JavaScript Spessore

A Thick Shot of Objects, Metaobjects, & Protocols

by Reginald "raganwald" Braithwaite

JavaScript Spessore

A Thick Shot of Objects, Metaobjects, & Protocols

Reginald Braithwaite

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

This version was published on 2013-12-21

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.

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].

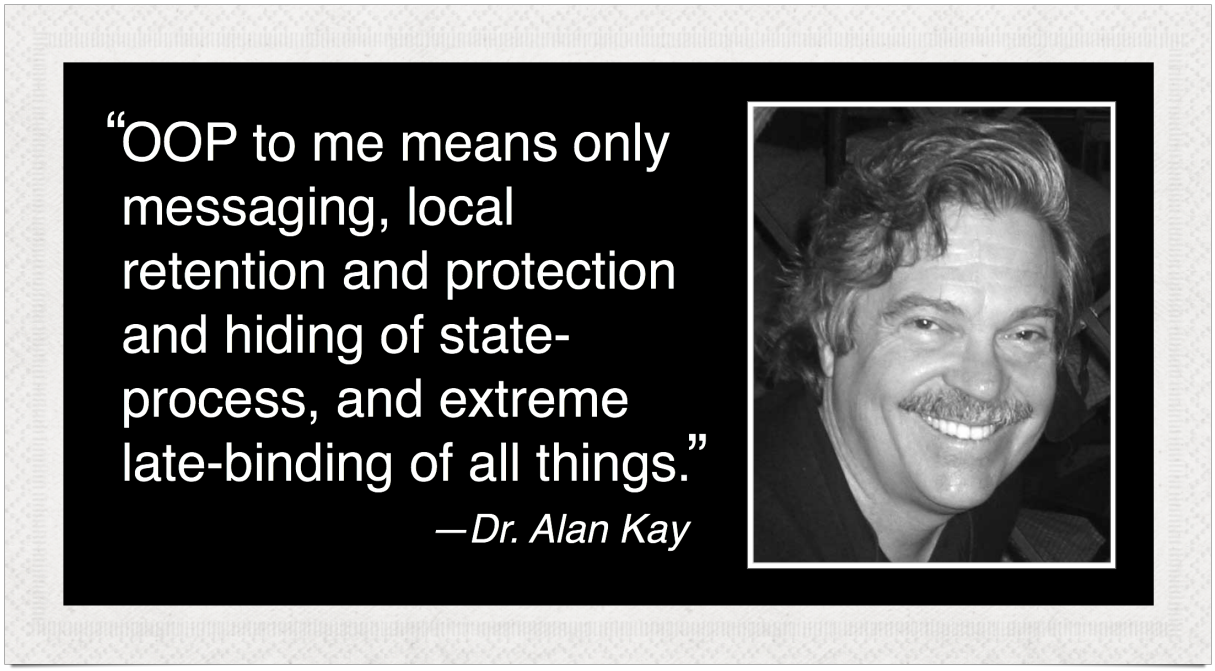
> If you buy a Leanpub book you get all the updates to the book for free! All books are available in PDF, EPUB (for iPad) and MOBI (for Kindle). There is no DRM. There is no risk, just guaranteed happiness or your money back.

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

What is Object-Oriented Programming?



Alan Kay

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[9]

Objects seem simple enough: They hold state, they receive messages, they process those messages, and in the course of processing messages, they also send messages. This brief definition implies 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.

messages and method invocations

A nine year-old messenger boy

Smalltalk speaks of "messages." The metaphor of a message is very clear. A message is composed and sent from one entity (the "sender") to one entity (the "recipient"). The sender specifies the identity of the recipient. The message may contain information for the recipient, instructions to perform some task, or a question to be answered. An immediate reply may be requested, or the sender may trust the message's recipient to act appropriately. The metaphor of the "message" emphasizes the arms-length relationship between sender and recipient.[10]

Popular languages don't usually discuss messages. Instead, they speak of "invoking methods." The word "invoke" means to conjure up and to declare in law. "Invoking a method" has a much more imperative implication than "sending a message." It implies that the entity doing the invoking knows exactly what is going to happen. The relationship is not arms-length.

A method is a kind of recipe. It represents how an object will respond to a message. In JavaScript, methods are functions. In other languages, methods are a separate kind of thing than functions.

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

[10] More exotic messaging protocols are possible. Instead of a message being couriered from one entity to another entity, it could be posted on a public or semi-private space where many recipients could view it and decide for themselves whether to respond. Or perhaps there is a dispatching entity that examines each message and decides who ought to respond, much as an operator might direct your call to the right person within an organization.

references and state

You can imagine sending a message to a mathematician: "What's the biggest number: one, five, or four?" You can do that is JavaScrip as well:

```
Math.max(1, 5, 4)
  //=> 5
```

Although this is a message, it is unsatisfying to think of `Math` as an object, because it doesn't have any state. Objects were invented fifty years ago[11] to model entities when building simulations. When making a simulation, an essential design technique is to make provide entity with its own independent decision-making ability.

Let's say we're modeling traffic. In real life, each car has its own characteristics like maximum speed. Each car has a driver with their own particular style of driving, and of course different cars have different destinations and perhaps different senses of urgency. Each driver independently responds to the local situation around their car. The same goes for traffic lights, roads... Everything has its own independent behavior.

The way to build a simulation is to provide each simulated entity such as the cars, roads, and traffic lights, with their own little programs. You then embed them in a simulated city with a supervisory program doling out events such as rain. Finally, you start the simulation and see what happens.

The key need is to be able to have entities be independent decision-making units that respond to events from outside of themselves. In essence, we're describing computing units. Computation has, at its heart, a program and some kind of storage representing its state. Although impractical, each entity in a simulation could be a Turing Machine with a long tape.

And thus when we think of "objects," we think of independent computing devices, each with their own storage representing their state.

the javascript state of affairs

In JavaScript, objects have one obvious way to represent state. All JavaScript objects can store references to values by key. For example:

```
var snafu = { acronym: 'SNAFU', full: "Situation Normal, All Feduck Up" };
var fubar = { acronym: 'FUBAR', full: "Feduck Up Beyond All Recognition" };
```

In JavaScript, an object can store any value by associating a reference to that value with a key. JavaScript objects are values, so you can also write:

[11] https://en.wikipedia.org/wiki/Simula "The Simula Programming Language"

```
var dictionary = {
  snafu: { acronym: 'SNAFU', full: "Situation Normal, All Feduck Up" },
  fubar: { acronym: 'FUBAR', full: "Feduck Up Beyond All Recognition" }
};
```

JavaScript functions are values, so you can easily associate functions with keys:

```
var math = {
  plusOne: function (number) { return number + 1; }
  minusOne: function (number) { return number - 1; }
};
```

While these are all technically objects, they miss the essence of objects, namely that they are not independent decision-making units with their own storage. Here's something a little closer to the Smalltalk notion of an object:

```
var counter = {
  state: 0,
  prev: function () { return --counter.state; },
  succ: function () { return ++counter.state; }
};
```

This object has two functions, but they are not simply values hanging off it. They are methods, they exist to query and modify the counter's state. When we write:

```
counter.succ();
  //=> 1
```

We are sending the `succ` message to the counter, and it is responding to our message with `1`. This is a very simple example of what we can call a "proper" object, an object that has state and methods that interact with its own state.

In the Smalltalk ideal the only way for objects to interact with each other is through messages. This is not the case in JavaScript. In our above example, we can directly mutate the counter's state without sending it a message:

```
counter.state = 42;
  //=> 42

counter.succ();
  //=> 43
```

In JavaScript, object state is not private by default. You have to either find a way to hide object state, or you have to employ a convention of not having objects meddle with each other's internals.

so what is an object?

In summary, an object is an entity that use handlers to respond to messages. It maintains internal state, and its handlers are responsible for querying and/or updating its state. In a language like JavaScript, we reply on convention to prevent objects from meddling with each other's internal state.

Why objects matter

Coming soon.

abstraction

Stepping away from the precise idea of an object, there is a general principle here, that of abstraction: An *abstraction* is an idea that is independent of it specifics and implementation. For example, a “map” or “dictionary” is an abstraction representing a mapping from keys to values. A “HashMap” is a specific kind of map that distributes values in buckets by hashing their keys.[12]

The notion of abstraction implies some interesting things like information hiding. In the case of a map, some code that uses a map can be ignorant of whether a specific map instance is implemented with a hash or is a naïve list of key-value pairs.

more to come

[12] https://en.wikipedia.org/wiki/Abstraction_(computer_science)

What is Late Binding?

In the preceding discussion about objects, we established that an object has state and it has methods that query and/or update its state. We touched upon the "ideal" being that objects do not directly meddle with each other's state. We did not discuss how an object came to have its behavior defined.

In this example:

```
var counter = {
  state: 0,
  prev: function () { return --counter.state; },
  succ: function () { return ++counter.state; }
};
```

We have an object with two methods. Where are those methods defined? In the code, of course. When will they be defined? When this snippet is evaluated at run time. All that seems obvious, so much so that many people live, love, and make a living writing JavaScript without giving it another thought.

But there is more to think about.

Objects and Methods

encapsulation

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