWith`:

```
const mapWith = (fn, node, head = null, tail = null) => {
  if (node === EMPTY) {
    return head;
  }
  else if (tail === null) {
    const { first, rest } = node;
    const newNode = { first: fn(first), rest };
    return mapWith(fn, rest, newNode, newNode);
  }
  else {
    const { first, rest } = node;
    const newNode = { first: fn(first), rest };
    tail.rest = newNode;
    return mapWith(fn, node.rest, head, newNode);
  }
}

mapWith((x) => 1.0 / x, OneToFive)
```

```
//=> {"first":1,"rest":{"first":0.5,"rest":{"first":0.3333333333333333,"rest":\n  {"first":0.25,"rest":{"first":0.2,"rest":{}}}}}}
```

Reassignment

Like some imperative programming languages, JavaScript allows you to re-assign the value bound to parameters. We saw this earlier in [rebinding](#):

By default, JavaScript permits us to *rebind* new values to names bound with a parameter. For example, we can write:

```
const evenStevens = (n) => {
  if (n === 0) {
    return true;
  }
  else if (n == 1) {
    return false;
  }
  else {
    n = n - 2;
    return evenStevens(n);
  }
}

evenStevens(42)
//=> true
```

The line `n = n - 2;` *rebinds* a new value to the name `n`. We will discuss this at much greater length in [Reassignment](#), but long before we do, let's try a similar thing with a name bound using `const`. We've already bound `evenStevens` using `const`, let's try rebinding it:

```
evenStevens = (n) => {
  if (n === 0) {
    return true;
  }
  else if (n == 1) {
    return false;
  }
  else {
    return evenStevens(n - 2);
  }
}

//=> ERROR, evenStevens is read-only
```

JavaScript does not permit us to rebind a name that has been bound with `const`. We can *shadow* it by using `const` to declare a new binding with a new function or block scope, but we cannot rebind a name that was bound with `const` in an existing scope.

Rebinding parameters is usually avoided, but what about rebinding names we declare within a function? What we want is a statement that works like `const`, but permits us to rebind variables. JavaScript has such a thing, it's called `let`:

```
let age = 52;

age = 53;
age
//=> 53
```

We took the time to carefully examine what happens with bindings in environments. Let's take the time to explore what happens with reassigning values to variables. The key is to understand that we are rebinding a different value to the same name in the same environment.

So let's consider what happens with a shadowed variable:

```
(( ) => {
  let age = 49;

  if (true) {
    let age = 50;
  }
  return age;
})()
//=> 49
```

Using `let` to bind `50` to `age` within the block does not change the binding of `age` in the outer environment because the binding of `age` in the block shadows the binding of `age` in the outer environment, just like `const`. We go from:

```
{age: 49, ...: global-environment}
```

To:

```
{age: 50, ...: {age: 49, ...: global-environment}}
```

Then back to:

```
{age: 49, ...: global-environment}
```

However, if we don't shadow `age` with `let`, reassigning within the block changes the original:

```
(( ) => {
  let age = 49;

  if (true) {
    age = 50;
  }
  return age;
})()
//=> 50
```

Like evaluating variable labels, when a binding is rebound, JavaScript searches for the binding in the current environment and then each ancestor in turn until it finds one. It then rebinds the name in that environment.

mixing `let` and `const`

Some programmers dislike deliberately shadowing variables. The suggestion is that shadowing a variable is confusing code. If you buy that argument, the way that shadowing works in JavaScript exists to protect us from accidentally shadowing a variable when we move code around.

If you dislike deliberately shadowing variables, you'll probably take an even more opprobrious view of mixing `const` and `let` semantics with a shadowed variable:

```
(( ) => {
  let age = 49;

  if (true) {
    const age = 50;
  }
  age = 51;
  return age;
})()
//=> 51
```

Shadowing a `let` with a `const` does not change our ability to rebind the variable in its original scope. And:

```
((() => {
  const age = 49;

  if (true) {
    let age = 50;
  }
  age = 52;
  return age;
})()
//=> ERROR: age is read-only
```

Shadowing a `const` with a `let` does not permit it to be rebound in its original scope.

`var`

JavaScript has one *more* way to bind a name to a value, `var`.⁷⁰

`var` looks a lot like `let`:

```
const factorial = (n) => {
  let x = n;
  if (x === 1) {
    return 1;
  }
  else {
    --x;
    return n * factorial(x);
  }
}
```

```
factorial(5)
//=> 120
```

```
const factorial2 = (n) => {
  var x = n;
  if (x === 1) {
    return 1;
  }
  else {
    --x;
```

⁷⁰How many have we seen so far? Well, parameters bind names. Function declarations bind names. Named function expressions bind names. `const` and `let` bind names. So that's five different ways so far. And there are more!

```

    return n * factorial2(x);
}
}

factorial2(5)
//=> 120

```

But of course, it's not exactly like let. It's just different enough to present a source of confusion. First, var is not block scoped, it's function scoped, just like **function declarations**:

```

() => {
  var age = 49;

  if (true) {
    var age = 50;
  }
  return age;
})()
//=> 50

```

Declaring age twice does not cause an error(!), and the inner declaration does not shadow the outer declaration. All var declarations behave as if they were hoisted to the top of the function, a little like function declarations.

But, again, it is unwise to expect consistency. A function declaration can appear anywhere within a function, but the declaration *and* the definition are hoisted. Note this example of a function that uses a helper:

```

const factorial = (n) => {

  return innerFactorial(n, 1);

  function innerFactorial (x, y) {
    if (x == 1) {
      return y;
    }
    else {
      return innerFactorial(x-1, x * y);
    }
  }
}

factorial(4)
//=> 24

```

JavaScript interprets this code as if we had written:

```
const factorial = (n) => {
  let innerFactorial = function innerFactorial (x, y) {
    if (x == 1) {
      return y;
    }
    else {
      return innerFactorial(x-1, x * y);
    }
  }

  return innerFactorial(n, 1);
}
```

JavaScript hoists the `let` and the assignment. But not so with `var`:

```
const factorial = (n) => {

  return innerFactorial(n, 1);

  var innerFactorial = function innerFactorial (x, y) {
    if (x == 1) {
      return y;
    }
    else {
      return innerFactorial(x-1, x * y);
    }
  }
}

factorial(4)
//=> undefined is not a function (evaluating 'innerFactorial(n, 1)')
```

JavaScript hoists the declaration, but not the assignment. It is as if we'd written:

```

const factorial = (n) => {

  let innerFactorial = undefined;

  return innerFactorial(n, 1);

  innerFactorial = function innerFactorial (x, y) {
    if (x == 1) {
      return y;
    }
    else {
      return innerFactorial(x-1, x * y);
    }
  }
}

factorial(4)
//=> undefined is not a function (evaluating 'innerFactorial(n, 1)')

```

In that way, `var` is a little like `const` and `let`, we should always declare and bind names before using them. But it's not like `const` and `let` in that it's function scoped, not block scoped.

why `const` and `let` were invented

`const` and `let` are recent additions to JavaScript. For nearly twenty years, variables were declared with `var` (not counting parameters and function declarations, of course). However, its functional scope was a problem.

We haven't looked at it yet, but JavaScript provides a `for` loop for your iterating pleasure and convenience. It looks a lot like the `for` loop in C. Here it is with `var`:

```

var sum = 0;
for (var i = 1; i <= 100; i++) {
  sum = sum + i
}
sum
#=> 5050

```

Hopefully, you can think of a faster way to calculate this sum.⁷¹ And perhaps you have noticed that `var i = 1` is tucked away instead of being at the top as we prefer. But is this ever a problem?

⁷¹There is a well known story about Karl Friedrich Gauss when he was in elementary school. His teacher got mad at the class and told them to add the numbers 1 to 100 and give him the answer by the end of the class. About 30 seconds later Gauss gave him the answer. The other kids were adding the numbers like this: $1 + 2 + 3 + \dots + 99 + 100 = ?$ But Gauss rearranged the numbers to add them like this: $(1 + 100) + (2 + 99) + (3 + 98) + \dots + (50 + 51) = ?$ If you notice every pair of numbers adds up to 101. There are 50 pairs of numbers, so the answer is $50 \times 101 = 5050$. Of course Gauss came up with the answer about 20 times faster than the other kids.

Yes. Consider this variation:

```
var introductions = [],
  names = ['Karl', 'Friedrich', 'Gauss'];

for (var i = 0; i < 3; i++) {
  introductions[i] = "Hello, my name is " + names[i]
}
introductions
//=> [ 'Hello, my name is Karl',
//      'Hello, my name is Friedrich',
//      'Hello, my name is Gauss' ]
```

So far, so good. Hey, remember that functions in JavaScript are values? Let's get fancy!

```
var introductions = [],
  names = ['Karl', 'Friedrich', 'Gauss'];

for (var i = 0; i < 3; i++) {
  introductions[i] = (soAndSo) =>
    `Hello, ${soAndSo}, my name is ${names[i]}`
}
introductions
//=> [ [Function],
//      [Function],
//      [Function] ]
```

Again, so far, so good. Let's try one of our functions:

```
introductions[1]('Raganwald')
//=> 'Hello, Raganwald, my name is undefined'
```

What went wrong? Why didn't it give us 'Hello, Raganwald, my name is Friedrich'? The answer is that pesky `var i`. Remember that `i` is bound in the surrounding environment, so it's as if we wrote:

```
var introductions = [],
  names = ['Karl', 'Friedrich', 'Gauss'],
  i = undefined;

for (i = 0; i < 3; i++) {
  introductions[i] = function (soAndSo) {
    return "Hello, " + soAndSo + ", my name is " + names[i]
  }
}
introductions
```

Now, at the time we created each function, `i` had a sensible value, like `0`, `1`, or `2`. But at the time we *call* one of the functions, `i` has the value `3`, which is why the loop terminated. So when the function is called, JavaScript looks `i` up in its enclosing environment (its closure, obviously), and gets the value `3`. That's not what we want at all.

The error wouldn't exist at all if we'd used `let` in the first place

```
let introductions = [],
  names = ['Karl', 'Friedrich', 'Gauss'];

for (let i = 0; i < 3; i++) {
  introductions[i] = (soAndSo) =>
    `Hello, ${soAndSo}, my name is ${names[i]}`
}
introductions[1]('Raganwald')
//=> 'Hello, Raganwald, my name is Friedrich'
```

This small error was a frequent cause of confusion, and in the days when there was no block-scoped `let`, programmers would need to know how to fake it, usually with an IIFE:

```
var introductions = [],
  names = ['Karl', 'Friedrich', 'Gauss'];

for (var i = 0; i < 3; i++) {
  ((i) => {
    introductions[i] = (soAndSo) =>
      `Hello, ${soAndSo}, my name is ${names[i]}`
  })
}(i)
introductions[1]('Raganwald')
//=> 'Hello, Raganwald, my name is Friedrich'
```

Now we're creating a new inner parameter, `i` and binding it to the value of the outer `i`. This works, but `let` is so much simpler and cleaner that it was added to the language in the ECMAScript 2015 specification.

In this book, we will use function declarations sparingly, and not use `var` at all. That does not mean that you should follow the exact same practice in your own code: The purpose of this book is to illustrate certain principles of programming. The purpose of your own code is to get things done. The two goals are often, but not always, aligned.

Copy on Write



The Coffee Cow

We've seen how to build lists with arrays and with linked lists. We've touched on an important difference between them:

- When you take the rest of an array with destructuring (`[first, ...rest]`), you are given a *copy* of the elements of the array.
- When you take the rest of a linked list with its reference, you are given the exact same nodes of the elements of the original list.

The consequence of this is that if you have an array, and you take its "rest," your "child" array is a copy of the elements of the parent array. And therefore, modifications to the parent do not affect the child, and modifications to the child do not affect the parent.

Whereas if you have a linked list, and you take it's "rest," your "child" list shares its nodes with the "parent" list. And therefore, modifications to the parent also modify the child, and modifications to the child also modify the parent.

Let's confirm our understanding:

```
const parentArray = [1, 2, 3];
const [aFirst, ...childArray] = parentArray;

parentArray[2] = "three";
childArray[0] = "two";

parentArray
//=> [1, 2, "three"]
childArray
//=> ["two", 3]

const EMPTY = { first: {}, rest: {} };
const parentList = { first: 1, rest: { first: 2, rest: { first: 3, rest: EMPTY } \ 
} };
const childList = parentList.rest;

parentList.rest.rest.first = "three";
childList.first = "two";

parentList
//=> {"first":1, "rest": {"first": "two", "rest": {"first": "three", "rest": {"first": \ 
{}, "rest": {}}}}}
childList
//=> {"first": "two", "rest": {"first": "three", "rest": {"first": {}, "rest": {}}}}
```

This is remarkably unsafe. If we *know* that a list doesn't share any elements with another list, we can safely modify it. But how do we keep track of that? Add a bunch of bookkeeping to track references? We'll end up reinventing reference counting and garbage collection.

a few utilities

before we go any further, let's write a few naïve list utilities so that we can work at a slightly higher level of abstraction:

```
const copy = (node, head = null, tail = null) => {
  if (node === EMPTY) {
    return head;
  }
  else if (tail === null) {
    const { first, rest } = node;
    const newNode = { first, rest };
    return copy(rest, newNode, newNode);
  }
  else {
    const { first, rest } = node;
    const newNode = { first, rest };
    tail.rest = newNode;
    return copy(node.rest, head, newNode);
  }
}

const first = ({first, rest}) => first;
const rest = ({first, rest}) => rest;

const reverse = (node, delayed = EMPTY) =>
  node === EMPTY
  ? delayed
  : reverse(rest(node), { first: first(node), rest: delayed });

const mapWith = (fn, node, delayed = EMPTY) =>
  node === EMPTY
  ? reverse(delayed)
  : mapWith(fn, rest(node), { first: fn(first(node)), rest: delayed });

const at = (index, list) =>
  index === 0
  ? first(list)
  : at(index - 1, rest(list));

const set = (index, value, list, originalList = list) =>
  index === 0
  ? (list.first = value, originalList)
  : set(index - 1, value, rest(list), originalList)

const parentList = { first: 1, rest: { first: 2, rest: { first: 3, rest: EMPTY } }\};
```

```

const childList = rest(parentList);

set(2, "three", parentList);
set(0, "two", childList);

parentList
  //=> {"first":1, "rest":{"first":"two", "rest":{"first":"three", "rest":{"first":\n[], "rest":[]}}}}
childList
  //=> {"first":"two", "rest":{"first":"three", "rest":{"first":[], "rest":[]}}}

```

Our new `at` and `set` functions behave similarly to `array[index]` and `array[index] = value`. The main difference is that `array[index] = value` evaluates to `value`, while `set(index, value, list)` evaluates to the modified list.

copy-on-read

So back to the problem of structure sharing. One strategy for avoiding problems is to be *pessimistic*. Whenever we take the rest of a list, make a copy.

```

const rest = ({first, rest}) => copy(rest);

const parentList = { first: 1, rest: { first: 2, rest: { first: 3, rest: EMPTY }\n}};
const childList = rest(parentList);

const newParentList = set(2, "three", parentList);
set(0, "two", childList);

parentList
  //=> {"first":1, "rest":{"first":2, "rest":{"first":"three", "rest":{"first":[], "\nrest":[]}}}}
childList
  //=> {"first":"two", "rest":{"first":3, "rest":{"first":[], "rest":[]}}}

```

This strategy is called “copy-on-read”, because when we attempt to “read” the value of a child of the list, we make a copy and read the copy of the child. Thereafter, we can write to the parent or the copy of the child freely.

As we expected, making a copy lets us modify the copy without interfering with the original. This is, however, expensive. Sometimes we don’t need to make a copy because we won’t be modifying the list. Our `mapWith` function would be very expensive if we make a copy every time we call `rest(node)`.

There’s also a bug: What happens when we modify the first element of a list? But before we fix that, let’s try being lazy about copying.

copy-on-write

Why are we copying? In case we modify a child list. Ok, what if we do this: Make the copy when we know we are modifying the list. When do we know that? When we call `set`. We'll restore our original definition for `rest`, but change `set`:

```
const rest = ({first, rest}) => rest;

const set = (index, value, list) =>
  index === 0
    ? { first: value, rest: list.rest }
    : { first: list.first, rest: set(index - 1, value, list.rest) };

const parentList = { first: 1, rest: { first: 2, rest: { first: 3, rest: EMPTY } } };
const childList = rest(parentList);

const newParentList = set(2, "three", parentList);
const newChildList = set(0, "two", childList);
```

Our original parent and child lists remain unmodified:

```
parentList
//=> {"first":1,"rest":{"first":2,"rest":{"first":3,"rest":{"first":{},"rest":{}}}}}
childList
//=> {"first":2,"rest":{"first":3,"rest":{"first":{},"rest":{}}}}
```

But our new parent and child lists are copies that contain the desired modifications, without interfering with each other:

```
newParentList
//=> {"first":1,"rest":{"first":2,"rest":{"first":"three","rest":{"first":{},"rest":{}}}}}
newChildList
//=> {"first":"two","rest":{"first":3,"rest":{"first":{},"rest":{}}}}
```

And now functions like `mapWith` that make copies without modifying anything, work at full speed.

This strategy of waiting to copy until you are writing is called copy-on-write, or “COW:”

Copy-on-write is the name given to the policy that whenever a task attempts to make a change to the shared information, it should first create a separate (private) copy of that information to prevent its changes from becoming visible to all the other tasks.—
[Wikipedia](#)⁷²

Like all strategies, it makes a tradeoff: It's much cheaper than pessimistically copying structures when you make an infrequent number of small changes, but if you tend to make a lot of changes to some that you aren't sharing, it's more expensive.

Looking at the code again, you see that the `copy` function doesn't copy on write: It follows the pattern that while constructing something, we own it and can be liberal with mutation. Once we're done with it and give it to someone else, we need to be conservative and use a strategy like copy-on-read or copy-on-write.

⁷²<https://en.wikipedia.org/wiki/Copy-on-write>

Tortoises, Hares, and Teleporting Turtles

A good long while ago (The First Age of Internet Startups), someone asked me one of those pet algorithm questions. It was, “Write an algorithm to detect a loop in a linked list, in constant space.”

I’m not particularly surprised that I couldn’t think up an answer in a few minutes at the time. And to the interviewer’s credit, he didn’t terminate the interview on the spot, he asked me to describe the kinds of things going through my head.

I think I told him that I was trying to figure out if I could adapt a hashing algorithm such as XORing everything together. This is the “trick answer” to a question about finding a missing integer from a list, so I was trying the old, “Transform this into a problem you’ve already solved⁷³” meta-algorithm. We moved on from there, and he didn’t reveal the “solution.”

I went home and pondered the problem. I wanted to solve it. Eventually, I came up with something and tried it (In Java!) on my home PC. I sent him an email sharing my result, to demonstrate my ability to follow through. I then forgot about it for a while. Some time later, I was told that the correct solution was:

```
const EMPTY = null;

const isEmpty = (node) => node === EMPTY;

const pair = (first, rest = EMPTY) => ({first, rest});

const list = (...elements) => {
  const [first, ...rest] = elements;

  return elements.length === 0
    ? EMPTY
    : pair(first, list(...rest))
}

const forceAppend = (list1, list2) => {
  if (isEmpty(list1)) {
    return "FAIL!"
  }
  if (isEmpty(list1.rest)) {
    list1.rest = list2;
  }
  else {
    forceAppend(list1.rest, list2);
  }
}
```

⁷³<http://www-users.cs.york.ac.uk/susan/joke/3.htm#boil>

```
}

}

const tortoiseAndHare = (aPair) => {
  let tortoisePair = aPair,
    harePair = aPair.rest;

  while (true) {
    if (isEmpty(tortoisePair) || isEmpty(harePair)) {
      return false;
    }
    if (tortoisePair.first === harePair.first) {
      return true;
    }

    harePair = harePair.rest;

    if (isEmpty(harePair)) {
      return false;
    }
    if (tortoisePair.first === harePair.first) {
      return true;
    }

    tortoisePair = tortoisePair.rest;
    harePair = harePair.rest;
  }
};

const aList = list(1, 2, 3, 4, 5);

tortoiseAndHare(aList)
//=> false

forceAppend(aList, aList.rest.rest);

tortoiseAndHare(aList);
//=> true
```

This algorithm is called “The Tortoise and the Hare,” and was discovered by Robert Floyd in the 1960s. You have two node references, and one traverses the list at twice the speed of the other. No matter how large it is, you will eventually have the fast reference equal to the slow reference, and thus you’ll detect the loop.

At the time, I couldn't think of any way to use hashing to solve the problem, so I gave up and tried to fit this into a powers-of-two algorithm. My first pass at it was clumsy, but it was roughly equivalent to this:

```
const teleportingTurtle = (list) => {
  let speed = 1,
    rabbit = list,
    turtle = rabbit;

  while (true) {
    for (let i = 0; i <= speed; i += 1) {
      rabbit = rabbit.rest;
      if (rabbit == null) {
        return false;
      }
      if (rabbit === turtle) {
        return true;
      }
    }
    turtle = rabbit;
    speed *= 2;
  }
  return false;
};

const aList = list(1, 2, 3, 4, 5);

teleportingTurtle(aList)
//=> false

forceAppend(aList, aList.rest.rest);

teleportingTurtle(aList);
//=> true
```

Years later, I came across a discussion of this algorithm, [The Tale of the Teleporting Turtle⁷⁴](#). It seems to be faster under certain circumstances, depending on the size of the loop and the relative costs of certain operations.

What's interesting about these two algorithms is that they both *tangle* two separate concerns: How to traverse a data structure, and what to do with the elements that you encounter. In [Functional Iterators](#), we'll investigate one pattern for separating these concerns.

⁷⁴<http://www.penzba.co.uk/Writings/TheTeleportingTurtle.html>

Functional Iterators

Let's consider a remarkably simple problem: Finding the sum of the elements of an array. In tail-recursive style, it looks like this:

```
const arraySum = ([first, ...rest], accumulator = 0) =>
  first === undefined
    ? accumulator
    : arraySum(rest, first + accumulator)

arraySum([1, 4, 9, 16, 25])
//=> 55
```

As we saw earlier, this entangles the mechanism of traversing the array with the business of summing the bits. So we can separate them using `fold`:

```
const callLeft = (fn, ...args) =>
  (...remainingArgs) =>
    fn(...args, ...remainingArgs);

const foldArrayWith = (fn, terminalValue, [first, ...rest]) =>
  first === undefined
    ? terminalValue
    : fn(first, foldArrayWith(fn, terminalValue, rest));

const arraySum = callLeft(foldArrayWith, (a, b) => a + b, 0);

arraySum([1, 4, 9, 16, 25])
//=> 55
```

The nice thing about this is that the definition for `arraySum` mostly concerns itself with summing, and not with traversing over a collection of data. But it still relies on `foldArrayWith`, so it can only sum arrays.

What happens when we want to sum a tree of numbers? Or a linked list of numbers?

Well, we call `arraySum` with an array, and it has baked into it a method for traversing the array. Perhaps we could extract both of those things. Let's rearrange our code a bit:

```

const callRight = (fn, ...args) =>
  (...remainingArgs) =>
    fn(...remainingArgs, ...args);

const foldArrayWith = (fn, terminalValue, [first, ...rest]) =>
  first === undefined
    ? terminalValue
    : fn(first, foldArrayWith(fn, terminalValue, rest));

const foldArray = (array) => callRight(foldArrayWith, array);

const sumFoldable = (folder) => folder((a, b) => a + b, 0);

sumFoldable(foldArray([1, 4, 9, 16, 25]))
//=> 55

```

What we've done is turn an array into a function that folds an array with `const foldArray = (array) => callRight(foldArrayWith, array);`. The `sumFoldable` function doesn't care what kind of data structure we have, as long as it's foldable.

Here it is summing a tree of numbers:

```

const callRight = (fn, ...args) =>
  (...remainingArgs) =>
    fn(...remainingArgs, ...args);

const foldTreeWith = (fn, terminalValue, [first, ...rest]) =>
  first === undefined
    ? terminalValue
    : Array.isArray(first)
      ? fn(foldTreeWith(fn, terminalValue, first), foldTreeWith(fn, terminalValue, rest))
      : fn(first, foldTreeWith(fn, terminalValue, rest));

const foldTree = (tree) => callRight(foldTreeWith, tree);

const sumFoldable = (folder) => folder((a, b) => a + b, 0);

sumFoldable(foldTree([1, [4, [9, 16]], 25]))
//=> 55

```

We've found another way to express the principle of separating traversing a data structure from the operation we want to perform on that data structure, we've completely separated the knowledge of how to sum from the knowledge of how to fold an array or tree (or anything else, really).

iterating

Folding is a universal operation, and with care we can accomplish any task with folds that could be accomplished with that stalwart of structured programming, the `for` loop. Nevertheless, there is some value in being able to express some algorithms as iteration.

JavaScript has a particularly low-level version of `for` loop that mimics the semantics of the C language. Summing the elements of an array can be accomplished with:

```
const arraySum = (array) => {
  let sum = 0;

  for (let i = 0; i < array.length; ++i) {
    sum += array[i];
  }
  return sum
}

arraySum([1, 4, 9, 16, 25])
//=> 55
```

Once again, we're mixing the code for iterating over an array with the code for calculating a sum. And worst of all, we're getting really low-level with details like knowing that the elements of an array are indexed with consecutive integers that begin with 0.

We can write this a slightly different way, using a `while` loop:

```
const arraySum = (array) => {
  let done,
    sum = 0,
    i = 0;

  while ((done = i == array.length, !done)) {
    const value = array[i++];
    sum += value;
  }
  return sum
}

arraySum([1, 4, 9, 16, 25])
//=> 55
```

Notice that buried inside our loop, we have bound the names `done` and `value`. We can put those into a POJO (a [Plain Old JavaScript Object](#)). It'll be a little awkward, but we'll be patient:

```

const arraySum = (array) => {
  let iter,
    sum = 0,
    index = 0;

  while (
    eachIteration = {
      done: index === array.length,
      value: index < array.length ? array[index] : undefined
    },
    ++index,
    !eachIteration.done)
  ) {
    sum += eachIteration.value;
  }
  return sum;
}

arraySum([1, 4, 9, 16, 25])
//=> 55

```

With this code, we make a POJO that has `done` and `value` keys. All the summing code needs to know is to add `eachIteration.value`. Now we can extract the ickiness into a separate function:

```

const arrayIterator = (array) => {
  let i = 0;

  return () => {
    const done = i === array.length;

    return {
      done,
      value: done ? undefined : array[i++]
    }
  }
}

const iteratorSum = (iterator) => {
  let eachIteration,
    sum = 0;

  while ((eachIteration = iterator()), !eachIteration.done)) {

```

```

        sum += eachIteration.value;
    }
    return sum;
}

iteratorSum(arrayIterator([1, 4, 9, 16, 25]))
//=> 55

```

Now this is something else. The `arrayIterator` function takes an array and returns a function we can call repeatedly to obtain the elements of the array. The `iteratorSum` function iterates over the elements by calling the `iterator` function repeatedly until it returns `{ done: true }`.

We can write a different iterator for a different data structure. Here's one for linked lists:

```

const EMPTY = null;

const isEmpty = (node) => node === EMPTY;

const pair = (first, rest = EMPTY) => ({first, rest});

const list = (...elements) => {
    const [first, ...rest] = elements;

    return elements.length === 0
        ? EMPTY
        : pair(first, list(...rest))
}

const print = (aPair) =>
    isEmpty(aPair)
    ? ""
    : `${aPair.first} ${print(aPair.rest)} `

const listIterator = (aPair) =>
    () => {
        const done = isEmpty(aPair);
        if (done) {
            return {done};
        }
        else {
            const {first, rest} = aPair;

            aPair = aPair.rest;

```

```

    return { done, value: first }
}
}

const iteratorSum = (iterator) => {
  let eachIteration,
  sum = 0;

  while ((eachIteration = iterator()), !eachIteration.done)) {
    sum += eachIteration.value;
  }
  return sum
}

const aListIterator = listIterator(list(1, 4, 9, 16, 25));

iteratorSum(aListIterator)
//=> 55

```

unfolding and laziness

Iterators are functions. When they iterate over an array or linked list, they are traversing something that is already there. But they could just as easily manufacture the data as they go. Let's consider the simplest example:

```

const NumberIterator = (number = 0) =>
  () => ({ done: false, value: number++ })

fromOne = NumberIterator(1);

fromOne().value;
//=> 1
fromOne().value;
//=> 2
fromOne().value;
//=> 3
fromOne().value;
//=> 4
fromOne().value;
//=> 5

```

And here's another one:

```

const FibonacciIterator = () => {
  let previous = 0,
    current = 1;

  return () => {
    const value = current;

    [previous, current] = [current, current + previous];
    return {done: false, value};
  };
};

const fib = FibonacciIterator()

fib().value
  //=> 1
fib().value
  //=> 1
fib().value
  //=> 2
fib().value
  //=> 3
fib().value
  //=> 5

```

A function that starts with a seed and expands it into a data structure is called an *unfold*. It's the opposite of a fold. It's possible to write a generic unfold mechanism, but let's pass on to what we can do with unfolded iterators.

For starters, we can map an iterator, just like we map a collection:

```

const mapIteratorWith = (fn, iterator) =>
() => {
  const {done, value} = iterator();

  return ({done, value: done ? undefined : fn(value)});
}

const squares = mapIteratorWith((x) => x * x, NumberIterator(1));

squares().value
  //=> 1
squares().value

```

```
//=> 4
squares().value
//=> 9
```

This business of going on forever has some drawbacks. Let's introduce an idea: A function that takes an iterator and returns another iterator. We can start with `take`, an easy function that returns an iterator that only returns a fixed number of elements:

```
const take = (iterator, numberToTake) => {
  let count = 0;

  return () => {
    if (++count <= numberToTake) {
      return iterator();
    } else {
      return {done: true};
    }
  };
};

const toArray = (iterator) => {
  let eachIteration,
    array = [];

  while ((eachIteration = iterator()), !eachIteration.done) {
    array.push(eachIteration.value);
  }
  return array;
}

toArray(take(FibonacciIterator(), 5))
//=> [1, 1, 2, 3, 5]

toArray(take(squares, 5))
//=> [1, 4, 9, 16, 25]
```

How about the squares of the first five odd numbers? We'll need an iterator that produces odd numbers. We can write that directly:

```

const odds = () => {
  let number = 1;

  return () => {
    const value = number;

    number += 2;
    return {done: false, value};
  }
}

const squareOf = callLeft(mapIteratorWith, (x) => x * x)

toArray(take(squareOf(odds()), 5))
//=> [1, 9, 25, 49, 81]

```

We could also write a filter for iterators to accompany our mapping function:

```

const filterIteratorWith = (fn, iterator) =>
() => {
  do {
    const {done, value} = iterator();
  } while (!done && !fn(value));
  return {done, value};
}

const oddsOf = callLeft(filterIteratorWith, (n) => n % 2 === 1);

toArray(take(squareOf(oddsOf(NumberIterator(1))), 5))
//=> [1, 9, 25, 49, 81]

```

Mapping and filtering iterators allows us to compose the parts we already have, rather than writing a tricky bit of code with ifs and whiles and boundary conditions.

bonus

Many programmers coming to JavaScript from other languages are familiar with three “canonical” operations on collections: folding, filtering, and finding. In Smalltalk, for example, they are known as `collect`, `select`, and `detect`.

We haven’t written anything that finds the first element of an iteration that meets a certain criteria. Or have we?

```
const firstInIteration = (fn, iterator) =>
  take(filterIteratorWith(fn, iterator), 1);
```

This is interesting, because it is lazy: It doesn't apply `fn` to every element in an iteration, just enough to find the first that passes the test. Whereas if we wrote something like:

```
const firstInArray = (fn, array) =>
  array.filter(fn)[0];
```

JavaScript would apply `fn` to every element. If `array` was very large, and `fn` very slow, this would consume a lot of unnecessary time. And if `fn` had some sort of side-effect, the program could be buggy.

caveat

Please note that unlike most of the other functions discussed in this book, iterators are *stateful*. There are some important implications of stateful functions. One is that while functions like `take(...)` appear to create an entirely new iterator, in reality they return a decorated reference to the original iterator. So as you traverse the new decorator, you're changing the state of the original!

For all intents and purposes, once you pass an iterator to a function, you can expect that you no longer "own" that iterator, and that its state either has changed or will change.

Making Data Out Of Functions



Coffee served at the CERN particle accelerator

In our code so far, we have used arrays and objects to represent the structure of data, and we have extensively used the ternary operator to write algorithms that terminate when we reach a base case.

For example, this `length` function uses a function to bind values to names, POJOs to structure nodes, and the ternary function to detect the base case, the empty list.

```

const EMPTY = {};
const OneTwoThree = { first: 1, rest: { first: 2, rest: { first: 3, rest: EMPTY \
} } };

OneTwoThree.first
//=> 1

OneTwoThree.rest.first
//=> 2

OneTwoThree.rest.rest.first
//=> 3

const length = (node, delayed = 0) =>
  node === EMPTY
    ? delayed
    : length(node.rest, delayed + 1);

length(OneTwoThree)
//=> 3

```

A very long time ago, mathematicians like Alonzo Church, Moses Schönfinkel, Alan Turning, and Haskell Curry and asked themselves if we really needed all these features to perform computations. They searched for a radically simpler set of tools that could accomplish all of the same things.

They established that arbitrary computations could be represented a small set of axiomatic components. For example, we don't need arrays to represent lists, or even POJOs to represent nodes in a linked list. We can model lists just using functions.

To Mock a Mockingbird⁷⁵ established the metaphor of songbirds for the combinators, and ever since then logicians have called the K combinator a “kestrel,” the B combinator a “bluebird,” and so forth.

The oscin.es⁷⁶ library contains code for all of the standard combinators and for experimenting using the standard notation.

Let's start with some of the building blocks of combinatory logic, the K, I, and V combinators, nicknamed the “Kestrel”, the “Idiot Bird”, and the “Vireo.”

⁷⁵http://www.amazon.com/gp/product/0192801422/ref=as_li_ss_tl?ie=UTF8&tag=raganwald001-20&linkCode=as2&camp=1789&creative=390957&creativeASIN=0192801422

⁷⁶<http://oscin.es>

```
const K = (x) => (y) => x;
const I = (x) => (x);
const V = (x) => (y) => (z) => z(x)(y);
```

the kestrel and the idiot

A *constant function* is a function that always returns the same thing, no matter what you give it. For example, $(x) \Rightarrow 42$ is a constant function that always evaluates to 42. The kestrel, or K , is a function that makes constant functions. You give it a value, and it returns a constant function that gives that value.

For example:

```
const K = (x) => (y) => x;
```

```
const fortyTwo = K(42);
```

```
fortyTwo(6)
//=> 42
```

```
fortyTwo("Hello")
//=> 42
```

The *identity function* is a function that evaluates to whatever parameter you pass it. So $I(42) \Rightarrow 42$. Very simple, but useful. Now we'll take it one more step forward: Passing a value to K gets a function back, and passing a value to that function gets us a value.

Like so:

```
K(6)(7)
//=> 6
```

```
K(12)(24)
//=> 12
```

This is very interesting. Given two values, we can say that K always returns the *first* value: $K(x)(y) \Rightarrow x$ (that's not valid JavaScript, but it's essentially how it works).

Now, an interesting thing happens when we pass functions to each other. Consider $K(I)$. From what we just wrote, $K(x)(y) \Rightarrow x$. So $K(I)(x) \Rightarrow I$. Makes sense. Now let's tack one more invocation on: What is $K(I)(x)(y)$? If $K(I)(x) \Rightarrow I$, then $K(I)(x)(y) === I(y)$ which is y .

Therefore, $K(I)(x)(y) \Rightarrow y$:

```
K(I)(6)(7)
//=> 7
```

```
K(I)(12)(24)
//=> 24
```

Aha! Given two values, `K(I)` always returns the *second* value.

```
K("primus")("secundus")
//=> "primus"
```

```
K(I)("primus")("secundus")
//=> "secundus"
```

If we are not feeling particularly academic, we can name our functions:

```
const first = K,
      second = K(I);

first("primus")("secundus")
//=> "primus"

second("primus")("secundus")
//=> "secundus"
```

This is very interesting. Given two values, we can say that `K` always returns the *first* value, and given two values, `K(I)` always returns the *second* value.

backwardness

Our `first` and `second` functions are a little different than what most people are used to when we talk about functions that access data. If we represented a pair of values as an array, we'd write them like this:

```
const first = ([first, second]) => first,
      second = ([first, second]) => second;

const latin = ["primus", "secundus"];

first(latin)
//=> "primus"

second(latin)
//=> "secundus"
```

Or if we were using a POJO, we'd write them like this:

```
const first = ({first, second}) => first,
      second = ({first, second}) => second;

const latin = {first: "primus", second: "secundus"};

first(latin)
//=> "primus"

second(latin)
//=> "secundus"
```

In both cases, the functions `first` and `second` know how the data is represented, whether it be an array or an object. You pass the data to these functions, and they extract it.

But the `first` and `second` we built out of `K` and `I` don't work that way. You call them and pass them the bits, and they choose what to return. So if we wanted to use them with a two-element array, we'd need to have a piece of code that calls some code.

Here's the first cut:

```
const first = K,
      second = K(I);

const latin = (selector) => selector("primus")("secundus");

latin(first)
//=> "primus"

latin(second)
//=> "secundus"
```

Our `latin` data structure is no longer a dumb data structure, it's a function. And instead of passing `latin` to `first` or `second`, we pass `first` or `second` to `latin`. It's *exactly backwards* of the way we write functions that operate on data.

the vireo

Given that our `latin` data is represented as the function `(selector) => selector("primus")("secundus")`, our obvious next step is to make a function that makes data. For arrays, we'd write `cons = (first, second) => [first, second]`. For objects we'd write: `cons = (first, second) => {first, second}`. In both cases, we take two parameters, and return the form of the data.

For "data" we access with `K` and `K(I)`, our "structure" is the function `(selector) => selector("primus")("secundus")`. Let's extract those into parameters:

```
(first, second) => (selector) => selector(first)(second)
```

For consistency with the way combinators are written as functions taking just one parameter, we'll [curry⁷⁷](#) the function:

```
(first) => (second) => (selector) => selector(first)(second)
```

Let's try it, we'll use the word `pair` for the function that makes data (When we need to refer to a specific pair, we'll use the name `aPair` by default):

```
const first = K,
      second = K(I),
      pair = (first) => (second) => (selector) => selector(first)(second);

const latin = pair("primus")("secundus");

latin(first)
//=> "primus"

latin(second)
//=> "secundus"
```

It works! Now what is this `pair` function? If we change the names to `x`, `y`, and `z`, we get: `(x) => (y) => (z) => z(x)(y)`. That's the V combinator, the Vireo! So we can write:

⁷⁷<https://en.wikipedia.org/wiki/Currying>

```

const first = K,
  second = K(I),
  pair = V;

const latin = pair("primus")("secundus");

latin(first)
//=> "primus"

latin(second)
//=> "secundus"

```

As an aside, the Vireo is a little like JavaScript's `.apply` function. It says, “take these two values and apply them to this function.” There are other, similar combinators that apply values to functions. One notable example is the “thrush” or T combinator: It takes one value and applies it to a function. It is known to most programmers as `.tap`.

Armed with nothing more than `K`, `I`, and `V`, we can make a little data structure that holds two values, the `cons` cell of Lisp and the node of a linked list. Without arrays, and without objects, just with functions. We'd better try it out to check.

lists with functions as data

Here's another look at linked lists using POJOs. We use the term `rest` instead of `second`, but it's otherwise identical to what we have above:

```

const first = ({first, rest}) => first,
  rest = ({first, rest}) => rest,
  pair = (first, rest) => ({first, rest}),
  EMPTY = ({});

const l123 = pair(1, pair(2, pair(3, EMPTY)));

first(l123)
//=> 1

first(rest(l123))
//=> 2

first(rest(rest(l123)))
//=> 3

```

We can write `length` and `mapWith` functions over it:

```

const length = (aPair) =>
  aPair === EMPTY
    ? 0
    : 1 + length(rest(aPair));

length(1123)
//=> 3

const reverse = (aPair, delayed = EMPTY) =>
  aPair === EMPTY
    ? delayed
    : reverse(rest(aPair), pair(first(aPair), delayed));

const mapWith = (fn, aPair, delayed = EMPTY) =>
  aPair === EMPTY
    ? reverse(delayed)
    : mapWith(fn, rest(aPair), pair(fn(first(aPair)), delayed));

const doubled = mapWith((x) => x * 2, 1123);

first(doubled)
//=> 2

first(rest(doubled))
//=> 4

first(rest(rest(doubled)))
//=> 6

```

Can we do the same with the linked lists we build out of functions? Yes:

```

const first = K,
  rest = K(I),
  pair = V,
  EMPTY = (() => {});

const l123 = pair(1)(pair(2)(pair(3)(EMPTY)));

l123(first)
//=> 1

l123(rest)(first)

```

```
//=> 2

return l123(rest)(rest)(first)
//=> 3
```

We write them in a backwards way, but they seem to work. How about length?

```
const length = (aPair) =>
  aPair === EMPTY
    ? 0
    : 1 + length(aPair(rest));

length(l123)
//=> 3
```

And mapWith?

```
const reverse = (aPair, delayed = EMPTY) =>
  aPair === EMPTY
    ? delayed
    : reverse(aPair(rest), pair(aPair(first))(delayed));

const mapWith = (fn, aPair, delayed = EMPTY) =>
  aPair === EMPTY
    ? reverse(delayed)
    : mapWith(fn, aPair(rest), pair(fn(aPair(first)))(delayed));

const doubled = mapWith((x) => x * 2, l123)

doubled(first)
//=> 2

doubled(rest)(first)
//=> 4

doubled(rest)(rest)(first)
//=> 6
```

Presto, we can use pure functions to represent a linked list. And with care, we can do amazing things like use functions to represent numbers, build more complex data structures like trees, and in fact, anything that can be computed can be computed using just functions and nothing else.

But without building our way up to something insane like writing a JavaScript interpreter using JavaScript functions and no other data structures, let's take things another step in a slightly different direction.

We used functions to replace arrays and POJOs, but we still use JavaScript's built-in operators to test for equality (==) and to branch ?:.

say “please”

We keep using the same pattern in our functions: `aPair === EMPTY ? doSomething : doSomethingElse`. This follows the philosophy we used with data structures: The function doing the work inspects the data structure.

We can reverse this: Instead of asking a pair if it is empty and then deciding what to do, we can ask the pair to do it for us. Here's `length` again:

```
const length = (aPair) =>
  aPair === EMPTY
    ? 0
    : 1 + length(aPair(rest));
```

Let's presume we are working with a slightly higher abstraction, we'll call it a `list`. Instead of writing `length(list)` and examining a list, we'll write something like:

```
const length = (list) => list(
  () => 0,
  (aPair) => 1 + length(aPair(rest)))
);
```

Now we'll need to write `first` and `rest` functions for a list, and those names will collide with the `first` and `rest` we wrote for pairs. So let's disambiguate our names:

```
const pairFirst = K,
      pairRest  = K(I),
      pair = V;

const first = (list) => list(
  () => "ERROR: Can't take first of an empty list",
  (aPair) => aPair(pairFirst)
);

const rest = (list) => list(
```

```

() => "ERROR: Can't take first of an empty list",
(aPair) => aPair(pairRest)
);

const length = (list) => list(
() => 0,
(aPair) => 1 + length(aPair(pairRest)))
);

```

We'll also write a handy list printer:

```

const print = (list) => list(
() => "",
(aPair) => `${aPair(pairFirst)} ${print(aPair(pairRest))}`
);

```

How would all this work? Let's start with the obvious. What is an empty list?

```
const EMPTYLIST = (whenEmpty, unlessEmpty) => whenEmpty()
```

And what is a node of a list?

```

const node = (x) => (y) =>
(whenEmpty, unlessEmpty) => unlessEmpty(pair(x)(y));

```

Let's try it:

```

const l123 = node(1)(node(2)(node(3)(EMPTYLIST)));
print(l123)
//=> 1 2 3

```

We can write `reverse` and `mapWith` as well. We aren't being super-strict about emulating combinatorial logic, we'll use default parameters:

```

const reverse = (list, delayed = EMPTYLIST) => list(
  () => delayed,
  (aPair) => reverse(aPair(pairRest), node(aPair(pairFirst))(delayed))
);

print(reverse(1123));
//=> 3 2 1

const mapWith = (fn, list, delayed = EMPTYLIST) =>
  list(
    () => reverse(delayed),
    (aPair) => mapWith(fn, aPair(pairRest), node(fn(aPair(pairFirst)))(delayed))
  );

print(mapWith(x => x * x, reverse(1123)))
//=> 941

```

We have managed to provide the exact same functionality that `==` and `:` provided, but using functions and nothing else.

functions are not the real point

There are lots of similar texts explaining how to construct complex semantics out of functions. You can establish that `K` and `K(I)` can represent `true` and `false`, model magnitudes with [Church Numerals⁷⁸](#) or [Surreal Numbers⁷⁹](#), and build your way up to printing FizzBuzz.

The superficial conclusion reads something like this:

Functions are a fundamental building block of computation. They are “axioms” of combinatory logic, and can be used to compute anything that JavaScript can compute.

However, that is not the interesting thing to note here. Practically speaking, languages like JavaScript already provide arrays with mapping and folding methods, choice operations, and other rich constructs. Knowing how to make a linked list out of functions is not really necessary for the working programmer. (Knowing that it can be done, on the other hand, is very important to understanding computer science.)

Knowing how to make a list out of just functions is a little like knowing that photons are the [Gauge Bosons⁸⁰](#) of the electromagnetic force. It’s the QED of physics that underpins the Maxwell’s Equations of programming. Deeply important, but not practical when you’re building a bridge.

So what *is* interesting about this? What nags at our brain as we’re falling asleep after working our way through this?

⁷⁸https://en.wikipedia.org/wiki/Church_encoding

⁷⁹https://en.wikipedia.org/wiki/Surreal_number

⁸⁰https://en.wikipedia.org/wiki/Gauge_boson

a return to backward thinking

To make pairs work, we did things *backwards*, we passed the `first` and `rest` functions to the pair, and the pair called our function. As it happened, the pair was composed by the vireo (or V combinator): $(x) \Rightarrow (y) \Rightarrow (z) \Rightarrow z(x)(y)$.

But we could have done something completely different. We could have written a pair that stored its elements in an array, or a pair that stored its elements in a POJO. All we know is that we can pass the pair function a function of our own, at it will be called with the elements of the pair.

The exact implementation of a pair is hidden from the code that uses a pair. Here, we'll prove it:

```
const first = K,
      second = K(I),
      pair = (first) => (second) => {
        const pojo = {first, second};

        return (selector) => selector(pojo.first)(pojo.second);
      };

const latin = pair("primus")("secundus");

latin(first)
//=> "primus"

latin(second)
//=> "secundus"
```

This is a little gratuitous, but it makes the point: The code that uses the data doesn't reach in and touch it: The code that uses the data provides some code and asks the data to do something with it.

The same thing happens with our lists. Here's `length` for lists:

```
const length = (list) => list(
  () => 0,
  (aPair) => 1 + length(aPair(pairRest)))
);
```

We're passing `list` what we want done with an empty list, and what we want done with a list that has at least one element. We then ask `list` to do it, and provide a way for `list` to call the code we pass in.

We won't bother here, but it's easy to see how to swap our functions out and replace them with an array. Or a column in a database. This is fundamentally *not* the same thing as this code for the length of a linked list:

```
const length = (node, delayed = 0) =>
  node === EMPTY
    ? delayed
    : length(node.rest, delayed + 1);
```

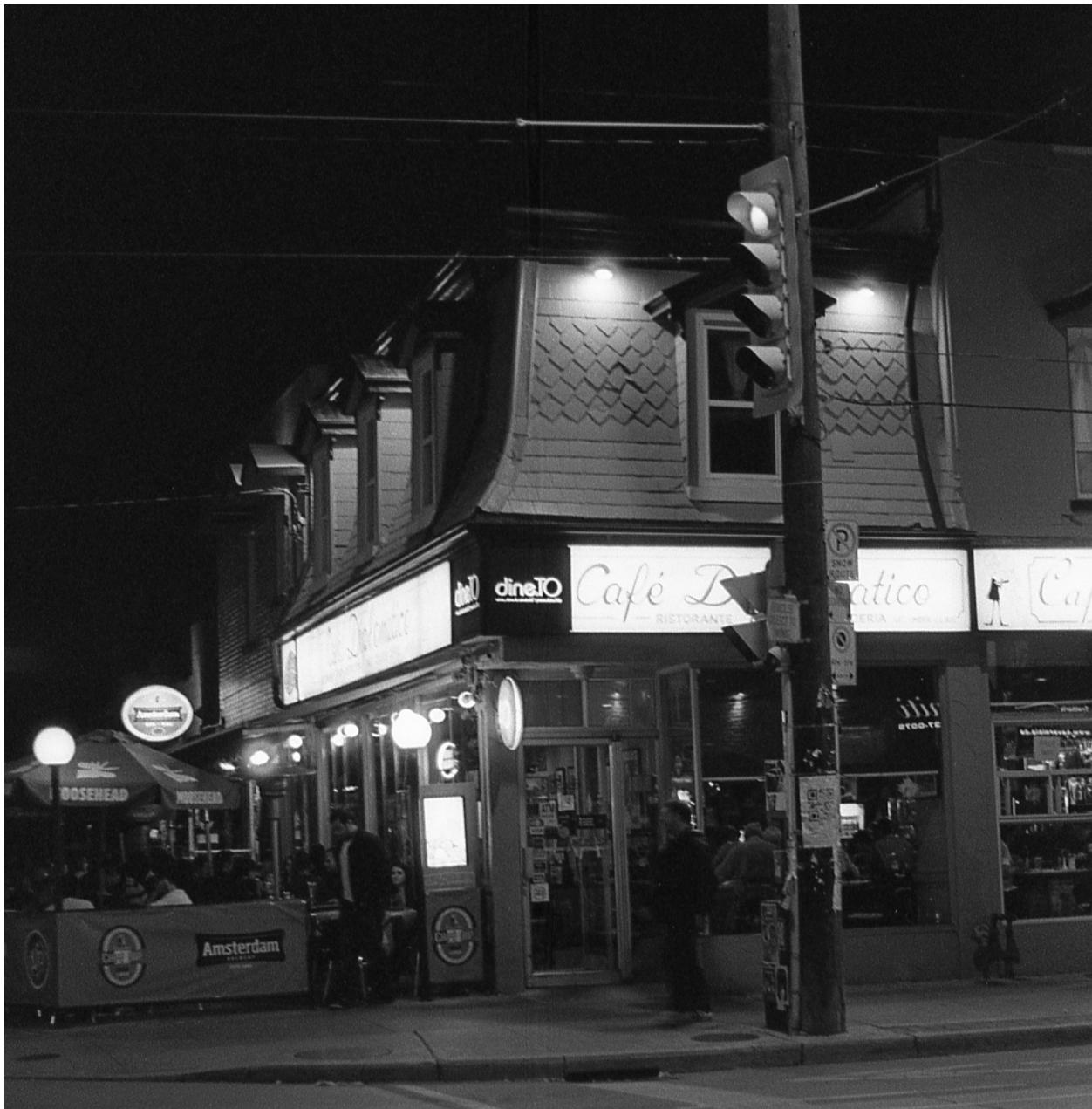
The line `node === EMPTY` presumes a lot of things. It presumes there is one canonical empty list value. It presumes you can compare these things with the `==` operator. We can fix this with an `isEmpty` function, but now we're pushing even more knowledge about the structure of lists into the code that uses them.

Having a list know itself whether it is empty hides implementation information from the code that uses lists. This is a fundamental principle of good design. It is a tenet of Object-Oriented Programming, but it is **not** exclusive to OOP: We can and should design data structures to hide implementation information from the code that use them, whether we are working with functions, objects, or both.

There are many tools for hiding implementation information, and we have now seen two particularly powerful patterns:

- Instead of directly manipulating part of an entity, pass it a function and have it call our function with the part we want.
- And instead of testing some property of an entity and making a choice of our own with `?:` (or `if`), pass the entity the work we want done for each case and let it test itself.

Recipes with Data



Café Diplomatico in Toronto's Little Italy

Disclaimer

The recipes are written for practicality, and their implementation may introduce JavaScript features that haven't been discussed in the text to this point, such as methods and/or prototypes. The overall *use* of each recipe will fit within the spirit of the language discussed so far, even if the implementations may not.

mapWith

In JavaScript, arrays have a `.map` method. `Map` takes a function as an argument, and applies it to each of the elements of the array, then returns the results in another array. For example:

```
[1, 2, 3, 4, 5].map(x => x * x)
//=> [1, 4, 9, 16, 25]
```

We could write a function that behaves like the `.map` method if we wanted:

```
const map = (list, fn) =>
  list.map(fn);
```

This recipe isn't for `map`: It's for `mapWith`, a function that wraps around `map` and turns any other function into a mapper. `mapWith` is very simple:⁸¹

```
const mapWith = (fn) => (list) => list.map(fn);
```

`mapWith` differs from `map` in two ways. It reverses the arguments, taking the function first and the list second. It also “curries” the function: Instead of taking two arguments, it takes one argument and returns a function that takes another argument.

That means that you can pass a function to `mapWith` and get back a function that applies that mapping to any array. For example, we might need a function to return the squares of an array. Instead of writing a wrapper around `.map`:

```
const squaresOf = (list) =>
  list.map(x => x * x);

squaresOf([1, 2, 3, 4, 5])
//=> [1, 4, 9, 16, 25]
```

We can call `mapWith` in one step:

⁸¹If we were always `mapWith`-ing arrays, we could write `list.mapWith(fn)`. However, there are some objects that have a `.length` property and `[]` accessors that can be `mapWith`ed but do not have a `.map` method. `mapWith` works with those objects. This points to a larger issue around the question of whether containers really ought to implement methods like `.map`. In a language like JavaScript, we are free to define objects that know about their own implementations, such as exactly how `[]` and `.length` works and then to define standalone functions that do the rest.

```
const squaresOf = mapWith(n => n * n);

squaresOf([1, 2, 3, 4, 5])
//=> [1, 4, 9, 16, 25]
```

If we didn't use `mapWith`, we'd could have also used `callRight` with `map` to accomplish the same result:

```
const squaresOf = callRight(map, (n => n * n);

squaresOf([1, 2, 3, 4, 5])
//=> [1, 4, 9, 16, 25]
```

Both patterns take us to the same destination: Composing functions out of common pieces, rather than building them entirely from scratch. `mapWith` is a very convenient abstraction for a very common pattern.

mapWith was suggested by ludicast⁸²

⁸²<http://github.com/ludicast>

Flip

We wrote `mapWith` like this:

```
const mapWith = (fn) => (list) => list.map(fn);
```

Let's consider the case whether we have a `map` function of our own, perhaps from the [allong.es⁸³](http://allong.es) library, perhaps from [Underscore⁸⁴](http://underscorejs.org). We could write our function something like this:

```
const mapWith = (fn) => (list) => map(list, fn);
```

Looking at this, we see we're conflating two separate transformations. First, we're reversing the order of arguments. You can see that if we simplify it:

```
const mapWith = (fn, list) => map(list, fn);
```

Second, we're "currying" the function so that instead of defining a function that takes two arguments, it returns a function that takes the first argument and returns a function that takes the second argument and applies them both, like this:

```
const mapper = (list) => (fn) => map(list, fn);
```

Let's return to the implementation of `mapWith` that relies on a `map` function rather than a method:

```
const mapWith = (fn) => (list) => map(list, fn);
```

We're going to extract these two operations by refactoring our function to paramaterize `map`. The first step is to give our parameters generic names:

```
const mapWith = (first) => (second) => map(second, first);
```

Then we wrap the entire thing in a function and extract `map`

```
const wrapper = (fn) =>
  (first) => (second) => fn(second, first);
```

What we have now is a function that takes a function and "flips" the order of arguments around, then curries it. So let's call it `flipAndCurry`:

⁸³<http://allong.es>

⁸⁴<http://underscorejs.org>

```
const flipAndCurry = (fn) =>
  (first) => (second) => fn(second, first);
```

Sometimes you want to flip, but not curry:

```
const flip = (fn) =>
  (first, second) => fn(second, first);
```

This is gold. Consider how we define `mapWith` now:

```
var mapWith = flipAndCurry(map);
```

Much nicer!

self-currying flip

Sometimes we'll want to flip a function, but retain the flexibility to call it in its curried form (pass one parameter) or non-curried form (pass both). We *could* make that into `flip`:

```
const flip = (fn) =>
  function (first, second) {
    if (arguments.length === 2) {
      return fn(second, first);
    }
    else {
      return function (second) {
        return fn(second, first);
      };
    };
  };
};
```

Now if we write `mapWith = flip(map)`, we can call `mapWith(fn, list)` or `mapWith(fn)(list)`, our choice.

flipping methods

When we learn about context and methods, we'll see that `flip` throws the current context away, so it can't be used to flip methods. A small alteration gets the job done:

```
const flipAndCurry = (fn) =>
  (first) =>
    function (second) {
      return fn.call(this, second, first);
    }
}

const flip = (fn) =>
  function (first, second) {
    return fn.call(this, second, first);
  }
}

const flip = (fn) =>
  function (first, second) {
    if (arguments.length === 2) {
      return fn.call(this, second, first);
    }
    else {
      return function (second) {
        return fn.call(this, second, first);
      };
    };
  };
};
```

Object.assign

It's very common to want to "extend" an object by assigning properties to it:

```
const inventory = {
  apples: 12,
  oranges: 12
};

inventory.bananas = 54;
inventory.pears = 24;
```

It's also common to want to assign the properties of one object to another:

```
for (let fruit in shipment) {
  inventory[fruit] = shipment[fruit]
}
```

Both needs can be met with `Object.assign`, a standard function. You can copy an object by extending an empty object:

```
Object.assign({}, {
  apples: 12,
  oranges: 12
})
//=> { apples: 12, oranges: 12 }
```

You can extend one object with another:

```
const inventory = {
  apples: 12,
  oranges: 12
};

const shipment = {
  bananas: 54,
  pears: 24
}

Object.assign(inventory, shipment)
```

```
//=> { apples: 12,  
//       oranges: 12,  
//       bananas: 54,  
//       pears: 24 }
```

And when we discuss prototypes, we will use `Object.assign` to turn this:

```
const Queue = function () {  
  this.array = [];  
  this.head = 0;  
  this.tail = -1  
};  
  
Queue.prototype.pushTail = function (value) {  
  // ...  
};  
Queue.prototype.pullHead = function () {  
  // ...  
};  
Queue.prototype.isEmpty = function () {  
  // ...  
}
```

Into this:

```
const Queue = function () {  
  Object.assign(this, {  
    array: [],  
    head: 0,  
    tail: -1  
  })  
};  
  
Object.assign(Queue.prototype, {  
  pushTail (value) {  
    // ...  
  },  
  pullHead () {  
    // ...  
  },  
  isEmpty () {  
    // ...  
  }  
});
```

```
    }  
});
```

Assigning properties from one object to another (also called “cloning” or “shallow copying”) is a basic building block that we will later use to implement more advanced paradigms like mixins.

Why?

This is the canonical Y Combinator⁸⁵:

```
const Y = (f) =>
  ( x => f(v => x(x)(v)) )( 
    x => f(v => x(x)(v))
  );
```

You use it like this:

```
const factorial = Y(function (fac) {
  return function (n) {
    return (n == 0 ? 1 : n * fac(n - 1));
  }
});

factorial(5)
//=> 120
```

Why? It enables you to make recursive functions without needing to bind a function to a name in an environment. This has little practical utility in JavaScript, but in combinatory logic it's essential: With fixed-point combinators it's possible to compute everything computable without binding names.

So again, why include the recipe? Well, besides all of the practical applications that combinators provide, there is this little thing called *The joy of working things out*.

There are many explanations of the Y Combinator's mechanism on the internet, but resist the temptation to read any of them: Work it out for yourself. Use it as an excuse to get familiar with your environment's debugging facility.

One tip is to use JavaScript to name things. For example, you could start by writing:

```
const Y = (f) => {
  const something = x => f(v => x(x)(v));

  return something(something);
};
```

What is this `something` and how does it work? Another friendly tip: Change some of the fat arrow functions inside of it into named function expressions to help you decipher stack traces.

Work things out for yourself!

⁸⁵https://en.wikipedia.org/wiki/Fixed-point_combinator#Example_in_JavaScript

A Warm Cup: Basic Strings and Quasi-Literals



Coffee and a Book

An expression is any valid unit of code that resolves to a value.—[Mozilla Development Network: Expressions and operators](#)⁸⁶

Like most programming languages, JavaScript also has string literals, like 'fubar' or 'fizzbuzz'. Special characters can be included in a string literal by means of an *escape sequence*. For example, the escape sequence \n inserts a newline character in a string literal, like this: 'first line\nsecond line'.

There are operators that can be used on strings. The most common is +, it *concatenates*:

```
'fu' + 'bar'  
//=> 'fubar'
```

String manipulation is extremely common in programming. Writing is a big part of what makes us human, and strings are how JavaScript and most other languages represent writing.

quasi-literals

JavaScript supports *quasi-literal strings*, a/k/a “Template Strings” or “String Interpolation Expressions.” A quasi-literal string is something that looks like a string literal, but is actually an expression. Quasi-literal strings are denoted with back quotes, and most strings that can be expressed as literals have the exact same meaning as quasi-literals, e.g.

```
`foobar`  
//=> 'foobar'  
  
`fizz` + `buzz`  
//=> 'fizzbuzz'
```

Quasi-literals go much further. A quasi-literal can contain an expression to be evaluated. Old-school lispers call this “unquoting,” the more contemporary term is “interpolation.” An unquoted expression is inserted in a quasi-literal with \${expression}. The expression is evaluated, and the result is coerced to a string, then inserted in the quasi-string.

For example:

```
`A popular number for nerds is ${40 + 2}`  
//=> 'A popular number for nerds is 42'
```

A quasi-literal is computationally equivalent to an expression using +. So the above expression could also be written:

⁸⁶https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Expressions_and_Operators

```
'A popular number for nerds is ' + (40 + 2)
//=> 'A popular number for nerds is 42'
```

However, there is a big semantic difference between a quasi-literal and an expression. Quasi-literals are expressions that resemble their result. They're easier to read and it's easier to avoid errors like the following:

```
'A popular number for nerds is' + (40 + 2)
//=> 'A popular number for nerds is42'
```

evaluation time

Like any other expression, quasi-literals are evaluated *late*, when that line or lines of code is evaluated.

So for example,

```
const name = "Harry";

const greeting = (name) => `Hello my name is ${name}`;

greeting('Arthur Dent')
//=> 'Hello my name is Arthur Dent'
```

JavaScript evaluates the quasi-literal when the function is invoked and the quasi-literal inside the function's body is evaluated. Thus, `name` is not bound to "Harry", it is bound to 'Arthur Dent', the value of the parameter when the function is invoked.

This is exactly what we'd expect if we'd written it like this:

```
const greeting = (name) => 'Hello my name is ' + name;

greeting('Arthur Dent')
//=> 'Hello my name is Arthur Dent'
```

Stir the Allongé: Objects and State



Life measured out by coffee spoons

So far, we have discussed what many call “pure functional” programming, where every expression is necessarily **idempotent⁸⁷**, because we have no way of changing state within a program using the tools we have examined.

We’ve also explored functions that rebind names within themselves as part of performing their calculations. And we briefly touched upon the notion of mutating an object as part of building it. But we have avoided objects that are meant to be changed, objects that model *state*.

It’s time to change *everything*.

⁸⁷<https://en.wikipedia.org/wiki/Idempotence>

Encapsulating State with Closures

OOP to me means only messaging, local retention and protection and hiding of state-process, and extreme late-binding of all things.—[Alan Kay](#)⁸⁸

We're going to look at encapsulation using JavaScript's functions and objects. We're not going to call it object-oriented programming, mind you, because that would start a long debate. This is just plain encapsulation,⁸⁹ with a dash of information-hiding.

what is hiding of state-process, and why does it matter?

In computer science, information hiding is the principle of segregation of the design decisions in a computer program that are most likely to change, thus protecting other parts of the program from extensive modification if the design decision is changed. The protection involves providing a stable interface which protects the remainder of the program from the implementation (the details that are most likely to change).

Written another way, information hiding is the ability to prevent certain aspects of a class or software component from being accessible to its clients, using either programming language features (like private variables) or an explicit exporting policy.

—[Wikipedia](#)⁹⁰

Consider a [stack](#)⁹¹ data structure. There are three basic operations: Pushing a value onto the top (`push`), popping a value off the top (`pop`), and testing to see whether the stack is empty or not (`isEmpty`). These three operations are the stable interface.

Many stacks have an array for holding the contents of the stack. This is relatively stable. You could substitute a linked list, but in JavaScript, the array is highly efficient. You might need an index, you might not. You could grow and shrink the array, or you could allocate a fixed size and use an index to keep track of how much of the array is in use. The design choices for keeping track of the head of the list are often driven by performance considerations.

If you expose the implementation detail such as whether there is an index, sooner or later some programmer is going to find an advantage in using the index directly. For example, she may need to know the size of a stack. The ideal choice would be to add a `size` function that continues to hide the implementation. But she's in a hurry, so she reads the `index` directly. Now her code is coupled to the existence of an index, so if we wish to change the implementation to grow and shrink the array, we will break her code.

⁸⁸http://userpage.fu-berlin.de/~ram/pub/pub_jf47ht81Ht/doc_kay_oop_en

⁸⁹“A language construct that facilitates the bundling of data with the methods (or other functions) operating on that data.”—[Wikipedia](#)

⁹⁰https://en.wikipedia.org/wiki/Information_hiding

⁹¹[https://en.wikipedia.org/wiki/Stack_\(abstract_data_type\)](https://en.wikipedia.org/wiki/Stack_(abstract_data_type))

The way to avoid this is to hide the array and index from other code and only expose the operations we have deemed stable. If and when someone needs to know the size of the stack, we'll add a `size` function and expose it as well.

Hiding information (or “state”) is the design principle that allows us to limit the coupling between components of software.

how do we hide state using javascript?

We've been introduced to JavaScript's objects, and it's fairly easy to see that objects can be used to model what other programming languages call (variously) records, structs, frames, or what-have-you. And given that their elements are mutable, they can clearly model state.

Given an object that holds our state (an array and an index⁹²), we can easily implement our three operations as functions. Bundling the functions with the state does not require any special “magic” features. JavaScript objects can have elements of any type, including functions.

To make our stack work, we need a way for our functions to refer to our stack. We'll do that by making sure it has a name. We can do that with an IIFE:

```
const stack = (() => {
  const obj = {
    array: [],
    index: -1,
    push (value) {
      return obj.array[obj.index += 1] = value
    },
    pop () {
      const value = obj.array[obj.index];
      obj.array[obj.index] = undefined;
      if (obj.index >= 0) {
        obj.index -= 1
      }
      return value
    },
    isEmpty () {
      return obj.index < 0
    }
  };
  return obj;
});
```

⁹²Yes, there's another way to track the size of the array, but we don't need it to demonstrate encapsulation and hiding of state.

```

});();

stack.isEmpty()
  //=> true
stack.push('hello')
  //=> 'hello'
stack.push('JavaScript')
  //=> 'JavaScript'
stack.isEmpty()
  //=> false
stack.pop()
  //=> 'JavaScript'
stack.pop()
  //=> 'hello'
stack.isEmpty()
  //=> true

```

method-ology

In this text, we lurch from talking about “functions that belong to an object” to “methods.” Other languages may separate methods from functions very strictly, but in JavaScript every method is a function, but not all functions are methods.

The view taken in this book is that a function is a method of an object if it belongs to that object and interacts with that object in some way. So the functions implementing the operations on the stack are all absolutely methods of the stack.

But these two wouldn’t be methods. Although they “belong” to an object, they don’t interact with it:

```
{
  min: (x, y) =>
    x < y ? x : y
  max: (x, y) =>
    x > y ? x : y
}
```

hiding state

Our stack does bundle functions with data, but it doesn’t hide its state. “Foreign” code could interfere with its array or index. So how do we hide these? We already have a closure, let’s use it:

```
const stack = () => {
  let array = [],
    index = -1;

  const obj = {
    push (value) { return array[index += 1] = value },
    pop () {
      const value = array[index];

      array[index] = undefined;
      if (index >= 0) {
        index -= 1
      }
      return value
    },
    isEmpty () { return index < 0 }
  };

  return obj;
})();

stack.isEmpty()
  //=> true
stack.push('hello')
  //=> 'hello'
stack.push('JavaScript')
  //=> 'JavaScript'
stack.isEmpty()
  //=> false
stack.pop()
  //=> 'JavaScript'
stack.pop()
  //=> 'hello'
stack.isEmpty()
  //=> true
```



Coffee DOES grow on trees

We don't want to repeat this code every time we want a stack, so let's make ourselves a "stack maker." The temptation is to wrap what we have above in a function:

```
const Stack = () =>
  () => {
    let array = [],
      index = -1;

    const obj = {
      push (value) { return array[index += 1] = value },
      pop () {
        const value = array[index];

        array[index] = undefined;
        if (index >= 0) {
          index -= 1
        }
        return value
      },
      isEmpty () { return index < 0 }
    };

    return obj;
  }();
}
```

But there's an easier way :-)

```
const Stack = () => {
  const array = [];
  let index = -1;

  return {
    push (value) { return array[index += 1] = value },
    pop () {
      const value = array[index];

      array[index] = undefined;
      if (index >= 0) {
        index -= 1
      }
      return value
    },
    isEmpty () { return index < 0 }
  }
}

const stack = Stack();
stack.push("Hello");
stack.push("Good bye");

stack.pop()
//=> "Good bye"
stack.pop()
//=> "Hello"
```

Now we can make stacks freely, and we've hidden their internal data elements. We have methods and encapsulation, and we've built them out of JavaScript's fundamental functions and objects. In [Constructors and Classes](#), we'll look at JavaScript's support for class-oriented programming and some of the idioms that functions bring to the party.

is encapsulation “object-oriented?”

We've built something with hidden internal state and “methods,” all without needing special `def` or `private` keywords. Mind you, we haven't included all sorts of complicated mechanisms to support inheritance, mixins, and other opportunities for debating the nature of the One True Object-Oriented Style on the Internet.

Then again, the key lesson experienced programmers repeat—although it often falls on deaf ears—is

Composition instead of Inheritance^a. So maybe we aren't missing much.

^a<http://www.c2.com/cgi/wiki?CompositionInsteadOfInheritance>

Composition and Extension

composition

A deeply fundamental practice is to build components out of smaller components. The choice of how to divide a component into smaller components is called *factoring*, after the operation in number theory⁹³.

The simplest and easiest way to build components out of smaller components in JavaScript is also the most obvious: Each component is a value, and the components can be put together into a single object or encapsulated with a closure.

Here's an abstract "model" that supports undo and redo composed from a pair of stacks (see [Encapsulating State](#)), and a Plain Old JavaScript Object:

We can set and get attributes on a model

```
// helper function
//
// For production use, consider what to do about
// deep copies and own keys
const shallowCopy = (source) => {
  const dest = {};

  for (let key in source) {
    dest[key] = source[key]
  }
  return dest
};

const Stack = () => {
  const array = [];
  let index = -1;

  return {
    push (value) {
      array[index += 1] = value
    },
    pop () {
      let value = array[index];
      if (index >= 0) {
        index -= 1;
        return value;
      }
    }
  };
}
```

⁹³And when you take an already factored component and rearrange things so that it is factored into a different set of subcomponents without altering its behaviour, you are *refactoring*.

```
        index -= 1
    }
    return value
},
isEmpty () {
    return index < 0
}
}
}

const Model = function (initialAttributes) {
    const redoStack = Stack();
    let attributes = shallowCopy(initialAttributes || {});

    const undoStack = Stack(),
        obj = {
            set: (attrsToSet) => {
                undoStack.push(shallowCopy(attributes));
                if (!redoStack.isEmpty()) {
                    redoStack.length = 0;
                }
                for (let key in (attrsToSet || {})) {
                    attributes[key] = attrsToSet[key]
                }
                return obj
            },
            undo: () => {
                if (!undoStack.isEmpty()) {
                    redoStack.push(shallowCopy(attributes));
                    attributes = undoStack.pop()
                }
                return obj
            },
            redo: () => {
                if (!redoStack.isEmpty()) {
                    undoStack.push(shallowCopy(attributes));
                    attributes = redoStack.pop()
                }
                return obj
            },
            get: (key) => attributes[key],
            has: (key) => attributes.hasOwnProperty(key),
        }
}
```

```

        attributes: () => shallowCopy(attributes)
    };
    return obj
};

const model = Model();
model.set({ "Doctor": "de Grasse" });
model.set({ "Doctor": "Who" });
model.undo()
model.get("Doctor")
//=> "de Grasse"

```

The techniques used for encapsulation work well with composition. In this case, we have a “model” that hides its attribute store as well as its implementation that is composed of an undo stack and redo stack.

extension

Another practice that many people consider fundamental is to *extend* an implementation. Meaning, they wish to define a new data structure in terms of adding new operations and semantics to an existing data structure.

Consider a [queue](#)⁹⁴:

```

const Queue = () => {
  let array = [],
    head = 0,
    tail = -1;

  return {
    pushTail: (value) => array[++tail] = value,
    pullHead: () => {
      if (tail >= head) {
        const value = array[head];

        array[head] = undefined;
        ++head;
        return value
      }
    },
    isEmpty: () => tail < head
  }
}

```

⁹⁴http://duckduckgo.com/Queue_

```

    }
};

const queue = Queue();
queue.pushTail("Hello");
queue.pushTail("JavaScript");
queue.pushTail("Allongé");

queue.pullHead()
 //=> "Hello"
queue.pullHead()
 //=> "JavaScript"

```

Now we wish to create a `deque`⁹⁵ by adding `pullTail` and `pushHead` operations to our `queue`.⁹⁶ Unfortunately, encapsulation prevents us from adding operations that interact with the hidden data structures.

This isn't really surprising: The entire point of encapsulation is to create an opaque data structure that can only be manipulated through its public interface. The design goals of encapsulation and extension are always going to exist in tension.

Let's “de-encapsulate” our `queue`:

```

const Queue = function () {
  const queue = {
    array: [],
    head: 0,
    tail: -1,
    pushTail: (value) =>
      queue.array[++queue.tail] = value,
    pullHead: () => {
      if (queue.tail >= queue.head) {
        const value = queue.array[queue.head];

        queue.array[queue.head] = undefined;
        queue.head += 1;
        return value
      }
    },
}

```

⁹⁵https://en.wikipedia.org/wiki/Double-ended_queue

⁹⁶Before you start wondering whether a `deque` is-a `queue`, we said nothing about types and classes. This relationship is called was-a, or “implemented in terms of a.”

```

isEmpty: () =>
  queue.tail < queue.head
};

return queue
};

```

Now we can extend a queue into a deque:

```

const extend = function (consumer, ...providers) {
  for (let i = 0; i < providers.length; ++i) {
    const provider = providers[i];
    for (let key in provider) {
      if (provider.hasOwnProperty(key)) {
        consumer[key] = provider[key]
      }
    }
  }
  return consumer
};

const Dequeue = function () {
  const deque = Queue(),
    INCREMENT = 4;

  return Object.assign(deque, {
    size: () => deque.tail - deque.head + 1,
    pullTail: () => {
      if (!deque.isEmpty()) {
        const value = deque.array[deque.tail];

        deque.array[deque.tail] = undefined;
        deque.tail -= 1;
        return value
      }
    },
    pushHead: (value) => {
      if (deque.head === 0) {
        for (let i = deque.tail; i <= deque.head; i++) {
          deque.array[i + INCREMENT] = deque.array[i]
        }
        deque.tail += INCREMENT
        deque.head += INCREMENT
      }
    }
  });
}

```

```
        }
        return deque.array[deque.head -= 1] = value
    }
})
};
```

Presto, we have reuse through extension, at the cost of encapsulation.



Encapsulation and Extension exist in a natural state of tension. A program with elaborate encapsulation resists breakage but can also be difficult to refactor in other ways. Be mindful of when it's best to Compose and when it's best to Extend.

This and That

Let's take another look at [extensible objects](#). Here's a Queue:

```
const Queue = () => {
  const queue = {
    array: [],
    head: 0,
    tail: -1,
    pushTail (value) {
      return queue.array[queue.tail] = value
    },
    pullHead () {
      if (queue.tail >= queue.head) {
        const value = queue.array[queue.head];
        queue.array[queue.head] = undefined;
        queue.head += 1;
        return value
      }
    },
    isEmpty () {
      return queue.tail < queue.head;
    }
  };
  return queue
};

const queue = Queue();
queue.pushTail('Hello');
queue.pushTail('JavaScript');
```

Let's make a copy of our queue using `Object.assign`:

```
const copyOfQueue = Object.assign({}, queue);

queue !== copyOfQueue
//=> true
```

Wait a second. We know that array values are references. So it probably copied a reference to the original array. Let's make a copy of the array as well:

```
copyOfQueue.array = [];
for (let i = 0; i < 2; ++i) {
  copyOfQueue.array[i] = queue.array[i]
}
```

Now let's pull the head off the original:

```
queue.pullHead()
//=> 'Hello'
```

If we've copied everything properly, we should get the exact same result when we pull the head off the copy:

```
copyOfQueue.pullHead()
//=> 'JavaScript'
```

What!? Even though we carefully made a copy of the array to prevent aliasing, it seems that our two queues behave like aliases of each other. The problem is that while we've carefully copied our array and other elements over, *the closures all share the same environment*, and therefore the functions in `copyOfQueue` all operate on the first queue's private data, not on the copies.

This is a general issue with closures. Closures couple functions to environments, and that makes them very elegant in the small, and very handy for making opaque data structures. Alas, their strength in the small is their weakness in the large. When you're trying to make reusable components, this coupling is sometimes a hindrance.

Let's take an impossibly optimistic flight of fancy:

```
const AmnesiacQueue = () =>
  ({
    array: [],
    head: 0,
    tail: -1,
    pushTail (myself, value) {
      return myself.array[myself.tail += 1] = value
    },
    pullHead (myself) {
      if (myself.tail >= myself.head) {
        let value = myself.array[myself.head];
        myself.array[myself.head] = undefined;
        myself.tail -= 1;
        return value;
      }
    }
  })
```

```
        myself.array[myself.head] = void 0;
        myself.head += 1;
        return value
    }
},
isEmpty (myself) {
    return myself.tail < myself.head
}
});
};

const queueWithAmnesia = AmnesiacQueue();

queueWithAmnesia.pushTail(queueWithAmnesia, 'Hello');
queueWithAmnesia.pushTail(queueWithAmnesia, 'JavaScript');
queueWithAmnesia.pullHead(queueWithAmnesia)
//=> "Hello"
```

The `AmnesiacQueue` makes queues with amnesia: They don't know who they are, so every time we invoke one of their functions, we have to tell them who they are. You can work out the implications for copying queues as a thought experiment: We don't have to worry about environments, because every function operates on the queue you pass in.

The killer drawback, of course, is making sure we are always passing the correct queue in every time we invoke a function. What to do?

what's all this?

Any time we must do the same repetitive thing over and over and over again, we industrial humans try to build a machine to do it for us. JavaScript is one such machine. When we write a function expression using the compact method syntax (or use the `function` keyword instead of the fat arrow), and then invoke that function using `. notation`, JavaScript binds the “receiver” of a “method invocation” to the special name `this`.

Our `AmnesiacQueue` already uses compact method notation. So, we'll remove `myself` from the parameter list, and rename it to `this` within the body of each function:

```
const BetterQueue = () =>
  ({
    array: [],
    head: 0,
    tail: -1,
    pushTail (value) {
      return this.array[this.tail += 1] = value
    },
    pullHead () {
      if (this.tail >= this.head) {
        let value = this.array[this.head];
        this.array[this.head] = undefined;
        this.head += 1;
        return value
      }
    },
    isEmpty () {
      return this.tail < this.head
    }
  });

```

Now we are relying on JavaScript to set the value of `this` whenever we invoke one of these functions using the `.` or `[]` operators.

In other words, when we write:

```
const betterQueue = BetterQueue();

betterQueue.pushTail('Hello');
betterQueue.pushTail('JavaScript');
betterQueue.pullHead()
```

We expect that JavaScript will invoke the functions we've bound to `pushTail` and `pullHead`, and automatically bind `betterQueue` to the name `this` within them. And indeed it does: Every time you invoke a function that is a member of an object, JavaScript binds that object to the name `this` in the environment of the function just as if it was an argument.⁹⁷

Now, does this solve our original problem? Can we make copies of an object? Recall that the problem was that when we used a closure for private data, copying references to an object's functions meant that we were using functions that still referred to the original closure, and therefore shared the same private data.

⁹⁷JavaScript also does other things with `this` as well, but this is all we care about right now.

Now our functions refer to members of the object, and use `this` to ensure that they are referring to the object receiving a message. Let's see if this does, indeed, allow us to copy objects:

```
const copyOfQueue = Object.assign({}, betterQueue)
copyOfQueue.array = []
for (let i = 0; i < 2; ++i) {
  copyOfQueue.array[i] = betterQueue.array[i]
}

betterQueue.pullHead()
//=> 'Hello'

copyOfQueue.pullHead()
//=> 'Hello'
```

Presto, we now have a way to copy arrays. By getting rid of the closure and taking advantage of `this`, we have functions that are more easily portable between objects, and the code is simpler as well. **This is very important.** Being able to copy objects is an example of a larger concern: Being able to share functions between objects. That's how classes work. That's how extending objects works. Being able to share functions means being able to compose and reuse functionality.

There is more to `this` than we've discussed here. We'll explore things in more detail later, in [What Context Applies When We Call a Function?](#).



Closures tightly couple functions to the environments where they are created limiting their flexibility. Using `this` alleviates the coupling. Copying objects is but one example of where that flexibility is needed.

What Context Applies When We Call a Function?

In [This and That](#), we learned that when a function is denoted using the `function` keyword, and is called as an object method, the name `this` is bound in its environment to the object acting as a “receiver.” For example:

```
const someObject = {
  returnMyThis () {
    return this;
  }
};

someObject.returnMyThis() === someObject
//=> true
```

We’ve constructed a method that returns whatever value is bound to `this` when it is called. It returns the object when called, just as described.

it's all about the way the function is called

JavaScript programmers talk about functions having a “context” when being called. `this` is bound to the context.⁹⁸ The important thing to understand is that the context for a function being called is set by the way the function is called, not the function itself.

This is an important distinction. Consider closures: As we discussed in [Closures and Scope](#), a function’s free variables are resolved by looking them up in their enclosing functions’ environments. You can always determine the functions that define free variables by examining the source code of a JavaScript program, which is why this scheme is known as [Lexical Scope](#)⁹⁹.

A function’s context cannot be determined by examining the source code of a JavaScript program. Let’s look at our example again:

⁹⁸Too bad the language binds the context to the name `this` instead of the name `context`!

⁹⁹[https://en.wikipedia.org/wiki/Scope_\(computer_science\)#Lexical_scoping](https://en.wikipedia.org/wiki/Scope_(computer_science)#Lexical_scoping)

```
const someObject = {
  someFunction () {
    return this;
  }
};

someObject.someFunction() === someObject
//=> true
```

What is the context of the function `someObject.someFunction`? Don't say `someObject`! Watch this:

```
const someFunction = someObject.someFunction;

someFunction === someObject.someFunction
//=> true

someFunction() === someObject
//=> false
```

It gets weirder:

```
const anotherObject = {
  someFunction: someObject.someFunction
}

anotherObject.someFunction === someObject.someFunction
//=> true

anotherObject.someFunction() === anotherObject
//=> true

anotherObject.someFunction() === someObject
//=> false
```

So it amounts to this: The exact same function can be called in two different ways, and you end up with two different contexts. If you call it using `someObject.someFunction()` syntax, the context is set to the receiver. If you call it using any other expression for resolving the function's value (such as `someFunction()`), you get something else.

Let's investigate:

```
(someObject.someFunction)() === someObject
//=> true

someObject['someFunction']() === someObject
//=> true

const name = 'someFunction';

someObject[name]() === someObject
//=> true
```

Interesting!

```
let baz;

(baz = someObject.someFunction)() === this
//=> true
```

How about:

```
const arr = [ someObject.someFunction ];

arr[0]() === arr
//=> true
```

It seems that whether you use `a.b()` or `a['b']()` or `a[n]()` or `(a.b)()`, you get context `a`.

```
const returnThis = function () { return this };

const aThirdObject = {
  someFunction () {
    return returnThis()
  }
}

returnThis() === this
//=> true

aThirdObject.someFunction() === this
//=> true
```

And if you don't use `a.b()` or `a['b']()` or `a[n]()` or `(a.b)()`, you get the global environment for a context, not the context of whatever function is doing the calling. To simplify things, when you call a function with `.` or `[]` access, you get an object as context, otherwise you get the global environment.

setting your own context

There are actually two other ways to set the context of a function. And once again, both are determined by the caller. At the very end of [objects everywhere?](#), we'll see that everything in JavaScript behaves like an object, including functions. We'll learn that functions have methods themselves, and one of them is `call`.

Here's `call` in action:

```
returnThis() === aThirdObject
//=> false

returnThis.call(aThirdObject) === aThirdObject
//=> true

anotherObject.someFunction.call(someObject) === someObject
//=> true
```

When You call a function with `call`, you set the context by passing it in as the first parameter. Other arguments are passed to the function in the normal manner. Much hilarity can result from `call` shenanigans like this:

```
const a = [1,2,3],
      b = [4,5,6];

a.concat([2,1])
//=> [1,2,3,2,1]

a.concat.call(b,[2,1])
//=> [4,5,6,2,1]
```

But now we thoroughly understand what `a.b()` really means: It's synonymous with `a.b.call(a)`. Whereas in a browser, `c()` is synonymous with `c.call(window)`.

arguments

JavaScript has another automagic binding in every function's environment. `arguments` is a special object that behaves a little like an array.¹⁰⁰

For example:

¹⁰⁰Just enough to be frustrating, to be perfectly candid!

```
const third = function () {
    return arguments[2]
}
```

```
third(77, 76, 75, 74, 73)
//=> 75
```

Gathering arguments with ... accomplishes most of the use cases people have for using the arguments special binding, and in addition, gathering works with both fat arrows and with the function keyword, whereas arguments only works with the function keyword.

There are a few things that arguments can do that gathering cannot do, for example if you declare a function with `function (a, b, c) { ... }`, arguments holds the arguments passed to the function even though you haven't declared a parameter to be gathered. It works alongside the declared parameters.

But by and large, we will gather parameters in this book.

application and contextualization

Hold that thought for a moment. JavaScript also provides a fourth way to set the context for a function. `apply` is a method implemented by every function that takes a context as its first argument, and it takes an array or array-like thing of arguments as its second argument. That's a mouthful, let's look at an example:

```
third.call(this, 1,2,3,4,5)
//=> 3
```

```
third.apply(this, [1,2,3,4,5])
//=> 3
```

Now let's put the two together. Here's another travesty:

```
const a = [1,2,3],
      accrete = a.concat;

accrete([4,5])
//=> Gobbledygook!
```

We get the result of concatenating [4,5] onto an array containing the global environment. Not what we want! Behold:

```
const contextualize = (fn, context) =>
  (...args) =>
    fn.apply(context, args)

const accrete2 = contextualize(a.concat, a);
accrete2([4,5]);
//=> [ 1, 2, 3, 4, 5 ]
```

Our contextualize function returns a new function that calls a function with a fixed context. It can be used to fix some of the unexpected results we had above. Consider:

```
var aFourthObject = {},
  returnThis = function () { return this; }

aFourthObject.uncontextualized = returnThis;
aFourthObject.contextualized = contextualize(returnThis, aFourthObject);

aFourthObject.uncontextualized() === aFourthObject
//=> true
aFourthObject.contextualized() === aFourthObject
//=> true
```

Both are true because we are accessing them with aFourthObject. Now we write:

```
var uncontextualized = aFourthObject.uncontextualized,
  contextualized = aFourthObject.contextualized;

uncontextualized() === aFourthObject;
//=> false
contextualized() === aFourthObject
//=> true
```

When we call these functions without using aFourthObject., only the contextualized version maintains the context of aFourthObject.

We'll return to contextualizing methods later, in [Binding](#). But before we dive too deeply into special handling for methods, we need to spend a little more time looking at how functions and methods work.

Method Decorators

In [function decorators](#), we learned that a decorator takes a function as an argument, returns a function, and there's a semantic relationship between the two. If a function is a verb, a decorator is an adverb.

Decorators can be used to decorate methods provided that they carefully preserve the function's context. For example, here is a naïve version of `maybe` for one argument:

```
const maybe = (fn) =>
  x => x != null ? fn(x) : x;
```

We use it like this:

```
const plus1 = x => x + 1;

plus1(1)
  //=> 2
plus1(0)
  //=> 1
plus1(null)
  //=> 1
plus1(undefined)
  //=> null

const maybePlus1 = maybe(plus1);

maybePlus1(1)
  //=> 2
maybePlus1(0)
  //=> 1
maybePlus1(null)
  //=> null
maybePlus1(undefined)
  //=> undefined
```

This version doesn't preserve the context, so it can't be used as a method decorator. Instead, we have to convert the decoration from a fat arrow to a function function:

```
const maybe = (fn) =>
  function (x) {
    return x != null ? fn(x) : x;
  };

```

And then use `.call` to preserve this:

```
const maybe = (fn) =>
  function (x) {
    return x != null ? fn.call(this, x) : x;
  };

```

Now that we have a “proper function,” we can also handle variadic functions and methods. This variation only invokes the decorated function if none of the arguments are `null` or `undefined`:

```
const maybe = (fn) =>
  function (...args) {
    for (let i in args) {
      if (args[i] == null) return args[i];
    }
    return fn.apply(this, args);
  };

```

But back to basics. As long as we are correctly preserving `this` by one, using a function, and two, invoking the decorated function with `.call(this, ...)` or `.apply(this, ...)`, we can decorate methods as well as functions.

Now we can write things like:

```
const someObject = {
  setSize: maybe(function (size) {
    this.size = size;
  })
}

```

And `this` is correctly set:

```
someObject.setSize(5);
someObject
//=> { setSize: [Function], size: 5 }

someObject.setSize(null);
someObject
//=> { setSize: [Function], size: 5 }
```

Using `.call` or `.apply` and `arguments` is substantially slower than writing function decorators that don't set the context, so it might be right to sometimes write function decorators that aren't usable as method decorators. However, in practice you're far more likely to introduce a defect by failing to pass the context through a decorator than by introducing a performance pessimization, so the default choice should be to write all function decorators in such a way that they are "context agnostic."

In some cases, there are other considerations to writing a method decorator. If the decorator introduces state of any kind (such as `once` and `memoize do`), this must be carefully managed for the case when several objects share the same method through the mechanism of the `prototype` or through sharing references to the same function.

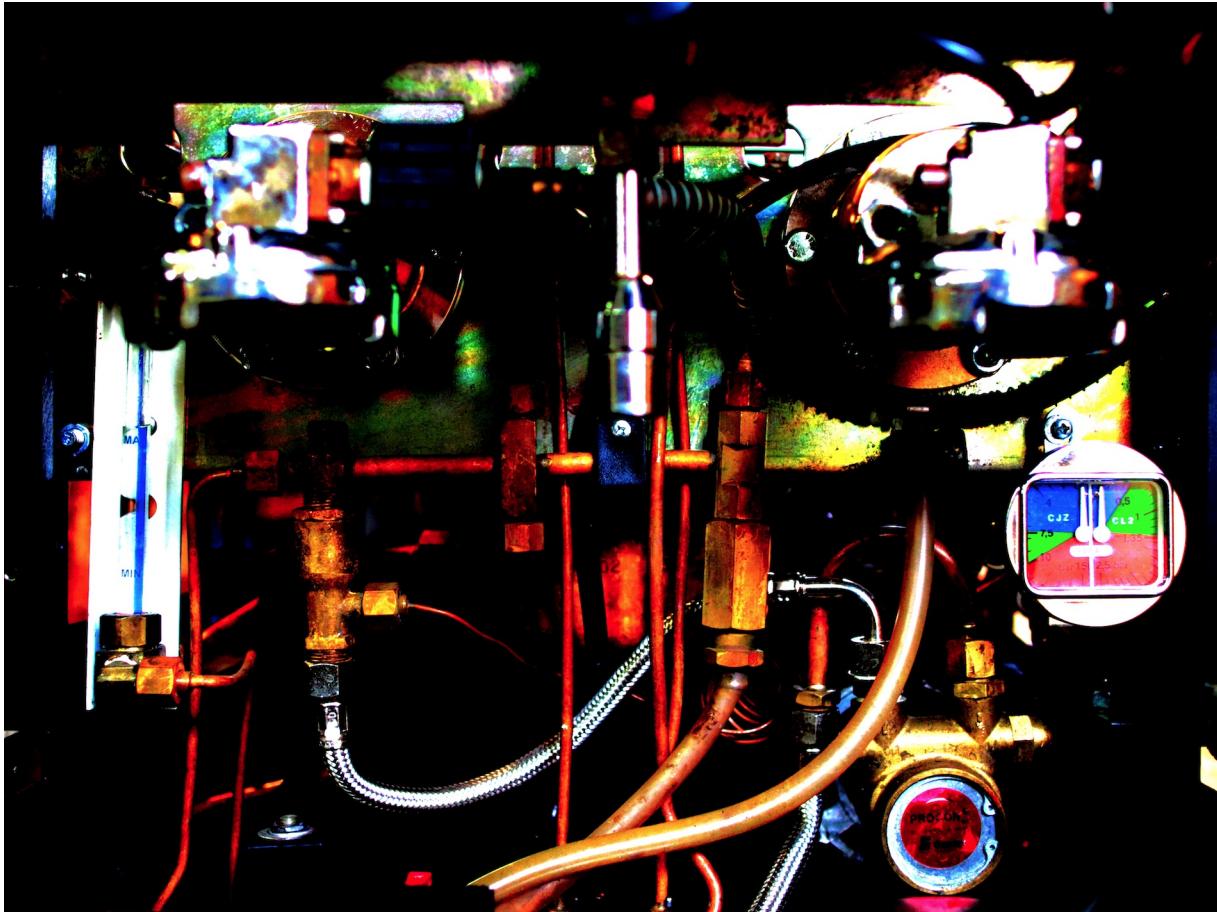
Summary



Objects, Mutation, and State

- State can be encapsulated/hidden with closures.
- Encapsulations can be aggregated with composition.
- Encapsulation resists extension.
- The automagic binding `this` facilitates sharing of functions.
- Functions can be named and declared with a name.

Recipes with Objects, Mutations, and State



The Intestines of an Espresso Machine

Disclaimer

The recipes are written for practicality, and their implementation may introduce JavaScript features that haven't been discussed in the text to this point, such as methods and/or prototypes. The overall *use* of each recipe will fit within the spirit of the language discussed so far, even if the implementations may not.

Memoize

Consider that age-old interview quiz, writing a recursive fibonacci function (there are other ways to derive a fibonacci number, of course). Here's an implementation that doesn't use a [named function expression](#). The reason for that omission will be explained later:

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

[0,1,2,3,4,5,6,7,8].map(fibonacci)
//=> [0,1,1,2,3,5,8,13,21]
```

We'll time it:

```
s = (new Date()).getTime()
fibonacci(45)
( (new Date()).getTime() - s ) / 1000
//=> 15.194
```

Why is it so slow? Well, it has a nasty habit of recalculating the same results over and over and over again. We could rearrange the computation to avoid this, but let's be lazy and trade space for time. What we want to do is use a lookup table. Whenever we want a result, we look it up. If we don't have it, we calculate it and write the result in the table to use in the future. If we do have it, we return the result without recalculating it.

Here's our recipe:

```
const memoized = (fn) => {
  const lookupTable = [];

  return function (...args) {
    const key = JSON.stringify(this, args);

    return lookupTable[key] || (lookupTable[key] = fn.apply(this, args));
  }
}
```

We can apply `memoized` to a function and we will get back a new function that "memoizes" its results so that it never has to recalculate the same value twice. It only works for functions that are "idempotent," meaning functions that always return the same result given the same argument(s). Like `fibonacci`:

Let's try it:

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

fastFibonacci(45)
//=> 1134903170
```

We get the result back instantly. It works! You can use `memoize` with all sorts of “idempotent” pure functions. by default, it works with any function that takes arguments which can be transformed into JSON using JavaScript’s standard library function for this purpose.

If you have another strategy for turning the arguments into a string key, we’ll need to make a version that allows you to supply an optional `keymaker` function:

```
const memoized = (fn, keymaker = JSON.stringify) => {
  const lookupTable = {};

  return function (...args) {
    const key = keymaker.apply(this, args);

    return lookupTable[key] || (lookupTable[key] = fn.apply(this, args));
  }
}
```

memoizing recursive functions

We deliberately picked a recursive function to memoize, because it demonstrates a pitfall when combining decorators with named functional expressions. Consider this implementation that uses a named functional expression:

```
var fibonacci = function fibonacci (n) {
  if (n < 2) {
    return n
  }
  else {
    return fibonacci(n-2) + fibonacci(n-1)
  }
}
```

If we try to memoize it, we don’t get the expected speedup:

```
var fibonacci = memoized( function fibonacci (n) {
  if (n < 2) {
    return n
  }
  else {
    return fibonacci(n-2) + fibonacci(n-1)
  }
});
```

That's because the function bound to the name `fibonacci` in the outer environment has been memoized, but the named functional expression binds the name `fibonacci` inside the unmemoized function, so none of the recursive calls to `fibonacci` are *ever* memoized. Therefore we must write:

```
var fibonacci = memoized( function (n) {
  if (n < 2) {
    return n
  }
  else {
    return fibonacci(n-2) + fibonacci(n-1)
  }
});
```

If we need to prevent a rebinding from breaking the function, we'll need to use the [module](#) pattern.

getWith

`getWith` is a very simple function. It takes the name of an attribute and returns a function that extracts the value of that attribute from an object:

```
const getWith = (attr) => (object) => object[attr]
```

You can use it like this:

```
const inventory = {
  apples: 0,
  oranges: 144,
  eggs: 36
};

getWith('oranges')(inventory)
//=> 144
```

This isn't much of a recipe yet. But let's combine it with `mapWith`:

```
const inventories = [
  { apples: 0, oranges: 144, eggs: 36 },
  { apples: 240, oranges: 54, eggs: 12 },
  { apples: 24, oranges: 12, eggs: 42 }
];

mapWith(getWith('oranges'))(inventories)
//=> [ 144, 54, 12 ]
```

That's nicer than writing things out "longhand:"

```
mapWith((inventory) => inventory.oranges)(inventories)
//=> [ 144, 54, 12 ]
```

`getWith` plays nicely with `maybe` as well. Consider a sparse array. You can use:

```
mapWith(maybe(getWith('oranges')))
```

To get the orange count from all the non-null inventories in a list.

what's in a name?

Why is this called `getWith`? Consider this function that is common in languages that have functions and dictionaries but not methods:

```
const get = (object, attr) => object[attr];
```

You might ask, “Why use a function instead of just using []?” The answer is, we can manipulate functions in ways that we can’t manipulate syntax. For example, do you remember from `flip` that we can define `mapWith` from `map`?

```
var mapWith = flip(map);
```

We can do the same thing with `getWith`, and that’s why it’s named in this fashion:

```
var getWith = flip(get)
```

pluckWith

This pattern of combining `mapWith` and `getWith` is very frequent in JavaScript code. So much so, that we can take it up another level:

```
const pluckWith = (attr) => mapWith(getWith(attr));
```

Or even better:

```
const pluckWith = compose(mapWith, getWith);
```

And now we can write:

```
const inventories = [
  { apples: 0, oranges: 144, eggs: 36 },
  { apples: 240, oranges: 54, eggs: 12 },
  { apples: 24, oranges: 12, eggs: 42 }
];

pluckWith('eggs')(inventories)
//=> [ 36, 12, 42 ]
```

Libraries like `Underscore`¹⁰¹ provide `pluck`, the flipped version of `pluckWith`:

```
_pluck(inventories, 'eggs')
//=> [ 36, 12, 42 ]
```

Our recipe is terser when you want to name a function:

```
const eggsByStore = pluckWith('eggs');
```

vs.

```
const eggsByStore = (inventories) =>
  _pluck(inventories, 'eggs');
```

And of course, if we have `pluck` we can use `flip` to derive `pluckWith`:

¹⁰¹<http://underscorejs.org>

```
const pluckWith = flip(_.pluck);
```

Deep Mapping

`mapWith` is an excellent tool, but from time to time you will find yourself working with arrays that represent trees rather than lists. For example, here is a partial list of sales extracted from a report of some kind. It's grouped in some mysterious way, and we need to operate on each item in the report.

```
const report =
  [ [ { price: 1.99, id: 1 },
    { price: 4.99, id: 2 },
    { price: 7.99, id: 3 },
    { price: 1.99, id: 4 },
    { price: 2.99, id: 5 },
    { price: 6.99, id: 6 } ],
  [ { price: 5.99, id: 21 },
    { price: 1.99, id: 22 },
    { price: 1.99, id: 23 },
    { price: 1.99, id: 24 },
    { price: 5.99, id: 25 } ],
  // ...
  [ { price: 7.99, id: 221 },
    { price: 4.99, id: 222 },
    { price: 7.99, id: 223 },
    { price: 10.99, id: 224 },
    { price: 9.99, id: 225 },
    { price: 9.99, id: 226 } ] ];
```

We could nest some `mapWith`s, but we humans are tool users. If we can use a stick to extract tasty ants from a hole to eat, we can automate working with arrays:

```
const deepMapWith = (fn) =>
  function innerdeepMapWith (tree) {
    return Array.prototype.map.call(tree, (element) =>
      Array.isArray(element)
        ? innerdeepMapWith(element)
        : fn(element));
  };
};
```

And now we can use `deepMapWith` on a tree the way we use `mapWith` on a flat array:

```
deepMapWith(getWith('price'))(report)
//=> [ [ 1.99,
  4.99,
  7.99,
  1.99,
  2.99,
  6.99 ],
[ 5.99,
  1.99,
  1.99,
  1.99,
  5.99 ],

// ...
[ 7.99,
  4.99,
  7.99,
  10.99,
  9.99,
  9.99 ] ]
```

We'll have another look at trees of data when we look at TreeIterators for [Collections](#).

The Coffee Factory: “Object-Oriented Programming”

Programming with objects and classes began in Norway in the late 1960s with the [Simula¹⁰²](#) programming language. Its creators, Ole-Johan Dahl and Kristen Nygaard, did not use those words to describe what would eventually become the dominant paradigm in computing.

A decade later, Dr. Alan Kay coined the phrase “Object-Oriented Programming” along with co-creating the [Smalltalk¹⁰³](#) programming language. He has famously said that to him, “OOP” was objects communicating with each other using messages, and that other languages copied the things that didn’t matter from Smalltalk, and ignored the things he thought did matter.

Since that time, languages have either bolted object-ish ideas on top of their existing paradigms (like [Object Pascal¹⁰⁴](#) and [OCaml¹⁰⁵](#)), baked them in alongside other paradigms (like JavaScript), or embraced objects wholeheartedly.

That being said, there really is no one definition of “object-oriented.” For one thing, there is no one definition of “object.”

objects

Some languages, like Smalltalk and [Ruby¹⁰⁶](#), treat an object as a fully encapsulated entity. There is no access to an object’s private state, all you can do is invoke one of its methods. Other languages, like Java, permit objects to access each other’s state.

Some languages (again, like Java) have very rigid objects and classes, it is impossible or awkward to add new methods or properties to objects at run time. Some are flexible about adding methods and properties at run time. And yet other languages treat objects as dictionaries, where properties and even methods can be added, modified, or removed with abandon.

So we can see that the concept of “object” is flexible across languages.

classes

The concept of “class” is also flexible across languages. Object-oriented languages do not uniformly agree on whether classes are necessary, much less how they work. For example, The Common Lisp

¹⁰²<https://en.wikipedia.org/wiki/Simula>

¹⁰³<https://en.wikipedia.org/wiki/Smalltalk>

¹⁰⁴https://en.wikipedia.org/wiki/Object_Pascal

¹⁰⁵<https://en.wikipedia.org/wiki/OCaml>

¹⁰⁶https://en.wikipedia.org/wiki/Ruby_

Object System defines behaviour with classes, and it also defines behaviour with generic functions. The [Self¹⁰⁷](#) and [NewtonScript¹⁰⁸](#) languages have prototypes instead of classes.

So some “OO” languages have objects, but not classes.

C++ has classes, but they are not “first-class entities.” You can’t assign a class to a variable or pass it to a function. You can, however, manipulate the constructors for classes, the functions that make new objects. But you can’t manipulate those constructors to change the behaviour of objects that have already been constructed, instance behaviour is early-bound by default.

Ruby has classes, and they’re first-class entities. You can ask an object for its class, you can put a class in a variable, pass it to a method, or return it from a method, just like every other entity in the language. Classes in Ruby and Smalltalk even have their own class, they are instances of `Class`¹⁰⁹. Instance behaviour is late-bound and open for extension.¹¹⁰

constructors

Some languages allow programs to construct objects independently, others (notably those that are heavily class-centric) require that objects always be constructed by their classes. Some languages allow any function or method to be used as a constructor, others require a special syntax or declaration for constructors.

prototypes are not classes

Prototypical languages like Self and NewtonScript eschew classes altogether, using *prototypes* to define common behaviour for a set of objects. The difference between a prototype and a class is similar to the difference between a model home and a blueprint for a home.

You can say to a builder, “make me a home just like that model home,” and the builder makes you a home that has a lot in common with the model home. You then decorate your home with additional personalization. But the model home is, itself, a home. Although you may choose to keep it empty, you could in principle move a family into it. This is different than asking a builder to make you a home based on a blueprint. The blueprint may specify the features of the home, but it isn’t a home. It could never be used as a home.

Prototypes are like model homes, and classes are like blueprints. Classes are not like the objects they describe.¹¹¹

¹⁰⁷https://en.wikipedia.org/wiki/Self_

¹⁰⁸<https://en.wikipedia.org/wiki/NewtonScript>

¹⁰⁹If the class of a class is `Class`, what class is the class of `Class`? In Ruby, `Class.class == Class`. In Smalltalk, it is `MetaClass`, which opens up the possibility for changing the way classes behave in a deep way.

¹¹⁰Abuse of this feature by extending the behaviour of built-in classes is a controversial topic.

¹¹¹Well, *actually*, the difference between prototypes and classes is like the difference between model homes and blueprints. But prototypes are not like model homes. In actual fact, the relationship between an object and its prototype is one of *delegation*. So if a model home had a kitchen, and you asked the builder to make you a home using the model as a prototype, you could customize your own kitchen. But if you didn’t want to have your own custom kitchen, you would just use the model home’s kitchen to do all your own cooking. The relationship between a model home and a house is sometimes described as [concatenative inheritance](#), and JavaScript lets you do that too.

"object-oriented programming" can mean almost anything

From this whirlwind tour of "object-oriented programming," we can see that the ideas behind "object-oriented programming" have some common roots in the history of programming languages, but each language implements its own particular flavour in its own particular way.

Thus, when we talk about "objects" and "prototypes" and "classes" in JavaScript, we're talking about objects, prototypes, and classes *as implemented in JavaScript*. And we must keep in mind that other languages can have a radically different take on these ideas.

the javascript approach

JavaScript has objects, and by default, those objects are dictionaries. By default, objects directly manipulate each other's state. Methods can be added to, or removed from objects at run time.

JavaScript has optional prototypes. Prototypes are objects in the same sense that model homes are homes.

In JavaScript, object and array literals construct objects that delegate behaviour to the standard library's object prototype and array prototype, respectively. JavaScript also supports using `Object.create` to construct objects with or without a prototype, and `new` to construct objects using a constructor function.

Using prototypes and constructor functions, JavaScript programs can emulate many of the features of classes in other languages. JavaScript also has a `class` keyword that provides syntactic sugar for writing constructor functions and prototypes in a declarative fashion.

By default, a JavaScript class is a constructor composed with an object as its associated prototype. This can be denoted with the `class` keyword, by working with a function's default `.prototype` property, or by composing functions and objects independently.

JavaScript classes are constructors, but they are more than C++ constructors, in that manipulation of their prototype extends or modifies the behaviour of the instances they create. JavaScript classes take a minimalist approach to OO in the same sense that JavaScript objects take a minimal approach to OO. For example, behaviour can be mixed into an object, a prototype, or a class using the exact same mechanism, because objects, prototypes, and a constructor's prototype are all objects that are open to extension.

In sum, JavaScript is not exactly like any other object-oriented programming language, and its classes aren't like any other language that features classes, but then again, neither is any other object-oriented programming language, and neither are any other classes.

Served by the Pot: Collections



Some different sized and coloured coffee pots by Antti Nurmesniemi, perhaps his most known design.

Iteration and Iterables



Coffee Labels at the Saltspring Coffee Processing Facility

Many objects in JavaScript can model collections of things. A collection is like a box containing stuff. Sometimes you just want to move the box around. But sometimes you want to open it up and do things with its contents.

Things like “put a label on every bag of coffee in this box,” Or, “Open the box, take out the bags of decaf, and make a new box with just the decaf.” Or, “go through the bags in this box, and take out the first one marked ‘Espresso’ that contains at least 454 grams of beans.”

All of these actions involve going through the contents one by one. Acting on the elements of a collection one at a time is called *iterating over the contents*, and JavaScript has a standard way to iterate over the contents of collections.

a look back at functional iterators

When discussing functions, we looked at the benefits of writing [Functional Iterators](#). We can do the same thing for objects. Here’s a stack that has its own functional iterator method:

```
const Stack1 = () =>
  ({
    array:[],
    index: -1,
    push (value) {
      return this.array[this.index += 1] = value;
    },
    pop () {
      const value = this.array[this.index];

      this.array[this.index] = undefined;
      if (this.index >= 0) {
        this.index -= 1
      }
      return value
    },
    isEmpty () {
      return this.index < 0
    },
    iterator () {
      let iterationIndex = this.index;

      return () => {
        if (iterationIndex > this.index) {
          iterationIndex = this.index;
        }
        if (iterationIndex < 0) {
          return {done: true};
        }
        else {
          return {done: false, value: this.array[iterationIndex--]}
        }
      }
    }
  });
}

const stack = Stack1();

stack.push("Greetings");
stack.push("to");
stack.push("you!")
```

```
const iter = stack.iterator();
iter().value
//=> "you!"
iter().value
//=> "to"
```

The way we've written `.iterator` as a method, each object knows how to return an iterator for itself.

The `.iterator()` method is defined with shorthand equivalent to `function iterator() { ... }`. Note that it uses the `function` keyword, so when we invoke it with `stack.iterator()`, JavaScript sets `this` to the value of `stack`. But what about the function `.iterator()` returns? It is defined with a fat arrow `() => { ... }`. What is the value of `this` within that function?

Since JavaScript doesn't bind `this` within a fat arrow function, we follow the same rules of variable scoping as any other variable name: We check in the environment enclosing the function. Although the `.iterator()` method has returned, its environment is the one that encloses our `() => { ... }` function, and that's where `this` is bound to the value of `stack`.

Therefore, the iterator function returned by the `.iterator()` method has `this` bound to the `stack` object, even though we call it with `iter()`.

And here's a `sum` function implemented as a fold over a functional iterator:

```
const iteratorSum = (iterator) => {
  let eachIteration,
    sum = 0;

  while ((eachIteration = iterator()), !eachIteration.done) {
    sum += eachIteration.value;
  }
  return sum
}
```

We can use it with our stack:

```
const stack = Stack1();

stack.push(1);
stack.push(2);
stack.push(3);

iteratorSum(stack.iterator())
//=> 6
```

We could save a step and write `collectionSum`, a function that folds over any object, provided that the object implements an `.iterator` method:

```
const collectionSum = (collection) => {
  const iterator = collection.iterator();

  let eachIteration,
    sum = 0;

  while ((eachIteration = iterator()), !eachIteration.done)) {
    sum += eachIteration.value;
  }
  return sum
}

collectionSum(stack)
//=> 6
```

If we write a program with the presumption that “everything is an object,” we can write maps, folds, and filters that work on objects. We just ask the object for an iterator, and work on the iterator. Our functions don’t need to know anything about how an object implements iteration, and we get the benefit of lazily traversing our objects.

This is a good thing.

iterator objects

Iteration for functions and objects has been around for many, many decades. For simple linear collections like arrays, linked lists, stacks, and queues, functional iterators are the simplest and easiest way to implement iterators.

In programs involving large collections of objects, it can be handy to implement iterators as objects, rather than functions. The mechanics of iterating can then be factored using the same tools that are used to factor the mechanics of all other objects in the system.

Fortunately, an iterator object is almost as simple as an iterator function. Instead of having a function that you call to get the next element, you have an object with a `.next()` method.

Like this:

```
const Stack2 = () =>
  ({
    array: [],
    index: -1,
    push (value) {
      return this.array[this.index += 1] = value;
    },
    pop () {
      const value = this.array[this.index];

      this.array[this.index] = undefined;
      if (this.index >= 0) {
        this.index -= 1
      }
      return value
    },
    isEmpty () {
      return this.index < 0
    },
    iterator () {
      let iterationIndex = this.index;

      return {
        next () {
          if (iterationIndex > this.index) {
            iterationIndex = this.index;
          }
          if (iterationIndex < 0) {
            return {done: true};
          }
          else {
            return {done: false, value: this.array[iterationIndex--]}
          }
        }
      }
    }
  });
});
```

```

const stack = Stack2();

stack.push(2000);
stack.push(10);
stack.push(5)

const collectionSum = (collection) => {
  const iterator = collection.iterator();

  let eachIteration,
    sum = 0;

  while ((eachIteration = iterator.next(), !eachIteration.done)) {
    sum += eachIteration.value;
  }
  return sum
}

collectionSum(stack)
//=> 2015

```

Now our `.iterator()` method is returning an iterator object. When working with objects, we do things the object way. But having started by building functional iterators, we understand what is happening underneath the object's scaffolding.

iterables

People have been writing iterators since JavaScript was first released in the late 1990s. Since there was no particular standard way to do it, people used all sorts of methods, and their methods returned all sorts of things: Objects with various interfaces, functional iterators, you name it.

So, when a standard way to write iterators was added to the JavaScript language, it didn't make sense to use a method like `.iterator()` for it: That would conflict with existing code. Instead, the language encourages new code to be written with a different name for the method that a collection object uses to return its iterator.

To ensure that the method would not conflict with any existing code, JavaScript provides a *symbol*. Symbols are unique constants that are guaranteed not to conflict with existing strings. Symbols are a longstanding technique in programming going back to Lisp, where the `GENDSYM` function generated... You guessed it... Symbols.¹¹²

The expression `Symbol.iterator` evaluates to a special symbol representing the name of the method that objects should use if they return an iterator object.

¹¹²You can read more about JavaScript symbols in Axel Rauschmayer's [Symbols in ECMAScript 2015](#).

Our stack does, so instead of binding the existing iterator method to the name `iterator`, we bind it to the `Symbol.iterator`. We'll do that using the `[]` syntax for using an expression as an object literal key:

```
const Stack3 = () =>
  ({
    array: [],
    index: -1,
    push (value) {
      return this.array[this.index += 1] = value;
    },
    pop () {
      const value = this.array[this.index];

      this.array[this.index] = undefined;
      if (this.index >= 0) {
        this.index -= 1
      }
      return value
    },
    isEmpty () {
      return this.index < 0
    },
    [Symbol.iterator] () {
      let iterationIndex = this.index;

      return {
        next () {
          if (iterationIndex > this.index) {
            iterationIndex = this.index;
          }
          if (iterationIndex < 0) {
            return {done: true};
          }
          else {
            return {done: false, value: this.array[iterationIndex--]}
          }
        }
      }
    }
  });
}

const stack = Stack3();
```

```

stack.push(2000);
stack.push(10);
stack.push(5)

const collectionSum = (collection) => {
  const iterator = collection[Symbol.iterator]();

  let eachIteration,
    sum = 0;

  while ((eachIteration = iterator.next(), !eachIteration.done)) {
    sum += eachIteration.value;
  }
  return sum
}

collectionSum(stack)
//=> 2015

```

Using [Symbol.iterator] instead of .iterator seems like adding an extra moving part for nothing. Do we get anything in return?

Indeed we do. Behold the for...of loop:

```

const iterableSum = (iterable) => {
  let sum = 0;

  for (let num of iterable) {
    sum += num;
  }
  return sum
}

iterableSum(stack)
//=> 2015

```

The for...of loop works directly with any object that is *iterable*, meaning it works with any object that has a Symbol.iterator method that returns a object iterator. Here's another linked list, this one is iterable:

```
const EMPTY = {
  isEmpty: () => true
};

const isEmpty = (node) => node === EMPTY;

const Pair1 = (first, rest = EMPTY) =>
({
  first,
  rest,
  isEmpty () { return false },
  [Symbol.iterator] () {
    let currentPair = this;

    return {
      next () {
        if (currentPair.isEmpty()) {
          return {done: true}
        }
        else {
          const value = currentPair.first;

          currentPair = currentPair.rest;
          return {done: false, value}
        }
      }
    }
  }
});

const list = (...elements) => {
  const [first, ...rest] = elements;

  return elements.length === 0
    ? EMPTY
    : Pair1(first, list(...rest))
}

const someSquares = list(1, 4, 9, 16, 25);

iterableSum(someSquares)
//=> 55
```

As we can see, we can use `for...of` with linked lists just as easily as with stacks. And there's one more thing: You recall that the spread operator (`...`) can spread the elements of an array in an array literal or as parameters in a function invocation.

Now is the time to note that we can spread any iterable. So we can spread the elements of an iterable into an array literal:

```
[ 'some squares', ...someSquares]
//=> [ "some squares", 1, 4, 9, 16, 25]
```

And we can also spread the elements of an array literal into parameters:

```
const firstAndSecondElement = (first, second) =>
  ({first, second})

firstAndSecondElement(...stack)
//=> { "first":5, "second":10}
```

This can be extremely useful.

One caveat of spreading iterables: JavaScript creates an array out of the elements of the iterable. That might be very wasteful for extremely large collections. For example, if we spread a large collection just to find an element in the collection, it might have been wiser to iterate over the element using its iterator directly.

And if we have an infinite collection, spreading is going to fail outright.

iterables out to infinity

Iterables needn't represent finite collections:

```
const Numbers = {
  [Symbol.iterator] () {
    let n = 0;

    return {
      next: () =>
        ({done: false, value: n++})
    }
  }
}
```

There are useful things we can do with iterables representing an infinitely large collection. But let's point out what we can't do with them:

```
[ 'all the numbers', ...Numbers]
//=> infinite loop!

firstAndSecondElement(...Numbers)
//=> infinite loop!
```

Attempting to spread an infinite iterable into an array is always going to fail.

ordered collections

The iterables we're discussing represent *ordered collections*. One of the semantic properties of an ordered collection is that every time you iterate over it, you get its elements in order, from the beginning. For example:

```
const abc = ["a", "b", "c"];

for (let i of abc) {
  console.log(i)
}
//=>
a
b
c

for (let i of abc) {
  console.log(i)
}
//=>
a
b
c
```

This is accomplished with our own collections by returning a brand new iterator every time we call `[Symbol.iterator]`, and ensuring that our iterators start at the beginning and work forward.

Iterables needn't represent ordered collections. We could make an infinite iterable representing random numbers:

```
const RandomNumbers = {
  [Symbol.iterator]: () =>
  ({
    next () {
      return {value: Math.random()};
    }
  })
}

for (let i of RandomNumbers) {
  console.log(i)
}
//=>
0.494052127469331
0.835459444206208
0.1408337657339871
...
for (let i of RandomNumbers) {
  console.log(i)
}
//=>
0.7845381607767195
0.4956772483419627
0.20259276474826038
...
```

Whether you work with the same iterator over and over, or get a fresh iterable every time, you are always going to get fresh random numbers. Therefore, `RandomNumbers` is not an ordered collection.

Right now, we're just looking at ordered collections. To reiterate (hah), an ordered collection represents a (possibly infinite) collection of elements that are in some order. Every time we get an iterator from an ordered collection, we start iterating from the beginning.

operations on ordered collections

Let's define some operations on ordered collections. Here's `mapIterableWith`, it takes an ordered collection, and returns another ordered collection representing a mapping over the original:

```
const mapIterableWith = (fn, collection) =>
  ({
    [Symbol.iterator] () {
      const iterator = collection[Symbol.iterator]();
      return {
        next () {
          const {done, value} = iterator.next();

          return ({done, value: done ? undefined : fn(value)});
        }
      }
    });
  });
});
```

This illustrates the general pattern of working with ordered collections: We make them *iterables*, meaning that they have a `[Symbol.iterator]` method, that returns an *iterator*. An iterator is also an object, but with a `.next()` method that is invoked repeatedly to obtain the elements in order.

Many operations on ordered collections return another ordered collection. They do so by taking care to iterate over a result freshly every time we get an iterator for them. Consider this example for `mapIterableWith`:

```
const Evens = mapIterableWith((x) => 2 * x, Numbers);

for (let i of Evens) {
  console.log(i)
}
//=>
0
2
4
...
for (let i of Evens) {
  console.log(i)
}
//=>
0
2
4
...

```

Numbers is an ordered collection. We invoke `mapIterableWith((x) => 2 * x, Numbers)` and get Evens. Evens works just as if we'd written this:

```
const Evens = [
  [Symbol.iterator] () {
    const iterator = Numbers[Symbol.iterator]();

    return {
      next () {
        const {done, value} = iterator.next();

        return ({done, value: done ? undefined : 2 * value});
      }
    }
  }
];
```

Every time we write `for (let i of Evens)`, JavaScript calls `Evens[Symbol.iterator]()`. That in turns means it executes `const iterator = Numbers[Symbol.iterator]();` every time we write `for (let i of Evens)`, and that means that iterator starts at the beginning of Numbers.

So, Evens is also an ordered collection, because it starts at the beginning each time we get a fresh iterator over it. Thus, `mapIterableWith` has the property of preserving the collection semantics of the iterable we give it. So we call it a *collection operation*.

Mind you, we can also map non-collection iterables, like `RandomNumbers`:

```
const ZeroesToNines = mapIterableWith((n) => Math.floor(10 * limit), RandomNumbers);

for (let i of ZeroesToNines) {
  console.log(i)
}
//=>
5
1
9
...
for (let i of ZeroesToNines) {
  console.log(i)
}
//=>
```

```
3
6
1
...
...
```

`mapIterableWith` can get a new iterator from `RandomNumbers` each time we iterate over `ZeroesToNines`, but if `RandomNumbers` doesn't behave like an ordered collection, that's not `mapIterableWith`'s fault. `RandomNumbers` is a *stream*, not an ordered collection, and thus `mapIterableWith` returns another iterable behaving like a stream.

Here are two more operations on ordered collections, `filterIterableWith` and `untilIterableWith`:

```
const filterIterableWith = (fn, iterable) =>
  ({
    [Symbol.iterator] () {
      const iterator = iterable[Symbol.iterator]();

      return {
        next () {
          do {
            const {done, value} = iterator.next();
            } while (!done && !fn(value));
            return {done, value};
          }
        }
      });
    });

const untilIterableWith = (fn, iterable) =>
  ({
    [Symbol.iterator] () {
      const iterator = iterable[Symbol.iterator]();

      return {
        next () {
          let {done, value} = iterator.next();

          done = done || fn(value);

          return ({done, value: done ? undefined : value});
        }
      };
    });
  });
```

Like `mapIterableWith`, they preserve the ordered collection semantics of whatever you give them.

And here's a computation performed using operations on ordered collections: We'll create an ordered collection of square numbers that end in one and are less than 1,000:

```
const Squares = mapIterableWith((x) => x * x, Numbers);
const EndWithOne = filterIterableWith((x) => x % 10 === 1, Squares);
const UpTo1000 = untilIterableWith((x) => (x > 1000), EndWithOne);
```

```
[...UpTo1000]
//=>
[1,81,121,361,441,841,961]
```

```
[...UpTo1000]
//=>
[1,81,121,361,441,841,961]
```

As we expect from an ordered collection, each time we iterate over `UpTo1000`, we begin at the beginning.

For completeness, here are two more handy iterable functions. `firstOfIterable` returns the first element of an iterable (if it has one), and `restOfIterable` returns an iterable that iterates over all but the first element of an iterable. They are equivalent to destructuring arrays with `[first, ...rest]`:

```
const firstOfIterable = (iterable) =>
  iterable[Symbol.iterator]().next().value;

const restOfIterable = (iterable) =>
  ({
    [Symbol.iterator] () {
      const iterator = iterable[Symbol.iterator]();
      iterator.next();
      return iterator;
    }
  });
});
```

like our other operations, `restOfIterable` preserves the ordered collection semantics of its argument.

from

Having iterated over a collection, are we limited to `for .. do` and/or gathering the elements in an array literal and/or gathering the elements into the parameters of a function? No, of course not, we can do anything we like with them.

One useful thing is to write a `.from` function that gathers an iterable into a particular collection type. JavaScript's built-in `Array` class already has one:

```
Array.from(UpTo1000)
//=> [1, 81, 121, 361, 441, 841, 961]
```

We can do the same with our own collections. As you recall, functions are mutable objects. And we can assign properties to functions with `a .` or even `[]`. And if we assign a function to a property, we've created a method.

So let's do that:

```
Stack3.from = function (iterable) {
  const stack = this();

  for (let element of iterable) {
    stack.push(element);
  }
  return stack;
}

Pair1.from = (iterable) =>
  (function iterationToList (iteration) {
    const {done, value} = iteration.next();

    return done ? EMPTY : Pair1(value, iterationToList(iteration));
  })(iterable[Symbol.iterator]())
```

Now we can go "end to end." If we want to map a linked list of numbers to a linked list of the squares of some numbers, we can do that:

```
const numberList = Pair1.from(untilIterableWith((x) => x > 10, Numbers));  
  
Pair1.from(Squares)  
//=> {"first":0,  
      "rest": {"first":1,  
              "rest": {"first":4,  
                      "rest": { ...
```

summary

Iterators are a JavaScript feature that allow us to separate the concerns of how to iterate over a collection from what we want to do with the elements of a collection. *Iterable* ordered collections can be iterated over or gathered into another collection.

Separating concerns with iterators speaks to JavaScript's fundamental nature: It's a language that *wants* to compose functionality out of small, single-responsibility pieces, whether those pieces are functions or objects built out of functions.

Generating Iterables



Banco do Café

Iterables look cool, but then again, everything looks amazing when you're given cherry-picked examples. What is there they don't do well?

Let's consider how they work. Whether it's a simple functional iterator, or an iterable object with a `.next()` method, an iterator is something we call repeatedly until it tells us that it's done.

Iterators have to arrange its own state such that when you call them, they compute and return the next item. This seems blindingly obvious and simple. If, for example, you want numbers, you write:

```
const Numbers = {
  [Symbol.iterator]: () => {
    let n = 0;

    return {
      next: () =>
        ({done: false, value: n++})
    }
  }
};
```

The `Numbers` iterable returns an object that updates a mutable variable, `n`, to deliver number after number. How hard can this be?

Well, we've written our iterator as a *server*. It waits until given a request, and then it returns exactly one item. Then it waits for the next request. There is no concept of pushing numbers out from the iterator, just waiting until a number is pulled out of the iterator by whatever code consumes numbers.

Of course, when we have some code that makes a bunch of something, we don't usually write it like that. We usually just write something like:

```
let n = 0;

while (true) {
  console.log(n++)
}
```

And magically, the numbers would pour forth. We would *generate* numbers. Let's put that beside the code for the iterator, minus the iterable scaffolding:

```
// Iteration
let n = 0;

() =>
  ({done: false, value: n++})

// Generation
let n = 0;

while (true) {
  console.log(n++)
}
```

They're of approximately equal complexity. So why bring up generation? Well, there are some collections that are much easier to generate than to iterate over. Let's look at one:

recursive iterators

Iterators maintain state, that's what they do. Generators have to manage the exact same amount of state, but sometimes, it's much easier to manage that state in a generator. One of those cases is when we have to recursively enumerate something.

For example, iterating over a tree. Given an array that might contain arrays, let's say we want to generate all the "leaf" elements, i.e. elements that are not, themselves, iterable.

```
// Generation
const isIterable = (something) =>
  !!something[Symbol.iterator];

const generate = (iterable) => {
  for (let element of iterable) {
    if (isIterable(element)) {
      generate(element)
    }
    else {
      console.log(element)
    }
  }
}

generate([1, [2, [3, 4], 5]])
//=>
1
2
3
4
5
```

Very simple. Now for the iteration version. We'll write a functional iterator to keep things simple, but it's easy to see the shape of the basic problem:

```
// Iteration
const isIterable = (something) =>
  !!something[Symbol.iterator];

const treeIterator = (iterable) => {
  const iterators = [ iterable[Symbol.iterator]() ];

  return () => {
    while (!!iterators[0]) {
      const iterationResult = iterators[0].next();

      if (iterationResult.done) {
        iterators.shift();
      }
      else if (isIterable(iterationResult.value)) {
        iterators.unshift(iterationResult.value[Symbol.iterator]());
      }
      else {
        return iterationResult.value;
      }
    }
    return;
  }
}

const i = treeIterator([1, [2, [3, 4], 5]]);
let n;

while (n = i()) {
  console.log(n)
}
//=>
1
2
3
4
5
```

If you peel off `isIterable` and ignore the way that the iteration version uses `[Symbol.iterator]` and `.next`, we're left with the fact that the generating version calls itself recursively, and the iteration version maintains an explicit stack. In essence, both the generation and iteration implementations have stacks, but the generation version's stack is *implicit*, while the iteration version's stack is *explicit*.

A less kind way to put it is that the iteration version is greenspunning something built into our programming language: We're reinventing the use of a stack to manage recursion, because writing our code to respond to a function call makes us turn a simple recursive algorithm inside-out.

state machines

Some iterables can be modelled as state machines. Let's revisit the Fibonacci sequence. Again. One way to define it is:

- The first element of the fibonacci sequence is zero.
- The second element of the fibonacci sequence is one.
- Every subsequent element of the fibonacci sequence is the sum of the previous two elements.

Let's write a generator:

```
// Generation
const fibonacci = () => {
  let a, b;

  console.log(a = 0);

  console.log(b = 1);

  while (true) {
    [a, b] = [b, a + b];
    console.log(b);
  }
}

fibonacci()
//=>
0
1
1
2
3
5
8
13
21
34
```

```
55  
89  
144  
...
```

The thing to note here is that our fibonacci generator has three states: generating `0`, generating `1`, and generating everything after that. This isn't a good fit for an iterator, because iterators have one functional entry point and therefore, we'd have to represent our three states explicitly, perhaps using a [state pattern](#)¹¹³:

We'll keep it simple:

```
// Iteration
let a, b, state = 0;

const fibonacci = () => {
  switch (state) {
    case 0:
      state = 1;
      return a = 0;
    case 1:
      state = 2;
      return b = 1;
    case 2:
      [a, b] = [b, a + b];
      return b
  }
};

while (true) {
  console.log(fibonacci());
}
//=>
0
1
1
2
3
5
8
13
```

¹¹³https://en.wikipedia.org/wiki/State_pattern

```
21  
34  
55  
89  
144  
. . .
```

Again, this is not particularly horrendous, but like the recursive example, we're explicitly greenspunning the natural linear state. In a generator, we write "do this, then this, then this." In an iterator, we have to wrap that up and explicitly keep track of what step we're on.

So we see the same thing: The generation version has state, but it's implicit in JavaScript's linear control flow. Whereas the iteration version must make that state explicit.

generators

It would be very nice if we could sometimes write iterators as a `.next()` method that gets called, and sometimes write out a generator. Given the title of this chapter, it is not a surprise that JavaScript makes this possible.

We can write an iterator, but use a generation style of programming. An iterator written in a generation style is called a *generator*. To write a generator, we write a function, but we make two changes:

1. We declare the function using the `function*` keyword. Not a fat arrow. Not a plain `function`.
2. We don't return values or output them to `console.log`. We "yield" values using the `yield` keyword.

When we invoke the function, we get an iterator object back. Let's start with the degenerate example, the empty iterator:

```
const empty = function * () {};  
  
empty().next()  
//=>  
{ "done": true }
```

When we invoke `empty`, we get an iterator with no elements. This makes sense, because `empty` never yields anything. We call its `.next()` method, but it's done immediately.

Generator functions can take an argument. Let's use that to illustrate `yield`:

```
const only = function * (something) {
  yield something;
};

only("you").next()
//=>
{"done":false, value: "you"}
```

Invoking `only("you")` returns an iterator that we can call with `.next()`, and it yields "you". Invoking `only` more than once gives us fresh iterators each time:

```
only("you").next()
//=>
{"done":false, value: "you"}

only("the lonely").next()
//=>
{"done":false, value: "the lonely"}
```

We can invoke the same iterator twice:

```
const sixteen = only("sixteen");

sixteen.next()
//=>
{"done":false, value: "sixteen"}

sixteen.next()
//=>
{"done":true}
```

It yields the value of `something`, and then it's done.

generators are coroutines

Here's a generator that yields three numbers:

```
const oneTwoThree = function * () {
  yield 1;
  yield 2;
  yield 3;
};

oneTwoThree().next()
//=>
{ "done":false, value: 1}

oneTwoThree().next()
//=>
{ "done":false, value: 1}

oneTwoThree().next()
//=>
{ "done":false, value: 1}

const iterator = oneTwoThree();

iterator.next()
//=>
{ "done":false, value: 1}

iterator.next()
//=>
{ "done":false, value: 2}

iterator.next()
//=>
{ "done":false, value: 3}

iterator.next()
//=>
{ "done":true}
```

This is where generators behave very, very differently from ordinary functions. What happens *semantically*?

1. We call `oneTwoThree()` and get an iterator.
2. The iterator is in a nascent or “newborn” state.
3. When we call `iterator.next()`, the body of our generator begins to be evaluated.

4. The body of our generator runs until it returns, ends, or encounters a `yield` statement, which is `yield 1;`.
 - The iterator *suspends its execution*.
 - The iterator wraps 1 in `{done: false, value: 1}` and returns that from the call to `.next()`.
 - The rest of the program continues along its way until it makes another call to `iterator.next()`.
 - The iterator *resumes execution* from the point where it yielded the last value.
5. The body of our generator runs until it returns, ends, or encounters the next `yield` statement, which is `yield 2;`.
 - The iterator *suspends its execution*.
 - The iterator wraps 2 in `{done: false, value: 2}` and returns that from the call to `.next()`.
 - The rest of the program continues along its way until it makes another call to `iterator.next()`.
 - The iterator *resumes execution* from the point where it yielded the last value.
6. The body of our generator runs until it returns, ends, or encounters the next `yield` statement, which is `yield 3;`.
 - The iterator *suspends its execution*.
 - The iterator wraps 3 in `{done: false, value: 3}` and returns that from the call to `.next()`.
 - The rest of the program continues along its way until it makes another call to `iterator.next()`.
 - The iterator *resumes execution* from the point where it yielded the last value.
7. The body of our generator runs until it returns, ends, or encounters the next `yield` statement. There are no more lines of code, so it ends.
 - The iterator returns `{done: true}` from the call to `.next()`, and every call to this iterator's `.next()` method will return `{done: true}` from now on.

This behaviour is not unique to JavaScript, generators are called [coroutines¹¹⁴](#) in other languages:

Coroutines are computer program components that generalize subroutines for nonpreemptive multitasking, by allowing multiple entry points for suspending and resuming execution at certain locations. Coroutines are well-suited for implementing more familiar program components such as cooperative tasks, exceptions, event loop, iterators, infinite lists and pipes.

Instead of thinking of there being one execution context, we can imagine that there are two execution contexts. With an iterator, we can call them the *producer* and the *consumer*. The iterator is the producer, and the code that iterates over it is the consumer. When the consumer calls `.next()`,

¹¹⁴<https://en.wikipedia.org/wiki/Coroutine>

it “suspends” and the producer starts running. When the producer yields a value, the producer suspends and the consumer starts running, taking the value from the result of calling `.next()`.

Of course, generators need not be implemented exactly as coroutines. For example, a “transpiler” might implement `oneTwoThree` as a state machine, a little like this (there is more to generators, but we’ll see that later):

```
const oneTwoThree = function () {
  let state = 'newborn';

  return {
    next () {
      switch (state) {
        case 'newborn':
          state = 1;
          return {value: 1};
        case 1:
          state = 2;
          return {value: 2}
        case 2:
          state = 3;
          return {value: 3}
        case 3:
          return {done: true};
      }
    }
  }
};
```

But no matter how JavaScript implements it, our mental model is that a generator function returns an iterator, and that when we call `.next()`, it runs until it returns, ends, or yields. If it yields, it suspends its own execution and the consuming code resumes execution, until `.next()` is called again, at which point the iterator resumes its own execution from the point where it yielded.

generators and iterables

Our generator function `oneTwoThree` is not an iterator. It’s a function that returns an iterator when we invoke it. We write the function to `yield` values instead of `return` a single value, and JavaScript takes care of turning this into an object with a `.next()` function we can call.

If we call our generator function more than once, we get new iterators. As we saw above, we called `oneTwoThree` three times, and each time we got an iterator that begins at 1 and counts to 3. Recalling the way we wrote ordered collections, we could make a collection that uses a generator function:

```
const ThreeNumbers = {
  [Symbol.iterator]: function* () {
    yield 1;
    yield 2;
    yield 3
  }
}

for (let i of ThreeNumbers) {
  console.log(i);
}
//=>
1
2
3

[...ThreeNumbers]
//=>
[1,2,3]

const iterator = ThreeNumbers[Symbol.iterator]();

iterator.next()
//=>
{ "done":false, value: 1}

iterator.next()
//=>
{ "done":false, value: 2}

iterator.next()
//=>
{ "done":false, value: 3}

iterator.next()
//=>
{ "done":true}
```

Now we can use it in a `for...of` loop, spread it into an array literal, or spread it into a function invocation, because we have written an iterable that uses a generator to return an iterator from its `[Symbol.iterator]` method.

This pattern is encouraged, so much so that JavaScript provides a concise syntax for writing

generator methods for objects:

```
const ThreeNumbers = {
  *[Symbol.iterator] () {
    yield 1;
    yield 2;
    yield 3
  }
}
```

more generators

Generators can produce infinite streams of values:

```
const Numbers = {
  *[Symbol.iterator] () {
    let i = 0;

    while (true) {
      yield i++;
    }
  }
};

for (let i of Numbers) {
  console.log(i);
}
//=>
0
1
2
3
4
5
6
7
8
9
10
...
```

Our OneTwoThree example used implicit state to output the numbers in sequence. Recall that we wrote Fibonacci using explicit state:

```
const Fibonacci = {
  [Symbol.iterator]: () => {
    let a = 0, b = 1, state = 0;

    return {
      next: () => {
        switch (state) {
          case 0:
            state = 1;
            return {value: a};
          case 1:
            state = 2;
            return {value: b};
          case 2:
            [a, b] = [b, a + b];
            return {value: b};
        }
      }
    }
  }
};

for (let n of Fibonacci) {
  console.log(n)
}
//=>
0
1
1
2
3
5
8
13
21
34
55
89
144
...
```

And here is the Fibonacci ordered collection, implemented with a generator method:

```
const Fibonacci = {
  *[Symbol.iterator] () {
    let a, b;

    yield a = 0;

    yield b = 1;

    while (true) {
      [a, b] = [b, a + b]
      yield b;
    }
  }
}

for (let i of Fibonacci) {
  console.log(i);
}
//=>
0
1
1
2
3
5
8
13
21
34
55
89
144
...
```

We've writing a function that returns an iterator, but we used a generator to do it. And the generator's syntax allows us to use JavaScript's natural management of state instead of constantly rolling our own.

yielding iterables

Here's a generator for iterating over trees:

```
const iterable = (something) =>
  !!something[Symbol.iterator];

const TreeIterable = (iterable) =>
  ({
    [Symbol.iterator]: function* () {
      for (let e of iterable) {
        if (iterable(e)) {
          for (let ee of TreeIterable(e)) {
            yield ee;
          }
        } else {
          yield e;
        }
      }
    }
  })
}

for (let i of TreeIterable([1, [2, [3, 4], 5]])) {
  console.log(i);
}
//=>
1
2
3
4
5
```

We've gone with the full iterable here, a `TreeIterable(iterable)` returns an iterable that treats `iterable` as a tree. We take advantage of the `for...of` loop in a plain and direct way: For each element `e`, if it is iterable, treat it as a tree and iterate over it, yielding each of its elements. If `e` is not an iterable, yield `e`.

JavaScript handles the recursion for us using its own execution stack. This is clearly simpler than trying to maintain our own stack and remembering whether we are shifting and unshifting, or pushing and popping. And while this version has the extra scaffolding to make it a first-class iterable, it matches our simple generation code from above more-or-less directly.

But while we're here, let's look at one bit of this code:

```
for (let ee of TreeIterable(e)) {
  yield ee;
}
```

These three lines say, in essence, “yield all the elements of TreeIterable(e), in order.” This comes up quite often when we have collections that are compounds, collections made from other collections.

Consider this operation on iterables:

```
const appendIterables = (...iterables) =>
  ({
    [Symbol.iterator]: function* () {
      for (let iterable of iterables) {
        for (let element of iterable) {
          yield element;
        }
      }
    }
  })
}

const lyrics = appendIterables(["a", "b", "c"], ["one", "two", "three"], ["do", \
"re", "me"]);

for (let word of lyrics) {
  console.log(word);
}
//=>
a
b
c
one
two
three
do
re
me
```

appendIterables iterates over a collection of iterables, one element at a time. Things like arrays can be easily catenated, but appendIterables iterates lazily, so there’s no need to construct intermediary results.

Tucked inside of it is the same three-line idiom for yielding each element of an iterable. There is an abbreviation for this, we can use **yield*** to yield all the elements of an iterable:

```

const appendIterables = (...iterables) =>
  ({
    [Symbol.iterator]: function* () {
      for (let iterable of iterables) {
        yield* iterable;
      }
    }
  })

const lyrics = appendIterables(["a", "b", "c"], ["one", "two", "three"], ["do", \
"re", "me"]);

for (let word of lyrics) {
  console.log(word);
}
//=>
a
b
c
one
two
three
do
re
me

```

`yield*` yields all of the elements of an iterable, in order. We can use it in `TreeIterable` too:

```

const isIterable = (something) =>
  !!something[Symbol.iterator];

const TreeIterable = (iterable) =>
  ({
    [Symbol.iterator]: function* () {
      for (let e of iterable) {
        if (isIterable(e)) {
          yield* TreeIterable(e);
        }
        else {
          yield e;
        }
      }
    }
  })

```

```

        }
    })

for (let i of TreeIterable([1, [2, [3, 4], 5]])) {
    console.log(i);
}
//=>
1
2
3
4
5

```

`yield*` is handy when writing generator functions that operate on or create iterables.

rewriting iterable operations

Now that we know about iterables, we can rewrite our iterable operations to use generators. Instead of:

```

const mapIterableWith = (fn, iterable) =>
  ({
    [Symbol.iterator]: () => {
      const iterator = iterable[Symbol.iterator]();

      return {
        next: () => {
          const {done, value} = iterator.next();

          return ({done, value: done ? undefined : fn(value)});
        }
      }
    });
  );

```

We can write:

```
const mapIterableWith = (fn, iterable) =>
  ({
    [Symbol.iterator]: function* () {
      for (let element of iterable) {
        yield fn(element);
      }
    }
  });
});
```

We still return an object that has a `[Symbol.iterator]` method, only now that method is a generator. No need to return an object with a `.next()` method. No need to fool around with `{done}` or `{value}`, just `yield` values until we're done.

And as we recall from [operations on ordered collections](#), to preserve collection's ordered collection semantics, the object we return has a `[Symbol.iterator]` method, and when we invoke that, it must in turn invoke collection's own `[Symbol.iterator]` method.

We don't do that explicitly, but `for (let element of collection)` invokes collection's `[Symbol.iterator]` method, and thus `mapIterableWith` returns an iterable that invokes collection's `[Symbol.iterator]`, just as we wish. And it is simpler and easier to read.

We can do the same thing with our other operations like `filterIterableWith` and `untilIterableWith`. Here're our iterable methods rewritten as generators:

```
const mapIterableWith = (fn, iterable) =>
  ({
    [Symbol.iterator]: function* () {
      for (let element of iterable) {
        yield fn(element);
      }
    }
  });
});
```

```
const filterIterableWith = (fn, iterable) =>
  ({
    [Symbol.iterator]: function* () {
      for (let element of iterable) {
        if (!!fn(element)) yield element;
      }
    }
  });
});
```

```
const untilIterableWith = (fn, iterable) =>
  ({
```

```
[Symbol.iterator]: function* () {
  for (let element of iterable) {
    if (fn(element)) break;
    yield fn(element);
  }
}
});
```

`firstOfIterable` and `restOfIterable` both work directly with iterators and remain unchanged:

```
const firstOfIterable = (iterable) =>
  iterable[Symbol.iterator]().next().value;

const restOfIterable = (iterable) =>
  ({
    [Symbol.iterator]: () => {
      const iterator = iterable[Symbol.iterator]();

      iterator.next();
      return iterator;
    }
  });
});
```

Summary

A generator is a function that is defined with `function*` and uses `yield` (or `yield*`) to generate values. Using a generator instead of writing an iterator object that has a `.next()` method allows us to write code that can be much simpler for cases like recursive iterations or state patterns. And we don't need to worry about wrapping our values in an object with `.done` and `.value` properties.

This is especially useful for making iterables.

Lazy and Eager Collections

The operations on iterables are tremendously valuable, but let's reiterate why we care: In JavaScript, we build single-responsibility objects, and single-responsibility functions, and we compose these together to build more full-featured objects and algorithms.

Composing an iterable with a `mapIterable` method cleaves the responsibility for knowing how to map from the fiddly bits of how a linked list differs from a stack

in the older style of object-oriented programming, we built “fat” objects. Each collection knew how to map itself (`.map`), how to fold itself (`.reduce`), how to filter itself (`.filter`) and how to find one element within itself (`.find`). If we wanted to flatten collections to arrays, we wrote a `.toArray` method for each type of collection.

Over time, this informal “interface” for collections grows by accretion. Some methods are only added to a few collections, some are added to all. But our objects grow fatter and fatter. We tell ourselves that, well, a collection ought to know how to map itself.

But we end up recreating the same bits of code in each `.map` method we create, in each `.reduce` method we create, in each `.filter` method we create, and in each `.find` method. Each one has its own variation, but the overall form is identical. That's a sign that we should work at a higher level of abstraction, and working with iterables is that higher level of abstraction.

This “fat object” style springs from a misunderstanding: When we say a collection should know how to perform a map over itself, we don't need for the collection to handle every single detail. That would be like saying that when we ask a bank teller for some cash, they personally print every bank note.

implementing methods with iteration

Object-oriented collections should definitely have methods for mapping, reducing, filtering, and finding. And they should know how to accomplish the desired result, but they should do so by delegating as much of the work as possible to operations like `mapIterableWith`.

Composing an iterable with a `mapIterable` method cleaves the responsibility for knowing how to map from the fiddly bits of how a linked list differs from a stack. And if we want to create convenience methods, we can reuse common pieces.

Here is `LazyCollection`, a mixin we can use with any ordered collection that is also an iterable:

```
const extend = function (consumer, ...providers) {
  for (let i = 0; i < providers.length; ++i) {
    const provider = providers[i];
    for (let key in provider) {
      if (provider.hasOwnProperty(key)) {
        consumer[key] = provider[key]
      }
    }
  }
  return consumer
};

const LazyCollection = {
  map(fn) {
    return Object.assign({
      [Symbol.iterator]: () => {
        const iterator = this[Symbol.iterator]();

        return {
          next: () => {
            const {
              done, value
            } = iterator.next();

            return ({
              done, value: done ? undefined : fn(value)
            });
          }
        }
      }
    }, LazyCollection);
  },
  reduce(fn, seed) {
    const iterator = this[Symbol.iterator]();
    let iterationResult,
      accumulator = seed;

    while ((iterationResult = iterator.next(), !iterationResult.done)) {
      accumulator = fn(accumulator, iterationResult.value);
    }
    return accumulator;
  }
};
```

```
},  
  
filter(fn) {  
  return Object.assign({  
    [Symbol.iterator]: () => {  
      const iterator = this[Symbol.iterator]();  
  
      return {  
        next: () => {  
          do {  
            const {  
              done, value  
            } = iterator.next();  
            } while (!done && !fn(value));  
          return {  
            done, value  
          };  
        };  
      }  
    }  
  }, LazyCollection)  
},  
  
find(fn) {  
  return Object.assign({  
    [Symbol.iterator]: () => {  
      const iterator = this[Symbol.iterator]();  
  
      return {  
        next: () => {  
          let {  
            done, value  
          } = iterator.next();  
  
          done = done || fn(value);  
  
          return ({  
            done, value: done ? undefined : value  
          });  
        };  
      }  
    }  
  }  
}
```

```
    }, LazyCollection)
  ,

until(fn) {
  return Object.assign({
    [Symbol.iterator]: () => {
      const iterator = this[Symbol.iterator]();

      return {
        next: () => {
          let {
            done, value
          } = iterator.next();

          done = done || fn(value);

          return ({
            done, value: done ? undefined : value
          });
        }
      }
    },
  }, LazyCollection)
},

first() {
  return this[Symbol.iterator]().next().value;
},

rest() {
  return Object.assign({
    [Symbol.iterator]: () => {
      const iterator = this[Symbol.iterator]();

      iterator.next();
      return iterator;
    }
  }, LazyCollection);
},

take(numberToTake) {
  return Object.assign({
```

```
[Symbol.iterator]: () => {
  const iterator = this[Symbol.iterator]();
  let remainingElements = numberToTake;

  return {
    next: () => {
      let {
        done, value
      } = iterator.next();

      done = done || remainingElements-- <= 0;

      return ({
        done, value: done ? undefined : value
      });
    }
  }
}, LazyCollection);
}
}
```

To use `LazyCollection`, we mix it into an any iterable object. For simplicity, we'll show how to mix it into `Numbers` and `Pair`. But it can also be mixed into prototypes (a/k/a "classes"), traits, or other OO constructs:

```
const Numbers = Object.assign({
  [Symbol.iterator]: () => {
    let n = 0;

    return {
      next: () =>
        ({done: false, value: n++})
    }
  }
}, LazyCollection);
```

// Pair, a/k/a linked lists

```
const EMPTY = {
  isEmpty: () => true
```

```
};

const isEmpty = (node) => node === EMPTY;

const Pair = (car, cdr = EMPTY) =>
  Object.assign({
    car,
    cdr,
    isEmpty: () => false,
    [Symbol.iterator]: function () {
      let currentPair = this;

      return {
        next: () => {
          if (currentPair.isEmpty()) {
            return {done: true}
          }
          else {
            const value = currentPair.car;

            currentPair = currentPair.cdr;
            return {done: false, value}
          }
        }
      }
    }
  }, LazyCollection);

Pair.from = (iterable) =>
  (function iterationToList (iteration) {
    const {done, value} = iteration.next();

    return done ? EMPTY : Pair(value, iterationToList(iteration));
  })(iterable[Symbol.iterator]());

// Stack

const Stack = () =>
  Object.assign({
    array: [],
    index: -1,
    push: function (value) {
```

```
    return this.array[this.index += 1] = value;
},
pop: function () {
  const value = this.array[this.index];

  this.array[this.index] = undefined;
  if (this.index >= 0) {
    this.index -= 1
  }
  return value
},
isEmpty: function () {
  return this.index < 0
},
[Symbol.iterator]: function () {
  let iterationIndex = this.index;

  return {
    next: () => {
      if (iterationIndex > this.index) {
        iterationIndex = this.index;
      }
      if (iterationIndex < 0) {
        return {done: true};
      }
      else {
        return {done: false, value: this.array[iterationIndex--]}
      }
    }
  }
},
LazyCollection);

Stack.from = function (iterable) {
  const stack = this();

  for (let element of iterable) {
    stack.push(element);
  }
  return stack;
}
```

```
// Pair and Stack in action
```

```
Stack.from([1, 2, 3, 4, 5, 6, 7, 8, 9, 10])
  .map((x) => x * x)
  .filter((x) => x % 2 == 0)
  .first()
```

//=> 100

```
Pair.from([1, 2, 3, 4, 5, 6, 7, 8, 9, 10])
  .map((x) => x * x)
  .filter((x) => x % 2 == 0)
  .reduce((seed, element) => seed + element, 0)
```

//=> 220

lazy collection operations

“Laziness” is a very pejorative word when applied to people. But it can be an excellent strategy for efficiency in algorithms. Let’s be precise: *Laziness* is the characteristic of not doing any work until you know you need the result of the work.

Here’s an example. Compare these two:

```
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
  .map((x) => x * x)
  .filter((x) => x % 2 == 0)
  .reduce((seed, element) => seed + element, 0)

Pair.from([1, 2, 3, 4, 5, 6, 7, 8, 9, 10])
  .map((x) => x * x)
  .filter((x) => x % 2 == 0)
  .reduce((seed, element) => seed + element, 0)
```

Both expressions evaluate to 220. And they array is faster in practice, because it is a built-in data type that performs its work in the engine, while the linked list does its work in JavaScript.

But it’s still illustrative to dissect something important: Array’s `.map` and `.filter` methods gather their results into new arrays. Thus, calling `.map`, `.filter`, `.reduce` produces two temporary arrays that are discarded when `.reduce` performs its final computation.

Whereas the `.map` and `.filter` methods on `Pair` work with iterators. They produce small iterable objects that refer back to the original iteration. This reduces the memory footprint. When working with very large collections and many operations, this can be important.

The effect is even more pronounced when we use methods like `first`, `until`, or `take`:

```
Stack.from([ 0, 1, 2, 3, 4, 5, 6, 7, 8, 9,
            10, 11, 12, 13, 14, 15, 16, 17, 18, 19,
            20, 21, 22, 23, 24, 25, 26, 27, 28, 29])
  .map((x) => x * x)
  .filter((x) => x % 2 == 0)
  .first()
```

This expression begins with a stack containing 30 elements. The top two are 29 and 28. It maps to the squares of all 30 numbers, but our code for mapping an iteration returns an iterable that can iterate over the squares of our numbers, not an array or stack of the squares. Same with `.filter`, we get an iterable that can iterate over the even squares, but not an actual stack or array.

Finally, we take the first element of that filtered, squared iterable and now JavaScript actually iterates over the stack's elements, and it only needs to square two of those elements, 29 and 28, to return the answer.

We can confirm this:

```
Stack.from([ 0, 1, 2, 3, 4, 5, 6, 7, 8, 9,
            10, 11, 12, 13, 14, 15, 16, 17, 18, 19,
            20, 21, 22, 23, 24, 25, 26, 27, 28, 29])
  .map((x) => {
    console.log(`squaring ${x}`);
    return x * x
  })
  .filter((x) => {
    console.log(`filtering ${x}`);
    return x % 2 == 0
  })
  .first()

//=>
squaring 29
filtering 841
squaring 28
filtering 784
784
```

If we write the almost identical thing with an array, we get a different behaviour:

```
[ 0, 1, 2, 3, 4, 5, 6, 7, 8, 9,
 10, 11, 12, 13, 14, 15, 16, 17, 18, 19,
 20, 21, 22, 23, 24, 25, 26, 27, 28, 29]
  .reverse()
  .map((x) => {
    console.log(`squaring ${x}`);
    return x * x
  })
  .filter((x) => {
    console.log(`filtering ${x}`);
    return x % 2 == 0
  } )[0]
```

```
//=>
squaring 0
squaring 1
squaring 2
squaring 3
...
squaring 28
squaring 29
filtering 0
filtering 1
filtering 4
...
filtering 784
filtering 841
784
```

Arrays copy-on-read, so every time we perform a map or filter, we get a new array and perform all the computations. This might be expensive.

You recall we briefly touched on the idea of infinite collections? Let's make iterable numbers. They *have* to be lazy, otherwise we couldn't write things like:

```

const Numbers = Object.assign({
  [Symbol.iterator]: () => {
    let n = 0;

    return {
      next: () =>
        ({done: false, value: n++})
    }
  }
}, LazyCollection);

const firstCubeOver1234 =
  Numbers
    .map((x) => x * x * x)
    .filter((x) => x > 1234)
    .first()

//=> 1331

```

Balanced against their flexibility, our “lazy collections” use structure sharing. If we mutate a collection after taking an iterable, we might get an unexpected result. This is why “pure” functional languages like Haskell combine lazy semantics with immutable collections, and why even “impure” languages like Clojure emphasize the use of immutable collections.

eager collections

An *eager* collection, like an array, returns a collection of its own type from each of the methods. We can make an eager collection out of any collection that is *gatherable*, meaning it has a `.from` method:

```

const extend = function (consumer, ...providers) {
  for (let i = 0; i < providers.length; ++i) {
    const provider = providers[i];
    for (let key in provider) {
      if (provider.hasOwnProperty(key)) {
        consumer[key] = provider[key]
      }
    }
  }
  return consumer
};

```

```
const EagerCollection = (gatherable) =>
  ({
    map(fn) {
      const original = this;

      return gatherable.from(
        (function* () {
          for (let element of original) {
            yield fn(element);
          }
        })()
      );
    },
    reduce(fn, seed) {
      let accumulator = seed;

      for(let element of this) {
        accumulator = fn(accumulator, element);
      }
      return accumulator;
    },
    filter(fn) {
      const original = this;

      return gatherable.from(
        (function* () {
          for (let element of original) {
            if (fn(element)) yield element;
          }
        })()
      );
    },
    find(fn) {
      for (let element of this) {
        if (fn(element)) return element;
      }
    },
    until(fn) {
```

```
const original = this;

return gatherable.from(
  (function* () {
    for (let element of original) {
      if (fn(element)) break;
      yield element;
    }
  })()
);

first() {
  return this[Symbol.iterator]().next().value;
},

rest() {
  const iteration = this[Symbol.iterator]();

  iteration.next();
  return gatherable.from(
    (function* () {
      yield* iteration;
    })()
  );
  return gatherable.from(iterable);
},

take(numberToTake) {
  const original = this;
  let numberRemaining = numberToTake;

  return gatherable.from(
    (function* () {
      for (let element of original) {
        if (numberRemaining-- <= 0) break;
        yield element;
      }
    })()
  );
}
});
```

Here is our Pair implementation. Pair is gatherable, because it implements `.from()`. We mix `EagerCollection(Pair)` into it, and this gives it all of our collection methods, which each method returning a new list of pairs:

```

const EMPTY = {
  isEmpty: () => true
};

const isEmpty = (node) => node === EMPTY;

const Pair = (car, cdr = EMPTY) =>
  Object.assign({
    car,
    cdr,
    isEmpty: () => false,
    [Symbol.iterator]: function () {
      let currentPair = this;

      return {
        next: () => {
          if (currentPair.isEmpty()) {
            return {done: true}
          }
          else {
            const value = currentPair.car;

            currentPair = currentPair.cdr;
            return {done: false, value}
          }
        }
      }
    }
  }, EagerCollection(Pair));

Pair.from = (iterable) =>
  (function iterationToList (iteration) {
    const {done, value} = iteration.next();

    return done ? EMPTY : Pair(value, iterationToList(iteration));
  })(iterable[Symbol.iterator]());

Pair.from([1, 2, 3, 4, 5]).map(x => x * 2)
//=>
```

```
{"car": 2,  
 "cdr": {"car": 4,  
          "cdr": {"car": 6,  
                    "cdr": {"car": 8,  
                              "cdr": {"car": 10,  
                                        "cdr": {}  
                                      }  
                                    }  
                                  }  
                                }  
}
```

Interlude: The Carpenter Interviews for a Job

“The Carpenter” was a JavaScript programmer, well-known for a meticulous attention to detail and love for hand-crafted, exquisitely joined code. The Carpenter normally worked through personal referrals, but from time to time a recruiter would slip through his screen. One such recruiter was Bob Plissken. Bob was well-known in the Python community, but his clients often needed experience with other languages.

Plissken lined up a technical interview with a well-funded startup in San Francisco. The Carpenter arrived early for his meeting with “Thing Software,” and was shown to conference room 13. A few minutes later, he was joined by one of the company’s developers, Christine.

the problem

After some small talk, Christine explained that they liked to ask candidates to whiteboard some code. Despite his experience and industry longevity, the Carpenter did not mind being asked to demonstrate that he was, in fact, the person described on the resumé.

Many companies use white-boarding code as an excuse to have a technical conversation with a candidate, and The Carpenter felt that being asked to whiteboard code was an excuse to have a technical conversation with a future colleague. “Win, win” he thought to himself.



Christine intoned the question, as if by rote:

Consider a finite checkerboard of unknown size. On each square, we randomly place an arrow pointing to one of its four sides. A chequer is placed randomly on the checkerboard. Each move consists of moving the chequer one square in the direction of the arrow in the square it occupies. If the arrow should cause the chequer to move off the edge of the board, the game halts.

The problem is this: The game board is hidden from us. A player moves the chequer, following the rules. As the player moves the chequer, they call out the direction of movement, e.g. “ \uparrow , \rightarrow , \uparrow , \downarrow , \uparrow , \rightarrow ...” Write an algorithm that will determine whether the game halts, strictly from the called out directions, in finite time and space.

“So,” The Carpenter asked, “I am to write an algorithm that takes a possibly infinite stream of...”

Christine interrupted. “To save time, we have written a template of the solution for you in ECMAScript 2015 notation. Fill in the blanks. Your code should not presume anything about the

¹¹⁵<https://www.flickr.com/photos/stigrudeholm/6710684795>

game-board's size or contents, only that it is given an arrow every time though the while loop. You may use [babeljs.io¹¹⁶](http://babeljs.io), or [ES6Fiddle¹¹⁷](http://www.esofiddle.net) to check your work. “

Christine quickly scribbled on the whiteboard:

```
const Game = (size = 8) => {

    // initialize the board
    const board = [];
    for (let i = 0; i < size; ++i) {
        board[i] = [];
        for (let j = 0; j < size; ++j) {
            board[i][j] = '□□□' [Math.floor(Math.random() * 4)];
        }
    }

    // initialize the position
    let initialPosition = [
        2 + Math.floor(Math.random() * (size - 4)),
        2 + Math.floor(Math.random() * (size - 4))
    ];

    // ???
    let [x, y] = initialPosition;

    const MOVE = {
        "↖": ([x, y]) => [x - 1, y],
        "↗": ([x, y]) => [x + 1, y],
        "↙": ([x, y]) => [x, y - 1],
        "↘": ([x, y]) => [x, y + 1]
    };
    while (x >= 0 && y >= 0 && x < size && y < size) {
        const arrow = board[x][y];

        // ???

        [x, y] = MOVE[arrow]([x, y]);
    }
    // ???
};
```

¹¹⁶<http://babeljs.io>

¹¹⁷<http://www.esofiddle.net>

“What,” Christine asked, “Do you write in place of the three // ??? placeholders to determine whether the game halts?”

the carpenter’s solution

The Carpenter was not surprised at the problem. Bob Plissken was a crafty, almost reptilian recruiter that traded in information and secrets. Whenever Bob sent a candidate to a job interview, he debriefed them afterwards and got them to disclose what questions were asked in the interview. He then coached subsequent candidates to give polished answers to the company’s pet technical questions.

And just as companies often pick a problem that gives them broad latitude for discussing alternate approaches and determining that depth of a candidate’s experience, The Carpenter liked to sketch out solutions that provided an opportunity to judge the interviewer’s experience and provide an easy excuse to discuss the company’s approach to software design.

Bob had, in fact, warned The Carpenter that “Thing” liked to ask either or both of two questions: Determine how to detect a loop in a linked list, and determine whether the chequerboard game would halt. To save time, The Carpenter had prepared the same answer for both questions.

The Carpenter coughed softly, then began. “To begin with, I’ll transform a game into an iterable that generates arrows, using the ‘Starman’ notation for generators. I’ll refactor a touch to make things clearer, for example I’ll extract the board to make it easier to test.”

```
const MOVE = {
    "↖": ([x, y]) => [x - 1, y],
    "↗": ([x, y]) => [x + 1, y],
    "↙": ([x, y]) => [x, y + 1],
    "↘": ([x, y]) => [x, y - 1]
};

const Board = (size = 8) => {

    // initialize the board
    const board = [];
    for (let i = 0; i < size; ++i) {
        board[i] = [];
        for (let j = 0; j < size; ++j) {
            board[i][j] = '◻◻◻◻'[Math.floor(Math.random() * 4)];
        }
    }

    // initialize the position
    const position = [

```

```

    Math.floor(Math.random() * size),
    Math.floor(Math.random() * size)
];

return {board, position};
};

const Game = ({board, position}) => {

  const size = board[0].length;

  return ({
    *[Symbol.iterator] () {
      let [x, y] = position;

      while (x >= 0 && y >=0 && x < size && y < size) {
        const direction = board[y][x];

        yield direction;
        [x, y] = MOVE[direction]([x, y]);
      }
    }
  });
};

```

“Now that we have an iterable, we can transform the iterable of arrows into an iterable of positions.” The Carpenter sketched quickly. “We want to take the arrows and convert them to positions. For that, we’ll map the Game iterable to positions. A `statefulMap` is a lazy map that preserves state from iteration to iteration. That’s what we need, because we need to know the current position to map each move to the next position.”

“This is a standard idiom we can obtain from libraries, we don’t reinvent the wheel. I’ll show it here for clarity:”

```

const statefulMapIterableWith = (fn, seed, iterable) =>
({
  *[Symbol.iterator] () {
    let value,
      state = seed;

    for (let element of iterable) {
      [state, value] = fn(state, element);
      yield value;
    }
  }
});

```

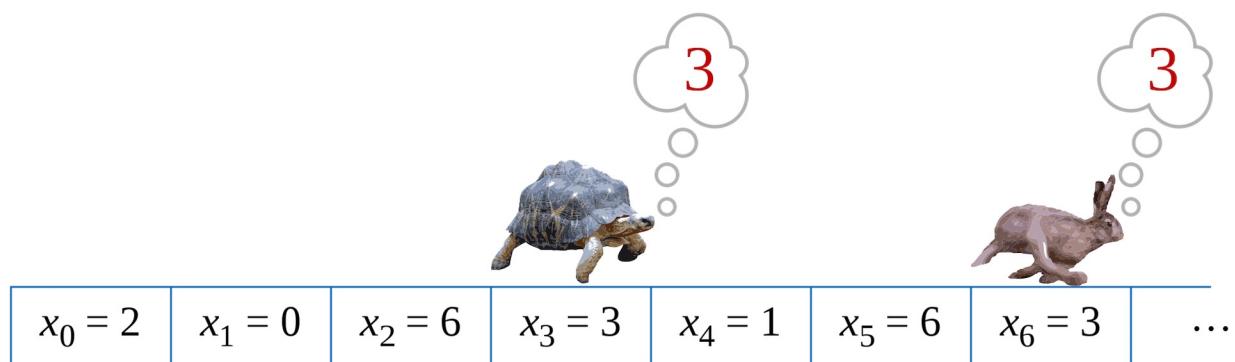
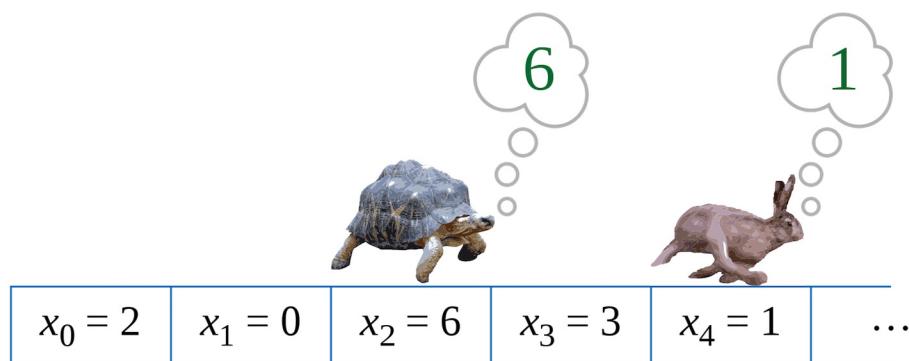
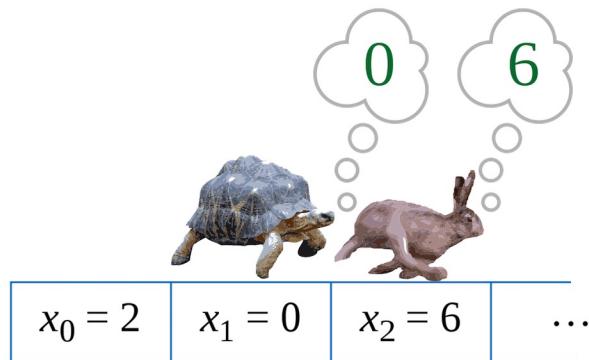
```
        }
    }
});
```

“Armed with this, it’s straightforward to map an iterable of directions to an iterable of strings representing positions:”

```
const positionsOf = (game) =>
  statefulMapIterableWith(
    (position, direction) => {
      const [x, y] = MOVE[direction](position);
      position = [x, y];
      return [position, `x: ${x}, y: ${y}`];
    },
    [0, 0],
    game);
```

The Carpenter reflected. “Having turned our game loop into an iterable, we can now see that our problem of whether the game terminates is isomorphic to the problem of detecting whether the positions given ever repeat themselves: If the chequer ever returns to a position it has previously visited, it will cycle endlessly.”

“We could draw positions as nodes in a graph, connected by arcs representing the arrows. Detecting whether the game terminates is equivalent to detecting whether the graph contains a cycle.”



The Tortoise and the Hare

“There’s an old joke that a mathematician is someone who will take a five-minute problem, then spend an hour proving it is equivalent to another problem they have already solved. I approached this question in that spirit. Now that we have created an iterable of values that can be compared with `==`, I can show you this function:”

```

const tortoiseAndHare = (iterable) => {
  const hare = iterable[Symbol.iterator]();
  let hareResult = (hare.next(), hare.next());

  for (let tortoiseValue of iterable) {

    hareResult = hare.next();

    if (hareResult.done) {
      return false;
    }
    if (tortoiseValue === hareResult.value) {
      return true;
    }

    hareResult = hare.next();

    if (hareResult.done) {
      return false;
    }
    if (tortoiseValue === hareResult.value) {
      return true;
    }
  }
  return false;
};

```

“A long time ago,” The Carpenter explained, “Someone asked me a question in an interview. I have never forgotten the question, or the general form of the solution. The question was, *Given a linked list, detect whether it contains a cycle. Use constant space.*”

“This is, of course, the most common solution, it is [Floyd’s cycle-finding algorithm](#)¹¹⁸, although there is some academic dispute as to whether Robert Floyd actually discovered it or was misattributed by Knuth.”

“Thus, the solution to the game problem is:”

¹¹⁸https://en.wikipedia.org/wiki/Cycle_detection#Tortoise_and_hare

```

const terminates = (game) =>
  tortoiseAndHare(positionsOf(game))

const test = [
  ["□", "□", "□", "□"],
  ["□", "□", "□", "□"],
  ["□", "□", "□", "□"],
  ["□", "□", "□", "□"]
];

terminates(Game({board: test, position: [0, 0]}))
  //=> false
terminates(Game({board: test, position: [3, 0]}))
  //=> true
terminates(Game({board: test, position: [0, 3]}))
  //=> false
terminates(Game({board: test, position: [3, 3]}))
  //=> false

```

“This solution makes use of iterables and a single utility function, `statefulMapIterableWith`. It also cleanly separates the mechanics of the game from the algorithm for detecting cycles in a graph.”

the aftermath

The Carpenter sat down and waited. This type of solution provided an excellent opportunity to explore lazy versus eager evaluation, the performance of iterators versus native iteration, single responsibility design, and many other rich topics.

The Carpenter was confident that although nobody would write this exact code in production, prospective employers would also recognize that nobody would try to detect whether a chequer game terminates in production, either. It’s all just a pretext for kicking off an interesting conversation, right?

Christine looked at the solution on the board, frowned, and glanced at the clock on the wall. “*Well, where has the time gone?*”

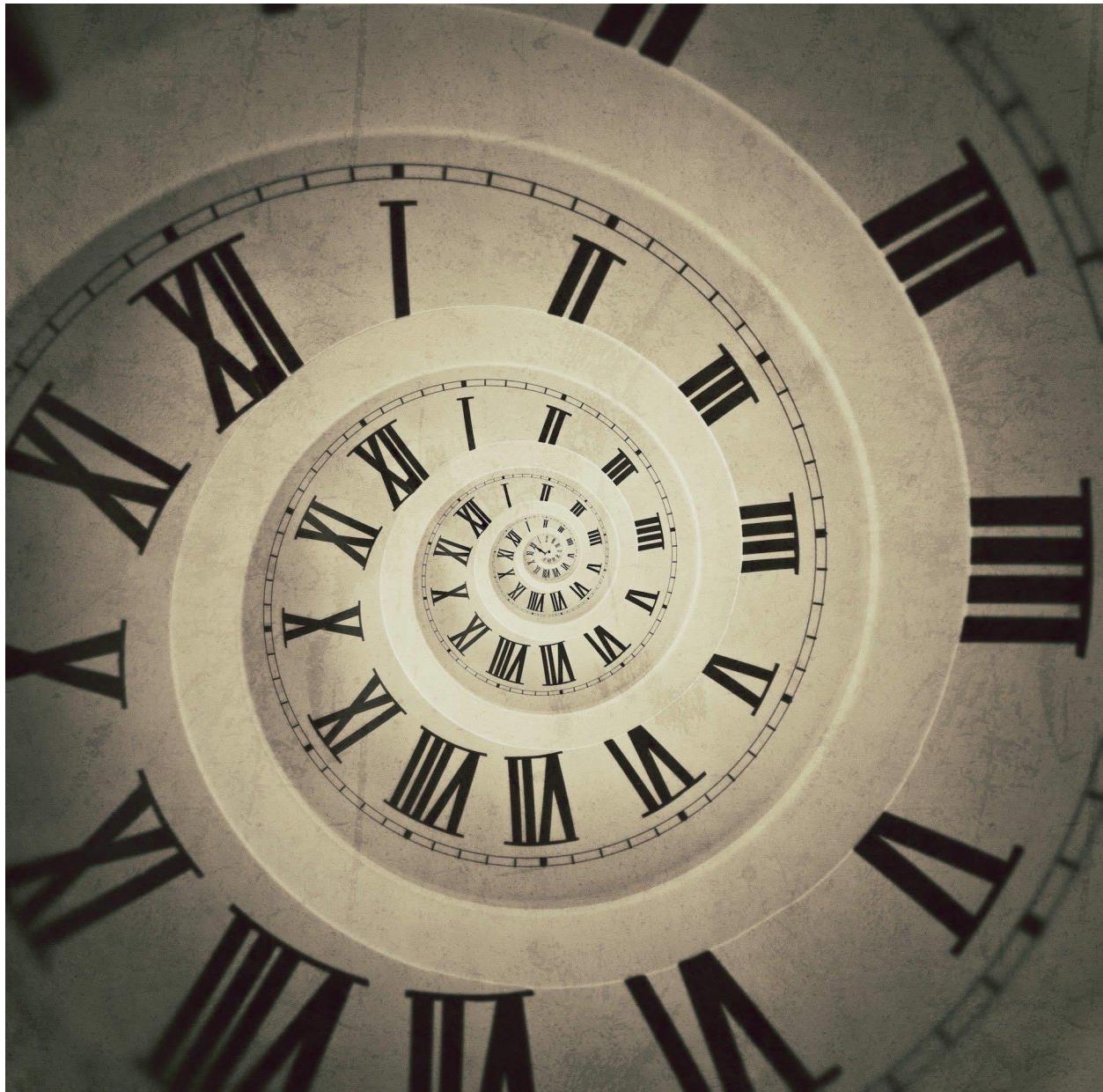
“We at the Thing Software company are very grateful you made some time to visit with us, but alas, that is all the time we have today. If we wish to talk to you further, we’ll be in touch.”

The Carpenter never did hear back from them, but the next day there was an email containing a generous contract from Friends of Ghosts (“FOG”), a codename for a stealth startup doing interesting work, and the Thing interview was forgotten.

Some time later, The Carpenter ran into Bob Plissken at a local technology meet-up. “John! What happened at Thing?” Bob wanted to know, “I asked them what they thought of you, and all they

would say was, *Writes unreadable code*. I thought it was a lock! I thought you'd finally make your escape from New York."

The Carpenter smiled. "I forgot about them, it's been a while. So, do They Live?"



119

after another drink

A few drinks later, The Carpenter was telling his Thing story and an engineer named Kidu introduced themselves.

¹¹⁹<https://www.flickr.com/photos/jlhopgood/6795353385>

“I worked at Thing, and Christine told us about your solution. I had a look at the code you left on the whiteboard. Of course, white-boarding in an interview situation is notoriously unreliable, so small defects are not important. But I couldn’t help but notice that your solution doesn’t actually meet the stated requirements for a different reason.”

“The `hasCycle` function, a/k/a Tortoise and Hare, requires two separate iterators to do its job. Whereas the problem as stated involves a single stream of directions. You’re essentially calling for the player to clone themselves and call out the directions in parallel.”

The Carpenter thought about this for a moment. “Kidu, you’re right, that’s a fantastic observation. I should have used a Teleporting Tortoise!”

```
// implements Teleporting Tortoise
// cycle detection algorithm.
const hasCycle = (iterable) => {
  let iterator = iterable[Symbol.iterator](),
    teleportDistance = 1;

  while (true) {
    let {value, done} = iterator.next(),
      tortoise = value;
    if (done) return false;

    for (let i = 0; i < teleportDistance; ++i) {
      let {value, done} = iterator.next(),
        hare = value;
      if (done) return false;

      if (tortoise === hare) return true;
    }
    teleportDistance *= 2;
  }
  return false;
};
```

Kidu shrugged. “You know, the requirement asked for a finite space algorithm, not a constant state algorithm. Doesn’t it make sense to go with a faster finite space algorithm? There’s no benefit to constant space if finite space is sufficient.”

```
const hasCycle = (orderedCollection) => {
  const visited = new Set();

  for (let element of orderedCollection) {
    if (visited.has(element)) {
      return true;
    }
    visited.add(element);
  }
  return false;
};
```

The Carpenter stared at Kidu's solution. "I guess," he allowed, "It isn't always necessary to make a solution so awesome it would please the Ghosts of Mars."

More Iterable Mapping Operations

stateful mapping

In our interlude [The Carpenter Interviews for a Job](#), we saw the `statefulMap` operation. A `statefulMap` is a lazy map that preserves state from iteration to iteration. A normal map is stateless:

```
const mapIterableWith = (fn, iterable) =>
  ({
    [Symbol.iterator]: function* () {
      for (let element of iterable) {
        yield fn(element);
      }
    }
  });
});
```

If we write `mapIterableWith(x => x * x, [1, 2, 3, 4, 5])`, we get a lazy iterable. As we iterate over the array, we invoke the function `x => x * x` with the successive values 1, 2, 3, 4, and 5 and yield the result. Semantically, the function we pass in is intended to be stateless, to make a strict mapping from inputs to outputs.

But sometimes, we want to map the inputs to outputs that reflect the previous inputs. A very simple example would be a cumulative sum of the inputs (e.g. from `[1, 2, 3, 4, 5]` to `1, 3, 6, 10, 15`). We're mapping inputs to outputs, but the mapping function must somehow track the cumulative sum as it goes.

We can make our mapping function track state externally:

```
let sum = 0,
  cumulativeSum = x => sum += x;

Array.from(mapIterableWith(cumulativeSum, [1, 3, 5, 7, 9, 11, 13]))
//=> [1, 4, 9, 16, 25, 36, 49]
```

This first solution works, but relies on not using `sum` elsewhere, and also we cannot reuse `cumulativeSum`. If we use it more than once, we get the wrong answer, because it carries on adding to `sum`:

```
let sum = 0,
  cumulativeSum = x => sum += x;

Array.from(mapIterableWith(cumulativeSum, [1, 3, 5, 7, 9, 11, 13]))
//=> [1,4,9,16,25,36,49]

Array.from(mapIterableWith(cumulativeSum, [2, 4, 6, 8, 10, 12, 14]))
//=> [51,55,61,69,79,91,105]
```

We can solve this problem with a function that makes stateful functions. Every time we map over our iterables, we call our function maker, and get a fresh function with a fresh `sum`:

```
const cumulativeSumMaker = () => {
  let sum = 0;

  return x => sum += x;
}

Array.from(mapIterableWith(cumulativeSumMaker(), [1, 3, 5, 7, 9, 11, 13]))
//=> [1,4,9,16,25,36,49]

Array.from(mapIterableWith(cumulativeSumMaker(), [2, 4, 6, 8, 10, 12, 14]))
//=> [2,6,12,20,30,42,56]
```

As long as we're careful to always make new stateful mapping functions, this works fine.

Our function returns the same thing as its state. That isn't always the case. Here's a function that counts maps an iteration to the number of instances of each element seen so far:

```
const cumulativeTimesSeen = () => {
  let times = {};

  return x =>
    times[x]
    ? times[x] += 1
    : times[x] = 1
}

Array.from(mapIterableWith(cumulativeTimesSeen(), [1, 3, 1, 5, 1, 3, 7]))
//=> [1,1,2,1,3,2,1]
```

This business of writing stateful functions by wrapping them in a closure is quite readable, but it can be awkward. When we want to square an iteration of numbers, we just write: `mapIterableWith(x`

`=> x * x, [1, 2, 3, 4, 5]).` We don't need to set anything up, we can write short functions inline with the literal fat arrow syntax.

But if we want to set up the cumulative sum of numbers with literal inline syntax, are we really going to write:

```
mapIterableWith((( ) => {
  let sum = 0;

  return x => sum += x;
})(), [1, 3, 5, 7, 9, 11, 13])
```

Or the shorter (but slightly more subtle) `mapIterableWith(((sum = 0) => x => sum += x)(), [1, 3, 5, 7, 9, 11, 13])?`

If this were to happen more than once, we'd consider extracting this idea of stateful mapping, so that the mechanics of maintaining state are kept separate from the details of the mapping itself.

statefulMapIterableWith

If we don't want to have to build stateful scaffolding around the mapping function, we can have the mapping function manage the state. So instead of a function like `x => times[x] ? times[x] += 1 : times[x] = 1` that refers to `times` from its enclosing scope, we'll pass `times` in as a parameter: `(times, x) => times[x] ? times[x] += 1 : times[x] = 1`.

And of course, this particular stateful function updates its state with mutation, but others, like `sum`, might not. So we'll need to return the updated state as well as the value of the iteration, like this: `(times, x) => (times[x] ? times[x] += 1 : times[x] = 1, [times, times[x]])`

Now we can take our `mapIterableWith` function, and modify it to take an initial seed state, pass it to the mapping function, and extract it again using destructuring:

```
const statefulMapIterableWith = (fn, seed, iterable) =>
  ({
    *[Symbol.iterator] () {
      let value,
        state = seed;

      for (let element of iterable) {
        [state, value] = fn(state, element);
        yield value;
      }
    }
  });
});
```

Let's try it with our function that counts the number of times an item has been seen:

```
Array.from(
  statefulMapIterableWith(
    (times, x) => {
      times[x] ? times[x] += 1 : times[x] = 1;
      return [times, times[x]];
    },
    {},
    [1, 3, 1, 5, 1, 3, 7]
  )
)
//=> [1, 4, 9, 16, 25, 36, 49]
```

We can do the same thing with cumulative sums. We just have to remember to return the sum twice, as it's our state *and* our iteration value:

```
Array.from(
  statefulMapIterableWith(
    (sum, x) => [sum += x, sum],
    0,
    [1, 3, 5, 7, 9, 11, 13]
  )
)
//=> [1, 1, 2, 1, 3, 2, 1]
```

Our recipe for `statefulMapIterableWith` simplifies the use of mapping an iteration in a stateful way.

Interactive Generators

We used generators to build iterators that maintain implicit state. We saw how to use them for recursive unfolds and state machines. But there are other times we want to build functions that maintain implicit state. Let's start by looking at a very simple example of a function that can be written statefully.



Coffee and Chess

Consider, for example, the moves in a game. The moves a player makes are a stream of values, just like the contents of an array can be considered a stream of values. But of course, iterating over a stream of moves requires us to wait for the game to be over so we know what moves were made.

Let's take a look at a very simple example, [naughts and crosses](#)¹²⁰ (We really ought to do something like Chess, but that might be a little out of scope for this chapter). To save space, we'll ignore rotations and reflections, and we'll model the first player's moves as a stream.

The first player will always be o, and they will always place their chequer in the top-left corner, coincidentally numbered o:

¹²⁰<https://en.wikipedia.org/wiki/Naughts-and-crosses>

```

o |   |
---+---+---
|   |
---+---+---
|   |

```

The second player has five possible moves if we ignore reflections:

```

o | 1 | 2
---+---+---
| 4 | 5
---+---+---
|   | 8

```

Let's consider move 1. That produces this board:

```

o | x |
---+---+---
|   |
---+---+---
|   |

```

We will always play into position 6:

```

o | x |
---+---+---
|   |
---+---+---
o |   |

```

x has six possible moves, but they are really just two choices: 3 and anything else:

```

o | x | 2
---+---+---
3 | 4 | 5
---+---+---
o | 7 | 8

```

For 2, 4, 5, 7, or 8, we play 3 and win. But if x moves 3, we play 8:

```

o | x |
---+---+---
x |   |
---+---+---
o |   | o

```

x now has three significant moves: 4, 7, and anything else:

```

o | x | 2
---+---+---
x | 4 | 5
---+---+---
x | 7 | 8

```

If x plays 4, we play 7 and win. If x plays anything else, including 7, we play 4 and win.

representing naughts and crosses as a stateless function

We could play naughts and crosses as a stateless function. We encode each position of the board in some fashion, and then we build a dictionary from positions to moves. For example, the entry for:

```

o | x |
---+---+---
x |   |
---+---+---
o |   |

```

Would be 8, producing:

```

o | x |
---+---+---
x |   |
---+---+---
o |   | o

```

And the entry for:

```

o | x |
---+---+---
      | x |
---+---+---
o |   |

```

Would be 3, producing:

```

o | x |
---+---+---
o | x |
---+---+---
o |   |

```

We can encode the board in several different ways. We could use multiline strings with formatting just as we've written it here, but it is a design smell to couple presentation with modelling. Our function should be just as useful on a teletype as it would be backing a DOM game that uses a table, or a browser game that draws on Canvas.

Let's use an array. So this:

```

o | x |
---+---+---
      |   |
---+---+---
      |   |

```

Will be represented as:

```
[
  [ 'o', 'x', ' ', ],
  [ ' ', ' ', ' ', ],
  [ ' ', ' ', ' ', ],
]
```

And this:

```

o | x |
---+---+---+
x |   |
---+---+---+
o |   |

```

Will be represented as:

```
[
  [
    'o', 'x', ' ', ,
    'x', ' ', ' ', ,
    'o', ' ', ' ', ,
  ]
]
```

We can use a POJO to make a map from positions to moves. We'll use the [] notation for keys, it allows us to use any expression as a key, and JavaScript will convert it to a string. So if we write:

```
const moveLookupTable = {
  [
    [
      ' ', ' ', ' ', ' ',
      ' ', ' ', ' ', ' ',
      ' ', ' ', ' ', ' ',
    ]]: 0,
  [
    [
      'o', 'x', ' ', ,
      ' ', ' ', ' ', ,
      ' ', ' ', ' ', ,
    ]]: 6,
  [
    [
      'o', 'x', 'x',
      ' ', ' ', ' ',
      'o', ' ', ' ',
    ]]: 3,
  [
    [
      'o', 'x', ' ', ,
      'x', ' ', ' ',
      'o', ' ', ' ',
    ]]: 8,
  [
    [
      'o', 'x', ' ', ,
      ' ', 'x', ' ',
      'o', ' ', ' ',
    ]]
}
```

```
]]: 3,
[[[
  'o', 'x', ' ',
  ' ', ' ', 'x',
  'o', ' ', ' '
]]: 3,
[[[
  'o', 'x', ' ',
  ' ', ' ', ' ',
  'o', 'x', ' '
]]: 3,
[[[
  'o', 'x', ' ',
  ' ', ' ', ' ',
  'o', ' ', 'x'
]]: 3
// ...
};
```

We get:

```
{
  "o,x, , , , , ,":6,
  "o,x,x, , ,o, ,":3,
  "o,x, ,x, ,o, ,":8,
  "o,x, , ,x,o, ,":3,
  "o,x, , , ,x,o, ,":3,
  "o,x, , , ,o,x, ,":3,
  "o,x, , , , ,o, ,x":3
}
```

And if we want to look up what move to make, we can write:

```
moveLookupTable[[
  'o', 'x', ' ',
  ' ', ' ', ' ',
  'o', 'x', ' '
]]
//=> 3
```

And from there, a stateless function to play naughts-and-crosses is trivial:

```
statelessNaughtsAndCrosses([
  ['o', 'x', ' ', ''],
  [' ', ' ', ' ', ' '],
  ['o', 'x', ' ', ''],
])
//=> 3
```

representing naughts and crosses as a stateful function

Our `statelessNaughtsAndCrosses` function pushes the work of tracking the game's state onto us, the player. What if we want to exchange moves with the function? In that case, we need a stateful function. Our "API" will work like this: When we want a new game, we'll call a function that will return a game function. We'll call the game function repeatedly, passing our moves, and get the opponent's moves from it.

Something like this:

```
const aNaughtsAndCrossesGame = statefulNaughtsAndCrosses();

// our opponent makes the first move
aNaughtsAndCrossesGame()
//=> 0

// then we move, and get its next move back
aNaughtsAndCrossesGame(1)
//=> 6

// then we move, and get its next move back
aNaughtsAndCrossesGame(4)
//=> 3
```

We can build this out of our `statelessNaughtsAndCrosses` function:

```
const statefulNaughtsAndCrosses = () => {
  const state = [
    [' ', ' ', ' ', ' '],
    [' ', ' ', ' ', ' '],
    [' ', ' ', ' ', ' '],
    [' ', ' ', ' ', ' '],
  ];
  return (x = false) => {
    if (x) {
```

```

if (state[x] === ' ') {
    state[x] = 'x';
}
else throw "occupied!"
}
let o = moveLookupTable[state];
state[o] = 'o';
return o;
}
};

const aNaughtsAndCrossesGame = statefulNaughtsAndCrosses();

// our opponent makes the first move
aNaughtsAndCrossesGame()
//=> 0

// then we move, and get its next move back
aNaughtsAndCrossesGame(1)
//=> 6

// then we move, and get its next move back
aNaughtsAndCrossesGame(4)
//=> 3

```

Let's recap what we have: We have a stateful function, but we built it by wrapping a stateless function in a function that updates state based on the moves we provide. The state is encoded entirely in data.

this seems familiar

When we looked at [generators](#), we saw that some iterators are inherently stateful, but sometimes it is awkward to represent them in a fully stateless fashion. Sometimes there is a state machine that is naturally represented implicitly in JavaScript's control flow rather than explicitly in data.

We've done almost the exact same thing here with our naughts and crosses game. A game like this is absolutely a state machine, and we've explicitly coded those states into the lookup table. Which leads us to wonder: Is there a way to encode those states *implicitly*, in JavaScript control flow?

If we were in full control of the interaction, it would be easy to encode the game play as a decision tree instead of as a lookup table. For example, we could do this in a browser:

```
function browserNaughtsAndCrosses () {
  const x1 = parseInt(prompt('o plays 0, where does x play?'));
  switch (x1) {

    case 1:
      const x2 = parseInt(prompt('o plays 6, where does x play?'));
      switch (x2) {

        case 2:
        case 4:
        case 5:
        case 7:
        case 8:
          alert('o plays 3');
          break;

        case 3:
          const x3 = parseInt(prompt('o plays 8, where does x play?'));
          switch (x3) {

            case 2:
            case 5:
            case 7:
              alert('o plays 4');
              break;

            case 4:
              alert('o plays 7');
              break;
          }
      }
      break;

    // ...
  }
}
```

Naughts and crosses is simple enough that the lookup function seems substantially simpler, in part because linear code doesn't represent trees particularly well. But we can clearly see that if we wanted to, we could represent the state of the program implicitly in a decision tree.

However, our solution inverts the control. We aren't calling our function with moves, it's calling us. With iterators, we wrote a generator function using `function *`, and then used `yield` to yield

values while maintaining the implicit state of the generator's control flow.

Can we do the same thing here? At first glance, no. How do we get the player's moves to the generator function? But the first glance is deceptive, because we only see what we've seen so far. Let's see how it would actually work.

interactive generators

So far, we have called iterators (and generators) with `.next()`. But what if we pass a value to `.next()`? If we could do that, a generator function that played naughts and crosses would look like this:

If it *was* possible, how would it work?

```
function* generatorNaughtsAndCrosses () {
  const x1 = yield 0;
  switch (x1) {

    case 1:
      const x2 = yield 6;
      switch (x2) {

        case 2:
        case 4:
        case 5:
        case 7:
        case 8:
          yield 3;
          break;

        case 3:
          const x3 = yield 8;
          switch (x3) {

            case 2:
            case 5:
            case 7:
              yield 4;
              break;

            case 4:
              yield 7;
              break;
      }
  }
}
```

```

        }
    }
break;

// ...
}
}

const aNaughtsAndCrossesGame = generatorNaughtsAndCrosses();

```

We can then get the first move by calling `.next()`. Thereafter, we call `.next(...)` and pass in our moves (The very first call has to be `.next()` without any arguments, because the generator hasn't started yet. If we wanted to pass some state to the generator before it begins, we'd do that with parameters.):

```
aNaughtsAndCrossesGame.next().value
//=> 0
```

```
aNaughtsAndCrossesGame.next(1).value
//=> 6
```

```
aNaughtsAndCrossesGame.next(3).value
//=> 8
```

```
aNaughtsAndCrossesGame.next(7).value
//=> 4
```

Our generator function maintains state implicitly in its control flow, but returns an iterator that we call, it doesn't call us. It isn't a collection, it has no meaning if we try to spread it into parameters or as the subject of a `for...of` block.

But the generator function allows us to maintain state implicitly. And sometimes, we want to use implicit state instead of explicitly storing state in our data.

summary

We have looked at generators as ways of making iterators over static collections, where state is modelled implicitly in control flow. But as we see here, it's also possible to use a generator interactively, passing values in and receiving a value in return, just like an ordinary function.

Again, the salient difference is that an "interactive" generator is stateful, and it embodies its state in its control flow.

Basic Operations on Iterables

Here are the operations we've defined on Iterables. As discussed, they preserve the collection semantics of the iterable they are given:

operations that transform one iterable into another

```
const mapIterableWith = (fn, iterable) =>
  ({
    [Symbol.iterator]: function* () {
      for (let element of iterable) {
        yield fn(element);
      }
    }
  });
}

const mapAllIterableWith = (fn, iterable) =>
  ({
    [Symbol.iterator]: function* () {
      for (let element of iterable) {
        yield* fn(element);
      }
    }
  });
}

const filterIterableWith = (fn, iterable) =>
  ({
    [Symbol.iterator]: function* () {
      for (let element of iterable) {
        if (!!fn(element)) yield element;
      }
    }
  });
}

const compactIterable = (iterable) =>
  ({
    [Symbol.iterator]: function* () {
      for (let element of iterable) {
        if (element != null) yield element;
      }
    }
  });
}
```

```

const untilIterableWith = (fn, iterable) =>
  ({
    [Symbol.iterator]: function* () {
      for (let element of iterable) {
        if (fn(element)) break;
        yield fn(element);
      }
    }
  });
}

const restOfIterable = (iterable) =>
  ({
    [Symbol.iterator]: () => {
      const iterator = iterable[Symbol.iterator]();

      iterator.next();
      return iterator;
    }
  });
}

const take = function* (numberToTake, iterable) {
  let remaining = numberToTake;

  for (let value of iterable) {
    if (remaining-- <= 0) break;
    yield value;
  }
}

```

operations that compose two or more iterables into an iterable

```

const zipIterables = (...iterables) =>
  ({
    [Symbol.iterator]: function * () {
      const iterators = iterables.map(i => i[Symbol.iterator]());

      while (true) {
        const pairs = iterators.map(j => j.next()),
          done = pairs.map(p => p.done),
          values = pairs.map(p => p.value);
      }
    }
  });

```

```
    if (dones.indexOf(true) >= 0) break;
    yield values;
}
}
});
```

operations that transform an iterable into a value

```
const reduceIterableWith = (fn, seed, iterable) => {
  let accumulator = seed;

  for (let element in iterable) {
    accumulator = fn(accumulator, element);
  }
  return accumulator;
};

const firstOfIterable = (iterable) =>
  iterable[Symbol.iterator]().next().value;
```

A Coffeehouse: Symbols



“Uniqueness” is an important quality in society and in programs.

Programmers often spend a lot of time trying to define “sameness:” JavaScript programmers know that `"foo" === "foo"` is always true, but `new String("foo") === new String("foo")` is always false, and how tricky it is to define what we mean when we say that `{ foo: "bar" }` is *semantically equivalent* to `{ foo: "bar" }`.

Programmers don’t think about it quite as much, but entities being different from each other is also important. We know that `function () {} !== function () {}`. But having objects that we know to be different from each other can be very useful.

Any sufficiently complicated C or Fortran program contains an ad hoc, informally-specified, bug-ridden, slow implementation of half of Common Lisp.—[Greenspun’s](#)

Tenth Rule¹²¹

In older versions of JavaScript, programmers would hack together unique objects, using timestamps, GUIDS, counters and other techniques. None of which are individually wrong, but when there are 99 different ways to do the same thing that everybody ends up doing, the important parts of our code become obfuscated under the weight of our ad hoc, informally-specified, bug-ridden, slow implementations of Common Lisp's [gensym](#)¹²².

So `Symbol` was added to the language. In its simplest form, `Symbol` is a function that returns a unique entity. No two symbols are alike, ever:

```
Symbol() !== Symbol()
```

Symbols have string representations, although they may appear cryptic:¹²³

```
Symbol().toString()
//=> Symbol(undefined)_u.mwf0b1vw5
Symbol().toString()
//=> Symbol(undefined)_s.nik1xrko8m
Symbol().toString()
//=> Symbol(undefined)_s.mbsi4nduh
```

You can add your own text to help make it intelligible:

```
Symbol("Allongé").toString()
//=> Symbol(Allongé)_s.52x692eab
Symbol("Allongé").toString()
//=> Symbol(Allongé)_s.q6hq51x01p
Symbol("Allongé").toString()
//=> Symbol(Allongé)_s.jii7eyiyza
```

There are some ways that JavaScript makes symbols especially handy. Using symbols as property names, for example.

privacy with symbols

When we use a symbol as a property name, it is automatically unique and non-enumerable. It is still possible to discover its existence and retrieve its value, but it is not possible for accidentally access or overwrite a property that uses a symbol as its key.

Therefore, we can give objects private properties with symbols. Consider this:

¹²¹[https://en.wikipedia.org/wiki/Greenspun's_tenth_rule](https://en.wikipedia.org/wiki/Greenspun%27s_tenth_rule)

¹²²<http://www.lispdoc.com/?q=gensym>

¹²³The exact representation depends upon the implementation

```

const Queue = () =>
  ({
    array: [],
    head: 0,
    tail: -1,
    pushTail (value) {
      return this[array][this[tail] += 1] = value
    },
    pullHead () {
      if (this[tail] >= this[head]) {
        let value = this[array][this[head]];
        this[array][this[head]] = undefined;
        this[head] += 1;
        return value
      }
    },
    isEmpty () {
      return this[tail] < this[head]
    }
  });
}

let q = Queue();
q.pushTail('hello');
q.pushTail('symbols');

q.pullHead()
 //=> 'hello'

q
 //=> {"array": ["hello", "symbols"], "head": 0, "tail": 1}

q.tail
 //=> 1

```

Because we used compact method syntax, the `pushTail`, `pullHead`, and `isEmpty` properties are not “enumerable,” so they don’t show up in the console. But other code can access them. The `array`, `head`, and `tail` properties are enumerable and accessible.

Let’s use symbols for these properties instead:

```

const array = Symbol(),
  head  = Symbol(),
  tail  = Symbol();

const Queue = () =>
  ({
    [array]: [],
    [head]: 0,
    [tail]: -1,
    pushTail (value) {
      return this[array][this[tail] += 1] = value
    },
    pullHead () {
      if (this[tail] >= this[head]) {
        let value = this[array][this[head]];
        this[array][this[head]] = undefined;
        this[head] += 1;
        return value
      }
    },
    isEmpty () {
      return this[tail] < this[head]
    }
  });
}

let q = Queue();
q.pushTail('hello');
q.pushTail('symbols');

q.pullHead()
//=> 'hello'

q
//=> {}

q.tail
//=> undefined

```

Now the `array`, `head`, and `tail` properties are not enumerable and they aren't accessible by those names because they're actually symbols assigned to the `array`, `head`, and `tail` variables.

Life on the Plantation: Metaobjects



124

(Note: The material in this section has been condensed and adapted from the book [JavaScript Spessore¹²⁵](#))

¹²⁴Krups Machines (c) 2010 Shadow Becomes White, some rights reserved

¹²⁵<https://leanpub.com/javascript-spessore>

Why Metaobjects?



In computer science, a metaobject is an object that manipulates, creates, describes, or implements other objects (including itself). The object that the metaobject is about is called the base object. Some information that a metaobject might store is the base object's type, interface, class, methods, attributes, parse tree, etc.

–Wikipedia¹²⁶

It is possible to write software using objects alone. When we need behaviour for an object, we can give it methods by binding functions to keys in the object:

```
const sam = {
  firstName: 'Sam',
  lastName: 'Lowry',
  fullName () {
    return this.firstName + " " + this.lastName;
  },
  rename (first, last) {
    this.firstName = first;
    this.lastName = last;
    return this;
  }
}
```

We call this a “naïve” object. It has state and behaviour, but it lacks division of responsibility between its state and its behaviour.

This lack of separation has two drawbacks. First, it intermingles properties that are part of the model domain (such as `firstName`), with methods (and possibly other properties, although none are shown here) that are part of the implementation domain. Second, when we needed to share common behaviour, we could have objects share common functions, but does it not scale: There's no sense of organization, no clustering of objects and functions that share a common responsibility.

Metaobjects solve the lack-of-separation problem by separating the domain-specific properties of objects from their implementation-specific properties and the functions that represent their behaviour.

The basic principle of the metaobject is that we separate the mechanics of behaviour from the domain properties of the base object. This has immediate engineering benefits, and it's also the foundation for designing programs with formal classes, expectations, and delegation.

¹²⁶<https://en.wikipedia.org/wiki/Metaobject>

Mixins, Forwarding, and Delegation

The simplest possible metaobject in JavaScript is a *mixin*. Consider our naïve object:

```
const sam = {
  firstName: 'Sam',
  lastName: 'Lowry',
  fullName () {
    return this.firstName + " " + this.lastName;
  },
  rename (first, last) {
    this.firstName = first;
    this.lastName = last;
    return this;
  }
}
```

We can separate its domain properties from its behaviour:

```
const sam = {
  firstName: 'Sam',
  lastName: 'Lowry'
};

const Person = {
  fullName () {
    return this.firstName + " " + this.lastName;
  },
  rename (first, last) {
    this.firstName = first;
    this.lastName = last;
    return this;
  }
};
```

And use `Object.assign` to mix the behaviour in:

```
Object.assign(sam, Person);

sam.rename
//=> [Function]
```

This allows us to separate the behaviour from the properties in our code.

Our Person object is a *mixin*, it provides functionality to be mixed into an object with a function like `Object.assign`. Mixins are not “copied” into objects in the sense of making brand new versions of each of their functions: `Object.assign` copies *references* to each function from the mixin into the target object.

We can test this for ourselves:

```
sam.fullName === Person.fullName
//=> true

sam.rename === Person.rename
//=> true
```

If we want to use the same behaviour with another object, we can do that:

```
const peck = {
  firstName: 'Sam',
  lastName: 'Peckinpah'
};

Object.assign(peck, Person);
```

And of course, that object gets references to the original functions as well:

```
sam.fullName === peck.fullName
//=> true

sam.rename === peck.rename
//=> true
```

Thus, many objects can mix one object in.

Things get even better: One object can mix many objects in:

```

const HasCareer = {
  career () {
    return this.chosenCareer;
  },
  setCareer (career) {
    this.chosenCareer = career;
    return this;
  }
};

Object.assign(peck, Person, HasCareer);

peck.setCareer('Director');

```

Since many objects can all mix the same object in, and since one object can mix many objects into itself, there is a *many-to-many* relationship between objects and mixins.

forwarding

Another way to build a metaobject that defines behaviour for another object is by having the object *forward* one or more method calls to a metaobject.

```

function forward (receiver, metaobject, ...methods) {
  methods.forEach(function (methodName) {
    receiver[methodName] = (...args) => metaobject[methodName](...args)
  });

  return receiver;
};

```

This function *forwards* methods to another object. Any other object, it could be a metaobject specifically designed to define behaviour, or it could be a domain object that has other responsibilities. Like mixins, one object might forward method invocations to more than one metaobject.

In this example, we start with an investment portfolio metaobject that has a *netWorth* method:

```
const portfolio = (function () {
  const investments = Symbol();

  return {
    [investments]: [],
    addInvestment (investment) {
      this[investments].push(investment);
    },
    netWorth () {
      return this[investments].reduce(
        function (acc, investment) {
          return acc + investment.value;
        },
        0
      );
    }
  };
})();
```

And next we create an investor who has a portfolio of investments:

```
const investor = forward({}, portfolio, "addInvestment", "netWorth");

investor.addInvestment({ type: "art", value: 1000000 })
investor.addInvestment({ type: "art", value: 2000000 })
investor.netWorth()
//=> 3000000
```

forwarding

Forwarding is a relationship between an object that receives a method invocation receiver and a provider object. They may be peers. The provider may be contained by the consumer. Or perhaps the provider is a metaobject.

When forwarding, the provider object has its own state. There is no special binding of function contexts, instead the consumer object has its own methods that forward to the provider and return the result. Our `forward` function above handles all of that, iterating over the provider's properties and making forwarding methods in the consumer.

The key idea is that when forwarding, the provider object handles each method *in its own context*. And because there is a forwarding method in the consumer object and a handling method in the provider, the two can be varied independently. Each forwarding function invokes the method in the provider *by name*. So we can do this:

```
portfolio.netWorth = function () {
  return "I'm actually bankrupt!";
}
```

We're overwriting the method in the `portfolio` object, but not the forwarding function. So now, our `investor` object will forward invocations of `netWorth` to the new function, not the original.

We say that mixing in is “early bound,” while forwarding is “late bound.” We’ll look up the method when it’s invoked.

shared forwarding

The premise of a mixin is that every time you mix the metaobject’s behaviour into an object, the receiver holds the state for the behaviour being mixed in. Thus, you can mix the same metaobject into many objects, and they each will have their own state.

Forwarding does not work this way. When objects A and B both forward to C, the private state for C is held in C, and thus A and B share state. Sometimes this is what we want. but if it isn’t, we must be very careful about using forwarding.

summarizing what we know so far

So now we have two things: Mixing in a mixin, and forwarding to a first-class object. And we’ve seen that mixins *execute in the context of the receiver*, but forwarding is *late-bound*.

Which provokes a question: What is evaluated in the receiver’s context, but late-bound, not early-bound?

delegation

Let’s build it. Here’s a version of the `forward` function, modified to evaluate method invocation in the receiver’s context:

```
function delegate (receiver, metaobject, ...methods) {
  methods.forEach(function (methodName) {
    receiver[methodName] = (...args) => metaobject[methodName].apply(receiver, args)
  });
  return receiver;
};
```

This new `delegate` function does exactly the same thing as the `forward` function, but the line that does the delegation looks like this:

```
receiver[methodName] = (...args) => metaobject[methodName].apply(receiver, args)
```

It uses the receiver as the context instead of the provider. This has all the same coupling implications that our mixins have, of course. And it layers in additional indirection. But unlike a mixin and like forwarding, the indirection gives us some late binding, allowing us to modify the metaobject's methods *after* we have delegated behaviour from a receiver to it.

delegation vs. forwarding

Delegation and forwarding are both very similar. One metaphor that might help distinguish them is to think of receiving an email asking you to donate some money to a worthy charity.

- If you *forward* the email to a friend, and the friend donates money, the friend is donating their own money and getting their own tax receipt.
- If you *delegate* responding to your accountant, the accountant donates *your* money to the charity and you receive the tax receipt.

In both cases, the other entity does the work when you receive the email.

Later Binding

When comparing Mixins to Delegation, we noted that Mixins are early bound and Delegation is late bound. Let's be specific. Given:

```
const Incrementor = {
  increment () {
    ++this._value;
    return this;
  },
  value (optionalValue) {
    if (optionalValue != null) {
      this._value = optionalValue;
    }
    return this._value;
  }
};

const counter = Object.assign({}, Incrementor);
```

We are mixing Incrementor into counter. At some point later, we encounter:

```
counter.value(42);
```

What function handles the invocation of .value? because we mixed Incrementor into counter, it's the same function as Incrementor.counter. We don't look that up when counter.value(42) is evaluated, because that was bound to counter.value when we extended counter. This is early binding.

However, given:

```
const counter = {};
delegate(counter, Incrementor);

// ...time passes...

counter.value(42);
```

We again are most likely invoking Incrementor.value, but now we are determining this *at the time counter.value(42) is evaluated*. We bound the target of the delegation, Incrementor, to

counter, but we are going to look the actual property of `Incrementor.value` up when it is invoked. This is late binding, and it is useful in that we can make some changes to `Incrementor` after the delegation has been set up, perhaps to add some logging.

It is very nice not to have to do things like this in a very specific order: When things have to be done in a specific order, they are *coupled in time*. Late binding is a decoupling technique.

but wait, there's more

But we can get *even later than that*. Although the specific function is late bound, the target of the delegation, `Incrementor`, is early bound. We can late bind that too! Here's a variation on `delegate`:

```
function delegateToOwn (receiver, propertyName, ...methods) {
  methods.forEach(function (methodName) {
    receiver[methodName] = function () {
      const metaobject = receiver[propertyName];
      return metaobject[methodName].apply(receiver, arguments);
    };
  });
}

return receiver;
};
```

This function sets things up so that an object can delegate to one of its own properties, instead of an arbitrary object. It's quite common for an object to forward methods to one of its own properties. In this manner, objects can be constructed using *composition*.

Let's take another look at the investor example. Here's the `portfolio` we used before, modified to use the receiver's context like a mixin:

```
const portfolio = (function () {
  const investmentsProperty = Symbol();

  return {
    addInvestment (investment) {
      this[investmentsProperty] || (this[investmentsProperty] = []);
      return this[investmentsProperty].push(investment);
    },
    netWorth () {
      this[investmentsProperty] || (this[investmentsProperty] = []);
      return this[investmentsProperty].reduce(
        function (acc, investment) {
          return acc + investment.value;
        }
      );
    }
  };
});
```

```

    },
    0
);
}
};

})();

```

Next we'll make that a property of our investor, and delegate to the `nestEgg` property by name, not the object itself:

```

const investor = {
  nestEgg: portfolio
}

delegateToOwn(investor, 'nestEgg', 'addInvestment', 'netWorth');

investor.addInvestment({ type: "art", value: 1000000 })
investor.addInvestment({ type: "art", value: 2000000 })
investor.netWorth()
//=> 3000000

```

Our `investor` object delegates the `addInvestment` and `netWorth` methods to its own `nestEgg` property. So far, this is just like the `delegate` method above. But consider what happens if we decide to assign a new portfolio to our investor:

```

const companyRetirementPlan = {
  netWorth () {
    return 1500000;
  }
}

investor.nestEgg = companyRetirementPlan;

investor.netWorth()
//=> 1500000

```

The `delegateToOwn` delegation now delegates to `companyRetirementPlan`, because it is bound to the property name, not to the original object. This seems questionable for portfolios—what happens to the old portfolio when you assign a new one?—but has tremendous application for modeling classes of behaviour that change dynamically.

state machines

A very common use case for this delegation is when building [finite state machines](#)¹²⁷. As described in the book [Understanding the Four Rules of Simple Design](#)¹²⁸ by Corey Haines, you *could* implement [Conway's Game of Life](#)¹²⁹ using `if` statements. Hand waving furiously over other parts of the system, you might get:

```
const Universe = {
  // ...
  numberofNeighbours (location) {
    // ...
  }
};

const thisGame = Object.assign({}, Universe);

const Cell = {
  alive () {
    return this._alive;
  },
  numberofNeighbours () {
    return thisGame.numberofNeighbours(this._location);
  },
  aliveInNextGeneration () {
    if (this.alive()) {
      return (this.numberofNeighbours() === 3);
    }
    else {
      return (this.numberofNeighbours() === 2 || this.numberofNeighbours() === 3);
    }
  }
};

const someCell = Object.assign({
  _alive: true,
  _location: {x: -15, y: 12}
}, Cell);
```

¹²⁷https://en.wikipedia.org/wiki/Finite-state_machine

¹²⁸<https://leanpub.com/4rulesof simplesdesign>

¹²⁹https://en.wikipedia.org/wiki/Conway%27s_Game_of_Life

One of the many insights from [Understanding the Four Rules of Simple Design¹³⁰](#) is that this business of having an `if (alive())` in the middle of a method is a hint that cells are stateful.

We can extract this into a state machine using delegation to a property:

```
const Alive = {
  alive () {
    return true;
  },
  aliveInNextGeneration () {
    return (this.numberOfNeighbours() === 3);
  }
};

const Dead = {
  alive () {
    return false;
  },
  aliveInNextGeneration () {
    return (this.numberOfNeighbours() === 2 || this.numberOfNeighbours() === 3);
  }
};

const FsmCell = {
  numberOfNeighbours () {
    return thisGame.numberOfNeighbours(this._location);
  }
}

delegateToOwn(Cell, '_state', ['alive', 'aliveInNextGeneration']);

const someFsmCell = Object.assign({
  _state: Alive,
  _location: {x: -15, y: 12}
}, FsmCell);
```

`someFsmCell` delegates `alive` and `aliveInNextGeneration` to its `_state` property, and you can change its state with assignment:

```
someFsmCell._state = Dead;
```

¹³⁰<https://leanpub.com/4rulesof simplesdesign>

In practice, states would be assigned en masse, but this demonstrates one of the simplest possible state machines. In the wild, most business objects are state machines, sometimes with multiple, loosely coupled states. Employees can be:

- In or out of the office;
- On probation, on contract, or permanent;
- Part time or full time.

Delegation to a property representing state takes advantage of late binding to break behaviour into smaller components that have cleanly defined responsibilities.

late bound forwarding

The exact same technique can be used for forwarding to a property, and forwarding to a property can also be used for some kinds of state machines. Forwarding to a property has lower coupling than delegation, and is preferred where appropriate.

Delegation via Prototypes

At this point, we're discussed separating behaviour from object properties using metaobjects while avoiding discussion of prototypes. This is deliberate, because what we have achieved is developing a vocabulary for describing what a prototype is.

Let's review what we know so far:

```
const Person = {
  fullName: function () {
    return this.firstName + " " + this.lastName;
  },
  rename: function (first, last) {
    this.firstName = first;
    this.lastName = last;
    return this;
  }
};
```

So far, just like any other metaobject we'd use as a mixin, or perhaps with delegation.

```
const sam = Object.create(Person);
sam
//=> {}
```

This is different. Instead of creating an object and then using `Object.assign` to incorporate behaviour from a metaobject, we're using `Object.create`, a built-in method that creates the object while simultaneously associating it with a prototype.

The methods `fullName` and `rename` do not appear in its string representation. We'll find out why in a moment.

```
sam.fullName
//=> [Function]
sam.rename
//=> [Function]
sam.rename('Samuel', 'Ballard')
//=> { firstName: 'Samuel', lastName: 'Ballard' }
sam.fullName()
//=> 'Samuel Ballard'
```

And yet, they appear to be properties of `sam`, and we can invoke them in the usual fashion. Furthermore, we can tell that when the methods are invoked, the current context is being set to the receive, `sam`: That's why invoking `rename` sets `sam.firstName` and `sam.lastName`.

So far this is almost identical to using a mixin or delegation, but not a private mixin or forwarding because methods are evaluated in `sam`'s scope. The only difference appears to be how `sam` is displayed in the console. We recall that the big difference between a mixin and delegation is whether the methods are early or late bound.

So, if we *change* a method in `Person`, then if prototypes are early bound, `sam`'s behaviour will not change. Whereas if methods are late bound, `sam`'s behaviour will change. Let's try it:

```
Person.fullName = function () {
  return this.lastName + ', ' + this.firstName;
};

sam.fullName()
//=> 'Ballard, Samuel'
```

Aha! Prototypes have *delegation semantics*: They are late bound, and evaluated in the receiver's context. This is exactly why many programmers say that prototypes are a delegation mechanism.

We've already seen delegation implemented via *method proxies*. Now we see it implemented via prototypes.

prototypes are strictly many-to-one.

Delegation is a many-to-many relationship. One object can directly delegate to more than one metaobject, and more than one object can directly delegate to the same metaobject. This is not the case with prototypes: `Object.create` only allows you to specific *one* prototype for an object you're creating. You can *change* the prototype of an Object with `Object.setPrototypeOf`, but each object can only have one prototype at a time.

sharing prototypes

Several objects can share one prototype:

```
const sam = Object.create(Person);
const saywhatagain = Object.create(Person);
```

`sam` and `saywhatagain` both share the `Person` prototype, so they both share the `rename` and `fullName` methods. But they each have their own properties, so:

```

sam.rename('Samuel', 'Ballard');
saywhatagain.rename('Samuel', 'Jackson');

sam.fullName()
//=> 'Samuel Ballard'

saywhatagain.fullName()
//=> 'Samuel Jackson'

```

The limitation of this scheme becomes apparent when we consider behaviours that need to be composed. Given Person, IsAuthor, and HasBooks, if we have some people that are authors, some that have children, some that aren't authors and don't have children, and some authors that have children, prototypes cannot directly manage these behaviours without duplication.

prototypes are open for extension

With forwarding and delegation, the body of the method being proxied is late-bound because it is looked up by name when the method is invoked. This differs from mixins, where the body of the method is early bound by reference at the time the metaobject is mixed in.

```

const methodNames = (object) =>
  Object.keys(object).filter(key => typeof(object[key]) == 'function')

function delegate (receiver, metaobject, ...methods = methodNames(metaobject)) {
  methods.forEach(function (methodName) {
    receiver[methodName] = (...args) => metaobject[methodName].apply(receiver, args)
  });
}

return receiver;
};

const lowry = {};
delegate(lowry, Person);
lowry.rename('Sam', 'Lowry');

Person.fullName = function () {
  return this.firstName[0] + '.' + this.lastName;
};
lowry.fullName();
//=> 'S. Lowry'

```

Prototypes and delegation both allow you to change the body of a method after a metaobject has been bound to an object.

But what happens if we add an entirely new method to our metaobject?

```
Person.surname = function () {  
    return this.lastName;  
}
```

An object using our method proxies *does not* delegate the new method to its metaobject, because we never created a method proxy for `surname`:

```
lowry.surname()  
//=> TypeError: Object #<Object> has no method 'surname'
```

Whereas, an object using a prototype *does* delegate the new method to the prototype:

```
sam.surname()  
//=> 'Ballard'
```

Prototypes late bind the method bodies *and* they late bind the identities of the methods being delegated. So you can add and remove methods to a prototype, and the behaviour of all of the objects bound to that prototype will be changed.

We say that *prototypes are open for extension*, because you can extend their behaviour *after* creating objects with them. We say that *mixins are closed for extension*, because behaviour added to a mixin does not change any of the objects that have already incorporated it.

summarizing

Prototypes are a special kind of delegation mechanism that is built into JavaScript. Delegating through prototypes is:

1. Late bound on method bodies, just like delegation through method proxies;
2. Late bound on the method identities, which is superior to delegation through method proxies;
3. Evaluated in the receiver's context, just like delegation.
4. Open for extension, unlike mixins, forwarding, and explicit delegation.

Prototypes are usually the first form of metaobject that many developers learn in JavaScript, and quite often the last.

...one more thing!

There is one more way that delegation via prototypes differs from delegation via method proxies, and it's very important. We recall from above that object delegating to a prototype appear differently in the console than objects delegating via method proxies:

```
sam
//=> { firstName: 'Samuel', lastName: 'Ballard' }

lowry
//=>
{ fullName: [Function],
  rename: [Function],
  firstName: 'Sam',
  lastName: 'Lowry' }
```

The reason is very simple: The code for representing an object in the console iterates over its “own” properties, properties that belong to the object itself and not its prototype. In the case of `sam`, those are `firstName` and `lastName`, but not `fullName` or `rename` because those are properties of the prototype.

Whereas in the case of `lowry`, `fullName` and `rename` are properties of `Person`, but there are also function proxies that are properties of the `lowry` object itself.

We can test this for ourselves using the `.hasOwnProperty` method:

```
sam.hasOwnProperty('fullName');
//=> false
lowry.hasOwnProperty('fullName');
//=> true
```

One of the goals of metaobjects is to separate domain properties (such as `firstName`) from behaviour (such as `.fullName()`). All of our metaobject techniques allow us to do that in our written code, but prototypes do this extremely effectively in the runtime structure of the objects themselves.

This is extremely useful.

Shared Prototypes

We can create a very simple object and associate it with a prototype:

```
const Person = {
  fullName () {
    return this.firstName + " " + this.lastName;
  },
  rename (first, last) {
    this.firstName = first;
    this.lastName = last;
    return this;
  }
};

const sam = Object.create(Person);
```

This associates behaviour with our object:

```
sam.rename('sam', 'hill');
sam.fullName();
//=> 'sam hill'
```

There is no way to associate more than one prototype with the same object, but we can associate more than one object with the same prototype:

```
const bewitched = Object.create(Person);
bewitched.rename('Samantha', 'Stephens');
```

Although they share the prototype, their individual properties (as access with `this`), are separate:

```
sam
//=> { firstName: 'sam', lastName: 'hill' }
bewitched
//=> { firstName: 'Samantha', lastName: 'Stephens' }
```

This is very convenient.

prototype chains

Consider our `HasCareer` mixin:

```
const HasCareer = {
  career () {
    return this.chosenCareer;
  },
  setCareer (career) {
    this.chosenCareer = career;
    return this;
  }
};
```

We can use it as a prototype, of course. But we already want to use Person as a prototype. What can we do? Obviously, we can combine Person and HasCareer into a “fat prototype” called PersonWithCareer. This is not great, a general principle of software is that entities should have a single clearly defined responsibility.

Even if we weren’t hung up on single responsibility, another issue is that not all people have careers, so we need one prototype for people, and another for people with careers.

The catch is, another principle of good design is that every piece of knowledge should have one unambiguous representation. The knowledge of what makes a person falls into this category. If we were to add another method to Person, would we remember to add it to PersonWithCareer?

Let’s work from two principles:

1. Any object can have an object as its prototype, and any object can be a prototype.
2. The behaviour of an object consists of all of its own behaviour, plus all the behaviour of its prototype.

When we say *any* object can have a prototype, does that include objects used as prototypes? Yes. Objects used as prototypes can have prototypes of their own.

Let’s try it. First things first:

```
const PersonWithCareer = Object.create(Person);
```

Now let’s mix HasCareer into PersonWithCareer:

```
Object.assign(PersonWithCareer, HasCareer);
```

And now we can use PersonWithCareer as a prototype:

```
const goldie = Object.create(PersonWithCareer);
goldie.rename('Samuel', 'Goldwyn');
goldie.setCareer('Producer');
```

Why does this work?

Imagine that we were writing a JavaScript (or other OO) language implementation. Method invocation is incredibly messy, full of special optimizations and so forth, but perhaps we only have ten days to get it done, so we might proceed like this without even thinking about prototype chains:

```
function invokeMethod(receiver, methodName, listOfArguments) {
  return invokeMethodWithContext(receiver, receiver, methodName, listOfArguments\
);
}

function invokeMethodWithContext(context, receiver, methodName, listOfArguments)\
{
  const prototype;

  if (receiver.hasOwnProperty(methodName)) {
    return receiver[methodName].apply(context, listOfArguments);
  }
  else if (prototype = Object.getPrototypeOf(receiver)) {
    return invokeMethodWithContext(context, prototype, methodName, listOfArgumen\
ts);
  }
  else {
    throw 'Method Missing ' + methodName;
  }
}
```

Very simple: If the object implements the method, invoke it with `.apply`. If the object doesn't implement it but has a prototype, ask the prototype to implement it in the original receiver's context.

What if the prototype doesn't implement it but has a prototype of its own? Well, we'll recursively try that object too. Conceptually, this is what happens when we write:

```
goldie.fullName()
//=> 'Samuel Goldwyn'
```

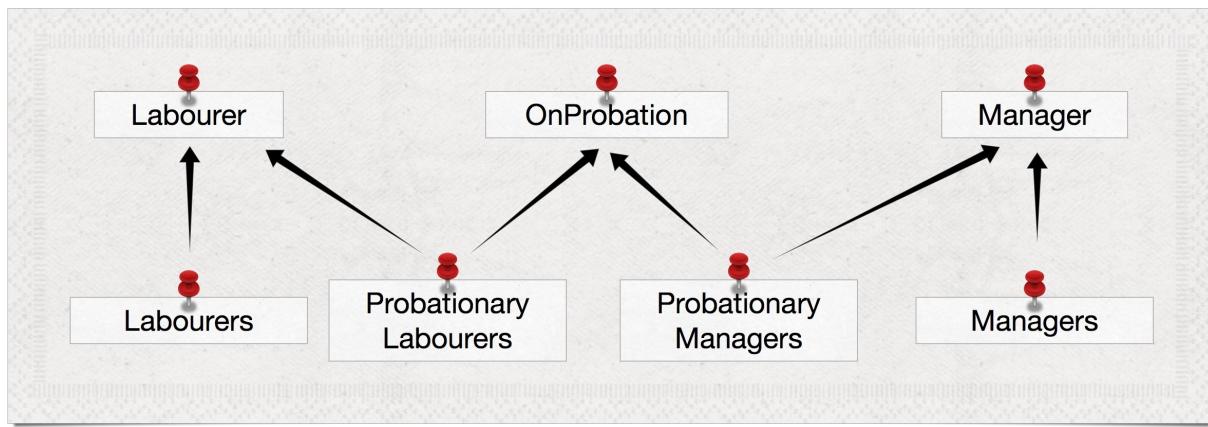
In theory, the JavaScript engine walks up a chain starting with the `goldie` object, followed by our `PersonWithCareer` prototype followed by our `Person` prototype.

trees

Chaining prototypes is a useful technique, however it has some limitations. Because objects can have only one prototype, you cannot model all combinations of responsibilities solely with prototype chains. The classic example is known as “The W Pattern.”

Let’s consider three prototypes to be used for employees in a factory: Laborer, Manager, and OnProbation.

All employees are either Laborer or Manager, but not both. So far, very easy, they can be prototypes. Some labourers are also on probation, as are some managers. How do we handle this with prototype chains?



Labourers, Managers, and OnProbation

Well, we can’t have Laborer or Manager share OnProbation as a prototype, because then *all* labourers and managers would be on probation. And if we make OnProbation have Laborer as its prototype, there’s no way to have a manager also be on probation without making it also a laborer, and that’s not allowed.

Quite simply, a tree is an insufficient mechanism for modeling this relationship.

Prototype chains model trees, but most domain responsibilities cannot be represented as trees, so we must either revert to using “fat prototypes,” or find another way to represent responsibilities, such as mixing metaobjects into prototypes.

prototypes and mixins

We’ve seen how to use `Object.assign` to mix functionality directly into objects. Prototypes are objects, so it follows that we can mix functionality into prototypes. We used this technique when we created the `PersonWithCareer` shared prototype.

We can extend this technique when we run into limitations with prototype chains:

```
const Laborer = {  
  // ...  
};  
const Manager = {  
  // ...  
};  
const Probationary = {  
  // ...  
};  
  
const LaborerOnProbation = Object.assign({}, Laborer, Probationary);  
const ManagerOnProbation = Object.assign({}, Manager, Probationary);
```

Using mixins, we have created prototypes that model combining labor/management with probationary status.

caveat programmer

Whether we're using prototype chains or mixins, we're introducing coupling. As discussed in [Mixins, Forwarding, and Delegation](#), prototypes that are brought into proximity with each other (by placing them anywhere in the same chain, or by mixing them into the same object) become deeply coupled because they both have complete access to an object's private internal state through `this`.

To reduce this coupling, we have to find a way to insulate prototypes from each other. Techniques like forwarding, while straightforward to use directly on an object or through a singleton prototype, require special handling when used in a shared prototype.

We'll discuss this at more length when we look at classes.

Decaffeinated: Impostors



Decaf espresso

Now that we've explored objects in some depth, it's time to acknowledge something that even small children know: Everything in JavaScript behaves like an object, everything in JavaScript behaves like an instance of a function, and therefore everything in JavaScript behaves as if it inherits some methods from a prototype and/or has some elements of its own.

For example:

```

3.14159265.toPrecision(5)
//=> '3.1415'

'FORTRAN, SNOBOL, LISP, BASIC'.split(',')
//=> [ 'FORTRAN',
#      'SNOBOL',
#      'LISP',
#      'BASIC' ]

[ 'FORTRAN',
'SNOBOL',
'LISP',
'BASIC' ].length
//=> 4

```

Functions themselves are instances, and they have methods. For example, every function has a method `call`. `call`'s first argument is a *context*: When you invoke `.call` on a function, it invokes the function, setting `this` to the context. It passes the remainder of the arguments to the function. It seems like objects are everywhere in JavaScript!

You may have noticed that we use “weasel words” to describe how everything in JavaScript *behaves like* an object. Everything *behaves as if* it delegates behaviour to a prototype.

The full explanation is this: As you know, JavaScript has “value types” like `String`, `Number`, and `Boolean`. As noted in the first chapter, value types are also called *primitives*, and one consequence of the way JavaScript implements primitives is that they aren't objects. Which means they can be identical to other values of the same type with the same contents, but the consequence of certain design decisions is that value types don't actually have methods or prototypes.

So. Value types don't have methods or prototypes. And yet:

```

"Spence Olham".split(' ')
//=> ["Spence", "Olham"]

```

Somehow, when we write `"Spence Olham".split(' ')`, the string `"Spence Olham"` isn't an object, it doesn't have methods, but it does a damn fine job of impersonating an object with a `String` prototype. How does `"Spence Olham"` impersonate an object?

JavaScript pulls some legerdemain. When you do something that treats a value like an object, JavaScript checks to see whether the value actually is an object. If the value is actually a primitive,¹³¹ JavaScript temporarily makes an object that is a kinda-sorta copy of the primitive and that kinda-sorta copy has methods and you are temporarily fooled into thinking that `"Spence Olham"` has a `.split` method.

¹³¹Recall that `Strings`, `Numbers`, `Booleans` and so forth are value types and primitives. We're calling them primitives here.

These kinda-sorta copies are called *String instances* as opposed to *String primitives*. And the instances have methods, while the primitives do not. How does JavaScript make an instance out of a primitive? With `new`, of course.¹³² Let's try it:

The string instance looks just like our string primitive. But does it behave like a string primitive? Not entirely:

```
new String("Spence Olham") === "Spence Olham"
//=> false
```

Aha! It's an object with its own identity, unlike string primitives that behave as if they have a canonical representation. If we didn't care about their identity, that wouldn't be a problem. But if we carelessly used a string instance where we thought we had a string primitive, we could run into a subtle bug:

```
if (userName === "Spence Olham") {
  getMarried();
  goCamping()
}
```

That code is not going to work as we expect should we accidentally bind `new String("Spence Olham")` to `userName` instead of the primitive `"Spence Olham"`.

This basic issue that instances have unique identities but primitives with the same contents have the same identities—is true of all primitive types, including numbers and booleans: If you create an instance of anything with `new`, it gets its own identity.

There are more pitfalls to beware. Consider the truthiness of string, number and boolean primitives:

```
'' ? 'truthy' : 'falsy'
//=> 'falsy'
0 ? 'truthy' : 'falsy'
//=> 'falsy'
false ? 'truthy' : 'falsy'
//=> 'falsy'
```

Compare this to their corresponding instances:

¹³²We'll read all about the `new` keyword in [COnstructors and new](#).
`new String("Spence Olham") //=> "Spence Olham"`

```
new String('') ? 'truthy' : 'falsy'  
//=> 'truthy'  
new Number(0) ? 'truthy' : 'falsy'  
//=> 'truthy'  
new Boolean(false) ? 'truthy' : 'falsy'  
//=> 'truthy'
```

Our notion of “truthiness” and “falsiness” is that all instances are truthy, even string, number, and boolean instances corresponding to primitives that are falsy.

There is one sure cure for “JavaScript Impostor Syndrome.” Just as `new PrimitiveType(...)` creates an instance that is an impostor of a primitive, `PrimitiveType(...)` creates an original, canonicalized primitive from a primitive or an instance of a primitive object.

For example:

```
String(new String("Spence Olham")) === "Spence Olham"  
//=> true
```

Getting clever, we can write this:

```
const original = function (unknown) {  
  return unknown.constructor(unknown)  
}  
  
original(true) === true  
//=> true  
original(new Boolean(true)) === true  
//=> true
```

Of course, `original` will not work for your own creations unless you take great care to emulate the same behaviour. But it does work for strings, numbers, and booleans.

Finish the Cup: Constructors and Classes



Other languages call their objects “beans,” but serve extra-weak coffee in an attempt to be all things to all people

As discussed in [Encapsulating State](#), JavaScript objects are very simple, yet the combination of objects, functions, and closures can create powerful data structures. We’ve also seen how to use [Metaobjects](#) to separate behaviour from domain properties, and to share functionality amongst many different objects. And finally, we saw that one particular type of metaobject, a *prototype*, provides us with a robust model for delegation.

In this section, we will return to prototypes, and see how to use JavaScript’s `class` keyword to write one style of “object-oriented” JavaScript.

Constructors and new

Let's strip a function down to the bare essentials:

```
const Ur = function () {};
```

Or the equivalent:

```
function Ur () {};
```

This doesn't look like it has anything to do with objects and constructing things: It doesn't have an expression that yields a Plain Old JavaScript Object when the function is applied. Yet, there is a way to make an object out of it. Behold the power of the `new` keyword:

```
new Ur()  
//=> {}
```

We got an object back! What can we find out about this object?

```
new Ur() === new Ur()  
//=> false
```

Every time we call `new` with a function and get an object back, we get a unique object. We could call these "Objects created with the `new` keyword," but this would be cumbersome. So we're going to call them *instances*. Instances of what? Instances of the function that creates them. So given `const i = new Ur()`, we say that `i` is an instance of `Ur`.

We also say that `Ur` is the *constructor* of `i`, and that `Ur` is a *constructor function*. Therefore, an instance is an object created by using the `new` keyword on a constructor function, and that function is the instance's constructor.

An instance is an object created by using the `new` keyword on a constructor function, and that function is the instance's constructor.

constructors, instances, and prototypes

There's more. Here's something you may not know about functions, every function has a `.prototype` property by default:

```
Ur.prototype  
//=> {}
```

We remember [prototypes](#). What do we know about the prototype property of every function? Let's run our standard test:

```
(function () {}).prototype === (function () {}).prototype  
//=> false
```

Every function is initialized with its own unique value for the `.prototype` property. What does it do? Is it related to the prototypes we saw with Metaobjects? Let's try something:

```
Ur.prototype.language = 'JavaScript';  
  
const continent = new Ur();  
//=> {}  
continent.language  
//=> 'JavaScript'
```

That's very interesting! Instances seem to behave as if they *delegate* to their constructors `prototype`, just as if we'd created them using `Object.create(Ur.prototype)`.

We can actually test this directly:

```
Ur.prototype.isPrototypeOf(continent)  
//=> true
```

And we can inspect the prototype of our `continent` directly:

```
Object.getPrototypeOf(continent) === Ur.prototype  
//=> true
```

Let's try a few things:

```
continent.language = 'CoffeeScript';
continent
//=> {language: 'CoffeeScript'}
continent.language
//=> 'CoffeeScript'
Ur.prototype.language
'JavaScript'
```

You can set elements of an instance, and they “override” the constructor’s prototype, but they don’t actually change the constructor’s prototype. Let’s make another instance and try something else.

```
const another = new Ur();
//=> {}
another.language
//=> 'JavaScript'
```

New instances don’t acquire any changes made to other instances. Makes sense. And:

```
Ur.prototype.language = 'Sumerian'
another.language
//=> 'Sumerian'
```

Even more interesting: Changing the constructor’s prototype changes the behaviour of all of its instances. This *is* the prototype/delegation relationship we have already seen with `Object.create`.

Speaking of prototypes, here’s something else that’s very interesting:

```
continent.constructor
//=> [Function]
continent.constructor === Ur
//=> true
```

Every instance we create with `new` acquires a `constructor` element that is initialized to their constructor function. Objects we don’t create with `new` still have a `constructor` element, it’s a built-in function:

```
{}.constructor
//=> [Function: Object]
```

If that’s true, what about prototypes? Do they have constructors?

```
Ur.prototype.constructor
//=> [Function]
Ur.prototype.constructor === Ur
//=> true
```

Very interesting!

revisiting this idea of queues

Let's rewrite our Queue to use new and .prototype, using this and Object.assign:

```
const Queue = function () {
  Object.assign(this, {
    array: [],
    head: 0,
    tail: -1
  })
};

Object.assign(Queue.prototype, {
  pushTail (value) {
    return this.array[this.tail += 1] = value
  },
  pullHead () {
    let value;

    if (!this.isEmpty()) {
      value = this.array[this.head]
      this.array[this.head] = void 0;
      this.head += 1;
      return value
    }
  },
  isEmpty () {
    return this.tail < this.head
  }
});
```

You recall that when we first looked at this, we only covered the case where a function that belongs to an object is invoked. Now we see another case: When a function is invoked by the new operator, this is set to the new object being created. Thus, our code for Queue initializes the queue.

You can see why `this` is so handy in JavaScript: We wouldn't be able to define functions in the prototype that worked on the instance if JavaScript didn't give us an easy way to refer to the instance itself.

how do constructors compare to `Object.create`?

Let's summarize what we know:

When we use the `new` keyword with a function, we *construct* an object. The function is called with its context (`this`) set to the new object, and the new object delegates behaviour to whatever object is in the function's `.prototype` property.

When we use `Object.create`, we create a new object and that object delegates its behaviour to whatever object we pass to `Object.create`. If we want to do any other initialization with the object, we can do that in a separate step.

Roughly speaking, we could use `Object.create` to emulate the obvious features of the `new` keyword. Let's try it. We'll start with `worksLikeNew`, a function that takes a constructor and some optional arguments, and acts like the `new` keyword:

```
function worksLikeNew (constructor, ...args) {
  const instance = Object.create(constructor.prototype);

  instance.constructor = constructor;

  const result = constructor.apply(instance, args);

  return result === undefined ? instance : result;
}

function NamedContinent (name) {
  this.name = name;
}
NamedContinent.prototype.description = function () { return `A continent named "\` ${this.name}\`` };

const na = worksLikeNew(NamedContinent, "North America");

na.description()
//=> A continent named "North America"
```

So do we *need* the `new` keyword, given that we can emulate it? Well, one could argue that we don't *need* multiplication for positive integers:

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

Programming is a process of choosing and making abstractions, and combining constructor functions with the `new` keyword provides a single abstraction that handles several duties:

- The constructor's prototype provides a metaobject for describing the behaviour of every instance created with the constructor.
- The `.constructor` property of each instance provides an identifier for associating instances with constructors.
- The constructor's own code provides initialization for each instance.

We can do all these things with `Object.create`, but if we want to do exactly these things, and little else, `new` and a constructor function are easier, simpler, and familiar at a glance to other JavaScript programmers.

But when we want to do more, or different things, it might be better to use `Object.create` directly.

Why Classes in JavaScript?

JavaScript programmers have been using constructors for a very long time. Long enough to notice several drawbacks with them:

1. There are too many “moving parts.” Why is it necessary to define a constructor function, then manipulate its prototype property in a separate step?
2. Why is chaining prototypes so complicated?

Experienced JavaScript programmers generally responded by moving in either of two directions: Some programmers noticed that working directly with prototypes was simpler than doing everything with constructors, and gravitated towards using `Object.create` directly, using the techniques we’ve discussed in the section on [Metaobjects](#).

This approach is more flexible and powerful than using constructors, however it often seems *unfamiliar* to people who have been taught that objects should always be associated with a hierarchy of classes.

abstractioneering

Other experienced JavaScript programmers embraced classes, but paved over the awkwardness of constructors and prototypes by building their own class abstractions. For example:

```
const clazz = (...args) => {
  let superclazz, properties, constructor;

  if (args.length === 1) {
    [superclazz, properties] = [Object, args[0]];
  }
  else [superclazz, properties] = args;

  if (properties.constructor) {
    constructor = function (...args) {
      return properties.constructor.apply(this, args)
    }
  }
  else constructor = function () {};

  constructor.prototype = Object.create(superclazz.prototype);
  Object.assign(constructor.prototype, properties);
  Object.defineProperty(
    constructor.prototype,
```

```
'constructor',
{ value: constructor }
);

return constructor;
}
```

With this `clazz` function, we can write a Queue like this:

```
const Queue = clazz({
  constructor: function () {
    Object.assign(this, {
      array: [],
      head: 0,
      tail: -1
    });
  },
  pushTail: function (value) {
    return this.array[this.tail += 1] = value
  },
  pullHead: function () {
    if (!this.isEmpty()) {
      let value = this.array[this.head]
      this.array[this.head] = void 0;
      this.head += 1;
      return value
    }
  },
  isEmpty: function () {
    return this.tail < this.head
  }
});
```

And we can write a Dequeue that “subclasses” a Queue like this:

```
const Dequeue = clazz(Queue, {
  constructor: function () {
    Queue.prototype.constructor.call(this)
  },
  size: function () {
    return this.tail - this.head + 1
  },
  pullTail: function () {
    if (!this.isEmpty()) {
      let value = this.array[this.tail];
      this.array[this.tail] = void 0;
      this.tail -= 1;
      return value
    }
  },
  pushHead: function (value) {
    if (this.head === 0) {
      for (let i = this.tail; i >= this.head; --i) {
        this.array[i + this.constructor.INCREMENT] = this.array[i]
      }
      this.tail += this.constructor.INCREMENT;
      this.head += this.constructor.INCREMENT
    }
    this.array[this.head -= 1] = value
  }
});
Dequeue.INCREMENT = 4;
```

Chaining prototypes is handled for us, and we can set up the constructor function and the prototype's methods in one step. And there's a lot to be said for making "classes" out of prototypes. Because prototypes are "just objects," and methods are "just functions," we can re-use a lot of the techniques we've already developed for objects and functions with our prototypes and methods.

why prototypes being objects is a win

For example, we can use `Object.assign` to mix functionality into our classes:

```
const HasManager = {
  setManager: function (manager) {
    this.removeManager();
    this.manager = manager;
    manager.addReport(this);
    return this;
  },
  removeManager: function () {
    if (this.manager) {
      this.manager.removeReport(this);
      this.manager = undefined;
    }
    return this;
  }
};

const Manager = clazz(Person, {
  constructor: function (first, last) {
    Person.call(this, first, last);
  },
  addReport: function (report) {
    this.reports || (this.reports = new Set());
    this.reports.add(report);
    return this;
  },
  removeReport: function (report) {
    this.reports || (this.reports = new Set());
    this.reports.delete(report);
    return this;
  },
  reports: function () {
    return this._reports || (this._reports = new Set());
  }
});

const MiddleManager = clazz(Manager, {
  constructor: function (first, last) {
    Manager.call(this, first, last);
  }
});
Object.assign(MiddleManager.prototype, HasManager);
```

```
const Worker = clazz(Person, {
  constructor: function (first, last) {
    Person.call(this, first, last);
  }
});
Object.assign(Worker.prototype, HasManager);
```

Or even more declaratively:

```
const HasManager = {
  setManager: function (manager) {
    this.removeManager();
    this.manager = manager;
    manager.addReport(this);
    return this;
  }
  removeManager: function () {
    if (this.manager) {
      this.manager.removeReport(this);
      this.manager = undefined;
    }
    return this;
  }
};

const Manager = clazz(Person, {
  constructor: function (first, last) {
    Person.call(this, first, last);
  },
  addReport: function (report) {
    this.reports || (this.reports = new Set());
    this.reports.add(report);
    return this;
  },
  removeReport: function (report) {
    this.reports || (this.reports = new Set());
    this.reports.delete(report);
    return this;
  },
  reports: function () {
    return this._reports || (this._reports = new Set());
  }
}
```

```
});

const MiddleManager = clazz(Manager, Object.assign({
  constructor: function (first, last) {
    Manager.call(this, first, last);
  }
}, HasManager));

const Worker = clazz(Person, Object.assign({
  constructor: function (first, last) {
    Person.call(this, first, last);
  }
}, HasManager));
```

Likewise, decorating methods is as easy with these “classes” as it is with any other method:

```
const fluent = (methodBody) =>
  function (...args) {
    methodBody.apply(this, args);
    return this;
}

const Manager = clazz(Person, {
  constructor: function (first, last) {
    Person.call(this, first, last);
  },
  addReport: fluent(function (report) {
    this.reports || (this.reports = new Set());
    this.reports.add(report);
  }),
  removeReport: fluent(function (report) {
    this.reports || (this.reports = new Set());
    this.reports.delete(report);
  }),
  reports: function () {
    return this.reports || (this.reports = new Set());
  }
});

const MiddleManager = clazz(Manager, Object.assign({
  constructor: function (first, last) {
    Manager.call(this, first, last);
  }
},
```

```
    }
}, HasManager));

const Worker = clazz(Person, Object.assign({
  constructor: function (first, last) {
    Person.call(this, first, last);
  }
}, HasManager));
```

the problem with rolling our own classes

Building abstractions is a fundamental activity in programming. So it is not wrong to take basic tools like prototypes and build upwards from them.

However, JavaScript is a simple and elegant language, and being able to write something like `clazz` in 20-ish lines of code is wonderful. It is not a hardship to read 20 lines of code to figure out how something works. Unless you have to read twenty lines of code every time you read a new program.

If everyone, or a very large number of people, are building roughly the same abstractions, but doing them in slightly different ways, each program is nice, but the ecosystem as a whole is a mess. Every time we read a new program, we have to figure out whether they are using raw constructors, rolling their own class abstraction, or using classes from various libraries.

For this reason (and perhaps others), the `class` keyword was added to the JavaScript language.

Classes with class



Vac Pot Upper Chamber

JavaScript now has a simple way to write a “class.” Here’s a simple class written with `clazz`:

```
const Person = clazz({
  constructor: function (first, last) {
    this.rename(first, last);
  },
  fullName: function () {
    return this.firstName + " " + this.lastName;
  },
  rename: function (first, last) {
    this.firstName = first;
    this.lastName = last;
    return this;
  }
});
```

```

    }
});
```

And here it is with the `class` keyword:

```
class Person {
  constructor (first, last) {
    this.rename(first, last);
  }
  fullName () {
    return this.firstName + " " + this.lastName;
  }
  rename (first, last) {
    this.firstName = first;
    this.lastName = last;
    return this;
  }
};
```

And here's a Dequeue to show “inheritance”

```
class Dequeue extends Queue {
  constructor: function () {
    Queue.prototype.constructor.call(this)
  },
  size: function () {
    return this.tail - this.head + 1
  },
  pullTail: function () {
    if (!this.isEmpty()) {
      let value = this.array[this.tail];
      this.array[this.tail] = void 0;
      this.tail -= 1;
      return value
    }
  },
  pushHead: function (value) {
    if (this.head === 0) {
      for (let i = this.tail; i >= this.head; --i) {
        this.array[i + this.constructor.INCREMENT] = this.array[i]
      }
      this.tail += this.constructor.INCREMENT;
    }
  }
};
```

```

    this.head += this.constructor.INCREMENT
}
this.array[this.head -= 1] = value
}
});
}

Dequeue.INCREMENT = 4;

```

The interesting thing about Dequeue is that it works whether we write our Queue like this:

```

function Queue () {
Object.assign(this, {
  array: [],
  head: 0,
  tail: -1
});
}

Object.assign(Queue.prototype, {
  pushTail: function (value) {
    return this.array[this.tail += 1] = value
  },
  pullHead: function () {
    if (!this.isEmpty()) {
      let value = this.array[this.head]
      this.array[this.head] = void 0;
      this.head += 1;
      return value
    }
  },
  isEmpty: function () {
    return this.tail < this.head
  }
});

```

Or like this:

```
const Queue = clazz({
  constructor: function () {
    Object.assign(this, {
      array: [],
      head: 0,
      tail: -1
    });
  },
  pushTail: function (value) {
    return this.array[this.tail += 1] = value
  },
  pullHead: function () {
    if (!this.isEmpty()) {
      let value = this.array[this.head]
      this.array[this.head] = void 0;
      this.head += 1;
      return value
    }
  },
  isEmpty: function () {
    return this.tail < this.head
  }
});
```

Or even like this:

```
class Queue {
  constructor () {
    Object.assign(this, {
      array: [],
      head: 0,
      tail: -1
    });
  }
  pushTail (value) {
    return this.array[this.tail += 1] = value
  }
  pullHead () {
    if (!this.isEmpty()) {
      let value = this.array[this.head]
      this.array[this.head] = void 0;
      this.head += 1;
    }
  }
}
```

```

        return value
    }
}
isEmpty () {
    return this.tail < this.head
}
}
}

```

It turns out that “classes” in JavaScript are fully compatible with constructors and prototypes. That’s because behind the scenes, *they’re almost indistinguishable*. In basic use, the `class` keyword is syntactic sugar for writing constructor functions with prototypes.

There is some extra magic for handling `super` (and a few other nice-to-have features like getters and setters), but by design, and to maximize compatibility with existing code bases, the `class` keyword is a declarative way to write functions and prototypes.

classes are values

When we write:

```

class Person {
    constructor (first, last) {
        this.rename(first, last);
    }
    fullName () {
        return this.firstName + " " + this.lastName;
    }
    rename (first, last) {
        this.firstName = first;
        this.lastName = last;
        return this;
    }
};

```

It looks like we are creating a global class named `Person`. Some other languages sometimes have this idea that class names have a special significance and that they’re always global, although you can namespace them in certain ways, and the mechanism behind class names and namespaces is different than the mechanism behind variable bindings.

JavaScript does not do this. `Person` is a name bound in the environment where we evaluate the code. So yes, at the topmost level, that code creates a global binding.

But we could also write something like this, taking advantage of privacy with symbols:

```
const PrivatePerson = (() => {
  const firstName = Symbol('firstName'),
        lastName = Symbol('lastName');

  return class Person {
    constructor (first, last) {
      ++population;
      this.rename(first, last);
    }
    fullName () {
      return this(firstName] + " " + this[lastName];
    }
    rename (first, last) {
      this.firstName] = first;
      this.lastName] = last;
      return this;
    }
  };
})();
```

What does this do? It creates some symbols, then creates a class (also named person) within the same environment and uses those symbols to create private properties. It then returns the newly created class, which we bind to the name `PrivatePerson`. This hides the symbols `firstName` and `lastName` from other code.

Notice also that we *returned* the class. This implies (correctly) that the `class` keyword creates a *class expression*, and an expression is a value that can be used everywhere, just like a named function expression.

Of course, we could have bound the value returned from the IIFE to any name we like, even `Person`, but we give it a different name just to show that we have a value, just like any other value, and we bind it to a name in the environment, just like any other name in the environment. In this case, even the name `Person` is encapsulated within the IIFE.

In JavaScript, “classes” and “class expressions” are values just like any other value, and that means we can do anything with them that we can do with other values, like return them from functions, pass them to functions, and bind them to different names as we see fit.

Object Methods

An *instance method* is a function defined in the constructor's prototype. Every instance acquires this behaviour unless otherwise “overridden.” Instance methods usually have some interaction with the instance, such as references to `this` or to other methods that interact with the instance. A *constructor method* is a function belonging to the constructor itself.

There is a third kind of method, one that any object (obviously including all instances) can have. An *object method* is a function defined in the object itself. Like instance methods, object methods usually have some interaction with the object, such as references to `this` or to other methods that interact with the object.

Object methods are really easy to create with Plain Old JavaScript Objects, because they're the only kind of method you can use. Recall from [This and That](#):

```
const BetterQueue = () =>
  ({
    array: [],
    head: 0,
    tail: -1,
    pushTail: function (value) {
      return this.array[this.tail += 1] = value
    },
    pullHead: function () {
      if (this.tail >= this.head) {
        let value = this.array[this.head];
        this.array[this.head] = void 0;
        this.head += 1;
        return value
      }
    },
    isEmpty: function () {
      this.tail < this.head
    }
  });

```

`pushTail`, `pullHead`, and `isEmpty` are object methods. Also, from [encapsulation](#):

```

const stack = () => {
  const obj = {
    array: [],
    index: -1,
    push: (value) => obj.array[obj.index += 1] = value,
    pop: () => {
      const value = obj.array[obj.index];

      obj.array[obj.index] = undefined;
      if (obj.index >= 0) {
        obj.index -= 1
      }
      return value
    },
    isEmpty: () => obj.index < 0
  };
  return obj;
})();

```

Although they don't refer to the object, `push`, `pop`, and `isEmpty` semantically interact with the opaque data structure represented by the object, so they are object methods too.

object methods within instances

Instances of constructors can have object methods as well. Typically, object methods are added in the constructor. Here's a gratuitous example, a widget model that has a read-only `id`:

```

const WidgetModel = function (id, attrs) {
  Object.assign(this, attrs || {});
  this.id = function () { return id }
}

Object.assign(WidgetModel.prototype, {
  set: function (attr, value) {
    this[attr] = value;
    return this;
  },
  get: function (attr) {
    return this[attr]
  }
});

```

set and get are instance methods, but id is an object method: Each object has its own id closure, where id is bound to the id of the widget by the argument id in the constructor. The advantage of this approach is that instances can have different object methods, or object methods with their own closures as in this case. The disadvantage is that every object has its own methods, which uses up much more memory than instance methods, which are shared amongst all instances.



Object methods are defined within the object. So if you have several different “instances” of the same object, there will be an object method for each object. Object methods can be associated with any object, not just those created with the new keyword. Instance methods apply to instances, objects created with the new keyword. Instance methods are defined in a prototype and are shared by all instances.

Why Not Classes?

Classes are popular, and if classes map neatly to the way we wish to model something, we should use them.

That being said, there are some caveats to understand.

the `class` keyword is a minimal notation

By design, the `class` keyword provides the very minimum set of features needed to implement “classes.” Everything else must be done in some other way. For example, if you write constructors or prototypes directly, you can use method decorators (as we saw [earlier](#)):

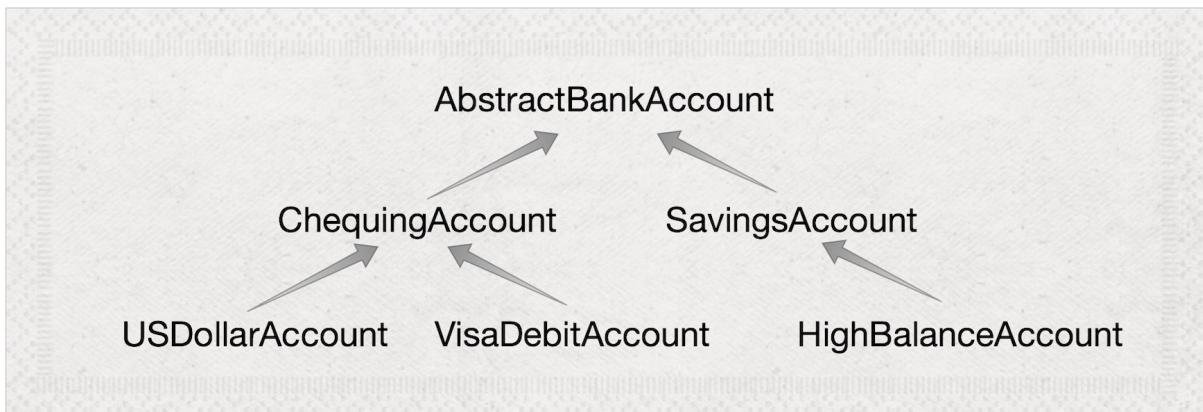
```
const fluent = (methodBody) =>
  function (...args) {
    methodBody.apply(this, args);
    return this;
}

const Manager = clazz(Person, {
  constructor: function (first, last) {
    Person.call(this, first, last);
  },
  addReport: fluent(function (report) {
    this.reports || (this.reports = new Set());
    this.reports.add(report);
  }),
  removeReport: fluent(function (report) {
    this.reports || (this.reports = new Set());
    this.reports.delete(report);
  }),
  reports: function () {
    return this._reports || (this._reports = new Set());
  }
});
```

But at this time, you cannot use method decorators when you use the `class` syntax. There are plans to introduce a new, purpose-built decorator syntax for this purpose, which highlights one of the issues with the `class` syntax: By writing what amounts to a new language on top of JavaScript, it must inevitably reinvent all of the things that are already possible in JavaScript.

classes encourage the construction of class hierarchies

The easy thing to do with classes is to create [class hierarchies¹³³](#). These are implemented by chaining prototypes. And there is a problem with chained prototypes: They couple classes to each other.



A class hierarchy

When one class extends another, its methods can access any of the properties and methods defined anywhere on the prototype chain. Given hierarchies designed as trees, a change to a class can break the behaviour of any of the classes below it or above it on the tree.

When two or more metaobjects all have access to the same base object via [open recursion¹³⁴](#), they become tightly coupled because they can interact via setting and reading all the base object's properties. It is impossible to restrict their interaction to a well-defined set of methods.

This coupling exists for all metaobject patterns that include open recursion, such as mixins, delegation, and delegation through prototypes. In particular, when chains of naive prototypes form class hierarchies, this coupling leads to the [fragile base class problem¹³⁵](#).



The **fragile base class problem** is a fundamental architectural problem of object-oriented programming systems where base classes (superclasses) are considered “fragile” because seemingly safe modifications to a base class, when inherited by the derived classes, may cause the derived classes to malfunction. The programmer cannot determine whether a base class change is safe simply by examining in isolation the methods of the base class.– [Wikipedia¹³⁶](#)

In JavaScript, prototype chains are vulnerable because changes to one prototype's behaviour may break another prototype's behaviour in the same chain.

¹³³https://en.wikipedia.org/wiki/Class_hierarchy

¹³⁴https://en.wikipedia.org/wiki/Open_recursion#Open_recursion

¹³⁵https://en.wikipedia.org/wiki/Fragile_base_class

¹³⁶https://en.wikipedia.org/wiki/Fragile_base_class

In the next section we will look at a technique for [reducing coupling between classes](#). And we will look at avoiding deep hierarchies with mixins.

Summary



Instances and Classes

- The `new` keyword turns any function into a *constructor* for creating *instances*.
- All functions have a `prototype` element.
- Instances behave as if the elements of their constructor's prototype are their elements.
- The `class` keyword acts as *syntactic sugar* for writing constructor functions.
- Classes created with the `class` keyword are actually constructor functions with optionally chained prototypes.
- Classes should be used in moderation, the syntax deliberately limits the flexibility and class hierarchies can lead to overly coupled code.

Recipes with Constructors and Classes



These recipes are being roasted to perfection.

Disclaimer

The recipes are written for practicality, and their implementation may introduce JavaScript features that haven't been discussed in the text to this point, such as methods and/or prototypes. The overall *use* of each recipe will fit within the spirit of the language discussed so far, even if the implementations may not.

Bound

Earlier, we saw a recipe for `getWith` that plays nicely with properties:

```
const getWith = (attr) => (object) => object[attr]
```

Simple and useful. But now that we've spent some time looking at objects with methods we can see that `get` (and `pluck`) has a failure mode. Specifically, it's not very useful if we ever want to get a *method*, since we'll lose the context. Consider some hypothetical class:

```
function InventoryRecord (apples, oranges, eggs) {
  this.record = {
    apples: apples,
    oranges: oranges,
    eggs: eggs
  }
}

InventoryRecord.prototype.apples = function apples () {
  return this.record.apples
}

InventoryRecord.prototype.oranges = function oranges () {
  return this.record.oranges
}

InventoryRecord.prototype.eggs = function eggs () {
  return this.record.eggs
}

const inventories = [
  new InventoryRecord( 0, 144, 36 ),
  new InventoryRecord( 240, 54, 12 ),
  new InventoryRecord( 24, 12, 42 )
];
```

Now how do we get all the egg counts?

```
mapWith(getWith('eggs'))(inventories)
//=> [ [Function: eggs],
//      [Function: eggs],
//      [Function: eggs] ]
```

And if we try applying those functions...

```
mapWith(getWith('eggs'))(inventories).map(
  unboundmethod => unboundmethod()
)
//=> TypeError: Cannot read property 'eggs' of undefined
```

It doesn't work, because these are unbound methods we're "getting" from each object. The context has been lost! Here's a new version of get that plays nicely with methods:

```
const bound = (messageName, ...args) =>
  (args === [])
    ? instance => instance[messageName].bind(instance)
    : instance => Function.prototype.bind.apply(
        instance[messageName], [instance].concat(args)
      );

mapWith(bound('eggs'))(inventories).map(
  boundmethod => boundmethod()
)
//=> [ 36, 12, 42 ]
```

bound is the recipe for getting a bound method from an object by name. It has other uses, such as callbacks. `bound('render')(aView)` is equivalent to `aView.render.bind(aView)`. There's an option to add a variable number of additional arguments, handled by:

```
instance => Function.prototype.bind.apply(
  instance[messageName], [instance].concat(args)
);
```

The exact behaviour will be covered in [Binding Functions to Contexts](#). You can use it like this to add arguments to the bound function to be evaluated:

```
InventoryRecord.prototype.add = function (item, amount) {
  this.record[item] || (this.record[item] = 0);
  this.record[item] += amount;
  return this;
}

mapWith(bound('add', 'eggs', 12))(inventories).map(
  boundmethod => boundmethod()
)
//=> [ { record:
//        { apples: 0,
//         oranges: 144,
//         eggs: 48 } },
//      { record:
//        { apples: 240,
//         oranges: 54,
//         eggs: 24 } },
//      { record:
//        { apples: 24,
//         oranges: 12,
//         eggs: 54 } } ]
```

Send

Previously, we saw that the recipe `bound` can be used to get a bound method from an instance. Unfortunately, invoking such methods is a little messy:

```
mapWith(bound('eggs'))(inventories).map(  
  boundmethod => boundmethod()  
)  
 //=> [ 36, 12, 42 ]
```

As we noted, it's ugly to write

```
boundmethod => boundmethod()
```

So instead, we write a new recipe:

```
const send = (methodName, ...args) =>  
  (instance) => instance[methodName].apply(instance, args);  
  
mapWith(send('apples'))(inventories)  
 //=> [ 0, 240, 24 ]
```

`send('apples')` works very much like `&:apples` in the Ruby programming language. You may ask, why retain `bound`? Well, sometimes we want the function but don't want to evaluate it immediately, such as when creating callbacks. `bound` does that well.

Invoke

`Send` is useful when invoking a function that's a member of an object (or of an instance's prototype). But we sometimes want to invoke a function that is designed to be executed within an object's context. This happens most often when we want to "borrow" a method from one "class" and use it on another object.

It's not an unprecedented use case. The Ruby programming language has a handy feature called `instance_exec`¹³⁷. It lets you execute an arbitrary block of code in the context of any object. Does this sound familiar? JavaScript has this exact feature, we just call it `.apply` (or `.call` as the case may be). We can execute any function in the context of any arbitrary object.

The only trouble with `.apply` is that being a method, it doesn't compose nicely with other functions like combinators. So, we create a function that allows us to use it as a combinator:

```
const invoke = (fn, ...args) =>
  instance => fn.apply(instance, args);
```

For example, let's say someone else's code gives you an array of objects that are in part, but not entirely like arrays. Something like:

```
const data = [
  { 0: 'zero',
    1: 'one',
    2: 'two',
    length: 3 },
  { 0: 'none',
    length: 1 },
  // ...
];
```

If they were arrays, and we wanted to copy them, we would use:

```
mapWith(send('slice', 0))(data)
```

Because arrays have a `.send` method. But our quasi-arrays have no such thing. So... We want to borrow the `.slice` method from arrays, but have it work on our data. `invoke([].slice, 0)` does the trick:

¹³⁷http://www.ruby-doc.org/core-1.8.7/Object.html#method-i-instance_exec

```
mapWith(invoker([] .slice, 0))(data)
//=> [
  ["zero", "one", "two"],
  ["none"],
  // ...
]
```

instance eval

`invoke` is useful when you have the function and are looking for the instance. It can be written “the other way around,” for when you have the instance and are looking for the function:

```
const instanceEval = instance =>
  (fn, ...args) => fn.apply(instance, args);
```

Fluent

Object and instance methods can be bifurcated into two classes: Those that query something, and those that update something. Most design philosophies arrange things such that update methods return the value being updated. For example:

```
class Cake {
    setFlavour (flavour) {
        return this.flavour = flavour
    },
    setLayers (layers) {
        return this.layers = layers
    },
    bake () {
        // do some baking
    }
}

const cake = new Cake();
cake.setFlavour('chocolate');
cake.setLayers(3);
cake.bake();
```

Having methods like `setFlavour` return the value being set mimics the behaviour of assignment, where `cake.flavour = 'chocolate'` is an expression that in addition to setting a property also evaluates to the value 'chocolate'.

The [fluent¹³⁸](#) style presumes that most of the time when you perform an update, you are more interested in doing other things with the receiver than the values being passed as argument(s). Therefore, the rule is to return the receiver unless the method is a query:

```
class Cake {
    setFlavour (flavour) {
        this.flavour = flavour;
        return this;
    },
    setLayers (layers) {
        this.layers = layers;
        return this;
    },
    bake () {
```

¹³⁸https://en.wikipedia.org/wiki/Fluent_interface

```
// do some baking
return this;
}
}
```

The code to work with cakes is now easier to read and less repetitive:

```
const cake = new Cake().
    setFlavour('chocolate').
    setLayers(3).
    bake();
```

For one-liners like setting a property, this is fine. But some functions are longer, and we want to signal the intent of the method at the top, not buried at the bottom. Normally this is done in the method's name, but fluent interfaces are rarely written to include methods like `setLayersAndReturnThis`.

When we write our own prototypes, the `fluent` method decorator solves this problem:

```
const fluent = (methodBody) =>
  function (...args) {
    methodBody.apply(this, args);
    return this;
}
```

Now you can write methods like this:

```
function Cake () {}

Cake.prototype.setFlavour = fluent( function (flavour) {
  this.flavour = flavour;
});
```

It's obvious at a glance that this method is "fluent."

When we use the `class` keyword, we can decorate functions in a similar manner:

```

class Cake {
  setFlavour (flavour) {
    this.flavour = flavour;
  },
  setLayers (layers) {
    this.layers = layers;
  },
  bake () {
    // do some baking
  }
}
Cake.prototype.setFlavour = fluent(Cake.prototype.setFlavour);
Cake.prototype.setLayers = fluent(Cake.prototype.setLayers);
Cake.prototype.bake = fluent(Cake.prototype.bake);

```

Or, we could write ourselves a slight variation:

```

const fluent = (methodBody) =>
  function (...args) {
    methodBody.apply(this, args);
    return this;
}

const fluentClass = (clazz, ...methodNames) {
  for (let methodName of methodNames) {
    clazz.prototype[methodName] = fluent(clazz.prototype[methodName]);
  }
  return clazz;
}

```

Now we can simply write:

```
fluentClass(Cake, 'setFlavour', 'setLayers', 'bake');
```

Colourful Mugs: Symmetry, Colour, and Charm



Pantone Coffee Mugs

We've seen that functions are *first-class entities*, meaning we can store them in data structures, pass them to other functions, and return them from functions. An amazing number of very strong programming techniques arise as a consequence of functions-as-first-class-entities.

We've also seen that we can use functions-as-first-class-entities to write decorators like [maybe](#):

```
const maybe = (fn) =>
  (...args) => {
    for (let arg of args) {
      if (arg == null) return arg;
    }
    return fn(...args);
}
```

And [combinators](#) like compose:

```
const compose = (a, b) =>
  (x) => a(b(x));

compose(x => x + 1, y => y * y)(10)
//=> 101
```

The power arising from functions-as-first-class-entities is that we have a very flexible way to make functions out of functions, using functions. We are not “multiplying our entities unnecessarily.” On the surface, decorators and combinators are made possible by the fact that we can pass functions to functions, and return functions that invoke our original functions.

But there’s something else: The fact that all functions are called in the exact same way. We write `foo(bar)` and know that we will evaluate `bar`, and pass the resulting value to the function we get by evaluating `foo`. This allows us to write decorators and combinators that work with any function.

Or does it?

Imagine, if you will, that functions came in two colours: “blue,” and “yellow.” Now imagine that when we invoke a function in a variable, we type the name of the function in the proper colour. So if we write `const square = (x) => x * x` in blue code, we also have to write `square(5)` in blue code, so that `square` is always blue.

If we write `const square = (x) => x * x` in blue code, but elsewhere we write `square(5)` in yellow code, it won’t work because `square` is a blue function and `square(5)` would be a yellow invocation.

blue and yellow functions

If functions worked like that, decorators would be very messy. We’d have to make colour-coded decorators, like a blue `maybe` and a yellow `maybe`. We’d have to carefully track which functions have which colours, much as in gendered languages like French, you need to know the gender of all inanimate objects so that you can use the correct gendered grammar when talking about them.

This sounds bad, and for programming tools, it is.¹³⁹ The general principle is: *Have fewer kinds of similar things, but allow the things you do have to combine in flexible ways*. You can't just remove things, you have to also make it very easy to combine things. Functions as first-class-entities are a good example of this, because they allow you to combine functions in flexible ways.

Coloured functions would be an example of how not to do it, because you'd be making it harder to combine functions by balkanizing them.¹⁴⁰

Functions don't have colours in JavaScript. But there are things that have this kind of asymmetry that make things just as awkward. For example, methods in JavaScript are functions. But, when you invoke them, you have to get `this` set up correctly. You have to either:

1. Invoke a method as a property of an object. e.g. `foo.bar(baz)` or `foo['bar'](baz)`.
2. Bind an object to a method before invoking it, e.g. `bar.bind(foo)`.
3. Invoke the method with `.call` or `.apply`, e.g `bar.call(foo, baz)`.

Thus, we can imagine that calling a function directly (e.g. `bar(baz)`) is blue, invoking a function and setting `this` (e.g. `bar.call(foo, baz)`) is yellow.

Or in other words, functions are blue, and methods are yellow.

the composability problem

We often write decorators in blue, a/k/a pure functional style. Here's a decorator that makes a function throw an exception if its argument is not a finite number:

```
const requiresFinite = (fn) =>
  (n) => {
    if (Number.isFinite(n)){
      return fn(n);
    }
    else throw "Bad Wolf";
  }
}
```

```
const plusOne = x => x + 1;
```

```
plus1(1)
//=> 2
```

```
plus1([])
//=> 1 WTF!?
```

¹³⁹Bad for programming languages, of course. French is a lovely human language.

¹⁴⁰See the aforelinked [The Symmetry of JavaScript Functions](#)

```
const safePlusOne = requiresFinite(plusOne);

safePlusOne(1)
//=> 2

safePlusOne([])
//=> throws "Bad Wolf"
```

But it won't work on methods. Here's a `Circle` class that has an unsafe `.scaleBy` method:

```
class Circle {
  constructor (radius) {
    this.radius = radius;
  }
  diameter () {
    return Math.PI * 2 * this.radius;
  }
  scaleBy (factor) {
    return new Circle(factor * this.radius);
  }
}

const two = new Circle(2);

two.scaleBy(3).diameter()
//=> 37.69911184307752

two.scaleBy(null).diameter()
//=> 0 WTF!?
```

Let's decorate the `scaleBy` method to check its argument:

```
Circle.prototype.scaleBy = requiresFinite(Circle.prototype.scaleBy);

two.scaleBy(null).diameter()
//=> throws "Bad Wolf"
```

Looks good, let's put it into production:

```
Circle.prototype.scaleBy = requiresFinite(Circle.prototype.scaleBy);

two.scaleBy(3).diameter()
//=> undefined is not an object (evaluating 'this.radius')
```

Whoops, we forgot that method invocation is “yellow” code, so our “blue” `requiresFinite` decorator will not work on methods. This is the problem of “yellow” and “blue” code colliding.

composing functions with “green” code

Fortunately, we can write higher-order functions like decorators and combinators in a style that works for both “pure” functions and for methods. We have to use the `function` keyword so that `this` is bound, and then invoke our decorated function using `.call` so that we can pass `this` along.

Here’s `requiresFinite` written in this style, which we will call “green.” It works for decorating both methods *and* functions:

```
const requiresFinite = (fn) =>
  function (n) {
    if (Number.isFinite(n)){
      return fn.call(this, n);
    }
    else throw "Bad Wolf";
  }
```

```
Circle.prototype.scaleBy = requiresFinite(Circle.prototype.scaleBy);
```

```
two.scaleBy(3).diameter()
//=> 37.69911184307752
```

```
two.scaleBy("three").diameter()
//=> throws "Bad Wolf"
```

```
const safePlusOne = requiresFinite(x => x + 1);
```

```
safePlusOne(1)
//=> 2
```

```
safePlusOne([])
//=> throws "Bad Wolf"
```

We can write all of our decorators and combinators in “green” style. For example, instead of writing `maybe` in functional (“blue”) style like this:

```
const maybe = (fn) =>
  (...args) => {
    for (let arg of args) {
      if (arg == null) return arg;
    }
    return fn(...args);
}
```

We can write it in both functional and method style (“green”) style like this:

```
const maybe = (method) =>
  function (...args) {
    for (let arg of args) {
      if (arg == null) return arg;
    }
    return method.apply(this, args);
}
```

And instead of writing our simple compose in functional (“blue”) style like this:

```
const compose = (a, b) =>
  (x) => a(b(x));
```

We can write it in both functional and method style (“green”) style like this:

```
const compose = (a, b) =>
  function (x) {
    return a.call(this, b.call(this, x));
}
```

What makes JavaScript tolerable is that green handling works for both functional (“blue”) and method invocation (“yellow”) code. But when writing large code bases, we have to remain aware that some functions are blue and some are yellow, because if we write a mostly blue program, we could be lured into complacency with with blue decorators and combinators for years. But everything would break if a “yellow” method was introduced that didn’t play nicely with our blue combinators.

The safe thing to do is to write all our higher-order functions in “green” style, so that they work for functions or methods. And that’s why we might talk about the simpler, “blue” form when introducing an idea, but we write out the more complete, “green” form when implementing it as a recipe.

red functions vs. object factories

JavaScript classes (and the equivalent prototype-based patterns) rely on creating objects with the `new` keyword. As we saw in the example above:

```

class Circle {
  constructor (radius) {
    this.radius = radius;
  }
  diameter () {
    return Math.PI * 2 * this.radius;
  }
  scaleBy (factor) {
    return new Circle(factor * this.radius);
  }
}

const round = new Circle(1);

round.diameter()
//=> 6.2831853

```

That `new` keyword introduces yet *another* colour of function, constructors are “red” functions. We can’t make circles using “blue” function calls:

```

const round2 = Circle(2);
//=> Cannot call a class as a function

[1, 2, 3, 4, 5].map(Circle)
//=> Cannot call a class as a function

```

And we certainly can’t use a decorator on them:

```

const CircleRequiringFiniteRadius = requiresFinite(Circle);

const round3 = new CircleRequiringFiniteRadius(3);
//=> Cannot call a class as a function

```

Some experienced developers dislike `new` because of this problem: It introduces one more kind of function that doesn’t compose neatly with other functions using our existing decorators and combinators.

We could eliminate “red” functions by using prototypes and `Object.create` instead of using the `class` and `new` keywords. A “factory function” is a function that makes new objects. So instead of writing a `Circle` class, we would write a `CirclePrototype` and a `CircleFactory` function:

```

const CirclePrototype = {
  diameter () {
    return Math.PI * 2 * this.radius;
  },
  scaleBy (factor) {
    return CircleFactory(factor * this.radius);
  }
};

const CircleFactory = (radius) =>
  Object.create(CirclePrototype, {
    radius: { value: radius, enumerable: true }
  })
}

CircleFactory(2).scaleBy(3).diameter()
//=> 37.69911184307752

```

Now we have a “blue” CircleFactory function, and we have the benefits of objects and methods, along with the benefits of decorating and composing factories like any other function. For example:

```

const requiresFinite = (fn) =>
  function (n) {
    if (Number.isFinite(n)){
      return fn.call(this, n);
    }
    else throw "Bad Wolf";
  }

const FiniteCircleFactory = requiresFinite(CircleFactory);

FiniteCircleFactory(2).scaleBy(3).diameter()
//=> 37.69911184307752

FiniteCircleFactory(null).scaleBy(3).diameter()
//=> throws "Bad Wolf"

```

All that being said, programming with factory functions instead of with classes and new is not a cure-all. Besides losing some of the convenience and familiarity for other developers, we’d also have to use extreme discipline for fear that accidentally introducing some “red” classes would break our carefully crafted “blue in green” application.

In the end, there’s no avoiding the need to know which functions are functions, and which are actually classes. Tooling can help: Some linting applications can enforce a naming convention where classes start with an upper-case letter and functions start with a lower-case letter.

charmed functions

Consider:

```
const likesToDoink = (whom) => {
  switch (whom) {
    case 'Bob':
      return 'Ristretto';
    case 'Carol':
      return 'Cappuccino';
    case 'Ted':
      return 'Allongé';
    case 'Alice':
      return 'Cappuccino';
  }
}

likesToDoink('Alice')
//=> 'Cappuccino'

likesToDoink('Peter')
//=> undefined;
```

That's a pretty straightforward function that implements a mapping from Bob, Carol, Ted, and Alice to the drinks 'Ristretto', 'Cappuccino', and 'Allongé'. The mapping is encoded implicitly in the code's switch statement.

We can use it in combination with other functions. For example, we can find out if the first letter of what someone likes is "c:"

```
const startsWithC = (something) => !!something.match(/^c/i)

startsWithC(likesToDoink('Alice'))
//=> true

const likesSomethingStartingWithC =
  compose(startsWithC, likesToDoink);

likesSomethingStartingWithC('Ted')
//=> false
```

So far, that's good, clean blue function work. But there's yet another kind of "function call." If you are a mathematician, this is a mapping too:

```
const personToDrink = {
  Bob: 'Ristretto',
  Carol: 'Cappuccino',
  Ted: 'Allongé',
  Alice: 'Cappuccino'
}
```

```
personToDrink['Alice']
//=> 'Cappuccino'
```

```
personToDrink['Ted']
//=> 'Allongé'
```

personToDrink also maps the names ‘Bob’, ‘Carol’, ‘Ted’, and ‘Alice’ to the drinks ‘Ristretto’, ‘Cappuccino’, and ‘Allongé’, just like likesToDrink. But even though it does the same thing as a function, we can’t use it as a function:

```
const personMapsToSomethingStartingWithC =
compose(startsWithC, personToDrink);

personMapsToSomethingStartingWithC('Ted')
//=> undefined is not a function (evaluating 'b.call(this, x)')
```

As you can see, [and] are a little like (and), because we can pass Alice to personToDrink and get back Cappuccino. But they are just different enough, that we can’t write personToDrink(...). Objects (as well as ECMAScript 2015 maps and sets) are “charmed functions.”

And you need a different piece of code to go with them. We’d need to write things like this:

```
const composeblueWithCharm =
(bluefunction, charmedfunction) =>
(arg) =>
  bluefunction(charmedfunction[arg]);

const composeCharmWithblue =
(charmedfunction, bluefunction) =>
(arg) =>
  charmedfunction[bluefunction(arg)]

// ...
```

That would get really old, really fast.

adapting to handle red and charmed functions

We can work our way around some of these cross-colour and charm issues by writing *adaptors*, wrappers that turn red and charmed functions into blue functions. As we saw above, a “factory function” is a function that is called in the normal way, and returns a freshly created object.

If we wanted to create a `CircleFactory`, we could use `Object.create` as we saw above. We could also wrap `new Circle(...)` in a function:

```
class Circle {
  constructor (radius) {
    this.radius = radius;
  }
  diameter () {
    return Math.PI * 2 * this.radius;
  }
  scaleBy (factor) {
    return new Circle(factor * this.radius);
  }
}

const CircleFactory = (radius) =>
  new Circle(radius);

CircleFactory(2).scaleBy(3).diameter()
//=> 37.69911184307752
```

With some argument jiggery-pokery, we could abstract `Circle` from `CircleFactory` and make a factory for making factories, a `FactoryFactory`:

We would write a `CircleFactory` function:

```
const FactoryFactory = (clazz) =>
  (...args) =>
    new clazz(...args);

const CircleFactory = FactoryFactory(Circle);

circleFactory(5).diameter()
//=> 31.4159265
```

`FactoryFactory` turns any “red” class into a “blue” function. So we can use it anywhere we like:

```
[1, 2, 3, 4, 5].map(FactoryFactory(Circle))
//=>
[{"radius":1}, {"radius":2}, {"radius":3}, {"radius":4}, {"radius":5}]
```

Sadly, we still have to remember that `Circle` is a class and be sure to wrap it in `FactoryFactory` when we need to use it as a function, but that does work.

We can do a similar thing with our “charmed” maps (and arrays, for that matter). Here’s `Dictionary`, a function that turns objects and arrays (our “charmed” functions) into ordinary (“blue”) functions:

```
const Dictionary = (data) => (key) => data[key];

const personToDrink = {
  Bob: 'Ristretto',
  Carol: 'Cappuccino',
  Ted: 'Allongé',
  Alice: 'Cappuccino'
}

['Bob', 'Ted', 'Carol', 'Alice'].map(Dictionary(personToDrink))
//=> ["Ristretto", "Allongé", "Cappuccino", "Cappuccino"]
```

`Dictionary` makes it easier for us to use all of the same tools for combining and manipulating functions on arrays and objects that we do with functions.

dictionaries as proxies

As David Nolen¹⁴¹ has pointed out, languages like Clojure have maps that can be called as functions automatically. This is superior to wrapping a map in a plain function, because the underlying map is still available to be iterated over and otherwise treated as a map. Once we wrap a map in a function, it becomes opaque, useless for anything except calling as a function.

If we wish, we can create a dictionary function that is a partial proxy for the underlying collection object. For example, here is an `IterableDictionary` that turns a collection into a function that is also iterable if its underlying data object is iterable:

¹⁴¹<http://swannodette.github.io>

```

const IterableDictionary = (data) => {
  const proxy = (key) => data[key];
  proxy[Symbol.iterator] = function* (...args) {
    yield* data[Symbol.iterator](...args);
  }
  return proxy;
}

const people = IterableDictionary(['Bob', 'Ted', 'Carol', 'Alice']);
const drinks = IterableDictionary(personToDrink);

for (let name of people) {
  console.log(`#${name} prefers to drink ${drinks(name)}`)
}
//=>
  Bob prefers to drink Ristretto
  Ted prefers to drink Allongé
  Carol prefers to drink Cappuccino
  Alice prefers to drink Cappuccino

```

This technique has limitations. For example, objects in JavaScript are not iterable by default. So we can't write:

```

for (let [name, drink] of personToDrink) {
  console.log(`#${name} prefers to drink ${drink}`)
}
//=> undefined is not a function (evaluating 'personToDrink[Symbol.iterator]()')

```

We could write:

```

for (let [name, drink] of Object.entries(personToDrink)) {
  console.log(`#${name} prefers to drink ${drink}`)
}
//=>
  Bob prefers to drink Ristretto
  Carol prefers to drink Cappuccino
  Ted prefers to drink Allongé
  Alice prefers to drink Cappuccino

```

It would be an enormous hack to make `Object.entries(IterableDictionary(personToDrink))` work. While we're at it, how would we make `.length` work? Functions implement `.length` as the

number of arguments they accept. Arrays implement it as the number of entries they hold. If we wrap an array in a dictionary, what is its `.length`?

Proxying collections, meaning “creating an object that behaves like the collection,” works for specific and limited contexts, but it is enormously fragile to attempt to make a universal proxy that also acts as a function.

summary

JavaScript’s elegance comes from having a simple thing, functions, that can be combined in many flexible ways. Exceptions to the ways functions combine, like the `new` keyword, handling `this`, and `[. . .]`, make combining awkward, but we can work around that by writing adaptors to convert these exceptions to regular function calls.

Con Panna: Composing Class Behaviour



Espresso Con Panna mixes sweet whipping cream into the strong coffee

Because prototypes are just objects, and because “classes” actually use prototypes under the hood, we can use all of the techniques we’ve learned about working with objects, when working with prototypes.

Extending Classes with Mixins

We've seen that a "class" is simply a constructor function that is associated with a prototype, and that the `class` keyword is a declarative way to write our own constructor functions and prototypes. When we use the `new` keyword, we are invoking a mechanism that creates a new object that delegates to a prototype, just like `Object.create`, and then the constructor function takes over and performs any initialization we desire.

Because "classes" use the exact same model of delegating behaviour to prototypes, all the things we learned about prototypes apply to classes. We saw that we can create "subclasses" by chaining prototypes.

We can also share behaviour between classes in a more flexible way by mixing functionality into classes. This is the exact same thing as mixing functionality into prototypes, of course.

Recall Person:

```
class Person {
  constructor (first, last) {
    this.rename(first, last);
  }
  fullName () {
    return this.firstName + " " + this.lastName;
  }
  rename (first, last) {
    this.firstName = first;
    this.lastName = last;
    return this;
  }
}

const misterRogers = new Person('Fred', 'Rogers');
misterRogers.fullName()
//=> Fred Rogers
```

We might be building some enterprisey thing and need Manager and Worker:

```

class Manager extends Person {
    constructor (first, last) {
        super(first, last)
    }
    addReport (report) {
        this.reports().add(report);
        return this;
    }
    removeReport (report) {
        this.reports().delete(report);
        return this;
    }
    reports () {
        return this._reports || (this._reports = new Set());
    }
}

class Worker extends Person {
    constructor (first, last) {
        super(first, last);
    }
    setManager (manager) {
        this.removeManager();
        this.manager = manager;
        manager.addReport(this);
        return this;
    }
    removeManager () {
        if (this.manager) {
            this.manager.removeReport(this);
            this.manager = undefined;
        }
        return this;
    }
}

```

This works for our company, so well that we grow and develop the dreaded “Middle Manager,” who both manages people and has a manager of their own. We could subclass Manager with MiddleManager, but how do Worker and MiddleManager share the functionality for having a manager?

With a mixin, of course:

```
const HasManager = {
  setManager: function (manager) {
    this.removeManager();
    this.manager = manager;
    manager.addReport(this);
    return this;
  },
  removeManager: function () {
    if (this.manager) {
      this.manager.removeReport(this);
      this.manager = undefined;
    }
    return this;
  }
};

class Manager extends Person {
  constructor (first, last) {
    super(first, last)
  }
  addReport (report) {
    this.reports().add(report);
    return this;
  }
  removeReport (report) {
    this.reports().delete(report);
    return this;
  }
  reports () {
    return this._reports || (this._reports = new Set());
  }
}

class MiddleManager extends Manager {
  constructor (first, last) {
    super(first, last);
  }
}
Object.assign(MiddleManager.prototype, HasManager);

class Worker extends Person {
  constructor (first, last) {
```

```
super(first, last);
}
}
Object.assign(Worker.prototype, HasManager);
```

We can mix functionality into the prototypes of “classes” just as easily as we can mix functionality directly into objects, because prototypes *are* objects, and JavaScript builds its “classes” out of prototypes.

Were classes “something else,” like they are in other languages, we would gain many advantages that we do not enjoy in JavaScript, but we would also give up the flexibility of being able to use the same tools and techniques on prototypes that we do on objects.

Functional Mixins

In [Extending Classes with Mixins](#), we saw that you can emulate “mixins” using `Object.assign` on classes. We’ll revisit this subject now and spend more time looking at mixing functionality into classes.

First, a quick recap: In JavaScript, a “class” is implemented as a constructor function and its prototype, whether you write it directly, or use the `class` keyword. Instances of the class are created by calling the constructor with `new`. They “inherit” shared behaviour from the constructor’s prototype property.¹⁴²

the object mixin pattern

One way to share behaviour scattered across multiple classes, or to untangle behaviour by factoring it out of an overweight prototype, is to extend a prototype with a *mixin*.

Here’s a class of todo items:

```
class Todo {
  constructor (name) {
    this.name = name || 'Untitled';
    this.done = false;
  }
  do () {
    this.done = true;
    return this;
  }
  undo () {
    this.done = false;
    return this;
  }
}
```

And a “mixin” that is responsible for colour-coding:

¹⁴²A much better way to put it is that objects with a prototype *delegate* behaviour to their prototype (and that may in turn delegate behaviour to its prototype if it has one, and so on).

```
const Coloured = {
  setColourRGB ({r, g, b}) {
    this.colourCode = {r, g, b};
    return this;
  },
  getColourRGB () {
    return this.colourCode;
  }
};
```

Mixing colour coding into our Todo prototype is straightforward:

```
Object.assign(Todo.prototype, Coloured);

new Todo('test')
  .setColourRGB({r: 1, g: 2, b: 3})
  //=> {"name": "test", "done": false, "colourCode": {"r": 1, "g": 2, "b": 3}}
```

So far, very easy and very simple. This is a *pattern*, a recipe for solving a certain problem using a particular organization of code.

functional mixins

The object mixin we have above works properly, but our little recipe had two distinct steps: Define the mixin and then extend the class prototype. Angus Croll pointed out that it's more elegant to define a mixin as a function rather than an object. He calls this a **functional mixin**¹⁴³. Here's Coloured again, recast in functional form:

```
const Coloured = (target) =>
  Object.assign(target, {
    setColourRGB ({r, g, b}) {
      this.colourCode = {r, g, b};
      return this;
    },
    getColourRGB () {
      return this.colourCode;
    }
  });
Coloured(Todo.prototype);
```

We can make ourselves a *factory function* that also names the pattern:

¹⁴³<https://javascriptweblog.wordpress.com/2011/05/31/a-fresh-look-at-javascript-mixins/>

```
const FunctionalMixin = (behaviour) =>
  target => Object.assign(target, behaviour);
```

This allows us to define functional mixins neatly:

```
const Coloured = FunctionalMixin({
  setColourRGB ({r, g, b}) {
    this.colourCode = {r, g, b};
    return this;
  },
  getColourRGB () {
    return this.colourCode;
  }
});
```

enumerability

If we look at the way `class` defines prototypes, we find that the methods defined are not enumerable by default. This works around a common error where programmers iterate over the keys of an instance and fail to test for `.hasOwnProperty`.

Our object mixin pattern does not work this way, the methods defined in a mixin *are* enumerable by default, and if we carefully defined them to be non-enumerable, `Object.assign` wouldn't mix them into the target prototype, because `Object.assign` only assigns enumerable properties.

And thus:

```
Coloured(Todo.prototype)
```

```
const urgent = new Todo("finish blog post");
urgent.setColourRGB({r: 256, g: 0, b: 0});

for (let property in urgent) console.log(property);
// =>
name
done
colourCode
setColourRGB
getColourRGB
```

As we can see, the `setColourRGB` and `getColourRGB` methods are enumerated, although the `do` and `undo` methods are not. This can be a problem with naïve code: we can't always rewrite all the *other* code to carefully use `.hasOwnProperty`.

One benefit of functional mixins is that we can solve this problem and transparently make mixins behave like `class`:

```
const FunctionalMixin = (behaviour) =>
  function (target) {
    for (let property of Reflect.ownKeys(behaviour))
      if (!target[property])
        Object.defineProperty(target, property, {
          value: behaviour[property],
          writable: true
        })
    return target;
  }
}
```

Writing this out as a pattern would be tedious and error-prone. Encapsulating the behaviour into a function is a small win.

mixin responsibilities

Like classes, mixins are metaobjects: They define behaviour for instances. In addition to defining behaviour in the form of methods, classes are also responsible for initializing instances. But sometimes, classes and metaobjects handle additional responsibilities.

For example, sometimes a particular concept is associated with some well-known constants. When using a class, can be handy to namespace such values in the class itself:

```
class Todo {
  constructor (name) {
    this.name = name || Todo.DEFAULT_NAME;
    this.done = false;
  }
  do () {
    this.done = true;
    return this;
  }
  undo () {
    this.done = false;
    return this;
  }
}

Todo.DEFAULT_NAME = 'Untitled';

// If we are sticklers for read-only constants, we could write:
// Object.defineProperty(Todo, 'DEFAULT_NAME', {value: 'Untitled'});
```

We can't really do the same thing with simple mixins, because all of the properties in a simple mixin end up being mixed into the prototype of instances we create by default. For example, let's say we want to define `Coloured.RED`, `Coloured.GREEN`, and `Coloured.BLUE`. But we don't want any specific coloured instance to define RED, GREEN, or BLUE.

Again, we can solve this problem by building a functional mixin. Our `FunctionalMixin` factory function will accept an optional dictionary of read-only mixin properties:

```
function FunctionalMixin (behaviour, sharedBehaviour = {}) {
  const instanceKeys = Reflect.ownKeys(behaviour);
  const sharedKeys = Reflect.ownKeys(sharedBehaviour);

  function mixin (target) {
    for (let property of instanceKeys)
      if (!target[property])
        Object.defineProperty(target, property, {
          value: behaviour[property],
          writable: true
        });
    return target;
  }
  for (let property of sharedKeys)
    Object.defineProperty(mixin, property, {
      value: sharedBehaviour[property],
      enumerable: sharedBehaviour.propertyIsEnumerable(property)
    });
  return mixin;
}
```

And now we can write:

```
const Coloured = FunctionalMixin({
  setColourRGB ({r, g, b}) {
    this.colourCode = {r, g, b};
    return this;
  },
  getColourRGB () {
    return this.colourCode;
  }
}, {
  RED: { r: 255, g: 0, b: 0 },
  GREEN: { r: 0, g: 255, b: 0 },
  BLUE: { r: 0, g: 0, b: 255 },
});
```

```

});;

Coloured(Todo.prototype)

const urgent = new Todo("finish blog post");
urgent.setColourRGB(Coloured.RED);

urgent.getColourRGB()
//=> {"r":255, "g":0, "b":0}

```

mixin methods

Such properties need not be values. Sometimes, classes have methods. And likewise, sometimes it makes sense for a mixin to have its own methods. One example concerns `instanceof`.

In earlier versions of ECMAScript, `instanceof` is an operator that checks to see whether the prototype of an instance matches the prototype of a constructor function. It works just fine with “classes,” but it does not work “out of the box” with mixins:

```

urgent instanceof Todo
//=> true

urgent instanceof Coloured
//=> false

```

To handle this and some other issues where programmers are creating their own notion of dynamic types, or managing prototypes directly with `Object.create` and `Object.setPrototypeOf`, ECMAScript 2015 provides a way to override the built-in `instanceof` behaviour: An object can define a method associated with a well-known symbol, `Symbol.hasInstance`.

We can test this quickly:¹⁴⁴

```

Coloured[Symbol.hasInstance] = (instance) => true
urgent instanceof Coloured
//=> true
{} instanceof Coloured
//=> true

```

Of course, that is not semantically correct. But using this technique, we can write:

¹⁴⁴This may not work with various transpilers and other incomplete ECMAScript 2015 implementations. Check the documentation. For example, you must enable the “high compliancy” mode in `BabelJS`. This is off by default to provide the highest possible performance for code bases that do not need to use features like this.

```

function FunctionalMixin (behaviour, sharedBehaviour = {}) {
  const instanceKeys = Reflect.ownKeys(behaviour);
  const sharedKeys = Reflect.ownKeys(sharedBehaviour);
  const typeTag = Symbol("isA");

  function mixin (target) {
    for (let property of instanceKeys)
      if (!target[property])
        Object.defineProperty(target, property, {
          value: behaviour[property],
          writable: true
        })
    target[typeTag] = true;
    return target;
  }
  for (let property of sharedKeys)
    Object.defineProperty(mixin, property, {
      value: sharedBehaviour[property],
      enumerable: sharedBehaviour.propertyIsEnumerable(property)
    });
  Object.defineProperty(mixin, Symbol.hasInstance, { value: (instance) => !!instance[typeTag] });
  return mixin;
}

urgent instanceof Coloured
//=> true
{} instanceof Coloured
//=> false

```

Do you need to implement `instanceof`? Quite possibly not. “Rolling your own polymorphism” is usually a last resort. But it can be handy for writing test cases, and a few daring framework developers might be working on multiple dispatch and pattern-matching for functions.

summary

The charm of the object mixin pattern is its simplicity: It really does not need an abstraction wrapped around an object literal and `Object.assign`.

However, behaviour defined with the mixin pattern is *slightly* different than behaviour defined with the `class` keyword. Two examples of these differences are enumerability and mixin properties (such as constants and mixin methods like `[Symbol.hasInstance]`).

Functional mixins provide an opportunity to implement such functionality, at the cost of some complexity in the `FunctionalMixin` function that creates functional mixins.

As a general rule, it's best to have things behave as similarly as possible in the domain code, and this sometimes does involve some extra complexity in the infrastructure code. But that is more of a guideline than a hard-and-fast rule, and for this reason there is a place for both the object mixin pattern *and* functional mixins in JavaScript.

Emulating Multiple Inheritance

If you want to mix behaviour into a class, mixins do the job very nicely. But sometimes, people want more. They want **multiple inheritance**. Meaning, what they really want is to create a new class that inherits from both Todo *and* from Coloured.

If JavaScript had multiple inheritance, we could accomplish this by extending a class with more than one superclass:

```
class Todo {  
    constructor (name) {  
        this.name = name || 'Untitled';  
        this.done = false;  
    }  
  
    do () {  
        this.done = true;  
        return this;  
    }  
  
    undo () {  
        this.done = false;  
        return this;  
    }  
  
    toHTML () {  
        return this.name; // highly insecure  
    }  
}  
  
class Coloured {  
    setColourRGB ({r, g, b}) {  
        this.colourCode = {r, g, b};  
        return this;  
    }  
  
    getColourRGB () {  
        return this.colourCode;  
    }  
}  
  
let yellow = {r: 'FF', g: 'FF', b: '00'},  
      red    = {r: 'FF', g: '00', b: '00'},
```

```

green  = {r: '00', g: 'FF', b: '00'},
grey   = {r: '80', g: '80', b: '80'};

let oneDayInMilliseconds = 1000 * 60 * 60 * 24;

class TimeSensitiveTodo extends Todo, Coloured {
  constructor (name, deadline) {
    super(name);
    this.deadline = deadline;
  }

  getColourRGB () {
    let slack = this.deadline - Date.now();

    if (this.done) {
      return grey;
    }
    else if (slack <= 0) {
      return red;
    }
    else if (slack <= oneDayInMilliseconds){
      return yellow;
    }
    else return green;
  }

  toHTML () {
    let rgb = this.getColourRGB();

    return `<span style="color: ${rgb.r}${rgb.g}${rgb.b};">${super.toHTML()}</span>`;
  }
}

```

This hypothetical `TimeSensitiveTodo` extends both `Todo` and `Coloured`, and it overrides `toHTML` from `Todo` as well as overriding `getColourRGB` from `Coloured`.

subclass factories

However, JavaScript does not have “true” multiple inheritance, and therefore this code does not work. But we can simulate multiple inheritance for cases like this. The way it works is to step back and ask ourselves, “What would we do if we didn’t have mixins or multiple inheritance?”

The answer is, we'd force a square multiple inheritance peg into a round single inheritance hole, like this:

```
class Todo {
    // ...
}

class ColouredTodo extends Todo {
    // ...
}

class TimeSensitiveTodo extends ColouredTodo {
    // ...
}
```

By making ColouredTodo extend Todo, TimeSensitiveTodo can extend ColouredTodo and override methods from both. This is exactly what most programmers do, and we know that it is an anti-pattern, as it leads to duplicated class behaviour and deep class hierarchies.

But.

What if, instead of manually creating this hierarchy, we use our simple mixins to do the work for us? We can take advantage of the fact that [classes are expressions¹⁴⁵](#), like this:

```
let Coloured = FunctionalMixin({
    setColourRGB ({r, g, b}) {
        this.colourCode = {r, g, b};
        return this;
    },
    getColourRGB () {
        return this.colourCode;
    }
});

let ColouredTodo = Coloured(class extends Todo {});
```

Thus, we have a ColouredTodo that we can extend and override, but we also have our Coloured behaviour in a mixin we can use anywhere we like without duplicating its functionality in our code. The full solution looks like this:

¹⁴⁵<http://raganwald.com/2015/06/04/classes-are-expressions.html>

```

class Todo {
  constructor (name) {
    this.name = name || 'Untitled';
    this.done = false;
  }

  do () {
    this.done = true;
    return this;
  }

  undo () {
    this.done = false;
    return this;
  }

  toHTML () {
    return this.name; // highly insecure
  }
}

let Coloured = FunctionalMixin({
  setColourRGB ({r, g, b}) {
    this.colourCode = {r, g, b};
    return this;
  },

  getColourRGB () {
    return this.colourCode;
  }
});

let ColouredTodo = Coloured(class extends Todo {});

let yellow = {r: 'FF', g: 'FF', b: '00'},
  red = {r: 'FF', g: '00', b: '00'},
  green = {r: '00', g: 'FF', b: '00'},
  grey = {r: '80', g: '80', b: '80'};

let oneDayInMilliseconds = 1000 * 60 * 60 * 24;

class TimeSensitiveTodo extends ColouredTodo {

```

```

constructor (name, deadline) {
  super(name);
  this.deadline = deadline;
}

getColourRGB () {
  let slack = this.deadline - Date.now();

  if (this.done) {
    return grey;
  }
  else if (slack <= 0) {
    return red;
  }
  else if (slack <= oneDayInMilliseconds){
    return yellow;
  }
  else return green;
}

toHTML () {
  let rgb = this.getColourRGB();

  return `<span style="color: ${rgb.r}${rgb.g}${rgb.b};">${super.toHTML()}</span>`;
}
}

let task = new TimeSensitiveTodo('Finish JavaScript Allongé', Date.now() + oneDayInMilliseconds);

task.toHTML()
//=> <span style="color: #FFFF00;">Finish JavaScript Allongé</span>

```

The key snippet is `let ColouredTodo = Coloured(class extends Todo {});`, it turns behaviour into a subclass that can be extended and overridden.

subclass factories

We can turn this pattern into a function:

```
const SubclassFactory = (behaviour) => { let mixBehaviourInto = FunctionalMixin(behaviour);
```

```
return (superclazz) => mixBehaviourInto(class extends superclazz {}); } ~~~~~
```

Using SubclassFactory, we wrap the class we want to extend, instead of the class we are declaring. Like this:

```
const SubclassFactory = (behaviour) => {
  let mixBehaviourInto = FunctionalMixin(behaviour);

  return (superclazz) => mixBehaviourInto(class extends superclazz {}); }

const ColouredAsWellAs = SubclassFactory({
  setColourRGB ({r, g, b}) {
    this.colourCode = {r, g, b};
    return this;
  },

  getColourRGB () {
    return this.colourCode;
  }
});

class TimeSensitiveTodo extends ColouredAsWellAs(Todo) {
  constructor (name, deadline) {
    super(name);
    this.deadline = deadline;
  }

  getColourRGB () {
    let slack = this.deadline - Date.now();

    if (this.done) {
      return grey;
    }
    else if (slack <= 0) {
      return red;
    }
    else if (slack <= oneDayInMilliseconds){
      return yellow;
    }
    else return green;
  }
}
```

```
toHTML () {  
  let rgb = this.getColourRGB();  
  
  return `<span style="color: ${rgb.r}${rgb.g}${rgb.b};">${super.toHTML()}</s>  
pan>`;  
}  
}
```

The syntax of `class TimeSensitiveTodo extends ColouredAsWellAs(Todo)` says exactly what we mean: We are extending our `Coloured` behaviour as well as extending `ToDo`.¹⁴⁶

¹⁴⁶Justin Fagnani named this pattern “subclass factory” in his essay [“Real” Mixins with JavaScript Classes](#). It’s well worth a read, and his implementation touches on other matters such as optimizing performance on modern JavaScript engines.

Preventing Property Conflicts

When mixing behaviour onto classes, (and equally, when chaining prototypes, or extending classes in a hierarchy), we are engaging in [open recursion¹⁴⁷](#). The methods in each mixin (or prototype in a chain) all have the same context, and therefore refer to the same properties.

When chaining prototypes or extending classes, this does not typically result in two functions accidentally using the same property for two different purposes. For example, if we write:

```
class Person {
  constructor (first, last) {
    this.rename(first, last);
  }
  fullName () {
    return this.firstName + " " + this.lastName;
  }
  rename (first, last) {
    this.firstName = first;
    this.lastName = last;
    return this;
  }
};

class Bibliophile extends Person {
  addToCollection (name) {
    this.collection().push(name);
    return this;
  },
  collection () {
    return this._books || (this._books = []);
  }
}
```

And later we wanted to write:

```
class Author extends Bibliophile {
  // ...
}
```

It is very unlikely that we would attempt to use the same `_books` property to refer to both the books an author writes and the books a bibliophile collects. For some odd reason, our ontology has it that

¹⁴⁷https://en.wikipedia.org/wiki/Open_recursion#Open_recursion

all authors are also bibliophiles, so it's natural that we would inspect the `Bibliophile` superclass when designing `Author`, and all of our tests for `Author` would be performed on objects that are instances of `Bibliophile`, by definition.

However, this is not the case for mixins. If we wrote:

```
const IsBibliophile = {
  addToCollection (name) {
    this.collection().push(name);
    return this;
  },
  collection () {
    return this._books || (this._books = []);
  }
};
```

And a colleague wrote:

```
const IsAuthor = {
  addBook (name) {
    this.books().push(name);
    return this;
  },
  books () {
    return this._books || (this._books = []);
  }
};
```

This code could easily work for months or years. `IsAuthor` could be tested independently of `Bibliophile`, and both would appear to behave correctly. Until the fateful day someone wrote something like:

```
class BookLovingAuthor extends Person {
}

Object.assign(BookLovingAuthor.prototype, IsBibliophile, IsAuthor);

new BookLovingAuthor('Isaac', 'Asimov')
  .addBook('I Robot')
  .addToCollection('The Mysterious Affair at Styles')
  .collection()
  //=> ["I Robot", "The Mysterious Affair at Styles"]
```

And bam! We have a property conflict: The books Isaac Asimov has written and collects have become intermingled, because the two mixins refer to the same property.

decoupling mixins with symbols

The simplest way to avoid these property conflicts is to use symbols for property names:

```
class Person {
  constructor (first, last) {
    this.rename(first, last);
  }
  fullName () {
    return this.firstName + " " + this.lastName;
  }
  rename (first, last) {
    this.firstName = first;
    this.lastName = last;
    return this;
  }
};

const IsAuthor = (function () {
  const books = Symbol();

  return {
    addBook (name) {
      this.books().push(name);
      return this;
    },
    books () {
      return this[books] || (this[books] = []);
    }
  };
})();

const IsBibliophile = (function () {
  const books = Symbol();

  return {
    addToCollection (name) {
      this.collection().push(name);
      return this;
    },
    collection () {
      return this[books] || (this[books] = []);
    }
}
```

```
};

})();

class BookLovingAuthor extends Person {

}

Object.assign(BookLovingAuthor.prototype, IsBibliophile, IsAuthor);

new BookLovingAuthor('Isaac', 'Asimov')
  .addBook('I Robot')
  .addToCollection('The Mysterious Affair at Styles')
  .collection()
  //=> ["The Mysterious Affair at Styles"]
  .books().
  //=> ["I Robot"]
```

Using symbols for property keys eliminates property conflicts between mixins.

Reducing Coupling

When classes are built in a hierarchy, or mixins are distributed across a code base, coupling arises over time. Typically, as a code base evolves, each iteration of programmer uses whatever methods or properties have been made available by the accumulated efforts of previous iterations.

As time goes on, the number of methods and properties increases, and each new piece of behaviour touches more and more methods and properties. When it comes time to refactor the code base, it can be very difficult to tease behaviour apart, since so many pieces naturally end up depending on each other.

One way to resist this natural tendency toward coupling is by making sure that each metaobject exposes only the methods it confers upon its receivers. All other methods and properties should be kept private.

Note that making properties private is not an ideological issue: It's not a question of "purity in OO theory." It's a practical issue: It's a question of minimizing the surface area of the metaobject in order to minimize the ways in which it can become coupled to other objects.

using symbols to reduce coupled properties

We have seen that using symbols as property keys prevents mixins from accidentally sharing the same property name for different purposes. They can also help prevent programmers from *deliberately* using the same property name for different purposes.

Here's why we care about that. Consider:

```
class Person {
    constructor (first, last) {
        this.rename(first, last);
    }
    fullName () {
        return this.firstName + " " + this.lastName;
    }
    rename (first, last) {
        this.firstName = first;
        this.lastName = last;
        return this;
    }
}

class Bibliophile extends Person {
    constructor (first, last) {
        super(first, last);
    }
}
```

```

    this._books = [];
}
addToCollection (name) {
  this._books.push(name);
  return this;
}
hasInCollection (name) {
  return this._books.indexOf(name) >= 0;
}
}

const bezos = new Bibliophile('jeff', 'bezos')
.addToCollection("The Everything Store: Jeff Bezos and the Age of Amazon")
.hasInCollection("Matthew and the Wellington Boots")
//=> false

bezos
.hasInCollection("The Everything Store: Jeff Bezos and the Age of Amazon")
//=> true

```

Note that `_books` is an array. Now consider:

```

class BookGlutton extends Bibliophile {
  buyInBulk (...names) {
    this.books().push(...names);
    return this;
  }
}

```

Book gluttons can buy books in bulk, ordinary bibliophiles cannot. So far, so good. But we have a very naïve implementation of book collections: an array is a linear data structure, the performance of `hasInCollection` is order n . The moment we have a bibliophile with a really large collection, the operation becomes excruciatingly slow.

Simplifying greatly, what if we refactor `Bibliophile` to use a Set?

```
class Bibliophile extends Person {
    constructor (first, last) {
        super(first, last);
        this._books = new Set();
    }
    addToCollection (name) {
        this._books.add(name);
        return this;
    }
    hasInCollection (name) {
        return this._books.has(name);
    }
}
```

Much faster, but we just broke our BookGlutton subclass. This is a very small and contrived example, but the phenomenon is very real, and the larger the class hierarchy, the more it occurs. The author of our BookGlutton subclass coupled BookGlutton to an implementation detail of `Bibliophile`. That's a "feature" of open recursion, but it is far wiser to prevent this from happening.

Naturally, we can use the same technique to prevent deliberate coupling of subclasses that we used to prevent accidental property conflicts: Symbols.

```
const Bibliophile = (function () {
    const books = Symbol("books");

    return class Bibliophile extends Person {
        constructor (first, last) {
            super(first, last);
            this[books] = [];
        }
        addToCollection (name) {
            this[books].push(name);
            return this;
        }
        hasInCollection (name) {
            return this[books].indexOf(name) >= 0;
        }
    };
})();
```

Now anyone subclassing `Bibliophile` is strongly discouraged from directly accessing the "books" property:

```
class BookGlutton extends Bibliophile {  
    buyInBulk (...names) {  
        for (let name of names) {  
            this.addToCollection(name);  
        }  
        return this;  
    }  
}
```

Problem solved.

More Decorators



The delight of coffee is that it transports you to another world

(this bonus chapter is a work-in-progress)

Stateful Method Decorators

As noted in [Method Decorators](#), and again in [Symmetry, Colour, and Charm](#), simple function decorators work well for ordinary functions. But in JavaScript, functions can be invoked in different ways, and some of those ways are slightly incompatible with each other.

Of great interest to us are *methods* in JavaScript, functions that are used to define the behaviour of instances. When a function is invoked as a method, the name `this` is bound to the instance, and most methods rely on that binding to work properly.

Consider, for example the simple decorator `requireAll`, that raises an exception if a function is invoked without at least as many arguments as declared parameters:

```
const requireAll = (fn) =>
  function (...args) {
    if (args.length < fn.length)
      throw new Error('missing required arguments');
    else
      return fn(...args);
  }
```

`requireAll` works perfectly with ordinary functions, what we called “blue” invocations. But if we want to use `requireAll` with methods, we have to write it in such a way that it preserves `this` when it invokes the underlying function:

```
const requireAll = (fn) =>
  function (...args) {
    if (args.length < fn.length)
      throw new Error('missing required arguments');
    else
      return fn.apply(this, args);
  }
```

It now works properly, including ignoring invocations that do not pass all the arguments. But you have to be very careful when writing higher-order functions to make sure they work as both function decorators and as method decorators.

We called this style of decorator a “green” decorator, because it handles blue (ordinary function) and yellow (method) invocations.

the problem with state

Handling `this` properly is not the only way in which ordinary function decorators differ from method decorators. Some decorators are stateful, like `once`. Here’s a version that correctly sets `this`:

```
const once = (fn) => {
  let hasRun = false;

  return function (...args) {
    if (hasRun) return;
    hasRun = true;
    return fn.apply(this, args);
  }
}
```

Imagining for a moment that we wish to only allow a person to have their name set once, we might write:

```
const once = (fn) => {
  let hasRun = false;

  return function (...args) {
    if (hasRun) return;
    hasRun = true;
    return fn.apply(this, args);
  }
}

class Person {
  setName (first, last) {
    this.firstName = first;
    this.lastName = last;
    return this;
  }
  fullName () {
    return this.firstName + " " + this.lastName;
  }
};

Object.defineProperty(Person.prototype, 'setName', { value: once(Person.prototype.setName) });

const logician = new Person()
  .setName('Raymond', 'Smullyan')
  .setName('Haskell', 'Curry');

logician.fullName()
//=> Raymond Smullyan
```

As we expect, only the first call to `.setName` has any effect, and it works on a method. But there is a subtle bug that could easily evade naïve attempts to write unit tests:

```
const logician = new Person()
  .setName('Raymond', 'Smullyan');

const musician = new Person()
  .setName('Miles', 'Davis');

logician.fullName()
//=> Raymond Smullyan

musician.fullName()
//=> Raymond Smullyan
```

!?!?!

What has happened here is that when we write `Object.defineProperty(Person.prototype, 'setName', { value: once(Person.prototype.setName) })`, we wrapped a function bound to `Person.prototype`. That function is shared between all instances of `Person`. That's deliberate, it's the whole point of prototypical inheritance (and the “class-based inheritance” JavaScript builds with prototypes).

Since our `once` decorator returns a decorated function with private state (the `hasRun` variable), all the instances share the same private state, and thus the bug.

writing stateful method decorators

If we don't need to use the same decorator for functions and for methods, we can rewrite our decorator to use a `WeakSet`¹⁴⁸ to track whether a method has been invoked for an instance:

```
const once = (fn) => {
  let invocations = new WeakSet();

  return function (...args) {
    if (invocations.has(this)) return;
    invocations.add(this);
    return fn.apply(this, args);
  }
}
```

¹⁴⁸https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/WeakSet

```
const logician = new Person()
    .setName('Raymond', 'Smullyan');

logician.setName('Haskell', 'Curry');

const musician = new Person()
    .setName('Miles', 'Davis');

logician.fullName()
//=> Raymond Smullyan

musician.fullName()
//=> Miles Davis
```

Now each instance stores whether `.setName` has been invoked on each instance a `WeakSet`, so `logician` and `musician` can share the method without sharing its state.

incompatibility

To handle methods, we have introduced “accidental complexity” to handle `this` and to handle state. Worse, our implementation of `once` for methods won’t work properly with ordinary functions in “strict” mode:

```
"use strict"

const hello = once(() => 'hello!');

hello()
//=> undefined is not an object!
```

If you haven’t invoked it as a method, `this` is bound to `undefined` in strict mode, and `undefined` cannot be added to a `WeakSet`.

Correcting our decorator to deal with `undefined` is straightforward:

```
const once = (fn) => {
  let invocations = new WeakSet(),
    undefinedContext = Symbol('undefined-context');

  return function (...args) {
    const context = this === undefined
      ? undefinedContext
      : this;
    if (invocations.has(context)) return;
    invocations.add(context);
    return fn.apply(this, args);
  }
}
```

However, we're adding more accidental complexity to handle the fact that function invocation is `blue`, and method invocation is `khaki`.¹⁴⁹

In the end, we can either write specialized decorators designed specifically for methods, or tolerate the additional complexity of trying to handle method invocation and function invocation in the same decorator.

¹⁴⁹See the aforelinked [The Symmetry of JavaScript Functions](#)

Class Decorators beyond ES6/ECMAScript 2015

In [Functional Mixins](#), we discussed mixing functionality *into* JavaScript classes, changing the class. We observed that this has pitfalls when applied to a class that might already be in use elsewhere, but is perfectly cromulant when used as a technique to build a class from scratch. When used strictly to build a class, mixins help us decompose classes into smaller entities with focused responsibilities that can be shared between classes as necessary.

Let's recall our helper for making a functional mixin:

```
function FunctionalMixin (behaviour, sharedBehaviour = {}) {
  const instanceKeys = Reflect.ownKeys(behaviour);
  const sharedKeys = Reflect.ownKeys(sharedBehaviour);
  const typeTag = Symbol("isA");

  function mixin (target) {
    for (let property of instanceKeys)
      if (!target[property])
        Object.defineProperty(target, property, {
          value: behaviour[property],
          writable: true
        })
    target[typeTag] = true;
    return target;
  }
  for (let property of sharedKeys)
    Object.defineProperty(mixin, property, {
      value: sharedBehaviour[property],
      enumerable: sharedBehaviour.propertyIsEnumerable(property)
    });
  Object.defineProperty(mixin, Symbol.hasInstance, { value: (instance) => !!instance[typeTag] });
  return mixin;
}
```

This creates a function that mixes behaviour into any target, be it a class prototype or a standalone object. There is a convenience capability of making “static” or “shared” properties of the the function, and it even adds some simple `hasInstance` handling so that the `instanceof` operator will work.

Here we are using it on a class’ prototype:

```
const BookCollector = FunctionalMixin({  
    addToCollection (name) {  
        this.collection().push(name);  
        return this;  
    },  
    collection () {  
        return this._collected_books || (this._collected_books = []);  
    }  
});  
  
class Person {  
    constructor (first, last) {  
        this.rename(first, last);  
    }  
    fullName () {  
        return this.firstName + " " + this.lastName;  
    }  
    rename (first, last) {  
        this.firstName = first;  
        this.lastName = last;  
        return this;  
    }  
};  
  
BookCollector(Person.prototype);  
  
const president = new Person('Barak', 'Obama')  
  
president  
.addToCollection("JavaScript Allongé")  
.addToCollection("Kestrels, Quirky Birds, and Hopeless Egocentricity");  
  
president.collection()  
//=> ["JavaScript Allongé", "Kestrels, Quirky Birds, and Hopeless Egocentricity"]
```

mixins that target classes

It's very nice that our mixins support any kind of target, but let's make them class-specific:

```

function ClassMixin (behaviour, sharedBehaviour = {}) {
  const instanceKeys = Reflect.ownKeys(behaviour);
  const sharedKeys = Reflect.ownKeys(sharedBehaviour);
  const typeTag = Symbol("isA");

  function mixin (clazz) {
    for (let property of instanceKeys)
      if (!clazz.prototype[property])
        Object.defineProperty(clazz.prototype, property, {
          value: behaviour[property],
          writable: true
        });
    clazz.prototype[typeTag] = true;
    return clazz;
  }
  for (let property of sharedKeys)
    Object.defineProperty(mixin, property, {
      value: sharedBehaviour[property],
      enumerable: sharedBehaviour.propertyIsEnumerable(property)
    });
  Object.defineProperty(mixin, Symbol.hasInstance, { value: (instance) => !!instance[typeTag] });
  return mixin;
}

```

This version's `mixin` function mixes instance behaviour into a class's prototype, so we gain convenience at the expense of flexibility:

```

const BookCollector = ClassMixin({
  addToCollection (name) {
    this.collection().push(name);
    return this;
  },
  collection () {
    return this._collected_books || (this._collected_books = []);
  }
});

class Person {
  constructor (first, last) {
    this.rename(first, last);
  }
}

```

```

fullName () {
  return this.firstName + " " + this.lastName;
}
rename (first, last) {
  this.firstName = first;
  this.lastName = last;
  return this;
}
};

BookCollector(Person);

const president = new Person('Barak', 'Obama')

president
  .addToCollection("JavaScript Allongé")
  .addToCollection("Kestrels, Quirky Birds, and Hopeless Egocentricity");

president.collection()
  //=> ["JavaScript Allongé", "Kestrels, Quirky Birds, and Hopeless Egocentricity"]

```

So far, nice, but it feels a bit bolted-on-after-the-fact. Let's take advantage of the fact that **Classes are Expressions**¹⁵⁰:

```

const BookCollector = ClassMixin({
  addToCollection (name) {
    this.collection().push(name);
    return this;
  },
  collection () {
    return this._collected_books || (this._collected_books = []);
  }
});

const Person = BookCollector(class {
  constructor (first, last) {
    this.rename(first, last);
  }
  fullName () {
    return this.firstName + " " + this.lastName;
  }
});

```

¹⁵⁰<http://raganwald.com/2015/06/04/classes-are-expressions.html>

```

    }
    rename (first, last) {
      this.firstName = first;
      this.lastName = last;
      return this;
    }
  );
}

```

This is structurally nicer, it binds the mixing in of behaviour with the class declaration in one expression, so we're getting away from this idea of mixing things into classes after they're created.

But (there's always a but), our pattern has three different elements (the name being bound, the mixin, and the class being declared). And if we wanted to mix two or more behaviours in, we'd have to nest the functions like this:

```

const Author = ClassMixin({
  writeBook (name) {
    this.books().push(name);
    return this;
  },
  books () {
    return this._books_written || (this._books_written = []);
  }
});

const Person = Author(BookCollector(class {
  // ...
}));
```

Some people find this “clear as day,” arguing that this is a simple expression taking advantage of JavaScript’s simplicity. The code behind `Mixin` is simple and easy to read, and if you understand prototypes, you understand everything in this expression.

But others want a language to give them “magic,” an abstraction that they learn on the outside. At the moment, JavaScript has no “magic” for mixing functionality into classes. But what if there were?

class decorators

There is a well-regarded [proposal¹⁵¹](#) to add Python-style class decorators to JavaScript in the future, nicknamed “ES.later.”¹⁵²

¹⁵¹<https://github.com/wycats/javascript-decorators>

¹⁵²By “ES.later,” we mean some future version of ECMAScript that is likely to be approved eventually, but for the moment exists only in transpilers like [Babel](#). Obviously, using any ES.later feature in production is a complex decision requiring many more considerations than can be enumerated in a book.

A decorator is a function that operates on a class. Here's a very simple example from the aforelinked implementation:

```
function annotation(target) {
  // Add a property on target
  target.annotated = true;
}

@annotation
class MyClass {
  // ...
}

MyClass.annotated
//=> true
```

As you can see, `annotation` is a class decorator, and it takes a class as an argument. The function can do anything, including modifying the class or the class's prototype. If the decorator function doesn't return anything, the class' name is bound to the modified class.¹⁵³

A class is “decorated” with the function by preceding the definition with `@` and an expression evaluating to the decorator. In the simple example, we use a variable name.

Hmmm. A function that modifies a class, you say? Let's try it:

```
const BookCollector = ClassMixin({
  addToCollection (name) {
    this.collection().push(name);
    return this;
  },
  collection () {
    return this._collected_books || (this._collected_books = []);
  }
});

@BookCollector
class Person {
  constructor (first, last) {
    this.rename(first, last);
  }
  fullName () {
```

¹⁵³Although this example doesn't show it, if it returns a constructor function, that is what will be assigned to the class' name. This allows the creation of purely functional mixins and other interesting techniques that are beyond the scope of this post.

```
    return this.firstName + " " + this.lastName;
}
rename (first, last) {
  this.firstName = first;
  this.lastName = last;
  return this;
}
};

const president = new Person('Barak', 'Obama')

president
  .addToCollection("JavaScript Allongé")
  .addToCollection("Kestrels, Quirky Birds, and Hopeless Egocentricity");

president.collection()
  //=> ["JavaScript Allongé", "Kestrels, Quirky Birds, and Hopeless Egocentricity"]
```

You can also mix in multiple behaviours with decorators:

```
const BookCollector = ClassMixin({
  addToCollection (name) {
    this.collection().push(name);
    return this;
  },
  collection () {
    return this._collected_books || (this._collected_books = []);
  }
});

const Author = ClassMixin({
  writeBook (name) {
    this.books().push(name);
    return this;
  },
  books () {
    return this._books_written || (this._books_written = []);
  }
});
```

@BookCollector @Author

```
class Person {
  constructor (first, last) {
    this.rename(first, last);
  }
  fullName () {
    return this.firstName + " " + this.lastName;
  }
  rename (first, last) {
    this.firstName = first;
    this.lastName = last;
    return this;
  }
};
```

Class decorators provide a compact, “magic” syntax that is closely tied to the construction of the class. They also require understanding one more kind of syntax. But some argue that having different syntax for different things aids understandability, and that having both `@foo` for decoration and `bar(...)` for function invocation is a win.

Decorators have not been formally approved, however there are various implementations available for transpiling decorator syntax to ES5 syntax. These examples were evaluated with [Babel](#)¹⁵⁴.

¹⁵⁴<http://babeljs.io>

Method Decorators beyond ES6/ECMAScript 2015

Before ES6/ECMAScript 2015, we decorated a method in a simple and direct way. Given a method decorator like `fluent` (a/k/a chain):

```
const fluent = (method) =>
  function (...args) {
    method.apply(this, args);
    return this;
}
```

We would wrap functions in our decorator and bind them to names to create methods, like this:

```
const Person = function () {};

Person.prototype.setName = fluent(function setName (first, last) {
  this.firstName = first;
  this.lastName = last;
});

Person.prototype.fullName = function fullName () {
  return this.firstName + " " + this.lastName;
};
```

With the `class` keyword, we have a more elegant way to do everything in one step:

```
class Person {

  setName (first, last) {
    this.firstName = first;
    this.lastName = last;
    return this;
}

  fullName () {
    return this.firstName + " " + this.lastName;
}

}
```

Since the ECMAScript 2015 syntaxes for classes doesn't give us any way to decorate a method where we are declaring it, we have to introduce this ugly "post-decoration" after we've declared `Person`:

```
Object.defineProperty(Person.prototype, 'setName', { value: fluent(Person.prototype.setName) });
```

This is weak for two reasons. First, it's fugly and full of accidental complexity. Second, modifying the prototype after defining the class separates two things that conceptually ought to be together. The `class` keyword giveth, but it also taketh away.

es.later method decorators

To solve a problem created by ECMAScript 2015, [method decorators¹⁵⁵](#) have been proposed for a future version of JavaScript (nicknamed “ES.later.”¹⁵⁶) The syntax is similar to [class decorators](#), but where a class decorator takes a class as an argument and returns the same (or a different) class, a method decorator actually intercedes when a property is defined on the prototype.

An ES.later decorator version of `fluent` would look like this:

```
function fluent (target, name, descriptor) {
  const method = descriptor.value;

  descriptor.value = function (...args) {
    method.apply(this, args);
    return this;
  }
}
```

And we'd use it like this:

```
class Person {

  @fluent
  setName (first, last) {
    this.firstName = first;
    this.lastName = last;
  }

  fullName () {
    return this.firstName + " " + this.lastName;
  }

};
```

¹⁵⁵<https://github.com/wycats/javascript-decorators>

¹⁵⁶By “ES.later,” we mean some future version of ECMAScript that is likely to be approved eventually, but for the moment exists only in transpilers like [Babel](#). Obviously, using any ES.later feature in production is a complex decision requiring many more considerations than can be enumerated in a book.

That is much nicer: It lets us use the new class syntax, and it also lets us decompose functionality with method decorators. Best of all, when we write our classes in a “declarative” way, we also write our decorators in a declarative way.

Mind you, we are once again creating two kinds of decorators: One for functions, and one for methods, with different structures. We need a new [colour!](#)

But all elegance is not lost. Since decorators are expressions, we can alleviate the pain with an adaptor:

```
const wrapWith = (decorator) =>
  function (target, name, descriptor) {
    descriptor.value = decorator(descriptor.value);
  }

function fluent (method) {
  return function (...args) {
    method.apply(this, args);
    return this;
  }
}

class Person {

  @wrapWith(fluent)
  setName (first, last) {
    this.firstName = first;
    this.lastName = last;
  }

  fullName () {
    return this.firstName + " " + this.lastName;
  }
};
```

Or if we prefer:

```
const wrapWith = (decorator) =>
  function (target, name, descriptor) {
    descriptor.value = decorator(descriptor.value);
  }

const returnsItself = wrapWith(
  function fluent (method) {
    return function (...args) {
      method.apply(this, args);
      return this;
    }
  }
);

class Person {

  @returnsItself
  setName (first, last) {
    this.firstName = first;
    this.lastName = last;
  }

  fullName () {
    return this.firstName + " " + this.lastName;
  }
};
```

(Although ES.later has not been approved, there is extensive support for ES.later method decorators in transpilation tools. The examples in this post were evaluated with [Babel¹⁵⁷](#).)

¹⁵⁷<http://babeljs.io>

Lightweight Traits

A **trait** is a concept used in object-oriented programming: a trait represents a collection of methods that can be used to extend the functionality of a class. Essentially a trait is similar to a class made only of concrete methods that is used to extend another class with a mechanism similar to multiple inheritance, but paying attention to name conflicts, hence with some support from the language for a name-conflict resolution policy to use when merging.—[Wikipedia¹⁵⁸](#)

A trait is like a [mixin](#), however with a trait, we can not just define new behaviour, but also define ways to extend or override existing behaviour. Traits are a first-class feature languages like [Scala¹⁵⁹](#). Traits are also available as a standard library in other languages, like [Racket¹⁶⁰](#). Most interestingly, traits are a feature of the [Self¹⁶¹](#) programming language, one of the inspirations for JavaScript.

Traits are not a JavaScript feature as this essay is being written, but we can easily make lightweight traits out of the features JavaScript already has.

Our problem is that we want to be able to override or extend functionality from shared behaviour, whether that shared behaviour is defined as a class or as functionality to be mixed in.

our toy problem

Here's a toy problem we solved elsewhere with a [subclass factory](#) that in turn is made out of an extremely simple mixin.¹⁶²

To recapitulate from the very beginning, we have a Todo class:

¹⁵⁸https://en.wikipedia.org/wiki/Trait_

¹⁵⁹<http://www.scala-lang.org>

¹⁶⁰<http://docs.racket-lang.org/reference/trait.html>

¹⁶¹[https://en.wikipedia.org/wiki/Self_\(programming_language\)#Traits](https://en.wikipedia.org/wiki/Self_(programming_language)#Traits)

¹⁶²The implementations given here are extremely simple in order to illustrate a larger principle of how the pieces fit together. A production library based on these principles would handle needs we've seen elsewhere, like defining "class" or "static" properties, making `instanceof` work, and appeasing the V8 compiler's optimizations.

```

class Todo {
    constructor (name) {
        this.name = name || 'Untitled';
        this.done = false;
    }

    do () {
        this.done = true;
        return this;
    }

    undo () {
        this.done = false;
        return this;
    }

    toHTML () {
        return this.name; // highly insecure
    }
}

```

And we have the idea of “things that are coloured:”

```

let toSixteen = (c) => '0123456789ABCDEF'.indexOf(c),
    toTwoFiftyFive = (cc) => toSixteen(cc[0]) * 16 + toSixteen(cc[1]);

class Coloured {
    setColourRGB ({r, g, b}) {
        this.colourCode = {r, g, b};
        return this;
    }

    luminosity () {
        let {r, g, b} = this.getColourRGB();

        return 0.21 * toTwoFiftyFive(r) +
            0.72 * toTwoFiftyFive(g) +
            0.07 * toTwoFiftyFive(b);
    }

    getColourRGB () {
        return this.colourCode;
    }
}

```

```

    }
}
```

And we want to create a time-sensitive to-do that has colour according to whether it is overdue, close to its deadline, or has plenty of time left. If we had multiple inheritance, we would write:

```

let yellow = {r: 'FF', g: 'FF', b: '00'},
  red    = {r: 'FF', g: '00', b: '00'},
  green  = {r: '00', g: 'FF', b: '00'},
  grey   = {r: '80', g: '80', b: '80'};

let oneDayInMilliseconds = 1000 * 60 * 60 * 24;

class TimeSensitiveTodo extends Todo, Coloured {
  constructor (name, deadline) {
    super(name);
    this.deadline = deadline;
  }

  getColourRGB () {
    let slack = this.deadline - Date.now();

    if (this.done) {
      return grey;
    }
    else if (slack <= 0) {
      return red;
    }
    else if (slack <= oneDayInMilliseconds){
      return yellow;
    }
    else return green;
  }

  toHTML () {
    let rgb = this.getColourRGB();

    return `<span style="color: ${rgb.r}${rgb.g}${rgb.b};">${super.toHTML()}</s>
pan>`;
  }
}
```

But we don't have multiple inheritance. In languages where mixing in functionality is difficult, we can fake a solution by having ColouredTodo inherit from Todo:

```
class ColouredTodo extends Todo {
    setColourRGB ({r, g, b}) {
        this.colourCode = {r, g, b};
        return this;
    }

    luminosity () {
        let {r, g, b} = this.getColourRGB();

        return 0.21 * toTwoFiftyFive(r) +
            0.72 * toTwoFiftyFive(g) +
            0.07 * toTwoFiftyFive(b);
    }

    getColourRGB () {
        return this.colourCode;
    }
}

class TimeSensitiveTodo extends ColouredTodo {
    constructor (name, deadline) {
        super(name);
        this.deadline = deadline;
    }

    getColourRGB () {
        let slack = this.deadline - Date.now();

        if (this.done) {
            return grey;
        }
        else if (slack <= 0) {
            return red;
        }
        else if (slack <= oneDayInMilliseconds){
            return yellow;
        }
        else return green;
    }
}
```

```
toHTML () {
    let rgb = this.getColourRGB();

    return `<span style="color: ${rgb.r}${rgb.g}${rgb.b};">${super.toHTML()}</span>`;
}
```

The drawback of this approach is that we can no longer make other kinds of things “coloured” without making them also todos. For example, if we had coloured meetings in a time management application, we’d have to write:

```
class Meeting {
    // ...

class ColouredMeeting extends Meeting {
    setColourRGB ({r, g, b}) {
        this.colourCode = {r, g, b};
        return this;
    }

    luminosity () {
        let {r, g, b} = this.getColourRGB();

        return 0.21 * toTwoFiftyFive(r) +
            0.72 * toTwoFiftyFive(g) +
            0.07 * toTwoFiftyFive(b);
    }

    getColourRGB () {
        return this.colourCode;
    }
}
```

This forces us to duplicate “coloured” functionality throughout our code base. But thanks to mixins, we can have our cake and eat it to: We can make `ColouredAsWellAs` a kind of mixin that makes a new subclass and then mixes into the subclass. We call this a “subclass factory.”

```
function ClassMixin (behaviour) {
  const instanceKeys = Reflect.ownKeys(behaviour);

  return function mixin (clazz) {
    for (let property of instanceKeys)
      Object.defineProperty(clazz.prototype, property, {
        value: behaviour[property],
        writable: true
      });
    return clazz;
  }
}

const SubclassFactory = (behaviour) =>
  (superclazz) => ClassMixin(behaviour)(class extends superclazz {});

const ColouredAsWellAs = SubclassFactory({
  setColourRGB ({r, g, b}) {
    this.colourCode = {r, g, b};
    return this;
  ,

  luminosity () {
    let {r, g, b} = this.getColourRGB();

    return 0.21 * toTwoFiftyFive(r) +
      0.72 * toTwoFiftyFive(g) +
      0.07 * toTwoFiftyFive(b);
  ,

  getColourRGB () {
    return this.colourCode;
  }
});

class TimeSensitiveTodo extends ColouredAsWellAs(Todo) {
  constructor (name, deadline) {
    super();
    this.deadline = deadline;
  }

  getColourRGB () {
```

```

let slack = this.deadline - Date.now();

if (this.done) {
  return grey;
}
else if (slack <= 0) {
  return red;
}
else if (slack <= oneDayInMilliseconds){
  return yellow;
}
else return green;
}

toHTML () {
  let rgb = this.getColourRGB();

  return `<span style="color: ${rgb.r}${rgb.g}${rgb.b};">${super.toHTML()}</span>`;
}
}

```

This allows us to override both our Todo methods and the ColourAsWellAs methods. And elsewhere, we can write:

```
const ColouredMeeting = ColouredAsWellAs(Meeting);
```

Or perhaps:

```
class TimeSensitiveMeeting extends ColouredAsWellAs(Meeting) {
  // ...
}
```

To summarize, our problem is that we want to be able to override or extend functionality from shared behaviour, whether that shared behaviour is defined as a class or as functionality to be mixed in. Subclass factories are one way to solve that problem.

Now we'll solve the same problem with traits.

defining lightweight traits

Let's start with our ClassMixin. We'll modify it slightly to insist that it never attempt to define a method that already exists, and we'll use that to create Coloured, a function that defines two methods:

```

function Define (behaviour) {
  const instanceKeys = Reflect.ownKeys(behaviour);

  return function define (clazz) {
    for (let property of instanceKeys)
      if (!clazz.prototype[property]) {
        Object.defineProperty(clazz.prototype, property, {
          value: behaviour[property],
          writable: true
        });
      }
      else throw `illegal attempt to override ${property}, which already exists.
    }
  }
}

const Coloured = Define({
  setColourRGB ({r, g, b}) {
    this.colourCode = {r, g, b};
    return this;
  },

  luminosity () {
    let {r, g, b} = this.getColourRGB();

    return 0.21 * toTwoFiftyFive(r) +
      0.72 * toTwoFiftyFive(g) +
      0.07 * toTwoFiftyFive(b);
  },

  getColourRGB () {
    return this.colourCode;
  }
});

```

Coloured is now a function that modifies a class, adding two methods provided that they don't already exist in the class.

But we need a variation that "overrides" getColourRGB. We can write a variation of Define that always overrides the target's methods, and passes in the original method as the first parameter. This is similar to "around" [method advice][ma-mj]:

```
function Override (behaviour) {
  const instanceKeys = Reflect.ownKeys(behaviour);

  return function overrides (clazz) {
    for (let property of instanceKeys)
      if (!!clazz.prototype[property]) {
        let overriddenMethodFunction = clazz.prototype[property];

        Object.defineProperty(clazz.prototype, property, {
          value: function (...args) {
            return behaviour[property].call(this, overriddenMethodFunction.bind(
              this), ...args);
          },
          writable: true
        });
      }
      else throw `attempt to override non-existent method ${property}`;
    return clazz;
  }
}

const DeadlineSensitive = Override({
  getColourRGB () {
    let slack = this.deadline - Date.now();

    if (this.done) {
      return grey;
    }
    else if (slack <= 0) {
      return red;
    }
    else if (slack <= oneDayInMilliseconds){
      return yellow;
    }
    else return green;
  },
  toHTML (original) {
    let rgb = this.getColourRGB();

    return `<span style="color: ${rgb.r}${rgb.g}${rgb.b};">${original()}</span>`;
  }
});
```

```
    }
});
```

Define and Override are *protocols*: They define whether methods may conflict, and if they do, how that conflict is resolved. Define prohibits conflicts, forcing us to pick another protocol. Override permits us to write a method that overrides an existing method and (optionally) call the original.

composing protocols

We *could* now write:

```
const TimeSensitiveTodo = DeadlineSensitive(
  Coloured(
    class TimeSensitiveTodo extends Todo {
      constructor (name, deadline) {
        super(name);
        this.deadline = deadline;
      }
    }
  )
);
```

Or:

```
@DeadlineSensitive
@Coloured
class TimeSensitiveTodo extends Todo {
  constructor (name, deadline) {
    super(name);
    this.deadline = deadline;
  }
}
```

But if we want to use DeadlineSensitive and Coloured together more than once, we can make a lightweight trait with the `pipeline` function:

```
const SensitizeTodos = pipeline(Coloured, DeadlineSensitive);

@SensitizeTodos
class TimeSensitiveTodo extends Todo {
  constructor (name, deadline) {
    super(name);
    this.deadline = deadline;
  }
}
```

Now `SensitizeTodos` combines defining methods with overriding existing methods: We've built a lightweight trait by composing protocols.

And that's all a trait is: The composition of protocols. And we don't need a bunch of new keywords or decorators (like `@overrides`) to do it, we just use the functional composition that is so easy and natural in JavaScript.

other protocols

We can incorporate other protocols. Two of the most common are prepending behaviour to an existing method, or appending behaviour to an existing method:

```
function Prepends (behaviour) {
  const instanceKeys = Reflect.ownKeys(behaviour);

  return function prepend (clazz) {
    for (let property of instanceKeys)
      if (!!clazz.prototype[property]) {
        let overriddenMethodFunction = clazz.prototype[property];

        Object.defineProperty(clazz.prototype, property, {
          value: function (...args) {
            const prependValue = behaviour[property].apply(this, args);

            if (prependValue === undefined || !prependValue) {
              return overriddenMethodFunction.apply(this, args);
            }
          },
          writable: true
        });
      }
    else throw `attempt to override non-existent method ${property}`;
  }
}
```

```
    return clazz;
}
}

function Append (behaviour) {
  const instanceKeys = Reflect.ownKeys(behaviour);

  function append (clazz) {
    for (let property of instanceKeys)
      if (!!clazz.prototype[property]) {
        let overriddenMethodFunction = clazz.prototype[property];

        Object.defineProperty(clazz.prototype, property, {
          value: function (...args) {
            const returnValue = overriddenMethodFunction.apply(this, args);

            behaviour[property].apply(this, args);
            return returnValue;
          },
          writable: true
        });
      }
    else throw `attempt to override non-existant method ${property}`;
  }
  return clazz;
}
```

We can compose a lightweight trait using any combination of Define, Override, Prepend, and Append, and the composition is handled by pipeline, a plain old function composition tool.

Lightweight traits are nothing more than protocols, composed in a simple and easy-to-understand way. And then applied to simple classes, in a direct and obvious manner.

More Decorator Recipes



For the love of coffee: a collection

“The entire history of Mankind’s relationship with coffee is a futile attempt to have the reality of its taste live up to the promise of its aroma.”

After Method Advice

Consider the bare bones of this Todo class that we might use as part of a ViewModel in a front-end application. Many front-end libraries have special features that allow views or other models to persist changes to one or more actual models and/or data stores.

We'll just hand-wave by pretending there is a `persist` method. It could be mixed in or inherited, we'll sketch it in for illustration:

```
class Todo {  
    constructor (name) {  
        this.name = name || 'Untitled';  
        this.done = false  
    }  
  
    do () { this.done = true; }  
  
    undo () { this.done = false; }  
  
    setName (name) { this.name = name; }  
  
    persist () {  
        // persist changes to model(s) and/or  
        // data stores...  
    }  
}
```

Naturally, updating a todo should persist changes. So we could write:

```
class Todo {  
    constructor (name) {  
        this.name = name || 'Untitled';  
        this.done = false  
    }  
  
    do () {  
        this.done = true;  
        this.persist();  
    }  
  
    undo () {  
        this.done = false;
```

```

    this.persist();
}

setName (name) {
  this.name = name;
  this.persist();
}

persist () {
  // persist changes to model(s) and/or
  // data stores...
}
}
}

```

This is very similar to making methods fluent. We're obscuring the primary responsibility of the method with cross-cutting concerns. We can and should abstract persistence into an `ES.later` decorator:

```

const persists = function (target, name, descriptor) {
  const method = descriptor.value;

  descriptor.value = function (...args) {
    const value = method.apply(this, args);

    this.persist();
    return value;
  }
}

class Todo {
  constructor (name) {
    this.name = name || 'Untitled';
    this.done = false
  }

  @persists
  do () {
    this.done = true;
  }

  @persists
  undo () {

```

```

    this.done = false;
}

@persists
setName (name) {
  this.name = name;
}

persist () {
  console.log(`persisting ${this.name}`);
}
}

```

after decorators

Combinators for functions come in an unlimited panoply of purposes and effects. So do method combinators, but whether from intrinsic utility or custom, certain themes have emerged. One of them that forms a core part of the original [Lisp Flavors¹⁶³](#) system and also the [Aspect-Oriented Programming¹⁶⁴](#) movement, is decorating a method with some functionality to be performed *after* the method's body is evaluated.

What we see above is this pattern: We want to decorate a method with some functionality. Instead of writing a decorator from scratch each time, let's abstract the wrapping into a combinator that makes an `ES.later` method decorator:

```

const after = (...fns) =>
  function (target, name, descriptor) {
    const method = descriptor.value;

    descriptor.value = function (...args) {
      const value = method.apply(this, args);

      for (let fn of fns) {
        fn.apply(this, args);
      }
      return value;
    }
  }
}

```

Now we could write:

¹⁶³https://en.wikipedia.org/wiki/Flavors_

¹⁶⁴https://en.wikipedia.org/wiki/Aspect-oriented_programming

```
const persists = after(function () { this.persist(); });
```

Or we could write:

```
const persists = after(Todo.prototype.persist);
```

Or we could even write these things inline:

```
class Todo {
  constructor (name) {
    this.name = name || 'Untitled';
    this.done = false
  }

  @after(Todo.prototype.persist)
  do () {
    this.done = true;
  }

  @after(Todo.prototype.persist)
  undo () {
    this.done = false;
  }

  @after(Todo.prototype.persist)
  setName (name) {
    this.name = name;
  }

  persist () {
    console.log(`persisting ${this.name}`);
  }
}
```

`Todo.prototype.persist` is a little clunky. We could special-case `after` to allow us to write `@after('persist')` as some libraries do, but the beauty of combinators is that they, well, *combine*. Recall `send`. It's perfect for this:

```
const send = (methodName, ...args) =>
  (instance) => instance[methodName].apply(instance, args);

class Todo {
  constructor (name) {
    this.name = name || 'Untitled';
    this.done = false
  }

  @after(send('persist'))
  do () {
    this.done = true;
  }

  @after(send('persist'))
  undo () {
    this.done = false;
  }

  @after(send('persist'))
  setName (name) {
    this.name = name;
  }

  persist () {
    console.log(`persisting ${this.name}`);
  }
}
```

after is a combinator that makes ES.later method decorators, and it's handy for separating concerns.

Before Method Advice

Just as we often wish to decorate a method with [after advice](#), we also often wish to decorate methods with some behaviour that happens before a method is invoked. The canonical (and greatly overused) example is logging invocations. But let's consider another example, a Person:

```
const firstName = Symbol('firstName'),
lastName = Symbol('lastName');

class Person {
  constructor (first, last) {
    this(firstName) = first;
    this(lastName) = last;
  }

  fullName () {
    return this.firstName + " " + this.lastName;
  }

  rename (first, last) {
    this(firstName) = first;
    this(lastName) = last;
    return this;
  }
};
```

What if we wish to make `rename` an undoable action? Let's add a stack. For reasons known only to a secret cabal of enterprisey architects, we wish to make the undo stack something that is lazily initialized, like this:

```
const firstName = Symbol('firstName'),
lastName = Symbol('lastName'),
undoStack = Symbol('undoStack'),
redoStack = Symbol('redoStack');

class Person {
  constructor (first, last) {
    this(firstName) = first;
    this(lastName) = last;
  }

  fullName () {
```

```
    return this.firstName] + " " + this.lastName];
}

rename (first, last) {
  this.undoStack || (this.undoStack = []);
  this.undoStack.push({
    [firstName]: this.firstName,
    [lastName]: this.lastName
  });
  this.firstName = first;
  this.lastName = last;
  return this;
}

undo () {
  this.undoStack || (this.undoStack = []);
  let oldState = this.undoStack.pop();

  if (oldState != null) Object.assign(this, oldState);
  return this;
};

const b = new Person('barak', 'obama');

b.rename('Barak', 'Obama');
b.fullName()
//=> 'Barak Obama'

b.undo();
b.fullName()
//=> 'barak obama'
```

We can follow the same pattern as we did with [after advice](#): Extract the common functionality into a decorator. We'll write the `before` combinator to help:

```
const before = (...fns) =>
  function (target, name, descriptor) {
    const method = descriptor.value;

    descriptor.value = function (...args) {
      for (let fn of fns) {
        fn.apply(this, args);
      }
      return method.apply(this, args);
    }
  }

const firstName = Symbol('firstName'),
lastName = Symbol('lastName'),
undoStack = Symbol('undoStack'),
redoStack = Symbol('redoStack');

const usingUndoStack = before(function () {
  this[undoStack] || (this[undoStack] = []);
})

class Person {
  constructor (first, last) {
    this(firstName) = first;
    this(lastName) = last;
  }

  fullName () {
    return this.firstName + " " + this.lastName;
  }

  @usingUndoStack
  rename (first, last) {
    this.undoStack.push({
      [firstName]: this.firstName,
      [lastName]: this.lastName
    });
    this.firstName = first;
    this.lastName = last;
    return this;
  }
}
```

```
@usingUndoStack
undo () {
  let oldState = this[undoStack].pop();

  if (oldState != null) Object.assign(this, oldState);
  return this;
};

const b = new Person('barak', 'obama')
b.rename('Barak', 'Obama')
console.log(b.fullName())
b.undo()
console.log(b.fullName())
```

We could, of course, also abstract functionality into a method that we invoke with `@after(send('usingUndoStack'))` just as we did with our `after advice` examples.

Provided and Unless

Neither the `before` and `after` ES.later method decorators actually terminate evaluation without throwing something. Normal execution always results in the base method being evaluated. The `provided` and `unless` recipes are combinators that produce method decorators that apply a precondition to evaluating the base method body.

The `provided` combinator turns a function into an ES.later method decorator. The function (or functions) is passed the method arguments before the base method, and it must evaluate to truthy for the base method to be evaluated. The `unless` combinator does the same thing, but the logic is reversed, the decorating function must not evaluate to truthy:

```
const provided = (...fns) =>
  function (target, name, descriptor) {
    const method = descriptor.value;

    descriptor.value = function (...args) {
      for (let fn of fns) {
        const result = fn.apply(this, args);

        if (!result) return;
      }
      return method.apply(this, args);
    }
  }

const unless = (...fns) =>
  function (target, name, descriptor) {
    const method = descriptor.value;

    descriptor.value = function (...args) {
      for (let fn of fns) {
        const result = fn.apply(this, args);

        if (result) return;
      }
      return method.apply(this, args);
    }
  }
```

`provided` can be used to check that non-empty strings are provided for names:¹⁶⁵

¹⁶⁵Beware, validating names is a stygian task. Read [falsehoods programmers believe about names](#) before proceeding with ideas like this in production. For example, many people do NOT have both a first and last name.

```
const firstName = Symbol('firstName'),
lastName = Symbol('lastName');

const nonEmptyStrings = (...strs) =>
  strs.reduce((truth, str) =>
    truth
    && (str instanceof String || typeof str === 'string')
    && str !== '',
  true);

class Person {
  constructor (first, last) {
    this[firstName] = first;
    this[lastName] = last;
  }

  fullName () {
    return this(firstName] + " " + this[lastName];
  }

  @provided(nonEmptyStrings)
  rename (first, last) {
    this[firstName] = first;
    this[lastName] = last;
    return this;
  }
};

const b = new Person('barak', 'obama')
b.rename('Barak', 'Obama')
console.log(b.fullName())
b.undo()
console.log(b.fullName())
```

You may wonder why we didn't decorate the constructor. Alas, we can't use a method decorator on a constructor, because it isn't a method. It just *looks like one*. It's still a constructor function, and if we want to modify it, we have to either write a class decorator, or punt all the work of construction to a method, like this:

```
class Person {
  constructor (first, last) {
    this.rename(first, last);
  }

  fullName () {
    return this.firstName + " " + this.lastName;
  }

  @provided(nonEmptyStrings)
  rename (first, last) {
    this.firstName = first;
    this.lastName = last;
    return this;
  }
};
```

There are many variations on decorators that check preconditions for methods. For example, a decorator can be made that throws an exception if the preconditions fail rather than silently skipping the method invocation.

We can use these patterns in many ways. JavaScript is very flexible!

Method Advice

We've [previously](#) looked at method decorators like this:

```
const once = (fn) => {
  let invocations = new WeakSet();

  return function (...args) {
    if (invocations.has(this)) return;
    invocations.add(this);
    return fn.apply(this, args);
  }
}

const logician = new Person()
  .setName('Raymond', 'Smullyan');

logician.setName('Haskell', 'Curry');

const musician = new Person()
  .setName('Miles', 'Davis');

logician.fullName()
//=> Raymond Smullyan

musician.fullName()
//=> Miles Davis
```

We also saw that if our tooling supports ES.later¹⁶⁶ decorators, we can write:

¹⁶⁶By "ES.later," we mean some future version of ECMAScript that is likely to be approved eventually, but for the moment exists only in transpilers like [Babel](#). Obviously, using any ES.later feature in production is a complex decision requiring many more considerations than can be enumerated in a book.

```
const wrapWith = (decorator) =>
  function (target, name, descriptor) {
    descriptor.value = decorator(descriptor.value);
  }

function fluent (method) {
  return function (...args) {
    method.apply(this, args);
    return this;
  }
}

class Person {

  @wrapWith(fluent)
  setName (first, last) {
    this.firstName = first;
    this.lastName = last;
  }

  fullName () {
    return this.firstName + " " + this.lastName;
  }
};

}
```

The `wrapWith` function takes an ordinary method decorator and turns it into an ES.later method decorator.

what question do method decorators answer?

ES.later method decorators put the decorations right next to the method body. This makes it easy to answer the question “What is the precise behaviour of this method?”

But sometimes, this is not what you want. Consider a responsibility like authentication. Let’s imagine that we validate permissions in our model classes. We might write something like this:

```
const wrapWith = (decorator) =>
  function (target, name, descriptor) {
    descriptor.value = decorator(descriptor.value);
  }

const mustBeMe = (method) =>
  function (...args) {
    if (currentUser() && currentUser().person().equals(this))
      return method.apply(this, args);
    else throw new PermissionsException("Must be me!");
  }

class Person {

  @wrapWith(mustBeMe)
  setName (first, last) {
    this.firstName = first;
    this.lastName = last;
  }

  fullName () {
    return this.firstName + " " + this.lastName;
  }

  @wrapWith(mustBeMe)
  setAge (age) {
    this.age = age;
  }

  @wrapWith(mustBeMe)
  age () {
    return this.age;
  }
};

};
```

(Obviously real permissions systems involve roles and all sorts of other important things.)

Now we can look at `setName` and see that users can only set their own name, likewise if we look at `setAge`, we see that users can only set their own age.

In a tiny toy example the next question is easy to answer: *What methods can only be invoked by the person themselves?* We see at a glance that the answer is `setName`, `setAge`, and `age`.

But as classes grow, this becomes more difficult to answer. This especially becomes difficult if we decompose classes using mixins. For example, what if `setAge` and `age` come from a [class mixin](#):

```
const Person = HasAge(class {
  @wrapWith(mustBeMe)
  setName (first, last) {
    this.firstName = first;
    this.lastName = last;
  }

  fullName () {
    return this.firstName + " " + this.lastName;
  }
});
```

Are the methods provided by `HasAge` wrapped with `mustBeMe`? Quite possibly not, because the mixin is responsible for defining the behaviour. It's up to the model class to decide the permissions required. But how would you know if they were?

Method decorators make it easy to answer the question “what is the behaviour of this method?” But they don't make it easy to answer the question “what methods share this behaviour?”

That question matters, because when decomposing responsibilities, we often decide that a *cross-cutting* responsibility like permissions should be distinct from an implementation responsibility like storing a name.

cross-cutting method decorators

There is another way to decorate methods: We can decorate multiple methods in a single declaration. This is called providing *method advice*.

In JavaScript, we can implement method advice by decorating the entire class. We already have a combinator for making class mixins, it's a function that takes a class as an argument and returns the same or different class. We can use the same technique to write a class decorator that decorates one or more methods of the class being passed in. (We'll use ES.later syntax, but it works just as well with functional syntax):

```

const aroundAll = (behaviour, ...methodNames) =>
  (clazz) => {
    for (let methodName of methodNames)
      Object.defineProperty(clazz.prototype, property, {
        value: behaviour(clazz.prototype[methodName]),
        writable: true
      });
    return clazz;
  }

@HasAge
@aroundAll(mustBeMe, 'setName', 'setAge', 'age')
class Person {

  setName (first, last) {
    this.firstName = first;
    this.lastName = last;
  }

  fullName () {
    return this.firstName + " " + this.lastName;
  }

};

```

Now when you look at `setName`, you don't see what permissions apply. However, when we look at `@aroundAll(mustBeMe, 'setName', 'setAge', 'age')`, we see that we're wrapping `setName`, `setAge` and `age` with `mustBeMe`.

This focuses the responsibility for permissions in one place. Of course, we could make things simpler. For one thing, some actions are only performed *before* a method, and some only *after* a method. We can make class decorators that work just like our `before` and `after` method decorators:

```

const beforeAll = (behaviour, ...methodNames) =>
  (clazz) => {
    for (let methodName of methodNames) {
      const method = clazz.prototype[methodName];

      Object.defineProperty(clazz.prototype, property, {
        value: function (...args) {
          behaviour.apply(this, args);
          return method.apply(this, args);
        },
      });
    }
  }

```

```

        writable: true
    });
}
return clazz;
}

const afterAll = (behaviour, ...methodNames) =>
(clazz) => {
for (let methodName of methodNames) {
const method = clazz.prototype[methodName];

Object.defineProperty(clazz.prototype, property, {
value: function (...args) {
const returnValue = method.apply(this, args);

behaviour.apply(this, args);
return returnValue;
},
writable: true
});
}
return clazz;
}

```

Precondition checks like `mustBeMe` are good candidates for `beforeAll`. Here's `mustBeLoggedIn` and `mustBeMe` set up to use `beforeAll`. They're far simpler since `beforeAll` handles the wrapping:

```

const mustBeLoggedIn = () => {
if (currentUser() == null)
throw new PermissionsException("Must be logged in!");
}

const mustBeMe = () => {
if (currentUser() == null || !currentUser().person().equals(this))
throw new PermissionsException("Must be me!");
}

@HasAge
@beforeAll(mustBeMe, 'setName', 'setAge', 'age')
@beforeAll(mustBeLoggedIn, 'fullName')
class Person {

```

```

setName (first, last) {
  this.firstName = first;
  this.lastName = last;
}

fullName () {
  return this.firstName + " " + this.lastName;
}

};

```

This style of moving the responsibility for decorating methods to a single declaration will appear familiar to Ruby on Rails developers. As you can see, it does not require “deep magic” or complex libraries, it is a pattern that can be written out in just a few lines of code.

Mind you, there’s always room for polish and gold plate. We could enhance `beforeAll`, `afterAll`, and `aroundAll` to include conveniences like regular expressions to match method names, or special declarations like `except:` or `only:` if we so desired.

Although decorating methods in bulk has appeared in other languages and paradigms, it’s not something special and alien to JavaScript, it’s really the same pattern we see over and over again: Programming by composing small and single-responsibility entities, and using functions to transform and combine the entities into their final form.

a word about es6

If we don’t want to use ES.later decorators, we can use the exact same decorators as *ordinary functions*, like this:

```

const mustBeLoggedIn = () => {
  if (currentUser() == null)
    throw new PermissionsException("Must be logged in!");
}

const mustBeMe = () => {
  if (currentUser() == null || !currentUser().person().equals(this))
    throw new PermissionsException("Must be me!");
}

const Person =
  HasAge(
    beforeAll(mustBeMe, 'setName', 'setAge', 'age')(
      beforeAll(mustBeLoggedIn, 'fullName')(


```

```
class {
    setName (first, last) {
        this.firstName = first;
        this.lastName = last;
    }

    fullName () {
        return this.firstName + " " + this.lastName;
    }
}

);

);

);
```

Composition could also help:

```
const mustBeLoggedIn = () => {
  if (currentUser() == null)
    throw new PermissionsException("Must be logged in!");
}

const mustBeMe = () => {
  if (currentUser() == null || !currentUser().person().equals(this))
    throw new PermissionsException("Must be me!");
}

const Person = compose(
  HasAge,
  beforeAll(mustBeMe, 'setName', 'setAge', 'age'),
  beforeAll(mustBeLoggedIn, 'fullName'),
)(class {
  setName (first, last) {
    this.firstName = first;
    this.lastName = last;
  }

  fullName () {
    return this.firstName + " " + this.lastName;
  }
});
```


Closing Time at the Coffeeshop: Final Remarks



The Future of Coffee is Black

We began this book with the most basic of basic ideas in programming: What is a value? What is a reference to a value? What is a function? What is applying or invoking a function with values?

We then looked at one of the “big ideas” that JavaScript shares with other powerful languages: The idea that functions are values, and thus that you can invoke a function with another function as an argument, and you can return a function from a function.

This led directly to exploring the idea of composing functions: Creating new functions by putting together functions that represent smaller pieces of behaviour. The idea of function decorators emerges naturally from this approach.

From there we went on to explore objects and methods, but underlying our exploration was the constant rediscovery that we can program with objects using the same approach: Composing behaviour out of smaller pieces of behaviour, such as composing object behaviour using delegation.

javascript beyond es6/ecmascript 2015

When this edition was written, ECMAScript 2015 had been standardized, and almost all of its features were available via polyfills and transpilation tools like Babel. Some post-2015 features, like method and class decorators, had not yet been scheduled for inclusion in the language, but were available in transpilers.

Others, such as fully private properties, mixins, and traits, have been discussed and/or proposed. By the time you are reading this, they might be available experimentally, formally approved, or even widely available.

You may have noticed that as the book progressed, it delved into implementing programming language ideas like method decorators, mixins, and traits. Some books might implement a to-do list, or a content management system, or a MMRPG to provide an opportunity to write example code. This book chooses to explain JavaScript by implementing ideas that hopefully will become standard features by the time you read this.

Many other computer science textbooks do the same thing: They explain how to make a “toy” operating system, or a compiler, or how to make a Lisp in Lisp. And as you have learned, JavaScript is a language strong enough to implement many of its ideas in itself.

By implementing simple versions of features like decorators, mixins, and traits, we examined how to program in a lightweight fashion, and we also gained a deeper understanding of the semantics of functions, methods, classes, delegation, and behaviour.

That is valuable whether we use those features in production or not. And that is valuable whether those features are added to JavaScript, or not.



Espresso, Empty

the lightweight way

When creating a new abstraction, (for example, [traits](#)), there are two ways to do it: The heavyweight way, and the lightweight way.

The lightweight way, as explained throughout this book, attempts to be as “JavaScript-y” as possible. For example, using functions for protocols and composing them. With the lightweight way, everything is still just a function, or just an object, or just a class with just a prototype. Lightweight code interoperates 100% with code from other libraries. Lightweight approaches can be incrementally added to an existing code base, refactoring a bit here and a bit there.

The heavyweight way would [greenspun¹⁶⁷](#) a special class hierarchy with support for traits baked in. The heavyweight way would produce “classes” that don’t easily interoperate with other libraries or code, so you can’t incrementally make changes: You have to “boil the ocean” and commit 100% to the new approach. Heavyweight approaches often demand new kinds of tooling in the build pipeline.

When we do things the lightweight way, we make very small bets on their benefits. It’s easy to change our mind and abandon the approach in favour of something else. because we make small

¹⁶⁷https://en.wikipedia.org/wiki/Greenspun%27s_tenth_rule

bets along the way, we collect on the small benefits continuously: We don't have to kick off a massive rewrite of our code base to start using lightweight traits, for example. We just start using them as little or as much as we like, and immediately start benefiting from them.

"A language that doesn't affect the way you think about programming isn't worth learning."—Alan J. Perlis

Every tool affects the way we think about programming. But heavyweight tools force us to think about the heavyweight tooling. That thinking isn't always portable to another tool or another code base.

Whereas lightweight tools are simple things, composed together in simple ways. If we move to a different code base or tool, we can take our experience with the simple things along. With lightweight traits, for example, we are not teaching ourselves how to "program with traits," we're teaching ourselves how to "decompose behaviour," how to "compose functions" and how to "write functions that decorate entities."

These are all fundamental ideas that apply everywhere, even if we don't end up applying them to build a feature like traits. Lightweight thinking is portable and future-proof.



The End

The Golden Crema: Appendices and Afterwards



La Marzocco

How to run the examples

At the time this book was written, ECMAScript 2015 was not yet widely available. All of the examples in this book were tested using either [Google Traceur Compiler¹⁶⁸](#), [Babel¹⁶⁹](#), or both. Traceur and Babel are both *transpilers*, they work by parsing ECMAScript 2015 code, then emitting valid ECMAScript-5 code that produces the same semantics.

For example, this ECMAScript 2015 code:

```
const before = (decoration) =>
  (method) =>
    function () {
      decoration.apply(this, arguments);
      return method.apply(this, arguments)
    };
  
```

Is translated into this ECMAScript-5 code:

```
"use strict"

var before = function (decoration) {
  return function (method) {
    return function () {
      decoration.apply(this, arguments);
      return method.apply(this, arguments);
    };
  };
};
```

The screenshot shows the Babel 'try it out' interface. On the left, under the 'Experimental' tab, the ES6 code is pasted:

```
1 var before =
2   (decoration) =>
3     (method) =>
4       function (...args) {
5         decoration.apply(this, args);
6         return method.apply(this, args)
7       };
  
```

A blue box highlights the text "EcmaScript-6 Code".

On the right, under the 'Evaluate' tab, the resulting ES5 code is shown:

```
1 "use strict";
2
3 var before = function (decoration) {
4   return function (method) {
5     return function () {
6       for (var _len = arguments.length, args = Array(_len), _i = 0; _i < _len; _i++) {
7         args[_key] = arguments[_key];
8       }
9       decoration.apply(this, args);
10      return method.apply(this, args);
11    };
12  };
13};
14
```

A blue box highlights the text "EcmaScript-5 Equivalent".

The Babel “try it out” page

If we make it even more idiomatic, we could write:

¹⁶⁸<https://github.com/google/traceur-compiler>

¹⁶⁹<http://babeljs.io/>

```
const before = (decoration) =>
  (method) =>
    function (...args) {
      decoration.apply(this, args);
      return method.apply(this, args)
    };
  
```

And it would be “transpiled” into:

```
var before = function (decoration) {
  return function (method) {
    return function () {
      for (let _len = arguments.length, args = Array(_len), _key = 0; _key < _le\
n; _key++) {
        args[_key] = arguments[_key];
      }

      decoration.apply(this, args);
      return method.apply(this, args);
    };
  };
};
```

Both tools offer an online area where you can type ECMAScript code into a web browser and see the ECMAScript-5 equivalent, and you can run the code as well. To see the result of your expressions, you may have to use the console in your web browser.

So instead of just writing:

```
((() => 2 + 2))()
```

And having 4 displayed, you’d need to write:

```
console.log(
  ((() => 2 + 2))()
)
```

And 4 would appear in your browser’s development console.

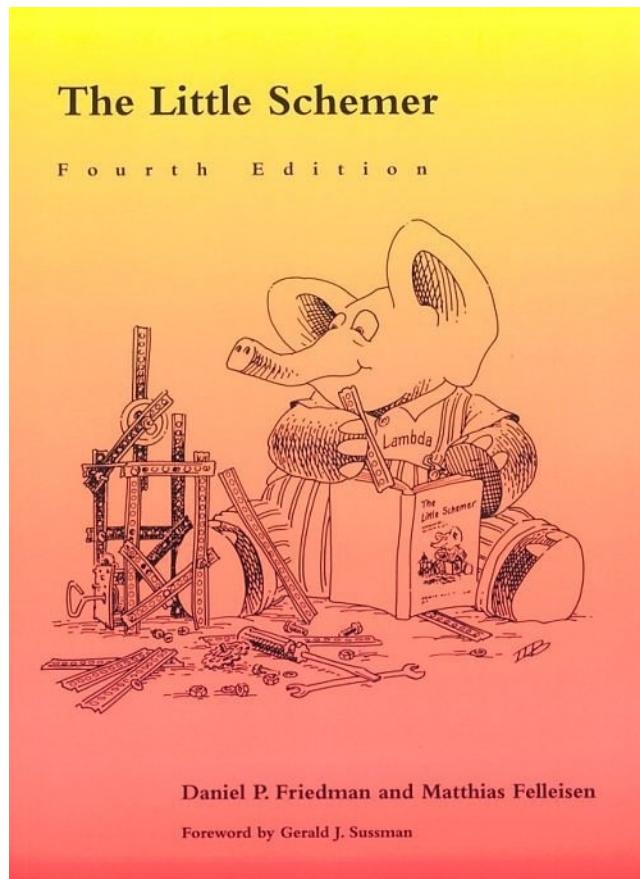
You can also install the transpilers on your development system and use them with [Node¹⁷⁰](#) on [the command line¹⁷¹](#). The care and feeding of node and npm are beyond the scope of this book, but both tools offer clear instructions for those who have already installed node.

¹⁷⁰<http://nodejs.org/>

¹⁷¹<https://en.wikipedia.org/wiki/REPL>

Thanks!

Daniel Friedman and Matthias Felleisen

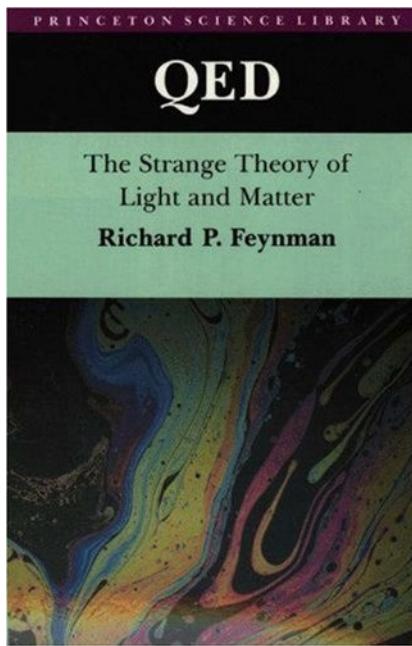


The Little Schemer

JavaScript Allongé was inspired by [The Little Schemer¹⁷²](#) by Daniel Friedman and Matthias Felleisen. But where *The Little Schemer*'s primary focus is recursion, *JavaScript Allongé*'s primary focus is functions as first-class values.

¹⁷²<http://www.amazon.com/0262560992?tag=raganwald001-20>

Richard Feynman



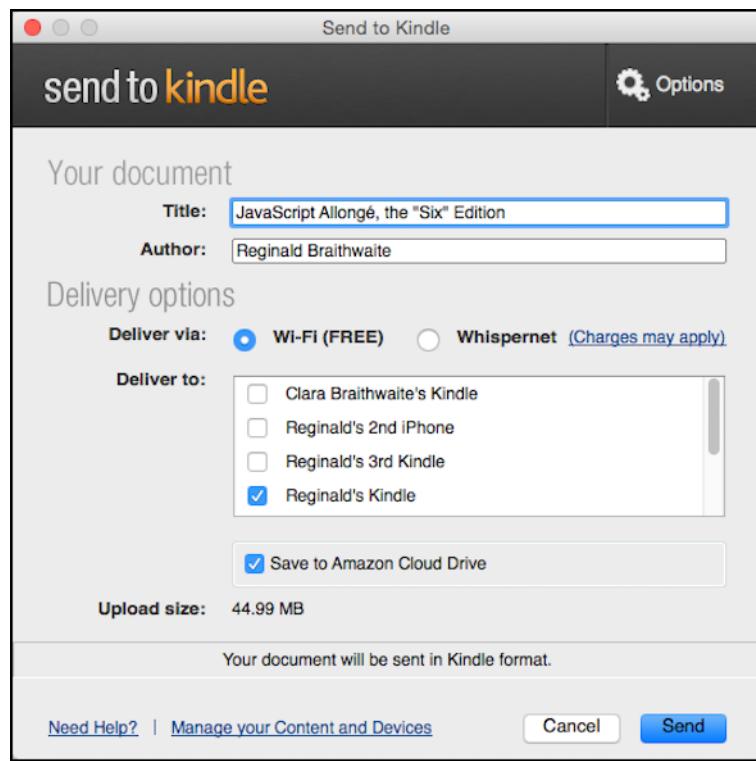
QED: The Strange Theory of Light and Matter

Richard Feynman's [QED¹⁷³](#) was another inspiration: A book that explains Quantum Electrodynamics and the “Sum of the Histories” methodology using the simple expedient of explaining how light reflects off a mirror, and showing how most of the things we think are happening—such as light travelling on a straight line, the angle of reflection equalling the angle of refraction, or that a beam of light only interacts with a small portion of the mirror, or that it reflects off a plane—are all wrong. And everything is explained in simple, concise terms that build upon each other logically.

¹⁷³<http://www.amazon.com/0691125759?tag=raganwald001-20>

Reading JavaScript Allongé on Kindle

JavaScript Allongé has over 400 pages and many photographs. For this reason, the .mobi version of the book is too big to be sent to your Kindle via email, and that is the feature that LeanPub uses when you purchase the book.



Send to Kindle

So, if you wish to read JavaScript Allongé on your Kindle:

- Download it to your Windows or OS X device (a/k/a “PC” or “Macintosh”).
- Use [Send to Kindle for PC¹⁷⁴](http://www.amazon.com/gp/sendtokindle/pc) or [Send to Kindle for Mac¹⁷⁵](http://www.amazon.com/gp/sendtokindle/mac) to send it to the Kindle.
- From time to time while editing, an uncompressed image sneaks into the manuscript. When this happens, the .mobi may exceed the 50MB limit for the Send to Kindle desktop application. If this happens, please attach your Kindle to your computer with a USB cable and synchronize directly.

Thank you!

¹⁷⁴<http://www.amazon.com/gp/sendtokindle/pc>

¹⁷⁵<http://www.amazon.com/gp/sendtokindle/mac>

Copyright Notice

The original words in this book are (c) 2012-2015, Reginald Braithwaite. All rights reserved.

images

- The picture of the author is (c) 2008, Joseph Hurtado¹⁷⁶, All Rights Reserved.
- Cover image¹⁷⁷ (c) 2010, avlxyz. Some rights reserved¹⁷⁸.
- Double ristretto menu¹⁷⁹ (c) 2010, Michael Allen Smith. Some rights reserved¹⁸⁰.
- Short espresso shot in a white cup with blunt handle¹⁸¹ (c) 2007, EVERYDAYLIFEMODERN. Some rights reserved¹⁸².
- Espresso shot in a caffe molinari cup¹⁸³ (c) 2007, EVERYDAYLIFEMODERN. Some rights reserved¹⁸⁴.
- Beans in a Bag¹⁸⁵ (c) 2008, Stirling Noyes. Some Rights Reserved¹⁸⁶.
- Free Samples¹⁸⁷ (c) 2011, Myrtle Bech Digitel. Some Rights Reserved¹⁸⁸.
- Free Coffees¹⁸⁹ image (c) 2010, Michael Francis McCarthy. Some Rights Reserved¹⁹⁰.
- La Marzocco¹⁹¹ (c) 2009, Michael Allen Smith. Some rights reserved¹⁹².
- Cafe Diplomatico¹⁹³ (c) 2011, Missi. Some rights reserved¹⁹⁴.
- Sugar Service¹⁹⁵ (c) 2008 Tiago Fernandes. Some rights reserved¹⁹⁶.
- Biscotti on a Rack¹⁹⁷ (c) 2010 Kirsten Loza. Some rights reserved¹⁹⁸.
- Coffee Spoons¹⁹⁹ (c) 2010 Jenny Downing. Some rights reserved²⁰⁰.

¹⁷⁶<http://www.flickr.com/photos/trumpetca/>

¹⁷⁷<http://www.flickr.com/photos/avlxyz/4907262046>

¹⁷⁸<http://creativecommons.org/licenses/by-sa/2.0/deed.en>

¹⁷⁹<http://www.flickr.com/photos/digitalcolony/5054568279/>

¹⁸⁰<http://creativecommons.org/licenses/by-sa/2.0/deed.en>

¹⁸¹<http://www.flickr.com/photos/everydaylifemodern/1353570874/>

¹⁸²<http://creativecommons.org/licenses/by-nd/2.0/deed.en>

¹⁸³<http://www.flickr.com/photos/everydaylifemodern/434299813/>

¹⁸⁴<http://creativecommons.org/licenses/by-nd/2.0/deed.en>

¹⁸⁵http://www.flickr.com/photos/the_rev/2295096211/

¹⁸⁶<http://creativecommons.org/licenses/by/2.0/deed.en>

¹⁸⁷<http://www.flickr.com/photos/thedigitelmyr/6199419022/>

¹⁸⁸<http://creativecommons.org/licenses/by-sa/2.0/deed.en>

¹⁸⁹<http://www.flickr.com/photos/sagamiono/4391542823/>

¹⁹⁰<http://creativecommons.org/licenses/by-sa/2.0/deed.en>

¹⁹¹<http://www.flickr.com/photos/digitalcolony/3924227011/>

¹⁹²<http://creativecommons.org/licenses/by-sa/2.0/deed.en>

¹⁹³<http://www.flickr.com/photos/15481483@N06/6231443466/>

¹⁹⁴<http://creativecommons.org/licenses/by-sa/2.0/deed.en>

¹⁹⁵<http://www.flickr.com/photos/tjgfernandes/2785677276/>

¹⁹⁶<http://creativecommons.org/licenses/by/2.0/deed.en>

¹⁹⁷<http://www.flickr.com/photos/kirstenloza/4805716699/>

¹⁹⁸<http://creativecommons.org/licenses/by/2.0/deed.en>

¹⁹⁹<http://www.flickr.com/photos/jenny-pics/5053954146/>

²⁰⁰<http://creativecommons.org/licenses/by/2.0/deed.en>

- Drawing a Doppio²⁰¹ (c) 2008 Osman Bas. Some rights reserved²⁰².
- Cupping Coffees²⁰³ (c) 2011 Dennis Tang. Some rights reserved²⁰⁴.
- Three Coffee Roasters²⁰⁵ (c) 2009 Michael Allen Smith. Some rights reserved²⁰⁶.
- Blue Diedrich Roaster²⁰⁷ (c) 2010 Michael Allen Smith. Some rights reserved²⁰⁸.
- Red Diedrich Roaster²⁰⁹ (c) 2009 Richard Masoner. Some rights reserved²¹⁰.
- Roaster with Tree Leaves²¹¹ (c) 2007 ting. Some rights reserved²¹².
- Half Drunk²¹³ (c) 2010 Nicholas Lundgaard. Some rights reserved²¹⁴.
- Anticipation²¹⁵ (c) 2012 Paul McCoubrie. Some rights reserved²¹⁶.
- Ooh!²¹⁷ (c) 2012 Michael Coghlan. Some rights reserved²¹⁸.
- Intestines of an Espresso Machine²¹⁹ (c) 2011 Angie Chung. Some rights reserved²²⁰.
- Bezzera Espresso Machine²²¹ (c) 2011 Andrew Nash. Some rights reserved²²². *Beans Ripening on a Branch²²³ (c) 2008 John Pavelka. Some rights reserved²²⁴.
- Cafe Macchiato on Gazotta Della Sport²²⁵ (c) 2008 Jon Shave. Some rights reserved²²⁶.
- Jars of Coffee Beans²²⁷ (c) 2012 Memphis CVB. Some rights reserved²²⁸.
- Types of Coffee Drinks²²⁹ (c) 2012 Michael Coghlan. Some rights reserved²³⁰.
- Coffee Trees²³¹ (c) 2011 Dave Townsend. Some rights reserved²³².

²⁰¹<http://www.flickr.com/photos/33388953@N04/4017985434/>

²⁰²<http://creativecommons.org/licenses/by/2.0/deed.en>

²⁰³<http://www.flickr.com/photos/tangysd/5953453156/>

²⁰⁴<http://creativecommons.org/licenses/by-sa/2.0/deed.en>

²⁰⁵<http://www.flickr.com/photos/digitalcolony/4000837035/>

²⁰⁶<http://creativecommons.org/licenses/by-sa/2.0/deed.en>

²⁰⁷<http://www.flickr.com/photos/digitalcolony/4309812256/>

²⁰⁸<http://creativecommons.org/licenses/by-sa/2.0/deed.en>

²⁰⁹<http://www.flickr.com/photos/bike/3237859728/>

²¹⁰<http://creativecommons.org/licenses/by-sa/2.0/deed.en>

²¹¹<http://www.flickr.com/photos/lacerabbit/2102801319/>

²¹²<http://creativecommons.org/licenses/by-nd/2.0/deed.en>

²¹³<http://www.flickr.com/photos/nalundgaard/4785922266/>

²¹⁴<http://creativecommons.org/licenses/by-sa/2.0/deed.en>

²¹⁵<http://www.flickr.com/photos/paulmccoubrie/6828131856/>

²¹⁶<http://creativecommons.org/licenses/by-nd/2.0/deed.en>

²¹⁷<http://www.flickr.com/photos/mikecogh/7676649034/>

²¹⁸<http://creativecommons.org/licenses/by-sa/2.0/deed.en>

²¹⁹<http://www.flickr.com/photos/yellowskyphotography/5641003165/>

²²⁰<http://creativecommons.org/licenses/by-sa/2.0/deed.en>

²²¹<http://www.flickr.com/photos/andynash/6204253236/>

²²²<http://creativecommons.org/licenses/by-sa/2.0/deed.en>

²²³<http://www.flickr.com/photos/28705377@N04/5306009552/>

²²⁴<http://creativecommons.org/licenses/by/2.0/deed.en>

²²⁵<http://www.flickr.com/photos/shavejonathan/2343081208/>

²²⁶<http://creativecommons.org/licenses/by/2.0/deed.en>

²²⁷<http://www.flickr.com/photos/ilovememphis/7103931235/>

²²⁸<http://creativecommons.org/licenses/by-nd/2.0/deed.en>

²²⁹<http://www.flickr.com/photos/mikecogh/7561440544/>

²³⁰<http://creativecommons.org/licenses/by-sa/2.0/deed.en>

²³¹<http://www.flickr.com/photos/dtownsend/6171015997/>

²³²<http://creativecommons.org/licenses/by-sa/2.0/deed.en>

- [Cafe do Brasil²³³](#) (c) 2003 Temporalata. [Some rights reserved²³⁴](#).
- [Brown Cups²³⁵](#) (c) 2007 Michael Allen Smith. [Some rights reserved²³⁶](#).
- [Mirage²³⁷](#) (c) 2010 Mira Helder. [Some rights reserved²³⁸](#).
- [Coffee Van with Bullet Holes²³⁹](#) (c) 2006 Jon Crel. [Some rights reserved²⁴⁰](#).
- [Disassembled Elektra²⁴¹](#) (c) 2009 Nicholas Lundgaard. [Some rights reserved²⁴²](#).
- [Nederland Buffalo Bills Coffee Shop²⁴³](#) (c) 2009 Charlie Stinchcomb. [Some rights reserved²⁴⁴](#).
- [For the love of coffee²⁴⁵](#) (c) 2007 Lotzman Katzman. [Some rights reserved²⁴⁶](#).
- [Saltspring Processing Facility Pictures²⁴⁷](#) (c) 2011 Kris Krug. [Some rights reserved²⁴⁸](#).
- [Coffee and Mathematics²⁴⁹](#) (c) 2007 [Some rights reserved²⁵⁰](#).
- [Coffee and a Book²⁵¹](#) (c) 2009 [Some rights reserved²⁵²](#).
- [Stacked Coffee Cups²⁵³](#) (c) 2010 Sankarshan Sen. [Some rights reserved²⁵⁴](#).
- [Coffee Cow²⁵⁵](#) (c) 2012 Candy Schwartz²⁵⁶ [Some rights reserved²⁵⁷](#).
- [CERN Coffee²⁵⁸](#) (c) 2005 Karoly Lorentey²⁵⁹ [Some rights reserved²⁶⁰](#).
- [Coffee Labels²⁶¹](#) (c) 2011 Kris Krüg [Some rights reserved²⁶²](#).
- [banco do café²⁶³](#) (c) 2008 Fernando Mafra [Some rights reserved²⁶⁴](#).

²³³<http://www.flickr.com/photos/93425126@N00/313053257/>

²³⁴<http://creativecommons.org/licenses/by-sa/2.0/deed.en>

²³⁵<http://www.flickr.com/photos/digitalcolony/2833809436/>

²³⁶<http://creativecommons.org/licenses/by-sa/2.0/deed.en>

²³⁷<http://www.flickr.com/photos/citizenhelder/5006498068/>

²³⁸<http://creativecommons.org/licenses/by/2.0/deed.en>

²³⁹<http://www.flickr.com/photos/joncrel/237026246/>

²⁴⁰<http://creativecommons.org/licenses/by-nd/2.0/deed.en>

²⁴¹<http://www.flickr.com/photos/nalundgaard/3163852170/>

²⁴²<http://creativecommons.org/licenses/by-sa/2.0/deed.en>

²⁴³<http://www.flickr.com/photos/47000103@N05/6525288841/>

²⁴⁴<http://creativecommons.org/licenses/by-sa/2.0/deed.en>

²⁴⁵<http://www.flickr.com/photos/lotzman/978418891/>

²⁴⁶<http://creativecommons.org/licenses/by/2.0/deed.en>

²⁴⁷<http://www.flickr.com/photos/kk/sets/72157626168201654/with/5484839102/>

²⁴⁸<http://creativecommons.org/licenses/by-sa/2.0/deed.en>

²⁴⁹<https://www.flickr.com/photos/kellan/434503323>

²⁵⁰<http://creativecommons.org/licenses/by/2.0/deed.en>

²⁵¹<https://www.flickr.com/photos/whitneyinchicago/3835218626>

²⁵²<http://creativecommons.org/licenses/by/2.0/deed.en>

²⁵³<https://www.flickr.com/photos/sankarshan/5165312159>

²⁵⁴<http://creativecommons.org/licenses/by-sa/2.0/deed.en>

²⁵⁵<https://www.flickr.com/photos/candy-s/7619358284>

²⁵⁶<https://www.flickr.com/photos/candy-s/>

²⁵⁷<http://creativecommons.org/licenses/by/2.0/deed.en>

²⁵⁸<https://www.flickr.com/photos/lorentey/22193876>

²⁵⁹<https://www.flickr.com/photos/lorentey/>

²⁶⁰<http://creativecommons.org/licenses/by/2.0/deed.en>

²⁶¹<https://www.flickr.com/photos/kk/5484876862>

²⁶²<http://creativecommons.org/licenses/by-sa/2.0/deed.en>

²⁶³https://www.flickr.com/photos/f_mafra/2956649121

²⁶⁴<http://creativecommons.org/licenses/by-sa/2.0/deed.en>

- coffee pots²⁶⁵ (c) 2009 Jonas Forth [Some rights reserved²⁶⁶](#).
- 5 Barrel Roaster²⁶⁷ (c) 2013 David Lytle [Some rights reserved²⁶⁸](#).
- Pantone mugs²⁶⁹ (c) 2011 Joe Wolf [Some rights reserved²⁷⁰](#).
- Coffee and Chess²⁷¹ (c) 2013 Adam Tinworth [Some rights reserved²⁷²](#).
- Vac Pot Upper Chamber²⁷³ (c) 2007 Michael Allen Smith [Some rights reserved²⁷⁴](#).
- Decaf espresso²⁷⁵ (c) 2009 Aris Vrakas [Some rights reserved²⁷⁶](#).
- Con Panna²⁷⁷ (c) 2013 Vee Satayamas [Some rights reserved²⁷⁸](#).
- Tiny's Coffeehouse²⁷⁹ (c) 2004 Peter Merholz [Some rights reserved²⁸⁰](#).
- Thinking about programming²⁸¹ (c) 2011 Renaud Camus [Some rights reserved²⁸²](#).
- Biscotti og kaffe²⁸³ (c) 2008 [Some rights reserved²⁸⁴](#).
- Espresso, Empty²⁸⁵ (c) 2012 Till Westermayer [Some rights reserved²⁸⁶](#).
- The End²⁸⁷ (c) 2013 peddhapati [Some rights reserved²⁸⁸](#).
- The Future of Coffee is Black²⁸⁹ (c) 2013 mjaysplanet [Some rights reserved²⁹⁰](#).

²⁶⁵<https://www.flickr.com/photos/jforth/3360599750/>

²⁶⁶<http://creativecommons.org/licenses/by-nd/2.0/deed.en>

²⁶⁷<https://www.flickr.com/photos/dlytle/8720139854>

²⁶⁸<http://creativecommons.org/licenses/by/2.0/deed.en>

²⁶⁹<https://www.flickr.com/photos/joebehr/5504285781>

²⁷⁰<http://creativecommons.org/licenses/by-nd/2.0/deed.en>

²⁷¹<https://www.flickr.com/photos/adders/8372085101>

²⁷²<http://creativecommons.org/licenses/by-nd/2.0/deed.en>

²⁷³<https://www.flickr.com/photos/digitalcolony/2843767532>

²⁷⁴<http://creativecommons.org/licenses/by-sa/2.0/deed.en>

²⁷⁵<https://www.flickr.com/photos/arislvrakas/4217869291>

²⁷⁶<http://creativecommons.org/licenses/by/2.0/deed.en>

²⁷⁷<https://www.flickr.com/photos/vscript/8708520929>

²⁷⁸<http://creativecommons.org/licenses/by/2.0/deed.en>

²⁷⁹<https://www.flickr.com/photos/peterme/1271652>

²⁸⁰<http://creativecommons.org/licenses/by-sa/2.0/deed.en>

²⁸¹<https://www.flickr.com/photos/renaud-camus/6165559492>

²⁸²<http://creativecommons.org/licenses/by/2.0/deed.en>

²⁸³<https://www.flickr.com/photos/cyclonebill/2606398721>

²⁸⁴<http://creativecommons.org/licenses/by-sa/2.0/deed.en>

²⁸⁵<https://www.flickr.com/photos/tillwe/8154272083>

²⁸⁶<http://creativecommons.org/licenses/by-sa/2.0/deed.en>

²⁸⁷<https://www.flickr.com/photos/peddhapati/11671457605>

²⁸⁸<http://creativecommons.org/licenses/by/2.0/deed.en>

²⁸⁹<https://www.flickr.com/photos/mjaysplanet/8416343475>

²⁹⁰<http://creativecommons.org/licenses/by-sa/2.0/deed.en>

About The Author

When he's not shipping JavaScript, Ruby, CoffeeScript and Java applications scaling out to millions of users, Reg "Raganwald" Braithwaite has authored [libraries²⁹¹](#) for JavaScript, CoffeeScript, and Ruby programming such as Allong.es, Method Combinators, Katy, JQuery Combinators, YouAreDaChef, andand, and others.

He writes about programming on "[Raganwald²⁹²](#)," as well as general-purpose ruminations on "[Braythwayt Dot Com²⁹³](#)".

contact

Twitter: [@raganwald²⁹⁴](#) Email: reg@braythwayt.com²⁹⁵



Reg "Raganwald" Braithwaite

²⁹¹<http://github.com/raganwald>

²⁹²<http://raganwald>

²⁹³<http://braythwayt.com>

²⁹⁴<https://twitter.com/raganwald>

²⁹⁵<mailto:reg@braythwayt.com>