JavaScript

Allongé

JavaScript Allongé

A strong cup of functions, objects, combinators, and decorators

Reginald Braithwaite

This book is for sale at http://leanpub.com/javascript-allonge

This version was published on 2013-10-03

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

Tweet This Book!

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

The suggested hashtag for this book is #javascript-allonge.

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/#javascript-allonge

Also By Reginald Braithwaite

Kestrels, Quirky Birds, and Hopeless Egocentricity

What I've Learned From Failure

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

CoffeeScript Ristretto

This book is dedicated to my daughter, Clara Maude Braithwaite

Contents

A Pull of the Lever: Prefaces

Caffe Molinari

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

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

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

Foreword by Michael Fogus

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

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

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

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

Enjoy.

– Fogus, fogus.me

About The Sample PDF

This sample edition of the book includes just a portion of the complete book. Buying the book in progress entitles you to free updates, so download it today! Besides, **there's really no risk at all**. If you read *JavaScript Allongé* and it doesn't blow your mind, your money will be cheerfully refunded.

–Reginald "Raganwald" Braithwaite, Toronto, 2012

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

Why JavaScript Allongé?

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

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

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

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

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

Second, *JavaScript Allongé* provides recipes for using functions to write software that is simpler, cleaner, and less complicated than alternative approaches that are object-centric or code-centric. JavaScript idioms like function combinators and decorators leverage JavaScript's power to make code easier to read, modify, debug and refactor, thus *avoiding* problems before they happen.

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

how the book is organized

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

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

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

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

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

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

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

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

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

This doesn't look like a factory function: It doesn't have an expression that yields a Plain Old JavaScript Object when the function is applied. Yet, there is a way to make an object out of it. Behold the power of the `new` keyword:

```
new Ur()
  //=> {}
```

We got an object back! What can we find out about this object?

```
new Ur() === new Ur()
  //=> false
```

Every time we call `new` with a function and get an object back, we get a unique object. We could call these "Objects created with the `new` keyword," but this would be cumbersome. So we're going to call them *instances*. Instances of what? Instances of the function that creates them. So given `var i = new Ur()`, we say that `i` is an instance of `Ur`.

For reasons that will be explained after we've discussed prototypes, we also say that `Ur` is the *constructor* of `i`, and that `Ur` is a *constructor function*. Therefore, an instance is an object created by using the `new` keyword on a constructor function, and that function is the instance's constructor.

prototypes

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

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

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

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

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

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

var continent = new Ur();
  //=> {}
continent.language
  //=> 'JavaScript'
```

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

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

You can set elements of an instance, and they "override" the constructor's prototype, but they don't actually change the constructor's prototype. Let's make another instance and try something else.

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

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

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

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

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

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

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

continent.constructor === 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 === 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 our `extends` helper from Composition and Extension:

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

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

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

You recall that when we first looked at `this`, we only covered the case where a function that belongs to an object is invoked. Now we see another case: When a function is invoked by the `new` operator, `this` is set to the new object being created. Thus, our code for `Queue` initializes the queue.

You can see why `this` is so handy in JavaScript: We wouldn't be able to define functions in the prototype that worked on the instance if JavaScript didn't give us an easy way to refer to the instance itself.

objects everywhere?

Now that you know about prototypes, it's time to acknowledge something that even small children know: Everything in JavaScript behaves like an object, everything in JavaScript behaves like an instance of a function, and therefore everything in JavaScript behaves as if it inherits some methods from its constructor's prototype and/or has some elements of its own.

For example:

```
3.14159265.toPrecision(5)
  //=> '3.1415'

'FORTRAN, SNOBOL, LISP, BASIC'.split(', ')
  //=> [ 'FORTRAN',
  #      'SNOBOL',
  #      'LISP',
  #      'BASIC' ]

[ 'FORTRAN',
  'SNOBOL',
  'LISP',
  'BASIC' ].length
//=> 4
```

Functions themselves are instances, and they have methods. For example, every function has a method `call`. `call`'s first argument is a *context*: When you invoke `.call` on a function, it invoked the function, setting `this` to the context. It passes the remainder of the arguments to the function. It seems like objects are everywhere in JavaScript!

impostors

You may have noticed that we use "weasel words" to describe how everything in JavaScript *behaves like* an instance. Everything *behaves as if* it was created by a function with a prototype.

The full explanation is this: As you know, JavaScript has "value types" like `String`, `Number`, and `Boolean`. As noted in the first chapter, value types are also called *primitives*, and one consequence of the way JavaScript implements primitives is that they aren't objects. Which means they can be identical to other values of the same type with the same contents, but the consequence of certain design decisions is that value types don't actually have methods or constructors. They aren't instances of some constructor.

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

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

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

JavaScript pulls some legerdemain. When you do something that treats a value like an object, JavaScript checks to see whether the value actually is an object. If the value is actually a primitive,[3] JavaScript temporarily makes an object that is a kinda-sorta copy of the primitive and that kinda-sorta copy has methods and you are temporarily fooled into thinking that `"Spence Olham"` has a `.split` method.

These kinda-sorta copies are called String *instances* as opposed to String *primitives*. And the instances have methods, while the primitives do not. How does JavaScript make an instance out of a primitive? With `new`, of course. Let's try it:

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

The string instance looks just like our string primitive. But does it behave like a string primitive? Not entirely:

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

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

Aha! It's an object with its own identity, unlike string primitives that behave as if they have a canonical representation. If we didn't care about their identity, that wouldn't be a problem. But if we carelessly used a string instance where we thought we had a string primitive, we could run into a subtle bug:

```
if (userName === "Spence Olham") {
  getMarried();
  goCamping()
}
```

That code is not going to work as we expect should we accidentally bind `new String("Spence Olham")` to `userName` instead of the primitive `"Spence Olham"`.

This basic issue that instances have unique identities but primitives with the same contents have the same identities–is true of all primitive types, including numbers and booleans: If you create an instance of anything with `new`, it gets its own identity.

There are more pitfalls to beware. Consider the truthiness of string, number and boolean primitives:

```
'' ? 'truthy' : 'falsy'
  //=> 'falsy'
0 ? 'truthy' : 'falsy'
  //=> 'falsy'
false ? 'truthy' : 'falsy'
  //=> 'falsy'
```

Compare this to their corresponding instances:

```
new String('') ? 'truthy' : 'falsy'
  //=> 'truthy'
new Number(0) ? 'truthy' : 'falsy'
  //=> 'truthy'
new Boolean(false) ? 'truthy' : 'falsy'
  //=> 'truthy'
```

Our notion of "truthiness" and "falsiness" is that all instances are truthy, even string, number, and boolean instances corresponding to primitives that are falsy.

There is one sure cure for "JavaScript Impostor Syndrome." Just as `new PrimitiveType(...)` creates an instance that is an impostor of a primitive, `PrimitiveType(...)` creates an original, canonicalized primitive from a primitive or an instance of a primitive object.

For example:

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

Getting clever, we can write this:

```
var original = function (unknown) {
  return unknown.constructor(unknown)
}

original(true) === true
  //=> true
original(new Boolean(true)) === true
  //=> true
```

Of course, `original` will not work for your own creations unless you take great care to emulate the same behaviour. But it does work for strings, numbers, and booleans.

Binding Functions to Contexts

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

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

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

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

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

someModel.on('change', function () {
  someView.render()
});
```

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

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

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

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

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

binding methods

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

When we write something like:

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

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

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

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

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

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

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

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

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

Now we know where to find it:

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

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

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

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

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

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

This is one of the realities of "meta-programming." Each technique looks useful and interesting in isolation, but when multiple techniques are used together, they can have unpredictable results. It's not surprising, because most popular languages consider classes and methods to be fairly global, and they handle dynamic changes through side-effects. This is roughly equivalent to programming in 1970s-era BASIC by imperatively changing global variables.

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

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

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

```
AccountModel.prototype.getBalancePromise(forceRemote) = {
  // if forceRemote is true, always goes to the remote
  // database for the most real-time value, returns
  // a promise.
};

var account = new AccountModel(...);

var boundGetRemoteBalancePromise = account.
  getBalancePromise.
  bind(account, true);
```

Very handy, and not just for binding contexts!

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

Partial Application, Binding, and Currying

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

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

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

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

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

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

You can write:

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

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

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

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

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

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

We could write:

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

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

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

currying

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

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

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

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

Here is the curried version:

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

A Class By Any Other Name

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

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

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

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

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

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

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

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

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

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

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

This does exactly the same thing as declaring a "class," defining a "method," and adding a "mixin." How does it differ? It doesn't use the words *class*, *method*, *def(ine)* or *mixin*. And it has this `prototype` property that most other popular languages eschew. It also doesn't deal with inheritance, a deal-breaker for programmers who are attached to taxonomies.

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

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

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

Object Methods

An *instance method* is a function defined in the constructor's prototype. Every instance acquires this behaviour unless otherwise "overridden." Instance methods usually have some interaction with the instance, such as references to `this` or to other methods that interact with the instance. A *constructor method* is a function belonging to the constructor itself.

There is a third kind of method, one that any object (obviously including all instances) can have. An *object method* is a function defined in the object itself. Like instance methods, object methods usually have some interaction with the object, such as references to `this` or to other methods that interact with the object.

Object methods are really easy to create with Plain Old JavaScript Objects, because they're the only kind of method you can use. Recall from This and That:

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

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

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

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

  return obj;
})();
```

Although they don't refer to the object, `push`, `pop`, and `isEmpty` semantically interact with the opaque data structure represented by the object, so they are object methods too.

object methods within instances

Instances of constructors can have object methods as well. Typically, object methods are added in the constructor. Here's a gratuitous example, a widget model that has a read-only `id`:

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

extend(WidgetModel.prototype, {
  set: function (attr, value) {
    this[attr] = value;
    return this;
  },
  get: function (attr) {
```

```
    return this[attr]
  }
});
```

`set` and `get` are instance methods, but `id` is an object method: Each object has its own `id` closure, where `id` is bound to the id of the widget by the argument `id` in the constructor. The advantage of this approach is that instances can have different object methods, or object methods with their own closures as in this case. The disadvantage is that every object has its own methods, which uses up much more memory than instance methods, which are shared amongst all instances.

Object methods are defined within the object. So if you have several different "instances" of the same object, there will be an object method for each object. Object methods can be associated with any object, not just those created with the `new` keyword. Instance methods apply to instances, objects created with the `new` keyword. Instance methods are defined in a prototype and are shared by all instances.

Extending Classes with Inheritance

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

Here's our `Queue`:

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

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

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

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

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

Dequeue.INCREMENT = 4;

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

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

    if (this.head === 0) {
      for (i = this.tail; i >= this.head; --i) {
        this.array[i + INCREMENT] = this.array[i]
      }
      this.tail += this.constructor.INCREMENT;
      this.head += this.constructor.INCREMENT
    }
    this.array[this.head -= 1] = value
  }
});
```

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

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

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

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

Here's such a proxy:

```
var QueueProxy = function () {}

QueueProxy.prototype = Queue.prototype
```

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

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

Dequeue.INCREMENT = 4;

Dequeue.prototype = new QueueProxy();

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

    if (!this.isEmpty()) {
      value = this.array[this.tail];
      this.array[this.tail] = void 0;
      this.tail -= 1;
```

```
      return value
    }
  },
  pushHead: function (value) {
    var i;

    if (this.head === 0) {
      for (i = this.tail; i >= this.head; --i) {
        this.array[i + INCREMENT] = this.array[i]
      }
      this.tail += this.constructor.INCREMENT;
      this.head += this.constructor.INCREMENT
    }
    this.array[this.head -= 1] = value
  }
});
```

And it seems to work:

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

Wonderful!

getting the constructor element right

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

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

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

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

Repeat. Now it works:

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

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

extracting the boilerplate

Let's turn our mechanism into a function:

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

And use it in `Dequeue`:

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

Dequeue.INCREMENT = 4;

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

```
    var value;

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

    if (this.head === 0) {
      for (i = this.tail; i >= this.head; --i) {
        this.array[i + INCREMENT] = this.array[i]
      }
      this.tail += this.constructor.INCREMENT;
      this.head += this.constructor.INCREMENT
    }
    this.array[this.head -= 1] = value
  }
});
```

future directions

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

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

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

Summary

Instances and Classes

- The `new` keyword turns any function into a *constructor* for creating *instances*.
- All functions have a `prototype` element.
- Instances behave as if the elements of their constructor's prototype are their elements.
- Instances can override their constructor's prototype without altering it.
- The relationship between instances and their constructor's prototype is dynamic.
- `this` works seamlessly with methods defined in prototypes.
- Everything behaves like an object.
- JavaScript can convert primitives into instances and back into primitives.
- Object methods are typically created in the constructor and are private to each object.
- Prototypes can be chained to allow extension of instances.

And most importantly:

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

2 Selected Recipes

Time to enjoy some tasty JavaScript

mapWith

In recent versions of JavaScript, arrays have a `.map` method. Map takes a function as an argument, and applies it to each of the elements of the array, then returns the results in another array. For example:

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

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

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

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

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

Here's the above code written using `mapWith`:

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

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

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

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

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

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

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

`mapWith` *was suggested by ludicast*

getWith

`getWith` is a very simple function. It takes the name of an attribute and returns a function that extracts the value of that attribute from an object:

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

You can use it like this:

```
var inventory = {
  apples: 0,
  oranges 144,
  eggs: 36
};

getWith('oranges')(inventory)
  //=> 144
```

This isn't much of a recipe yet. But let's combine it with mapWith:

```
var inventories = [
  { apples: 0, oranges: 144, eggs: 36 },
  { apples: 240, oranges: 54, eggs: 12 },
  { apples: 24, oranges: 12, eggs: 42 }
];

mapWith(getWith('oranges'))(inventories)
  //=> [ 144, 54, 12 ]
```

That's nicer than writing things out "longhand:"

```
mapWith(function (inventory) { return inventory.oranges })(inventories)
  //=> [ 144, 54, 12 ]
```

`getWith` plays nicely with maybe as well. Consider a sparse array. You can use:

```
mapWith(maybe(getWith('oranges')))
```

To get the orange count from all the non-null inventories in a list.

what's in a name?

Why is this called `getWith`? Consider this function that is common in languages that have functions and dictionaries but not methods:

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

You might ask, "Why use a function instead of just using `[]`?" The answer is, we can manipulate functions in ways that we can't manipulate syntax. For example, do you remember from filp that we can define `mapWith` from `map`?

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

We can do the same thing with `getWith`, and that's why it's named in this fashion:

```
var getWith = flip(get)
```

pluckWith

This pattern of combining mapWith and getWith is very frequent in JavaScript code. So much so, that we can take it up another level:

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

Or even better:

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

And now we can write:

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

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

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

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

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

vs.

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

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

```
var pluckWith = flip(_.pluck);
```

Currying

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

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

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

  return given([]);

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

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

}

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

var curried = curry(sumOfFour);

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

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

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

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

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

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

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

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

Bound

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

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

Simple and useful. But now that we've spent some time looking at objects with methods we can see that `get` (and `pluck`) has a failure mode. Specifically, it's not very useful if we ever want to get a *method*, since we'll lose the context. Consider some hypothetical class:

```
function InventoryRecord (apples, oranges, eggs) {
  this.record = {
    apples: apples,
    oranges: oranges,
    eggs: eggs
  }
}

InventoryRecord.prototype.apples = function apples () {
  return this.record.apples
}

InventoryRecord.prototype.oranges = function oranges () {
  return this.record.oranges
}

InventoryRecord.prototype.eggs = function eggs () {
  return this.record.eggs
}

var inventories = [
  new InventoryRecord( 0, 144, 36 ),
  new InventoryRecord( 240, 54, 12 ),
  new InventoryRecord( 24, 12, 42 )
];
```

Now how do we get all the egg counts?

```
mapWith(getWith('eggs'))(inventories)
  //=> [ [Function: eggs],
  //     [Function: eggs],
  //     [Function: eggs] ]
```

And if we try applying those functions...

```
mapWith(getWith('eggs'))(inventories).map(
  function (unboundmethod) {
    return unboundmethod()
  }
)
  //=> TypeError: Cannot read property 'eggs' of undefined
```

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

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

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

mapWith(bound('eggs'))(inventories).map(
  function (boundmethod) {
    return boundmethod()
  }
)
  //=> [ 36, 12, 42 ]
```

`bound` is the recipe for getting a bound method from an object by name. It has other uses, such as callbacks. `bound('render')(aView)` is equivalent to `aView.render.bind(aView)`. There's an option to add a variable number of additional arguments, handled by:

```
return function (instance) {
  return Function.prototype.bind.apply(
    instance[messageName], [instance].concat(args)
  )
}
```

The exact behaviour will be covered in Binding Functions to Contexts. You can use it like this to add arguments to the bound function to be evaluated:

```
InventoryRecord.prototype.add = function (item, amount) {
  this.record[item] || (this.record[item] = 0);
  this.record[item] += amount;
  return this;
}

mapWith(bound('add', 'eggs', 12))(inventories).map(
  function (boundmethod) {
    return boundmethod()
  }
)
  //=> [ { record:
  //       { apples: 0,
  //         oranges: 144,
  //         eggs: 48 } },
  //     { record:
  //       { apples: 240,
  //         oranges: 54,
  //         eggs: 24 } },
  //     { record:
  //       { apples: 24,
  //         oranges: 12,
  //         eggs: 54 } } ]
```

Send

Previously, we saw that the recipe bound can be used to get a bound method from an instance. Unfortunately, invoking such methods is a little messy:

```
mapWith(bound('eggs'))(inventories).map(
  function (boundmethod) {
    return boundmethod()
  }
)
  //=> [ 36, 12, 42 ]
```

As we noted, it's ugly to write

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

So instead, we write a new recipe:

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

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

mapWith(send('apples'))(inventories)
  //=> [ 0, 240, 24 ]
```

`send('apples')` works very much like `&:apples` in the Ruby programming language. You may ask, why retain `bound`? Well, sometimes we want the function but don't want to evaluate it immediately, such as when creating callbacks. `bound` does that well.

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

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

Invoke

Send is useful when invoking a function that's a member of an object (or of an instance's prototype). But we sometimes want to invoke a function that is designed to be executed within an object's context. This happens most often when we want to "borrow" a method from one "class" and use it on another object.

It's not an unprecedented use case. The Ruby programming language has a handy feature called instance_exec. It lets you execute an arbitrary block of code in the context of any object. Does this sound familiar? JavaScript has this exact feature, we just call it `.apply` (or `.call` as the case may be). We can execute any function in the context of any arbitrary object.

The only trouble with `.apply` is that being a method, it doesn't compose nicely with other functions like combinators. So, we create a function that allows us to use it as a combinator:

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

function invoke (fn) {
  var args = __slice.call(arguments, 1);

  return function (instance) {
    return fn.apply(instance, args)
  }
}
```

For example, let's say someone else's code gives you an array of objects that are in part, but not entirely like arrays. Something like:

```
var data = [
  { 0: 'zero',
    1: 'one',
    2: 'two',
    foo: 'foo',
    length: 3 },
  // ...
];
```

We can use the pattern from Partial Application, Binding, and Currying to create a context-dependent copy function:

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

And now we can compose `mapWith` with `invoke` to convert the data to arrays:

```
mapWith(invoke(__copy))(data)
  //=> [
  //     [ 'zero', 'one', 'two' ],
  //     // ...
  //   ]
```

`invoke` is useful when you have the function and are looking for the instance. It can be written "the other way around," for when you have the instance and are looking for the function:

```
function instanceEval (instance) {
  return function (fn) {
    var args = __slice.call(arguments, 1);

    return fn.apply(instance, args)
  }
}

var args = instanceEval(arguments)(__slice, 0);
```

Fluent

Object and instance methods can be bifurcated into two classes: Those that query something, and those that update something. Most design philosophies arrange things such that update methods return the value being updated. For example:

```
function Cake () {}

extend(Cake.prototype, {
  setFlavour: function (flavour) {
    return this.flavour = flavour
  },
  setLayers: function (layers) {
    return this.layers = layers
  },
  bake: function () {
    // do some baking
  }
});

var cake = new Cake();
cake.setFlavour('chocolate');
cake.setLayers(3);
cake.bake();
```

Having methods like `setFlavour` return the value being set mimics the behaviour of assignment, where `cake.flavour = 'chocolate'` is an expression that in addition to setting a property also evaluates to the value `'chocolate'`.

The fluent style presumes that most of the time when you perform an update you are more interested in doing other things with the receiver then the values being passed as argument(s), so the rule is to return the receiver unless the method is a query:

```
function Cake () {}

extend(Cake.prototype, {
  setFlavour: function (flavour) {
    this.flavour = flavour;
    return this
  },
  setLayers: function (layers) {
    this.layers = layers;
```

```
    return this
  },
  bake: function () {
    // do some baking
    return this
  }
});
```

The code to work with cakes is now easier to read and less repetitive:

```
var cake = new Cake().
  setFlavour('chocolate').
  setLayers(3).
  bake();
```

For one-liners like setting a property, this is fine. But some functions are longer, and we want to signal the intent of the method at the top, not buried at the bottom. Normally this is done in the method's name, but fluent interfaces are rarely written to include methods like `setLayersAndReturnThis`.

The `fluent` method decorator solves this problem:

```
function fluent (methodBody) {
  return function () {
    methodBody.apply(this, arguments);
    return this;
  }
}
```

Now you can write methods like this:

```
Cake.prototype.bake = fluent( function () {
  // do some baking
  // using many lines of code
  // and possibly multiple returns
});
```

It's obvious at a glance that this method is "fluent."

The Golden Crema

You've earned a break!

Copyright Notice

images

- The picture of the author is (c) 2008, Joseph Hurtado, All Rights Reserved.
- Cover image (c) 2010, avlxyz. Some rights reserved.
- Double ristretto menu (c) 2010, Michael Allen Smith. Some rights reserved.
- Short espresso shot in a white cup with blunt handle (c) 2007, EVERYDAYLIFEMODERN. Some rights reserved.
- Espresso shot in a caffe molinari cup (c) 2007, EVERYDAYLIFEMODERN. Some rights reserved.
- Beans in a Bag (c) 2008, Stirling Noyes. Some Rights Reserved.
- Free Samples (c) 2011, Myrtle Bech Digitel. Some Rights Reserved.
- Free Coffees image (c) 2010, Michael Francis McCarthy. Some Rights Reserved.
- La Marzocco (c) 2009, Michael Allen Smith. Some rights reserved.
- Cafe Diplomatico (c) 2011, Missi. Some rights reserved.
- Sugar Service (c) 2008 Tiago Fernandes. Some rights reserved.
- Biscotti on a Rack (c) 2010 Kirsten Loza. Some rights reserved.
- Coffee Spoons (c) 2010 Jenny Downing. Some rights reserved.
- Drawing a Doppio (c) 2008 Osman Bas. Some rights reserved.
- Cupping Coffees (c) 2011 Dennis Tang. Some rights reserved.
- Three Coffee Roasters (c) 2009 Michael Allen Smith. Some rights reserved.
- Blue Diedrich Roaster (c) 2010 Michael Allen Smith. Some rights reserved.
- Red Diedrich Roaster (c) 2009 Richard Masoner. Some rights reserved.
- Roaster with Tree Leaves (c) 2007 ting. Some rights reserved.
- Half Drunk (c) 2010 Nicholas Lundgaard. Some rights reserved.
- Anticipation (c) 2012 Paul McCoubrie. Some rights reserved.
- Ooh! (c) 2012 Michael Coghlan. Some rights reserved.
- Intestines of an Espresso Machine (c) 2011 Angie Chung. Some rights reserved.
- Bezzera Espresso Machine (c) 2011 Andrew Nash. Some rights reserved. *Beans Ripening on a Branch (c) 2008 John Pavelka. Some rights reserved.
- Cafe Macchiato on Gazotta Della Sport (c) 2008 Jon Shave. Some rights reserved.

About The Author

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

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

contact

Twitter: @raganwald
Email: reg@braythwayt.com

Reg "Raganwald" Braithwaite