CoffeeScript

Ristretto

brought to you by:

Reg
"Raganwald"
Braithwaite

CoffeeScript Ristretto

An intense cup of code

raganwald

This book is for sale at http://leanpub.com/coffeescript-ristretto

This version was published on 2014-10-29

Tweet This Book!

Please help raganwald by spreading the word about this book on Twitter!

The suggested hashtag for this book is #coffeescriptristretto.

Find out what other people are saying about the book by clicking on this link to search for this hashtag on Twitter:

https://twitter.com/search?q=#coffeescriptristretto

This book is dedicated to my son, Thomas Aston Braithwaite

Contents

A Pull of the Lever: Prefaces

Caffe Molinari

“We always pour our coffee in the ristretto, or restricted, tradition. The coffee is restricted to the most flavourful part of the shot. This tradition offers the heaviest shot, thickest texture and finest flavour that the coffee has to offer.”–David Schomer[1]

[1]http://www.espressovivace.com/archives/9507scr.html

About This Book

> Learning about "for" loops is not learning to program, any more than learning about pencils is learning to draw.–Bret Victor, Learnable Programming[2]

Programming languages are characterized by their syntax and their semantics. The syntax of a language defines its user interface; If you understand a language's syntax, you understand what it makes easy. The semantics of a language defines its capabilities; If you understand a language's semantics, you understand what it does well.

CoffeeScript Ristretto is first and foremost about a book about programming with functions, because its flexible and powerful functions are what make the CoffeeScript[3] programming language so capable, and what CoffeeScript does well.

how this book is organized

CoffeeScript Ristretto 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 Chapter Five. Chapter Six, "An Extra Shot of Ideas," introduces advanced CoffeeScript idioms like method decorators. Since *CoffeeScript Ristretto* is a book about CoffeeScript's semantics, each topic is covered thoroughly, without hand-waving or simplification.

As a result, *CoffeeScript Ristretto* is a rich, dense read, much like the Espresso Ristretto beloved by coffee enthusiasts everywhere.

[2]http://worrydream.com/LearnableProgramming/
[3]http://coffeescript.org

Foreword by Jeremy Ashkenas

"I particularly enjoyed this small book because I've reached for it a hundred times before and come up empty-handed. Large and heavy manuals on object-oriented programming and JavaScript are all around us, but to find a book that tackles the fundamental features of functions and objects in a brief, strong gulp, is rare indeed.

"With the inimitable Mr. Braithwaite as your guide, you'll explore the nature of functions, the nooks and crannies of lexical scope, the essence of reference, the method of mutation, the construction of classes, callbacks, prototypes, promises, and more. And you'll learn a great deal about the internals of CoffeeScript and the semantics of JavaScript along the way. Every feature is broken apart into its basic pieces, and you put it back together yourself, brick by brick.

"This book is dense, but hopefully you've chosen it because you like your ristretto bold and strong. If Reg had pulled it any thicker you'd have to chew it."—Jeremy Ashkenas, CoffeeScript's creator

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.

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 CoffeeScript 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[4], and it returns a value[5], 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 Cafe Cubana. 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 Cafe Cubana is an expression (you can use it to place an order) and a value (you get it back from the barista).

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

```
42
```

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

The answer is, this is both an expression *and* a value.[6] The way you can tell that it's both is very easy: When you type it into CoffeeScript, you get the same thing back, just like our Cafe Cubana:

```
42
  #=> 42
```

[4]https://en.wikipedia.org/wiki/Expression_

[5]https://en.wikipedia.org/wiki/Value_

[6]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.

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.[7] 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:

```
"CoffeeScript" + " " + "Ristretto"
  #=> "CoffeeScript Ristretto"
```

These are "strings," values featured in almost every contemporary computer language. 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 CoffeeScript, we test whether two values are identical with the `is` operator, and whether they are not identical with the `isnt` operator:

```
        2 is 2
                #=> true

        'hello' isnt 'goodbye'
                #=> true
```

How does `is` 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 CoffeeScript, there are four possibilities:

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

First, sometimes, the cups are of different types. One is a demitasse, the other a mug. This corresponds to comparing two things in CoffeeScript 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 is '2'
  #=> false

true isnt '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 CoffeeScript values that have the same type but different "content." For example, the number `5` is not the same thing as the number `2`.

```
true is false
  #=> false

2 isnt 5
  #=> true

'two' is 'five'
  #=> false
```

What if the cups are of the same type *and* the contents are the same? Well, CoffeeScript'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 is 4
  #=> true

(2 + 2 is 4) is (2 isnt 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 CoffeeScript 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 Azzuri 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 CoffeeScript? 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).

Here are some expressions for arrays you can try typing for yourself:

```
[1, 2, 3]
[1,2,2]
[1..3]
```

These are expressions, and you can combine `[]` with other expressions. Go wild with things like:

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

We aren't going to spend a lot of time talking about it, but if you enable multiline mode (with ctrl-v), you can also type things like:

```
[
  1
  2
  3
  ]
```

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:

```
[1..3] is [1,2,3]
[1,2,3] is [1, 2, 3]
[1, 2, 3] is [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 CoffeeScript is generating new cups of coffee with serial numbers on the bottom.

They look the same, but if you examine them with `is`, 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.

interlude...

A short, intense shot of espresso

Wikipedia[8] on Ristretto:

> ”**Ristretto** is a very ‘short’ shot of espresso coffee. Originally this meant pulling a hand press faster than usual using the same amount of water as a regular shot of espresso. Since the water came in contact with the grinds for a much shorter time the caffeine is extracted in reduced ratio to the flavorful coffee oils. The resultant shot could be described as bolder, fuller, with more body and less bitterness.”

[8]https://en.wikipedia.org/wiki/Ristretto

CoffeeScript Ristretto

The perfect Espresso Ristretto begins with the right beans, properly roasted. CoffeeScript Ristretto begins with functions, properly dissected.

The first sip: Functions

Double Shot (2 oz.)
Double Shot Ristretto (1.5 oz.)
Single Shot (1 oz.)
Single Shot Ristretto (.75 oz.)

While Terroir tends toward pretty low body (particularly at its age when we pulled it), the crema's so thin on this shot due to it having been sipped from already!

As Little As Possible About Functions, But No Less

In CoffeeScript, 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 in CoffeeScript. Let's start with the very simplest possible function. In CoffeeScript, it looks like this:[9]

```
->
```

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

```
->
  #=> [Function]
```

What!? Why didn't it type back `->` for us? This *seems* to break our rule that if an expression is also a value, CoffeeScript will give the same value back to us. What's going on? The simplest and easiest answer is that although the CoffeeScript 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 CoffeeScript REPL. If you try the same thing in a browser (using "Try CoffeeScript" at coffeescript.org[10] for example), you'll get something else entirely that isn't CoffeeScript at all, it's JavaScript.

[9]If you have dabbled in CoffeeScript or look at other people's CoffeeScript programs, you may discover that it is also legal to write `->`. Conceptually, `->` is a function with no arguments and no body. `->` is a function with an empty list of arguments and no body. Generally, CoffeeScript programmers prefer `->`, so let's do that.

[10]http://coffeescript.org

I'd prefer something else, but I console myself with the thought 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 () {}` or–in some future version of CoffeeScript–`->`, internally CoffeeScript has a full and proper function.[a]

[a]The exact same thing will happen to you once you figure out how to make an array that contains itself. You'll try to print it out and you'll get `[[Circular]]` back. Never mind, internally CoffeeScript has constructed a perfectly fine Ouroborian array even if it won't try to print it out for you.

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 CoffeeScript parser, we'll enclose our functions in parentheses:

```
(->) is (->)
  #=> 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. Some folks call the arguments the *inputs* to a function. Whether you use the word "inputs" or "arguments," it's certainly a good thing to think of the function's arrow as pointing from the inputs to the output!

Here's how we apply a function to some values in CoffeeScript: 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: `->`, so let's use it. We'll put it in parentheses[11] to keep the parser happy, like we did above: `(->)`. Since we aren't giving it any arguments, we'll simply write `()` after the expression. So we write:

[11]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!

```
(->)()
  #=> undefined
```

What is this `undefined`?

`undefined`

In CoffeeScript, 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, CoffeeScript can print out the value `undefined`.

```
undefined is undefined
  # => true
(->)() is (->)()
  # => true
(->)() is 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 :-)

Speaking of `is undefined`, a common pattern in CoffeeScript programming is to test wither something `isnt undefined`:

```
undefined isnt undefined
  #=> false
'undefined' isnt undefined
  #=> true
false isnt undefined
  #=> true
```

This is so common that a shortcut is provided, the suffix operator `?`:

```
undefined?
  #=> false
'undefined'?
  #=> true
false?
  #=> true
```

You might think that `undefined` in CoffeeScript 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 CoffeeScript, every `undefined` is identical to every other `undefined`.

functions with no arguments

Back to our function. We evaluated this:

```
(->)()
  #=> undefined
```

Let's recall that we were applying the 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, we write the arguments it expects to the left of the `->` and an optional expression to the right. This expression is called the function's *body*. Like this:

`(`*args*`) -> `*body*

There is a funny rule: You can omit the body, and if you do, applying the function always evaluates to `undefined`.[12]

What about functions that have a body? Let's write a few. Here's the rule: We can use *anything* we've already learned how to use as an expression. Cutting and pasting, that means that the following are all expressions that evaluate to functions:

[12]Elsewhere, we've pledged to avoid optional bits that don't add a lot to our understanding. This optional bit gives us an excuse to learn about `undefined`, so that's why it's in. Now that we know this, we see that our expression `->` evaluates to a function taking no arguments and having no expression, therefore when you apply it to no arguments with `(->)()`, you get `undefined`.

```
-> 2
-> 2 + 2
-> "Hello" + " " + "CoffeeScript"
-> true is not false
-> false isnt true
```

And you can evaluate them by typing any of these into CoffeeScript:

```
(-> 2)()
  #=> 2
(-> 2 + 2)()
  #=> 4
(-> "Hello" + " " + "CoffeeScript")()
  #=> "Hello CoffeeScript"
(-> true is not false)()
  #=> true
(-> false isnt true)()
  #=> true
```

We haven't discussed arguments yet, but let's get clever with what we already have.

functions that evaluate to functions

If an expression that evaluates to a function is, well, an expression, and if a function expression 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:

```
-> ->
```

That's a function! It's a function that when applied, evaluates to a function that when applied, evaluates to `undefined`. Watch and see:

```
-> ->
  #=> [Function]
```

It evaluates to a function…

```
(-> ->)()
  #=> [Function]
```

That when applied, evaluates to a function…

```
(-> ->)()()
  #=> undefined
```

That when applied, evaluates to `undefined`. Likewise:

```
-> -> true
```

That's a function! It's a function that when applied, evaluates to a function, that when applied, evaluates to `true`:

```
(-> -> true)()()
  #=> true
```

Well. We've been very clever, but so far this all seems very abstract and computer science-y. 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.

showering felicitous encouragement to incentivise the practice of skewing the distribution of expression length towards the minimal mode

Or, *In praise of keeping it short.*

When describing the behaviour of functions, we often use the expression "that when applied, evaluates to…" For example, "The function `(x, y) -> x + y` is a function, that when applied to two integer arguments, evaluates to the sum of the arguments."[13] This is technically correct. But a mouthful. Another expression you will often hear is "returns," as in "The function `(x, y) -> x + y` is a function that returns the sum of its arguments."

"Returns" is a little less precise, and is context dependant. But it suits our purposes, so we will often use it. But when we use it, we will *always* mean "when applied, evaluates to…"

And with that's let's move on!

[13]It does something else when the first argument is a string, but let's ignore that bit of pedantry for now.

Ah. I'd Like to Have an Argument, Please.[14]

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

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

Let's make a function with an argument:

```
(room) ->
```

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

```
(room, board) ->
```

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

```
(diameter) -> diameter * 3.14159265
```

It's a function for calculating the circumference of a circle given the radius. I read that aloud as "When applied to a value representing the diameter, this function *gives* (that's my word for the arrow) the diameter times 3.14159265."

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

```
((diameter) -> diameter * 3.14159265)(2)
  #=> 6.2831853
```

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

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

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

a quick summary of functions and bodies

How arguments are used in a body's expression is probably perfectly obvious to you from the examples, especially if you've used any programming language (except, possibly for the dialect of BASIC 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`), and some special forms like `[1, 2, 3]` for creating arrays out of expressions and `(`*arguments*`) ->`*body-expression* for creating functions.

This loose definition is recursive, so we can intuit (or use our experience with other languages) that since a function has an expression on its right hand side, we can write a function that has a function as its expression, or an array that contains another array expression. Or a function that gives an array, an array of functions, a function that gives an array of functions, and so forth:

```
-> ->
-> [ 1, 2, 3]
[1, [2, 3], 4]
-> [(-> 1), (-> 2), (-> 3)]
```

call by value

Like most contemporary programming languages, CoffeeScript uses the "call by value" evaluation strategy[15]. That's a $2.75 way of saying that when you write some code that appears to apply a function to an expression or expressions, CoffeeScript evaluates all of those expressions and applies the functions to the resulting value(s).

So when you write:

```
((diameter) -> diameter * 3.14159265)(1 + 1)
  #=> 6.2831853
```

What happened internally is that the expression `1 + 1` was evaluated first, resulting in `2`. Then our circumference function was applied to `2`.[16]

[15]http://en.wikipedia.org/wiki/Evaluation_strategy

[16]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.

variables and bindings

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

```
(x) -> (y) -> x
```

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

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

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

Every time a function is invoked ("invoked" is a synonym for "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:

```
((x) -> x)(2)
  #=> 2
```

What happens is this:

1. CoffeeScript parses this whole thing as an expression made up of several sub-expressions.
2. It then starts evaluating the expression, including evaluating sub-expressions
3. One sub-expression, `(x) -> x` evaluates to a function.

4. Another, `2`, evaluates to the number `2`.
5. CoffeeScript 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[17] 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 CoffeeScript's *value types* from its *reference types*. At that time, we looked at how CoffeeScript 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 CoffeeScript 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 CoffeeScript binds names to values, but we didn't say what it means to bind a name to a value. Now we can elaborate: When CoffeeScript binds a name to a value type value, 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 CoffeeScript can make as many copies of strings, numbers, or booleans as it wishes.

What about reference types? CoffeeScript cannot place a copy of an array or object in an environment, because the copy would not be identical to the original. So instead, CoffeeScript does not place reference values in any environment. CoffeeScript places *references* to reference types in environments, and when the value needs to be used, CoffeeScript uses the reference to obtain the original.

Because many references can share the same value, and because CoffeeScript passes references as arguments, CoffeeScript 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 you apply it to:

[17]http://json.org/

```
(value) ->
  ((copy) ->
    copy is value
  )(value)
```

Closures and Scope

Before we explain `(x) -> (y) -> x`, we're going to toss in something that doesn't directly affect our explanation, but makes things easier to see *visually*. Up to now, every function has looked like this: `(`*arguments*`) ->` *body*. There's another way to write functions. For example here's the other way to write `(x) -> x`:

```
(x) ->
  x
```

You get the idea: You can indent the body instead of putting it on the same line. Let's introduce a new term: `(x) ->` is the function's *signature*, and `x` is its *body*, just as we've mentioned before.

That means inductively we can also write `(x) -> (y) -> x` in two other ways:

```
(x) ->
  (y) -> x
```

Or:

```
(x) ->
  (y) ->
    x
```

The indents help us see that the `x` is the body "belonging to" a function with signature `(y) ->`, and that it belongs to a function with signature `(x) ->`.

Time to see how a function within a function works:

```
((x) ->
  (y) ->
    x
)(1)(2)
  #=> 1
```

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

```
((x) ->
  (y) ->
    x
)(1)
  #=> [Function]
```

The environment belonging to the function with signature `(x) ->` 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 `(x) ->`'s body is:

```
  (y) ->
    x
```

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

```
  ((y) ->
    x)(2)
```

So we seem to get a new environment `{y: 2, ...}`. 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 characteristic of CoffeeScript and languages in the same family: Whether they allow things like functions to nest inside each other, and if so, how they handle variables from "outside" of a function that are referenced inside a function. For example, here's the equivalent code in Ruby:

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

Now let's have an Espresso before we continue!

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

The function `(y) -> x` is interesting. It contains a *free variable*, `x`.[18] 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

[18] You may also hear the term "non-local variable." Both are correct.

passing in an argument with the same name. Since the function `(y) -> 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:

```
->

(x) ->
  x

(x) ->
  (y) ->
    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 in side the other. `(y) ->` has a free variable, but the entire expression refers to `(x) ->`, and it doesn't have a free variable: The only variable anywhere in its body is `x`, which is certainly bound within `(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 free function `(x, y) -> x + y`, we know exactly what it does with `(2, 2)`. But what about this closure: `(y) -> 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 `((x) -> (y) -> x)(1)` is `{x: 1, ...}` and that the environment for `((y) -> x)(2)` is `{y: 2, ...}`? Let's fill in the blanks!

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

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

[a] http://www.amzn.com/0192801422?tag=raganwald001-20
[b] https://leanpub.com/combinators

Functions can have grandparents too:

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

This function does much the same thing as:

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

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

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

[a]https://en.wikipedia.org/wiki/Currying
[b]https://en.wikipedia.org/wiki/Partial_application

shadowy variables from a shadowy planet

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

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

The function `(x, y) -> x + y` is a pure function, because its `x` is defined within its own environment. Although its parent also defines an `x`, it is ignored when evaluating `x + y`. CoffeeScript 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:

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

When evaluating `x + y + z`, CoffeeScript 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.

There can be spirited discussions over shadowing variables. Some people argue that shadowing is a good thing, as it allows small pieces of code to be understood without checking any enclosing environments. Others argue that it's a bad thing, as it creates a confusing situation where two things with the same name actually are different things, a lot like naming both of your twins "Lesley."

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 CoffeeScript provides for working with variables and mutable state.

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

CoffeeScript always has the notion of at least one environment we do not control: A global environment in which many useful things are bound such as libraries full of standard functions. So when you invoke `((x) -> x)(1)` in the REPL, its full environment is going to look like this: `{x: 1, '..':` *global environment*`}`. When you use CoffeeScript to compile physical files for use in node or web applications, CoffeeScript does something interesting: It wraps your code in an invisible function, like this:

```
(->
  ((x) -> x)(1)
  )()
```

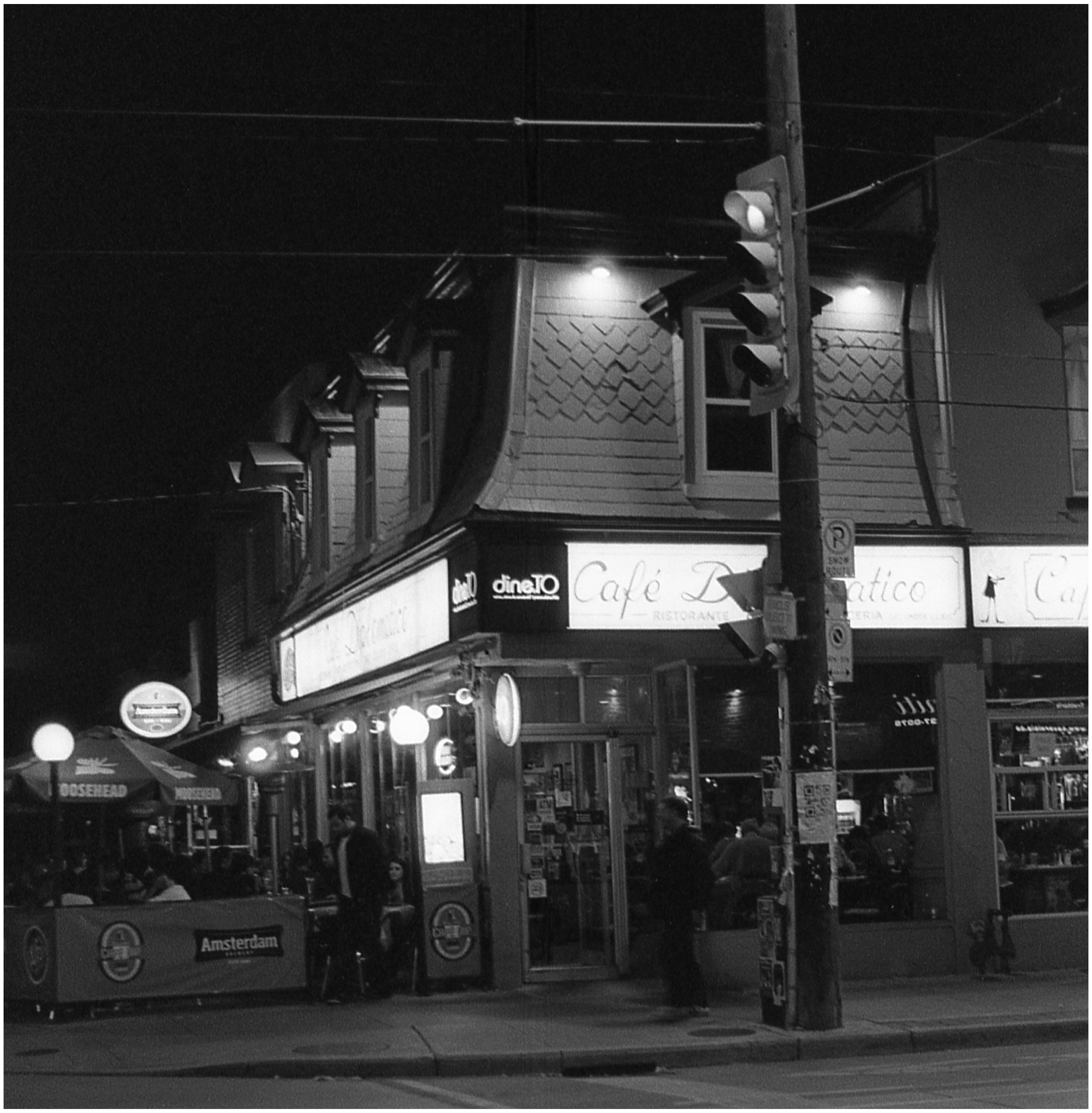
The effect of this is to insert a new, empty environment in between the global environment and your own functions: `{x: 1, '..': {'..':` *global environment*`}}`. As we'll see when we discuss mutable state, this helps to prevent programmers from accidentally changing the global state that is shared by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

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 to the value of the last expression evaluated or `undedfined`.
- Function application creates a scope. Scopes are nested and free variable references closed over.
- Variables can shadow variables in an enclosing scope.

Slurp: More About Functions and Scope



Cafe Diplomatico in Toronto's Little Italy

Let Me Show You What To Do

let

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

```
(diameter) ->
  diameter * 3.14159265
```

What is this "3.14159265" number? Pi[19], obviously. We'd like to name it so that we can write something like:

```
(diameter) ->
  diameter * Pi
```

In order to bind `3.14159265` to the name `Pi`, we'll need a function with a parameter of `Pi` and an argument of `3.14159265`:

```
((Pi) ->
  ???
)(3.14159265)
```

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

```
((Pi) ->
  (diameter) ->
    diameter * Pi
)(3.14159265)
```

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

[19] https://en.wikipedia.org/wiki/Pi

```
((Pi) ->
  (diameter) ->
    diameter * Pi
)(3.14159265)(2)
  #=> 6.2831853
```

That works! We can bind anything we want and use it in a function by wrapping the function in another function that is immediately invoked with the value we want to bind. This “functional programming pattern” was popularized in the Lisp programming language more than 50 years ago, where it is called `let`[20].[21] Although CoffeeScript doesn’t have a `let` keyword, when we discuss this programming pattern we will call it `let`.

`let` works,[22] but only a masochist would write programs this way in CoffeeScript. 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.

do

CoffeeScript programmers often wish to create a new environment and bind some values to names within it as `let` does. To make this easier to read and write, CoffeeScript provides some *syntactic sugar* called `do`.[23]

[20]http://jtra.cz/stuff/lisp/sclr/let.html

[21]`let` has made its way into other languages like JavaScript.

[22]`let` is limited in some ways. For example, you can’t define a recursive function without some fixed point combinator backflips. This will be discussed later when we look at the related pattern `letrec`.

[23]“Syntactic sugar causes cancer of the semicolon”–Alan Perlis

Italians seem to prefer espresso with plenty of sugar, while North Americans often drink it without

This is what our example looks like using `do`:

```
do (Pi = 3.14159265) ->
  (diameter) ->
    diameter * Pi
```

Much, MUCH cleaner.

If you need to create more than one binding, you separate them with commas:

```
do (republican = 'Romney', democrat = 'Obama') ->
  democrat
```

The value on the right side can be any expression. Try this for yourself:

```
do (Pi = 3.14159265, diameter = (radius) -> radius * 2) ->
  (radius) ->
    diameter(radius) * Pi
```

Did you try the example above? Did you notice what we slipped in? Yes, obviously, the value of a binding can be any expression. But notice also that we can invoke a function on any expression evaluating to a function, including a variable that looks up a binding in the environment.

Dozens of pages into the book, we’re finally calling a function the way you’ll see functions being called in most production code. Sheesh.

A Simple Question

Both of the following produce the exact same result:

```
do (Pi = 3.14159265) ->
  (diameter) ->
    diameter * Pi
```

And:

```
(diameter) ->
  do (Pi = 3.14159265) ->
    diameter * Pi
```

Why do we habitually prefer the former?

To understand this, we're going to take a simple step towards more complex, state-full programs by introducing sequences of expressions. If we had no other tools, we could evaluate a series of expressions with some legerdemain like this:

```
do (ignore1 = foo(),
    ignore2 = bar(),
    ignore3 = blitz(),
    value = bash()) ->
  bash
```

Or perhaps like this:

```
do (ignore = [foo(),  bar(), blitz()], value = bash()) ->
  bash
```

Or even this:

```
[
  foo()
  bar()
  blitz() ] and bash()
```

Any of these would evaluate `foo()`, `bar()`, `blitz()`, and then return the value of `bash()` (whatever they might be).

Why doesn't `foo() and bar() and blitz() and bash()` work reliably?

But let's learn another handy CoffeeScript feature, again because it helps us focus on what is actually going on. Whenever you want to work with the body of a function, you can always have it evaluate a simple sequence of one or more expressions by indenting them. The value of the body is the value of the final expression.

So in that case, we can write something like:

```
do ->
  foo()
  bar()
  blitz()
  bash()
```

Anywhere a simple expression is allowed, you could use a `do` with a sequence. This doesn't come up as much as you might think, because many of the places you want to do this, CoffeeScript already lets you indent and include more than one expression. For example, in a function body:

```
(foo) ->
  bar(foo)
  bash(foo)
  foo('blitz')
```

So back to our question. Here's a test framework:

```
do (circumference = do (Pi = 3.14159265) ->
                      (diameter) ->
                        diameter * Pi) ->
  circumference(1)
  circumference(2)
  circumference(3)
  circumference(4)
  circumference(5)
  circumference(6)
  circumference(7)
  circumference(8)
  circumference(9)
  circumference(10)
    #=> 31.4159265
```

Let's think about how many functions we are invoking. When this is first invoked, We invoke the outer `do (circumference = (...) ->`. As part of doing that, we invoke `do (Pi = 3.14159265) ->` and bind the result to `circumference`. Then every time we invoke `circumference`, we invoke `(diameter) ->`. All together, twelve.

But with:

```
do (circumference = (diameter) ->
                      do (Pi = 3.14159265) ->
                        diameter * Pi) ->
  circumference(1)
  circumference(2)
  circumference(3)
  circumference(4)
  circumference(5)
  circumference(6)
  circumference(7)
  circumference(8)
  circumference(9)
  circumference(10)
    #=> 31.4159265
```

What happens? There's one outer `do (circumference = (...) ->`, same as before. And then every time we invoke `circumference`, we also invoke `do (Pi = 3.14159265) ->`, so we have a total of twenty-one function invocations. This is nearly twice as expensive.

Making Things Easy

In *CoffeeScript Ristretto*, we are focusing on CoffeeScript's *semantics*, the meaning of CoffeeScript programs. As we go along, we're learning just enough CoffeeScript to understand the next concept simply and directly.

CoffeeScript actually supports a number of syntactic conveniences for making programs extremely readable, by which we mean, making them communicate their intent without asking the programmer to struggle in a Turing Tarpit[24], no matter how elegant.

if i were a rich man

For example, it is possible to implement boolean logic using functions, by carefully combining the Identity (`(x) -> x`), Kestrel (`(x) -> (y) -> x`), and Vireo (`(x) -> (y) -> (z) -> z(x(y))`) functions using a clever trick. It look something like this:

```
do (I = ((x) -> x),
    K = ((x) -> (y) -> x),
    V = ((x) -> (y) -> (z) -> z(x(y)))
) ->
  do (t = K, f = K(I)) ->
    # ...
    # implement logical operators here
    # ...
```

Did you notice that I slipped a new language feature in, one that allegedly allows a programmer to communicate their intent? Comments in CoffeeScript are signalled by a `#` and continue to the end of the line, much like `//` in C++ or JavaScript, and exactly like `#` in Ruby. If you have ever used `C` to make a comment line in FORTRAN, you are a real programmer[a] and ought not to be fooling around with a quiche-eater's language.

[a] http://www.pbm.com/~lindahl/real.programmers.html

This is extraordinarily fascinating computer science stuff, but you can read about that elsewhere[25]. CoffeeScript supplies `true`, `false`, `and`, `or`, and `not` so you don't need to roll your own out of functions. But while we're talking about logic, CoffeeScript also supplies conditional branches of execution, and we'll use those in examples to come.

The syntax is remarkably simple. Here's a conditional expression[26]:

[24] https://en.wikipedia.org/wiki/Turing_tarpit
[25] http://www.amzn.com/0192801422?tag=raganwald001-20
[26] http://coffeescript.org/#conditionals

```
if d < 32 then 'freezing' else 'warm'
```

Since it's an expression, you can put it in parentheses and stick it anywhere you like, including inside another conditional:

```
(d) ->
  if d < 32 then 'solid' else if d < 212 then 'liquid' else 'gas'
```

Like function bodies, there is an indented form that can be more readable:

```
(d) ->
  if d < 32
    'solid'
  else
    if d < 212
      'liquid'
    else
      'gas'
```

And the indented lines can have multiple expressions should you so desire:

```
if frobbish?
  alert("Frobbish value: #{frobbish}")
  snarglivate(frobbish)
  frobbish
```

Summary

More About Functions And Scope

- `let` is an idiom where we create a function and call it immediately in order to bind values to names.
- CoffeeScript uses `do` as syntactic sugar for `let`.
- CoffeeScript's comments are signalled with `#`.
- CoffeeScript has `if`, `then`, `'else` and boolean logic.

interlude...

A beautiful espresso machine

Michael Allen Smith[27] on Ristretto:

> “It must have been 1996. I was living in South Tampa at the time and the area finally got a great coffee house. The place was Jet City Espresso. Don’t go looking for it. It is no longer there. As the name implies, the owner Jessica was from Seattle and shared her coffee knowledge with her customers. After ordering numerous americanos and espressos, Jessica thought it was time I tried a ristretto. I expected the short pull of the espresso shot would result in a more bitter flavor. To my delight the shot was actually a sweeter and more intense version of her espresso blend.”

[27] http://www.ineedcoffee.com/07/ristretto-rant/

References, Identity, Arrays, and Objects

a simple question

Consider this code:

```
do (x = 'June 14, 1962') ->
  do (y = x) ->
    x is 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?

```
do (x = [2012, 6, 14]) ->
    do (y = x) ->
      x is 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 `do (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:

```
do (x = [2012, 6, 14]) ->
  ((y) ->
    x is 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

CoffeeScript 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:

```
do (unique = (-> ->)) ->

  unique()
    # => [Function]
  unique() is unique()
    # false
```

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

```
  do (x = unique()) ->
    do (y = x) ->
      x is 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?

```
  do (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 CoffeeScript, 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.[28]

Let's test it:

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

```
do (unique = (-> ->)) ->
  do (x = unique()) ->
    do (a = [ x ]) ->
      a[0] is 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:

```
do (unique = (-> ->)) ->
  do (x = unique(), y = unique(), z = unique()) ->
    do (a = [ x, y, z ]) ->
      a[0] is x and a[1] is y and a[2] is z
  #=> true
```

references and objects

CoffeeScript 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 CoffeeScript, Objects[29] are values that can store other values by name (including functions). The most common syntax for creating an object is simple:

```
{ year: 2012, month: 6, day: 14 }
```

Two objects created this way have differing identities, just like arrays:

```
{ year: 2012, month: 6, day: 14 } is { 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:

```
do (unique = (-> ->)) ->
  do (x = unique(), y = unique(), z = unique()) ->
    do (o = { a: x, b: y, c: z }) ->
      o['a'] is x and o['b'] is y and o['c'] is 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:

[29]Tradition would have us call objects that don’t contain any functions “POCOs,” meaning Plain Old CoffeeScript Objects. Given that *poco a poco* means “little by little” in musical notation, I’m tempted to go along with that.

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

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

```
do ( Mathematics = { abs: (a) -> if a < 0 then -a else a }) ->
  Mathematics.abs(-5)
  #=> 5
```

Funny we should mention `Mathematics`. If you recall, CoffeeScript provides a global environment that contains some existing object 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`.

```
Math.abs(-5)
  #=> 5
```

Stir the Espresso: Objects, Mutation, and State

Life measured out by coffee spoons

So far, we have discussed what many call "pure functional" programming, where every expression is necessarily idempotent[30], because we have no way of changing state within a program using the tools we have examined.

It's time to change *everything*.

[30]https://en.wikipedia.org/wiki/Idempotence

Reassignment and Mutation

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

```
do (age = 49) ->
  age = 50
  age
  #=> 50
```

In CoffeeScript, nearly everything is an expression, including statements that assign a value to a variable, so we could just as easily write `do (age = 49) -> age = 50`.

We took the time to carefully examine what happens with bindings in environments. Let's take the time to fully 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:

```
do (age = 49) ->
  do (age = 50) ->
    # yadda yadda
  age
  #=> 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:

```
{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`, reassigning it in a nested environment changes the original:

```
do (age = 49) ->
  do (height = 1.85) ->
    age = 50
  age
  #=> 50
```

Like evaluating variable labels, when a binding is rebound, CoffeeScript 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.

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:

```
do (oneTwoThree = [1, 2, 3]) ->
  oneTwoThree[0] = 'one'
  oneTwoThree
  #=> [ 'one', 2, 3 ]
```

You can even add a value:

```
do (oneTwoThree = [1, 2, 3]) ->
  oneTwoThree[3] = 'four'
  oneTwoThree
  #=> [ 1, 2, 3, 'four' ]
```

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

```
do (name = {firstName: 'Leonard', lastName: 'Braithwaite'}) ->
  name.middleName = 'Austin'
  name
  #=> { firstName: 'Leonard',
  #     lastName: 'Braithwaite',
  #     middleName: 'Austin' }
```

We have established that CoffeeScript's semantics allow for two different bindings to refer to the same value. For example:

```
do (allHallowsEve = [2012, 10, 31]) ->
  halloween = allHallowsEve
```

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

```
do (allHallowsEve = [2012, 10, 31]) ->
  do (allHallowsEve) ->
    # ...
```

Hello, what's this? What does `do (allHallowsEve) ->` mean? Well, when you put a name in the argument list for `do ->` but you don't supply a value, CoffeeScript assumes you are deliberately trying to shadow a variable. It acts as if you'd written:

```
((allHallowsEve) ->
  # ...
)(allHallowsEve)
```

There are two nested environments, and each one binds the name `allHallowsEve` 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:

```
do (allHallowsEve = [2012, 10, 31]) ->
  do (allHallowsEve) ->
    allHallowsEve = [2013, 10, 31]
  allHallowsEve
  #=> [ 2012, 10, 31 ]
```

The outer value of `allHallowsEve` was not changed because all we did was rebind the name `allHallowsEve` within the inner environment. However, what happens if we *mutate* the value in the inner environment?

```
do (allHallowsEve = [2012, 10, 31]) ->
  do (allHallowsEve) ->
    allHallowsEve[0] = 2013
  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.

The same thing is true whenever you have multiple aliases to the same value:

```
do (greatUncle = undefined, grandMother = undefined) ->
  greatUncle = {firstName: 'Leonard', lastName: 'Braithwaite'}
  grandMother = greatUncle
  grandMother['firstName'] = 'Lois'
  grandMother['lastName'] = 'Barzey'
  greatUncle
  #=> { firstName: 'Lois', lastName: 'Barzey' }
```

This example uses the `letrec` pattern for declaring bindings. Now that we've finished with mutation and aliases, let's have a look at it.

letrec

One way to exploit reassignment is to "declare" your bindings with `do` and bind them to something temporarily, and then rebind them inline, like so:

```
do (identity = undefined, kestrel = undefined) ->
  identity = (x) -> x
  kestrel = (x) -> (y) -> x
```

This pattern is called `letrec` after the Lisp special form. Recall that `let` looks like this in CoffeeScript:

```
do (identity = ((x) -> x), kestrel = (x) -> (y) -> x) ->
```

To see how `letrec` differs from `let`, consider writing a recursive function[31] like `pow`. `pow` takes two arguments, `n` and `p`, and returns `n` raised to the `p`th power. For simplicity, we'll assume that `p` is an integer.

[31]You may also find fixed point combinators interesting.

```
do (pow = undefined) ->
  pow = (n, p) ->
    if p < 0
      1/pow(n, -p)
    else if p is 0
      1
    else if p is 1
      n
    else
      do (half = pow(n, Math.floor(p/2)), remainder = pow(n, p % 2)) ->
        half * half * remainder
```

In order for `pow` to call itself, `pow` must be bound in the environment in which `pow` is defined. This wouldn't work if we tried to bind `pow` in the `do` itself. Here's a misguided attempt to create a recursive function using `let`:

```
do (odd = (n) -> if n is 0 then false else not odd(n-1)) ->
  odd(5)
```

To see why this doesn't work, recall that this is equivalent to writing:

```
((odd) ->
  odd(5)
)( (n) -> if n is 0 then false else not odd(n-1) )
```

The expression `(n) -> if n is 0 then false else not odd(n-1)` is evaluated in the parent environment, where `odd` hasn't been bound yet. Whereas, if we wrote `odd` with `letrec`, it would look like this:

```
do (odd = undefined) ->
  odd = (n) -> if n is 0 then false else not odd(n-1)
  odd(5)
```

Which is equivalent to:

```
((odd) ->
  odd = (n) -> if n is 0 then false else not odd(n-1)
  odd(5)
)( undefined )
```

Now the `odd` function is bound in an environment that has a binding for the name `odd`. `letrec` also allows you to make expressions that depend upon each other, recursively or otherwise, such as:

```
do (I = undefined, K = undefined, T = undefined, F = undefined) ->
  I = (x) -> x
  K = (x) -> (y) -> x
  T = K
  F = K(I)
```

takeaway

CoffeeScript 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.

The `letrec` pattern allows us to bind interdependent and recursive expressions.

Normal Variables

Now that we've discussed reassignment, it's time to discuss *assignment*.

> It sounds odd to say we've reassigned things without assigning them. Up to now, we've *bound* values to names through arguments, and `do`, which is really syntactic sugar for the `let` pattern.

In CoffeeScript, the syntax for assignment is identical to the syntax for reassignment:

```
birthday = { year: 1962, month: 6, day: 14 }
```

The difference comes when there is no value bound to the name `birthday` in any of the user-defined environments. In that case, CoffeeScript creates one in the current function's environment. The current function is any of the following:

1. A function created with an arrow operator (`->` that we've seen, and `=>` that we'll see when we look at objects in more detail).
2. A function created with the `do` syntax.
3. When compiling CoffeeScript in files, an empty `do ->` is invisibly created to enclose the entire file.

One good consequence of this feature is that you can dispense with all of the nested `do (...) ->` expressions you've seen so far if you wish. You can boldly write things like:

```
identity = (x) -> x
kestrel = (x) -> (y) -> x
truth = kestrel
falsehood = kestrel(identity)
```

You can also do your assignments wherever you like in a function, not just at the top. Some feel this makes code more readable by putting variable definitions closer to their use.

There are two unfortunate consequences. The first is that a misspelling creates a new binding rather than resulting in an error:

```
do (age = 49) ->
  # ...
  agee = 50
  # ...
  age
  #=> 49, not 50
```

The second is that you may accidentally alias an existing variable if you are not careful. If you're in the habit of creating a lot of your variables with assignments rather than with `do`, you must be careful to scan the source of all of your function's parents to ensure you haven't accidentally reused the name of an existing binding.[32]

> CoffeeScript calls creating new bindings with assignment "normal ," because it's how most programmers normally create bindings. Just remember that if anyone criticizes CoffeeScript for being loose with scoping and aliases, you can always show them how to use `do` to emulate `let` and `letrec`.

un-do

So, should we use `do` to bind variables or should we use "normal" variables? This is a very interesting question. Using `do` has a certain number of technical benefits. Then again, Jeremy Ashkenas, CoffeeScript's creator, only uses `do` when it's necessary, and most CoffeeScript programmers follow his lead. It hasn't done them any harm.

So here's what we suggest:

When writing new software, use Normal variables as much as possible. If and when you find there's a scoping problem, you can refactor to `do`, meaning, you can change a normal variable into a variable bound with `do` to solve the problem.

Programming philosophy is a little outside of the scope of this book, but there is a general principle worth knowing: A good programmer is familiar with many design patterns, idioms, and constructions. However, the good programmer does not attempt to design them into every piece of code from the outset. Instead, the good programmer proceeds along a simple, direct, and clear path until difficulties arise. Then, and only then, does the good programmer refactor to a pattern. In the end, the code is simple where it does not solve a difficult or edge case, and uses a technique or idiom where there is a problem that needed solving.

`do` is a good pattern to know and deeply understand, but it is generally sufficient to write with normal variables. If you see a lot of `do` in this book, that is because we are writing to be excruciatingly clear, not to construct software that is easy to read and maintain in a team setting.

[32]It could be worse. One very popular language assumes that if you haven't otherwise declared a variable local to a function, you must want a global variable that may clobber an existing global variable used by any piece of code in any file or module.

Comprehensions

Cupping Grinds

If you're the type of person who can "Write Lisp in any language," you could set about writing entire CoffeeScript programs using `let` and `letrec` patterns such that you don't have *any* normal variables. But being a CoffeeScript programmer, you will no doubt embrace normal variables. As you dive into CoffeeScript, you'll discover many helpful features that aren't "Lisp-y." Eschewing them is to cut against CoffeeScript's grain. One of those features is the comprehension[33], a mechanism for working with collections that was popularized by Python.

Here's a sample comprehension:

[33]http://coffeescript.org/#loops

```
        names = ['algernon', 'sabine', 'rupert', 'theodora']

"Hello #{yourName}" for yourName in names
  #=> [ 'Hello algernon',
  #     'Hello sabine',
  #     'Hello rupert',
  #     'Hello theodora' ]
```

An alternate syntax for the same thing that supports multiple expressions is:

```
for yourName in names
  "Hello #{yourName}"
  #=> [ 'Hello algernon',
  #     'Hello sabine',
  #     'Hello rupert',
  #     'Hello theodora' ]
```

Here's a question: There's a variable reference `yourName` in this code. Is it somehow bound to a new environment in the comprehension? Or is it a "normal variable" that is either bound in the current function's environment or in a parent function's environment?

Let's try it and see:

```
yourName = 'clyde'
"Hello #{yourName}" for yourName in names
yourName
  #=> 'theodora'
```

It's a normal variable. If it was somehow 'local' to the comprehension, `yourName` would still be `clyde` as the comprehension's binding would shadow the current environment's binding. This is usually fine, as creating a new environment for every comprehension could have performance implications.

However, there are two times you don't want that to happen. First, you might want `yourName` to shadow the existing `yourName` binding. You can use `do` to fix that:

```
yourName = 'clyde'
do (yourName) ->
  "Hello #{yourName}" for yourName in names
yourName
  #=> `clyde`
```

Recall that when you put a name in the argument list for `do ->` but you don't supply a value, CoffeeScript assumes you are deliberately trying to shadow a variable. It acts as if you'd written:

```
yourName = 'clyde'
((yourName) ->
  "Hello #{yourName}" for yourName in names
)(yourName)
yourName
  #=> `clyde`
```

So technically, the inner `yourName` will be bound to the same value as the outer `yourName` initially, but as the comprehension is evaluated, that value will be overwritten in the inner environment but not the outer environment.

preventing a subtle comprehensions bug

Consider this variation of the above comprehension:

```
for myName in names
  (yourName) -> "Hello #{yourName}, my name is #{myName}"
```

Now what we want is four functions, each of which can generate a sentence like "Hello reader, my name is rupert". We can test that with a comprehension:

```
fn('reader') for fn in for myName in names
  (yourName) -> "Hello #{yourName}, my name is #{myName}"
    #=> [ 'Hello reader, my name is theodora',
    #     'Hello reader, my name is theodora',
    #     'Hello reader, my name is theodora',
    #     'Hello reader, my name is theodora' ]
```

WTF!?

If we consider our model for binding, we'll quickly discover the problem. Each of the functions we generate has a closure that consists of a function and a local environment. `yourName` is bound in its local environment, but `myName` is bound in the comprehension's environment. At the time each closure was created, `myName` was bound to one of the four names, but at the time the closures are evaluated, `myName` is bound to the last of the four names.

Each of the four closures has its own local environment, but they *share* a parent environment, which means they share the exact same binding for `myName`. We can fix it using the "shadow" syntax for `do`:

```
fn('reader') for fn in for myName in names
  do (myName) ->
    (yourName) -> "Hello #{yourName}, my name is #{myName}"
  #=> [ 'Hello reader, my name is algernon',
  #     'Hello reader, my name is sabine',
  #     'Hello reader, my name is rupert',
  #     'Hello reader, my name is theodora' ]
```

Now, each time we create a function we're first creating its own environment and binding `myName` there, shadowing the comprehension's binding of `myName`. Thus, the comprehension's changes to `myName` don't change each closure's binding.

takeaway

Comprehensions[34] are extraordinarily useful for working with collections, but their loop variables are normal variables and may require special care to obtain the desired results. Also worth noting: Comprehensions may be the *only* place where `let` or `do` is *necessary* in CoffeeScript. Every other case can probably be handled with appropriate use of normal variables.

[34]http://coffeescript.org/#loops

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[35]

We're going to look at encapsulation using CoffeeScript'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[36], 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[37]

Consider a stack[38] 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 CoffeeScript, 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.

[35]http://userpage.fu-berlin.de/~ram/pub/pub_jf47ht81Ht/doc_kay_oop_en

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

[37]https://en.wikipedia.org/wiki/Information_hiding

[38]https://en.wikipedia.org/wiki/Stack_

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 coffeescript?

We've been introduced to CoffeeScript'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[39]), we can easily implement our three operations as functions. Bundling the functions with the state does not require any special "magic" features. CoffeeScript objects can have elements of any type, including functions:

```
stack = do (obj = undefined) ->
  obj =
    array: []
    index: -1
    push: (value) ->
      obj.array[obj.index += 1] = value
    pop: ->
      do (value = obj.array[obj.index]) ->
        obj.array[obj.index] = undefined
        obj.index -= 1 if obj.index >= 0
        value
    isEmpty: ->
      obj.index < 0

stack.isEmpty()
  #=> true
stack.push('hello')
  #=> 'hello'
stack.push('CoffeeScript')
 #=> 'CoffeeScript'
stack.isEmpty()
  #=> false
stack.pop()
 #=> 'CoffeeScript'
stack.pop()
```

[39]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.

```
 #=> 'hello'
stack.isEmpty()
  #=> true
```

method-ology

In this text, we lurch from talking about functions belong to an object to methods. Other languages may separate methods from functions very strictly, but in CoffeeScript 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 queue are all absolutely methods of the queue.

But these wouldn't be methods. Although they belong to an object, they don't interact with it:

```
{
  min: (x, y) -> if x < y then x else y
  max: (x, y) -> if x > y then x else 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:

```
stack = do (array = [], index = -1) ->
  push: (value) ->
    array[index += 1] = value
  pop: ->
    do (value = array[index]) ->
      array[index] = undefined
      index -= 1 if index >= 0
      value
  isEmpty: ->
    index < 0

stack.isEmpty()
  #=> true
stack.push('hello')
  #=> 'hello'
stack.push('CoffeeScript')
 #=> 'CoffeeScript'
```

```
stack.isEmpty()
  #=> false
stack.pop()
 #=> 'CoffeeScript'
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:"

```
StackMaker = ->
  do (array = [], index = -1) ->
    push: (value) ->
      array[index += 1] = value
    pop: ->
      do (value = array[index]) ->
        array[index] = undefined
        index -= 1 if index >= 0
        value
    isEmpty: ->
      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 CoffeeScript's fundamental functions and objects.

In Instances and Classes, we'll look at CoffeeScript's support for class-oriented programming and some of the idioms that functions bring to the party.

is encapsulation "object-oriented?"

We've built something with hidden internal state and "methods," all without needing special `def` or `private` keywords. Mind you, we haven't included all sorts of complicated mechanisms to support inheritance, mixins, and other opportunities for debating the nature of the One True Object-Oriented Style on the Internet.

Then again, the key lesson experienced programmers repeat (although it often falls on deaf ears) is, Composition instead of Inheritance[a]. So maybe we aren't missing much.

[a]http://www.c2.com/cgi/wiki?CompositionInsteadOfInheritance

Composition and Extension

composition

A deeply fundamental practice is to build components out of smaller components. The choice of how to divide a component into smaller components is called *factoring*, after the operation in number theory [40].

The simplest and easiest way to build components out of smaller components in CoffeeScript 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 CoffeeScript Object:

```
# helper function
shallowCopy = (source) ->
  do (dest = {}, key = undefined, value = undefined) ->
    dest[key] = value for own key, value of source
    dest

# our model maker
ModelMaker = (initialAttributes = {}) ->
  do (attributes = shallowCopy(initialAttributes),
      undoStack = StackMaker(),
      redoStack = StackMaker(),
      obj = undefined) ->
    obj = {
      set: (attrsToSet = {}) ->
        undoStack.push(shallowCopy(attributes))
        redoStack = StackMaker() unless redoStack.isEmpty()
        attributes[key] = value for own key, value of attrsToSet
        obj
      undo: ->
        unless undoStack.isEmpty()
          redoStack.push(shallowCopy(attributes))
          attributes = undoStack.pop()
        obj
      redo: ->
        unless redoStack.isEmpty()
          undoStack.push(shallowCopy(attributes))
```

[40]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*.

```
          attributes = redoStack.pop()
        obj
      get: (key) ->
        attributes(key)
      has: (key) ->
        attributes.hasOwnProperty(key)
      attributes: ->
        shallowCopy(attributes)
    }
    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 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[41]:

```
QueueMaker = ->
  do (array = [], head = 0, tail = -1) ->
    pushTail: (value) ->
      array[tail += 1] = value
    pullHead: ->
      if tail >= head
        do (value = array[head]) ->
          array[head] = undefined
          head += 1
          value
    isEmpty: ->
      tail < head
```

Now we wish to create a deque[42] by adding `pullTail` and `pushHead` operations to our queue.[43] Unfortunately, encapsulation prevents us from adding operations that interact with the hidden data structures.

[41]http://duckduckgo.com/Queue_

[42]https://en.wikipedia.org/wiki/Double-ended_queue

[43]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."

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:

```
QueueMaker = ->
  do (queue = undefined) ->
    queue =
      array: []
      head: 0
      tail: -1
      pushTail: (value) ->
        queue.array[queue.tail += 1] = value
      pullHead: ->
        unless queue.isEmpty()
          do (value = queue.array[queue.head]) ->
            queue.array[queue.head] = undefined
            queue.head += 1
            value
      isEmpty: ->
        queue.tail < queue.head
```

Now we can extend a queue into a deque, with a little help from a helper function `extend`:

```
# helper function
extend = (object, extensions) ->
  object[key] = value for key, value of extensions
  object

# a deque maker
DequeMaker = ->
  do (deque = QueueMaker()) ->
    extend(deque,
      size: ->
        deque.tail - deque.head + 1
      pullTail: ->
        unless deque.isEmpty()
          do (value = deque.array[deque.tail]) ->
            deque.array[deque.tail] = undefined
            deque.tail -= 1
            value
      pushHead: do (INCREMENT = 4) ->
```

```
        (value) ->
          if deque.head is 0
            for i in [deque.tail..deque.head]
              deque.array[i + INCREMENT] = deque.array[i]
            deque.tail += INCREMENT
            deque.head += INCREMENT
          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:

```
QueueMaker = ->
  do (queue = undefined) ->
    queue =
      array: []
      head: 0
      tail: -1
      pushTail: (value) ->
        queue.array[queue.tail += 1] = value
      pullHead: do (value = undefined) ->
                  ->
                    unless queue.isEmpty()
                      value = queue.array[queue.head]
                      queue.array[queue.head] = undefined
                      queue.head += 1
                      value
      isEmpty: ->
        queue.tail < queue.head

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

Let’s make a copy of our queue using a handy `extend` function and a comprehension to make sure we copy the array properly:

```
extend = (object, extensions) ->
  object[key] = value for key, value of extensions
  object

copyOfQueue = extend({}, queue)
copyOfQueue.array = (element for element in queue.array)

queue isnt copyOfQueue
  #=> true
```

And start playing with our copies:

```
copyOfQueue.pullHead()
  #=> 'Hello'

queue.pullHead()
  #=> 'CoffeeScript'
```

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.

> 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:

```
AmnesiacQueueMaker = ->
  array: []
  head: 0
  tail: -1
  pushTail: (myself, value) ->
    myself.array[myself.tail += 1] = value
  pullHead: do (value = undefined) ->
              (myself) ->
                unless myself.isEmpty(myself)
                  value = myself.array[myself.head]
                  myself.array[myself.head] = undefined
                  myself.head += 1
                  value
  isEmpty: (myself) ->
    myself.tail < myself.head

queueWithAmnesia = AmnesiacQueueMaker()
queueWithAmnesia.pushTail(queueWithAmnesia, 'Hello')
queueWithAmnesia.pushTail(queueWithAmnesia, 'CoffeeScript')
```

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. CoffeeScript is one such machine:

```
BanksQueueMaker = ->
  array: []
  head: 0
  tail: -1
  pushTail: (value) ->
    this.array[this.tail += 1] = value
  pullHead: do (value = undefined) ->
              ->
                unless this.isEmpty()
                  value = this.array[this.head]
                  this.array[this.head] = undefined
                  this.head += 1
                  value
  isEmpty: ->
    this.tail < this.head

banksQueue = BanksQueueMaker()
banksQueue.pushTail('Hello')
banksQueue.pushTail('CoffeeScript')
```

Every time you invoke a function that is a member of an object, CoffeeScript binds that object to the name `this` in the environment of the function just as if it was an argument.[44] Now we can easily make copies:

[44]CoffeeScript also does other things with `this` as well, but this is all we care about right now.

```
copyOfQueue = extend({}, banksQueue)
copyOfQueue.array = (element for element in banksQueue.array)

copyOfQueue.pullHead()
  #=> 'Hello'

banksQueue.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.

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.

fat arrows are the cure for obese idioms

Wait a second! Let's flip back a few pages and look at the code for a Queue:

```
QueueMaker = ->
  do (queue = undefined) ->
    queue =
      array: []
      head: 0
      tail: -1
      pushTail: (value) ->
        queue.array[queue.tail += 1] = value
      pullHead: ->
        unless queue.isEmpty()
          do (value = queue.array[queue.head]) ->
            queue.array[queue.head] = undefined
            queue.head += 1
            value
      isEmpty: ->
        queue.tail < queue.head
```

Spot the difference? Here's the `pullHead` function we're using now:

```
pullHead: do (value = undefined) ->
            ->
              unless this.isEmpty()
                value = this.array[this.head]
                this.array[this.head] = undefined
                this.head += 1
                value
```

Sneaky: The version of the `pullHead` function moves the `do` outside the function. Why? Let's rewrite it to look like the old version:

```
pullHead: ->
  unless this.isEmpty()
    do (value = this.array[this.head]) ->
      this.array[this.head] = undefined
      this.head += 1
      value
```

Notice that we have a function. We invoke it, and `this` is set to our object. Then, thanks to the `do`, we invoke another function inside that. The function invoked by the `do` keyword does not belong to our object, so `this` is not set to our object. Oops!

> Interestingly, this showcases one of CoffeeScript's greatest strengths and weaknesses. Since everything's a function, we have a set of tools that interoperate on everything the exact same way. However, there are some ways that functions don't appear to do exactly what we think they'll do.
>
> For example, if you put a `return 'foo'` inside a `do`, you don't return from the function enclosing the do, you return from the `do` itself. And as we see, `this` gets set "incorrectly." The Ruby programming language tries to solve this problem by having something–blocks–that look a lot like functions, but act more like syntax. The cost of that decision, of course, is that you have two different kinds of things that look similar but behave differently. (Make that five: Ruby has unbound methods, bound methods, procs, lambdas, and blocks.)

There are two solutions. The error-prone workaround is to write:

```
pullHead: ->
  unless this.isEmpty()
    do (value = this.array[this.head], that = this) ->
      that.array[that.head] = undefined
      that.head += 1
      value
```

Besides its lack of pulchritude, there are many opportunities to mistakingly write `this` when you meant to write `that`. Or `that` for `this`. Or something, especially when refactoring some code.

The better way is to force the function to have the `this` you want. CoffeeScript gives you the “fat arrow” or `=>` for this purpose. Here it is:

```
pullHead: ->
  unless this.isEmpty()
    do (value = this.array[this.head]) =>
      this.array[this.head] = undefined
      this.head += 1
      value
```

The fat arrow says, “Treat this function as if we did the `this` and `that` idiom, so that whenever I refer to `this`, I get the outer one.” Which is exactly what we want if we don’t care to rearrange our code.

Summary

Objects, Mutation, and State

- CoffeeScript permits reassignment/rebinding of variables.
- Arrays and Objects are mutable.
- References permit aliasing of reference types.
- The `letrec` pattern permits defining recursive or mutually dependent functions.
- "Normal Case" variables are automagically scoped.
- Comprehensions are convenient, but require care to avoid scoping bugs.
- State can be encapsulated/hidden with closures.
- Encapsulations can be aggregated with composition.
- Encapsulation resists extension.
- The automagic binding `this` facilitates sharing of functions.
- The fat arrow (`=>`) is syntactic sugar for binding `this`.

Finish the Cup: Instances and Classes

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

As discussed in References, Identity, Arrays, and Objects and again in Encapsulating State, CoffeeScript 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 CoffeeScript Objects, functions, and closures[45].

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

[45]Since the CoffeeScript 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.

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 CoffeeScript Object. The colloquial term for this kind of function is a "Factory Function."

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

```
Ur = ->
```

This doesn't look like a factory function: It doesn't have an expression that yields a Plain Old CoffeeScript 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() is 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 `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.

> We are going to look at CoffeeScript's `class` keyword later, but it's worth noting that what CoffeeScript calls a constructor function does almost everything that people think of when they use the word "class." It constructs instances, it defines their common behaviour, and it can be tested.

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:

```
(->).prototype is (->).prototype
  #=> false
```

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

```
Ur.prototype.language = 'CoffeeScript'

continent = new Ur()
  #=> {}
continent.language
  #=> 'CoffeeScript'
```

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:

```
continent.language = 'JavaScript'
continent
  #=> {language: 'JavaScript'}
continent.language
  #=> 'JavaScript'
Ur.prototype.language
  'CoffeeScript'
```

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.

```
another = new Ur()
  #=> {}
another.language
  #=> 'CoffeeScript'
```

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

```
Ur.prototype.language = 'Sumerian'
another.language
  #=> 'Sumerian'
```

Even more interesting: Changing the constructor's prototype changes the behaviour of all of its instances. This strongly implies that there is a dynamic relationship between instances and their constructors, rather than some kind of mechanism that makes objects by copying.[46]

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

```
continent.constructor
  #=> [Function]

continent.constructor is Ur
  #=> 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:

```
{}.constructor
  #=> [Function: Object]
```

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

```
Ur.prototype.constructor
  #=> [Function]
Ur.prototype.constructor is Ur
  #=> 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 `=>`:

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

```
extend = (object, extensions) ->
  object[key] = value for key, value of extensions
  object

Queue = ->
  extend(this, {
    array: []
    head: 0
    tail: -1
  })

extend(Queue.prototype,
  pushTail: (value) ->
    this.array[this.tail += 1] = value
  pullHead: ->
    unless this.isEmpty()
      do (value = this.array[this.head]) =>
        this.array[this.head] = undefined
        this.head += 1
        value
  isEmpty: ->
    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 CoffeeScript: We wouldn't be able to define functions in the prototype that worked on the instance if CoffeeScript 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 CoffeeScript behaves like an object, everything in CoffeeScript behaves like an instance of a function, and therefore everything in CoffeeScript 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
#=> 5
```

Functions themselves are instances, and they have methods. For example, we know that CoffeeScript treats the fat arrow as if you were using the `this` and `that` idiom. But if you didn't have a fat arrow and you didn't want to take a chance on getting the idiom wrong, you could take advantage of the fact that 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.

So, if we have:

```
  pullHead: ->
    unless this.isEmpty()
      do (value = this.array[this.head]) =>
        this.array[this.head] = undefined
        this.head += 1
        value
```

We could also write it like this:

```
  pullHead: ->
    unless this.isEmpty()
      ((value) ->
        this.array[this.head] = undefined
        this.head += 1
        value
      ).call(this, this.array[this.head])
```

It seems like are objects everywhere in CoffeeScript!

impostors

You may have noticed that we use “weasel words” to describe how everything in CoffeeScript *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, CoffeeScript 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 CoffeeScript 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?

CoffeeScript pulls some legerdemain. When you do something that treats a value like an object, CoffeeScript checks to see whether the value actually is an object. If the value is actually a primitive,[47] CoffeeScript 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 CoffeeScript 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") is "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:

[47]Recall that Strings, Numbers, Booleans and so forth are value types and primitives. We’re calling them primitives here.

```
if userName is "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 withthe 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:

```
if '' then 'truthy' else 'falsy'
  #=> 'falsy'
if 0 then 'truthy' else 'falsy'
  #=> 'falsy'
if false then 'truthy' else 'falsy'
  #=> 'falsy'
```

Compare this to their corresponding instances:

```
if new String('') then 'truthy' else 'falsy'
  #=> 'truthy'
if new Number(0) then 'truthy' else 'falsy'
  #=> 'truthy'
if new Boolean(false) then 'truthy' else '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 "CoffeeScript 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")) is "Spence Olham"
  #=> true
```

Getting clever, we can write this:

```
original = (unknown) ->
    unknown.constructor(unknown)

original(true) is true
  #=> true
original(new Boolean(true) is 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.

A Touch of Class

CoffeeScript 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.

This is simple enough, but there are some advantages to making it even simpler, so CoffeeScript does. Here’s our queue again:

```
class Queue
  constructor: ->
    extend(this,
      array: []
      head: 0
      tail: -1
    )
  pushTail: (value) ->
    this.array[this.tail += 1] = value
  pullHead: ->
    unless this.isEmpty()
      do (value = this.array[this.head]) =>
        this.array[this.head] = undefined
        this.head += 1
        value
  isEmpty: ->
    this.tail < this.head

q = new Queue()
q.pushTail('hello')
q.pushTail('CoffeeScript')
```

Behind the scenes, CoffeeScript acts as if you’d written things out by hand, with several small but relevant details.

the constructor method

As you’ve probably noticed, CoffeeScript turns what may look like a `constructor` method into the body of the `Queue` function. You recall that every object in CoffeeScript has a `constructor` element initialized to the function that created it. So it’s natural that in the class statement, you use `constructor` to define the body of the function.

scope

CoffeeScript wraps the entire class statement in a `do ->` so that you can work with some normal variables if you need them.

Here's a gratuitous example:

```
class Queue
  empty = 'UNUSED'
  constructor: ->
    extend(this,
      array: []
      head: 0
      tail: -1
    )
  pushTail: (value) ->
    this.array[this.tail += 1] = value
  pullHead: ->
    unless this.isEmpty()
      do (value = this.array[this.head]) =>
        this.array[this.head] = empty
        this.head += 1
        value
  isEmpty: ->
    this.tail < this.head
```

The value `'UNUSED'` is bound to the name `empty` within the class "statement" but not outside it (unless you are aliasing an `empty` variable). CoffeeScript allows this kind of thing but will get hissy if you try to get fancy and write something like:

```
class Queue
  do (empty = 'UNUSED') ->
    constructor: ->
      extend(this,
        array: []
        head: 0
        tail: -1
      )
    # ...
```

That won't work, you can't wrap a `do` around the instance methods of the class.

at-at walkers

CoffeeScript, in what may be an homage to Ruby, provides an abbreviation for `this.`, you can preface any label with an `@` as a shortcut. This small detail could easily be ignored, except for the fact that there's one place where it's mandatory. With that teaser in place, let's discuss a use case.

Let's modify our Queue to count how many queues have been created:

```
class Queue
  constructor: ->
    Queue.queues += 1
    extend(this,
      array: []
      head: 0
      tail: -1
    )
  # ...

Queue.queues = 0
```

To make this work properly, CoffeeScript has to wrap our code in a `do` so that the code in the constructor *always* refers to the correct function, even if we subsequently change the binding for `Queue` in the outer environment. CoffeeScript does this.

Assigning values to elements of the function outside of the `class` statement is awkward, so CoffeeScript lets us put `Queue.queues = 0` inside, anywhere we'd like. The top is fine. But interestingly, CoffeeScript also sets the context of the body of the `class` statement to be the class itself. So we can write:

```
class Queue
  this.queues = 0
  constructor: ->
    Queue.queues += 1
    extend(this,
      array: []
      head: 0
      tail: -1
    )
  # ...
```

And back to our shortcut. We can also write:

```
class Queue
  @queues = 0
  constructor: ->
    Queue.queues += 1
    extend(this,
      array: []
      head: 0
      tail: -1
    )
  pushTail: (value) ->
    @array[@tail += 1] = value
  pullHead: ->
    unless @isEmpty()
      do (value = @array[@head]) =>
        @array[@head] = undefined
        @head += 1
        value
  isEmpty: ->
    @tail < @head
```

Everything up to now has been a matter of taste. But should you wish, you can write:

```
class Queue
  @queues: 0
  # ...
```

Putting the `@` prefix (and not `this.`) on a label as part of the structure inside the class statement indicates that the element belongs to the constructor (or "class") and not the prototype. Obviously, if you put functions in the constructor, you get constructor methods and not instance methods. For example:

```
class Queue
  @queues: 0
  @resetQueues: ->
    @queues = 0
  # ...
```

We've added a constructor method to reset the count.

It seems there is Strong Typing in Coffeeland

Classes

CoffeeScript's `class` statement is a nice syntactic convenience over manually wiring everything up, and it may help avoid errors. Since most CoffeeScript programmers will use "classes," it's wise to use the class statement when the underlying semantics are what you want. That way your code will communicate its intent clearly and be a little more resistant to small errors.

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. 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 CoffeeScript Objects, because they're the only kind of method you can use. Recall from This and That:

```
QueueMaker = ->
  array: []
  head: 0
  tail: -1
  pushTail: (value) ->
    this.array[this.tail += 1] = value
  pullHead: ->
    unless this.isEmpty()
      do (value = this.array[this.head]) =>
        this.array[this.head] = undefined
        this.head += 1
        value
  isEmpty: ->
    this.tail < this.head
```

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

```
stack = do (obj = undefined) ->
  obj =
    array: []
    index: -1
    push: (value) ->
      obj.array[obj.index += 1] = value
    pop: ->
      do (value = obj.array[obj.index]) ->
        obj.array[obj.index] = undefined
        obj.index -= 1 if obj.index >= 0
        value
    isEmpty: ->
```

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`. We're using the class statement, but it could just as easily be rolled by hand:

```
class WidgetModel
  constructor: (id, attrs = {}) ->
    this[key] = value for key, value of own attrs
    @id = ->
      id
    this
  set: (attrs) ->
    # ...
  get: (key) ->
    # ...
  has: (key) ->
    # ...
```

`set`, `get`, and `has` 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.

Canonicalization

Early in this book, we discussed how objects, arrays, and functions are *reference types*. When we create a new object, even if it has the same contents as some other object, it is a different value, as we can tell when we test its identity with `is`:

```
{ foo: 'bar' } is { foo: 'bar' }
  #=> false
```

Sometimes, this is not what you want. A non-trivial example is the HashLife[48] algorithm for computing the future of Conway's Game of Life. HashLife aggressively caches both patterns on the board and their futures, so that instead of iteratively simulating the cellular automaton a generation at a time, it executes in logarithmic time.

In order to take advantage of cached results, HashLife must *canonicalize* square patterns. Meaning, it must guarantee that if two square patterns have the same contents, they must be the same object and share the same identity. This ensures that updates are shared everywhere.

One way to make this work is to eschew having all the code create new objects with a constructor. Instead, the construction of new objects is delegated to a cache. When a function needs a new object, it asks the cache for it. If a matching object already exists, it is returned. If not, a new one is created and placed in the cache.

This is the algorithm used by recursiveuniver.se[49], an experimental implementation of HashLife in CoffeeScript. The fully annotated source code for canonicalization is online[50], and it contains this method for the `Square.cache` object:

```
for: (quadrants, creator) ->
  found = Square.cache.find(quadrants)
  if found
    found
  else
    {nw, ne, se, sw} = quadrants
    Square.cache.add _for(quadrants, creator)
```

Instead of enjoying a stimulating digression explaining how that works, let's make our own. We're going to build a class for cards in a traditional deck. Without canonicalization, it looks like this:

[48]https://en.wikipedia.org/wiki/Hashlife
[49]http://recursiveuniver.se
[50]http://recursiveuniver.se/docs/canonicalization.html

```
class Card
  ranks = [2..10].concat ['J', 'Q', 'K', 'A']
  suits = ['C', 'D', 'H', 'S']
  constructor: ({@rank, @suit}) ->
    throw "#{@rank} is a bad rank" unless @rank in ranks
    throw "#{@suit} is a bad suit" unless @suit in suits
  toString: ->
    '' + @rank + @suit
```

The instances are not canonicalized:

```
new Card({rank: 4, suit: 'S'}) is new Card({rank: 4, suit: 'S'})
  #=> false
```

If a constructor function explicitly returns a value, that's what is returned. Otherwise, the newly constructed object is returned. Unlike other functions and methods, the last evaluated value is not returned by default.

We can take advantage of that to canonicalize cards:

```
class Card
  ranks = [2..10].concat ['J', 'Q', 'K', 'A']
  suits = ['C', 'D', 'H', 'S']
  cache = {}
  constructor: ({@rank, @suit}) ->
    throw "#{@rank} is a bad rank" unless @rank in ranks
    throw "#{@suit} is a bad suit" unless @suit in suits
    return cache[@toString()] or= this
  toString: ->
    '' + @rank + @suit
```

Now the instances are canonicalized:

```
new Card({rank: 4, suit: 'S'}) is new Card({rank: 4, suit: 'S'})
  #=> true
```

Wonderful!

caveats

Using techniques like this to canonicalize instances of a class has many drawbacks and takes careful consideration before use. First, while this code illustrates the possibilities inherent in having a constructor return a different object, it is wasteful in that it creates an object only to throw it away if it is already in the cache.

If there are a tractable number of possible instances of a class (such as cards in a deck), it may be more practical to enumerate them all in advance rather than lazily create them, and/or to use a factory method to retried them rather than changing the behaviour of the constructor.

More serious is that the engine that executes CoffeeScript programs does not support weak references.[51] As a result, if you wish to perform cache eviction for memory management purposes, you will have to implement your own reference management scheme. This may be non-trivial.

If you have many, many possible instances, your cache can end up holding onto what some programmers call *zombie objects*: Objects that are not in use anywhere in your program except the cache. If they are never accessed again, the memory they take up will never be released for reuse. An early version of the HashLife implementation did not clear objects from the cache. Some computations would consume as much as 700MB of data for the cache before the virtual machine was unable to continue. Most of that memory was consumed by zombie objects.

All that being said, canonicalization is sometimes the appropriate path forward, and even if it isn't, it serves to illustrate the possibilities latent in writing constructors that return objects explicitly.

[51]A weak reference is a reference that does not protect the referenced object from collection by a garbage collector; unlike a strong reference. An object referenced only by weak references is considered unreachable (or weakly reachable) and so may be collected at any time.

This Section Needs No Title

CoffeeScript is fundamentally an object-oriented language in the sense that Alan Kay first described object orientation. His vision was of software constructed from entities that communicate with message passing, with the system being extremely dynamic (what he described as “extreme late-binding”). However, words and phrases are only useful when both writer and reader share a common understanding, and for many people the words “object-oriented” carry with them a great deal of baggage related to constructing ontologies of domain entities.

The word “Inheritance” also means many different things to many different people. Some people take it extremely seriously, tugging thoughtfully on their long white beards as they ponder things like Strict Liskov Equivalence. We will avoid this term as well.

What we *will* discuss is extension. In the next section, we’re going to show how functions that create instances can extend each other through their prototypes. Since we just finished looking at the class statement, we’ll start by chaining two classes together, and then generalize extension so that you can use it with any two functions that create instances.

We’ll finish by looking at the excellent support CoffeeScript provides so that you can accomplish all of this with a single keyword.

Extending Classes

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

Here's our `Queue`:

```
class Queue
  constructor: ->
    @array = []
    @head  = 0
    @tail  = -1
  pushTail: (value) ->
    @array[@tail += 1] = value
  pullHead: ->
    unless @isEmpty()
      do (value = @array[@head]) =>
        @array[@head] = undefined
        @head += 1
        value
  isEmpty: ->
    @tail < @head
```

And our `Deque` before we wire things together:

```
class Deque
  size: ->
    @tail - @head + 1
  pullTail: ->
    unless @isEmpty()
      do (value = @array[@tail]) =>
        @array[@tail] = undefined
        @tail -= 1
        value
  INCREMENT = 4
  pushHead: (value) ->
    if @head is 0
      for i in [@tail..@head]
        @array[i + INCREMENT] = @array[i]
      @tail += INCREMENT
```

```
        @head += INCREMENT
      @array[@head -= 1] = value
```

So what do we want from dequeues?

1. A `Deque` function that initializes a deque when invoked with `new`
2. `Deque.prototype` must have all the behaviour of a deque and all the behaviour of a queue.

Hmmm. So, should we copy everything from `Queue.prototype` into `Deque.prototype`? No, there's a better idea. Prototypes are objects, right? Why must they be Plain Old CoffeeScript 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:

```
QueueProxy = ->

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:

```
class Deque
  QueueProxy = ->
  QueueProxy.prototype = Queue.prototype
  Deque.prototype = new QueueProxy()
  size: ->
    @tail - @head + 1
  # ...
```

Before we rush off to try this, we're missing something. How are we going to initialize our deques? We'd better call `Queue`'s constructor:

```
constructor: ->
  Queue.prototype.constructor.call(this)
```

Here's what we have so far:

```
class Deque
  QueueProxy = ->
  QueueProxy.prototype = Queue.prototype
  @prototype = new QueueProxy()
  constructor: ->
    Queue.prototype.constructor.call(this)
  # ...
```

And it seems to work:

```
d = new Deque()
d.pushTail('Hello')
d.pushTail('CoffeeScript')
d.pushTail('!')
d.pullHead()
  #=> 'Hello'
d.pullTail()
  #=> '!'
d.pullHead()
  #=> 'CoffeeScript'
```

Wonderful!

getting the constructor element right

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

```
d.constructor is Deque
  #=> 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`:

```
class Deque
  QueueProxy = ->
    @constructor = Deque
    this
  QueueProxy.prototype = Queue.prototype
  @prototype = new QueueProxy();
  # ...
```

Now it works:

```
d.constructor is Deque
  #=> true
```

The `QueueProxy` function now sets the `constructor` for every instance of a `QueueProxy` (hopefully just the one we need for the `Deque` 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 `Deque`'s prototype, which would break our code).

extracting the boilerplate

Let's turn our extension modifications into a function:

```
xtend = (child, parent) ->
  do (proxy = undefined) ->
    proxy = ->
      @constructor = child
      this
    proxy.prototype = parent.prototype
    child.prototype = new proxy()
```

And use it in `Deque`:

```
class Deque
  xtend(Deque, Queue)
  constructor: ->
    Queue.prototype.constructor.call(this)
  size: ->
    @tail - @head + 1
  pullTail: ->
    unless @isEmpty()
      do (value = @array[@tail]) =>
        @array[@tail] = undefined
        @tail -= 1
        value
  INCREMENT = 4
  pushHead: (value) ->
    if @head is 0
      for i in [@tail..@head]
        @array[i + INCREMENT] = @array[i]
      @tail += INCREMENT
      @head += INCREMENT
    @array[@head -= 1] = value
```

And you can use `xtend` even if you don't want to use the class statement:

```
A = ->
B = ->
xtend(B, A)
```

It's such a nice idea. Wouldn't it be great if CoffeeScript had it built-in? Behold:

```
B extends A
```

Most helpful! In fact, CoffeeScript's keyword is superior to the `xtend` function: It provides support for extending functions that have other properties, not just the prototype.[52]

How about the class statement? CoffeeScript does a lot more work for you if you wish. You can write:

[52]You should almost always use `extends` rather than rolling your own code to chain functions and instances. And even if you let CoffeeScript do the work, you should always *understand* what CoffeeScript is doing for you.

```
class Deque
  Deque extends Queue
  constructor: ->
    Queue.prototype.constructor.call(this)
  # ...
```

But there's more. If you instead write:

```
class Deque extends Queue
  # ...
```

1. CoffeeScript will do even more work for you. If you aren't doing any extra setup, you can leave the constructor out. CoffeeScript will handle calling the extended function's constructor for you.
2. If you do wish to do some extra setup, write your own constructor. Like Ruby, you can call `super` in any method to access the extended version of the same method.
3. CoffeeScript's `class` and `extends` keywords handle a lot of the boilerplate and play nicely together. Use them unless you have specific needs they don't cover.

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.
- CoffeeScript can convert primitives into instances and back into primitives.
- The `class` keyword and `constructor` method are syntactic sugar for creating functions and populating prototypes.
- `@` is a convenient shorthand for `this.`.
- Object methods are typically created in the constructor and are private to each object.
- Canoncialization is tricky but possible in CoffeeScript.
- Prototypes can be chained to allow extension of instances.
- CoffeeScript's `extends` keyword is syntactic sugar for extending prototypes.
- `extends` plays well with `class`.

interlude...

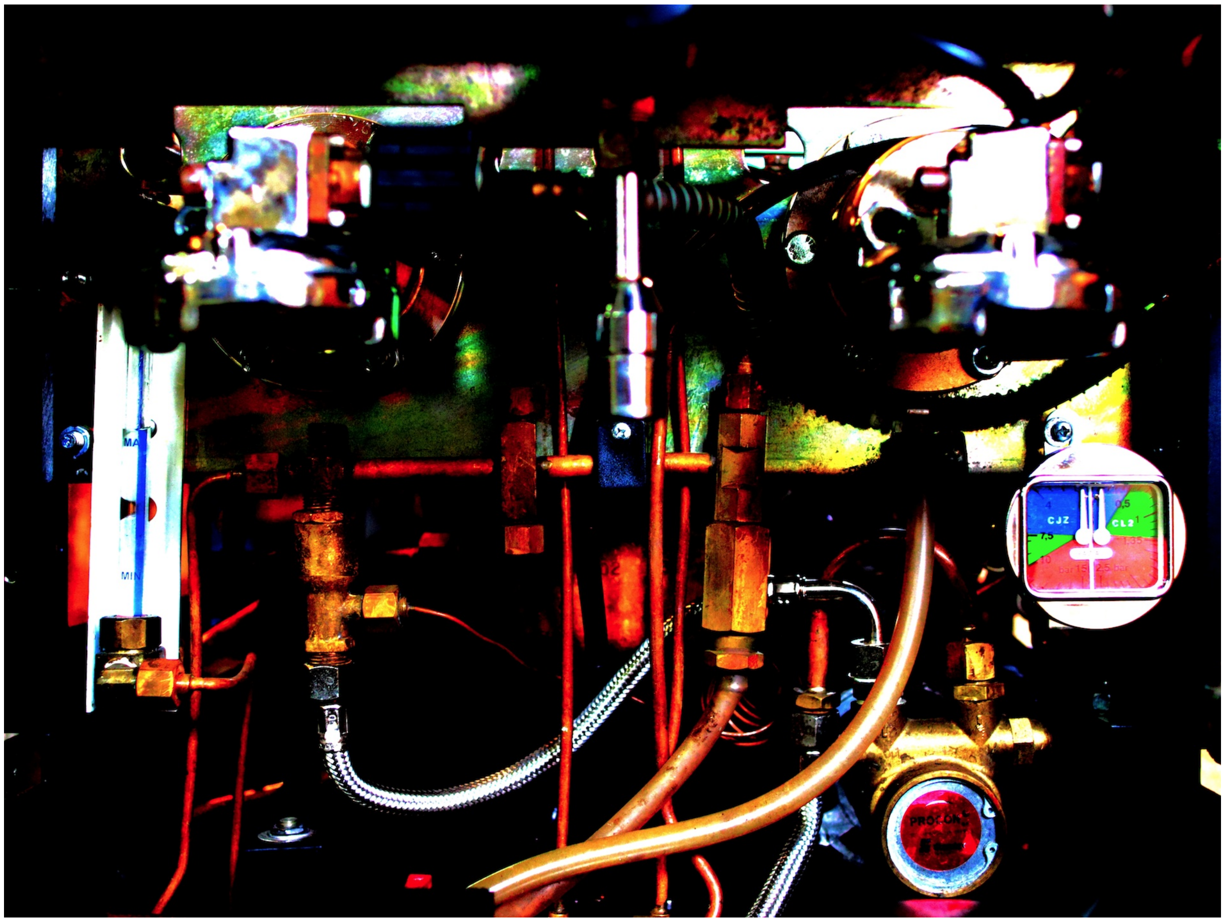
Drawing a Doppio

Aaron De Lazzer[53] on Ristretto:

> "The ristretto shot of espresso is one of the most fiercely debated and favourite topics amongst the coffee cognoscenti. It is the purists pour. The cutting edge of espresso extraction, flying in the face of the "Big Gulp" coffee drinker like nothing else around.
>
> In anything at all, perfection is finally attained not when there is no longer anything to add, but when there is no longer anything to take away.–Antoine de Saint-Exupery
>
> "Antoine would have drank ristretto shots.
>
> "There is no where to hide with a straight unadulterated shot of espresso. Even more (less?) so with a ristretto shot. Any weakness in the blend or in the preparation of the coffee will be brought to light here. Either the heavens open up and the angels sing after that first sip or....something significantly less. Which is always such a disappointment knowing all the potential distilled into the dribble of coffee liquor that barely coats the bottom of your cup."

[53]http://www.coffeegeek.com/opinions/aarondelazzer/02-24-2002

An Extra Shot of Ideas



The Intestines of an Espresso Machine

Refactoring to 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[54]

In this book, we will be using a much looser definition of “combinator:” Pure functions that act on other functions to produce functions. Combinators are the adverbs of functional programming.

memoize

Let’s begin with an example of a combinator, `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 simple implementation:

```
fibonacci = (n) ->
  if n < 2
    n
  else
    fibonacci(n-2) + fibonacci(n-1)
```

We’ll time it:

```
s = (new Date()).getTime()
new Fibonacci(45).toInt()
( (new Date()).getTime() - s ) / 1000
  #=> 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.

We could write something specific for fibonacci and then generalize it, but let’s skip right to a general solution (we’ll discuss extracting a combinator below). First, a new feature. Within any function the name `arguments` is always bound to an object that behaves like a collection of all the arguments passed to a function. Using `arguments`, here is a `memoize` implementation that works for many[55] kinds of functions:

[54] https://en.wikipedia.org/wiki/Combinatory_logic

[55] To be precise, it works for functions that take arguments that can be expressed with JSON. So you can’t memoize a function that is applied to functions, but it’s fine for strings, numbers, arrays of JSON, POCOs of JSON, and so forth.

```
memoized = (fn) ->
  do (lookupTable = {}, key = undefined, value = undefined) ->
    ->
      key = JSON.stringify(arguments)
      lookupTable[key] or= fn.apply(this, arguments)
```

We can apply `memoized` to a function and we will get back a new function that memoizes its results. Let's try it:

```
fastFibonacci =
  memoized (n) ->
    if n < 2
      n
    else
      fastFibonacci(n-2) + fastFibonacci(n-1)

fastFibonacci(45)
  #=> 1134903170
```

We get the result back instantly. It works!

Exercise:

Optimistic Ophelia tries the following code:

```
fibonacci = (n) ->
  if n < 2
    n
  else
    fibonacci(n-2) + fibonacci(n-1)

quickFibonacci = memoize(fibonacci)
```

Does `quickFibonacci` behave differently than `fastFibonacci`? Why?

By using a combinator instead of tangling lookup code with the actual "domain logic" of fibonacci, our `fastFibonacci` code is easy to read and understand. As a bonus, we DRY up our application, as the same `memoize` combinator can be used in many different places.

the once and future combinator refactoring

Combinators can often be extracted from your code. The result is cleaner than the kinds of refactorings possible in languages that have less flexible functions. Let's walk through the process of discovering a combinator to get a feel for the refactoring to combinator process.

Some functions should only be evaluated once, but might be invoked more than once. You want to evaluate them the first time they are called, but thereafter ignore the invocation.

We'd start with a function like this, it assumes there's some "it" that needs to be initialized once, and several pieces of code might call this function before using "it:"

```
ensureItIsInitialized = do (itIsInitialized = false) ->
  ->
    unless itIsInitialized
      itIsInitialized = true
      # ...
      # initialization code
      # ...
```

The typical meta-pattern is when several different functions all share a common precondition such as loading certain constant data from a server or initializing a resource. We can see that we're tangling the concern of initializing once with the concern of how to perform the initialization. Let's extract a method[56]:

```
initializeIt = ->
  # ...
  # initialization code
  # ...
ensureItIsInitialized = do (itIsInitialized = false) ->
  ->
    unless itIsInitialized
      itIsInitialized = true
      initializeIt()
```

In many other languages, we'd stop right there. But in CoffeeScript, we can see that `ensureItIsInitialized` is much more generic than its name suggests. Let's convert it to a combinator with a slight variation on the extracting a parameter[57]. We'll call the combinator `once`:

[56]http://refactoring.com/catalog/extractMethod.html
[57]http://www.industriallogic.com/xp/refactoring/extractParamter.html

```
once = (fn) ->
  do (done = false) ->
    ->
      unless done
        done = true
        fn.apply(this, arguments)
```

And now our code is very clean:

```
initializeIt = ->
  # ...
  # initialization code
  # ...
ensureItIsInitialized = once(initializeIt)
```

This is so clean you could get rid of `initializeIt` as a named function:

```
ensureItIsInitialized = once ->
  # ...
  # initialization code
  # ...
```

The concept of a combinator is more important than having a specific portfolio of combinators at your fingertips (although that is always nice). The meta-pattern is that when you are working with a function, identify the core "domain logic" the function should express. Try to extract that and turn what is left into one or more combinators that take functions as parameters rather than single-purpose methods.

(The `memoize` and `once` combinators are available in the underscore.js[58] library along with several other handy functions that operate on functions such as `throttle` and `debounce`.)

Composition and Combinators

Although you can write nearly any function and use it as a combinator, one property that is nearly essential is *composability*. It should be possible to pass the result of one combinator as the argument to another.

Let's consider this combinator as an example:

[58]http://underscorejs.org

```
requiresValues = (fn) ->
  ->
    throw "Value Required" unless arg? for arg in arguments
    fn.apply(this, arguments)
```

And this one:

```
requiresReturnValue = (fn) ->
  ->
    do (result = fn.apply(this, arguments)) ->
      throw "Value Required" unless result?
      result
```

You can use *both* of these combinators and the `once` combinator on a function to add some runtime validation that both input arguments and the returned value are defined:

```
checkedFibonacci = once requiresValues requiresReturnValue (n) ->
  if n < 2
    n
  else
    fibonacci(n-2) + fibonacci(n-1)
```

Combinators should be designed by default to compose. And speaking of composition, it's easy to compose functions in CoffeeScript:

```
checkBoth = (fn) ->
    requiresValues requiresReturnValue fn
```

Libraries like Functional[59] also provide `compose` and `sequence` functions, so you can write things like:

```
checkBoth = Functional.compose(requiresValues, requiresReturnValue)
```

All of this is made possible by the simple property of *composability*, the property of combinators taking a function as an argument and returning a function that operates on the same arguments and returns something meaningful to code expecting to call the original function.

[59]http://osteele.com/sources/javascript/functional/

Method Decorators

Now that we've seen how function combinators can make our code cleaner and DRYer, it isn't a great leap to ask if we can use combinators with methods. After all, methods are functions. That's one of the great strengths of CoffeeScript, since methods are "just" functions, we don't need to have one kind of tool for functions and another for methods, or a messy way of turning methods into functions and functions into methods.

And the answer is, "Yes we can." With some caveats. Let's get our terminology synchronized. A combinator is a function that modifies another function. A *method decorator* is a combinator that modifies a method expression used inline. So, all method decorators are combinators, but not all combinators are method decorators.[60]

decorating object methods

As you recall, an object method is a method belonging directly to a Plain Old CoffeeScript object or an instance. All combinators work as decorators for object methods. For example:

```
class LazyInitializedMechanism
  constructor: ->
    @initialize = once ->
      # ...
      # complicated stuff
      # ...
  someInstanceMethod: ->
    @initialize()
    # ...
  anotherInstanceMethod: (foo) ->
    @initialize()
    # ...
```

decorating constructor methods

Decorating constructor methods works just as well as decorating instance methods, for example:

[60]The term "method decorator" is borrowed from the Python programming language

```
class LazyClazz
  @setUpLazyClazz: once ->
    # ...
    # complicated stuff
    # ...
  constructor: ->
    this.constructor.setUpLazyClazz()
    # ...
```

For this class, there's some setup to be done, but it's deferred until the first instance is created.

decorating instance methods

Decorating instance methods can be tricky if they rely on closures to encapsulate state of any kind. For example, this will not work:

```
class BrokenMechanism
  initialize: once ->
    # ...
    # complicated stuff
    # ...
  someInstanceMethod: ->
    @initialize()
    # ...
  anotherInstanceMethod: (foo) ->
    @initialize()
    # ...
```

If you have more than one `BrokenMechanism`, only one will ever be initialized. There is one `initialize` method, and it belongs to `BrokenMechanism.prototype`, so once it is called for the first `BrokenMechanism` instance, all others calling it for the same or different instances will not execute.

The `initialize` method could be converted from an instance method to an object method as above. An alternate approach is to surrender the perfect encapsulation of `once`, and write a decorator designed for use on instance methods:

```
once = (name, method) ->
  ->
    unless @[name]
      @[name] = true
      method.apply(this, arguments)
```

Now the flag for being done has been changed to an element of the instance, and we use it like this:

```
class WorkingMechanism
  initialize: once 'doneInitializing', ->
    # ...
    # complicated stuff
    # ...
  someInstanceMethod: ->
    @initialize()
    # ...
  anotherInstanceMethod: (foo) ->
    @initialize()
    # ...
```

Since the flag is stored in the instance, the one function works with all instances. (You do need to make sure that each method using the decorator has its own unique name.)

a decorator for fluent interfaces

Fluent interfaces[61] are a style of API often used for configuration. The principle is to return an instance that has meaningful methods for the next thing you want to do. The simplest (but not only) type of fluent interface is a cascade of methods configuring the same object, such as:

```
car = new Automobile()
  .withBucketSeats(2)
  .withStandardTransmission(5)
  .withDoors(4)
```

To implement an interface with this simple API, methods need to return `this`. It's one line and easy to do, but you look at the top of the method to see its name and the bottom of the method to see what it returns. If there are multiple return paths, you must take care that they all return `this`.

It's easy to write a fluent decorator:

```
fluent = (method) ->
  ->
    method.apply(this, arguments)
    this
```

And it's easy to use:

[61] https://en.wikipedia.org/wiki/Fluent_interface

```
class Automobile
    withBucketSeats: fluent (num) ->
      # ...
    withStandardTransmission: fluent (gears) ->
      # ...
    withDoors: fluent (num) ->
      # ...
```

combinators for making decorators

Quite a few of the examples involve initializing something before doing some work. This is a very common pattern: Do something *before* invoking a method. Can we extract that into a combinator? Certainly. Here's a combinator that takes a method and returns a decorator:

```
before =
  (decoration) ->
    (base) ->
      ->
        decoration.apply(this, arguments)
        base.apply(this, arguments)
```

You would use it like this:

```
forcesInitialize = before -> @initialize()

class WorkingMechanism
  initialize: once 'doneInitializing', ->
    # ...
    # complicated stuff
    # ...
  someInstanceMethod: forcesInitialize ->
    # ...
  anotherInstanceMethod: forcesInitialize (foo) ->
    # ...
```

Of course, you could put anything in there, including the initialization code if you wanted to:

```
class WorkingMechanism
  forcesInitialize = before ->
    # ...
    # complicated stuff
    # ...
  someInstanceMethod: forcesInitialize ->
    # ...
  anotherInstanceMethod: forcesInitialize (foo) ->
    # ...
```

When writing decorators, the same few patterns tend to crop up regularly:

1. You want to do something *before* the method's base logic is executed.
2. You want to do something *after* the method's base logic is executed.
3. You want to wrap some logic *around* the method's base logic.
4. You only want to execute the method's base logic *provided* some condition is truthy.

We saw `before` above. Here are three more combinators that are very useful for writing method decorators:

```
after =
  (decoration) ->
    (base) ->
      ->
        decoration.call(this, __value__ = base.apply(this, arguments))
        __value__

around =
  (decoration) ->
    (base) ->
      (argv...) ->
        __value__ = undefined
        callback = =>
          __value__ = base.apply(this, argv)
        decoration.apply(this, [callback].concat(argv))
        __value__

provided =
  (condition) ->
    (base) ->
      ->
        if condition.apply(this, arguments)
          base.apply(this, arguments)
```

All four of these, and many more can be found in the method combinators[62] module. They can be used with all CoffeeScript and JavaScript projects.

[62]https://github.com/raganwald/method-combinators

Callbacks and Promises

Like nearly all languages in widespread use, CoffeeScript expresses programs as expressions that are composed together with a combination of operators, function application, and control flow constructs such as sequences of statements.

That's all baked into the underlying language, so it's easy to use it without thinking about it. Much as a fish (perhaps) exists in the ocean without being aware that there is an ocean. In this chapter, we're going to examine how to compose functions together when we have non-traditional forms of control-flow such as asynchronous function invocation.

composition

The very simplest example of composing functions is simply "pipelining" the values. CoffeeScript's optional parentheses make this quite readable. Given:

```
getIdFromSession = (session) ->
  # ...

fetchCustomerById = (id) ->
  # ...

currentSession = # ...
```

You can write either:

```
customerList.add(
  fetchCustomerById(
    getIdFromSession(
      currentSession
     )
  )
)
```

Or:

```
customerList.add fetchCustomerById getIdFromSession currentSession
```

The "flow" of data is from right-to-left. Some people find it more readable to go from left-to-right. The `sequence` function accomplishes this:

```
sequence = ->
  do (fns = arguments) ->
    (value) ->
      (value = fn(value)) for fn in fns
      value

sequence(
  getIdFromSession,
  fetchCustomerById,
  customerList.add
)(currentSession)
```

asynchronous code

CoffeeScript executes within an environment where code can be invoked asynchronously. For example, a browser application can asynchronously invoke a request to a remote server and invoke handler code when the request is satisfied or deemed to have failed.

A very simple example is that in a browser application, you can defer invocation of a function after all current processing has been completed:

```
defer (fn) ->
  window.setTimeout(fn, 0)
```

The result is that if you write:

```
defer -> console.log('Hello')
console.log('Asynchronicity')
```

The console will show:

```
Asynchronicity
Hello
```

The computer has no idea whether the result should be "Asynchronicity Hello" or "Hello Asynchronicity" or sometimes one and sometimes the other. But if we intend that the result be "Asynchronicity Hello," we say that the function `-> console.log('Hello')` *depends upon* the code `console.log('Asynchronicity')`.

This might be what you want. If it isn't, you need to have a way to force the order of evaluation when there is supposed to be a dependency between different evaluations. There are a number of different models and abstractions for controlling these dependencies.

We will examine two of them (callbacks and promises) briefly. Not for the purpose of learning the subtleties of using either model, but rather to obtain an understanding of how functions can be used to implement an abstraction over an underlying model.

callbacks

The underlying premise of the callback model is that every function that invoked code asynchronously is responsible for invoking code that depends on it. The simplest protocol for this is also the most popular: Functions that invoke asynchronous code take an extra parameter called a *callback*. That parameter is a function to be invoked when they have completed.

So our `defer` function looks like this if we want to use callbacks:

```
defer (fn, callback) ->
  window.setTimeout (-> callback fn), 0
```

Instead of handing `fn` directly to `window.setTimeout`, we're handing it a function that invokes `fn` and pipelines the result (if any) to `callback`. Now we can ensure that the output is in the correct order:

```
defer (-> console.log 'hello'), (-> console.log 'Asynchronicity')

#=> Hello
#   Asynchronicity
```

Likewise, let's say we have a `displayPhoto` function that is synchronous, and not callback-aware:

```
displayPhoto = (photoData) ->
  # ... synchronous function ...
```

It can also be converted to take a callback:

```
displayPhotoWithCallback = (photoData, callback) ->
  callback(displayPhoto(photoData))
```

There's a combinator we can extract:

```
callbackize = (fn) ->
  (arg, callback) ->
    callback(fn(arg))
```

You recall that with ordinary functions, you could chain them with function application. With callbacks, you can also chain them manually. Here's an example inspired by a blog post[63], fetching photos from a remote photo sharing site using their asynchronous API:

[63]http://elm-lang.org/learn/Escape-from-Callback-Hell.elm

```
tag = 'ristretto'

fotositeGetPhotosByTag tag, (photoList) ->
  fotositeGetOneFromList photos, (photoId) ->
    fotositeGetPhoto photoId, displayPhoto
```

We can also create a callback-aware function that represents the composition of functions:

```
displayPhotoForTag = (tag, callback) ->
  fotositeGetPhotosByTag tag, (photoList) ->
    fotositeGetOneFromList photos, (photoId) ->
      fotositeGetPhoto photoId, displayPhoto
```

This code is *considerably* less messy in CoffeeScript than other languages that require a lot of additional syntax for functions. As a bonus, although it has some extra scaffolding and indentation, it's already in sequence order from top to bottom and doesn't require re-ordering like normal function application did. That being said, you can avoid the indentation and extra syntax by writing a `sequenceWithCallbacks` function:

```
I = (x) -> x

sequenceWithCallbacks = ->
  do (fns = arguments,
      lastIndex = arguments.length - 1,
      helper = undefined) ->
    helper = (arg, index, callback = I) ->
      if index > lastIndex
        callback arg
      else
        fns[index] arg, (result) ->
          helper result, index + 1, callback
   (arg, callback) ->
     helper arg, 0, callback

 displayPhotoForTag = sequenceWithCallbacks(
   fotositeGetPhotosByTag,
   fotositeGetOneFromList,
   fotositeGetPhoto,
   displayPhotoWithCallback
 )
```

`sequenceWithCallbacks` is more complex than `sequence`, but it does help us make callback-aware code "linear" instead of nested/indented.

As we have seen, we can compose linear execution of asynchronous functions, using either the explicit invocation of callbacks or using `sequenceWithCallbacks` to express the execution as a list.

solving this problem with promises

Asynchronous control flow can also be expressed using objects and methods. One model is called promises[64]. A *promise* is an object that acts as a state machine.[65] Its permissible states are:

- unfulfilled
- fulfilled
- failed

The only permissible transitions are from *unfulfilled* to *fulfilled* and from *unfulfilled* to *failed*. Once in either the fulfilled or failed states, it remains there permanently.

Each promise must at a bare minimum implement a single method, `.then(fulfilledCallback, failedCallback)`.[66] `fulfilledCallback` is a function to be invoked by a fulfilled promise, `failedCallback` by a failed promise. If the promise is already in either state, that function is invoked immediately.

`.then` returns another promise that is fulfilled when the appropriate callback is fulfilled or fails when the appropriate callback fails. This allows chaining of `.then` calls.

If the promise is unfulfilled, the function(s) provided by the `.then` call are queued up to be invoked if and when the promise transitions to the appropriate state. In addition to this being an object-based protocol, the promise model also differs from the callback model in that `.then` can be invoked on a promise at any time, whereas callbacks must be specified in advance.

Here's how our fotosite API would be used if it implemented promises instead of callbacks (we'll ignore handling failures):

```
fotositeGetPhotosByTag(tag)
  .then fotositeGetOneFromList
  .then fotositeGetPhoto
  .then displayPhoto
```

Crisp and clean, no caffeine. The promises model provides linear code "out of the box," and it "scales up" to serve as a complete platform for managing asynchronous code and remote invocation. Be sure to look at libraries supporting promises like q[67], and when[68].

[64]http://wiki.commonjs.org/wiki/Promises/A

[65]A state machine is a mathematical model of computation used to design both computer programs and sequential logic circuits. It is conceived as an abstract machine that can be in one of a finite number of states. The machine is in only one state at a time; the state it is in at any given time is called the current state. It can change from one state to another when initiated by a triggering event or condition, this is called a transition. A particular FSM is defined by a list of its states, and the triggering condition for each transition.

[66]Interactive promises also support `.get` and `.call` methods for interacting with a potentially remote object.

[67]https://github.com/kriskowal/q

[68]https://github.com/cujojs/when

Summary

An Extra Shot of Ideas

- Combinators are pure functions that act on other functions to produce functions.
- Combinators are the adverbs of functional programming.
- Combinators can often be extracted from your code.
- Combinators make code more composeable.
- Many combinators are also Method Decorators.
- Some combinators decorate decorators.
- Callbacks abstract control flow.
- Callbacks hide asynchronicity.
- Promises represent control flow with an object API.

the last word...

Espresso a lungo, or the long pull, is thinner in texture, more acidic, and contains more caffeine than a ristretto pull

A Golden Crema

You’ve earned a break!

How to run the examples

If you follow the instructions at coffeescript.org[69] to install NodeJS and CoffeeScript,[70] you can run an interactive CoffeeScript REPL[71] on your command line simply by typing `coffee`. This is how the examples in this book were tested, and what many programmers will do. When running CoffeeScript on the command line, ctrl-V switches between single-line and multi-line input mode. If you need to enter more than one line of code, be sure to enter multi-line mode.

Some websites function as online REPLs[72], allowing you to type CoffeeScript programs right within a web page and see the results (as well as a translation from CoffeeScript to JavaScript). The examples in this book have all been tested on coffeescript.org[73]. You simply type a CoffeeScript expression into the blank window and you will see its JavaScript translation live. Clicking "Run" evaluates the expression in the browser.

To actually see the result of your expressions, you'll need to either include a call to `console.log` (and be using a browser that supports console logging) or you could go old-school and use `alert`, e.g. `alert 2+2` will cause the alert box to be displayed with the message `4`.

[69]http://coffeescript.org/#installation

[70]Instructions for installing NodeJS and modules like CoffeeScript 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.

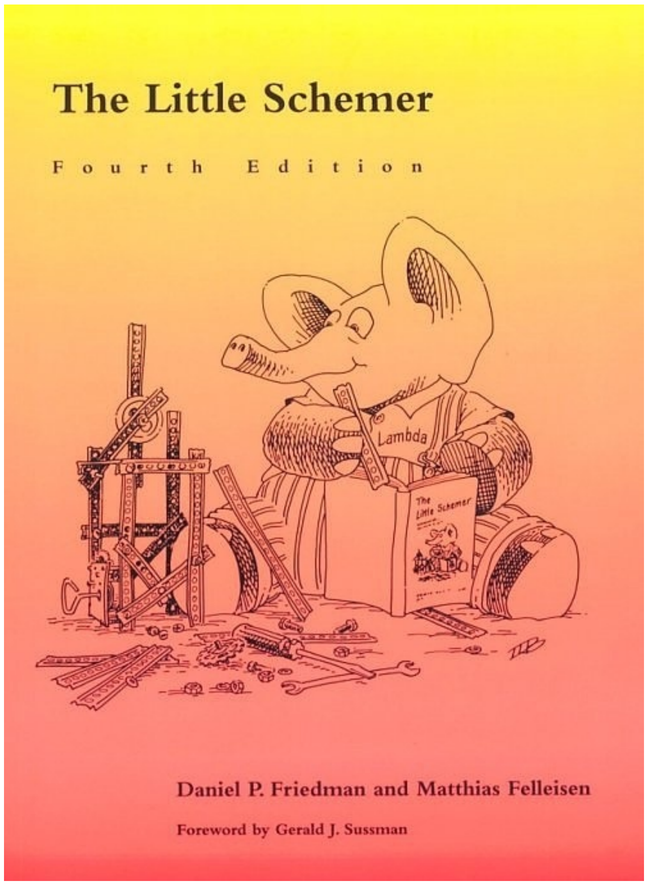
[71]https://en.wikipedia.org/wiki/REPL

[72]https://en.wikipedia.org/wiki/REPL

[73]http://coffeescript.org/#try:

Thanks!

Daniel Friedman and Matthias Felleisen

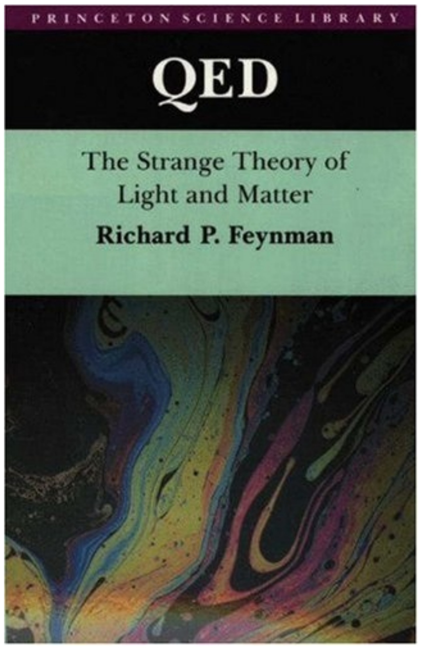

The Little Schemer
Fourth Edition
Lambda
Daniel P. Friedman and Matthias Felleisen
Foreword by Gerald J. Sussman

The Little Schemer

CoffeeScript Ristretto was inspired by The Little Schemer[74] by Daniel Friedman and Matthias Felleisen. But where *The Little Schemer's* primary focus is recursion, *CoffeeScript Ristretto's* primary focus is **functions as first-class values**.

[74]http://www.amzn.com/0262560992?tag=raganwald001-20

Richard Feynman

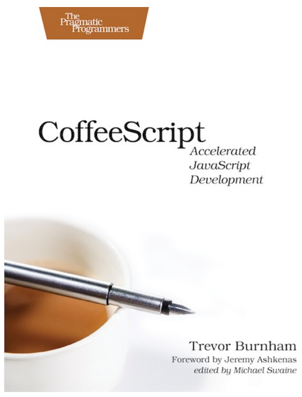

PRINCETON SCIENCE LIBRARY
QED
The Strange Theory of Light and Matter
Richard P. Feynman

QED: The Strange Theory of Light and Matter

Richard Feynman's QED[75] 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.

[75]http://www.amzn.com/0691125759?tag=raganwald001-20

Trevor Burnham


The Pragmatic Programmers
CoffeeScript
Accelerated JavaScript Development
Trevor Burnham
Foreword by Jeremy Ashkenas
edited by Michael Swaine

CoffeeScript: Accelerated JavaScript Development

Trevor Burnham provided invaluable assistance with this book. Trevor is the author of CoffeeScript: Accelerated JavaScript Development[76], an excellent resource for CoffeeScript programmers.

[76]http://pragprog.com/book/tbcoffee/coffeescript

JavaScript Allongé

a long and strong programming book

JavaScript Allongé[77] is the companion book to *CoffeeScript Ristretto*.

[77]http://leanpub.com/javascript-allonge

[78]https://leanpub.com/coffeescript-ristretto
[79]http://creativecommons.org/licenses/by-nc-nd/3.0/
[80]http://www.flickr.com/photos/trumpetca/
[81]http://www.flickr.com/photos/digitalcolony/5054568279/
[82]http://creativecommons.org/licenses/by-sa/2.0/deed.en
[83]http://www.flickr.com/photos/everydaylifemodern/1353570874/
[84]http://creativecommons.org/licenses/by-nd/2.0/deed.en
[85]http://www.flickr.com/photos/everydaylifemodern/434299813/
[86]http://creativecommons.org/licenses/by-nd/2.0/deed.en
[87]http://www.flickr.com/photos/the_rev/2295096211/
[88]http://creativecommons.org/licenses/by/2.0/deed.en
[89]http://www.flickr.com/photos/thedigitelmyr/6199419022/
[90]http://creativecommons.org/licenses/by-sa/2.0/deed.en
[91]http://www.flickr.com/photos/sagamiono/4391542823/
[92]http://creativecommons.org/licenses/by-sa/2.0/deed.en
[93]http://www.flickr.com/photos/digitalcolony/3924227011/
[94]http://creativecommons.org/licenses/by-sa/2.0/deed.en
[95]http://www.flickr.com/photos/15481483@N06/6231443466/
[96]http://creativecommons.org/licenses/by-sa/2.0/deed.en
[97]http://www.flickr.com/photos/tjgfernandes/2785677276/
[98]http://creativecommons.org/licenses/by/2.0/deed.en

- Biscotti on a Rack[99] (c) 2010 Kirsten Loza. Some rights reserved[100].
- Coffee Spoons[101] (c) 2010 Jenny Downing. Some rights reserved[102].
- Drawing a Doppio[103] (c) 2008 Osman Bas. Some rights reserved[104].
- Cupping Coffees[105] (c) 2011 Dennis Tang. Some rights reserved[106].
- Three Coffee Roasters[107] (c) 2009 Michael Allen Smith. Some rights reserved[108].
- Blue Diedrich Roaster[109] (c) 2010 Michael Allen Smith. Some rights reserved[110].
- Red Diedrich Roaster[111] (c) 2009 Richard Masoner. Some rights reserved[112].
- Roaster with Tree Leaves[113] (c) 2007 ting. Some rights reserved[114].
- Half Drunk[115] (c) 2010 Nicholas Lundgaard. Some rights reserved[116].
- Anticipation[117] (c) 2012 Paul McCoubrie. Some rights reserved[118].
- Ooh![119] (c) 2012 Michael Coghlan. Some rights reserved[120].
- Intestines of an Espresso Machine[121] (c) 2011 Angie Chung. Some rights reserved[122].
- Bezzera Espresso Machine[123] (c) 2011 Andrew Nash. Some rights reserved[124]. *Beans Ripening on a Branch[125] (c) 2008 John Pavelka. Some rights reserved[126].
- Cafe Macchiato on Gazotta Della Sport[127] (c) 2008 Jon Shave. Some rights reserved[128].
- Jars of Coffee Beans[129] (c) 2012 Memphis CVB. Some rights reserved[130].

[99] http://www.flickr.com/photos/kirstenloza/4805716699/
[100] http://creativecommons.org/licenses/by/2.0/deed.en
[101] http://www.flickr.com/photos/jenny-pics/5053954146/
[102] http://creativecommons.org/licenses/by/2.0/deed.en
[103] http://www.flickr.com/photos/33388953@N04/4017985434/
[104] http://creativecommons.org/licenses/by/2.0/deed.en
[105] http://www.flickr.com/photos/tangysd/5953453156/
[106] http://creativecommons.org/licenses/by-sa/2.0/deed.en
[107] http://www.flickr.com/photos/digitalcolony/4000837035/
[108] http://creativecommons.org/licenses/by-sa/2.0/deed.en
[109] http://www.flickr.com/photos/digitalcolony/4309812256/
[110] http://creativecommons.org/licenses/by-sa/2.0/deed.en
[111] http://www.flickr.com/photos/bike/3237859728/
[112] http://creativecommons.org/licenses/by-sa/2.0/deed.en
[113] http://www.flickr.com/photos/lacerabbit/2102801319/
[114] http://creativecommons.org/licenses/by-nd/2.0/deed.en
[115] http://www.flickr.com/photos/nalundgaard/4785922266/
[116] http://creativecommons.org/licenses/by-sa/2.0/deed.en
[117] http://www.flickr.com/photos/paulmccoubrie/6828131856/
[118] http://creativecommons.org/licenses/by-nd/2.0/deed.en
[119] http://www.flickr.com/photos/mikecogh/7676649034/
[120] http://creativecommons.org/licenses/by-sa/2.0/deed.en
[121] http://www.flickr.com/photos/yellowskyphotography/5641003165/
[122] http://creativecommons.org/licenses/by-sa/2.0/deed.en
[123] http://www.flickr.com/photos/andynash/6204253236/
[124] http://creativecommons.org/licenses/by-sa/2.0/deed.en
[125] http://www.flickr.com/photos/28705377@N04/5306009552/
[126] http://creativecommons.org/licenses/by/2.0/deed.en
[127] http://www.flickr.com/photos/shavejonathan/2343081208/
[128] http://creativecommons.org/licenses/by/2.0/deed.en
[129] http://www.flickr.com/photos/ilovememphis/7103931235/
[130] http://creativecommons.org/licenses/by-nd/2.0/deed.en

- Types of Coffee Drinks[131] (c) 2012 Michael Coghlan. Some rights reserved[132].
- Coffee Trees[133] (c) 2011 Dave Townsend. Some rights reserved[134].
- Cafe do Brasil[135] (c) 2003 Temporalata. Some rights reserved[136].
- Brown Cups[137] (c) 2007 Michael Allen Smith. Some rights reserved[138].
- Mirage[139] (c) 2010 Mira Helder. Some rights reserved[140].
- Coffee Van with Bullet Holes[141] (c) 2006 Jon Crel. Some rights reserved[142].

[131] http://www.flickr.com/photos/mikecogh/7561440544/
[132] http://creativecommons.org/licenses/by-sa/2.0/deed.en
[133] http://www.flickr.com/photos/dtownsend/6171015997/
[134] http://creativecommons.org/licenses/by-sa/2.0/deed.en
[135] http://www.flickr.com/photos/93425126@N00/313053257/
[136] http://creativecommons.org/licenses/by-sa/2.0/deed.en
[137] http://www.flickr.com/photos/digitalcolony/2833809436/
[138] http://creativecommons.org/licenses/by-sa/2.0/deed.en
[139] http://www.flickr.com/photos/citizenhelder/5006498068/
[140] http://creativecommons.org/licenses/by/2.0/deed.en
[141] http://www.flickr.com/photos/joncrel/237026246/
[142] http://creativecommons.org/licenses/by-nd/2.0/deed.en

About The Author

When he's not shipping CoffeeScript, Ruby, JavaScript and Java applications scaling out to millions of users, Reg "Raganwald" Braithwaite has authored libraries[143] for CoffeeScript, JavaScript and Ruby programming such as Method Combinators, Katy, JQuery Combinators, YouAreDaChef, andand, and others.

He writes about programming on his "Homoiconic[144]" un-blog as well as general-purpose ruminations on his posterous space[145]. He is also known for authoring the popular raganwald programming blog[146] from 2005-2008.

contact

Twitter: @raganwald[147]
Email: reg@braythwayt.com[148]

[143]http://github.com/raganwald
[144]http://github.com/raganwald/homoiconic
[145]http://raganwald.posterous.com
[146]http://weblog.raganwald.com
[147]https://twitter.com/raganwald
[148]mailto:reg@braythwayt.com

Reg “Raganwald” Braithwaite