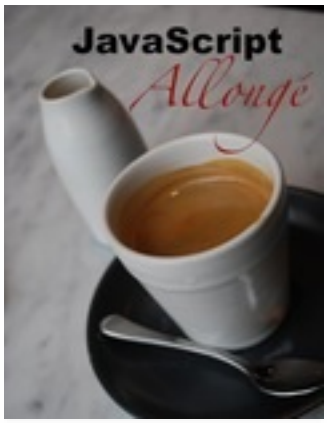
JavaScript

Allongé

JavaScript Allongé

A long pull of functions, combinators, & decorators

raganwald



$19.00 **$29.00**

MINIMUM PRICE SUGGESTED PRICE

Buy Ebook At Leanpub

JavaScript Allongé

Table of Contents

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."

Foreword by Michael Fogus

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.

– Fogus, fogus.me

Foreword by 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 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 Cafe Allongé, it's over too soon. Enjoy!

–Matthew Knox, mattknox.com

Why JavaScript Allongé?

JavaScript Allongé solves two important problems for the ambitious JavaScript programmer. First, JavaScript Allongé gives you the tools to deal with JavaScript bugs, hitches, edge cases, and other potential pitfalls.

There are plenty of good directions for how to write JavaScript programs. If you follow them without alteration or deviation, you will be satisfied. Unfortunately, software is a complex thing, full of interactions and side-effects. Two perfectly reasonable pieces of advice when taken separately may conflict with each other when taken together. An approach may seem sound at the outset of a project, but need to be revised when new requirements are discovered.

When you "leave the path" of the directions, you discover their limitations. In order to solve the problems that occur at the edges, in order to adapt and deal with changes, in order to refactor and rewrite as needed, you need to understand the underlying principles of the JavaScript programming language in detail.

You need to understand why the directions work so that you can understand how to modify them to work properly at or beyond their original limitations. That's where JavaScript Allongé comes in.

JavaScript Allongé is a book about programming with functions, because JavaScript is a programming language built on flexible and powerful functions. JavaScript Allongé begins at the beginning, with values and expressions, and builds from there to discuss types, identity, functions, closures, scopes, and many more subjects up to working with classes and instances. In each case, JavaScript Allongé takes care to explain exactly how things work so that when you encounter a problem, you'll know exactly what is happening and how to fix it.

Second, JavaScript Allongé 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, thus

avoiding problems before they happen.

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.

how the 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.

Following each chapter are a series of recipes designed to show the application of the chapters 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.

A Personal Word About The Recipes

As noted, JavaScript Allongé alternates between chapters describing the semantics of JavaScript's functions with chapters containing recipes for writing programs with functions. You can read the book in order or read the chapters explaining JavaScript first and return to the recipes later.

The recipes share a common theme: They hail from a style of programming inspired by the creation of small functions that compose with each other. Using these recipes, you'll learn when it's appropriate to write:

```
return mapWith(maybe(getWith('name')))(customerList);
```

Instead of:

```
return customerList.map(function (customer) {
  if (customer) {
    return customer.name
  }
});
```

As well as how it works and how to refactor it when you need. This style of programming is hardly the most common thing anyone does in JavaScript, so the argument can be made that more "practical" or "commonplace" recipes would be helpful. If you never read any other books about JavaScript, if you avoid blog posts and screen casts about

JavaScript, if you don't attend workshops or talks about JavaScript, then I agree that this is not One Book to Rule Them All.

But given that there are other resources out there, and that programmers are curious creatures with an unslakable thirst for personal growth, we choose to provide recipes that you are unlikely to find anywhere else in anything like this concentration. The recipes reinforce the lessons taught in the book about functions in JavaScript.

You'll find all of the recipes collected online at http://allong.es. They're free to share under the MIT license.

Reginald Braithwaite
reg@braythwayt.com
@raganwald

Legend

Some text in monospaced type like `this` in the text represents some code being discussed. Some monospaced code in its own lines also represents code being discussed:

```
this.async = do (async = undefined) ->

  async = (fn) ->
    (argv..., callback) ->
      callback(fn.apply(this, argv))
```

Sometimes it will contain some code for you to type in for yourself. When it does, the result of typing something in will often be shown using `//=>`, like this:

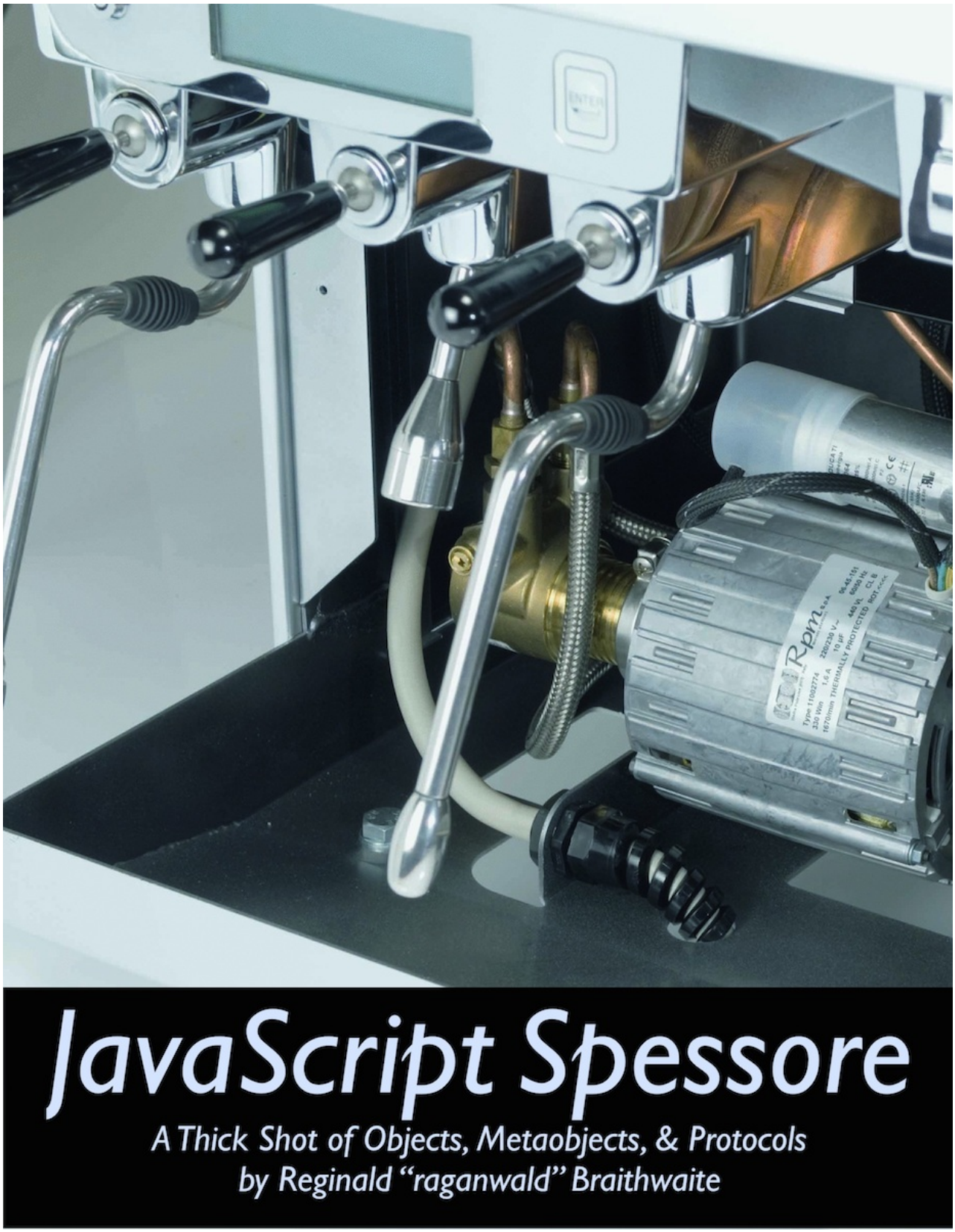
```
2 + 2
  //=> 4
```

A paragraph marked like this is a "key fact." It summarizes an idea without adding anything new.

A paragraph marked like this is a suggested exercise to be performed on your own.

A paragraph marked like this is an aside. It can be safely ignored. It contains whimsey and other doupleplusunserious logorrhea that will not be on the test.

JavaScript Spessore



JavaScript Spessore

FYI, the companion book to JavaScript Allongé is JavaScript Spessore:

Programming languages are (loosely) defined by their basic activity. In FORTRAN, we program with numbers. In C, we program with pointers. In ML, we program with types. And as will JavaScript Allongé explain, in JavaScript we program with functions.

Functions are very interesting building blocks for programs, because they compose: It's easy to build a programming style based on making many small things that can be combined and recombined to make bigger things.

This is the basis of the vaunted "Unix Philosophy:" Write small utilities and scripts that compose neatly. This is also the JavaScript philosophy: Make small things that can be combined and recombined to make bigger things.

Programming with objects can be done in this style, and JavaScript makes it particularly easy to combine and recombine small parts. Classes can be built from traits instead of from superclasses. Objects can delegate and forward behaviour to helpers and meta-objects. Adaptors can be written to change an object's interface without needing to create another class in a hierarchy.

JavaScript Spessore is a book that describes this approach to working with objects and metaobjects in JavaScript. It's exactly the same philosophy as you find in JavaScript Allongé, only it speaks to programming with objects instead of programming with functions.

JavaScript Spessore describes how to build JavaScript programs that scale in code, in time, and across a team, using the one technique that has passed the test of time: Objects and metaobjects that have a single responsibility, are decoupled from each other, and can be composed freely.

Once you've read JavaScript Allongé, JavaScript Spessore should be next.

Prelude: Values and Expressions

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 expression, and it returns a value, just as you express your wishes to a barista and receive a coffee in return.

values and 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:

```
1 42
```

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

The answer is, this is both an expression and a value.[1] 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:

```
1 42
2   //=> 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.[2] 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"
```

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:

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

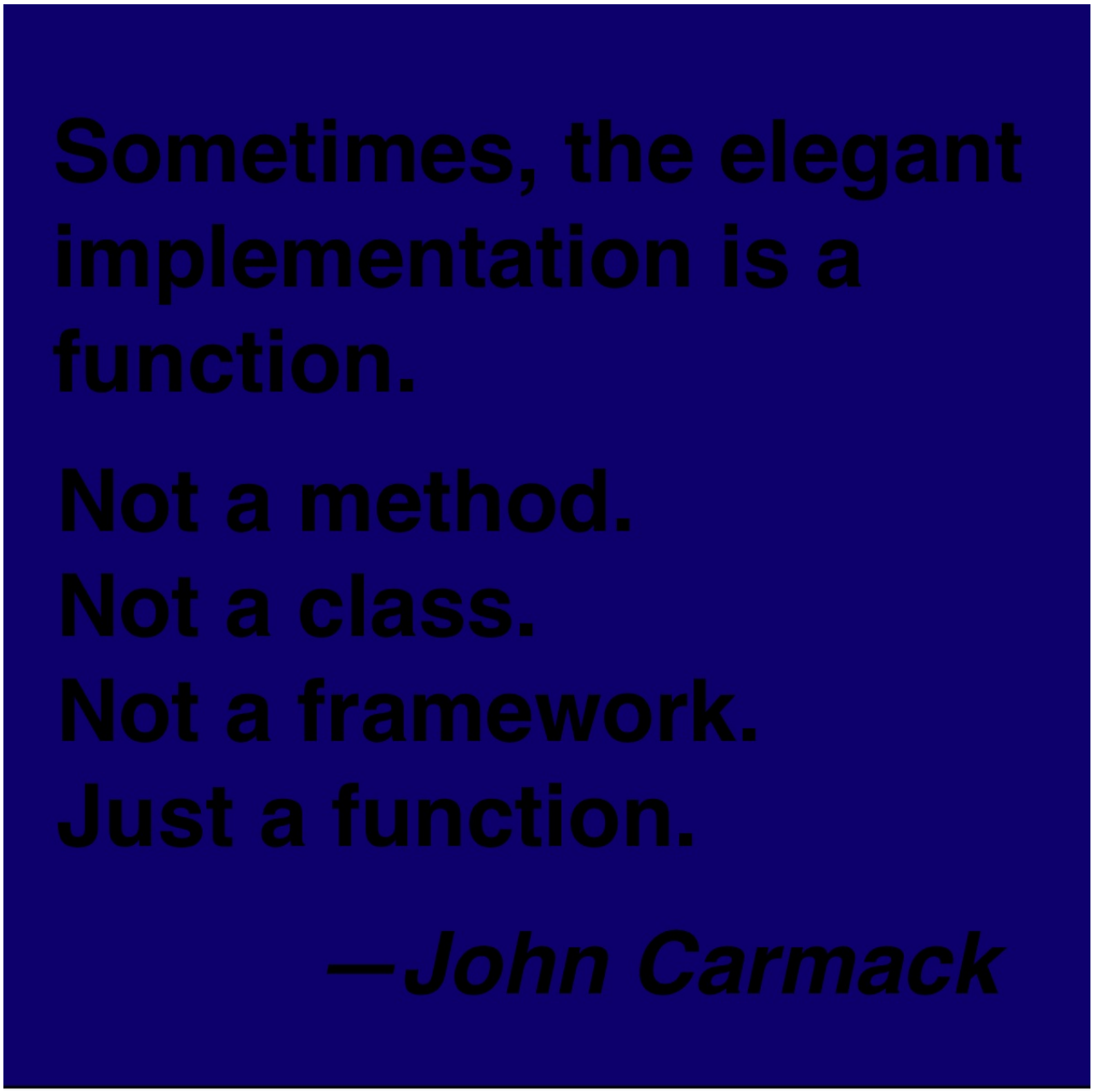
Notice that you are always generating arrays with the same contents. But are they identical the same way that every value of `42` is identical to every other value of `42`? Try these for yourself:

```
[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.

> Arrays look exceedingly simple, but this word "reference" is so laden with possibilities that there's an entire chapter devoted to discussing rebinding and references. Try typing this code out:
>
> ```
> var ouroboros = [];
> ouroboros[0] = ouroboros;
>   //=> [ [Circular] ]
> ```
>
> You've just created an ouroborian array, an array that contains itself.

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.



Technically, 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.↩

In 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.↩

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

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 very simplest possible function. In JavaScript, it looks like this:

```
1 function () {}
```

This is a function that is applied to no values and produces no value. How do we represent "no value" in JavaScript? We'll find out in a minute. First, let's verify that our function is a value:

```
1 (function () {})
2   //=> [Function]
```

What!? Why didn't it type back `function () {}` 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'll see the code you typed.

> 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 `function () {}`, 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 it. For reasons of appeasing the JavaScript parser, we'll enclose our functions in parentheses:

```
1 (function () {}) === (function () {})
2   //=> 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: `function () {}`, so let’s use it. We’ll put it in parentheses[1] to keep the parser happy, like we did above: `(function () {})`. Since we aren’t giving it any arguments, we’ll simply write `()` after the expression. So we write:

```
(function () {})()
  //=> 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
(function () {})() === (function () {})()
  //=> true
(function () {})() === 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 (function () {})(), 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?[2] 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.[3]

functions with no arguments and their bodies

Back to our function. We evaluated this:

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

Let’s recall that we were applying the function function () {} to no arguments (because there

was nothing inside of ()). So how do we know to expect undefined? That's easy:

When we define a function[4], we write the word function. We then put a (possibly empty) list of arguments, then we give the function a body that is enclosed in braces {...}. Function bodies are (possibly empty) lists of JavaScript statements separated by semicolons.

Something like: { statement^1; statement^2; statement^3; … ; statement^n }

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, the following are all valid JavaScript functions, and they all evaluate to undefined when applied:

```
1 (function () { 2 + 2 })
2
3 (function () { 1 + 1; 2 + 2 })
```

You can also separate statements with line breaks.[5] The convention is to use some form of consistent indenting:

```
 1 (function () {
 2   1 + 1;
 3   2 + 2
 4 })
 5
 6 (function () {
 7   (function () {
 8     (function () {
 9       (function () {
10       })
11     })
12   });
13   (function () {
14   })
15 })
```

That last one's a doozy, but since a function body can contain a statement, and a statement can be an expression, and a function is an expression…. You get the idea.

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

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

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

(function () { 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.

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:

```
function () {
  return (function () {})
}
```

That's a function! It's a function that when applied, evaluates to a function that when applied, evaluates to undefined.[6] Let's use a simpler terminology. Instead of saying "that when applied, evaluates to _____," we will say "gives _____." And instead of saying "gives undefined," we'll say "doesn't give anything."

So we have a function, that gives a function, that doesn't give anything. Likewise:

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

That's a function, that gives a function, that gives true:

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

```
5 })()()
6   //=> true
```

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.

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:

```
1 function (room) {}
```

This function has one argument, `room`, and no body. Here's a function with two arguments and no body:

```
1 function (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?

```
1 function (diameter) { return 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 `(function () {})()`. To apply a function with an argument (or arguments), we put the argument (or arguments) within the

parentheses, like this:

```
(function (diameter) { return diameter * 3.14159265 })(2)
  //=> 6.2831853
```

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

```
(function (room, board) { return 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:

```
function () {
  return function () {}
}

function () {
  return [ 1, 2, 3]
}

[1, [2, 3], 4]
```

```
10
11 function () {
12   return [
13     (function () { return 1}),
14     (function () { return 2}),
15     (function () { return 3})
16   ]
17 }
```

---

call by value

Like most contemporary programming languages, JavaScript uses the “call by value” evaluation strategy. 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).

So when you write:

```
1 (function (diameter) { return diameter * 3.14159265 })(1 + 1)
2   //=> 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`.[8]

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 `function (diameter) { return diameter * 3.14159265 }` to functions like:

```
1 function (x) { return (function (y) { return x }) }
```

> `function (x) { return (function (y) { return 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 `function (x) { return (function (y) { return 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 `function (x) ...`, is an argument. The `y` in `function (y) ...` is another argument. The second `x`, the one in `{ return 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 `function (x) { return (function (y) { return x }) }` or just plain `function (x) { return 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.

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:

```
1 (function (x) { return x })(2)
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, `function (x) { return 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 syntax 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 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 argument[9] you apply it to:

```
1 function (value) {
2   return (function (copy) {
3     return copy === value
4   })(value)
5 }
```

Closures and Scope

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

```
(function (x) {
  return function (y) {
    return 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:

```
(function (x) {
  return function (y) {
    return x
  }
})(1)
  //=> [Function]
```

The environment belonging to the function with signature `function (x) ...` becomes `{x: 1, ...}`, 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 `function (x) ...`'s body is:

```
  function (y) {
    return 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:

```
  (function (y) {
    return x
  })(2)
```

So we seem to get a new environment `{y: 2, ...}`. 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:

```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 `function (y) { return x }` is interesting. It contains a free variable, `x`.[10] 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 `function (y) { return 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:

```javascript
function () {}

function (x) {
  return x
}

function (x) {
  return function (y) {
    return 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. `function (y) ...` has a free variable, but the entire expression refers to `function (x) ...`, and it doesn't have a free variable: The only variable anywhere in its body is `x`, which is certainly bound within `function (x) ...`.

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 `function (x, y) { return x + y }`, we know exactly what it does with `(2, 2)`. But what about this closure: `function (y) { return x + y }`? We can't say what it will do with argument `(2)` without understanding the magic for evaluating the free variable `x`.

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 `(function (x) { return (function (y) { return x }) })(1)` is `{x: 1, ...}` and that the environment for `(function (y) { return x })(2)` is `{y: 2, ...}`? Let's fill in the blanks!

The environment for `(function (y) { return x })(2)` is actually `{y: 2, '..': {x: 1, ...}}`. `'..'` means something like "parent" or "enclosure" or "super-environment." It's `function (x) ...`'s environment, because the function `function (y) { return x }` is within `function (x) ...`'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 `(function (y) { return x })(2)` in the environment `{y: 2, '..': {x: 1, ...}}`. The variable `x` isn't in `function (y) ...`'s immediate environment, but it is in its parent's environment, so it evaluates to `1` and that's what `(function (y) { return x })(2)` returns even though it ended up ignoring its own argument.

> `function (x) { return x }` is called the I Combinator or Identity Function. `function (x) { return (function (y) { return 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, some–how shall I put this–are interesting if you use Ruby.

Functions can have grandparents too:

```
function (x) {
  return function (y) {
    return function (z) {
      return x + y + z
    }
  }
}
```

This function does much the same thing as:

```
function (x, y, z) {
  return 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 the second function. Calling a curried function with only some of its arguments is sometimes called partial application. Some programming languages automatically curry and partially evaluate functions without the need to manually nest them.

shadowy variables from a shadowy planet

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

```
function (x) {
  return function (x, y) {
    return x + y
  }
}
```

The function `function (x, y) { return 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 functions own environment and then each parent in turn until it finds one. The same is true of:

```
function (x) {
  return function (x, y) {
    return function (w, z) {
      return function (w) {
        return 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 `function (x) { return 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 `(function (x) { return 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
(function () {

  // ... 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 code in every file when they use the var keyword properly.

Let's Talk Var

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:

```
1 function (diameter) {
2   return diameter * 3.14159265
3 }
```

What is this "3.14159265" number? Pi, obviously. We'd like to name it so that we can write something like:

```
1 function (diameter) {
2   return diameter * Pi
3 }
```

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:

```
1 (function (Pi) {
2   return ????
3 })(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:

```
1 (function (Pi) {
2   return function (diameter) {
3     return diameter * Pi
4   }
```

```
5 })(3.14159265)
```

This expression, when evaluated, returns a function that calculates circumferences. It differs from our original in that it names the constant `Pi`. Let's test it:

```
1 (function (Pi) {
2   return function (diameter) {
3     return diameter * Pi
4   }
5 })(3.14159265)(2)
6   //=> 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.

immediately invoked function expressions

JavaScript 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 for it, often abbreviated "IIFE." As we'll see in a moment, an IIFE need not have parameters:

```
1 (function () {
2   // ... do something here...
3 })();
```

When an IIFE binds values to names (as we did above with `Pi`), retro-grouch programmers often call it "let."[11] And confusing the issue, upcoming versions of JavaScript have support for a `let` keyword that has a similar binding behaviour.

var

Using an IIFE to bind names works very well, but only a masochist would write programs this way in JavaScript. Besides all the extra characters, it suffers from a fundamental semantic problem: there is a big visual distance between the name `Pi` and the value `3.14159265` we bind to it. They should be closer. Is there another way?

Yes.

Another way to write our "circumference" function would be to pass `Pi` along with the diameter argument, something like this:

```
1 function (diameter, Pi) {
```

```
  return diameter * Pi
}
```

And you could use it like this:

```
(function (diameter, Pi) {
  return 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 `var` keyword. We’ll learn a lot more about `var` in future chapters, but here’s the most important thing you can do with `var`:

```
function (diameter) {
  var Pi = 3.14159265;

  return diameter * Pi
}
```

The `var` keyword introduces one or more bindings in the current function’s environment. It works just as we want:

```
(function (diameter) {
  var Pi = 3.14159265;

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

You can bind any expression. Functions are expressions, so you can bind helper functions:

```
function (d) {
  var calc = function (diameter) {
    var Pi = 3.14159265;

    return diameter * Pi
  };
```

```
7
8   return "The circumference is " + calc(d)
9 }
```

Notice `calc(d)`? This underscores what we've said: if you have an expression that evaluates to a function, you apply it with `()`. A name that's bound to a function is a valid expression evaluating to a function.[1][2]

> 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.

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

```
1 function (d) {
2   var Pi   = 3.14159265,
3       calc = function (diameter) {
4         return diameter * Pi
5       };
6
7   return "The circumference is " + calc(d)
8 }
```

These examples use the `var` keyword to bind names in the same environment as our function. We can also create a new scope using an IIFE if we wish to bind some names in part of a function:

```
1 function foobar () {
2
3   // do something without foo or bar
4
5   (function () {
6     var foo = 'foo',
7         bar = 'bar';
8
9     // ... do something with foo and bar ...
10
11  })();
12
```

```
13   // do something else without foo or bar
14
15 }
```

Naming Functions

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

```
1 var repeat = function (str) {
2   return str + str
3 };
```

It doesn't name the function "repeat" for the same reason that `var answer = 42` doesn't name the number `42`. That snippet of code binds an anonymous function to a name in an environment, but the function itself remains anonymous.

JavaScript does have a syntax for naming a function, it looks like this:

```
1 var bindingName = function actualName () {
2   //...
3 };
```

In this expression, `bindingName` is the name in the environment, but `actualName` 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 the function itself. And indeed the name is a property:

```
1 bindingName.name
2   //=> 'actualName'
```

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.

```
1 someBackboneView.on('click', function clickHandler () {
2   //...
3 });
```

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

```
var 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:

```
var fn = function even (n) {
  if (n === 0) {
    return true
  }
  else return !even(n - 1)
}

fn(5)
  //=> false

fn(2)
  //=> true
```

even is bound within the function itself, but not outside it. This is useful for making recursive functions.

function declarations

We’ve actually buried the lede.[13] Naming functions for the purpose of debugging is not as important as what we’re about to discuss. There is another syntax for naming and/or defining a function. It’s called a function declaration, and it looks like this:

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

This behaves a little like:

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

In that it binds a name in the environment to a named function. However, consider this piece of code:

```
(function () {
  return someName;

  var someName = function someName () {
    // ...
  }
})()
  //=> undefined
```

This is what we expect given what we learned about var: Although someName is declared later in the function, JavaScript behaves as if you'd written:

```
(function () {
  var someName;

  return someName;

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

What about a function declaration without var?

```
(function () {
  return someName;

  function someName () {
    // ...
  }
})()
  //=> [Function: someName]
```

Aha! It works differently, as if you'd written:

```
(function () {
  var someName = function someName () {
    // ...
  }
  return someName;
})()
```

That difference 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 var) is essential for working with production code.

function declaration caveats[14]

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

```
// function declarations should not happen inside of
// a block and/or be conditionally executed
if (frobbishes.arePizzled()) {
  function complainToFactory () {
    // ...
  }
}
```

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.

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:

```
function repeat (num, fn) {
  var i, value;

  for (i = 1; i <= num; ++i)
    value = fn(i);

  return value;
}

repeat(3, function () {
  console.log('Hello')
})
  //=>
    'Hello'
    'Hello'
    'Hello'
    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.”–Wikipedia

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 Mockingbird.

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 typical[15] programming implementation:

```
function compose (a, b) {
  return function (c) {
    return a(b(c))
  }
}
```

Let's say we have:

```
function addOne (number) {
  return number + 1
}

function doubleOf (number) {
  return number * 2
}
```

With compose, anywhere you would write

```
function doubleOfAddOne (number) {
  return doubleOf(addOne(number))
}
```

You could also write:

```
var 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 ridiculous example of a decorator:

```
function not (fn) {
  return function (argument) {
    return !fn(argument)
  }
}
```

So instead of writing `!someFunction(42)`, you can write `not(someFunction)(42)`. Hardly progress. But like for `compose`, if you have:

```
function something (x) {
  return x != null
}
```

Then you could write either:

```
function nothing (x) {
  return !something(x)
}
```

Or:

```
var 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, mapWith, 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:

```
function cookAndEat (food) {
  return 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:

```
function compose (a, b) {
  return function (c) {
    return a(b(c))
  }
}

var 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:

```
function actuallyTransfer(from, to, amount) {
  // do something
}

var 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 Underscore library provides a higher-order function called map.[16] It applies another function to each element of an array, like this:

```
1 _.map([1, 2, 3], function (n) { return n * n })
2   //=> [1, 4, 9]
```

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

```
1 function squareAll (array) {
2   return _.map(array, function (n) { return n * n })
3 }
```

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.

```
1  function mapWith (fn) {
2    return function (array) {
3      return _.map(array, fn)
4    }
5  }
6
7  var squareAll = mapWith(function (n) { return n * n });
8
9  squareAll([1, 2, 3])
10   //=> [1, 4, 9]
```

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

```
1 var safeSquareAll = mapWith(maybe(function (n) { return n * n }));
2
3 safeSquareAll([1, null, 2, 3])
4   //=> [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.

I'd Like to Have Some Arguments. Again.

As we've discussed, when a function is applied to arguments (or "called"), JavaScript binds the values of the arguments to the function's argument names in an environment created for the function's execution. What we didn't discuss is that JavaScript also binds some "magic" names in addition to any you put in the argument list.

You should never attempt to define your own bindings against these names. Consider them read-only at all times. The first is called this and it is bound to something called the function's context. We will explore that when we start discussing objects and classes. The second is very interesting, it's called arguments, and the most interesting thing about it is that it contains a list of arguments passed to the function:

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

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

Although arguments looks like an array, it isn't an array:[17] It's more like an object[18] that happens to bind some values to properties with names that look like integers starting with zero:

```
function args (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:

```
function plus () {
  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:

```
function howMany () {
  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.

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."
- Function bodies have zero or more expressions.
- Function application evaluates whatever is returned with the return keyword, or to undefined.
- Function application creates a scope. Scopes are nested and free variable references closed over.
- Variables can shadow variables in an enclosing scope.
- let is an idiom where we create a function and call it immediately in order to bind values to names.
- JavaScript uses var to bind variables within a function's scope.

If 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! ↩

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

As 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.↩

TODO: Named functions, probably discussed in a whole new section when we discuss `var` hoisting.↩

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.↩

What a mouthful! This is why other languages with a strong emphasis on functions come up with syntaxes like ` -> -> undefined`↩

The Argument Sketch from "Monty Python's Previous Record" and "Monty Python's Instant Record Collection"

We 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.↩

Unless the argument is NaN, which isn't equal to anything, including itself↩

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

To be pedantic, both main branches of Lisp today define a special construct called "let." One, Scheme, uses `define-syntax` to rewrite `let` into an immediately invoked function expression that binds arguments to values as shown above. The other, Common Lisp, leaves it up to implementations to decide how to implement `let`.↩

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

A lead (or lede) paragraph in literature refers to the opening paragraph of an article, essay, news story or book chapter. In journalism, the failure to mention the most important, interesting or attention-grabbing elements of a story in the first paragraph is sometimes called "burying the lede."↩

A number of the caveats discussed here were described in Jyrly Zaytsev's excellent article Named function expressions demystified.↩

As 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.↩

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

Tradition would have us call objects that don't contain any functions "POJOs," meaning Plain Old JavaScript Objects.

↩

Tradition would have us call objects that don’t contain any functions “POJOs,” meaning Plain Old JavaScript Objects.
↩

In the recipes, you may see one or more of the following JavaScript constructs being used before being fully explained in the text. Here're some brief explanations to tide you over:

apply and call

Functions are applied with `()`. But they also have methods for applying them to arguments. `.call` and `.apply` are explained when we discuss function contexts, but here are some examples:

```
function plus (a, b) {
  return a + b
}

plus(2, 3)
  //=> 5

plus.call(this, 2, 3)
  //=> 5

plus.apply(this, [2, 3])
  //=> 5
```

slice

Arrays have a `.slice` method. The function can always be found at `Array.prototype.slice`. It works like this:

```
[1, 2, 3, 4, 5].slice(0)
  //=> [1, 2, 3, 4, 5]

[1, 2, 3, 4, 5].slice(1)
  //=> [2, 3, 4, 5]

[1, 2, 3, 4, 5].slice(1, 4)
  //=> [2, 3, 4]
```

Note that `slice` always creates a new array, so `.slice(0)` makes a copy of an array. The arguments pseudo-variable is not an array, but you can use `.slice` with it like this to get an array of all or some of the arguments:

```
Array.prototype.slice.call(arguments, 0)
  //=> returns the arguments in an array.

```

```
function butFirst () {
  return Array.prototype.slice.call(arguments, 1)
}

butFirst('a', 'b', 'c', 'd')
  //=> [ 'b', 'c', 'd' ]
```

For simplicity and as a small speed improvement, slice is usually bound to a local variable:

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

function butFirst () {
  return __slice.call(arguments, 1)
}
```

Or even:

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

function slice (list, from, to) {
  return __slice.call(list, from, to)
}

function butFirst () {
  return slice(arguments, 1)
}
```

concat

Arrays have another useful method, .concat. Concat returns an array created by concatenating the receiver with its argument:

```
[1, 2, 3].concat([2, 1])
  //=> [1, 2, 3, 2, 1]
```

function lengths

Functions have a .length property that counts the number of arguments declared:

```
function (a, b, c) { return a + b + c }.length
  //=> 3
```

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 tool. You'll find examples in Lemonad from Michael Fogus, Functional JavaScript from Oliver Steele and the terse but handy node-ap from James Halliday.

These two recipes are for quickly and simply applying a single argument, either the leftmost or rightmost.[1] 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. It is context-agnostic.

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

function callFirst (fn, larg) {
  return function () {
    var args = __slice.call(arguments, 0);

    return fn.apply(this, [larg].concat(args))
  }
}

function callLast (fn, rarg) {
  return function () {
    var args = __slice.call(arguments, 0);

    return fn.apply(this, args.concat([rarg]))
  }
}

function greet (me, you) {
  return "Hello, " + you + ", my name is " + me
}

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

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

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

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

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.

Ellipses and improved Partial Application

The CoffeeScript programming language has a useful feature: If a parameter of a method is written with trailing ellipses, it collects a list of parameters into an array. It can be used in various ways, and the CoffeeScript transpiler does some pattern matching to sort things out, but 80% of the use is to collect a variable number of arguments without using the arguments pseudo-variable, and 19% of the uses are to collect a trailing list of arguments.

Here's what it looks like collecting a variable number of arguments and trailing arguments:

```
callLeft = (fn, args...) ->
  (remainingArgs...) ->
    fn.apply(this, args.concat(remainingArgs))
```

These are very handy features. Here's our bogus, made-up attempt to write our own mapper function:

```
mapper = (fn, elements...) ->
  elements.map(fn)

mapper ((x) -> x*x), 1, 2, 3
  #=> [1, 4, 9]

squarer = callLeft mapper, (x) -> x*x

squarer 1, 2, 3
  #=> [1, 4, 9]
```

JavaScript doesn't support ellipses, those trailing periods CoffeeScript uses to collect arguments into an array. JavaScript is a functional language, so here is the recipe for a function that collects trailing arguments into an array for us:

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

function variadic (fn) {
  var fnLength = fn.length;

  if (fnLength < 1) {
    return fn;
  }
  else if (fnLength === 1)  {
    return function () {
      return fn.call(
        this, __slice.call(arguments, 0))
    }
  }
  else {
    return function () {
      var numberOfArgs = arguments.length,
          namedArgs = __slice.call(
            arguments, 0, fnLength - 1),
          numberOfMissingNamedArgs = Math.max(
            fnLength - numberOfArgs - 1, 0),
```

```
          argPadding = new Array(numberOfMissingNamedArgs),
          variadicArgs = __slice.call(
            arguments, fn.length - 1);

      return fn.apply(
        this, namedArgs
              .concat(argPadding)
              .concat([variadicArgs]));
    }
  }
};

function unary (first) {
  return first
}

unary('why', 'hello', 'there')
  //=> 'why'

variadic(unary)('why', 'hello', 'there')
  //=> [ 'why', 'hello', 'there' ]

function binary (first, rest) {
  return [first, rest]
}

binary('why', 'hello', 'there')
  //=> [ 'why', 'hello' ]

variadic(binary)('why', 'hello', 'there')
  //=> [ 'why', [ 'hello', 'there' ] ]
```

Here's what we write to create our partial application functions gently:

```
var callLeft = variadic( function (fn, args) {
  return variadic( function (remainingArgs) {
    return fn.apply(this, args.concat(remainingArgs))
  })
})

// Let's try it!

var mapper = variadic( function (fn, elements) {
  return elements.map(fn)
});

mapper(function (x) { return x * x }, 1, 2, 3)
```

```
  //=> [1, 4, 9]

var squarer = callLeft(mapper, function (x) { return x * x });

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

While we're at it, here's our implementation of callRight using the same technique:

```
var callRight = variadic( function (fn, args) {
  return variadic( function (precedingArgs) {
    return fn.apply(this, precedingArgs.concat(args))
  })
})
```

Fine print: Of course, variadic introduces an extra function call and may not be the best choice in a highly performance-critical piece of code. Then again, using arguments is considerably slower than directly accessing argument bindings, so if the performance is that critical, maybe you shouldn't be using a variable number of arguments in that section.

Unary

In Ellipses, we saw a function decorator that takes a function with a fixed number of arguments and turns it into a variadic function, a function taking any number of arguments. "Unary" is another function decorator, and it also 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 common 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(function (s) { return parseInt(s); })`, or we could come up with a decorator to do the job for us:

```
function unary (fn) {
  if (fn.length == 1) {
    return fn
  }
  else return 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:"

```
function K (x) {
  return function (y) {
```

```
    return 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:

```
function tap (value) {
  return function (fn) {
    if (typeof(fn) === 'function') {
      fn(value)
    }
    return 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:

```
var drink = tap('espresso')(function (it) {
  console.log("Our drink is", it)
});

// outputs "Our drink is 'espresso'" to the console
```

It's easy to turn off:

```
var drink = tap('espresso')();

// doesn't output anything to the console
```

Libraries like Underscore use a version of tap that is "uncurried:"

```
var drink = _.tap('espresso', function () {
  console.log("Our drink is", this)
});
```

Let's enhance our recipe so it works both ways:

```
function tap (value, fn) {
  if (fn === void 0) {
```

```
    return curried
  }
  else return curried(fn);

  function curried (fn) {
    if (typeof(fn) === 'function') {
      fn(value)
    }
    return value
  }
}
```

Now you can write:

```
var drink = tap('espresso')(function (it) {
  console.log("Our drink is", it)
});
```

Or:

```
var drink = tap('espresso', function (it) {
  console.log("Our drink is", it)
});
```

And if you wish it to do nothing at all, You can write either:

```
var drink = tap('espresso')();
```

Or:

```
var drink = tap('espresso', null);
```

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:

```
function isSomething (value) {
  return value !== null && value !== void 0;
}

function 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 recipe for that, borrowed from Haskell's maybe monad, Ruby's andand, and CoffeeScript's existential method invocation:

```
function maybe (fn) {
  return function () {
    var i;

    if (arguments.length === 0) {
      return
    }
    else {
      for (i = 0; i < arguments.length; ++i) {
        if (arguments[i] == null) return
      }
      return fn.apply(this, arguments)
    }
  }
}
```

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

```
var checksForSomething = maybe(function (value) {
  // function's true logic
});
```

Or:

```
var something = maybe(doesntCheckForSomething)(value);
```

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.

callFirst and callLast were inspired by Michael Fogus' Lemonad. Thanks!↩

It is not enough that coffee taste beautiful. Everything about its creation and consumption should reflect coffee's beauty.

a simple question

Consider this code:

```
var x = 'June 14, 1962',
    y = x;

x === y
  //=> true
```

This makes obvious sense, because we know that strings are a value type, so no matter what expression you use to derive the value ‘June 14, 1962’, you are going to get a string with the exact same identity.

But what about this code?

```
var x = [2012, 6, 14],
    y = x;

x === y
  //=> true
```

Also true, even though we know that every time we evaluate an expression such as `[2012, 6, 14]`, we get a new array with a new identity. So what is happening in our environments?

arguments and references

In our discussion of closures, we said that environments bind values (like `[2012, 6, 14]`) to names (like `x` and `y`), and that when we use these names as expressions, the name evaluates as the value.

What this means is that when we write something like `y = x`, the name `x` is looked up in the current environment, and its value is a specific array that was created when the expression `[2012, 6, 14]` was first evaluated. We then bind that exact same value to the name `y` in a new environment, and thus `x` and `y` are both bound to the exact same value, which is identical to itself.

The same thing happens with binding a variable through a more conventional means of applying a function to arguments:

```
var x = [2012, 6, 14];

(function (y) {
  return x === y
})(x)
  //=> true
```

`x` and `y` both end up bound to the exact same array, not two different arrays that look the same to our eyes.

Arguments and Arrays

JavaScript provides two different kinds of containers for values. We've met one already, the array. Let's see how it treats values and identities. For starters, we'll learn how to extract a value from an array. We'll start with a function that makes a new value with a unique identity every time we call it. We already know that every function we create is unique, so that's what we'll use:

```
var unique = function () {
                return function () {}
              };

  unique()
    //=> [Function]

  unique() === unique()
    //=> false
```

Let's verify that what we said about references applies to functions as well as arrays:

```
  var x = unique(),
      y = x;

  x === y
    //=> true
```

Ok. So what about things inside arrays? We know how to create an array with something inside it:

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

That's an array with one of our unique functions in it. How do we get something out of it?

```
  var a = [ 'hello' ];

  a[0]
    //=> 'hello'
```

Cool, arrays work a lot like arrays in other languages and are zero-based. The trouble

with this example is that strings are value types in JavaScript, so we have no idea whether `a[0]` always gives us the same value back like looking up a name in an environment, or whether it does some magic that tries to give us a new value.

We need to put a reference type into an array. If we get the same thing back, we know that the array stores a reference to whatever you put into it. If you get something different back, you know that arrays store copies of things.[1]

Let's test it:

```
var unique = function () {
                return function () {}
             },
    x = unique(),
    a = [ x ];

a[0] === x
  //=> true
```

If we get a value out of an array using the `[]` suffix, it's the exact same value with the same identity. Question: Does that apply to other locations in the array? Yes:

```
var unique = function () {
                return function () {}
             },
    x = unique(),
    y = unique(),
    z = unique(),
    a = [ x, y, z ];

a[0] === x && a[1] === y && a[2] === z
  //=> true
```

References and Objects

JavaScript also provides 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[2] is a map from names to values, a lot like an environment. The most common syntax for creating an object is simple:

Two objects created this way 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:

```
var unique = function () {
                return function () {}
              },
    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:

```
{ year: 2012, month: 6, day: 14 }['day'] ===
    { year: 2012, month: 6, day: 14 }.day
  //=> true
```

All containers can contain any value, including functions or other containers:

```
var Mathematics = {
  abs: function (a) {
         return a < 0 ? -a : a
       }
};

Mathematics.abs(-5)
```

```
8   //=> 5
```

Funny we should mention Mathematics. If you recall, JavaScript provides a global environment that contains some existing objects that have handy functions you can use. One of them is called Math, and it contains functions for abs, max, min, and many others. Since it is always available, you can use it in any environment provided you don't shadow Math.

```
1 Math.abs(-5)
2   //=> 5
```

Reassignment and Mutation

Like most imperative programming languages, JavaScript allows you to re-assign the value of variables. The syntax is familiar to users of most popular languages:

```
1 var age = 49;
2 age = 50;
3 age
4   //=> 50
```

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:

```
1 (function () {
2   var age = 49;
3   (function () {
4     var age = 50;
5   })();
6   return age;
7 })()
8   //=> 49
```

Binding 50 to age in the inner environment does not change age in the outer environment because the binding of age in the inner environment shadows the binding of age in the outer environment. We go from:

```
1 {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` by explicitly using `var`, reassigning it in a nested environment changes the original:

```
(function () {
  var age = 49;
  (function () {
    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.

Cupping Grinds

mutation and aliases

Now that we can reassign things, there's another important factor to consider: Some values 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 [] as well:

```
var oneTwoThree = [1, 2, 3];
oneTwoThree[0] = 'one';
oneTwoThree
  //=> [ 'one', 2, 3 ]
```

You can even add a value:

```
var oneTwoThree = [1, 2, 3];
oneTwoThree[3] = 'four';
oneTwoThree
  //=> [ 1, 2, 3, 'four' ]
```

You can do the same thing with both syntaxes for accessing objects:

```
var 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:

```
var allHallowsEve = [2012, 10, 31]
var halloween = allHallowsEve;
```

Both halloween and allHallowsEve are bound to the same array value within the local environment. And also:

```
var 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:

```
var 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?

```
var 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.

How to Shoot Yourself in the Foot With Var

As we've seen, JavaScript's environments and bindings are quite powerful: You can bind and rebind names using function arguments or using variables declared with var. The takeaway is that when used properly, Javascript's var keyword is a great tool.

When used properly.

Let's look at a few ways to use it improperly.

loose use

JavaScript's var keyword is scoped to the function enclosing it. This makes sense, because bindings are made in environments, and the environments are associated with function calls. So if you write:

```
function foo (bar) {
  var baz = bar * 2;

  if (bar > 1) {
    var blitz = baz - 100;

    // ...
  }
}
```

The name `blitz` is actually scoped to the function `foo`, not to the block of code in the consequent of an `if` statement. There are roughly two schools of thought. One line of reasoning goes like this: Since `blitz` is scoped to the function `foo`, you should write the code like this:

```
function foo (bar) {
  var baz = bar * 2,
      blitz;

  if (bar > 1) {
    blitz = baz - 100;

    // ...
  }
}
```

We've separated the "declaration" from the "assignment," and we've made it clear that `blitz` is scoped to the entire function. The other school of thought is that programmers are responsible for understanding how the tools work, and even if you write it the first way, other programmers reading the code ought to know how it works.

So here's a question: Are both ways of writing the code equivalent? Let's set up a test case that would tell them apart. We'll try some aliasing:

```
var questionable = 'outer';

(function () {
  alert(questionable);

  if (true) {
    var questionable = 'inner';
    alert(questionable)
  }
})()
```

What will this code do if we type it into a browser console? One theory is that it will alert `outer` and then `inner`, because when it evaluates the first alert, `questionable` hasn't been bound in the function's environment yet, so it will be looked up in the enclosing environment. Then an alias is bound, shadowing the outer binding, and it will alert `inner`.

This theory is wrong! It actually alerts `undefined` and then `inner`. Even though we wrote the `var` statement later in the code, JavaScript acts as if we'd declared it at the top of the function. This is true even if we never execute the `var` statement:

```
var questionable = 'outer';

(function () {
  return questionable;

  var questionable = 'inner'
})()

  //=> undefined
```

So yes, both ways of writing the code work the same way, but only one represents the way it works directly and obviously. For this reason, we put the `var` declarations at the top of every function, always.

for pete's sake

JavaScript provides a `for` loop for your iterating pleasure and convenience. It looks a lot like the `for` loop in C:

```
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.[3] 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?

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] = function (soAndSo) {
    return "Hello, " + soAndSo + ", my name is " + names[i]
  }
}
introductions
  //=> [ [Function],
  //     [Function],
  //     [Function] ]
```

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;

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.

Here's how to fix it, once again with `let` as our guide:

```
var introductions = [],
    names = ['Karl', 'Friedrich', 'Gauss'];

for (var i = 0; i < 3; i++) {
  (function (i) {
    introductions[i] = function (soAndSo) {
      return "Hello, " + soAndSo + ", my name is " + names[i]
    }
  })(i)
}
introductions[1]('Raganwald')
  //=> 'Hello, Raganwald, my name is Friedrich'
```

That works. What did we do? Well, we created a new function and called it immediately, and we deliberately shadowed `i` by passing it as an argument to our function, which had an argument of exactly the same name. If you dislike shadowing, this alternative also works:

```
var introductions = [],
    names = ['Karl', 'Friedrich', 'Gauss'];

for (var i = 0; i < 3; i++) {
  (function () {
    var ii = i;
    introductions[ii] = function (soAndSo) {
      return "Hello, " + soAndSo + ", my name is " + names[ii]
    }
  })()
}
introductions[1]('Raganwald')
  //=> 'Hello, Raganwald, my name is Friedrich'
```

Now we're creating a new inner variable, `ii` and binding it to the value of `i`. The shadowing code seems simpler and less error-prone to us, but both work.

nope, nope, nope, nope, nope

The final caution about `var` concerns what happens if you omit to declare a variable with

var, boldly writing something like:

```
fizzBuzz = function () {
  // lots of interesting code elided
  // for the sake of hiring managers
}
```

So where is the name fizzBuzz bound? The answer is that if there is no enclosing var declaration for fizzBuzz, the name is bound in the global environment. And by global, we mean global. It is visible to every separate compilation unit. All of your npm modules. Every JavaScript snippet in a web page. Every included file.

This is almost never what you want. And when you do want it, JavaScript provides alternatives such as binding to window.fizzBuzz in a browser, or this.fizzBuzz in node. In short, eschew undeclared variables. Force yourself to make a habit of using var all of the time, and explicitly binding variables to the window or this objects when you truly want global visibility.

When Rebinding Meets Recursion

We've talked about binding values in environments, and now we're talking about rebinding values and mutating values. Let's take a small digression. As we've seen, in JavaScript functions are values. So you can bind a function just like binding a string, number or array. Here's a function that tells us whether a (small and positive) number is even:

```
var even = function (num) {
  return (num === 0) || !(even(num - 1))
}

even(0)
  //=> true

even(1)
  //=> false

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

You can alias a function value:

```
var divisibleByTwo = even;

```

```
divisibleByTwo(0)
  //=> true

divisibleByTwo(1)
  //=> false

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

What happens when we redefine a recursive function like even? Does divisibleByTwo still work? Let's try aliasing it and reassigning it:

```
even = void 0;

divisibleByTwo(0)
  //=> true

divisibleByTwo(1)
  //=> TypeError
```

What happened? Well, our new divisibleByTwo function wasn't really a self-contained value. When we looked at functions, we talked about "pure" functions that only access their arguments and we looked at "closures" that have free variables. Recursive functions defined like this are closures, not pure functions, because when they "call themselves," what they actually do is look themselves up by name in their enclosing environment. Thus, they depend upon a specific value (themselves) being bound in their enclosing environment. Reassign to that variable (or rebind the name, same thing), and you break their functionality.

named function expressions

You recall that in Naming Functions, we saw that when you create a named function expression, you bind the name of the function within its body but not the environment of the function expression, meaning you can write:

```
var even = function myself (num) {
  return (num === 0) || !(myself(num - 1))
}

var divisibleByTwo = even;
even = void 0;

divisibleByTwo(0)
  //=> true

```

```
divisibleByTwo(1)
  //=> false

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

This is different, because the function doesn't refer to a name bound in its enclosing environment, it refers to a name bound in its own body. It is now a pure function. In fact, you can even bind it to the exact same name in its enclosing environment and it will still work:

```
var even = function even (num) {
  return (num === 0) || !(even(num - 1))
}

var divisibleByTwo = even;
even = void 0;

divisibleByTwo(0)
  //=> true

divisibleByTwo(1)
  //=> false

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

The `even` inside the function refers to the name bound within the function by the named function expression. It may have the same name as the `even` bound in the enclosing environment, but they are two different bindings in two different environments. Thus, rebinding the name in the enclosing environment does not break the function.

You may ask, what if we rebind `even` inside of itself. Now will it break?

```
var even = function even (num) {
  even = void 0;
  return (num === 0) || !(even(num - 1))
}

var divisibleByTwo = even;
even = void 0;

divisibleByTwo(0)
  //=> true

```

```
divisibleByTwo(1)
  //=> false

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

Strangely, no it doesn't. The name bound by a named function expression is read-only. Why do we say strangely? Because other quasi-declarations like function declarations do not behave like this.

So, when we want to make a recursive function, the safest practice is to use a named function expression.

limits

Named function expressions have limits. Here's one such limit: You can do simple recursion, but not mutual recursion. For example:

```
var even = function even (num) { return (num === 0) || odd( num - 1) };
var odd  = function odd  (num) { return (num  >  0) && even(num - 1) };

odd = 'unusual';

even(0)
  //=> true

even(1)
  //=> TypeError
```

Using named function expressions doesn't help us, because even and odd need to be bound in an environment accessible to each other, not just to themselves. You either have to avoid rebinding the names of these functions, or use a closure to build a module:

```
var operations = (function () {
      var even = function (num) { return (num === 0) || odd( num - 1) };
      var odd  = function (num) { return (num  >  0) && even(num - 1) };
      return {
        even: even,
        odd:  odd
      }
    })(),
    even = operations.even,
    odd = operations.odd;
```

Now you can rebind one without breaking the other, because the names outside of the

closure have no effect on the bindings inside the closure:

```
odd = 'unusual;

even(0)
  //=> true

even(1)
  //=> false

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

As has often been noted, refactoring to a pattern is more important than designing with a pattern. So don't rush off to write all your recursive functions this way, but familiarize yourself with the technique so that if and when you run into a subtle bug, you can recognize the problem and know how to fix it.

From Let to Modules

transient let

In the section on let and Var, we learned that we can create a new environment any time we want by combining a function definition with a function invocation, to whit:

```
(function () {
  //
})();
```

Because this function is invoked, if it contains a return statement, it evaluates to a value of some kind. So you can, for example, write something like:

```
var factorialOfTwentyFive = (function () {
  var factorial = function (num) {
    if (num  <  2 ) {
      return 1
    }
    else return num * factorial (num - 1)
  }
  return factorial(25)
})();
```

This could have been written using a named function to avoid the need for a let, but as we'll see in the memoize later, sometimes there's good reason to write it like this. In any event, our let serves to create a scope for the factorial function. Presumably we write it this way to signal that we do not want to use it elsewhere, so putting it inside of a let keeps it invisible from the rest of the code.

You'll note that once we've calculated the factorial of 25, we have no further need for the environment of the function, so it will be thrown away. This is what we might call a **transient let**: Nothing we bind in the let is returned from the let or otherwise passed out through assignment, so the environment of the let is discarded when the let finishes being evaluated.

private closure

The transient let only uses its environment to generate the result, then it can be discarded. Another type of let is the **private closure**. This let returns a closure that references one or more bindings in the let's environment. For example:

```
1 var counter = (function () {
2   var value = 0;
3
4   return function () {
5     return value++
6   }
7 })();
8
9 counter()
10   //=> 0
11
12 counter()
13   //=> 1
14
15 counter()
16   //=> 2
```

counter is bound to a function closure that references the binding value in the let's environment. So the environment isn't transient, it remains active until the function bound to the name counter is discarded. Private closures are often used to manage state as we see in the counter example, but they can also be used for helper functions.

For example, this date format function cribbed from somewhere or other has a helper function that isn't used anywhere else:

```
1 function formatDate (time) {
```

```
  var date;

  if (time) {
    date = unformattedDate(time);
    // Have to massage the date because
    // we can't create a date
    // based on GMT which the server gives us

    if (!(/-\d\d:\d\d/.test(time))) {
      date.setHours(
        date.getHours() - date.getTimezoneOffset()/60);
    }

    var diff = (
        (new Date()).getTime() - date.getTime()
      ) / 1000;
    day_diff = Math.floor(diff / 86400);

    if ( isNaN(day_diff) || day_diff < 0  )
      return;

    return '<span title="' + date.toUTCString() + '">' + (day_diff == 0 && (
        diff < 60 && "just now" ||
        diff < 120 && "1 minute ago" ||
        diff < 3600 && Math.floor( diff / 60 ) + " minutes ago" ||
        diff < 7200 && "1 hour ago" ||
        diff < 86400 && Math.floor( diff / 3600 ) + " hours ago") ||
      day_diff == 1 && "Yesterday" ||
      day_diff < 7 && day_diff + " days ago" ||
      day_diff < 31 && Math.ceil( day_diff / 7 ) + " weeks ago" ||
      (day_diff < 360 && day_diff >= 31) && Math.ceil(day_diff / 31) +
        ' month' + (day_diff == 31 ? '' : 's') + ' ago' ||
        day_diff > 360 && Math.floor( day_diff / 360) + " years " +
        Math.floor(day_diff%360/32) + " months ago") + '</span>';
  }
  else return '-'

  function unformattedDate (time) {
    return new Date((time || "").replace(/[-+]/g,"/").
      replace(/[TZ]/g," ").replace(/\/\d\d:\d\d/, ''));
  }
}
```

Every time we call formatDate, JavaScript will create an entirely new unformattedDate function. That is not necessary, since it's a pure function. In theory, a sufficiently smart interpreter would notice this and only create one function. In practice, we can rewrite it to use a private closure and only create one helper function:

```
var formatDate = (function () {
  return function (time) {
    var date;

    if (time) {
      date = unformattedDate(time);
      // Have to massage the date because we can't create a date
      // based on GMT which the server gives us

      if (!(/-\d\d:\d\d/.test(time))) {
        date.setHours(date.getHours() - date.getTimezoneOffset()/60);
      }

      var diff = ((new Date()).getTime() - date.getTime()) / 1000;
      day_diff = Math.floor(diff / 86400);

      if ( isNaN(day_diff) || day_diff < 0  )
        return;

      return '<span title="' + date.toUTCString() + '">' + (day_diff == 0 && (
          diff < 60 && "just now" ||
          diff < 120 && "1 minute ago" ||
          diff < 3600 && Math.floor( diff / 60 ) + " minutes ago" ||
          diff < 7200 && "1 hour ago" ||
          diff < 86400 && Math.floor( diff / 3600 ) + " hours ago") ||
        day_diff == 1 && "Yesterday" ||
        day_diff < 7 && day_diff + " days ago" ||
        day_diff < 31 && Math.ceil( day_diff / 7 ) + " weeks ago" ||
        (day_diff < 360 && day_diff >= 31) && Math.ceil(day_diff / 31) +
          ' month' + (day_diff == 31 ? '' : 's') + ' ago' ||
          day_diff > 360 && Math.floor( day_diff / 360) +
          " years " + Math.floor(day_diff%360/32) + " months ago") + '</span>';
    }
    else return '-'
  }

  function unformattedDate (time) {
    return new Date((time || "").replace(/[-+]/g,"/").replace(/[TZ]/g," ").repla\
ce(/\/\d\d:\d\d/, ''));
  }
})();
```

The function unformattedDate is still private to formatDate, but now we no longer need to construct an entirely new function every time formatDate is called.

modules

Once the power of creating a new environment with a let (or “immediately invoked function expression”) is tasted, it won’t be long before you find yourself building modules with lets. Modules are any collection of functions that have some private and some public-facing elements.

Consider a module designed to draw some bits on a virtual screen. The public API consists of a series of draw functions. The private API includes a series of helper functions. This is exactly like the **private closure**, the only difference being that we want to return multiple public functions instead of just one.

It looks like this:

```
var DrawModule = (function () {

  return {
    drawLine: drawLine,
    drawRect: drawRect,
    drawCircle: drawCircle
  }

  // public methods
  function drawLine(screen, leftPoint, rightPoint) { ... }
  function drawRect(screen, topLeft, bottomRight) { ... }
  function drawCircle(screen, center, radius) { ... }

  // private helpers
  function bitBlt (screen, ...) { ... }
  function resize (screen, ...) { ... }

})();
```

You can then call the public functions using `DrawModule.drawCircle(...)`. The concept scales up to include the concept of state (such as setting default line styles), but when you look at it, it’s really just the private closure let with a little more complexity in the form of returning an object with more than one function.

Summary

Rebinding

- JavaScript permits reassignment/rebinding of variables.
- Arrays and Objects are mutable.
- References permit aliasing of reference types.
- We may need to take special care to prevent ourselves from accidentally

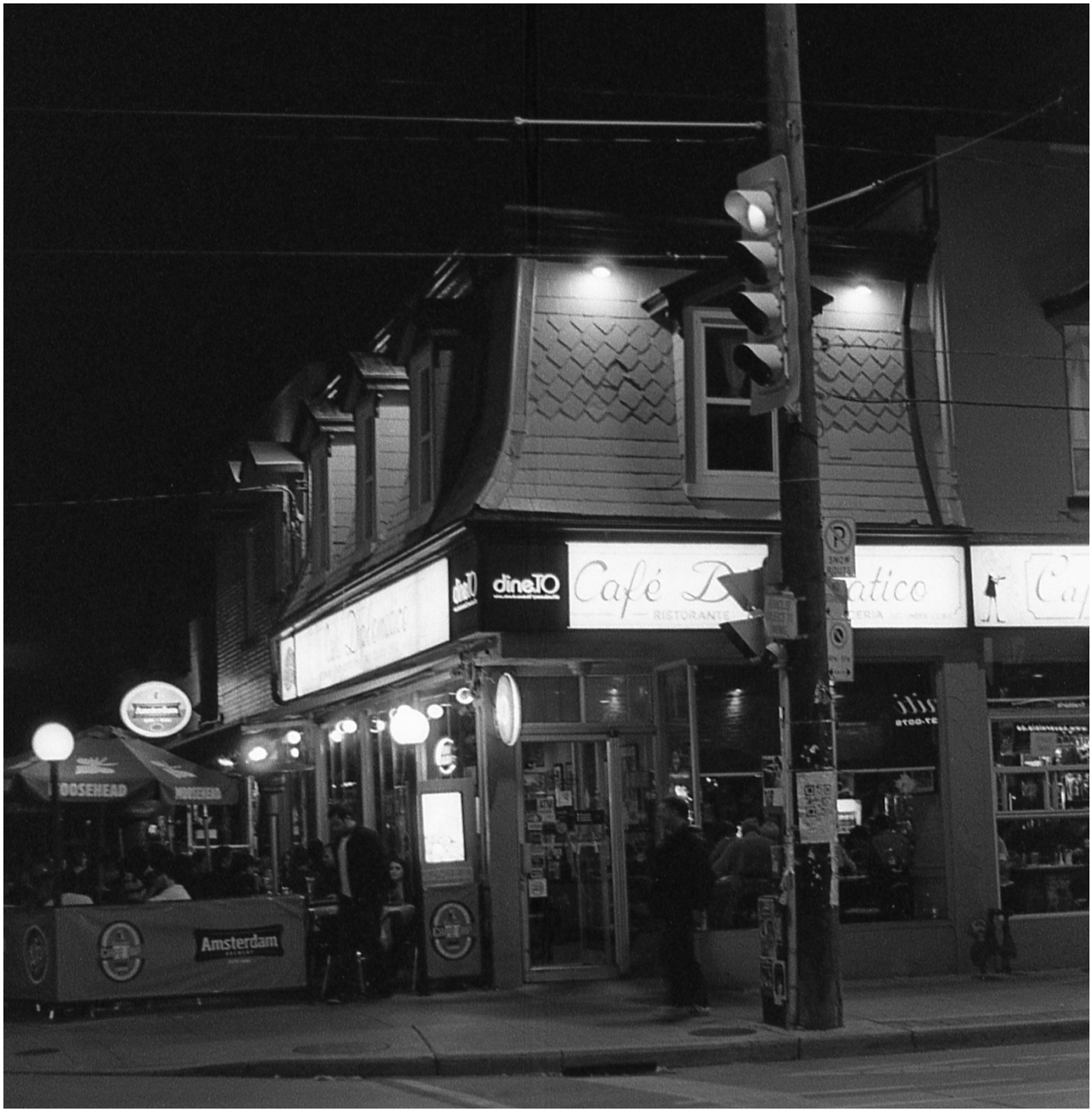
breaking recursive functions.

- For loops are convenient, but require care to avoid scoping bugs.

Arrays in all contemporary languages store references and not copies, so we can be forgiven for expecting them to work the same way in JavaScript. Nevertheless, it's a useful exercise to test things for ourself.↩

Tradition would have us call objects that don't contain any functions "POJOs," meaning Plain Old JavaScript Objects.
↩

There 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: $1 + 2 + 3 + \ldots + 99 + 100 = ?$ But Gauss rearranged the numbers to add them like this: $(1 + 100) + (2 + 99) + (3 + 98) + \ldots + (50 + 51) = ?$ If you notice every pair of numbers adds up to 101. There are 50 pairs of numbers, so the answer is 50*101 = 5050. Of course Gauss came up with the answer about 20 times faster than the other kids.↩



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.

Once

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

```
function once (fn) {
  var 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:

```
var askedOnBlindDate = once(function () {
  return 'sure, why not?'
});

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

askedOnBlindDate()
  //=> undefined

askedOnBlindDate()
  //=> undefined
```

It seems some people will only try blind dating once. But you do have to be careful that you are calling the function once returns multiple times. If you keep calling once, you'll get a new function that executes once, so you'll keep calling your function:

```
once(function () {
  return 'sure, why not?'
})()
  //=> 'sure, why not?'

once(function () {
  return 'sure, why not?'
})()
  //=> 'sure, why not?'
```

This is expected, but sometimes not what we want. So we must either be careful with our code, or use a variation, the named once recipe.

mapWith

In recent versions of 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(function (n) {
  return n*n
})
  //=> [1, 4, 9, 16, 25]
```

We say that .map maps its arguments over the receiver array's elements. Or if you prefer, that it defines a mapping between its receiver and its result. Libraries like Underscore provide a map function.[1] It usually works like this:

```
_.map([1, 2, 3, 4, 5], function (n) {
  return n*n
})
  //=> [1, 4, 9, 16, 25]
```

This recipe isn't for map: It's for mapWith, a function that wraps around map and turns any other function into a mapping. In concept, mapWith is very simple:[2]

```
function mapWith (fn) {
  return function (list) {
    return Array.prototype.map.call(list, function (something) {
      return fn.call(this, something);
    });
  };
};
```

Here's the above code written using mapWith:

```
var squareMap = mapWith(function (n) {
  return n*n;
});

squareMap([1, 2, 3, 4, 5])
  //=> [1, 4, 9, 16, 25]
```

If we didn't use mapWith, we'd have written something like this:

```
var squareMap = function (array) {
  return Array.prototype.map.call(array, function (n) {
    return n*n;
```

```
  });
};
```

And we'd do that every time we wanted to construct a method that maps an array to some result. mapWith is a very convenient abstraction for a very common pattern.

mapWith was suggested by ludicast

Flip

When we wrote mapWith, we wrote it like this:

```
function mapwith (fn) {
  return function (list) {
    return Array.prototype.map.call(list, function (something) {
      return fn.call(this, something);
    });
  };
};
```

Let's consider the case whether we have a map function of our own, perhaps from the allong.es library, perhaps from Underscore. We could write our function something like this:

```
function mapwith (fn) {
  return function (list) {
    return map.call(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:

```
function mapwith (fn, list) {
  return map.call(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:

```
function mapCurried (list) {
  return function (fn) {
```

```
    return map(list, fn);
  };
};
```

Let’s return to the implementation that does both:

```
function mapWith (fn) {
  return function (list) {
    return map.call(list, fn);
  };
};
```

Now let’s put a wrapper around it:

```
function wrapper () {
  return function (fn) {
    return function (list) {
      return map.call(list, fn);
    };
  };
};
```

Abstract the parameter names:

```
function wrapper () {
  return function (first) {
    return function (second) {
      return map.call(second, first);
    };
  };
};
```

And finally, extract the function as a parameter:

```
function wrapper (fn) {
  return function (first) {
    return function (second) {
      return fn.call(second, first);
    };
  };
};
```

What we have now is a function that takes a function and “flips” the order of arguments

around, then curries it:

```
function flip (fn) {
  return function (first) {
    return function (second) {
      return fn.call(this, second, first);
    };
  };
};
```

This is gold. Consider how we define mapWith now:

```
var mapWith = flip(map);
```

Much nicer!

There's one final decoration. Sometimes we'll want to flip a function but retain the flexibility to call it with both parameters at once. No problem:

```
function flip (fn) {
  return function (first, second) {
    if (arguments.length === 2) {
      return fn.call(this, second, first);
    }
    else {
      return function (second) {
        return fn.call(this, second, first);
      };
    };
  };
};
```

Now you can call mapWith(fn, list) or mapWith(fn)(list), your choice.

Extend

It's very common to want to "extend" an object by adding properties to it:

```
var inventory = {
  apples: 12,
  oranges: 12
};

inventory.bananas = 54;
```

```
inventory.pears = 24;
```

It's also common to want to add a shallow copy of the properties of one object to another:

```
for (var fruit in shipment) {
  inventory[fruit] = shipment[fruit]
}
```

Both needs can be met with this recipe for extend:

```
var extend = variadic( function (consumer, providers) {
  var key,
      i,
      provider;

  for (i = 0; i < providers.length; ++i) {
    provider = providers[i];
    for (key in provider) {
      if (provider.hasOwnProperty(key)) {
        consumer[key] = provider[key]
      }
    }
  }
  return consumer
});
```

You can copy an object by extending an empty object:

```
extend({}, {
  apples: 12,
  oranges: 12
})
  //=> { apples: 12, oranges: 12 }
```

You can extend one object with another:

```
var inventory = {
  apples: 12,
  oranges: 12
};

var shipment = {
  bananas: 54,
  pears: 24
```

```
}

extend(inventory, shipment)
  //=> { apples: 12,
  //     oranges: 12,
  //     bananas: 54,
  //     pears: 24 }
```

And when we discuss prototypes, we will use extend to turn this:

```
var 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:

```
var Queue = function () {
  extend(this, {
    array: [],
    head: 0,
    tail: -1
  })
};

extend(Queue.prototype, {
  pushTail: function (value) {
    // ...
  },
  pullHead: function () {
    // ...
  },
  isEmpty: function () {
    // ...
  }
```

```
19 });
```

Why?

This is the canonical Y Combinator:

```
1 function Y (f) {
2   return ((function (x) {
3     return f(function (v) {
4       return x(x)(v);
5     });
6   })(function (x) {
7     return f(function (v) {
8       return x(x)(v);
9     });
10  }));
11 }
```

You use it like this:

```
1 var factorial = Y(function (fac) {
2   return function (n) {
3     return (n == 0 ? 1 : n * fac(n - 1));
4   }
5 });
6
7 factorial(5)
8   //=> 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. A friendly tip: Name some of the anonymous functions inside it to help you decipher stack traces.

Work things out for yourself. And once you've grokked that recipe, this recipe is for a Y Combinator that is a little more idiomatic. Work it out too:

```
function Y (fn) {
  var f = function (f) {
    return function () {
      return fn.apply(f, arguments)
    }
  };

  return ((function (x) {
    return f(function (v) {
      return x(x)(v);
    });
  })(function (x) {
    return f(function (v) {
      return x(x)(v);
    });
  }));
}
```

You use this version like this:

```
var factorial = Y(function (n) {
  return (n == 0 ? 1 : n * this(n - 1));
});

factorial(5)
```

There are certain cases involving nested recursive functions it cannot handle due to the ambiguity of this, and obviously it is useless as a method combination, but it is an interesting alternative to the let pattern.

Why provide a map function? well, JavaScript is an evolving language, and when you're writing code that runs in a web browser, you may want to support browsers using older versions of JavaScript that didn't provide the .map function. One way to do that is to "shim" the map method into the Array class, the other way is to use a map function. Most library implementations of map will default to the .map method if its available.↩

If we were always mapWithting arrays, we could write list.map(fn). However, there are some objects that have a .length property and [] accessors that can be mapWithted 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.↩

Life measured out by coffee spoons

So far, we have discussed what many call “pure functional” programming, where every expression is necessarily idempotent, because we have no way of changing state within a program using the tools we have examined.

It’s time to change everything.

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

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,[1] 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

Consider a stack 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.

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 index[2]), 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:

```
var stack = (function () {
  var obj = {
    array: [],
    index: -1,
    push: function (value) {
      return obj.array[obj.index += 1] = value
    },
    pop: function () {
      var value = obj.array[obj.index];
      obj.array[obj.index] = void 0;
      if (obj.index >= 0) {
        obj.index -= 1
      }
      return value
    },
    isEmpty: function () {
      return obj.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
```

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: function (x, y) {
    if (x < y) {
      return x
    }
    else {
      return y
    }
  }
  max: function (x, y) {
    if (x > y) {
      return x
    }
    else {
      return 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:

```
var stack = (function () {
  var array = [],
      index = -1;

  return {
    push: function (value) {
      array[index += 1] = value
    },
```

```
    pop: function () {
      var value = array[index];
      if (index >= 0) {
        index -= 1
      }
      return value
    },
    isEmpty: function () {
      return index < 0
    }
  }
})();

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:

```
var StackMaker = function () {
  return (function () {
    var array = [],
        index = -1;

    return {
      push: function (value) {
        array[index += 1] = value
      },
      pop: function () {
        var value = array[index];
        if (index >= 0) {
          index -= 1
        }
        return value
      },
      isEmpty: function () {
        return index < 0
      }
    }
  })()
}
```

But there's an easier way :-)

```
var StackMaker = function () {
  var array = [],
      index = -1;

  return {
    push: function (value) {
      array[index += 1] = value
    },
    pop: function () {
      var value = array[index];
      if (index >= 0) {
        index -= 1
      }
      return value
    },
    isEmpty: function () {
```

```
      return index < 0
    }
  }
};

stack = StackMaker()
```

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 Instances 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. So maybe we aren't missing much.

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 theory [3].

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:

```
// helper function
//
// For production use, consider what to do about
```

```
// deep copies and own keys
var shallowCopy = function (source) {
  var dest = {},
      key;

  for (key in source) {
    dest[key] = source[key]
  }
  return dest
};

// our model maker
var ModelMaker = function (initialAttributes) {
  var attributes = shallowCopy(initialAttributes || {}),
      undoStack = StackMaker(),
      redoStack = StackMaker(),
      obj = {
        set: function (attrsToSet) {
          var key;

          undoStack.push(shallowCopy(attributes));
          if (!redoStack.isEmpty()) {
            redoStack = StackMaker()
          }
          for (key in (attrsToSet || {})) {
            attributes[key] = attrsToSet[key]
          }
          return obj
        },
        undo: function () {
          if (!undoStack.isEmpty()) {
            redoStack.push(shallowCopy(attributes));
            attributes = undoStack.pop()
          }
          return obj
        },
        redo: function () {
          if (!redoStack.isEmpty()) {
            undoStack.push(shallowCopy(attributes));
            attributes = redoStack.pop()
          }
          return obj
        },
        get: function (key) {
          return attributes(key)
        },
        has: function (key) {
          return attributes.hasOwnProperty(key)
```

```
      },
      attributes: function {
        shallowCopy(attributes)
      }
    };
  return obj
};
```

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:

```
var QueueMaker = function () {
  var array = [],
      head = 0,
      tail = -1;
  return {
    pushTail: function (value) {
      return array[tail += 1] = value
    },
    pullHead: function () {
      var value;

      if (tail >= head) {
        value = array[head];
        array[head] = void 0;
        head += 1;
        return value
      }
    },
    isEmpty: function () {
      return tail < head
    }
  }
};
```

Now we wish to create a deque by adding pullTail and pushHead operations to our queue.[4] 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:

```
var QueueMaker = function () {
  var queue = {
    array: [],
    head: 0,
    tail: -1,
    pushTail: function (value) {
      return queue.array[queue.tail += 1] = value
    },
    pullHead: function () {
      var value;

      if (queue.tail >= queue.head) {
        value = queue.array[queue.head];
        queue.array[queue.head] = void 0;
        queue.head += 1;
        return value
      }
    },
    isEmpty: function () {
      return queue.tail < queue.head
    }
  };
  return queue
};
```

Now we can extend a queue into a deque:

```
var DequeMaker = function () {
  var deque = QueueMaker(),
      INCREMENT = 4;

  return extend(deque, {
    size: function () {
      return deque.tail - deque.head + 1
    },
    pullTail: function () {
      var value;

```

```
      if (!deque.isEmpty()) {
        value = deque.array[deque.tail];
        deque.array[deque.tail] = void 0;
        deque.tail -= 1;
        return value
      }
    },
    pushHead: function (value) {
      var i;

      if (deque.head === 0) {
        for (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:

```
var QueueMaker = function () {
  var queue = {
    array: [],
    head: 0,
    tail: -1,
    pushTail: function (value) {
      return queue.array[queue.tail += 1] = value
    },
    pullHead: function () {
      var value;

```

```
      if (queue.tail >= queue.head) {
        value = queue.array[queue.head];
        queue.array[queue.head] = void 0;
        queue.head += 1;
        return value
      }
    },
    isEmpty: function () {
      return queue.tail < queue.head
    }
  };
  return queue
};

queue = QueueMaker()
queue.pushTail('Hello')
queue.pushTail('JavaScript')
```

Let's make a copy of our queue using the extend recipe:

```
copyOfQueue = extend({}, 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 (var 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:

```
var AmnesiacQueueMaker = function () {
  return {
    array: [],
    head: 0,
    tail: -1,
    pushTail: function (myself, value) {
      return myself.array[myself.tail += 1] = value
    },
    pullHead: function (myself) {
      var value;

      if (myself.tail >= myself.head) {
        value = myself.array[myself.head];
        myself.array[myself.head] = void 0;
        myself.head += 1;
        return value
      }
    },
    isEmpty: function (myself) {
      return myself.tail < myself.head
    }
  }
};

queueWithAmnesia = AmnesiacQueueMaker();
queueWithAmnesia.pushTail(queueWithAmnesia, 'Hello');
queueWithAmnesia.pushTail(queueWithAmnesia, 'JavaScript')
```

The AmnesiacQueueMaker 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:

```
BanksQueueMaker = function () {
  return {
    array: [],
    head: 0,
    tail: -1,
    pushTail: function (value) {
      return this.array[this.tail += 1] = value
    },
    pullHead: function () {
      var value;

      if (this.tail >= this.head) {
        value = this.array[this.head];
        this.array[this.head] = void 0;
        this.head += 1;
        return value
      }
    },
    isEmpty: function () {
      return this.tail < this.head
    }
  }
};

banksQueue = BanksQueueMaker();
banksQueue.pushTail('Hello');
banksQueue.pushTail('JavaScript')
```

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.[5] Now we can easily make copies:

```
copyOfQueue = extend({}, banksQueue)
```

```
copyOfQueue.array = []
for (var i = 0; i < 2; ++i) {
  copyOfQueue.array[i] = banksQueue.array[i]
}

banksQueue.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.

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 called as an object method, the name this is bound in its environment to the object acting as a "receiver." For example:

```
var someObject = {
  returnMyThis: function () {
    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.[6] 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.

A function's context cannot be determined by examining the source code of a JavaScript program. Let's look at our example again:

```
var someObject = {
  someFunction: function () {
    return this;
  }
};

someObject.someFunction() === someObject
  //=> true
```

What is the context of the function someObject.someFunction? Don't say someObject! Watch this:

```
var someFunction = someObject.someFunction;

someFunction === someObject.someFunction
  //=> true

someFunction() === someObject
  //=> false
```

It gets weirder:

```
var 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

var name = 'someFunction';

someObject[name]() === someObject
  //=> true
```

Interesting!

```
var baz;

(baz = someObject.someFunction)() === this
  //=> true
```

How about:

```
var 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`.

```
var returnThis = function () { return this };

var aThirdObject = {
  someFunction: function () {
    return returnThis()
  }
}

returnThis() === this
  //=> true

```

```
12 aThirdObject.someFunction() === this
13   //=> 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:

```
1 returnThis() === aThirdObject
2   //=> false
3
4 returnThis.call(aThirdObject) === aThirdObject
5   //=> true
6
7 anotherObject.someFunction.call(someObject) === someObject
8   //=> 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:

```
1 var a = [1,2,3],
2     b = [4,5,6];
3
4 a.concat([2,1])
5   //=> [1,2,3,2,1]
6
7 a.concat.call(b,[2,1])
8   //=> [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).

apply, arguments, and contextualization

JavaScript has another automagic binding in every function's environment. `arguments` is a special object that behaves a little like an array.[7]

For example:

```
var third = function () {
  return arguments[2]
}

third(77, 76, 75, 74, 73)
  //=> 75
```

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:

```
var 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:

```
var contextualize = function (fn, context) {
  return function () {
    return fn.apply(context, arguments);
  }
}

accrete = contextualize(a.concat, a);
accrete([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:

```
function maybe (fn) {
  return function (argument) {
    if (argument != null) {
```

```
    return fn(argument)
  }
 }
}
```

This version doesn't preserve the context, so it can't be used as a method decorator. Instead, we have to write:

```
function maybe (fn) {
  return function (argument) {
    if (argument != null) {
      return fn.call(this, argument)
    }
  }
}
```

Now we can write things like:

```
someObject = {
  setSize: maybe(function (size) {
    this.size = size;
    return this
  })
}
```

And this is correctly set:

```
someObject.setSize(5)
  //=> { 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.

"A language construct that facilitates the bundling of data with the methods (or other functions) operating on that data."–Wikipedia ↩

Yes, there's another way to track the size of the array, but we don't need it to demonstrate encapsulation and hiding of state. ↩

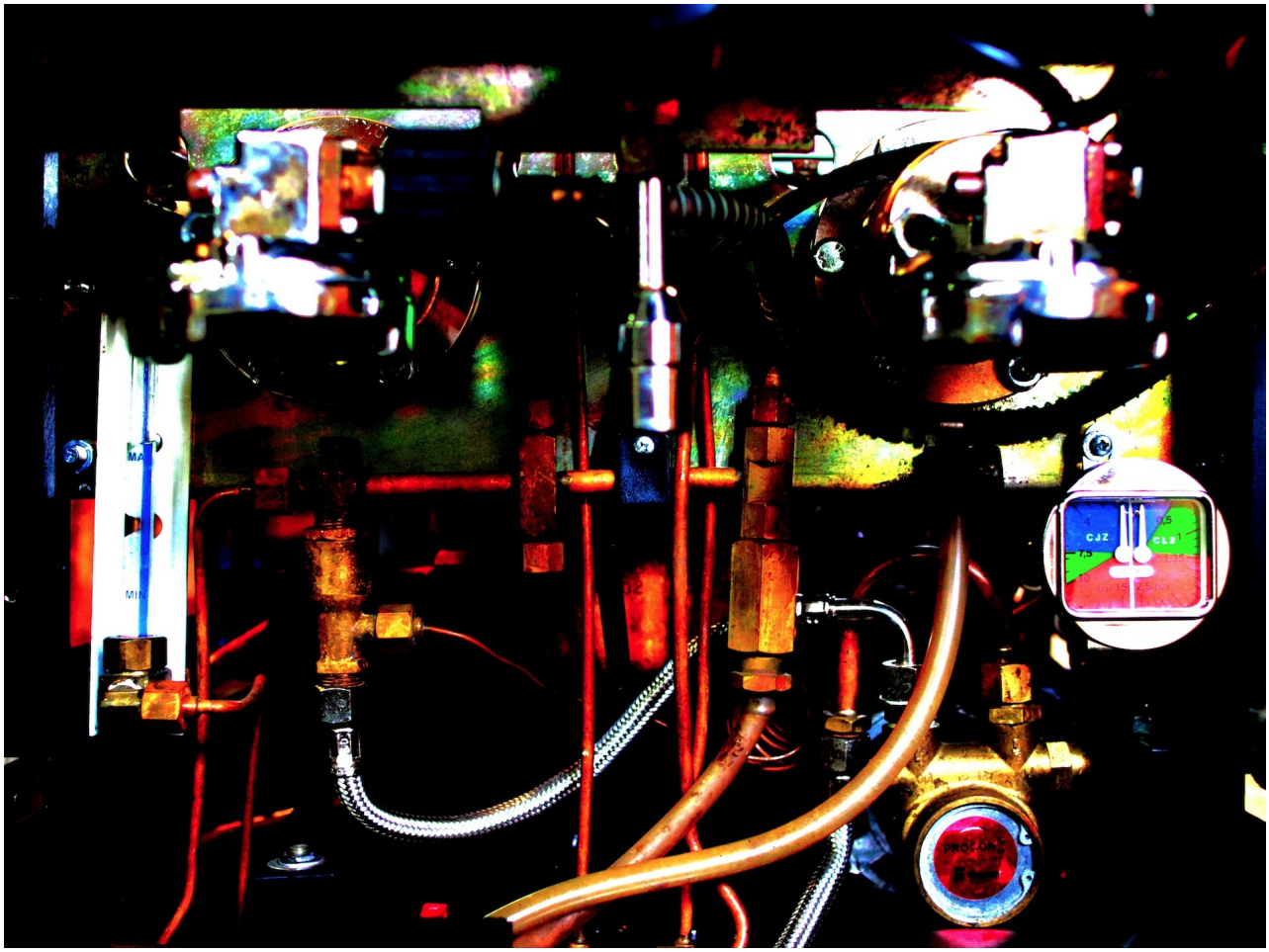
And when you take an already factored component and rearrange things so that it is factored into a different set of subcomponents without altering its behaviour, you are refactoring. ↩

Before 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." ↩

JavaScript also does other things with `this` as well, but this is all we care about right now. ↩

Too bad the language binds the context to the name `this` instead of the name `context`! ↩

Just enough to be frustrating, to be perfectly candid! ↩



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:

```
var fibonacci = function (n) {
  if (n < 2) {
    return n
  }
  else {
    return fibonacci(n-2) + fibonacci(n-1)
  }
```

```
8   }
```

We'll time it:

```
1 s = (new Date()).getTime()
2 fibonacci(45)
3 ( (new Date()).getTime() - s ) / 1000
4   //=> 28.565
```

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:

```
1 function memoized (fn, keymaker) {
2   var lookupTable = {},
3       key;
4
5   keymaker || (keymaker = function (args) {
6     return JSON.stringify(args)
7   });
8
9   return function () {
10    var key = keymaker.call(this, arguments);
11
12    return lookupTable[key] || (
13      lookupTable[key] = fn.apply(this, arguments)
14    )
15  }
16 }
```

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:

```
1 var fastFibonacci = memoized( function (n) {
2   if (n < 2) {
3     return n
```

```
  }
  else {
    return 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, you can supply it as a second parameter.

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:

```
function getwith (attr) {
  return function (object) { return object[attr]; }
}
```

You can use it like this:

```
var 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:

```
var 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(function (inventory) { return 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:

```
function get (object, attr) {
  return 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:

```
function pluckWith (attr) {
  return mapWith(getWith(attr))
}
```

Or even better:

```
var pluckWith = compose(mapWith, getWith);
```

And now we can write:

```
pluckWith('eggs')(inventories)
  //=> [ 36, 12, 42 ]
```

Libraries like Underscore provide pluck, the flipped version of pluckWith:

```
_.pluck(inventories, 'eggs')
  //=> [ 36, 12, 42 ]
```

Our recipe is terser when you want to name a function:

```
var eggsByStore = pluckWith('eggs');
```

vs.

```
function eggsByStore (inventories) {
  return _.pluck(inventories, 'eggs')
}
```

And of course, if we have pluck we can use flip to derive pluckWith:

```
var 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.

```
var 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 mapWiths, 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:

```
function deepMapWith (fn) {
  return function innerdeepMapWith (tree) {
    return Array.prototype.map.call(tree, function (element) {
      if (Array.isArray(element)) {
        return innerdeepMapWith(element);
      }
      else return 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,
```

```
20              9.99,
21              9.99 ] ]
```

Other languages call their objects “beans,” but serve extra-weak coffee in an attempt to be all things to all people

As discussed in Rebinding and References and again in Encapsulating State, JavaScript objects are very simple, yet the combination of objects, functions, and closures can create powerful data structures. That being said, there are language features that cannot be implemented with Plain Old JavaScript Objects, functions, and closures[1].

One of them is inheritance. In JavaScript, inheritance provides a cleaner, simpler mechanism for extending data structures, domain models, and anything else you represent as a bundle of state and operations.

Prototypes are Simple, it’s the Explanations that are Hard To Understand

As you recall from our code for making objects extensible, we wrote a function that returned a Plain Old JavaScript Object. The colloquial term for this kind of function is a “Factory Function.”

Let’s strip a function down to the very bare essentials:

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

This doesn't look like a factory function: 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 `var i = new Ur()`, we say that `i` is an instance of `Ur`.

For reasons that will be explained after we've discussed prototypes, 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.

prototypes

There's more. Here's something you may not know about functions:

```
Ur.prototype
  //=> {}
```

What's this prototype? Let's run our standard test:

```
(function () {}).prototype === (function () {}).prototype
  //=> false
```

Every function is initialized with its own unique `prototype`. What does it do? Let's try something:

```
Ur.prototype.language = 'JavaScript';
```

```
2
3 var continent = new Ur();
4   //=> {}
5 continent.language
6   //=> 'JavaScript'
```

That's very interesting! Instances seem to behave as if they had the same elements as their constructor's prototype. Let's try a few things:

```
1 continent.language = 'CoffeeScript';
2 continent
3   //=> {language: 'CoffeeScript'}
4 continent.language
5   //=> 'CoffeeScript'
6 Ur.prototype.language
7   '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.

```
1 var another = new Ur();
2   //=> {}
3 another.language
4   //=> 'JavaScript'
```

New instances don't acquire any changes made to other instances. Makes sense. And:

```
1 Ur.prototype.language = 'Sumerian'
2 another.language
3   //=> 'Sumerian'
```

Even more interesting: Changing the constructor's prototype changes the behaviour of all of its instances. This strongly implies that there is a dynamic relationship between instances and their constructors, rather than some kind of mechanism that makes objects by copying.[2]

Speaking of prototypes, here's something else that's very interesting:

```
1 continent.constructor
2   //=> [Function]
3
4 continent.constructor === Ur
```

```
5   //=> true
```

Every instance acquires a constructor element that is initialized to their constructor. This is true even for objects we don't create with new in our own code:

```
1 {}.constructor
2   //=> [Function: Object]
```

If that's true, what about prototypes? Do they have constructors?

```
1 Ur.prototype.constructor
2   //=> [Function]
3 Ur.prototype.constructor === Ur
4   //=> true
```

Very interesting! We will take another look at the constructor element when we discuss class extension.

revisiting this idea of queues

Let's rewrite our Queue to use new and .prototype, using this and our extend helper from Composition and Extension:

```
1 var Queue = function () {
2   extend(this, {
3     array: [],
4     head: 0,
5     tail: -1
6   })
7 };
8
9 extend(Queue.prototype, {
10   pushTail: function (value) {
11     return this.array[this.tail += 1] = value
12   },
13   pullHead: function () {
14     var value;
15
16     if (!this.isEmpty()) {
17       value = this.array[this.head]
18       this.array[this.head] = void 0;
19       this.head += 1;
20       return value
21     }
22   },
```

```
  isEmpty: function () {
    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.

objects everywhere?

Now that you know about prototypes, 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 its constructor's 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 invoked 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!

impostors

You may have noticed that we use “weasel words” to describe how everything in JavaScript behaves like an instance. Everything behaves as if it was created by a function with 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 constructors. They aren’t instances of some constructor.

So. Value types don’t have methods or constructors. And yet:

```
"Spence Olham".split(' ')
  //=> ["Spence", "Olham"]
```

Somehow, when we write `"Spence Olham".split(' ')`, the string `"Spence Olham"` isn’t an instance, it doesn’t have methods, but it does a damn fine job of impersonating an instance of a `String` constructor. How does `"Spence Olham"` impersonate an instance?

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,[3] 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.

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. Let’s try it:

```
new String("Spence Olham")
  //=> "Spence Olham"
```

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:

```
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:

```
var 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.

Binding Functions to Contexts

Recall that in What Context Applies When We Call a Function?, we adjourned our look at setting the context of a function with a look at a contextualize helper function:

```
var contextualize = function (fn, context) {
  return function () {
    return fn.apply(context, arguments)
  }
},
a = [1,2,3],
accrete = contextualize(a.concat, a);

accrete([4,5])
  //=> [ 1, 2, 3, 4, 5 ]
```

How would this help us in a practical way? Consider building an event-driven application. For example, an MVC application would bind certain views to update events when their models change. The Backbone framework uses events just like this:

```
var someView = ...,
    someModel = ...;

someModel.on('change', function () {
```

```
5   someView.render()
6 });
```

This tells someModel that when it invoked a change event, it should call the anonymous function that in turn invoked someView's .render method. Wouldn't it be simpler to simply write:

```
1 someModel.on('change', someView.render);
```

It would, except that the implementation for .on and similar framework methods looks something like this:

```
1 Model.prototype.on = function (eventName, callback) { ... callback() ... }
```

Although someView.render() correctly sets the method's context as someView, callback() will not. What can we do without wrapping someView.render() in a function call as we did above?

binding methods

Before enumerating approaches, let's describe what we're trying to do. We want to take a method call and treat it as a function. Now, methods are functions in JavaScript, but as we've learned from looking at contexts, method calls involve both invoking a function and setting the context of the function call to be the receiver of the method call.

When we write something like:

```
1 var unbound = someObject.someMethod;
```

We're binding the name unbound to the method's function, but we aren't doing anything with the identity of the receiver. In most programming languages, such methods are called "unbound" methods because they aren't associated with, or "bound" to the intended receiver.

So what we're really trying to do is get ahold of a bound method, a method that is associated with a specific receiver. We saw an obvious way to do that above, to wrap the method call in another function. Of course, we're responsible for replicating the arity of the method being bound. For example:

```
1 var boundSetter = function (value) {
2   return someObject.setSomeValue(value);
3 };
```

Now our bound method takes one argument, just like the function it calls. We can use a bound method anywhere:

```
someDomField.on('update', boundSetter);
```

This pattern is very handy, but it requires keeping track of these bound methods. One thing we can do is bind the method "in place," using the let pattern like this:

```
someObject.setSomeValue = (function () {
  var unboundMethod = someObject.setSomeValue;

  return function (value) {
    return unboundMethod.call(someObject, value);
  }
})();
```

Now we know where to find it:

```
someDomField.on('update', someObject.setSomeValue);
```

This is a very popular pattern, so much so that many frameworks provide helper functions to make this easy. Underscore, for example, provides _.bind to return a bound copy of a function and _.bindAll to bind methods in place:

```
// bind *all* of someObject's methods in place
_.bindAll(someObject);

// bind setSomeValue and someMethod in place
_.bindAll(someObject, 'setSomeValue', 'someMethod');
```

There are two considerations to ponder. First, we may be converting an instance method into an object method. Specifically, we're creating an object method that is bound to the object.

Most of the time, the only change this makes is that it uses slightly more memory (we're creating an extra function for each bound method in each object). But if you are a little more dynamic and actually change methods in the prototype, your changes won't "override" the object methods that you created. You'd have to roll your own binding method that refers to the prototype's method dynamically or reorganize your code.

This is one of the realities of "meta-programming." Each technique looks useful and interesting in isolation, but when multiple techniques are used together, they can have

unpredictable results. It's not surprising, because most popular languages consider classes and methods to be fairly global, and they handle dynamic changes through side-effects. This is roughly equivalent to programming in 1970s-era BASIC by imperatively changing global variables.

If you aren't working with old JavaScript environments in non-current browsers, you needn't use a framework or roll your own binding functions: JavaScript has a .bind method defined for functions:

```
1 someObject.someMethod = someObject.someMethod.bind(someObject);
```

.bind also does some currying for you, you can bind one or more arguments in addition to the context. For example:

```
1 AccountModel.prototype.getBalancePromise(forceRemote) = {
2   // if forceRemote is true, always goes to the remote
3   // database for the most real-time value, returns
4   // a promise.
5 };
6
7 var account = new AccountModel(...);
8
9 var boundGetRemoteBalancePromise = account.
10   getBalancePromise.
11   bind(account, true);
```

Very handy, and not just for binding contexts!

Getting the context right for methods is essential. The commonplace terminology is that we want bound methods rather than unbound methods. Current flavours of JavaScript provide a .bind method to help, and frameworks like Underscore also provide helpers to make binding methods easy.

Partial Application, Binding, and Currying

Now that we've seen how function contexts work, we can revisit the subject of partial application. Recall our recipe for a generalized left partial application:

```
1 var callLeft = variadic( function (fn, args) {
2   return variadic( function (remainingArgs) {
```

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

Function.prototype.bind can sometimes be used to accomplish the same thing, but will be much faster. For example, instead of:

```
function add (verb, a, b) {
  return "The " + verb + " of " + a + ' and ' + b + ' is ' + (a + b)
}

var sumFive = callLeft(add, 'sum', 5);

sumFive(6)
  //=> 'The sum of 5 and 6 is 11'
```

You can write:

```
var totalSix = add.bind(null, 'total', 6);

totalSix(5)
  //=> 'The total of 6 and 5 is 11'
```

The catch is the first parameter to .bind: It sets the context. If you write functions that don't use the context, like our .add, You can use .bind to do left partial application. But if you want to partially apply a method or other function where the context must be preserved, you can't use .bind. You can use the recipes given in JavaScript Allongé because they preserve the context properly.

Typically, context matters when you want to perform partial application on methods. So for an extremely simple example, we often use Array.prototype.slice to convert arguments to an array. So instead of:

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

var array = __slice.call(arguments, 0);
```

We could write:

```
var __copy = callFirst(Array.prototype.slice, 0);

var array = __copy.call(arguments)
```

The other catch is that .bind only does left partial evaluation. If you want to do right partial application, you'll need callLast or callRight.

currying

The terms "partial application" and "currying" are closely related but not synonymous. Currying is the act of taking a function that takes more than one argument and converting it to an equivalent function taking one argument. How can such a function be equivalent? It works provided that it returns a partially applied function.

Code is, as usual, much clearer than words. Recall:

```
function add (verb, a, b) {
  return "The " + verb + " of " + a + ' and ' + b + ' is ' + (a + b)
}

add('sum', 5, '6')
  //=> 'The sum of 5 and 6 is 11'
```

Here is the curried version:

```
function addCurried (verb) {
  return function (a) {
    return function (b) {
      return "The " + verb + " of " + a + ' and ' + b + ' is ' + (a + b)
    }
  }
}

addCurried('total')(6)(5)
  //=> 'The total of 6 and 5 is 11'
```

Currying by hand would be an incredible effort, but its close relationship with partial application means that if you have left partial application, you can derive currying. Or if you have currying, you can derive left partial application. Let's derive currying from callFirst. Recall:

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

function callFirst (fn, larg) {
  return function () {
    var args = __slice.call(arguments, 0);

    return fn.apply(this, [larg].concat(args))
```

```
  }
}
```

Here's a function that curries any function with two arguments:

```
function curryTwo (fn) {
  return function (x) {
    return callFirst(fn, x)
  }
}

function add2 (a, b) { return a + b }

curryTwo(add2)(5)(6)
  //=> 11
```

And from there we can curry a function with three arguments:

```
function curryThree (fn) {
  return function (x) {
    return curryTwo(callFirst(fn, x))
  }
}

function add3 (verb, a, b) {
  return "The " + verb + " of " + a + ' and ' + b + ' is ' + (a + b)
}

curryThree(add3)('sum')(5)(6)
  //=> 'The sum of 5 and 6 is 11'
```

We'll develop a generalized curry function in the recipes. But to summarize the difference between currying and partial application, currying is an operation that transforms a function taking two or more arguments into a function that takes a single argument and partially applies it to the function and then curries the rest of the arguments.

A Class By Any Other Name

JavaScript has "classes," for some definition of "class." You've met them already, they're constructors that are designed to work with the `new` keyword and have behaviour in their `.prototype` element. You can create one any time you like by:

1. Writing the constructor so that it performs any initialization on `this`, and:
2. Putting all of the method definitions in its prototype.

Let's see it again: Here's a class of todo items:

```
function Todo (name) {
  this.name = name || 'Untitled';
  this.done = false;
};

Todo.prototype.do = function () {
  this.done = true;
};

Todo.prototype.undo = function () {
  this.done = false;
};
```

You can mix other functionality into this class by extending the prototype with an object:

```
extend(Todo.prototype, {
  prioritize: function (priority) {
    this.priority = priority;
  };
});
```

Naturally, that allows us to define mixins for other classes:

```
var ColourCoded = {
  setColourRGB: function (r, g, b) {
    // ...
  },
  getColourRGB: function () {
    // ...
  },
  setColourCSS: function (css) {
    // ...
  },
  getColourCSS: function () {
    // ...
  }
};

extend(Todo.prototype, ColourCoded);
```

This does exactly the same thing as declaring a "class," defining a "method," and adding a "mixin." How does it differ? It doesn't use the words class, method, def(ine) or mixin. And it has this prototype property that most other popular languages eschew. It also doesn't deal

with inheritance, a deal-breaker for programmers who are attached to taxonomies.

For these reasons, many programmers choose to write their own library of functions to mimic the semantics of other programming languages. This has happened so often that most of the popular utility-belt frameworks like Backbone have some form of support for defining or extending classes baked in.

Nevertheless, JavaScript right out of the box has everything you need for defining classes, methods, mixins, and even inheritance (as we'll see in Extending Classes with Inheritance). If we choose to adopt a library with more streamlined syntax, it's vital to understand JavaScript's semantics well enough to know what is happening "under the hood" so that we can work directly with objects, functions, methods, and prototypes when needed.

> One note of caution: A few libraries, such as the vile creation YouAreDaChef, manipulate JavaScript such that ordinary programming such as extending a prototype either don't work at all or break the library's abstraction. Think long and carefully before adopting such a library. The best libraries "Cut with JavaScript's grain."

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:

```
1 QueueMaker = function () {
2   return {
3     array: [],
4     head: 0,
5     tail: -1,
```

```
    pushTail: function (value) {
      return this.array[this.tail += 1] = value
    },
    pullHead: function () {
      var value;

      if (this.tail >= this.head) {
        value = this.array[this.head];
        this.array[this.head] = void 0;
        this.head += 1;
        return value
      }
    },
    isEmpty: function () {
      return this.tail < this.head
    }
  }
};
```

pushTail, pullHead, and isEmpty are object methods. Also, from encapsulation:

```
var stack = (function () {
  var obj = {
    array: [],
    index: -1,
    push: function (value) {
      return obj.array[obj.index += 1] = value
    },
    pop: function () {
      var value = obj.array[obj.index];
      obj.array[obj.index] = void 0;
      if (obj.index >= 0) {
        obj.index -= 1
      }
      return value
    },
    isEmpty: function () {
      return 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`:

```
var WidgetModel = function (id, attrs) {
  extend(this, attrs || {});
  this.id = function () { return id }
}

extend(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.

Extending Classes with Inheritance

You recall from Composition and Extension that we extended a Plain Old JavaScript Queue to create a Plain Old JavaScript Deque. But what if we have decided to use JavaScript's prototypes and the `new` keyword instead of Plain Old JavaScript Objects? How do we extend a queue into a deque?

Here's our Queue:

```
var Queue = function () {
  extend(this, {
    array: [],
    head: 0,
    tail: -1
  })
};

extend(Queue.prototype, {
  pushTail: function (value) {
    return this.array[this.tail += 1] = value
  },
  pullHead: function () {
    var value;

    if (!this.isEmpty()) {
      value = this.array[this.head]
      this.array[this.head] = void 0;
      this.head += 1;
      return value
    }
  },
  isEmpty: function () {
    return this.tail < this.head
  }
});
```

And here's what our Deque would look like before we wire things together:

```
var Dequeue = function () {
  Queue.prototype.constructor.call(this)
};

Dequeue.INCREMENT = 4;

extend(Dequeue.prototype, {
  size: function () {
    return this.tail - this.head + 1
  },
  pullTail: function () {
    var value;

    if (!this.isEmpty()) {
      value = this.array[this.tail];
```

```
      this.array[this.tail] = void 0;
      this.tail -= 1;
      return value
    }
  },
  pushHead: function (value) {
    var i;

    if (this.head === 0) {
      for (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
  }
});
```

> We obviously want to do all of a Queue's initialization, thus we called Queue.prototype.constructor.call(this). But why not just call Queue.call(this)? As we'll see when we wire everything together, this ensures that we're calling the correct constructor even when Queue itself is wired to inherit from another constructor function.

So what do we want from dequeues such that we can call all of a Queue's methods as well as a Dequeue's? Should we copy everything from Queue.prototype into Deque.prototype, like extend(Deque.prototype, Queue.prototype)? That would work, except for one thing: If we later modified Queue, say by mixing in some new methods into its prototype, those wouldn't be picked up by Dequeue.

No, there's a better idea. Prototypes are objects, right? Why must they be Plain Old JavaScript Objects? Can't a prototype be an instance?

Yes they can. Imagine that Deque.prototype was a proxy for an instance of Queue. It would, of course, have all of a queue's behaviour through Queue.prototype. We don't want it to be an actual instance, mind you. It probably doesn't matter with a queue, but some of the things we might work with might make things awkward if we make random instances. A database connection comes to mind, we may not want to create one just for the convenience of having access to its behaviour.

Here's such a proxy:

```
var QueueProxy = function () {}

QueueProxy.prototype = Queue.prototype
```

Our QueueProxy isn't actually a Queue, but its prototype is an alias of Queue.prototype. Thus, it can pick up Queue's behaviour. We want to use it for our Deque's prototype. Let's insert that code in our class definition:

```
var Dequeue = function () {
  Queue.prototype.constructor.call(this)
};

Dequeue.INCREMENT = 4;

Dequeue.prototype = new QueueProxy();

extend(Dequeue.prototype, {
  size: function () {
    return this.tail - this.head + 1
  },
  pullTail: function () {
    var value;

    if (!this.isEmpty()) {
      value = this.array[this.tail];
      this.array[this.tail] = void 0;
      this.tail -= 1;
      return value
    }
  },
  pushHead: function (value) {
    var i;

    if (this.head === 0) {
      for (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
  }
});
```

And it seems to work:

```
d = new Dequeue()
d.pushTail('Hello')
d.pushTail('JavaScript')
d.pushTail('!')
d.pullHead()
  //=> 'Hello'
d.pullTail()
  //=> '!'
d.pullHead()
  //=> 'JavaScript'
```

Wonderful!

getting the constructor element right

How about some of the other things we've come to expect from instances?

```
d.constructor == Dequeue
  //=> false
```

Oops! Messing around with Dequeue's prototype broke this important equivalence. Luckily for us, the constructor property is mutable for objects we create. So, let's make a small change to QueueProxy:

```
var QueueProxy = function () {
  this.constructor = Dequeue;
}
QueueProxy.prototype = Queue.prototype
```

Repeat. Now it works:

```
d.constructor === Dequeue
  //=> true
```

The QueueProxy function now sets the constructor for every instance of a QueueProxy (hopefully just the one we need for the Dequeue class). It returns the object being created (it could also return undefined and work. But if it carelessly returned something else, that would be assigned to Dequeue's prototype, which would break our code).

extracting the boilerplate

Let's turn our mechanism into a function:

```
var child = function (parent, child) {
  var proxy = function () {
    this.constructor = child
  }
  proxy.prototype = parent.prototype;
  child.prototype = new proxy();
  return child;
}
```

And use it in Dequeue:

```
var Dequeue = child(Queue, function () {
  Queue.prototype.constructor.call(this)
});

Dequeue.INCREMENT = 4;

extend(Dequeue.prototype, {
  size: function () {
    return this.tail - this.head + 1
  },
  pullTail: function () {
    var value;

    if (!this.isEmpty()) {
      value = this.array[this.tail];
      this.array[this.tail] = void 0;
      this.tail -= 1;
      return value
    }
  },
  pushHead: function (value) {
    var i;

    if (this.head === 0) {
      for (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
  }
});
```

future directions

Some folks just **love** to build their own mechanisms. When all goes well, they become famous as framework creators and open source thought leaders. When all goes badly they create in-house proprietary one-offs that blur the line between application and framework with abstractions everywhere.

If you're keen on learning, you can work on improving the above code to handle extending constructor properties, automatically calling the parent constructor function, and so forth. Or you can decide that doing it by hand isn't that hard so why bother putting a thin wrapper around it?

It's up to you, while JavaScript isn't the tersest language, it isn't so baroque that building inheritance ontologies requires hundreds of lines of inscrutable code.

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.
- Instances can override their constructor's prototype without altering it.
- The relationship between instances and their constructor's prototype is dynamic.
- `this` works seamlessly with methods defined in prototypes.
- Everything behaves like an object.
- JavaScript can convert primitives into instances and back into primitives.
- Object methods are typically created in the constructor and are private to each object.
- Prototypes can be chained to allow extension of instances.

And most importantly:

- JavaScript has classes and methods, they just aren't formally called classes and methods in the language's syntax.

Since the JavaScript that we have presented so far is computationally universal, it is possible to perform any calculation with its existing feature set, including emulating any other programming language. Therefore, it is not theoretically necessary to have any further language features; If we need macros, continuations, generic functions, static typing, or anything else, we can greenspun them ourselves. In practice, however, this is buggy, inefficient, and presents our fellow developers with serious challenges understanding our code. ↩

For many programmers, the distinction between a dynamic relationship and a copying mechanism is too fine to worry about. However, it makes many dynamic program modifications possible.↩

Recall that Strings, Numbers, Booleans and so forth are value types and primitives. We're calling them primitives here.↩

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.

Currying

We discussed currying in Closures and Partial Application, Binding, and Currying. Here is the recipe for a higher-order function that curries its argument function. It works with any function that has a fixed length, and it lets you provide as many arguments as you like.

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

function curry (fn) {
  var arity = fn.length;

  return given([]);

  function given (argsSoFar) {
    return function helper () {
      var updatedArgsSoFar = argsSoFar.concat(__slice.call(arguments, 0));

      if (updatedArgsSoFar.length >= arity) {
        return fn.apply(this, updatedArgsSoFar)
      }
      else return given(updatedArgsSoFar)
    }
  }

}

function sumOfFour (a, b, c, d) { return a + b + c + d }

var curried = curry(sumOfFour);

curried(1)(2)(3)(4)
  //=> 10

curried(1,2)(3,4)
  //=> 10

curried(1,2,3,4)
  //=> 10
```

We saw earlier that you can derive a curry function from a partial application function. The reverse is also true:

```
function callLeft (fn) {
  return curry(fn).apply(null, __slice.call(arguments, 1))
}

```

```
callLeft(sumOfFour, 1)(2, 3, 4)
  //=> 10

callLeft(sumOfFour, 1, 2)(3, 4)
  //=> 10
```

(This is a little different from the previous left partial functions in that it returns a curried function).

Bound

Earlier, we saw a recipe for getWith that plays nicely with properties:

```
function get (attr) {
  return function (obj) {
    return obj[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
}

var 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(
  function (unboundmethod) {
    return unboundmethod()
  }
)
  //=> TypeError: Cannot read property 'eggs' of undefined
```

Of course, these are unbound methods we're "getting" from each object. Here's a new version of `get` that plays nicely with methods. It uses variadic:

```
var bound = variadic( function (messageName, args) {

  if (args.length === 0) {
    return function (instance) {
      return instance[messageName].bind(instance)
    }
  }
  else {
    return function (instance) {
      return Function.prototype.bind.apply(
        instance[messageName], [instance].concat(args)
      )
    }
  }
});

mapWith(bound('eggs'))(inventories).map(
  function (boundmethod) {
    return 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:

```
return function (instance) {
  return 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(
  function (boundmethod) {
    return boundmethod()
  }
)
  //=> [ { record:
  //       { apples: 0,
  //         oranges: 144,
  //         eggs: 48 } },
  //     { record:
  //       { apples: 240,
  //         oranges: 54,
  //         eggs: 24 } },
  //     { record:
  //       { apples: 24,
  //         oranges: 12,
  //         eggs: 54 } } ]
```

Unbinding

One of the specifications for Function.prototype.bind is that it creates a binding that cannot be overridden. In other words:

```
function myName () { return this.name }

var harpo   = { name: 'Harpo' },
```

```
    chico   = { name: 'Chico' },
    groucho = { name: 'Groucho' };

var fh = myName.bind(harpo);
fh()
  //=> 'Harpo'

var fc = myName.bind(chico);
fc()
  //=> 'Chico'
```

This looks great. But what happens if we attempt to **re**-bind a bound function, either with bind or .call?

```
var fhg = fh.bind(groucho);
fhg()
  //=> 'Harpo'

fc.call(groucho)
  //=> 'Chico'

fh.apply(groucho, [])
  //=> 'Harpo'
```

Bzzt! You cannot override the context of a function that has already been bound, even if you're creating a new function with .bind. You also don't want to roll your own bind function that allows rebinding, lest you be bitten by someone else's code that expects that a bound function cannot be rebound. (One such case is where bound functions–such as callbacks–are stored in an array. Evaluating callbacks[index]() will override the bound context with the array unless the context cannot be overridden.)[1]

the recipe

Our version of bind memoizes the original function so that you can later call unbind to restore it for rebinding.

```
var unbind = function unbind (fn) {
  return fn.unbound ? unbind(fn.unbound()) : fn
};

function bind (fn, context, force) {
  var unbound, bound;

  if (force) {
    fn = unbind(fn)
```

```
  }
  bound = function () {
    return fn.apply(context, arguments)
  };
  bound.unbound = function () {
    return fn;
  };

  return bound;
}

function myName () { return this.name }

var harpo   = { name: 'Harpo' },
    chico   = { name: 'Chico' },
    groucho = { name: 'Groucho' };

var fh = bind(myName, harpo);
fh()
  //=> 'Harpo'

var fc = bind(myName, chico);
fc()
  //=> 'Chico'

var fhg = bind(fh, groucho);
fhg()
  //=> 'Harpo'

var fhug = bind(fh, groucho, true);
fhug()
  //=> 'Groucho'

var fhug2 = bind(unbind(fh), groucho);
fhug()
  //=> 'Groucho'

fc.unbound().call(groucho)
  //=> 'Groucho'

unbind(fh).apply(groucho, [])
  //=> 'Groucho'
```

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(
  function (boundmethod) {
    return boundmethod()
  }
)
  //=> [ 36, 12, 42 ]
```

As we noted, it's ugly to write

```
function (boundmethod) {
  return boundmethod()
}
```

So instead, we write a new recipe:

```
var send = variadic( function (args) {
  var fn = bound.apply(this, args);

  return function (instance) {
    return fn(instance)();
  }
})

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.

Here's a robust version that doesn't rely on bound:

```
var send = variadic( function (methodName, leftArguments) {
  return variadic( function (receiver, rightArguments) {
    return receiver[methodName].apply(receiver, leftArguments.concat(rightArgume\
nts))
  })
});
```

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. 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:

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

function invoke (fn) {
  var args = __slice.call(arguments, 1);

  return function (instance) {
    return 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:

```
var data = [
  { 0: 'zero',
    1: 'one',
    2: 'two',
    foo: 'foo',
    length: 3 },
  // ...
];
```

We can use the pattern from Partial Application, Binding, and Currying to create a context-dependent copy function:

```
var __copy = callFirst(Array.prototype.slice, 0);
```

And now we can compose mapWith with invoke to convert the data to arrays:

```
mapWith(invoke(__copy))(data)
```

```
2   //=> [
3   //     [ 'zero', 'one', 'two' ],
4   //     // ...
5   //   ]
```

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:

```
1 function instanceEval (instance) {
2   return function (fn) {
3     var args = __slice.call(arguments, 1);
4
5     return fn.apply(instance, args)
6   }
7 }
8
9 var args = instanceEval(arguments)(__slice, 0);
```

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:

```
1  function Cake () {}
2
3  extend(Cake.prototype, {
4    setFlavour: function (flavour) {
5      return this.flavour = flavour
6    },
7    setLayers: function (layers) {
8      return this.layers = layers
9    },
10   bake: function () {
11     // do some baking
12   }
13 });
14
15 var cake = new Cake();
16 cake.setFlavour('chocolate');
17 cake.setLayers(3);
18 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 fluent 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), so the rule is to return the receiver unless the method is a query:

```
function Cake () {}

extend(Cake.prototype, {
  setFlavour: function (flavour) {
    this.flavour = flavour;
    return this
  },
  setLayers: function (layers) {
    this.layers = layers;
    return this
  },
  bake: function () {
    // do some baking
    return this
  }
});
```

The code to work with cakes is now easier to read and less repetitive:

```
var 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.

The fluent method decorator solves this problem:

```
function fluent (methodBody) {
  return function () {
    methodBody.apply(this, arguments);
    return this;
  }
}
```

Now you can write methods like this:

```
Cake.prototype.bake = fluent( function () {
  // do some baking
  // using many lines of code
  // and possibly multiple returns
});
```

It's obvious at a glance that this method is "fluent."

Once Again

As we noted when we saw the recipe for once, you do have to be careful that you are calling the function once returns multiple times. If you keep calling once, you'll get a new function that executes once, so you'll keep calling your function:

```
once(function () {
  return 'sure, why not?'
})()
  //=> 'sure, why not?'

once(function () {
  return 'sure, why not?'
})()
  //=> 'sure, why not?'
```

This is expected, but sometimes not what we want. Instead of the simple implementation, we can use a named once:

```
function once (fn) {
  var done = false,
      testAndSet;

  if (!!fn.name) {
    testAndSet = function () {
      this["__once__"] || (this["__once__"] = {})
      if (this["__once__"][fn.name]) return true;
      this["__once__"][fn.name] = true;
      return false
    }
  }
  else  {
    testAndSet = function (fn) {
      if (done) return true;
```

```
      done = true;
      return false
    }
  }

  return function () {
    return testAndSet.call(this) ? void 0 : fn.apply(this, arguments)
  }
}
```

If you call this with just a function, it behaves exactly like our first recipe. But if you supply a named function, you get a different behaviour:

```
once(function askedOnDate () {
  return 'sure, why not?'
})()
  //=> 'sure, why not?'

once(function askedOnDate () {
  return 'sure, why not?'
})()
  //=> undefined
```

The named once adds a property, __once__, to the context where the function is called and uses it to keep track of which named functions have and haven't been run. One very powerful use is for defining object methods that should only be evaluated once, such as an initialization method. Normally this is done in the constructor, but you might write a "fluent" object that lets you call various setters:

```
function Widget () {};

Widget.prototype.setVolume = function setVolume (volume) {
  this.volume = volume;
  return this;
}

Widget.prototype.setDensity = function setDensity (density) {
  this.density = density;
  return this;
}

Widget.prototype.setLength = function setLength (length) {
  this.length = length;
  return this;
}
```

```
17
18 Widget.prototype.initialize = once(function initialize() {
19   // do some initialization
20   return this;
21 });
22
23 var w = new Widget()
24   .setVolume(...)
25   .setDensity(...)
26   .setLength(...)
27   .initialize();
```

If you later call `w.initialize()`, it won't be initialized again. You need a named `once`, because an ordinary `once` would be called once for every instance sharing the same prototype, whereas the named once will keep track of whether it has been run separately for each instance.

Caveat: Every instance will have a `__once__` property. If you later write code that iterates over every property, you'll have to take care not to interact with it.

Isnotlupus on Reddit suggested this line of thinking against "weak binding" functions.↩

Saltspring Island Roasting Facility

Throughout the book, we've looked at how functions work, and more importantly, the many ways they can be decomposed into smaller parts and recombined in different ways. In this chapter. we will step back and look at a larger pattern, the use of sequence to control the processing of information.

Introduction: Compose and Pipeline

In Combinators and Function Decorators, we saw the function compose:

```
1 function compose (a, b) {
2   return function (c) {
3     return a(b(c))
4   }
5 }
```

As we saw before, given:

```
1 function addOne (number) {
2   return number + 1
3 }
4
```

```
function double (number) {
  return number * 2
}
```

Instead of:

```
function doubleOfAddOne (number) {
  return double(addOne(number))
}
```

We could write:

```
var doubleOfAddOne = compose(double, addOne);
```

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:

```
var 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 simpler to read `var setter = compose(fluent, maybe);` than:

```
function setter (fn) {
  return 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:

```
1 var pipeline = flip(compose);
2
3 var 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

callbacks

pipeline and compose both work with functions that take an argument and return a value. In our next section, we'll discuss pipelining functions that invoke a callback rather than

returning a value.

The delight of coffee is that it transports you to another world

(this bonus chapter is a work-in-progress)

How Prototypes and Constructors differ from Classes

In the previous section, we said that JavaScript has "classes" for some definition of the word "class," and we showed how JavaScript provides many of the features found in other "object-oriented languages." For those who want a fuller explanation, this section goes into more detail about how JavaScript's "prototypes" differ from the classes found in a language like Ruby. It is not necessary to read this section to understand programming in JavaScript, but it can be helpful when discussing JavaScript with programmers who are more comfortable talking about classes.

Although each "object-oriented" programming language has its own particular set of semantics, the majority in popular use have "classes." A class is an entity responsible for creating objects and defining the behaviour of objects. Classes may be objects in their own right, but if they are, they're different from other types of objects. For example, the `String` class in Ruby is not itself a string, it's an object whose class is `Class`. All objects in a "classical" system have a class, and their class is a "class."

That sounds tautological, until we look at JavaScript. But let's start with a quick review of a popular classist language, Ruby.

ruby

In Ruby, classes are objects, but they're special objects. For example, here are some of the methods associated with the Ruby class String:

```
String.methods
  #=> [:try_convert, :allocate, :new, :superclass, :freeze, :===, :==,
       :<=>, :<, :<=, :>, :>=, :to_s, :included_modules, :include?, :name,
       :ancestors, :instance_methods, :public_instance_methods,
       :protected_instance_methods, :private_instance_methods, :constants,
       :const_get, :const_set, :const_defined?, :const_missing,
       :class_variables, :remove_class_variable, :class_variable_get,
       :class_variable_set, :class_variable_defined?, :public_constant,
       :private_constant, :module_exec, :class_exec, :module_eval, :class_eval,
       :method_defined?, :public_method_defined?, :private_method_defined?,
       :protected_method_defined?, :public_class_method, :private_class_method,
       # ...
       :!=, :instance_eval, :instance_exec, :__send__, :__id__]
```

And here are some of the methods associated with an instance of a string:

```
String.new.methods
  #=> [:<=>, :==, :===, :eql?, :hash, :casecmp, :+, :*, :%, :[],
       :[]=, :insert, :length, :size, :bytesize, :empty?, :=~,
       :match, :succ, :succ!, :next, :next!, :upto, :index, :rindex,
       :replace, :clear, :chr, :getbyte, :setbyte, :byteslice,
       :to_i, :to_f, :to_s, :to_str, :inspect, :dump, :upcase,
       :downcase, :capitalize, :swapcase, :upcase!, :downcase!,
       :capitalize!, :swapcase!, :hex, :oct, :split, :lines, :bytes,
       :chars, :codepoints, :reverse, :reverse!, :concat, :<<,
       :prepend, :crypt, :intern, :to_sym, :ord, :include?,
       :start_with?, :end_with?, :scan, :ljust, :rjust, :center,
       # ...
       :instance_eval, :instance_exec, :__send__, :__id__]
```

As you can see, a "class" in Ruby is very different from an "instance of that class." And the methods of a class are very different from the methods of an instance of that class.

Here's how you define a Queue in Ruby:

```
class Queue
  def initialize
    @array, @head, @tail = [], 0, -1
  end

```

```
  def pushTail value
    @array[@tail += 1] = value
  end

  def pullHead
    if !@isEmpty
      @array[@head]).tap { |value|
        @array[@head] = null
        @head += 1
      }
    end
  end

  def isEmpty
    !!(@tail < @head)
  end
end
```

There is special syntax for defining a class, and special syntax for defining the behaviour of instances. There are different ways of defining the way new instances are created in classist languages. Ruby uses a “magic method” called initialize. Now let’s look at JavaScript.

javascript has constructors and prototypes

JavaScript objects don’t have a formal class, and thus there’s no special syntax for defining how to create an instance or define its behaviour.

JavaScript instances are created with a constructor. The constructor of an instance is a function that was invoked with the new operator. In JavaScript, any function can be a constructor, even if it doesn’t look like one:

```
function square (n) { return n * n; }
  //=> undefined
square(2)
  //=> 4
square(2).constructor
  //=> [Function: Number]
new square(2)
  //=> {}
new square(2).constructor
  //=> [Function: square]
```

As you can see, the square function will act as a constructor if you call it with new. There is no special kind of thing that constructs new objects, every function is (potentially) a constructor.

That's different from a true classical language, where the class is a special kind of object that creates new instances.

How does JavaScript define the behaviour of instances? JavaScript doesn't have a special syntax or special kind of object for that, it has "prototypes." Prototypes are objects, but unlike a classical system, there are no special methods or properties associated with a prototype. Any object can be a prototype, even an empty object. In fact, that's exactly what is associated with a constructor by default:

```
function Nullo () {};
Nullo.prototype
  //=> {}
```

There's absolutely nothing special about a prototype object. No special class methods, no special constructor of its own, nothing. Let's look at a simple Queue in JavaScript:

```
var Queue = function () {
  this.array = [];
  this.head = 0;
  this.tail = -1;
};

Queue.prototype.pushTail = function (value) {
  return this.array[this.tail += 1] = value;
};
Queue.prototype.pullHead = function () {
  var value;

  if (!this.isEmpty()) {
    value = this.array[this.head];
    this.array[this.head] = void 0;
    this.head += 1;
    return value;
  }
};
Queue.prototype.isEmpty = function () {
  return this.tail < this.head;
};

Queue.prototype
  //=>  { pushTail: [Function],
  //      pullHead: [Function],
  //      isEmpty: [Function] }
```

The first way a prototype in JavaScript is different from a class in Ruby is that the

prototype is an ordinary object with exactly the same properties that we expect to find in an instance: Methods `pushTail`, `pullHead`, and `isEmpty`.

The second way is that any object can be a prototype. It can have functions (which act like methods), it can have other values (like numbers, booleans, objects, or strings). It can be an object you're using for something else: An account, a view, a DOM object if you're in the browser, anything.

"Classes" are objects in most "classical" languages, but they are a special kind of object. In JavaScript, prototypes are not a special kind of object, they're just objects.

summary of the difference between classes and prototypes

A class in a formal classist language can be an object, but it's a special kind of object with special properties and methods. It is responsible for creating new instances and for defining the behaviour of instances.

Instance behaviour in a classist language is defined with special syntax. If changes are allowed dynamically, they are done with special syntax and/or special methods invoked on the class.

JavaScript splits the responsibility for instances into a constructor and a prototype. A constructor in JavaScript can be any function. Constructors are responsible for creating new instances.

A prototype in JavaScript can be any object. Prototypes are responsible for defining the behaviour of instances. prototypes don't have special methods or properties.

Instance behaviour in JavaScript is defined by modifying the prototype directly, e.g. by adding functions to it as properties. There is no special syntax for defining a class or modifying a class.

so why does this book say that javascript has "classes" for some definition of "class?"

Because, **if**:

1. You use a function as a constructor, and;
2. You use a prototype for defining instance methods, and;
3. The prototype is used strictly for defining the instance methods and nothing else;

Then:

You will have something that works just like a simple class-based system, with the

constructor function and its prototype acting as the “class.”

But if you want more, you have a flexible system that does allow you to do much much more. It’s up to you.

New-Agnostic Constructors

JavaScript is inflexible about certain things. One of them is invoking `new` on a constructor. In many of our recipes, we can write functions that can handle a variable number of arguments and use `.apply` to invoke a function. For example:

```
function fluent (methodBody) {
  return function () {
    methodBody.apply(this, arguments);
    return this
  }
}
```

You can’t do the same thing with calling a constructor. This will not work:

```
function User (name, password) {
  this.name = name || 'Untitled';
  this.password = password
};

function withDefaultPassword () {
  var args = Array.prototype.slice.call(arguments, 0);
  args[1] = 'swordfish';
  return new User.apply(this, args);
}

withDefaultPassword('James')
  //=> TypeError: function apply() { [native code] } is not a constructor
```

Another weakness of constructors is that if you call them without using `new`, you usually get nonsense:

```
User('James', 'swordfish')
  //=> undefined
```

In David Herman’s Effective JavaScript, he describes the “New-Agnostic Constructor Pattern.” He gives several variations, but the simplest is this:

```
function User (name, password) {
  if (!(this instanceof User)) {
    return new User(name, password);
  }
  this.name = name || 'Untitled';
  this.password = password
};
```

Now you can call the constructor without the new keyword:

```
User('James', 'swordfish')
  //=> { name: 'James', password: 'swordfish' }
```

This in turn opens up the possibility of doing dynamic things with constructors that didn't work when you were forced to use new:

```
function withDefaultPassword () {
  var args = Array.prototype.slice.call(arguments, 0);
  args[1] = 'swordfish';
  return User.apply(this, args);
}

withDefaultPassword('James')
  //=> { name: 'James', password: 'swordfish' }
```

(The pattern above has a tradeoff: It works for all circumstances except when you want to set up an inheritance hierarchy.)

Another New-Agnostic Constructor Pattern

Here's another way to write a new-agnostic constructor:

```
function User (name, password) {
  var self = this instanceof User ? this : new User();
  if (name != null) {
    self.name = name;
    self.password = password;
  }
  return self;
};
```

The principle is that the constructor initializes an object assigned to the variable self and returns it. When you call the constructor with new, then self will be assigned the current

context. But if you call this constructor as a standard function, then it will call itself without parameters and assign the newly created User to self.

Mixins

In A Class By Any Other Name, we saw that you can emulate “mixins” using our extend function. 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. Instances of the class are created by calling the constructor with new. They “inherit” shared behaviour from the constructor’s prototype property. 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 an evolved class of todo items we saw earlier:

```
function Todo (name) {
  var self = this instanceof Todo
             ? this
             : new Todo();
  self.name = name || 'Untitled';
  self.done = false;
  return self;
};

Todo.prototype.do = fluent( function () {
  this.done = true;
});

Todo.prototype.undo = fluent( function () {
  this.done = false;
});

Todo.prototype;
  //=> { do: [Function], undo: [Function] }
```

And a “mixin:”

```
var ColourCoded = {
  setColourRGB: fluent( function (r, g, b) {
    this.colourCode = { r: r, g: g, b: b };
  }),
  getColourRGB: function () {
    return this.colourCode;
```

```
  }
};
```

Mixing colour coding into our Todo prototype is straightforward:

```
extend(Todo.prototype, ColourCoded);

Todo.prototype;
  //=> { do: [Function],
  //     undo: [Function],
  //     setColourRGB: [Function],
  //     getColourRGB: [Function] }
```

what is a “mixin?”

Like “class,” the word “mixin” means different things to different people. A Ruby user will talk about modules, for example. And a JavaScript user could in truth say that everything is an object and we’re just extending one object (that happens to be a prototype) with the properties of another object (that just happens to contain some functions).

A simple definition that works for most purposes is to define a mixin as: A collection of behaviour that can be added to a class’s existing prototype. `ColourCoded` above is such a mixin. If we had to actually assign a new prototype to the `Todo` class, that wouldn’t be mixing functionality in, that would be replacing functionality.

functional mixins

The 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 far more elegant to define a mixin as a function rather than an object. He calls this a functional mixin. Here’s our `ColourCoded` recast in functional form:

```
function becomeColourCoded (target) {
  target.setColourRGB = fluent( function (r, g, b) {
    this.colourCode = { r: r, g: g, b: b };
  });

  target.getColourRGB = function () {
    return this.colourCode;
  };

  return target;
};

becomeColourCoded(Todo.prototype);
```

```
14
15 Todo.prototype;
16   //=> { do: [Function],
17   //     undo: [Function],
18   //     setColourRGB: [Function],
19   //     getColourRGB: [Function] }
```

Notice that we mix the functionality into the prototype. This keeps our mixing flexible: You could mix functionality directly into an object if you so choose. Twitter's Flight framework uses a variation on this technique that targets the mixin function's context:

```
1  function asColourCoded () {
2    this.setColourRGB = fluent( function (r, g, b) {
3      this.colourCode = { r: r, g: g, b: b };
4    });
5
6    this.getColourRGB = function () {
7      return this.colourCode;
8    };
9
10   return this;
11 };
12
13 asColourCoded.call(Todo.prototype);
```

This approach has some subtle benefits: You can use mixins as methods, for example. It's possible to write a context-agnostic functional mixin:

```
1  function colourCoded () {
2    if (arguments[0] !== void 0) {
3      return colourCoded.call(arguments[0]);
4    }
5    this.setColourRGB = fluent( function (r, g, b) {
6      this.colourCode = { r: r, g: g, b: b };
7    });
8
9    this.getColourRGB = function () {
10     return this.colourCode;
11   };
12
13   return this;
14 };
```

Bueno!

Class Decorators

Functional Mixins look a lot like the function and method decorators we've seen. The big difference is that the mixin alters its subject, whereas the function decorators return a new function that wraps the old one. That can be handy if we wish, for example, to have some Todos that are not colour coded and we don't want to have a wild hierarchy of inheritance or if we wish to dynamically mix functionality into a class.

> There is a strong caveat: At this time, JavaScript is inflexible about dynamically paramaterizing calls to constructors. Therefore, the function decorator pattern being discussed here only works with constructors that are new agnostic[1] and that can create an "empty object."

Once again, our Todo class:

```
function Todo (name) {
  var self = this instanceof Todo
             ? this
             : new Todo();
  self.name = name || 'Untitled';
  self.done = false;
  return self;
};

Todo.prototype.do = fluent( function () {
  this.done = true;
});

Todo.prototype.undo = fluent( function () {
  this.done = false;
});

Todo.prototype;
  //=> { do: [Function], undo: [Function] }
```

Here's our ColourCoded as a class decorator: It returns a new class rather than modifying ToDo:

```
function AndColourCoded (clazz) {
  function Decorated  () {
    var self = this instanceof Decorated
               ? this
               : new Decorated();

    return clazz.apply(self, arguments);
```

```
  };
  Decorated.prototype = new clazz();

  Decorated.prototype.setColourRGB = fluent( function (r, g, b) {
    this.colourCode = { r: r, g: g, b: b };
  });

  Decorated.prototype.getColourRGB = function () {
    return this.colourCode;
  };

  return Decorated;
};

var ColourTodo = AndColourCoded(Todo);

Todo.prototype;
  //=> { do: [Function], undo: [Function] }

var colourTodo = new ColourTodo('Write more JavaScript');
colourTodo.setColourRGB(255, 255, 0);
  //=> { name: 'Write more JavaScript',
  //     done: false,
  //     colourCode: { r: 255, g: 255, b: 0 } }

colourTodo instanceof Todo
  //=> true

colourTodo instanceof ColourTodo
  //=> true
```

Although the implementation is more subtle, class decorators can be an improvement on functional mixins when you wish to avoid destructively modifying an existing prototype.

Interlude: 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 solved" 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:

```
var LinkedList, list, tortoiseAndHareLoopDetector;

LinkedList = (function() {

  function LinkedList(content, next) {
    this.content = content;
    this.next = next != null ? next : void 0;
  }

  LinkedList.prototype.appendTo = function(content) {
    return new LinkedList(content, this);
  };

  LinkedList.prototype.tailNode = function() {
    var nextThis;
    return ((nextThis = this.next) != null ? nextThis.tailNode() : void 0) || th\
is;
  };

  return LinkedList;

})();

tortoiseAndHareLoopDetector = function(list) {
  var hare, tortoise, nextHare;
  tortoise = list;
  hare = list.next;
  while ((tortoise != null) && (hare != null)) {
    if (tortoise === hare) {
      return true;
    }
    tortoise = tortoise.next;
    hare = (nextHare = hare.next) != null ? nextHare.next : void 0;
  }
  return false;
};

list = new LinkedList(5).appendTo(4).appendTo(3).appendTo(2).appendTo(1);

```

```
tortoiseAndHareLoopDetector(list);
  //=> false

list.tailNode().next = list.next;

tortoiseAndHareLoopDetector(list);
  //=> 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:

```
var list, teleportingTurtleLoopDetector;

teleportingTurtleLoopDetector = function(list) {
  var i, rabbit, speed, turtle;
  speed = 1;
  turtle = rabbit = list;
  while (true) {
    for (i = 0; i <= speed; i += 1) {
      rabbit = rabbit.next;
      if (rabbit == null) {
        return false;
      }
      if (rabbit === turtle) {
        return true;
      }
    }
    turtle = rabbit;
    speed *= 2;
  }
  return false;
};

list = new LinkedList(5).appendTo(4).appendTo(3).appendTo(2).appendTo(1);

teleportingTurtleLoopDetector(list);
  //=> false

list.tailNode().next = list.next;

```

```
30 teleportingTurtleLoopDetector(list);
31   //=> true
```

Years later, I came across a discussion of this algorithm, The Tale of the Teleporting Turtle. 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.

Functional Iterators

Let's consider a remarkably simple problem: Finding the sum of the elements of an array. In iterative style, it looks like this:

```
1 function sum (array) {
2   var number, total, len;
3   total = 0;
4   for (i = 0, len = array.length; i < len; i++) {
5     number = array[i];
6     total += number;
7   }
8   return total;
9 };
```

What's the sum of a linked list of numbers? How about the sum of a tree of numbers (represented as an array of array of numbers)? Must we re-write the sum function for each data structure?

There are two roads ahead. One involves a generalized reduce or fold method for each data structure. The other involves writing an Iterator for each data structure and writing our sum to take an iterator as its argument. Let's use iterators, especially since we need two different iterators for the same data structure, so a single object method is inconvenient.

Since we don't have iterators baked into the underlying JavaScript engine yet, we'll write our iterators as functions:

```
1 var LinkedList, list;
2
3 LinkedList = (function() {
4
5   function LinkedList(content, next) {
```

```
    this.content = content;
    this.next = next != null ? next : void 0;
  }

  LinkedList.prototype.appendTo = function(content) {
    return new LinkedList(content, this);
  };

  LinkedList.prototype.tailNode = function() {
    var nextThis;
    return ((nextThis = this.next) != null ? nextThis.tailNode() : void 0) || th\
is;
  };

  return LinkedList;

})();

function ListIterator (list) {
  return function() {
    var node;
    node = list != null ? list.content : void 0;
    list = list != null ? list.next : void 0;
    return node;
  };
};

function sum (iter) {
  var number, total;
  total = 0;
  number = iter();
  while (number != null) {
    total += number;
    number = iter();
  }
  return total;
};

list = new LinkedList(5).appendTo(4).appendTo(3).appendTo(2).appendTo(1);

sum(ListIterator(list));
  //=> 15

function ArrayIterator (array) {
  var index;
  index = 0;
  return function() {
    return array[index++];
```

```
  };
};

sum(ArrayIterator([1, 2, 3, 4, 5]));
  //=> 15
```

Summing an array that can contain nested arrays adds a degree of complexity. Writing a function that iterates recursively over a data structure is an interesting problem, one that is trivial in a language with coroutines. Since we don't have Generators yet, and we don't want to try to turn our loop detection inside-out, we'll Greenspun our own coroutine by maintaining our own stack.

> This business of managing your own stack may seem weird to anyone born after 1970, but old fogeys fondly remember that after walking barefoot to and from University uphill in a blizzard both ways, the interview question brain teaser of the day was to write a "Towers of Hanoi" solver in a language like BASIC that didn't have reentrant subroutines.

```
function LeafIterator (array) {
  var index, myself, state;
  index = 0;
  state = [];
  myself = function() {
    var element, tempState;
    element = array[index++];
    if (element instanceof Array) {
      state.push({
        array: array,
        index: index
      });
      array = element;
      index = 0;
      return myself();
    } else if (element === void 0) {
      if (state.length > 0) {
        tempState = state.pop(), array = tempState.array, index = tempState.inde\
x;
        return myself();
      } else {
        return void 0;
      }
    } else {
      return element;
    }
  };
  return myself;
```

```
};

sum(LeafIterator([1, [2, [3, 4]], [5]]));
  //=> 15
```

We've successfully separated the issue of what one does with data from how one traverses over the elements.

folding

Just as pure functional programmers love to talk monads, newcomers to functional programming in multi-paradigm languages often drool over folding a/k/a mapping/injecting/reducing. We're just a level of abstraction away:

```
function fold (iter, binaryFn, seed) {
  var acc, element;
  acc = seed;
  element = iter();
  while (element != null) {
    acc = binaryFn.call(element, acc, element);
    element = iter();
  }
  return acc;
};

function foldingSum (iter) {
  return fold(iter, (function(x, y) {
    return x + y;
  }), 0);
};

foldingSum(LeafIterator([1, [2, [3, 4]], [5]]));
  //=> 15
```

Fold turns an iterator over a finite data structure into an accumulator. And once again, it works with any data structure. You don't need a different kind of fold for each kind of data structure you use.

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, in principle, manufacture the data as they go. Let's consider the simplest example:

```
function NumberIterator (base) {
  var number;
```

```
  if (base == null) {
    base = 0;
  }
  number = base;
  return function() {
    return number++;
  };
};

fromOne = NumberIterator(1);

fromOne();
  //=> 1
fromOne();
  //=> 2
fromOne();
  //=> 3
fromOne();
  //=> 4
fromOne();
  //=> 5
```

And here's another one:

```
function FibonacciIterator () {
  var current, previous;
  previous = 0;
  current = 1;
  return function() {
    var value, tempValues;
    value = current;
    tempValues = [current, current + previous], previous = tempValues[0], curren\
t = tempValues[1];
    return value;
  };
};

fib = FibonacciIterator()

fib()
  //=> 1
fib()
  //=> 1
fib()
  //=> 2
fib()
  //=> 3
```

```
fib()
  //=> 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.

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:

```
take = function(iter, numberToTake) {
  var count;
  count = 0;
  return function() {
    if (++count <= numberToTake) {
      return iter();
    } else {
      return void 0;
    }
  };
};

oneToFive = take(NumberIterator(1), 5);

oneToFive();
  //=> 1
oneToFive();
  //=> 2
oneToFive();
  //=> 3
oneToFive();
  //=> 4
oneToFive();
  //=> 5
oneToFive();
  //=> undefined
```

With take, we can do things like return the squares of the first five numbers:

```
square(take(NumberIterator(1), 5))

  //=> [ 1,
  //     4,
  //     9,
```

```
6   //     16,
7   //     25 ]
```

How about the squares of the odd numbers from the first five numbers?

```
1 square(odds(take(NumberIterator(1), 5)))
2   //=> TypeError: object is not a function
```

Bzzzt! Our odds function returns an array, not an iterator.

```
1 square(take(odds(NumberIterator(1)), 5))
2   //=> RangeError: Maximum call stack size exceeded
```

You can't take the first five odd numbers at all, because odds tries to get the entire set of numbers and accumulate the odd ones in an array. This can be fixed. For unfolds and other infinite iterators, we need more functions that transform one iterator into another:

```
1  function iteratorMap (iter, unaryFn) {
2    return function() {
3      var element;
4      element = iter();
5      if (element != null) {
6        return unaryFn.call(element, element);
7      } else {
8        return void 0;
9      }
10   };
11 };
12
13 function squaresIterator (iter) {
14   return iteratorMap(iter, function(n) {
15     return n * n;
16   });
17 };
18
19 function iteratorFilter (iter, unaryPredicateFn) {
20   return function() {
21     var element;
22     element = iter();
23     while (element != null) {
24       if (unaryPredicateFn.call(element, element)) {
25         return element;
26       }
27       element = iter();
28     }
```

```
    return void 0;
  };
};

function oddsFilter (iter) {
  return iteratorFilter(iter, odd);
};
```

Now we can do things like take the sum of the first five odd squares of fibonacci numbers:

```
foldingSum(take(oddsFilter(squaresIterator(FibonacciIterator())), 5))
  //=> 205
```

This solution composes the parts we already have, rather than writing a tricky bit of code with ifs and whiles and boundary conditions.

summary

Untangling the concerns of how to iterate over data from what to do with data leads us to thinking of iterators and working directly with iterators. For example, we can map and filter iterators rather than trying to write separate map and filter functions or methods for each type of data structure. This leads to the possibility of working with lazy or infinite iterators.

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.

Refactoring to Functional Iterators

In Tortoises, Hares, and Teleporting Turtles, we looked at the "Tortoise and Hare" algorithm for detecting a linked list. Like many such algorithms, it "tangles" two different concerns:

1. The mechanism for iterating over a list.
2. The algorithm for detecting a loop in a list.

```
var LinkedList = (function() {

  function LinkedList(content, next) {
    this.content = content;
    this.next = next != null ? next : void 0;
  }

  LinkedList.prototype.appendTo = function(content) {
    return new LinkedList(content, this);
  };

  LinkedList.prototype.tailNode = function() {
    var nextThis;
    return ((nextThis = this.next) != null ? nextThis.tailNode() : void 0) || th\
is;
  };

  return LinkedList;

})();

function tortoiseAndHareLoopDetector (list) {
  var hare, tortoise, nextHare;
  tortoise = list;
  hare = list.next;
```

```
  while ((tortoise != null) && (hare != null)) {
    if (tortoise === hare) {
      return true;
    }
    tortoise = tortoise.next;
    hare = (nextHare = hare.next) != null ? nextHare.next : void 0;
  }
  return false;
};
```

functional iterators

We then went on to discuss how to use functional iterators to untangle concerns like this. A functional iterator is a stateful function that iterates over a data structure. Every time you call it, it returns the next element from the data structure. If and when it completes its traversal, it returns undefined.

For example, here is a function that takes an array and returns a functional iterator over the array:

```
function ArrayIterator (array) {
  var index = 0;
  return function() {
    return array[index++];
  };
};
```

Iterators allow us to write (or refactor) functions to operate on iterators instead of data structures. That increases reuse. We can also write higher-order functions that operate directly on iterators such as mapping and selecting. That allows us to write lazy algorithms.

refactoring the tortoise and hare

Now we'll refactor the Tortoise and Hare to use iterators instead of directly operate on linked lists. We'll add an .iterator() method to linked lists, and we'll rewrite our loop detector function to take an "iterable" instead of a list:

```
LinkedList.prototype.iterator = function() {
  var list = this;
  return function() {
    var value = list != null ? list.content : void 0;
    list = list != null ? list.next : void 0;
    return value;
  };
```

```
};

function tortoiseAndHareLoopDetector (iterable) {
  var tortoise = iterable.iterator(),
      hare = iterable.iterator(),
      tortoiseValue,
      hareValue;
  while (((tortoiseValue = tortoise()) != null) && ((hare(), hareValue = hare())\
 != null)) {
    if (tortoiseValue === hareValue) {
      return true;
    }
  }
  return false;
};

list = new LinkedList(5).appendTo(4).appendTo(3).appendTo(2).appendTo(1);

tortoiseAndHareLoopDetector(list);
  //=> false

list.tailNode().next = list.next;

tortoiseAndHareLoopDetector(list);
  //=> true
```

We have now refactored it into a function that operates on anything that responds to the .iterator() method. It's classic "Duck Typed" Object-Orientation. So, how shall we put it to work?

A Drunken Walk Across A Chequerboard

Here's another job interview puzzle.[2]

Consider a finite checkerboard of unknown size. On each square we randomly place an arrow pointing to one of its four sides. For convenience, we shall uniformly label the directions: N, S, E, and W. A chequer is placed randomly on the checkerboard. Each move consists of moving the red 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.

As a player moves the chequer, he calls out the direction of movement, e.g. "N, E, N, S, N, E…" Write an algorithm that will determine whether the game halts strictly from the called out directions, in constant space.

the insight

Our solution will rest on the observation that as the chequer follows a path, if it ever visits a square for a second time, it will cycle indefinitely without falling off the board. Otherwise, on a finite board, it must eventually fall off the board after at most visiting every square once.

Therefore, if we think of this as detecting whether the chequer revisits a square in constant space, we can see this is isomorphic to detecting whether a linked list has a loop by checking to see whether it revisits the same node.

the game

In essence, we're given an object that has a .iterator() method. That gives us an iterator, and each time we call the iterator, we get a direction. Here it is:

```
var DIRECTION_TO_DELTA = {
  N: [1, 0],
  E: [0, 1],
  S: [-1, 0],
  W: [0, -1]
};

var Game = (function () {
  function Game (size) {
    var i,
        j;

    this.size = size
                ? Math.floor(Math.random() * 8) + 8
                : size ;
    this.board = [];
    for (i = 0; i < this.size; ++i) {
      this.board[i] = [];
      for (j = 0; j < this.size; ++j) {
        this.board[i][j] = 'NSEW'[Math.floor(Math.random() * 4)];
      }
    }
    this.initialPosition = [
      2 + Math.floor(Math.random() * (this.size - 4)),
      2 + Math.floor(Math.random() * (this.size - 4))
    ];
    return this;
  };

  Game.prototype.contains = function (position) {
    return position[0] >= 0 && position[0] < this.size && position[1] >= 0 && po\
sition[1] < this.size;
  };
```

```
34
35    Game.prototype.iterator = function () {
36      var position = [this.initialPosition[0], this.initialPosition[1]];
37      return function () {
38        var direction;
39        if (this.contains(position)) {
40          direction = this.board[position[0]][position[1]];
41          position[0] += DIRECTION_TO_DELTA[direction][0];
42          position[1] += DIRECTION_TO_DELTA[direction][1];
43          return direction;
44        }
45        else {
46          return void 0;
47        }
48      }.bind(this);
49    };
50
51    return Game;
52
53  })();
54
55  var i = new Game().iterator();
56    //=> [Function]
57  i();
58    //=> 'N'
59  i();
60    //=> 'S'
61  i();
62    //=> 'N'
63  i();
64    //=> 'S'
65    //   ...
```

In the example above, we have the smallest possible repeating path: The chequer shuttles back and forth between two squares. It will not always be so obvious when a game does not terminate.

stateful mapping

Our goal is to transform the iteration of directions into an iteration that the Tortoise and Hare can use to detect revisiting the same square. Our approach is to convert the directions into offsets representing the position of the chequer relative to its starting position.

We'll use a statefulMap:

```
1  function statefulMap (iter, binaryFn, initial) {
```

```
  var state = initial;
  return function () {
    element = iter();
    if (element == null) {
      return element;
    }
    else {
      if (state === void 0) {
        return (state = element);
      }
      else return (state = binaryFn.call(element, state, element));
    }
  }
};
```

statefulMap takes in iterator and maps it to a new iterator. Unlike a “regular” map, it computes its elements on demand, so it will not run indefinitely when given an iteration representing an infinitely looping chequer. We need a stateful map because we are tracking a position that changes over time even when given the same direction over and over again.

Here’s how we use statefulMap:

```
function RelativeIterator (directionIterator) {
  return statefulMap(directionIterator, function (relativePositionStr, direction\
Str) {
    var delta = DIRECTION_TO_DELTA[directionStr],
        matchData = relativePositionStr.match(/(-?\d+) (-?\d+)/),
        relative0 = parseInt(matchData[1], 10),
        relative1 = parseInt(matchData[2], 10);
    return "" + (relative0 + delta[0]) + " " + (relative1 + delta[1]);
  }, "0 0");
};

var i = RelativeIterator(new Game().iterator());
i();
  //=> '-1 0'
i();
  //=> '-1 -1'
i();
  //=> '-2 -1'
i();
  //=> '-2 0'
i();
  //=> '-2 1'
i();
```

```
24   //=> '-3 1'
25 i();
26   //=> '-3 2'
27 i();
28   //=> '-3 3'
```

We're almost there! The refactored tortoiseAndHareLoopDetector expects an "iterable," an object that implements the .iterator() method. Let's refactor RelativeIterable to accept a game and return an iterable instead of accepting an iteration and returning an iteration:

```
1 function RelativeIterable (game) {
2   return {
3     iterator: function () {
4         return statefulMap(game.iterator(), function (relativePositionStr, direc\
5 tionStr) {
6           var delta = DIRECTION_TO_DELTA[directionStr],
7               matchData = relativePositionStr.match(/(-?\d+) (-?\d+)/),
8               relative0 = parseInt(matchData[1], 10),
9               relative1 = parseInt(matchData[2], 10);
10          return "" + (relative0 + delta[0]) + " " + (relative1 + delta[1]);
11        }, "0 0");
12    }
13  };
14 };
15
16 var i = RelativeIterable(new Game()).iterator();
17 i();
18   //=> '0 -1'
19 i();
20   //=> '1 -1'
21 i();
22   //=> '1 0'
23 i();
24   //=> '2 0'
25 i();
26   //=> undefined
```

the solution

So. We can take a Game instance and produce an iterable that iterates over regular strings representing relative positions. If it terminates on its own, the game obviously terminates. And if it repeats itself, the game does not terminate.

Our refactored tortoiseAndHareLoopDetector takes an iterable and detects this for us. Writing a detector function is trivial:

```
function terminates (game) {
  return !tortoiseAndHareLoopDetector(RelativeIterable(game));
}

terminates(new Game(4));
  //=> false
terminates(new Game(4));
  //=> true
terminates(new Game(4));
  //=> false
terminates(new Game(4));
  //=> false
```

preliminary conclusion

Untangling the mechanism of following a linked list from the algorithm of searching for a loop allows us to repurpose the Tortoise and Hare algorithm to solve a question about a path looping.

no-charge extra conclusion

Can we also refactor the “Teleporting Turtle” algorithm to take an iterable? If so, we should be able to swap algorithms for our game termination detection without rewriting everything in sight. Let’s try it:

We start with:

```
function teleportingTurtleLoopDetector (list) {
  var i, rabbit, speed, turtle;
  speed = 1;
  turtle = rabbit = list;
  while (true) {
    for (i = 0; i <= speed; i += 1) {
      rabbit = rabbit.next;
      if (rabbit == null) {
        return false;
      }
      if (rabbit === turtle) {
        return true;
      }
    }
    turtle = rabbit;
    speed *= 2;
  }
  return false;
};

```

```
list = new LinkedList(5).appendTo(4).appendTo(3).appendTo(2).appendTo(1);

teleportingTurtleLoopDetector(list);
  //=> false

list.tailNode().next = list.next;

teleportingTurtleLoopDetector(list);
  //=> true
```

And refactor it to become:

```
function teleportingTurtleLoopDetector (iterable) {
  var i, rabbit, rabbitValue, speed, turtleValue;
  speed = 1;
  rabbit = iterable.iterator();
  turtleValue = rabbitValue = rabbit();
  while (true) {
    for (i = 0; i <= speed; i += 1) {
      rabbitValue = rabbit();
      if (rabbitValue == null) {
        return false;
      }
      if (rabbitValue === turtleValue) {
        return true;
      }
    }
    turtleValue = rabbitValue;
    speed *= 2;
  }
  return false;
};

list = new LinkedList(5).appendTo(4).appendTo(3).appendTo(2).appendTo(1);

teleportingTurtleLoopDetector(list);
  //=> false

list.tailNode().next = list.next;

teleportingTurtleLoopDetector(list);
  //=> true
```

Now we can plug it into our termination detector:

```
function terminates (game) {
```

```
  return !teleportingTurtleLoopDetector(RelativeIterable(game));
}

terminates(new Game(4));
  //=> false
terminates(new Game(4));
  //=> false
terminates(new Game(4));
  //=> false
terminates(new Game(4));
  //=> false
terminates(new Game(4));
  //=> true
```

Refactoring an algorithm to work with iterators allows us to use the same algorithm to solve different problems and to swap algorithms for the same problem. This is natural, we have created an abstraction that allows us to plug different items into either side of its interface.

Trampolining

> A trampoline is a loop that iteratively invokes thunk-returning functions (continuation-passing style). A single trampoline is sufficient to express all control transfers of a program; a program so expressed is trampolined, or in trampolined style; converting a program to trampolined style is trampolining. Trampolined functions can be used to implement tail-recursive function calls in stack-oriented programming languages.–Wikipedia

This description is exactly how one ought to answer the question “define trampolining” on an examination, because it demonstrates that you’ve learned the subject thoroughly. But if asked to explain trampolining, a more tutorial-focused approach is called for.

Let’s begin with a use case.

recursion, see recursion

Consider implementing factorial in recursive style:

```
function factorial (n) {
  return n
  ? n * factorial(n - 1)
  : 1
}
```

The immediate limitation of this implementation is that since it calls itself n times, to get a

result you need a stack on n stack frames in a typical stack-based programming language implementation. And JavaScript is such an implementation.

This creates two problems: First, we need space On for all those stack frames. It's as if we actually wrote out `1 x 1 x 2 x 3 x 4 x ...` before doing any calculations. Second, most languages have a limit on the size of the stack much smaller than the limit on the amount of memory you need for data.

For example:

```
factorial(10)
  //=> 3628800
factorial(32768)
  //=> RangeError: Maximum call stack size exceeded
```

We can easily rewrite this in iterative style, but there are other functions that aren't so amenable to rewriting and using a simple example allows us to concentrate on the mechanism rather than the "domain."

tail-call elimination

Lisp programmers in days of yore would rewrite functions like this into "Tail Recursive Form," and that made it possible for their compilers to perform Tail-Call Optimization. Meaning, that when a function returns the result of calling itself, the language doesn't actually perform another function call, it turns the whole thing into a loop for you.

What we need to do is take the expression `n * factorial(n - 1)` and push it down into a function so we can just call it with parameters. When a function is called, a stack frame is created that contains all the information needed to resume execution with the result. Stack frames hold a kind of pointer to where to carry on evaluating, the function parameters, and other bookkeeping information.[3]

If we use the symbol `_` to represent a kind of "hole" in an expression where we plan to put the result, every time `factorial` calls itself, it needs to remember `n * _` so that when it gets a result back, it can multiply it by `n` and return that. So the first time it calls itself, it remembers `10 * _`, the second time it calls itself, it remembers `9 * _`, and all these things stack up like this when we call `factorial(10)`:

```
1  1 * _
2  2 * _
3  3 * _
4  4 * _
5  5 * _
```

```
 6  6 * _
 7  7 * _
 8  8 * _
 9  9 * _
10 10 * _
```

Finally, we call `factorial(0)` and it returns `1`. Then the top is popped off the stack, so we calculate `1 * 1`. It returns `1` again and we calculate `2 * 1`. That returns `2` and we calculate `3 * 2` and so on up the stack until we return `10 * 362880` and return `3628800`, which we print.

How can we get around this? Well, imagine if we don't have a hole in a computation to return. In that case, we wouldn't need to "remember" anything on the stack. To make this happen, we need to either return a value or return the result of calling another function without any further computation.

Such a call is said to be in "tail position" and to be a "tail call." The "elimination" of tail-call elimination means that we don't perform a full call including setting up a new stack frame. We perform the equivalent of a "jump."

For example:

```
function factorial (n) {
  var _factorial = function myself (acc, n) {
    return n
    ? myself(acc * n, n - 1)
    : acc
  };

  return _factorial(1, n);
}
```

Now our function either returns a value or it returns the result of calling another function without doing anything with that result. This gives us the correct results, but we can see that current implementations of JavaScript don't perform this magic "tail-call elimination."

```
factorial(10)
  //=> 3628800
factorial(32768)
  //=> RangeError: Maximum call stack size exceeded
```

So we'll do it ourselves.

trampolining

One way to implement tail-call elimination is also handy for many other general things we might want to do with control flow, it's called trampolining. What we do is this:

When we call a function, it returns a thunk that we call to get a result. Of course, the thunk can return another thunk, so every time we get a result, we check to see if it's a thunk. If not, we have our final result.

A thunk is a function taking no arguments that delays evaluating an expression. For example, this is a thunk: `function () { return 'Hello World'; }`.

An extremely simple and useful implementation of trampolining can be found in the Lemonad library. It works provided that you want to trampoline a function that doesn't return a function. Here it is:

```
L.trampoline = function(fun /*, args */) {
  var result = fun.apply(fun, _.rest(arguments));

  while (_.isFunction(result)) {
    result = result();
  }

  return result;
};
```

We'll rewrite it in combinatorial style for consistency and composeability:

```
var trampoline = function (fn) {
  return variadic( function (args) {
    var result = fn.apply(this, args);

    while (result instanceof Function) {
      result = result();
    }

    return result;
  });
};
```

Now here's our implementation of `factorial` that is wrapped around a trampolined tail recursive function:

```
function factorial (n) {
  var _factorial = trampoline( function myself (acc, n) {
    return n
```

```
    ? function () { return myself(acc * n, n - 1); }
    : acc
  });

  return _factorial(1, n);
}

factorial(10);
  //=> 362800
factorial(32768);
  //=> Infinity
```

Presto, it runs for $n = 32768$. Sadly, JavaScript's built-in support for integers cannot keep up, so we'd better fix the "infinity" problem with a "big integer" library:[4]

```
npm install big-integer

var variadic = require('allong.es').variadic,
    bigInt = require("big-integer");

var trampoline = function (fn) {
  return variadic( function (args) {
    var result = fn.apply(this, args);

    while (result instanceof Function) {
      result = result();
    }

    return result;

  });
};

function factorial (n) {
  var _factorial = trampoline( function myself (acc, n) {
    return n.greater(0)
    ? function () { return myself(acc.times(n), n.minus(1)); }
    : acc
  });

  return _factorial(bigInt.one, bigInt(n));
}

factorial(10).toString()
  //=> '3628800'
factorial(32768)
  //=> GO FOR LUNCH
```

Well, it now takes a very long time to run, but it is going to get us the proper result and we can print that as a string, so we'll leave it calculating in another process and carry on.

The limitation of the implementation shown here is that because it tests for the function returning a function, it will not work for functions that return functions. If you want to trampoline a function that returns a function, you will need a more sophisticated mechanism, but the basic principle will be the same: The function will return a thunk instead of a value, and the trampolining loop will test the returned thunk to see if it represents a value or another computation to be evaluated.

trampolining co-recursive functions

If trampolining was only for recursive functions, it would have extremely limited value: All such functions can be re-written iteratively and will be much faster (although possibly less elegant). However, trampolining eliminates all calls in tail position, including calls to other functions.

Consider this delightfully simple example of two co-recursive functions:

```
function even (n) {
  return n == 0
    ? true
    : odd(n - 1);
};

function odd (n) {
  return n == 0
    ? false
    : even(n - 1);
};
```

Like our factorial, it consumes n stack space of alternating calls to even and odd:

```
even(32768);
  //=> RangeError: Maximum call stack size exceeded
```

Obviously we can solve this problem with modulo arithmetic, but consider that what this shows is a pair of functions that call other functions in tail position, not just themselves. As with factorial, we separate the public interface that is not trampolined from the trampolined implementation:

```
var even = trampoline(_even),
    odd  = trampoline(_odd);
```

```
3
4 function _even (n) {
5   return n == 0
6     ? true
7     : function () { return _odd(n - 1); };
8 };
9
10 function _odd (n) {
11   return n == 0
12     ? false
13     : function () { return _even(n - 1); };
14 };
```

And presto:

```
1 even(32768);
2   //=> true
```

Trampolining works with co-recursive functions, or indeed any function that can be rewritten in tail-call form.

summary

Trampolining is a technique for implementing tail-call elimination. Meaning, if you take a function (whether recursive, co-recursive, or any other form) and rewrite it in tail-call form, you can eliminate the need to create a stack frame for every 'invocation'.

Trampolining is very handy in a language like JavaScript, in that it allows you to use a recursive style for functions without worrying about limitations on stack sizes.

Another approach that works with ECMAScript 5 and later is to base all classes around Object.create↩

This book does not blindly endorse asking programmers to solve this or any abstract problem in a job interview.↩

Did you know that "bookkeeping" is the only word in the English language containing three consecutive letter pairs? You're welcome.↩

The use of a third-party big integer library is not essential to understand trampolining.↩

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.”

Before

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 Flavors system and also the Aspect-Oriented Programming movement, is decorating a method with some functionality to be performed before the method’s body is evaluated.

For example, using our fluent recipe:

```
1 function Cake () {
2   this.ingredients = {}
3 }
4
5 extend(Cake.prototype, {
6   setFlavour: fluent( function (flavour) {
7     this.flavour = flavour
```

```
  }),
  setLayers: fluent( function (layers) {
    this.layers = layers;
  }),
  add: fluent( function (ingredientMap) {
    var ingredient;

    for (ingredient in ingredientMap) {
      this.ingredients[ingredient] ||
        (this.ingredients[ingredient] = 0);
      this.ingredients[ingredient] = this.ingredients[ingredient] +
        ingredientMap[ingredient]
    }
  }),
  mix: fluent( function () {
    // mix ingredients together
  }),
  rise: fluent( function (duration) {
    // let the ingredients rise
  }),
  bake: fluent( function () {
    // do some baking
  })
});
```

This particular example might be better-served as a state machine, but what we want to encode is that we must always mix the ingredients before allowing the batter to rise or baking the cake. The direct way to write that is:

```
  rise: fluent( function (duration) {
    this.mix();
    // let the ingredients rise
  }),
  bake: fluent( function () {
    this.mix();
    // do some baking
  })
```

Nothing wrong with that, however it does clutter the core functionality of rising and baking with a secondary concern, preconditions. There is a similar problem with cross-cutting concerns like logging or checking permissions: You want functions to be smaller and more focused, and decomposing into smaller methods is ugly:

```
reallyRise: function (duration) {
  // let the ingredients rise
},
```

```
rise: fluent (function (duration) {
  this.mix();
  this.reallyRise(duration)
}),
reallyBake: function () {
  // do some baking
},
bake: fluent( function () {
  this.mix();
  this.reallyBake()
})
```

the before recipe

This recipe is for a combinator that turns a function into a method decorator. The decorator evaluates the function before evaluating the base method. Here it is:

```
function before (decoration) {
  return function (method) {
    return function () {
      decoration.apply(this, arguments);
      return method.apply(this, arguments)
    }
  }
}
```

And here we are using it in conjunction with fluent, showing the power of composing combinators:

```
var mixFirst = before(function () {
  this.mix()
});

extend(Cake.prototype, {

  // Other methods...

  mix: fluent( function () {
    // mix ingredients together
  }),
  rise: fluent( mixFirst( function (duration) {
    // let the ingredients rise
  })),
  bake: fluent( mixFirst( function () {
    // do some baking
  }))
```

```
});
```

The decorators act like keywords or annotations, documenting the method's behaviour but clearly separating these secondary concerns from the core logic of the method.

(before, after, and many more combinators for building method decorators can be found in the method combinators module.)

After

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 Flavors system and also the Aspect-Oriented Programming movement, is decorating a method with some functionality to be performed after the method's body is evaluated.

For example, consider this "class:"

```
Todo = function (name) {
  this.name = name || 'Untitled';
  this.done = false
}

extend(Todo.prototype, {
  do: fluent( function {
    this.done = true
  }),
  undo: fluent( function {
    this.done = false
  }),
  setName: fluent( maybe( function (name) {
    this.name = name
  }))
});
```

If we're rolling our own model class, we might mix in Backbone.Events. Now we can have views listen to our todo items and render themselves when there's a change. Since we've already seen before, we'll jump right to the recipe for after, a combinator that turns a function into a method decorator:

```
function after (decoration) {
  return function (method) {
    return function () {
```

```
      var value = method.apply(this, arguments);
      decoration.call(this, value);
      return value
    }
  }
}
```

And here it is in use to trigger change events on our `Todo` “class.” We’re going to be even more sophisticated and paramaterize our decorators.

```
extend(Todo.prototype, Backbone.Events);

function changes (propertyName) {
  return after(function () {
    this.trigger('changed changed:'+propertyName, this[propertyName])
  })
}

extend(Todo.prototype, {
  do: fluent( changes('done')( function {
    this.done = true
  })),
  undo: fluent( changes('done')( function {
    this.done = false
  })),
  setName: fluent( changes('name')( maybe( function (name) {
    this.name = name
  })))
});
```

The decorators act like keywords or annotations, documenting the method’s behaviour but clearly separating these secondary concerns from the core logic of the method.

(before, after, and many more combinators for building method decorators can be found in the method combinators module.)

Provided and Except

Neither the before and after decorators can actually terminate evaluation without throwing something. Normal execution always results in the base method being evaluated. The `provided` and `excepting` recipes are combinators that produce method decorators that apply a precondition to evaluating the base method body. If the precondition fails, nothing is returned.

The provided combinator turns a function into a method decorator. The function must evaluate to truthy for the base method to be evaluated:

```
function provided (predicate) {
  return function(base) {
    return function() {
      if (predicate.apply(this, arguments)) {
        return base.apply(this, arguments);
      }
    };
  };
};
```

provided can be used to create named decorators like maybe:

```
var maybe = provided( function (value) {
  return value != null
});

SomeModel.prototype.setAttribute = maybe( function (value) {
  this.attribute = value
});
```

You can build your own domain-specific decorators:

```
var whenNamed = provided( function (record) {
  return record.name && record.name.length > 0
})
```

except works identically, but with the logic reversed.

```
function except (predicate) {
  return function(base) {
    return function() {
      if (!predicate.apply(this, arguments)) {
        return base.apply(this, arguments);
      }
    };
  };
};

var exceptAdmin = except( function (user) {
  return user.role.isAdmin()
});
```

A Functional Mixin Factory

Functional Mixins extend an existing class's prototype. Let's start with:

```
function Todo (name) {
  var self = this instanceof Todo
             ? this
             : new Todo();
  self.name = name || 'Untitled';
  self.done = false;
  return self;
};

Todo.prototype.do = fluent( function () {
  this.done = true;
});

Todo.prototype.undo = fluent( function () {
  this.done = false;
});
```

We wish to decorate this with:

```
({
  setLocation: fluent( function (location) {
    this.location = location;
  }),
  getLocation: function () { return this.location; }
});
```

Instead of writing:

```
function becomeLocationAware () {
  this.setLocation = fluent( function (location) {
    this.location = location;
  });

  this.getLocation = function () { return this.location; };

  return this;
};
```

We'll extract the decoration into a parameter like this:

```
function mixin (decoration) {
  extend(this, decoration);
  return this;
};
```

And then “curry” the function manually like this:

```
function mixin (decoration) {

  return function () {
    extend(this, decoration);
    return this;
  };

};
```

We can try it:

```
var MixinLocation = mixin({
  setLocation: fluent( function (location) {
    this.location = location;
  }),
  getLocation: function () { return this.location; }
});

MixinLocation.call(Todo.prototype);

new Todo('Paint Bedroom').setLocation('Home');
  //=> { name: 'Paint Bedroom',
  //     done: false,
  //     location: 'Home'
```

Success! Our `mixin` function makes functional mixins. A final refinement is to make it “context-agnostic,” so that we can write either `MixinLocation.call(Todo.prototype)` or `MixinLocation(Todo.prototype)`:

```
function mixin (decoration) {

  return function decorate () {
    if (arguments[0] !== void 0) {
      return decorate.call(arguments[0]);
    }
    else {
      extend(this, decoration);
      return this;
```

```
    };
  };

};
```

A Class Decorator Factory

As discussed, a class decorator creates a new class with some additional decoration. It's lighter weight than subclassing. It's also easy to write a factory function that makes decorators for us. Recall:

```
function Todo (name) {
  var self = this instanceof Todo
             ? this
             : new Todo();
  self.name = name || 'Untitled';
  self.done = false;
  return self;
};

Todo.prototype.do = fluent( function () {
  this.done = true;
});

Todo.prototype.undo = fluent( function () {
  this.done = false;
});
```

We wish to decorate this with:

```
({
  setColourRGB: fluent( function (r, g, b) {
    this.colourCode = { r: r, g: g, b: b };
  }),
  getColourRGB: function () {
    return this.colourCode;
  }
});
```

Instead of writing:

```
function AndColourCoded (clazz) {
  function Decorated  () {
    var self = this instanceof Decorated
               ? this
```

```
              : new Decorated();

    return clazz.apply(self, arguments);
  };
  Decorated.prototype = new clazz();

  Decorated.prototype.setColourRGB = fluent( function (r, g, b) {
    this.colourCode = { r: r, g: g, b: b };
  });

  Decorated.prototype.getColourRGB = function () {
    return this.colourCode;
  };

  return Decorated;
};
```

We'll extract the decoration into a parameter like this:

```
function classDecorator (decoration, clazz) {
  function Decorated  () {
    var self = this instanceof Decorated
               ? this
               : new Decorated();

    return clazz.apply(self, arguments);
  };
  Decorated.prototype = extend(new clazz(), decoration);
  return Decorated;
};
```

And then "curry" the function manually like this:

```
function classDecorator (decoration) {

  return function (clazz) {
    function Decorated  () {
      var self = this instanceof Decorated
                 ? this
                 : new Decorated();

      return clazz.apply(self, arguments);
    };
    Decorated.prototype = extend(new clazz(), decoration);
    return Decorated;
  };
```

```
14
15 };
```

We can try it:

```
1 var AndColourCoded = classDecorator({
2   setColourRGB: fluent( function (r, g, b) {
3     this.colourCode = { r: r, g: g, b: b };
4   }),
5   getColourRGB: function () {
6     return this.colourCode;
7   }
8 });
9
10 var ColourTodo = AndColourCoded(Todo);
11
12 new ColourTodo('Use More Decorators').setColourRGB(0, 255, 0);
13   //=> { name: 'Use More Decorators',
14   //     done: false,
15   //     colourCode: { r: 0, g: 255, b: 0 } }
```

Success! Our classDecorator function makes class decorators.

Iterator Recipes

iterators for standard data structures

Note: Despite having capitalized names, iterators are not meant to be used with the new keyword.

```
1 function FlatArrayIterator (array) {
2   var index;
3   index = 0;
4   return function() {
5     return array[index++];
6   };
7 };
8
9 function RecursiveArrayIterator (array) {
10   var index, myself, state;
11   index = 0;
12   state = [];
13   myself = function() {
14     var element, tempState;
15     element = array[index++];
16     if (element instanceof Array) {
```

```
      state.push({
        array: array,
        index: index
      });
      array = element;
      index = 0;
      return myself();
    } else if (element === void 0) {
      if (state.length > 0) {
        tempState = state.pop(), array = tempState.array, index = tempState.inde\
x;
        return myself();
      } else {
        return void 0;
      }
    } else {
      return element;
    }
  };
  return myself;
};
```

unfolding iterators

```
function NumberIterator (base) {
  var number;
  if (base == null) {
    base = 0;
  }
  number = base;
  return function() {
    return number++;
  };
};

function FibonacciIterator () {
  var current, previous;
  previous = 0;
  current = 1;
  return function() {
    var value, tempValues;
    value = current;
    tempValues = [current, current + previous], previous = tempValues[0], curren\
t = tempValues[1];
    return value;
  };
};
```

decorators for slicing iterators

```
function take (iter, numberToTake) {
  var count = 0;
  return function() {
    if (++count <= numberToTake) {
      return iter();
    } else {
      return void 0;
    }
  };
};

function drop (iter, numberToDrop) {
  while (numberToDrop-- !== 0) {
    iter();
  }
  return iter;
};

function slice (iter, numberToDrop, numberToTake) {
  var count = 0;
  while (numberToDrop-- !== 0) {
    iter();
  }
  if (numberToTake != null) {
    return function() {
      if (++count <= numberToTake) {
        return iter();
      } else {
        return void 0;
      }
    };
  }
  else return iter;
};
```

(drop was suggested by Redditor skeeto. His code also cleaned up an earlier version of slice.)

catamorphic decorator

```
function fold (iter, binaryFn, seed) {
  var acc, element;
  acc = seed;
  element = iter();
  while (element != null) {
```

```
    acc = binaryFn.call(element, acc, element);
    element = iter();
  }
  return acc;
};
```

hylomorphic decorators

```
function map (iter, unaryFn) {
  return function() {
    var element;
    element = iter();
    if (element != null) {
      return unaryFn.call(element, element);
    } else {
      return void 0;
    }
  };
};

function statefulMap (iter, binaryFn, initial) {
  var state = initial;
  return function () {
    element = iter();
    if (element == null) {
      return element;
    }
    else {
      if (state === void 0) {
        return (state = element);
      }
      else return (state = binaryFn.call(element, state, element));
    }
  }
};

function filter (iter, unaryPredicateFn) {
  return function() {
    var element;
    element = iter();
    while (element != null) {
      if (unaryPredicateFn.call(element, element)) {
        return element;
      }
      element = iter();
    }
    return void 0;
  };
```

```
41 };
```

The Golden Crema

You've earned a break!

Author's Notes

Dear friends and readers:

On October 1st, 2013, I announced that JavaScript Allongé became free: It is now licensed under a Creative Commons Attribution-ShareAlike 3.0 Unported License. You are free:

- to Share—to copy, distribute and transmit the work
- to Remix—to adapt the work
- to make commercial use of the work

Read the license yourself for the full details. But the bottom line is, it's free, free, **FREE**!

And now, a few questions and answers…

---

allong.es sounds familiar…

The recipes in JavaScript Allonge inspired a companion library called allong.es. It's free to

use, of course. Please try it out. It complements the libraries you may already be using like Underscore.

I noticed that the recipes changed in early 2013. Why?

Based on feedback from people exposed to other programming languages, I've renamed some of the recipe functions. While doing so, I also rewrote the partial application and some other parts of allong.es to exploit symmetry.

The new nomenclature has a few conventions. First, unless suffixed with `Now`, all functions are already curried. So you can write either `map(list, function)` or `map(list)(function)`. There isn't one, but if there was a `map` that wasn't curried, it would be called `mapNow`.

By default, functions take a data structure first and an operation second and are named after a verb, i.e. `map`, `filter`. As noted, they are curried by default.

Binary functions like `map` have a variation with their arguments flipped to have the "verb" first and the subject second. They are suffixed `With`, so you call `map(list, function)` or `mapWith(function, list)`.

Under the new nomenclature, what used to be called `splat` is now called `mapWith`, and when you supply only the function, the currying takes care of returning a function that takes as list as its argument. You're mapping with a function.

The examples seem to use `apply` and `call` indiscriminately. Why?

In functional programming tradition, the function `apply` is used for functional application. Variations include `applyLeft` and `applyLast`. JavaScript is a little different: All functions have two methods: `.apply` takes an array of arguments, while `.call` takes separate arguments.

So the recipes in JavaScript Allongé follow the JavaScript convention: Those named `apply` take an array of arguments, while those named `call` take individual arguments. If you're coming to this book with some functional programming under your belt, you'll find the functions work the same way, it's just that there are two of them to handle the two different ways to apply arguments to a function.

I don't want to pay to download a PDF. Can I make my own?

Yes, you can take the HTML that is available online or the markdown source in the repository and make your own PDF. Or any other format. Please be aware that while it's technically possible to game the LeanPub system to produce the PDF or other formats, I ask you as a personal favour to find another way to make a PDF.

The license permits this choice, but IMO it is contrary to the spirit of sharing and

openness to use their resources and work to do something that isn't aligned with their mission. And of course, they may not care for the idea, I don't know and I don't want to undermine what has been a tremendous service for helping people write great technical books.

Great book! Can I share it?

And I'd also like you to share it, in this form, in PDF, or anything else. Go wild, just follow the attribution rules in the license.

Great book! Can I still buy it?

Yes, please do.

I don't want to buy it, but I'd like to say "thank you" with a tip.

Sure thing, you can send a donation via PayPal to reg@braythwayt.com, or click this button to donate $10 to help me write:

Hey, I have a great way to make money with this…

Go for it, you are free to make commercial use of the work. For example, you could host it on your site and make money from ads, or write a JS tool and use the book as part of the help content.

It's all good, just follow the license terms. It is technically possible to create an identical clone of the book on LeanPub. I do not prohibit this activity, but I do ask you as a personal favour to ask yourself whether you could do even better, for instance to add value by adding your own annotations, expansions, and commentary. I'd love to see an "Annotated JavaScript Allongé."

I'd also love to see translations, editions with large print, or anything else that brings something new to the world. Many people have asked for a hard-copy version. Who will be the first to set up shop on lulu.com?

I found a typo! How do I tell you about it?

My email inbox is a disaster zone, so let's treat the book like open source. In order of my preference, you should:

1. Create an issue, fork the repo, fix the issue, and then send me a pull request. (Best!!)
2. Fork the repo, fix things to your satisfaction, and then send me a pull request. (Better!)
3. Create an issue. (Good.)

   Thanks!

How to run the examples

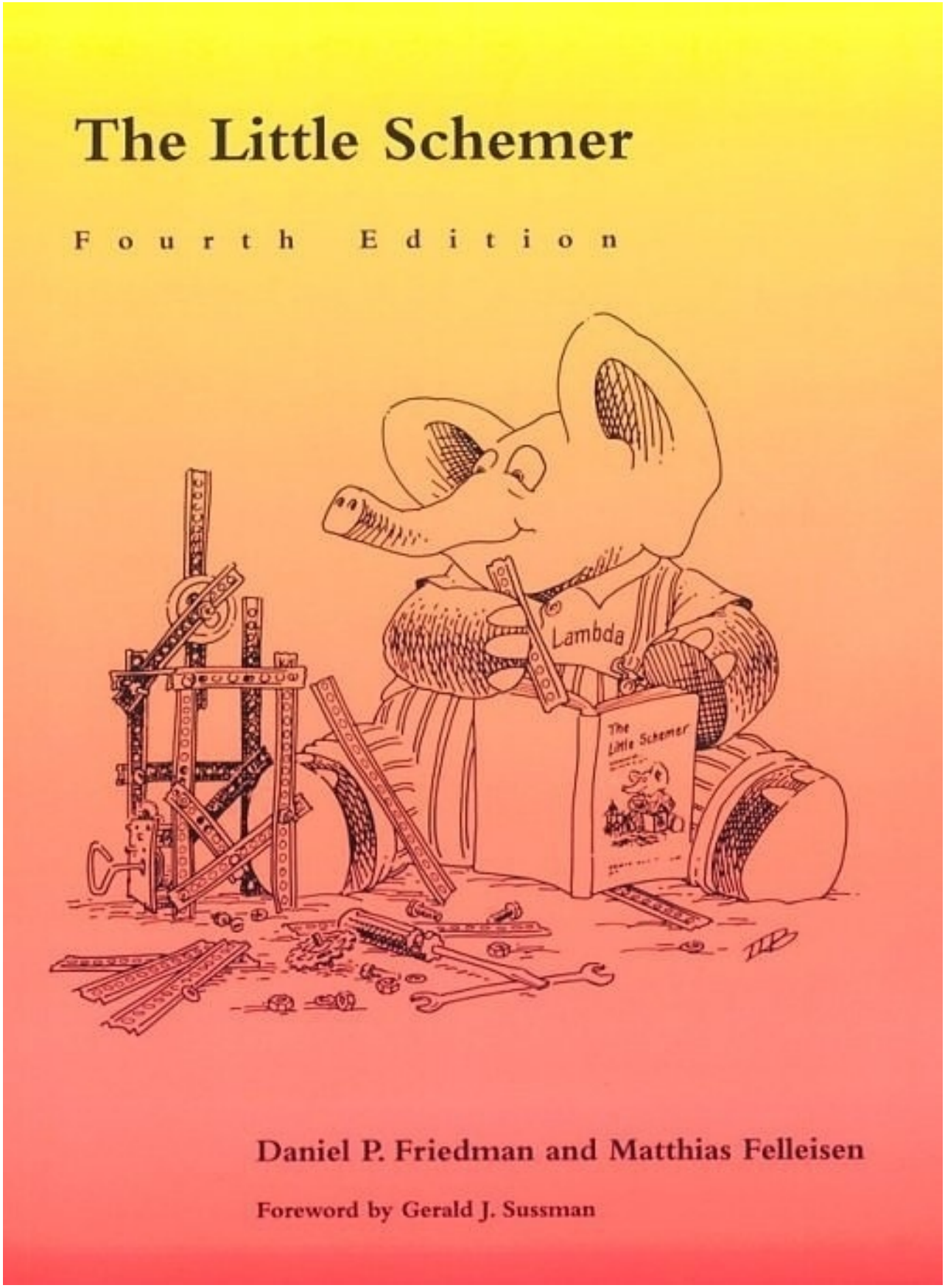
If you follow the instructions at nodejs.org to install NodeJS and JavaScript,[1] you can run an interactive JavaScript REPL on your command line simply by typing `node`. This is how the examples in this book were tested, and what many programmers will do.

On OS X, you have the option of running Safaris's JavaScript engine from the command line, or of installing a TextMate bundle. We didn't use this method to test the examples, so proceed at your own risk. But have fun!

Almost all browsers have a mechanism to function as JavaScript REPLs, allowing you to type JavaScript expressions into a debug console.

Thanks!

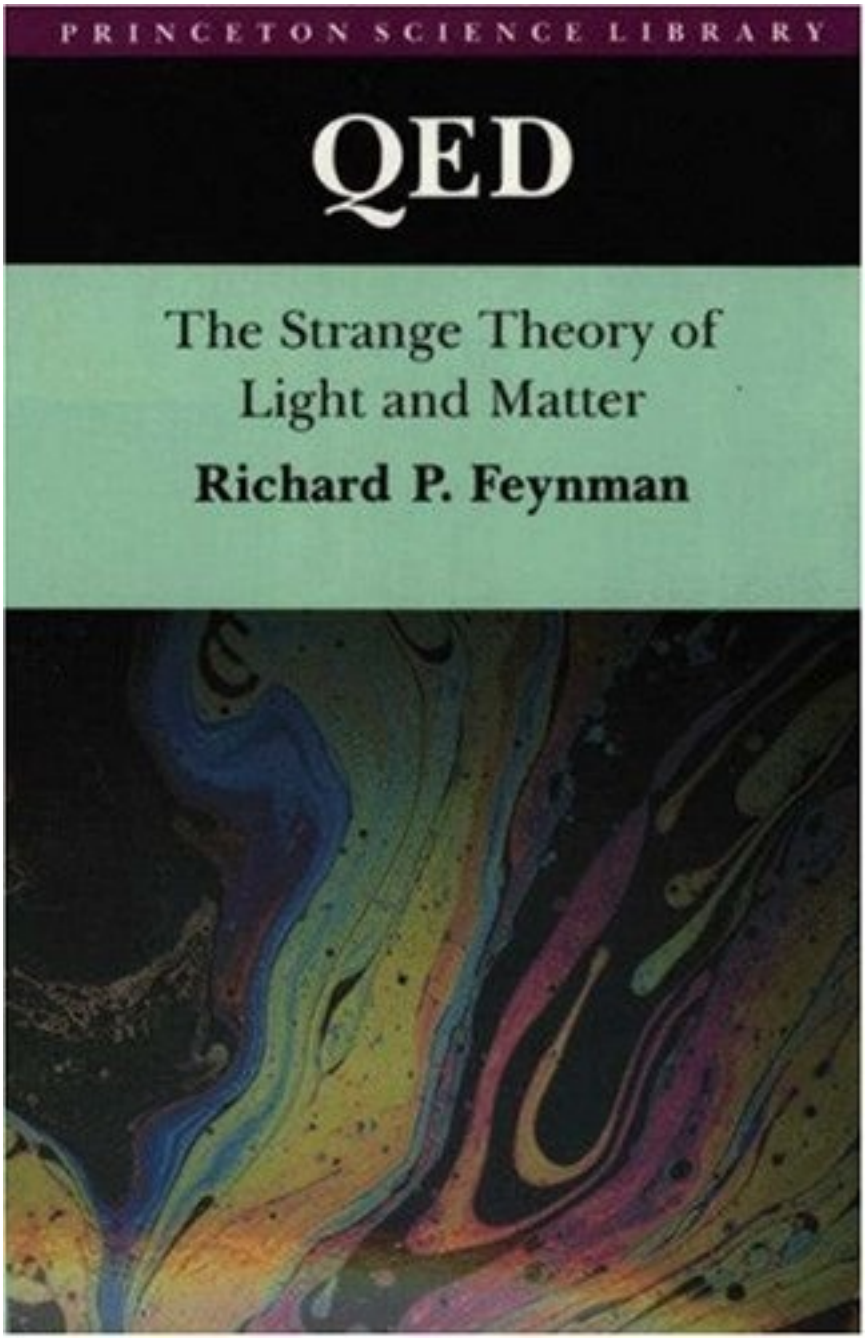
Daniel Friedman and Matthias Felleisen



The Little Schemer

JavaScript Allongé was inspired by The Little Schemer 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**.

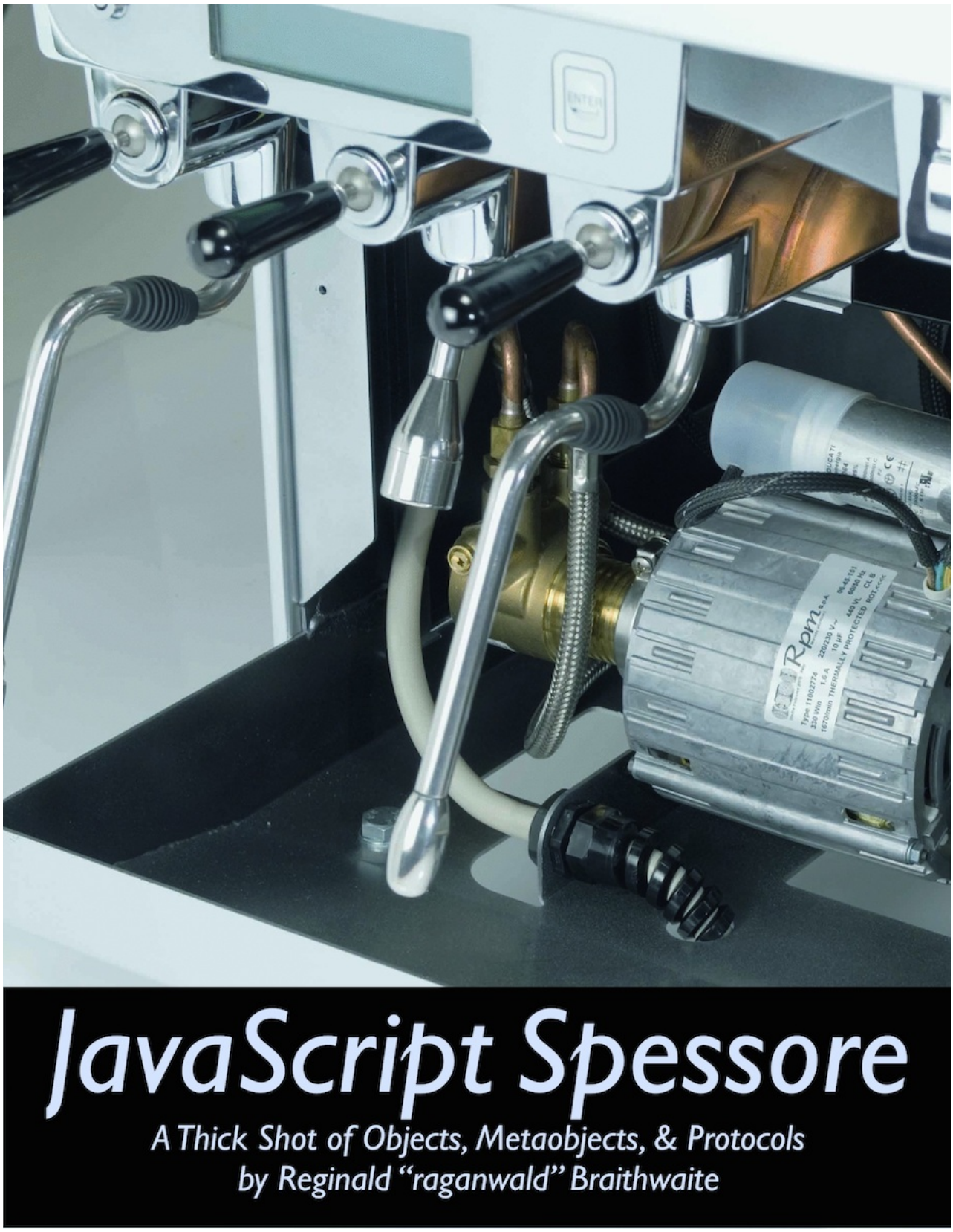
Richard Feynman



QED: The Strange Theory of Light and Matter

Richard Feynman's QED 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.

JavaScript Spessore



JavaScript Spessore

Programming languages are (loosely) defined by their basic activity. In FORTRAN, we program with numbers. In C, we program with pointers. In ML, we program with types. And as JavaScript Allongé explains, in JavaScript we program with functions.

Functions are very interesting building blocks for programs, because they compose: It's easy to build a programming style based on making many small things that can be

combined and recombined to make bigger things.

This is the basis of the vaunted “Unix Philosophy:” Write small utilities and scripts that compose neatly. This is also the JavaScript philosophy: Make small things that can be combined and recombined to make bigger things.

Programming with objects can be done in this style, and JavaScript makes it particularly easy to combine and recombine small parts. Classes can be built from traits instead of from superclasses. Objects can delegate and forward behaviour to helpers and meta-objects. Adaptors can be written to change an object’s interface without needing to create another class in a hierarchy.

JavaScript Spessore is a book that describes this approach to working with objects and metaobjects in JavaScript. It’s exactly the same philosophy as you find in JavaScript Allongé, only it talks to programming with objects instead of programming with functions.

JavaScript Spessore describes how to build JavaScript programs that scale in code, in time, and across a team, using the one technique that has passed the test of time: Objects and metaobjects that have a single responsibility, are decoupled from each other, and can be composed freely.

Now that you’ve read JavaScript Allongé, JavaScript Spessore should be next.

Copyright Notice

images

- The picture of the author is (c) 2008, Joseph Hurtado, All Rights Reserved.
- Cover image (c) 2010, avlxyz. Some rights reserved.
- Double ristretto menu (c) 2010, Michael Allen Smith. Some rights reserved.
- Short espresso shot in a white cup with blunt handle (c) 2007, EVERYDAYLIFEMODERN. Some rights reserved.
- Espresso shot in a caffe molinari cup (c) 2007, EVERYDAYLIFEMODERN. Some rights reserved.
- Beans in a Bag (c) 2008, Stirling Noyes. Some Rights Reserved.
- Free Samples (c) 2011, Myrtle Bech Digitel. Some Rights Reserved.
- Free Coffees image (c) 2010, Michael Francis McCarthy. Some Rights Reserved.
- La Marzocco (c) 2009, Michael Allen Smith. Some rights reserved.

- Cafe Diplomatico (c) 2011, Missi. Some rights reserved.
- Sugar Service (c) 2008 Tiago Fernandes. Some rights reserved.
- Biscotti on a Rack (c) 2010 Kirsten Loza. Some rights reserved.
- Coffee Spoons (c) 2010 Jenny Downing. Some rights reserved.
- Drawing a Doppio (c) 2008 Osman Bas. Some rights reserved.
- Cupping Coffees (c) 2011 Dennis Tang. Some rights reserved.
- Three Coffee Roasters (c) 2009 Michael Allen Smith. Some rights reserved.
- Blue Diedrich Roaster (c) 2010 Michael Allen Smith. Some rights reserved.
- Red Diedrich Roaster (c) 2009 Richard Masoner. Some rights reserved.
- Roaster with Tree Leaves (c) 2007 ting. Some rights reserved.
- Half Drunk (c) 2010 Nicholas Lundgaard. Some rights reserved.
- Anticipation (c) 2012 Paul McCoubrie. Some rights reserved.
- Ooh! (c) 2012 Michael Coghlan. Some rights reserved.
- Intestines of an Espresso Machine (c) 2011 Angie Chung. Some rights reserved.
- Bezzera Espresso Machine (c) 2011 Andrew Nash. Some rights reserved. *Beans Ripening on a Branch (c) 2008 John Pavelka. Some rights reserved.
- Cafe Macchiato on Gazotta Della Sport (c) 2008 Jon Shave. Some rights reserved.
- Jars of Coffee Beans (c) 2012 Memphis CVB. Some rights reserved.
- Types of Coffee Drinks (c) 2012 Michael Coghlan. Some rights reserved.
- Coffee Trees (c) 2011 Dave Townsend. Some rights reserved.
- Cafe do Brasil (c) 2003 Temporalata. Some rights reserved.
- Brown Cups (c) 2007 Michael Allen Smith. Some rights reserved.
- Mirage (c) 2010 Mira Helder. Some rights reserved.
- Coffee Van with Bullet Holes (c) 2006 Jon Crel. Some rights reserved.
- Disassembled Elektra (c) 2009 Nicholas Lundgaard. Some rights reserved.
- Nederland Buffalo Bills Coffee Shop (c) 2009 Charlie Stinchcomb. Some rights reserved.
- For the love of coffee (c) 2007 Lotzman Katzman. Some rights reserved.
- Saltspring Processing Facility Pictures (c) 2011 Kris Krug. Some rights reserved.

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 libraries for JavaScript, CoffeeScript, and Ruby programming such as Method Combinators, Katy, JQuery Combinators, YouAreDaChef, andand, and others.

He writes about programming on "Raganwald," as well as general-purpose ruminations on

“Braythwayt Dot Com”.

contact

Twitter: @raganwald Email: reg@braythwayt.com

Reg “Raganwald” Braithwaite

Instructions for installing NodeJS and modules like JavaScript onto a desktop computer is beyond

the scope of this book, especially given the speed with which things advance. Fortunately, there are always up-to-date instructions on the web.↩