JavaScript Spessore

A Thick Shot of Objects, Metaobjects, & Protocols

by Reginald "raganwald" Braithwaite

JavaScript Spessore

A Thick Shot of Objects, Metaobjects, & Protocols

Reginald Braithwaite

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

JavaScript Allongé

Contents

Author's Foreword

My last book, JavaScript Allongé[1], was about writing functions, combining functions, and decorating functions. It even explained JavaScript's methods and classes in terms of functions. It had a single-minded focus on thinking in functions. But of course, there is more to programming than just thinking in functions, there is also "thinking in objects."

And that's where **JavaScript Spessore** comes in: To celebrate thinking in objects, starting with the basics, building upon them, and then exploring new ways to think about object-oriented programming.

0, 1, ∞

To truly think in objects, you have to liberate yourself from thinking in terms of any one language's features, because there is more than one way to "do" objects and object-oriented programming. Let's compare a few other languages to JavaScript:

1. *Smalltalk* has objects, but a Smalltalk object's methods and instance variables are distinct from the contents of a Smalltalk container like a dictionary or array. JavaScript's objects are dictionaries, and an object's methods and instance variables are the same thing as its contents.
2. *Ruby* has classes, modules, the metaclasses, and eigenclasses. JavaScript just has objects that are related to each other either with prototype chaining or as instance values.
3. When you invoke a method in *Common Lisp*, You may also be invoking multiple "before," "after," or "around" demons in addition to the method handler. In JavaScript, each method is handled by exactly one function.
4. *Java's* methods cannot be added to or removed from classes once their bytecodes have been loaded. JavaScript's methods can be added and removed at any time.

These four "distinctions" between other languages and JavaScript are also the four pillars of object-oriented programming language semantics:

1. **Objects** are the things we use to encapsulate data and behaviour by exposing methods (and optionally properties).
2. **Metaobjects** like classes or prototypes are objects that define the behaviour of other objects.
3. **Protocols** are the rules by which we figure out what exactly happens when we send a message to an object.
4. **Binding Times** are the rules that determine *when* the behaviour of objects, metaobjects, and protocols can be added, removed, or changed.

When we are truly "thinking in objects," we are thinking in objects, thinking in metaobjects, and thinking in protocols. And for good measure, we are also thinking of when these things are "bound." And that's why JavaScript Spessore's mission is to explore objects, metaobjects, protocols, and to examine the implications of when these behaviours are bound.

[1] https://leanpub.com/javascript-allonge

J(oop)S[2]

You may be thinking to yourself, “This is all very well, but it sounds like it is about object-oriented programming in general and not really about the specifics in JavaScript. Why JavaScript? Why not Lisp or Smalltalk or OCaml or some other language with more built-in facility for different object-oriented approaches?”

The answer is that this is a book for programmers that is about thinking in objects, thinking that works in any OO language. It happens to be written in JavaScript instead of Lisp for the same reason that it happens to be written in English instead of Latin: Because it’s a language we share.

[2] J(oop)S is a plexer, it means “Object-Oriented Programming in JavaScript.”

Taking a page out of LiSP

Teaching Lisp by implementing Lisp is a long-standing tradition. We read book after book, lecture after lecture, blog post after blog post, all explaining how to implement Lisp in Lisp. Christian Queinnec's Lisp in Small Pieces[3] ("**LiSP**") is particularly notable, not just implementing a Lisp in Lisp, but covering a wide range of different semantics within Lisp.

LiSP's approach is to introduce a feature of Lisp, then develop an implementation. The book covers [Lisp-1 vs. Lisp-2[4], then discusses how to implement namespaces, building a simple Lisp-1 and a simple Lisp-2. Another chapter discusses scoping, and again you build interpreters for dynamic and block scoped Lisps.

Building interpreters (and eventually compilers) may seem esoteric compared to tutorials demonstrating how to build a blogging engine, but there's a method to this madness. If you implement block scoping in a "toy" language, you gain a deep understanding of how closures really work in any language. You gain some insight into the implications with respect to memory and performance. If you write a Lisp that rewrites function calls in Continuation Passing Style[5], you can't help but feel comfortable using JavaScript callbacks in Node.js[6].

The simple fact is that *implementing* a language feature teaches you a tremendous amount about how the feature works in a relatively short amount of time. And that goes double for implementing variations on the same feature–like dynamic vs block scoping or single vs multiple namespaces.

That being said, you get the most mileage out of implementing language semantics, not language syntax. Semantics are the *meanings* of programs, syntax is merely the appearance. Writing parsers for both `compose = (a, b) -> (c) -> a(b(c))` in CoffeeScript and `function comnpose (a, b) { return function (c) { return a(b(c)); }; }` in JavaScript wouldn't teach you nearly as much as writing the code that implements closures. And it's the closures that make `compose` work the way it does.

In this book, we are going to implement a number of different programming language semantics, all in JavaScript. We won't be choosing features at random; We aren't going to try to implement every possible type of programming language semantics. We won't explore dynamic vs block scoping, we won't implement call-by-name, and we will ignore the temptation to experiment with lazy evaluation.

We *are* going to implement different object semantics, implement different kinds of metaobjects, and implement different kinds of method protocols. We are going to focus on the semantics of objects, metaobjects, and protocols, because we're interested in understanding "object-oriented programming" and all of its rich possibilities.

---

[3] http://www.amazon.com/gp/product/B00AKE1U6O/ref=as_li_ss_tl?ie=UTF8&camp=1789&creative=390957&creativeASIN=B00AKE1U6O&linkCode=as2&tag=raganwald001-20

[4] A "Lisp-1" has a single namespace for both functions and other values. A "Lisp-2" has separate namespaces for functions and other values. To the extend that JavaScript resembles a Lisp, it resembles a Lisp-1. See The function namespace.

[5] https://en.wikipedia.org/wiki/Continuation-passing_style

[6] http://nodejs.org/about/

In doing so, we'll learn about the principles of object-oriented programming in far more depth than we would if we chose to implement a "practical" example like a blogging engine.

Disclaimer

Writing is a journey, not a destination. This sample documents the direction we're facing as we take the next step.

This sample "document" is provided to illustrate direction JavaScript Spessore[7] is taking. **It is not held out to contain any of the actual book's content**. "Purchases" are being offered to people whose primary motivation is to support the book and encourage me to get it done.

Your purchase does include the right to download any and all versions of the book, as per Leanpub's lean publishing model. You will never be asked to pay more. You have the right to a 100% no-questions-asked refund if you are not satisfied with the progress of the book:

> Within 45 days of purchase you can get a 100% refund on any Leanpub purchase, in two clicks. We process the refunds manually, so they may take a few days to show up. See full terms[8].

> If you buy a Leanpub book you get all the updates to the book for free! All books are available in PDF, EPUB (for iPad) and MOBI (for Kindle). There is no DRM. There is no risk, just guaranteed happiness or your money back.

That being said, writing like this cannot be rushed: progress may be slow relative to a "Learn JavaScript OO in 21 Days" type of book. If you would prefer that there be a substantial amount of work completed, please be patient and check back in a month or so.

Please also bear in mind that pricing and bundling may vary over time. The book may be offered at any price in the future or even be free to read or free to share at some point in the future.

[7] https://leanpub.com/javascript-spessore
[8] https://leanpub.com/terms#returns

A brief introduction to When Things Happen

> If you look at software today, through the lens of the history of engineering, it's certainly engineering of a sort—but it's the kind of engineering that people without the concept of the arch did. Most software today is very much like an Egyptian pyramid with millions of bricks piled on top of each other, with no structural integrity, but just done by brute force and thousands of slaves.
>
> "I would compare the Smalltalk stuff that we did in the '70s with something like a Gothic cathedral. We had two ideas, really. One of them we got from Lisp: late binding. The other one was the idea of objects. Those gave us something a little bit like the arch, so we were able to make complex, seemingly large structures out of very little material, but I wouldn't put us much past the engineering of 1,000 years ago."—Alan Kay[9]

[9] http://www.doc.ic.ac.uk/~susan/475/AlanKay.html

early binding

late binding

Objects and Methods

What is an Object?

Consider Smalltalk's definition of an object:

> A Smalltalk object can do exactly three things: Hold state (references to other objects), receive a message from itself or another object, and in the course of processing a message, send messages to itself or another object.â€"Smalltalk on Wikipedia[10]

Objects seem simple enough: They hold state, they receive messages, they process those messages, and in the course of processing the messages they also send messages. Not states in this definition is an important idea: Objects can *change state* in the course of processing messages. They do this directly by removing, changing, or adding to the references they hold.

We will use this definition throughout the book, as it is sufficient to explain and explore even complex object-oriented programming ideas. However, before plunging onwards we should be careful to recognize that "objects" in many programming languages have slightly different capabilities.

encapsulation

Smalltalk values encapsulation of state. As such, the only way to discover an object's state is to send it a message, and the only way to change an object's state is to send it a message. Some other "OO" languages do not value this highly, and allow objects to query and modify each other's internal state directly.

JavaScript does not enforce encapsulation. We can write:

Objects in JavaScript

```
var counter = new Object({
                value: 0,
                increment: function () {
                  return ++this.value;
                }
              });

counter.increment();
  //=> 1

counter.increment();
  //=> 2
```

[10] https://en.wikipedia.org/wiki/Smalltalk

```
15
16 counter.value = 100
17
18 counter.increment();
```

And we're directly updating our object's state. This is not a small language detail, like arguing whether there should be `unless` expressions as well as `if` expressions: It strikes at the very heart of the philosophy of object-oriented programming.

Now in JavaScript's defense, JavaScript "objects" are also JavaScript "dictionaries," with a convenient syntax for accessing a value by key. In the above example, we are clearly "violating object encapsulation." But what about this variation?

Objects in JavaScript

```
21 var counter = new Object({
22                 value: 0,
23                 increment: function (arg) {
24                   var by = arg == null
25                            ? 1
26                            : arg.by;
27                   return (this.value = this.value + by);
28                 }
29               });
30
31 counter.increment();
32   //=> 1
33
34 counter.increment();
35   //=> 2
36
37 counter.increment({by: 3})
```

stateful, idempotent, and degenerate objects

object-1s, object-2s, and primitive protocols

queries, updates, and degenerate methods

object composition and delegation

state machines and strategies

nouns, verbs and commands

pattern matching

method composition

immediate, forward, and late-binding

me, myself, and i

Metaobjects

templates and creation by value

immediate, forward, and late-binding of metaobjects

degenerate protocols

eigenclasses, again

prototypes vs. classes: metaobject-1s vs. metaobject-2s

contracts and liskov equivalence

metaobjects are not types, and types are not interfaces

Protocols

prototype chaining

single inheritance

mixins and multiple inheritance

resolution: merge, override, and final

template method protocols

method guards and contracts

early and late method composition

multiple dispatch and generic functions

pattern matching protocols