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

Also By raganwald

Kestrels, Quirky Birds, and Hopeless Egocentricity

JavaScript Allongé

JavaScript Spessore

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.

CoffeeScript Ristretto

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

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[4], because we have no way of changing state within a program using the tools we have examined.

It's time to change *everything*.

[4] 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[5] 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.

[5] 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:

```coffeescript
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.[6]

> 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.

[6] 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[7], a mechanism for working with collections that was popularized by Python.

Here's a sample comprehension:

[7] 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[8] 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.

[8] 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[9]

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[10], 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[11]

Consider a stack[12] 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.

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

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

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

[12] 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[13]), 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()
```

[13] 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 [14].

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))
```

[14] 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*.

```
27            attributes = redoStack.pop()
28          obj
29        get: (key) ->
30          attributes(key)
31        has: (key) ->
32          attributes.hasOwnProperty(key)
33        attributes: ->
34          shallowCopy(attributes)
35      }
36      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[15]:

```
1  QueueMaker = ->
2    do (array = [], head = 0, tail = -1) ->
3      pushTail: (value) ->
4        array[tail += 1] = value
5      pullHead: ->
6        if tail >= head
7          do (value = array[head]) ->
8            array[head] = undefined
9            head += 1
10           value
11     isEmpty: ->
12       tail < head
```

Now we wish to create a deque[16] by adding `pullTail` and `pushHead` operations to our queue.[17] Unfortunately, encapsulation prevents us from adding operations that interact with the hidden data structures.

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

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

[17] 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) ->
```

```
19          (value) ->
20            if deque.head is 0
21              for i in [deque.tail..deque.head]
22                deque.array[i + INCREMENT] = deque.array[i]
23              deque.tail += INCREMENT
24              deque.head += INCREMENT
25            deque.array[deque.head -= 1] = value
26      )
```

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.[18] Now we can easily make copies:

[18] 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`.

interlude...

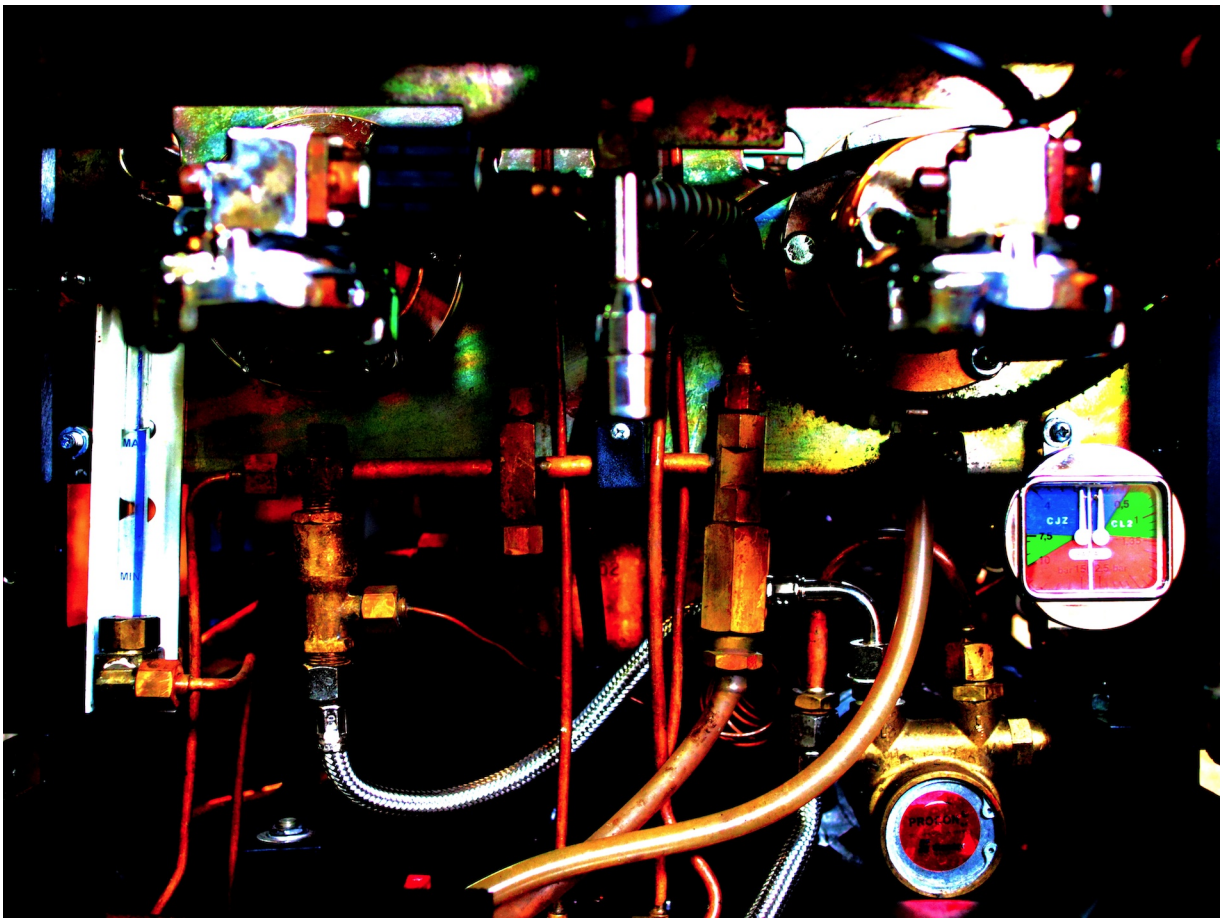
A beautiful espresso machine

Michael Allen Smith[19] 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."

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

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[20]

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:

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

We'll time it:

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

Why is it so slow? Well, it has a nasty habit of recalculating the same results over and over and over again. We could rearrange the computation to avoid this, but let's be lazy and trade space for time. What we want to do is use a lookup table. Whenever we want a result, we look it up. If we don't have it, we calculate it and write the result in the table to use in the future. If we do have it, we return the result without recalculating it.

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[21] kinds of functions:

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

[21] 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[22]:

```
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[23]. We'll call the combinator `once`:

[22] http://refactoring.com/catalog/extractMethod.html

[23] 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[24] 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:

[24] 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[25] 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.

[25] 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.[26]

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:

[26] 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[27] 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:

[27] 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[28] module. They can be used with all CoffeeScript and JavaScript projects.

[28] 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[29], fetching photos from a remote photo sharing site using their asynchronous API:

[29] 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[30]. A *promise* is an object that acts as a state machine.[31] 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)`.[32] `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):

```
1 fotositeGetPhotosByTag(tag)
2   .then fotositeGetOneFromList
3   .then fotositeGetPhoto
4   .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[33], and when[34].

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

[31] 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.

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

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

[34] 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!

About These Sample Chapters

This sample edition of the book includes both the fourth and the sixth full chapters. You have both "Objects, Mutation, and State" as well as "An Extra Shot of Ideas" in their entirety. There's a **lot** more in the full book, so download it today[35]! Besides, **there's really no risk at all**. If you read *CoffeeScript Ristretto* and it doesn't blow your mind, your money will be cheerfully refunded.

–Reginald "Raganwald" Braithwaite, Toronto, 2012

[35] http://leanpub.com/coffeescript-ristretto

No, this is not the author: But he has free coffee!

How to run the examples

If you follow the instructions at coffeescript.org[36] to install NodeJS and CoffeeScript,[37] you can run an interactive CoffeeScript REPL[38] 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[39], 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[40]. 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`.

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

[37] 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.

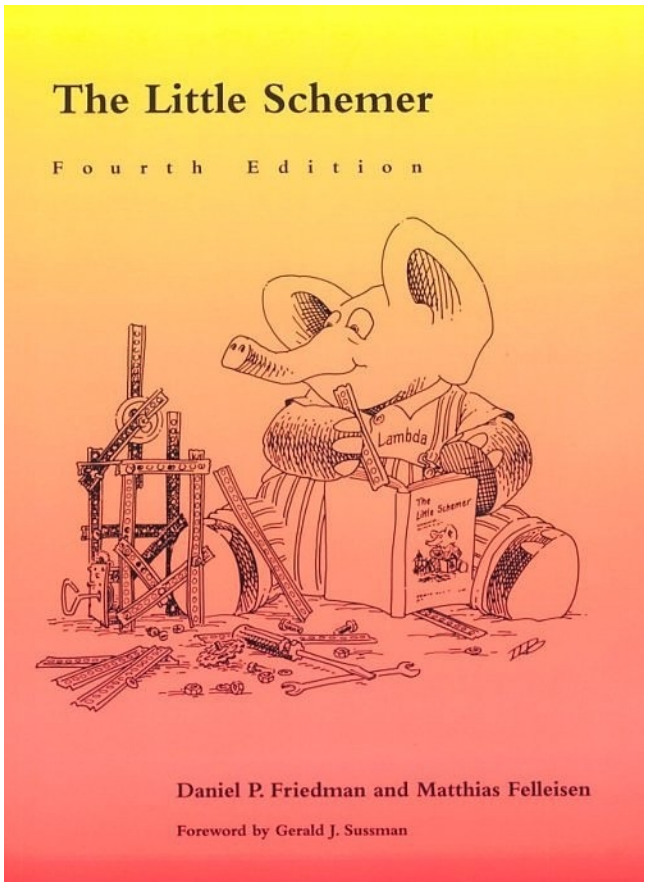
[38] https://en.wikipedia.org/wiki/REPL

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

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

Thanks!

Daniel Friedman and Matthias Felleisen

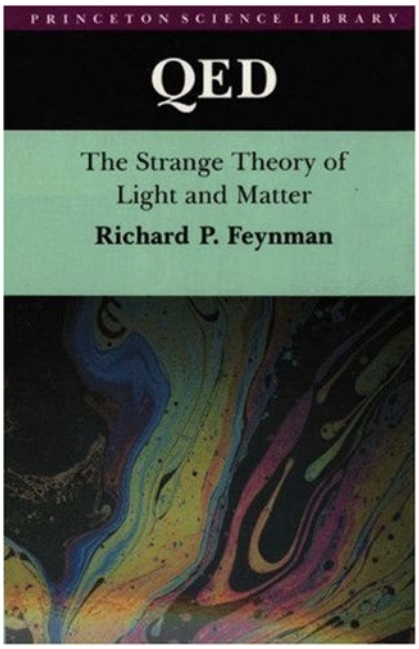


The Little Schemer

CoffeeScript Ristretto was inspired by The Little Schemer[41] 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**.

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

Richard Feynman

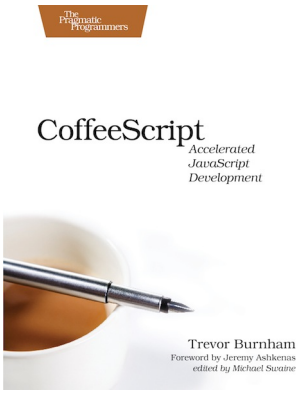


QED: The Strange Theory of Light and Matter

Richard Feynman's QED[42] 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.

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

Trevor Burnham



CoffeeScript: Accelerated JavaScript Development

Trevor Burnham provided invaluable assistance with this book. Trevor is the author of CoffeeScript: Accelerated JavaScript Development[43], an excellent resource for CoffeeScript programmers.

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

JavaScript Allongé

a long and strong programming book

JavaScript Allongé[44] is the companion book to *CoffeeScript Ristretto*.

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

Copyright Notice

images

- The picture of the author is (c) 2008, Joseph Hurtado[47], All Rights Reserved.
- Double ristretto menu[48] (c) 2010, Michael Allen Smith. Some rights reserved[49].
- Short espresso shot in a white cup with blunt handle[50] (c) 2007, EVERYDAYLIFEMODERN. Some rights reserved[51].
- Espresso shot in a caffe molinari cup[52] (c) 2007, EVERYDAYLIFEMODERN. Some rights reserved[53].
- Beans in a Bag[54] (c) 2008, Stirling Noyes. Some Rights Reserved[55].
- Free Samples[56] (c) 2011, Myrtle Bech Digitel. Some Rights Reserved[57].
- Free Coffees[58] image (c) 2010, Michael Francis McCarthy. Some Rights Reserved[59].
- La Marzocco[60] (c) 2009, Michael Allen Smith. Some rights reserved[61].
- Cafe Diplomatico[62] (c) 2011, Missi. Some rights reserved[63].
- Sugar Service[64] (c) 2008 Tiago Fernandes. Some rights reserved[65].

[45] https://leanpub.com/coffeescript-ristretto
[46] http://creativecommons.org/licenses/by-nc-nd/3.0/
[47] http://www.flickr.com/photos/trumpetca/
[48] http://www.flickr.com/photos/digitalcolony/5054568279/
[49] http://creativecommons.org/licenses/by-sa/2.0/deed.en
[50] http://www.flickr.com/photos/everydaylifemodern/1353570874/
[51] http://creativecommons.org/licenses/by-nd/2.0/deed.en
[52] http://www.flickr.com/photos/everydaylifemodern/434299813/
[53] http://creativecommons.org/licenses/by-nd/2.0/deed.en
[54] http://www.flickr.com/photos/the_rev/2295096211/
[55] http://creativecommons.org/licenses/by/2.0/deed.en
[56] http://www.flickr.com/photos/thedigitelmyr/6199419022/
[57] http://creativecommons.org/licenses/by-sa/2.0/deed.en
[58] http://www.flickr.com/photos/sagamiono/4391542823/
[59] http://creativecommons.org/licenses/by-sa/2.0/deed.en
[60] http://www.flickr.com/photos/digitalcolony/3924227011/
[61] http://creativecommons.org/licenses/by-sa/2.0/deed.en
[62] http://www.flickr.com/photos/15481483@N06/6231443466/
[63] http://creativecommons.org/licenses/by-sa/2.0/deed.en
[64] http://www.flickr.com/photos/tjgfernandes/2785677276/
[65] http://creativecommons.org/licenses/by/2.0/deed.en

- Biscotti on a Rack[66] (c) 2010 Kirsten Loza. Some rights reserved[67].
- Coffee Spoons[68] (c) 2010 Jenny Downing. Some rights reserved[69].
- Drawing a Doppio[70] (c) 2008 Osman Bas. Some rights reserved[71].
- Cupping Coffees[72] (c) 2011 Dennis Tang. Some rights reserved[73].
- Three Coffee Roasters[74] (c) 2009 Michael Allen Smith. Some rights reserved[75].
- Blue Diedrich Roaster[76] (c) 2010 Michael Allen Smith. Some rights reserved[77].
- Red Diedrich Roaster[78] (c) 2009 Richard Masoner. Some rights reserved[79].
- Roaster with Tree Leaves[80] (c) 2007 ting. Some rights reserved[81].
- Half Drunk[82] (c) 2010 Nicholas Lundgaard. Some rights reserved[83].
- Anticipation[84] (c) 2012 Paul McCoubrie. Some rights reserved[85].
- Ooh![86] (c) 2012 Michael Coghlan. Some rights reserved[87].
- Intestines of an Espresso Machine[88] (c) 2011 Angie Chung. Some rights reserved[89].
- Bezzera Espresso Machine[90] (c) 2011 Andrew Nash. Some rights reserved[91]. *Beans Ripening on a Branch[92] (c) 2008 John Pavelka. Some rights reserved[93].
- Cafe Macchiato on Gazotta Della Sport[94] (c) 2008 Jon Shave. Some rights reserved[95].
- Jars of Coffee Beans[96] (c) 2012 Memphis CVB. Some rights reserved[97].

---

[66] http://www.flickr.com/photos/kirstenloza/4805716699/
[67] http://creativecommons.org/licenses/by/2.0/deed.en
[68] http://www.flickr.com/photos/jenny-pics/5053954146/
[69] http://creativecommons.org/licenses/by/2.0/deed.en
[70] http://www.flickr.com/photos/33388953@N04/4017985434/
[71] http://creativecommons.org/licenses/by/2.0/deed.en
[72] http://www.flickr.com/photos/tangysd/5953453156/
[73] http://creativecommons.org/licenses/by-sa/2.0/deed.en
[74] http://www.flickr.com/photos/digitalcolony/4000837035/
[75] http://creativecommons.org/licenses/by-sa/2.0/deed.en
[76] http://www.flickr.com/photos/digitalcolony/4309812256/
[77] http://creativecommons.org/licenses/by-sa/2.0/deed.en
[78] http://www.flickr.com/photos/bike/3237859728/
[79] http://creativecommons.org/licenses/by-sa/2.0/deed.en
[80] http://www.flickr.com/photos/lacerabbit/2102801319/
[81] http://creativecommons.org/licenses/by-nd/2.0/deed.en
[82] http://www.flickr.com/photos/nalundgaard/4785922266/
[83] http://creativecommons.org/licenses/by-sa/2.0/deed.en
[84] http://www.flickr.com/photos/paulmccoubrie/6828131856/
[85] http://creativecommons.org/licenses/by-nd/2.0/deed.en
[86] http://www.flickr.com/photos/mikecogh/7676649034/
[87] http://creativecommons.org/licenses/by-sa/2.0/deed.en
[88] http://www.flickr.com/photos/yellowskyphotography/5641003165/
[89] http://creativecommons.org/licenses/by-sa/2.0/deed.en
[90] http://www.flickr.com/photos/andynash/6204253236/
[91] http://creativecommons.org/licenses/by-sa/2.0/deed.en
[92] http://www.flickr.com/photos/28705377@N04/5306009552/
[93] http://creativecommons.org/licenses/by/2.0/deed.en
[94] http://www.flickr.com/photos/shavejonathan/2343081208/
[95] http://creativecommons.org/licenses/by/2.0/deed.en
[96] http://www.flickr.com/photos/ilovememphis/7103931235/
[97] http://creativecommons.org/licenses/by-nd/2.0/deed.en

- Types of Coffee Drinks[98] (c) 2012 Michael Coghlan. Some rights reserved[99].
- Coffee Trees[100] (c) 2011 Dave Townsend. Some rights reserved[101].
- Cafe do Brasil[102] (c) 2003 Temporalata. Some rights reserved[103].
- Brown Cups[104] (c) 2007 Michael Allen Smith. Some rights reserved[105].
- Mirage[106] (c) 2010 Mira Helder. Some rights reserved[107].
- Coffee Van with Bullet Holes[108] (c) 2006 Jon Crel. Some rights reserved[109].

[98] http://www.flickr.com/photos/mikecogh/7561440544/
[99] http://creativecommons.org/licenses/by-sa/2.0/deed.en
[100] http://www.flickr.com/photos/dtownsend/6171015997/
[101] http://creativecommons.org/licenses/by-sa/2.0/deed.en
[102] http://www.flickr.com/photos/93425126@N00/313053257/
[103] http://creativecommons.org/licenses/by-sa/2.0/deed.en
[104] http://www.flickr.com/photos/digitalcolony/2833809436/
[105] http://creativecommons.org/licenses/by-sa/2.0/deed.en
[106] http://www.flickr.com/photos/citizenhelder/5006498068/
[107] http://creativecommons.org/licenses/by/2.0/deed.en
[108] http://www.flickr.com/photos/joncrel/237026246/
[109] 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[110] for CoffeeScript, JavaScript and Ruby programming such as Method Combinators, Katy, JQuery Combinators, YouAreDaChef, andand, and others.

He writes about programming on his "Homoiconic[111]" un-blog as well as general-purpose ruminations on his posterous space[112]. He is also known for authoring the popular raganwald programming blog[113] from 2005-2008.

contact

Twitter: @raganwald[114]
Email: reg@braythwayt.com[115]

[110] http://github.com/raganwald
[111] http://github.com/raganwald/homoiconic
[112] http://raganwald.posterous.com
[113] http://weblog.raganwald.com
[114] https://twitter.com/raganwald
[115] mailto:reg@braythwayt.com

Reg “Raganwald” Braithwaite