



# 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 2014-01-31



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.

©2013 - 2014 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

Prefaces . . . . .	i
Taking a page out of LiSP . . . . .	ii
JavaScript Allongé and allong.es . . . . .	iii
Preamble . . . . .	1
Caffé Lungo . . . . .	3
Plain Old JavaScript Objects . . . . .	5
Encapsulating State with Closures . . . . .	7
Composition and Extension . . . . .	13
This and That . . . . .	18
What Context Applies When We Call a Function? . . . . .	22
Extending Objects . . . . .	28
Prototypes are Simple, it's the Explanations that are Hard To Understand . . . . .	31
Binding Functions to Contexts . . . . .	36
Object Methods . . . . .	40
Extending Objects with Inheritance . . . . .	43
Summary . . . . .	49
The Object's The Thing . . . . .	50
Immutable Properties . . . . .	53
Copy On Write Semantics . . . . .	56
Accessors . . . . .	57
Hiding Object Properties . . . . .	62
Object-1s and Object-2s . . . . .	69
To Do . . . . .	74
Methods . . . . .	75
What is a Method? . . . . .	76
The Letter and the Spirit of the Law . . . . .	78
Composite Methods . . . . .	83
Meta-Methods . . . . .	90
Metaobjects . . . . .	97
Templates and Prototypes . . . . .	99

## CONTENTS

The “I” Word . . . . .	107
Constructing Objects . . . . .	110
Metaobject-1s and Metaobject-2s . . . . .	112
Singleton Prototypes . . . . .	113
To Do . . . . .	116
<b>Appendices . . . . .</b>	<b>117</b>
Utility Functions . . . . .	118

# Prefaces



1

---

<sup>1</sup>Group (c) 2013 J MacPherson, some rights reserved

## 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](#)<sup>2</sup> ("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<sup>3</sup>, 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. If you write a Lisp that rewrites function calls in [Continuation Passing Style](#)<sup>4</sup>, you can't help but feel comfortable using JavaScript callbacks in [Node.js](#)<sup>5</sup>.

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.

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.

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.

---

<sup>2</sup>[http://www.amazon.com/gp/product/B00AKE1U6O/ref=as\\_li\\_ss\\_til?ie=UTF8&camp=1789&creative=390957&creativeASIN=B00AKE1U6O&linkCode=as2&tag=raganwald001-20](http://www.amazon.com/gp/product/B00AKE1U6O/ref=as_li_ss_til?ie=UTF8&camp=1789&creative=390957&creativeASIN=B00AKE1U6O&linkCode=as2&tag=raganwald001-20)

<sup>3</sup>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](#).

<sup>4</sup>[https://en.wikipedia.org/wiki/Continuation-passing\\_style](https://en.wikipedia.org/wiki/Continuation-passing_style)

<sup>5</sup><http://nodejs.org/about/>

## JavaScript Allongé and `allong.es`

JavaScript Spessore<sup>6</sup> is written for the reader who has read [JavaScript Allongé](#)<sup>7</sup> or has equivalent experience with JavaScript, especially as it pertains to functions, closures, and prototypes. JavaScript Allongé is well-regarded amongst programmers:

“This is a must-read for any developer who wants to know Javascript better... Reg has a way of explaining things in a way that connected the dots for me. This is probably the only programming book I’ve re-read cover to cover a dozen times or more.”—etrinh

“I think it’s one of the best tech books I’ve read since Sedgewick’s Algorithms in C.”—Andrey Sidorov

“Your explanation of closures in JavaScript Allongé is the best I’ve read.”—Emehrkay

“It’s a different approach to JavaScript than you’ll find in most other places and shines a light on some of the more elegant parts of JavaScript the language.”—@jeremymorrell

Even if you know the material, you may want to read JavaScript Allongé to familiarize yourself its approach to functional combinators. You can [read it for free online](#)<sup>8</sup>.

### `allong.es`

[allong.es](#)<sup>9</sup> is a JavaScript library inspired by JavaScript Allongé. It contains many utility functions that are used in JavaScript Spessore’s examples, such as `map`, `variadic` and `tap`. It’s free, and you can even type a lot of the examples from this book into its [try allong.es](#)<sup>10</sup> page and see them work.

---

<sup>6</sup><https://leanpub.com/javascript-spessore>

<sup>7</sup><https://leanpub.com/javascript-allonge>

<sup>8</sup><https://leanpub.com/javascript-allonge/read>

<sup>9</sup><http://allong.es>

<sup>10</sup><http://allong.es/try/>

# Preamble

Dr. Alan Kay on the expression “OOP”<sup>11</sup>



12

Welcome to [JavaScript Spessore][js]. This is a book about “Thinking in Objects,” using JavaScript as the medium for discussion. The format of the book is as follows:

[Caffé Lungo](#) revisits the discussion of objects, prototypes, and simple inheritance from the book [JavaScript Allongé][ja]. This gives up a grounding in what JavaScript provides “out of the box” and how people typically apply it.

[The Object’s The Thing](#) examines JavaScript’s objects and properties more closely, with an emphasis on using defined properties to control an object’s interface.

[Methods](#) expands on the notion of a method, looking at constructing composite methods, method objects, and meta-methods.

---

<sup>11</sup>[images/oop.png](#)

<sup>12</sup>[Normalized Pump Pressure Reading](#) (c) 2009 Nicholas Lundgaard, some rights reserved

[Metaobjects](#) dives into the notion of metaobjects, objects that define the behaviour of other objects. It then examines the distinction between prototypes and other kinds of metaobjects. It then discusses *metaobject protocols*, the design of interfaces for manipulating metaobjects.

# Caffé Lungo



Lungo is Italian for ‘long’, and refers to the coffee beverage made by using an espresso machine to make an espresso (single or double dose or shot) with much more water (generally twice as much), resulting in a *stretched* espresso, a *lungo*. In French it is called *café allongé*.—[Wikipedia](#)<sup>14</sup>

## revisiting javascript allongé

This book is written for the reader who has read [JavaScript Allongé](#)<sup>15</sup> or has equivalent familiarity with functions and basic use of JavaScript’s objects, functions, closures, and the prototype chain.

<sup>13</sup> Transparent Sanremo espresso machine, London Coffee Festival, Truman Brewery, Brick Lane, Hackney, London, UK (c) 2013 Cory Doctorow, some rights reserved

<sup>14</sup><https://en.wikipedia.org/wiki/Lungo>

<sup>15</sup><https://leanpub.com/javascript-allonge>

This chapter is intended as a refresher: Some of the material from JavaScript Allongé is abbreviated and reproduced.

## Plain Old JavaScript Objects

In JavaScript, an object is a map from names to values. Tradition would have us call objects that don't contain any functions "POJOs," Plain Old JavaScript Objects.

The most common syntax for creating an object is called a "literal object expression:"

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

Two objects created this way have differing identities:

```
{ year: 2012, month: 6, day: 14 } === { year: 2012, month: 6, day: 14 }
//=> false
```

Objects use [] to access the values by name, using a string:

```
{ year: 2012, month: 6, day: 14 }['day']
//=> 14
```

Names in literal object expressions needn't be alphanumeric strings. For anything else, enclose the label in quotes:

```
{ 'first name': 'reginald', 'last name': 'lewis' }['first name']
//=> 'reginald'
```

If the name is an alphanumeric string conforming to the same rules as names of variables, there's a simplified syntax for accessing the values:

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

Like all containers, objects can contain any value, including functions or other containers:

```
var Mathematics = {  
    abs: function (a) {  
        return a < 0 ? -a : a  
    }  
};  
  
Mathematics.abs(-5)  
//=> 5
```

Funny we should mention `Mathematics`. JavaScript provides a global environment that contains some existing object that have handy functions you can use. One of them is called `Math`, and it contains functions for `abs`, `max`, `min`, and many others. Since it is always available, you can use it in any environment provided you don't shadow `Math`.

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

## Encapsulating State with Closures

OOP to me means only messaging, local retention and protection and hiding of state-process, and extreme late-binding of all things.—[Alan Kay](#)<sup>16</sup>

We're going to look at encapsulation using JavaScript's functions and objects. We're not going to call it object-oriented programming, mind you, because that would start a long debate. This is just plain encapsulation,<sup>17</sup> with a dash of information-hiding.

### what is hiding of state-process, and why does it matter?

In computer science, information hiding is the principle of segregation of the design decisions in a computer program that are most likely to change, thus protecting other parts of the program from extensive modification if the design decision is changed. The protection involves providing a stable interface which protects the remainder of the program from the implementation (the details that are most likely to change).

Written another way, information hiding is the ability to prevent certain aspects of a class or software component from being accessible to its clients, using either programming language features (like private variables) or an explicit exporting policy.

—[Wikipedia](#)<sup>18</sup>

Consider a [stack](#)<sup>19</sup> data structure. There are three basic operations: Pushing a value onto the top (`push`), popping a value off the top (`pop`), and testing to see whether the stack is empty or not (`isEmpty`). These three operations are the stable interface.

Many stacks have an array for holding the contents of the stack. This is relatively stable. You could substitute a linked list, but in JavaScript, the array is highly efficient. You might need an index, you might not. You could grow and shrink the array, or you could allocate a fixed size and use an index to keep track of how much of the array is in use. The design choices for keeping track of the head of the list are often driven by performance considerations.

If you expose the implementation detail such as whether there is an index, sooner or later some programmer is going to find an advantage in using the index directly. For example, she may need to know the size of a stack. The ideal choice would be to add a `size` function that continues to hide the implementation. But she's in a hurry, so she reads the `index` directly. Now her code is coupled to the existence of an index, so if we wish to change the implementation to grow and shrink the array, we will break her code.

<sup>16</sup>[http://userpage.fu-berlin.de/~ram/pub/pub\\_jf47ht81Ht/doc\\_kay\\_oop\\_en](http://userpage.fu-berlin.de/~ram/pub/pub_jf47ht81Ht/doc_kay_oop_en)

<sup>17</sup>“A language construct that facilitates the bundling of data with the methods (or other functions) operating on that data.”—[Wikipedia](#)

<sup>18</sup>[https://en.wikipedia.org/wiki/Information\\_hiding](https://en.wikipedia.org/wiki/Information_hiding)

<sup>19</sup>[https://en.wikipedia.org/wiki/Stack\\_](https://en.wikipedia.org/wiki/Stack_)

The way to avoid this is to hide the array and index from other code and only expose the operations we have deemed stable. If and when someone needs to know the size of the stack, we'll add a `size` function and expose it as well.

Hiding information (or “state”) is the design principle that allows us to limit the coupling between components of software.

## how do we hide state using javascript?

We've been introduced to JavaScript's objects, and it's fairly easy to see that objects can be used to model what other programming languages call (variously) records, structs, frames, or what-have-you. And given that their elements are mutable, they can clearly model state.

Given an object that holds our state (an array and an index<sup>20</sup>), we can easily implement our three operations as functions. Bundling the functions with the state does not require any special “magic” features. JavaScript objects can have elements of any type, including functions:

```
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;
})();

stack.isEmpty()
//=> true
```

---

<sup>20</sup>Yes, there's another way to track the size of the array, but we don't need it to demonstrate encapsulation and hiding of state.

```
stack.push('hello')
  //=> 'hello'
stack.push('JavaScript')
  //=> 'JavaScript'
stack.isEmpty()
  //=> false
stack.pop()
  //=> 'JavaScript'
stack.pop()
  //=> 'hello'
stack.isEmpty()
  //=> true
```

## method-ology

In this text, we lurch from talking about “functions that belong to an object” to “methods.” Other languages may separate methods from functions very strictly, but in JavaScript every method is a function but not all functions are methods.

The view taken in this book is that a function is a method of an object if it belongs to that object and interacts with that object in some way. So the functions implementing the operations on the stack are all absolutely methods of the stack.

But these two wouldn’t be methods. Although they “belong” to an object, they don’t interact with it:

```
{
  min: function (x, y) {
    if (x < y) {
      return x
    }
    else {
      return y
    }
  }
  max: function (x, y) {
    if (x > y) {
      return x
    }
    else {
      return y
    }
  }
}
```

## hiding state

Our stack does bundle functions with data, but it doesn't hide its state. "Foreign" code could interfere with its array or index. So how do we hide these? We already have a closure, let's use it:

```
var stack = (function () {
    var array = [],
        index = -1;

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

stack.isEmpty()
//=> true
stack.push('hello')
//=> 'hello'
stack.push('JavaScript')
//=> 'JavaScript'
stack.isEmpty()
//=> false
stack.pop()
//=> 'JavaScript'
stack.pop()
//=> 'hello'
stack.isEmpty()
//=> true
```

We don't want to repeat this code every time we want a stack, so let's make ourselves a "stack maker." The temptation is to wrap what we have above in a function:

```
var StackMaker = function () {
    return (function () {
        var array = [],
            index = -1;

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

But there's an easier way :-)

```
var StackMaker = function () {
    var array = [],
        index = -1;

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

```
        }
    }
};

stack = StackMaker()
```

Now we can make stacks freely, and we've hidden their internal data elements. We have methods and encapsulation, and we've built them out of JavaScript's fundamental functions and objects. A little further on, we'll look at JavaScript's support for class-oriented programming and some of the idioms that functions bring to the party.

## is encapsulation “object-oriented?”

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

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

---

<sup>a</sup><http://www.c2.com/cgi/wiki?CompositionInsteadOfInheritance>

# Composition and Extension

## composition

A deeply fundamental practice is to build components out of smaller components. The choice of how to divide a component into smaller components is called *factoring*, after the operation in number theory<sup>[21](#)</sup>.

The simplest and easiest way to build components out of smaller components in JavaScript is also the most obvious: Each component is a value, and the components can be put together into a single object or encapsulated with a closure.

Here's an abstract "model" that supports undo and redo composed from a pair of stacks (see [Encapsulating State](#)) and a Plain Old JavaScript Object:

```
// helper function
//
// For production use, consider what to do about
// deep copies and own keys
var shallowCopy = function (source) {
  var dest = {},
    key;

  for (key in source) {
    dest[key] = source[key]
  }
  return dest
};

// our model maker
var ModelMaker = function (initialAttributes) {
  var attributes = shallowCopy(initialAttributes || {}),
    undoStack = StackMaker(),
    redoStack = StackMaker(),
    obj = {
      set: function (attrsToSet) {
        var key;

        undoStack.push(shallowCopy(attributes));
        if (!redoStack.isEmpty()) {
          redoStack = StackMaker()
        }
        attributes = attrsToSet;
      }
    };
  return obj;
};
```

---

<sup>21</sup>And when you take an already factored component and rearrange things so that it is factored into a different set of subcomponents without altering its behaviour, you are *refactoring*.

```

        }
        for (key in (attrsToSet || {})) {
            attributes[key] = attrsToSet[key]
        }
        return obj
    },
    undo: function () {
        if (!undoStack.isEmpty()) {
            redoStack.push(shallowCopy(attributes));
            attributes = undoStack.pop()
        }
        return obj
    },
    redo: function () {
        if (!redoStack.isEmpty()) {
            undoStack.push(shallowCopy(attributes));
            attributes = redoStack.pop()
        }
        return obj
    },
    get: function (key) {
        return attributes(key)
    },
    has: function (key) {
        return attributes.hasOwnProperty(key)
    },
    attributes: function {
        shallowCopy(attributes)
    }
};
return obj
;

```

The techniques used for encapsulation work well with composition. In this case, we have a “model” that hides its attribute store as well as its implementation that is composed of an undo stack and redo stack.

## extension

Another practice that many people consider fundamental is to *extend* an implementation. Meaning, they wish to define a new data structure in terms of adding new operations and semantics to an existing data structure.

Consider a queue<sup>22</sup>:

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

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

Now we wish to create a deque<sup>23</sup> by adding pullTail and pushHead operations to our queue.<sup>24</sup> Unfortunately, encapsulation prevents us from adding operations that interact with the hidden data structures.

This isn't really surprising: The entire point of encapsulation is to create an opaque data structure that can only be manipulated through its public interface. The design goals of encapsulation and extension are always going to exist in tension.

Let's "de-encapsulate" our queue:

---

<sup>22</sup>[http://duckduckgo.com/Queue\\_](http://duckduckgo.com/Queue_)

<sup>23</sup>[https://en.wikipedia.org/wiki/Double-ended\\_queue](https://en.wikipedia.org/wiki/Double-ended_queue)

<sup>24</sup>Before you start wondering whether a deque is-a queue, we said nothing about types and classes. This relationship is called was-a, or "implemented in terms of a."

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

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

Now we can extend a queue into a deque:

```
var DequeMaker = function () {
  var deque = QueueMaker(),
    INCREMENT = 4;

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

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

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

    if (deque.head === 0) {
        for (i = deque.tail; i <= deque.head; i++) {
            deque.array[i + INCREMENT] = deque.array[i];
        }
        deque.tail += INCREMENT
        deque.head += INCREMENT
    }
    return deque.array[deque.head -= 1] = value
}
});
```

Presto, we have reuse through extension, at the cost of encapsulation.



Encapsulation and Extension exist in a natural state of tension. A program with elaborate encapsulation resists breakage but can also be difficult to refactor in other ways. Be mindful of when it's best to Compose and when it's best to Extend.

## This and That

Let's take another look at [extensible objects](#). Here's a Queue:

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

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

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

Let's make a copy of our queue using the extend recipe:

```
copyOfQueue = extend({}, queue);

queue !== copyOfQueue
//=> true
```

Wait a second. We know that array values are references. So it probably copied a reference to the original array. Let's make a copy of the array as well:

```
copyOfQueue.array = [];
for (var i = 0; i < 2; ++i) {
  copyOfQueue.array[i] = queue.array[i]
}
```

Now let's pull the head off the original:

```
queue.pullHead()
//=> 'Hello'
```

If we've copied everything properly, we should get the exact same result when we pull the head off the copy:

```
copyOfQueue.pullHead()
//=> 'JavaScript'
```

What!? Even though we carefully made a copy of the array to prevent aliasing, it seems that our two queues behave like aliases of each other. The problem is that while we've carefully copied our array and other elements over, *the closures all share the same environment*, and therefore the functions in `copyOfQueue` all operate on the first queue's private data, not on the copies.

This is a general issue with closures. Closures couple functions to environments, and that makes them very elegant in the small, and very handy for making opaque data structures. Alas, their strength in the small is their weakness in the large. When you're trying to make reusable components, this coupling is sometimes a hindrance.

Let's take an impossibly optimistic flight of fancy:

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

```

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

queueWithAmnesia = AmnesiacQueueMaker();
queueWithAmnesia.pushTail(queueWithAmnesia, 'Hello');
queueWithAmnesia.pushTail(queueWithAmnesia, 'JavaScript')

```

The `AmnesiacQueueMaker` makes queues with amnesia: They don't know who they are, so every time we invoke one of their functions, we have to tell them who they are. You can work out the implications for copying queues as a thought experiment: We don't have to worry about environments, because every function operates on the queue you pass in.

The killer drawback, of course, is making sure we are always passing the correct queue in every time we invoke a function. What to do?

## what's all this?

Any time we must do the same repetitive thing over and over and over again, we industrial humans try to build a machine to do it for us. JavaScript is one such machine:

```

BanksQueueMaker = 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
}
}
};

banksQueue = BanksQueueMaker();
banksQueue.pushTail('Hello');
banksQueue.pushTail('JavaScript')

```

Every time you invoke a function that is a member of an object, JavaScript binds that object to the name `this` in the environment of the function just as if it was an argument.<sup>25</sup> Now we can easily make copies:

```

copyOfQueue = extend({}, banksQueue)
copyOfQueue.array = []
for (var i = 0; i < 2; ++i) {
  copyOfQueue.array[i] = banksQueue.array[i]
}

banksQueue.pullHead()
//=> 'Hello'

copyOfQueue.pullHead()
//=> 'Hello'

```

Presto, we now have a way to copy arrays. By getting rid of the closure and taking advantage of `this`, we have functions that are more easily portable between objects, and the code is simpler as well.

There is more to `this` than we've discussed here. We'll explore things in more detail later, in [What Context Applies When We Call a Function?](#)



Closures tightly couple functions to the environments where they are created limiting their flexibility. Using `this` alleviates the coupling. Copying objects is but one example of where that flexibility is needed.

---

<sup>25</sup>JavaScript also does other things with `this` as well, but this is all we care about right now.

## What Context Applies When We Call a Function?

In [This and That](#), we learned that when a function is called as an object method, the name `this` is bound in its environment to the object acting as a “receiver.” For example:

```
var someObject = {
  returnMyThis: function () {
    return this;
  }
};

someObject.returnMyThis() === someObject
//=> true
```

We’ve constructed a method that returns whatever value is bound to `this` when it is called. It returns the object when called, just as described.

### it's all about the way the function is called

JavaScript programmers talk about functions having a “context” when being called. `this` is bound to the context.<sup>26</sup> The important thing to understand is that the context for a function being called is set by the way the function is called, not the function itself.

This is an important distinction. Consider closures: As we discussed in [Closures and Scope](#), a function’s free variables are resolved by looking them up in their enclosing functions’ environments. You can always determine the functions that define free variables by examining the source code of a JavaScript program, which is why this scheme is known as [Lexical Scope](#)<sup>27</sup>.

A function’s context cannot be determined by examining the source code of a JavaScript program. Let’s look at our example again:

```
var someObject = {
  someFunction: function () {
    return this;
  }
};

someObject.someFunction() === someObject
//=> true
```

What is the context of the function `someObject.someFunction`? Don’t say `someObject`! Watch this:

---

<sup>26</sup>Too bad the language binds the context to the name `this` instead of the name `context`!

<sup>27</sup>[https://en.wikipedia.org/wiki/Scope\\_\(computer\\_science\)#Lexical\\_scoping](https://en.wikipedia.org/wiki/Scope_(computer_science)#Lexical_scoping)

```
var someFunction = someObject.someFunction;

someFunction === someObject.someFunction
//=> true

someFunction() === someObject
//=> false
```

It gets weirder:

```
var anotherObject = {
  someFunction: someObject.someFunction
}

anotherObject.someFunction === someObject.someFunction
//=> true

anotherObject.someFunction() === anotherObject
//=> true

anotherObject.someFunction() === someObject
//=> false
```

So it amounts to this: The exact same function can be called in two different ways, and you end up with two different contexts. If you call it using `someObject.someFunction()` syntax, the context is set to the receiver. If you call it using any other expression for resolving the function's value (such as `someFunction()`), you get something else. Let's investigate:

```
(someObject.someFunction)() == someObject
//=> true

someObject['someFunction']() === someObject
//=> true

var name = 'someFunction';

someObject[name]() === someObject
//=> true
```

Interesting!

```
var baz;

(baz = someObject.someFunction)() === this
//=> true
```

How about:

```
var arr = [ someObject.someFunction ];

arr[0]() == arr
//=> true
```

It seems that whether you use `a.b()` or `a['b']()` or `a[n]()` or `(a.b)()`, you get context `a`.

```
var returnThis = function () { return this };

var aThirdObject = {
  someFunction: function () {
    return returnThis()
  }
}

returnThis() === this
//=> true

aThirdObject.someFunction() === this
//=> true
```

And if you don't use `a.b()` or `a['b']()` or `a[n]()` or `(a.b)()`, you get the global environment for a context, not the context of whatever function is doing the calling. To simplify things, when you call a function with `.` or `[]` access, you get an object as context, otherwise you get the global environment.

## setting your own context

There are actually two other ways to set the context of a function. And once again, both are determined by the caller. At the very end of [objects everywhere?](#), we'll see that everything in JavaScript behaves like an object, including functions. We'll learn that functions have methods themselves, and one of them is `call`.

Here's `call` in action:

```

returnThis() === aThirdObject
//=> false

returnThis.call(aThirdObject) === aThirdObject
//=> true

anotherObject.someFunction.call(someObject) === someObject
//=> true

```

When You call a function with `call`, you set the context by passing it in as the first parameter. Other arguments are passed to the function in the normal manner. Much hilarity can result from `call` shenanigans like this:

```

var a = [1,2,3],
    b = [4,5,6];

a.concat([2,1])
//=> [1,2,3,2,1]

a.concat.call(b,[2,1])
//=> [4,5,6,2,1]

```

But now we thoroughly understand what `a.b()` really means: It's synonymous with `a.b.call(a)`. Whereas in a browser, `c()` is synonymous with `c.call(window)`.

## apply, arguments, and contextualization

JavaScript has another automagic binding in every function's environment. `arguments` is a special object that behaves a little like an array.<sup>28</sup>

For example:

```

var third = function () {
  return arguments[2]
}

third(77, 76, 75, 74, 73)
//=> 75

```

Hold that thought for a moment. JavaScript also provides a fourth way to set the context for a function. `apply` is a method implemented by every function that takes a context as its first argument, and it takes an array or array-like thing of arguments as its second argument. That's a mouthful, let's look at an example:

---

<sup>28</sup>Just enough to be frustrating, to be perfectly candid!

```
third.call(this, 1,2,3,4,5)
//=> 3

third.apply(this, [1,2,3,4,5])
//=> 3
```

Now let's put the two together. Here's another travesty:

```
var a = [1,2,3],
    accrete = a.concat;

accrete([4,5])
//=> Gobbledygook!
```

We get the result of concatenating [4,5] onto an array containing the global environment. Not what we want! Behold:

```
var contextualize = function (fn, context) {
  return function () {
    return fn.apply(context, arguments);
  }
}

accrete = contextualize(a.concat, a);
accrete([4,5]);
//=> [ 1, 2, 3, 4, 5 ]
```

Our contextualize function returns a new function that calls a function with a fixed context. It can be used to fix some of the unexpected results we had above. Consider:

```
var aFourthObject = {},
    returnThis = function () { return this; };

aFourthObject.uncontextualized = returnThis;
aFourthObject.contextualized = contextualize(returnThis, aFourthObject);

aFourthObject.uncontextualized() === aFourthObject
//=> true
aFourthObject.contextualized() === aFourthObject
//=> true
```

Both are true because we are accessing them with aFourthObject. Now we write:

```
var uncontextualized = aFourthObject.uncontextualized,  
    contextualized = aFourthObject.contextualized;  
  
uncontextualized() === aFourthObject;  
//=> false  
contextualized() === aFourthObject  
//=> true
```

When we call these functions without using `aFourthObject.`, only the contextualized version maintains the context of `aFourthObject`.

We'll return to contextualizing methods later, in [Binding](#). But before we dive too deeply into special handling for methods, we need to spend a little more time looking at how functions and methods work.

## Extending Objects

It's very common to want to “extend” a simple object by adding properties to it:

```
var inventory = {
  apples: 12,
  oranges: 12
};

inventory.bananas = 54;
inventory.pears = 24;
```

It's also common to want to add a [shallow copy<sup>29</sup>](#) of the properties of one object to another:

```
for (var fruit in shipment) {
  inventory[fruit] = shipment[fruit]
}
```

Both needs can be met with this recipe for extend:

```
var extend = variadic( function (consumer, providers) {
  var key,
    i,
    provider;

  for (i = 0; i < providers.length; ++i) {
    provider = providers[i];
    for (key in provider) {
      if (provider.hasOwnProperty(key)) {
        consumer[key] = provider[key]
      }
    }
  }
  return consumer
});
```

You can copy an object by extending an empty object:

---

<sup>29</sup>[https://en.wikipedia.org/wiki/Object\\_copy#Shallow\\_copy](https://en.wikipedia.org/wiki/Object_copy#Shallow_copy)

```
extend({}, {
  apples: 12,
  oranges: 12
})
//=> { apples: 12, oranges: 12 }
```

You can extend one object with another:

```
var inventory = {
  apples: 12,
  oranges: 12
};

var shipment = {
  bananas: 54,
  pears: 24
}

extend(inventory, shipment)
//=> { apples: 12,
//      oranges: 12,
//      bananas: 54,
//      pears: 24 }
```

And when we discuss prototypes, we will use extend to turn this:

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

Queue.prototype.pushTail = function (value) {
  // ...
};

Queue.prototype.pullHead = function () {
  // ...
};

Queue.prototype.isEmpty = function () {
  // ...
}
```

Into this:

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

extend(Queue.prototype, {
  pushTail: function (value) {
    // ...
  },
  pullHead: function () {
    // ...
  },
  isEmpty: function () {
    // ...
  }
});
```

As we build more complex objects with more complex structures, we will revisit “extend” and improve on it.

## 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 interesting:

```
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.<sup>30</sup>

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

---

<sup>30</sup>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](#).

But let's get back to prototypes:

```
function C () {}  
  
C.prototype.bodyOfWater = 'sea'  
C.prototype  
//=> { bodyOfWater: 'sea' }  
  
var c = new C();  
  
Object.getPrototypeOf(c)  
//=> { bodyOfWater: 'sea' }  
  
C.prototype.isPrototypeOf(c)  
//=> true
```

`getPrototypeOf` and `isPrototypeOf` are very useful. Let's see how:

```
var oldProto = C.prototype;
C.prototype = { bodyOfWater: 'ocean' };

Object.getPrototypeOf(c)
//=> { bodyOfWater: 'sea' }

C.prototype.isPrototypeOf(c)
//=> false

oldProto.isPrototypeOf(c)
//=> true
```

Changing a the object bound to a function's prototype property doesn't change the prototype for any objects that have already been created with new.

## create

You can create objects and assign them prototypes *without* new. This object doesn't have a prototype:

```
var obj = Object.create(null);
Object.getPrototypeOf(obj)
//=> null
```

This object has the same prototype as an object we create using literal object syntax, or an object we create using new Object()

```
obj = Object.create(Object.prototype);
Object.getPrototypeOf(obj)
//=> {}

Object.getPrototypeOf(obj) === Object.getPrototypeOf({})
//=> true
Object.getPrototypeOf(obj) === Object.getPrototypeOf(new Object())
//=> true
```

This object has a prototype of our choosing:

```
var ourPrototype = { bodyOfWater: 'ocean' };  
obj = Object.create(ourPrototype);  
  
ourPrototype.isPrototypeOf(obj)  
//=> true
```

We can create an object with any prototype we like, without writing a constructor or using `new`.

## 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<sup>31</sup>](#) 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?

---

<sup>31</sup><http://backbonejs.org>

## 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<sup>32</sup>](#), 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 `.bind33` 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:

---

<sup>32</sup><http://underscorejs.org>

<sup>33</sup>[https://developer.mozilla.org/en-US/docs/JavaScript/Reference/Global\\_Objects/Function/bind](https://developer.mozilla.org/en-US/docs/JavaScript/Reference/Global_Objects/Function/bind)

```
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.

## 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 Objects 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.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.call(this)`.

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 `Dequeue.prototype`, like `extend(Dequeue.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 = new Queue();
```

And here's another:

```
var queueProxy = Object.create(Queue.prototype);
```

The key here is that the prototype of a Dequeue needs to be an object that has its prototype set to Queue.prototype. That way, we can add methods to Dequeue's prototype without modifying Queue's prototype. Let's double-check:

```
Queue.prototype.isPrototypeOf(queueProxy)  
//=> true
```

Let's insert queueProxy into our code:

```
var Dequeue = function () {  
    Queue.call(this)  
};  
  
Dequeue.INCREMENT = 4;  
  
Dequeue.prototype = 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!

## extracting the boilerplate

Let's turn our mechanism into a function:

```
var child = function (parent, child) {
  var proxy = Object.create(parent.prototype);
  child.prototype = proxy;
  return child;
}
```

And use it in Dequeue:

```
var Dequeue = child(Queue, function () {
  Queue.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 inheritance ontologies requires hundreds of lines of inscrutable code.

## Summary



- State can be encapsulated/hidden with closures.
- Encapsulations can be aggregated with composition.
- Encapsulation resists extension.
- The automagic binding `this` facilitates sharing of functions.
- Functions can be named and declared with a name.
- 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.

# The Object's The Thing



34

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.-[Smalltalk on Wikipedia](#)<sup>35</sup>

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.

---

<sup>34</sup>[Londinium 1 Lever Espresso Machine](#) (c) 2013 Alejandro Erickson, some rights reserved

<sup>35</sup><https://en.wikipedia.org/wiki/Smalltalk>

## 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.<sup>36</sup>

Popular languages don’t usually discuss messages. Instead, they speak of “invoking methods.” Invoking a method has a much more imperative implication than “sending a message.” It implies that the entity doing the invoking is causing something to happen, even if the precise implementation is the receiver’s responsibility. Most popular languages are synchronous: The code that invokes the method waits for a response.

## methods

A method is a kind of recipe for handling a message. In JavaScript, methods are functions. In other languages, methods are a separate kind of thing than functions. In many languages, the namespace for methods is distinct from the namespace for instance variables and other internal references. In JavaScript, methods and internal state are stored together as properties by default.

If we wish to organize methods separately from internal state, we must impose our own structure.

<sup>36</sup>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 in JavaScript:

```
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<sup>37</sup> 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: **An object is an entity that uses handlers to respond to messages. It maintains internal state, and its handlers are responsible for querying and/or updating its state.**

---

<sup>37</sup><https://en.wikipedia.org/wiki/Simula> “The Simula Programming Language”

## Immutable Properties

Sometimes we want to share objects by reference for performance and space reasons, but we don't want them to be mutable. One motivation is when we want many objects to be able to share a common entity without worrying that one of them may inadvertently change the common entity.

JavaScript provides a way to make properties immutable:

```
"use strict";

var rentAmount = {};

Object.defineProperty(rentAmount, 'dollars', {
  enumerable: true,
  writable: false,
  value: 420
});

Object.defineProperty(rentAmount, 'cents', {
  enumerable: true,
  writable: false,
  value: 0
});

rentAmount.dollars
//=> 420

rentAmount.dollars = 600;
//=> 600

rentAmount.dollars
//=> 420
```

`Object.defineProperty` is a general-purpose method for providing fine-grained control over the properties of any object. When we make a property `enumerable`, it shows up whenever we list the object's properties or iterate over them. When we make it `writable`, assignments to the property change its value. If the property isn't `writable`, assignments are ignored.

When we want to define multiple properties, we can also write:

```
var rentAmount = {};  
  
Object.defineProperties(rentAmount, {  
  dollars: {  
    enumerable: true,  
    writable: false,  
    value: 420  
  },  
  cents: {  
    enumerable: true,  
    writable: false,  
    value: 0  
  }  
});  
  
rentAmount.dollars  
//=> 420  
  
rentAmount.dollars = 600;  
//=> 600  
  
rentAmount.dollars  
//=> 420
```

While we can't make the entire object immutable, we can define the properties we want to be immutable. Naturally, we can generalize this:

```
function immutable (propertiesAndValues) {  
  return tap({}, function (object) {  
    for (var key in propertiesAndValues) {  
      if (propertiesAndValues.hasOwnProperty(key)) {  
        Object.defineProperty(object, key, {  
          enumerable: true,  
          writable: false,  
          value: propertiesAndValues[key]  
        });  
      }  
    }  
  });  
}  
  
var rentAmount = immutable({
```

```
    dollars: 420,  
    cents: 0  
});  
  
rentAmount.dollars  
//=> 420  
  
rentAmount.dollars = 600;  
//=> 600  
  
rentAmount.dollars  
//=> 420
```

## considerations

As a means for protecting our data structures from inadvertent modification, this “silent failure” isn’t great, but it at least *localizes* the failure mode to the objects our code is trying to change, and not to objects that may be sharing an object we’re trying to change.

But it does force us to test property assignments. Whenever you write some code like this:

```
rentCheque.amount.dollars = 600;
```

You ought to write a test case that checks to see whether the cheque’s dollar figure really changed. And now that you know that immutable properties are a JavaScript feature, you really need to check assignments whether you’re using them or not. Who knows what changes might be made to your code in the future? Testing assignments ensures that you will catch any regressions that might be caused in the future.

## Copy On Write Semantics

*Coming Soon*

## Accessors

The Java and Ruby folks are very comfortable with a general practice of not allowing objects to modify each other's properties. They prefer to write *getters and setters*, functions that do the getting and setting. If we followed this practice, we might write:

```
var mutableAmount = (function () {
  var _dollars = 0;
  var _cents = 0;
  return immutable({
    setDollars: function (amount) {
      return (_dollars = amount);
    },
    getDollars: function () {
      return _dollars;
    },
    setCents: function (amount) {
      return (_cents = amount);
    },
    getCents: function () {
      return _cents;
    }
  });
})();

mutableAmount.getDollars()
//=> 0

mutableAmount.setDollars(420);

mutableAmount.getDollars()
//=> 420
```

We've put functions in the object for getting and setting values, and we've hidden the values themselves in a *closure*, the environment of an [Immediately Invoked Function Expression<sup>38</sup>](#) ("IIFE").

Of course, this amount can still be mutated, but we are now mediating access with functions. We could, for example, enforce certain validity rules:

---

<sup>38</sup>[https://en.wikipedia.org/wiki/Immediately-invoked\\_function\\_expression](https://en.wikipedia.org/wiki/Immediately-invoked_function_expression)

```

var mutableAmount = (function () {
  var _dollars = 0;
  var _cents = 0;
  return immutable({
    setDollars: function (amount) {
      if (amount >= 0 && amount === Math.floor(amount))
        return (_dollars = amount);
    },
    getDollars: function () {
      return _dollars;
    },
    setCents: function (amount) {
      if (amount >= 0 && amount < 100 && amount === Math.floor(amount))
        return (_cents = amount);
    },
    getCents: function () {
      return _cents;
    }
  });
})();

mutableAmount.setDollars(-5)
//=> undefined

mutableAmount.getDollars()
//=> 0

```

Immutability is easy, just leave out the “getters:”

```

var rentAmount = (function () {
  var _dollars = 420;
  var _cents = 0;
  return immutable({
    getDollars: function () {
      return _dollars;
    },
    getCents: function () {
      return _cents;
    }
  });
})();

```

```
mutableAmount.setDollars(-5)
//=> undefined
```

```
mutableAmount.getDollars()
//=> 0
```

## using accessors for properties

Languages like Ruby allow you to write code that looks like you're doing direct access of properties but still mediate access with functions. JavaScript allows this as well. Let's revisit `Object.defineProperties`:

```
var mediatedAmount = (function () {
  var _dollars = 0;
  var _cents = 0;
  var amount = {};
  Object.defineProperties(amount, {
    dollars: {
      enumerable: true,
      set: function (amount) {
        if (amount >= 0 && amount === Math.floor(amount))
          return (_dollars = amount);
      },
      get: function () {
        return _dollars;
      }
    },
    cents: {
      enumerable: true,
      set: function (amount) {
        if (amount >= 0 && amount < 100 && amount === Math.floor(amount))
          return (_cents = amount);
      },
      get: function () {
        return _cents;
      }
    }
  });
  return amount;
})();
//=>
{ dollars: [Getter/Setter],
```

```

cents: [Getter/Setter] }

mediatedAmount.dollars = 600;

mediatedAmount.dollars
//=> 600

mediatedAmount.cents = 33.5

mediatedAmount.cents
//=> 0

```

We can leave out the setters if we wish:

```

var mediatedImmutableAmount = (function () {
  var _dollars = 420;
  var _cents = 0;
  var amount = {};
  Object.defineProperties(amount, {
    dollars: {
      enumerable: true,
      get: function () {
        return _dollars;
      }
    },
    cents: {
      enumerable: true,
      get: function () {
        return _cents;
      }
    }
  });
  return amount;
})();

mediatedImmutableAmount.dollars = 600;

mediatedImmutableAmount.dollars
//=> 420

```

Once again, the failure is silent. Of course, we can change that:

```
var noisyAmount = (function () {
  var _dollars = 0;
  var _cents = 0;
  var amount = {};
  Object.defineProperties(amount, {
    dollars: {
      enumerable: true,
      set: function (amount) {
        if (amount !== _dollars)
          throw new Error("You can't change that!");
      },
      get: function () {
        return _dollars;
      }
    },
    cents: {
      enumerable: true,
      set: function (amount) {
        if (amount !== _cents)
          throw new Error("You can't change that!");
      },
      get: function () {
        return _cents;
      }
    }
  });
  return amount;
})();

noisyAmount.dollars = 500
//=> Error: You can't change that!
```

# Hiding Object Properties

Many “OO” programming languages have the notion of private instance variables, properties that cannot be accessed by other entities. JavaScript has no such notion, we have to use specific techniques to create the illusion of private state for objects.

## enumerability

In JavaScript, there is only one kind of “privacy” for properties. But it’s not what you expect. When an object has properties, you can access them with the dot notation, like this:

```
var dictionary = {  
    abstraction: "an abstract or general idea or term",  
    encapsulate: "to place in or as if in a capsule",  
    object: "anything that is visible or tangible and is relatively stable in form"  
};  
  
dictionary.encapsulate  
//=> 'to place in or as if in a capsule'
```

You can also access properties indirectly through the use of [] notation and the value of an expression:

```
dictionary[abstraction]  
//=> ReferenceError: abstraction is not defined
```

Whoops, the value of an *expression*: The expression `abstraction` looks up the value associated with the variable “abstraction.” Alas, such a variable hasn’t been defined in this code, so that’s an error. This works, because ‘`abstraction`’ is an expression that evaluates to the string we want:

```
dictionary['abstraction']  
//=> 'an abstract or general idea or term'
```

One kind of privacy concerns who has access to properties. In JavaScript, all code has access to all properties of every object. There is no way to create a property of an object such that some functions can access it and others cannot.

So what kind of privacy does JavaScript provide? In order to access a property, you have to know its name. If you don’t know the names of an object’s properties, you can access the names in several ways. Here’s one:

```
Object.keys(dictionary)
//=>
[ 'abstraction',
  'encapsulate',
  'object' ]
```

This is called *enumerating* an object's properties. Not only are they "public" in the sense that any code that knows the property's names can access it, but also, any code at all can enumerate them. You can do neat things with enumerable properties, such as:

```
var descriptor = map(Object.keys(dictionary), function (key) {
  return key + ': "' + dictionary[key] + '"';
}).join('; ');

descriptor
//=>
'abstraction: "an abstract or general idea or term"; encapsulate: "to place i\
n or as if in a
capsule"; object: "anything that is visible or tangible and is relatively sta\
ble in form'"
```

So, our three properties are *accessible* and also *enumerable*. Are there any properties that are accessible, but not enumerable? There sure can be. You recall that we can define properties using `Object.defineProperty`. One of the options is called, appropriately enough, `enumerable`.

Let's define a getter that isn't enumerable:

```
Object.defineProperty(dictionary, 'length', {
  enumerable: false,
  get: function () {
    return Object.keys(this).length
  }
});

dictionary.length
//=> 3
```

Notice that `length` obviously isn't included in `Object.keys`, otherwise our little getter would return 4, not 3. And it doesn't affect our little descriptor expression, let's evaluate it again:

```
map(Object.keys(dictionary), function (key) {
  return key + ': "' + dictionary[key] + '"';
}).join('; ')
//=>
  'abstraction: "an abstract or general idea or term"; encapsulate: "to place i\
n or as if in a
  capsule"; object: "anything that is visible or tangible and is relatively sta\
ble in form"'
```

Non-enumerable properties don't have to be getters:

```
Object.defineProperty(dictionary, 'secret', {
  enumerable: false,
  writable: true,
  value: "kept from the knowledge of any but the initiated or privileged"
});

dictionary.secret
//=> 'kept from the knowledge of any but the initiated or privileged'

dictionary.length
//=> 3
```

`secret` is indeed a secret. It's fully accessible if you know it's there, but it's not enumerable, so it doesn't show up in `Object.keys`.

One way to “hide” properties in JavaScript is to define them as properties with `enumerable: false`.

## closures

We saw earlier that it is possible to fake private instance variables by hiding references in a closure, e.g.

```
function immutable (propertiesAndValues) {
  return tap({}, function (object) {
    for (var key in propertiesAndValues) {
      if (propertiesAndValues.hasOwnProperty(key)) {
        Object.defineProperty(object, key, {
          enumerable: true,
          writable: false,
          value: propertiesAndValues[key]
        });
      }
    }
  });
}
```

```
        }
    }
});
}

var rentAmount = (function () {
    var _dollars = 420;
    var _cents = 0;
    return immutable({
        dollars: function () {
            return _dollars;
        },
        cents: function () {
            return _cents;
        }
    });
})();
```

`_dollars` and `_cents` aren't properties of the `rentAmount` object at all, they're variables within the environment of an IIFE. The functions associated with `dollars` and `cents` are within its scope, so they have access to its variables.

This has some obvious space and performance implications. There's also the general problem that an environment like a closure is its own thing in JavaScript that exists outside of the language's usual features. For example, you can iterate over the enumerable properties of an object, but you can't iterate over the variables being used inside of an object's functions. Another example: you can access a property indirectly with `[expression]`. You can't access a closure's variable indirectly without some clever finagling using `eval`.

Finally, there's another very real problem: Each and every function belonging to each and every object must be a distinct entity in JavaScript's memory. Let's make another amount using the same pattern as above:

```
var rentAmount2 = (function () {
    var _dollars = 600;
    var _cents = 0;
    return immutable({
        dollars: function () {
            return _dollars;
        },
        cents: function () {
            return _cents;
        }
    });
})();
```

```
});  
})();
```

We now have defined four functions: Two getters for `rentAmount`, and two for `rentAmount2`. Although the two `dollars` functions have identical code, they're completely different entities to JavaScript because each has a different enclosing environment. The same thing goes for the two `cents` functions. In the end, we're going to create an enclosing environment and two new functions every time we create an amount using this pattern.

## naming conventions

Let's compare this to a different approach. We'll write almost the identical code, but we'll rely on a naming convention to hide our values in plain sight:

```
function dollars () {  
    return this._dollars;  
}  
  
function cents () {  
    return this._cents;  
}  
  
var rentAmount = immutable({  
    dollars: dollars,  
    cents: cents  
});  
rentAmount._dollars = 420;  
rentAmount._cents = 0;
```

Our convention is that other entities should not modify any property that has a name beginning with `_`. There's no enforcement, it's just a practice. Other entities can use getters and setters. We've created two functions, and we're using `this` to make sure they refer to the object's environment. With this pattern, we need two functions and one object to represent an amount.

One problem with this approach, of course, is that everything we're using is enumerable:

```
Object.keys(rentAmount)
//=>
[ 'dollars',
  'cents',
  '_dollars',
  '_cents' ]
```

We'd better fix that:

```
Object.defineProperties(rentAmount, {
  _dollars: {
    enumerable: false,
    writable: true
  },
  _cents: {
    enumerable: false,
    writable: true
  }
});
```

Let's create another amount:

```
var raisedAmount = immutable({
  dollars: dollars,
  cents: cents
});

Object.defineProperties(raisedAmount, {
  _dollars: {
    enumerable: false,
    writable: true
  },
  _cents: {
    enumerable: false,
    writable: true
  }
});

raisedAmount._dollars = 600;
raisedAmount._cents = 0;
```

We create another object, but we can reuse the existing functions. Let's make sure:

```
rentAmount.dollars()  
//=> 420  
  
raisedAmount.dollars()  
//=> 600
```

What does this accomplish? Well, it “hides” the raw properties by making them enumerable, then provides access (if any) to other objects through functions that can be shared amongst multiple objects.

As we saw earlier, this allows us to choose whether to expose setters as well as getters, it allows us to validate inputs, or even to have non-enumerable properties that are used by an object’s functions to hold state.

The naming convention is useful, and of course you can use whatever convention you like. My personal preference for a very long time was to preface private names with `my`, such as `myDollars`. Underscores work just as well, and that’s what we’ll use in this book.

## summary

JavaScript does not have a way to enforce restrictions on accessing an object’s properties: Any code that knows the name of a property can access the value, setter, or getter that has been defined for the object.

Private data can be faked with closures, at a cost in memory.

JavaScript does allow properties to be non-enumerable. In combination with a naming convention and/or setters and getters, a reasonable compromise can be struck between fully private instance variables and completely open access.

## Object-1s and Object-2s

In the discussion of hiding properties, we saw the example of a dictionary object. Here it is with its *domain properties*, the properties that correspond to the state of the object:

```
var dictionary = {
  abstraction: "an abstract or general idea or term",
  encapsulate: "to place in or as if in a capsule",
  object: "anything that is visible or tangible and is relatively stable in form"
};
```

By default, JavaScript permits us to add “behaviour” to the dictionary by binding functions to properties. We’ve already seen a better solution, but let’s back up for a moment and write:

```
dictionary.describe = function () {
  return map(['abstraction', 'encapsulate', 'object'], function (key) {
    return key + ': "' + dictionary[key] + '"';
  }).join('; ');
};

dictionary.describe()
//=>
'abstraction: "an abstract or general idea or term"; encapsulate: "to place i\
n or as if in a capsule";
object: "anything that is visible or tangible and is relatively stable in for\
m"'
```

What happens if we get the keys of our object? By now, you know the answer immediately:

```
Object.keys(dictionary)
//=>
[ 'abstraction',
  'encapsulate',
  'object',
  'describe' ]
```

The `describe` property is exactly the same as the `abstraction`, `encapsulate`, and `object` properties. This is not surprising once you’ve grasped the fact that JavaScript is sometimes described as a “Lisp-1.”

What what?

## **lisp-1s and lisp-2s**

One of the big schisms in the history of the Lisp programming languages is over namespaces. In one branch of the tree, functions live in the same namespace as every other kind of value. So a name like `setvar` can be bound to any kind of value: A symbol, a string, a list, a function, whatever. Lisps with this namespace system are called “Lisp-1s” because they have one namespace for everything.<sup>39</sup>

So in a Lisp-1, (`map someList myFun`) calls the function bound to the name `map`, passing along the values bound to the symbols `someList` and `myFun`. `myFun` can (and should) be a function, and that’s fine.

Other Lisps use a different system. Functions live in their own namespace. So `setvar` wouldn’t be bound to a function, but it could be a symbol, list, or anything else. If you want a function named “`setvar`,” you need special syntax to reach into the function namespace, like `#' setvar`.<sup>40</sup>

In a Lisp-2, (`map someList myFun`) calls the function bound to the name `map`, passing along the values bound to the symbols `someList` and `myFun`. But it’s not going to work, because it’s going to look up the non-function value bound to the name `myFun`. To make it work properly, we need something like (`map someList #'myFun`), indicating that we want to go into the function namespace to look up `myFun`.

## **javascript is a javascript-1**

In JavaScript, all values live in the same namespace. Anywhere you write a variable name like `foo`, it could be a function or it could be an “ordinary” value like an object, string, or number. When we write something like:

```
map(someLisp, myFun)
```

All three names (`map`, `someLisp`, and `myFun`) are looked up in the same namespace and can contain functions or ordinary values. This is simpler and cleaner than having special rules for how to look functions up.

## **javascript is also an object-1**

As we’ve seen repeatedly, JavaScript objects have properties, and those properties can be either functions or ordinary values. If we write:

---

<sup>39</sup>Some people argue that Lisp-1 and Lisp-2 are examples of “opaque jargon” and are a communication anti-pattern.

<sup>40</sup>Why are there two different ways to handle namespaces in the Lisp family of languages? Some theorize that it goes back to some early implementation detail, perhaps it was efficient to put all the functions in one big data structure and put everything else in another. or perhaps it saved checking that something really is a function before invoking it, a rudimentary form of static type-checking.

```
dictionary.length
```

We might be accessing a function, we might be accessing a number. The only way to know is to try it:

```
var dictionary = {
  abstraction: "an abstract or general idea or term",
  encapsulate: "to place in or as if in a capsule",
  object: "anything that is visible or tangible and is relatively stable in form"
};

Object.defineProperty(dictionary, 'length', {
  enumerable: false,
  writable: false,
  value: function () {
    return Object.keys(this).length;
  }
});

dictionary.length
//=> [Function]
```

Thus, by default, JavaScript's properties are a single namespace containing both functions and ordinary values. The functions we assign as behaviours of the object live alongside the values that belong to the domain.

Not all “OO” languages are “Object-1s.” Ruby, for example, is an “Object-2.” In Ruby, object methods live in a complete different namespace from their instance variables, and both of them live in a complete different namespace from the contents of containers like Hashes.

For example:

```
dictionary = Object.new

dictionary.instance_variable_set(:@abstraction, "an abstract or general idea or term")

def dictionary.abstraction
  "the act of considering something as a general quality or characteristic, " +
  "apart from concrete realities, specific objects, or actual instances."
end
```

Its methods and instance variables are assigned as if they're separate things. Let's access them from outside the object:

```
dictionary.instance_variable_get(:@abstraction)
#=> "an abstract or general idea or term"

dictionary.abstraction
#=> "the act of considering something as a general quality or characteristic,
apart from concrete realities, specific objects, or actual instances.
```

And from within its own methods?

```
def dictionary.tryThis
  puts @abstraction, nil, abstraction
end

dictionary.tryThis
#=>
an abstract or general idea or term

the act of considering something as a general quality or characteristic,
apart from concrete realities, specific objects, or actual instances.
```

In Ruby, instance variables live in their own namespace separately from methods, you have to use a sigil, @ to access them. Ruby is an Object-2.

## writing javascript in object-2 style

It's a huge benefit that JavaScript is a “Lisp-1” in the sense that there is one namespace for all variables. But it can be a benefit to write JavaScript in an “Object-2” style, separating our methods from our domain properties.

One of the practices we saw earlier was to hide properties by making them non-enumerable. This is often useful for methods:

```
var dictionary = {
  abstraction: "an abstract or general idea or term",
  encapsulate: "to place in or as if in a capsule",
  object: "anything that is visible or tangible and is relatively stable in form"
};

Object.defineProperty(dictionary, 'length', {
  enumerable: false,
  writable: false,
  value: function () {
```

```

    return Object.keys(this).length;
}
});
};

Object.keys(dictionary).indexOf('value') >= 0
//=> false

```

As we see, `Object.keys` gives us the names of the enumerable properties, the ones we're using for domain state. What about the non-enumerable properties? We have a partial answer with `Object.getOwnPropertyNames`:

```

Object.getOwnPropertyNames(dictionary)
//=>
[ 'abstraction',
  'encapsulate',
  'object',
  'length' ]

```

Given this, we can construct:

```

function methods (object) {
  var domainProperties = Object.keys(object);

  return Object.getOwnPropertyNames(object).filter( function (name) {
    return typeof(object[name]) === 'function' && domainProperties.indexOf(name) \
    === -1;
  })
}

methods(dictionary)
//=> ['length']

```

We will need to be strict about making all of your methods non-enumerable to use this function. We'll also have to rethink our approach to listing methods when we start working with metaobjects, the topic of the next chapter.

## To Do

**object composition and delegation**

**state machines and strategies**

**nouns, verbs and commands**

**immediate, forward, and late-binding**

# Methods



41

In object-oriented programming, a method (or member function) is a subroutine (or procedure or function) associated with an object, and which has access to its data, its member variables.—[Wikipedia](#)<sup>42</sup>

---

<sup>41</sup>[Vacuum Pots](#) (c) 2012 Olin Viydo, some rights reserved

<sup>42</sup>[https://en.wikipedia.org/wiki/Method\\_](https://en.wikipedia.org/wiki/Method_)

## What is a Method?

As an abstraction, an object is an independent entity that maintains internal state and that responds to messages by reporting its internal state and/or making changes to its internal state. In the course of handling a message, an object may send messages to other objects and receive replies from them.

A “method” is an another idea that is related to, but not the same as, handling a message. A method is a function that encapsulates an object’s behaviour. Methods are invoked by a calling entity much as a function is invoked by some code.

The distinction may seem subtle, but the easiest way to grasp the distinction is to focus on the word “message.” A message is an entity of its own. You can store and forward a message. You can modify it. You can copy it. You can dispatch it to multiple objects. You can put it on a “blackboard” and allow objects to decide for themselves whether they want to respond to it.

Methods, on the other hand, operate “closer to the metal.” They look and behave like function calls. In JavaScript, methods *are* functions. To be a method, a function must be the property of an object. We’ve seen methods earlier, here’s a naïve example:

```
var dictionary = {
    abstraction: "an abstract or general idea or term",
    encapsulate: "to place in or as if in a capsule",
    object: "anything that is visible or tangible and is relatively stable in form",
    descriptor: function () {
        return map(['abstraction', 'encapsulate', 'object'], function (key) {
            return key + ': "' + dictionary[key] + '"';
        }).join('; ');
    }
};

dictionary.descriptor()
//=>
'abstraction: "an abstract or general idea or term"; encapsulate: "to place i\
n or as if in a capsule";
object: "anything that is visible or tangible and is relatively stable in for\
m"'
```

In this example, `descriptor` is a method. As we saw earlier, this code has many problems, but let’s hand-wave them for a moment. Let’s be clear about our terminology. This `dictionary` object has a method called `descriptor`. The function associated with `descriptor` is called the *method handler*.

When we write `dictionary.descriptor()`, we’re *invoking or calling the descriptor method*. The object is then *handling the method invocation* by evaluating the function.

In describing objects, we refer to objects as encapsulating their internal state. The ideal is that objects **never** directly access or manipulate each other's state. Instead, objects interact with each other solely through methods.

There are many things that methods can do. Two of the most obvious are to *query* an object's internal state and to *update* its state. Methods that have no purpose other than to report internal state are called queries, while methods that have no purpose other than to update an object's internal state are called updates.

## The Letter and the Spirit of the Law

The ‘law’ that objects must only interact with each other through methods is generally accepted as an ideal, even though languages like JavaScript and Java do not enforce it. That being said, there are (roughly) two philosophies about the design of objects and their methods.

- *Literalists* believe that the important thing is the means of interaction be methods. Literalists are noted for using a profusion of getters and setters, which leads to code that is semantically identical to code where objects interact with each other’s internal state, but every interaction is performed indirectly through a query or update.
- *Semanticists* believe that the important thing is that objects provide abstractions over their internal state. They avoid getters and setters, preferring to provide methods that are a level of abstraction above their internal representations.

By way of example, let’s imagine that we have a person object. Our first cut at it involves storing a name. Here’s a first cut at a literalist implementation:

```
var person = Object.create(null, {
  _firstName: {
    enumerable: false,
    writable: true
  },
  _lastName: {
    enumerable: false,
    writable: true
  },
  getFirstName: {
    enumerable: false,
    writable: true,
    value: function () {
      return _firstName;
    }
  },
  setFirstName: {
    enumerable: false,
    writable: true,
    value: function (str) {
      return _firstName = str;
    }
  },
  getLastname: {
```

```

enumerable: false,
writable: true,
value: function () {
  return _lastName;
},
setLastName: {
  enumerable: false,
  writable: true,
  value: function (str) {
    return _lastName = str;
  }
}
});

```

This is largely pointless as it stands. Other objects now write `person.setFirstName('Bilbo')` instead of `person.firstName = 'Bilbo'`, but nothing of importance has been improved. The trouble with this approach as it stands is that it is a holdover from earlier times. Much of the trouble stems from design decisions made in languages like C++ and Java to preserve C-like semantics.

In those languages, once you have some code written as `person.firstName = 'Bilbo'`, you are forever stuck with `person` exposing a property to direct access. Without changing the semantics, you may later want to do something like make `person observable`<sup>43</sup>, that is, we add code such that other objects can be notified when the `person` object's name is updated.

If `firstName` is a property being directly updated by other entities, we have no way to insert any code to handle the updating. The same argument goes for something like validating the name. Although validating names is a morass in the real world, we might have simple ideas such as that the first name will either have at least one character or be `null`, but never an empty string. If other entities directly update the property, we can't enforce this within our object.

In days of old, programmers would have needed to go through the code base changing `person.firstName = 'Bilbo'` into `person.setName('Bilbo')` (or even worse, adding all the observable code and validation code to other entities).

Thus, the literalist tradition grew of defining getters and setters as methods even if no additional functionality was needed immediately. With the code above, it is straightforward to introduce validation and observability:<sup>44</sup>

---

<sup>43</sup>[https://en.wikipedia.org/wiki/Observer\\_pattern](https://en.wikipedia.org/wiki/Observer_pattern)

<sup>44</sup>Later on, we'll see how to use `method combinators` to do this more elegantly.

```
var person = Object.create(null, {
  // ...
  setFirstName: {
    enumerable: false,
    writable: true,
    value: function (str) {
      // insert validation and observable boilerplate here
      return _firstName = str;
    }
  },
  setLastName: {
    enumerable: false,
    writable: true,
    value: function (str) {
      // insert validation and observable boilerplate here
      return _lastName = str;
    }
  }
});
```

That seems very nice, but balanced against this is that contemporary implementations of JavaScript allow you to write getters and setters for properties that mediate access even when other entities are using property access syntax like `person.lastName = 'Baggins'`:

```
var person = Object.create(null, {
  _firstName: {
    enumerable: false,
    writable: true
  },
  firstName: {
    get: function () {
      return _firstName;
    },
    set: function (str) {
      // insert validation and observable boilerplate here
      return _firstName = str;
    }
  },
  _lastName: {
    enumerable: false,
    writable: true
  },
  lastName: {
    get: function () {
      return _lastName;
    },
    set: function (str) {
      // insert validation and observable boilerplate here
      return _lastName = str;
    }
  }
});
```

```

lastName: {
  get: function () {
    return _lastName;
  },
  set: function (str) {
    // insert validation and observable boilerplate here
    return _lastName = str;
  }
}
});

```

The preponderance of evidence suggests that if you are a literalist, you are better off not bothering with making getters and setters for everything in JavaScript, as you can add them later if need be.

## the semantic interpretation of object methods<sup>45</sup>

What about the semantic approach?

With the literalist, properties like `firstName` are decoupled from methods like `setFirstName` so that the implementation of the properties can be managed by the object. Other objects calling `person.setFirstName('Frodo')` are insulated from details such as whether other objects are to be notified when `person` is changed.

But while the implementation is hidden, there is no abstraction involved. The level of abstraction of the properties is identical to the level of abstraction of the methods.

The semanticist takes this one step further. To the semanticist, objects insulate other entities from implementation details like observables and validation, but objects also provide an abstraction to other entities.

In our `person` example, first and last name is a very low-level concern, the kind of thing you think about when you're putting things in a database and worrying about searching and sorting performance. But what would be a higher-level abstraction?

Just a name.

You ask someone their name, they tell you. You ask for a name, you get it. An object that takes and accepts names hides from us all the icky questions like:

1. How do we handle people who only have one name? (it's not just celebrities)
2. Where do we store the extra middle names like Tracy Christopher Anthony Lee?
3. How do we handle formal Spanish names like [Gabriel José de la Concordia García Márquez](#)<sup>46</sup>?

---

<sup>45</sup>With the greatest respect to [Chongo](#), author of “The Homeless Interpretation of Quantum Mechanics.”

<sup>46</sup>[https://en.wikipedia.org/wiki/Gabriel\\_Garc%C3%ADa\\_M%C3%A1rquez](https://en.wikipedia.org/wiki/Gabriel_Garc%C3%ADa_M%C3%A1rquez)

4. What do we do with [maiden names<sup>47</sup>](#) like Arlene Gwendolyn Lee née Barzey or Leocadia Blanco Álvarez de Pérez?

If we expose the low-level fields to other code, we demand that they know all about our object's internals and do the parsing where required. It may be simpler and easier to simply expose:

```
var person = Object.create(null, {  
    _givenNames: {  
        enumerable: false,  
        writable: true,  
        value: []  
    },  
    _maternalSurname: {  
        enumerable: false,  
        writable: true  
    },  
    _paternalSurname: {  
        enumerable: false,  
        writable: true  
    },  
    _premaritalName: {  
        enumerable: false,  
        writable: true  
    },  
    name: {  
        get: function () {  
            // ...  
        },  
        set: function (str) {  
            // ...  
        }  
    }  
});
```

The person object can then do the “icky” work itself. This centralizes responsibility for names.

Now, honestly, people have been handling names in a very US-centric way for a very long time, and few will put up a fuss if you make objects with highly literal name implementations. But the example illustrates the divide between a *literal design* where other objects operate at the same level of abstraction as the object's internals, and a *semantic design*, one that operates at a higher level of abstraction and is responsible for translating methods into queries and updates on the implementation.

---

<sup>47</sup>[https://en.wikipedia.org/wiki/Married\\_and\\_maiden\\_names](https://en.wikipedia.org/wiki/Married_and_maiden_names)

## Composite Methods

One of the primary activities in programming is to *factor* programs or algorithms, to break them into smaller parts that can be reused or recombined in different ways.

Common industry practice is to use the words “decompose” and “factor” interchangeably to refer to any breaking of code into smaller parts. Nevertheless, we will defy industry practice and use the word “decompose” to refer to breaking code into smaller parts whether those parts are to be recombined or reused or not, and use the word “factor” to refer to the stricter case of decomposition where the intention is to recombine or reuse the parts in different ways.

Both methods and objects can and should be factored into reusable components that have a single, well-defined responsibility<sup>48</sup><sup>49</sup>

The simplest way to decompose a method is to “extract” one or more helper methods. For example:

```
var person = Object.create(null, {
    // ...
    setFirstName: {
        enumerable: false,
        writable: true,
        value: function (str) {
            if (typeof(str) === 'string' && str !== '') {
                return this._firstName = str;
            }
        }
    },
    setLastName: {
        enumerable: false,
        writable: true,
        value: function (str) {
            if (typeof(str) === 'string' && str !== '') {
                return this._lastName = str;
            }
        }
    }
});
```

<sup>48</sup>[https://en.wikipedia.org/wiki/Single\\_responsibility\\_principle](https://en.wikipedia.org/wiki/Single_responsibility_principle)

<sup>49</sup>Robert Martin’s rule of thumb for determining whether a method has a single responsibility is to ask when and why it would ever change. If there is just one reason why you are likely to change a method, it has a single responsibility. If there is more than one reason why it might change, it should be decomposed into separate entities that each have a single responsibility.

The methods `setFirstName` and `setLastName` both have a “guard clause” that will not update the object’s hidden state unless the method is passed a non-empty string. The logic can be extracted into its own “helper method.”

```
var person = Object.create(null, {  
  // ...  
  
  isValidName: {  
    enumerable: false,  
    writable: false,  
    value: function (str) {  
      return (typeof(str) === 'string' && str != '');  
    }  
  },  
  setFirstName: {  
    enumerable: false,  
    writable: true,  
    value: function (str) {  
      if (this.isValidName(str)) {  
        return this._firstName = str;  
      }  
    }  
  },  
  setLastName: {  
    enumerable: false,  
    writable: true,  
    value: function (str) {  
      if (this.isValidName(str)) {  
        return this._lastName = str;  
      }  
    }  
  }  
});
```

The methods `setFirstName` and `setLastName` now call the helper method `isValidName`. The usual motivation for this is known as [DRY<sup>50</sup>](#) or “Don’t Repeat Yourself.” The DRY principle is stated as “Every piece of knowledge must have a single, unambiguous, authoritative representation within a system.”

In this case, presumably there is one idea, “person names must be non-empty strings,” and placing the implementation for this in the `isValidString` helper method ensures that now there is just the

---

<sup>50</sup>[https://en.wikipedia.org/wiki/Don%27t\\_repeat\\_yourself](https://en.wikipedia.org/wiki/Don%27t_repeat_yourself)

one authoritative source for the logic, instead of one in each name setter method.

Decomposing a method needn't always be for the purpose of DRYing up the logic. Sometimes, a method breaks down logically into a hierarchy of steps. For example:

```
var person = Object.create(null, {  
  
    // ...  
  
    doSomethingComplicated: {  
        enumerable: false,  
        writable: true,  
        value: function () {  
            this.setUp();  
            this.doTheWork();  
            this.breakdown();  
            this.cleanUp();  
        }  
    },  
    setUp: // ...  
    doTheWork: // ...  
    breakdown: // ...  
    cleanUp: // ...  
});
```

This is as true of methods as it is of functions in general. However, objects have some extra considerations. The most conspicuous is that an object is its own namespace. When you break a method down into helpers, you are adding items to the namespace, making the object as a whole more difficult to understand. What methods call `setUp`? Can `breakdown` be called independently of `cleanUp`? Everything is thrown into an object higgledy-piggledy.

## decluttering with closures

JavaScript provides us with tools for reducing object clutter. The first is the [Immediately Invoked Function Expression<sup>51</sup>](#) (“IIFE”). If our four helpers exist only to decompose `doSomethingComplicated`, we can write:

---

<sup>51</sup>[https://en.wikipedia.org/wiki/Immediately-invoked\\_function\\_expression](https://en.wikipedia.org/wiki/Immediately-invoked_function_expression)

```
var person = Object.create(null, {  
  
    // ...  
  
    doSomethingComplicated: {  
        enumerable: false,  
        writable: true,  
        value: (function () {  
            return function () {  
                setUp.call(this);  
                doTheWork.call(this);  
                breakdown.call(this);  
                cleanUp.call(this);  
            };  
            function setUp () {  
                // ...  
            }  
            function doTheWork () {  
                // ...  
            }  
            function breakdown () {  
                // ...  
            }  
            function cleanUp () {  
                // ...  
            }  
        })()  
    },  
});
```

Now our four helpers exist only within the closure created by the IIFE, and thus it is impossible for any other method to call them. You could even create a `setUp` helper with a similar name for another function without clashing with this one. Note that we're not invoking these functions with `this.`, because they aren't methods any more. And to preserve the local object's context, we're calling them with `.call(this)`.

## decluttering with method objects

In JavaScript, methods are represented by functions. And in JavaScript, *functions are objects*. Functions have properties, and the properties behave just like objects we create with `{}` or `Object.create`:

```
var K = function (x) {
  return function (y) {
    return x;
  };
};

Object.defineProperty(K, 'longName', {
  enumerable: true,
  writable: false,
  value: 'The K Combinator'
});

K.longName
//=> 'The K Combinator'

Object.keys(K)
//=> [ 'longName' ]
```

We can take advantage of this by using a function as a container for its own helper functions. There are several easy patterns for this. Of course, you could write it all out by hand:

```
function doSomethingComplicated () {
  doSomethingComplicated.setUp.call(this);
  doSomethingComplicated.doTheWork.call(this);
  doSomethingComplicated.breakdown.call(this);
  doSomethingComplicated.cleanUp.call(this);
}

doSomethingComplicated.setUp = function () {
  // ...
}

doSomethingComplicated.doTheWork = function () {
  // ...
}

doSomethingComplicated.breakdown = function () {
  // ...
}

doSomethingComplicated.cleanUp = function () {
  // ...
```

```
}
```

```
var person = Object.create(null, {
```

```
// ...
```

```
doSomethingComplicated: {
```

```
enumerable: false,
```

```
writable: true,
```

```
value: doSomethingComplicated
```

```
}
```

```
});
```

If we'd like to make it neat and tidy inline, `tap` is handy:

```
var person = Object.create(null, {
```

```
// ...
```

```
doSomethingComplicated: {
```

```
enumerable: false,
```

```
writable: true,
```

```
value: tap(
```

```
function doSomethingComplicated () {
```

```
doSomethingComplicated.setUp.call(this);
```

```
doSomethingComplicated.doTheWork.call(this);
```

```
doSomethingComplicated.breakdown.call(this);
```

```
doSomethingComplicated.cleanUp.call(this);
```

```
}, function (its) {
```

```
its.setUp = function () {
```

```
// ...
```

```
}
```

```
its.doTheWork = function () {
```

```
// ...
```

```
}
```

```
its.breakdown = function () {
```

```
// ...
```

```
}
```

```
its.cleanUp = function () {
```

```
// ...
}
})
}
});
```

In terms of code, this is no simpler than the IIFE solution. However, placing the helper methods inside the function itself does make them available for use or modification by other methods. For example, you can now use a method decorator on any of the helpers:

```
var logsTheReciver = after( function (value) {
  console.log(this);
  return value;
});

person.doSomethingComplicated.doTheWork = logsTheReciver(person.doSomethingCompli\
cated.doTheWork);
```

This would not have been possible if `doTheWork` was hidden inside a closure.

## summary

Like “ordinary” functions, methods can benefit from being decomposed or factored into smaller functions. Two of the motivations for doing so are to DRY up the code and to break a method into more easily understood and obvious parts. The parts can be represented as helper methods, functions hidden in a closure, or properties of the method itself.

## Meta-Methods

### function helpers

If functions are objects, and functions can have properties, then functions can have methods. We can give functions our own methods by assigning functions to their properties. We saw this previously when we decomposed an object's method into helper methods. Here's the same applied to a function:

```
function factorial (n) {
  return factorial.helper(n, 1);
}

factorial.helper = function helper (n, accumulator) {
  if (n === 0) {
    return accumulator;
  }
  else return helper(n - 1, n * accumulator);
}
```

Functions can have all sorts of properties. One of the more intriguing possibility is to maintain an array of functions:

```
function sequencer (arg) {
  var that = this;

  return sequencer._functions.reduce( function (acc, fn) {
    return fn.call(that, acc);
  }, arg);
}

Object.defineProperties(sequencer, {
  _functions: {
    enumerable: false,
    writable: false,
    value: []
  }
});
```

sequencer is an object-oriented way to implement the pipeline function from function-oriented libraries like [allong.es](http://allong.es)<sup>52</sup>. Instead of writing something like:

---

<sup>52</sup><http://allong.es>

```
function square (n) { return n * n; }
function increment (n) { return n + 1; }

pipeline(square, increment)(6)
//=> 37
```

We can write:

```
sequencer._functions.push(square);
sequencer._functions.push(increment);

sequencer(6)
//=> 37
```

The functions contained within the `_functions` array are helper methods. They work they same way as is we'd written something like:

```
function squarePlusOne (arg) {
  return squarePlusOne.increment(
    squarePlusOne.square(arg)
  );
}
squarePlusOne.increment = increment;
squarePlusOne.square = square;

squarePlusOne(6)
//=> 37
```

The only difference is that they're dynamically looked up helper methods instead of statically wired in place by the body of the function. But they're still helper methods.

## function methods

The obvious problem with our approach is that we aren't using our method as an object in the ideal sense. Why should other entities manipulate its internal state? If this were any other kind of object, we'd expose methods handle messages from other entities.

Let's try it:

```
Object.defineProperties(sequencer, {
  push: {
    enumerable: false,
    writable: false,
    value: function (fn) {
      return this._functions.push(fn);
    }
  }
});
```

Now we can manipulate our sequencer without touching its privates:

```
sequencer.push(square);
sequencer.push(increment);

sequencer(6)
//=> 37
```

Is it a big deal to eliminate the `_functions` reference? Yes!

1. This hides the implementation: Do we have an array of functions? Or maybe we are composing the functions with a combinator? Who knows?
2. This prevents us manipulating internal state in unauthorized ways, such as calling `sequencer._functions.reverse()`

`sequencer.push` is a proper method, a function that handles messages from other entities and in the course of handling the message, queries and/or updates the function's internal state.

If we're going to treat functions like objects, we ought to give them methods.

## aspect-oriented programming and meta-methods

When an object has a function as one of its properties, that's a method. And we just established that functions can have methods. So... Can a method have methods?

Most assuredly.

Here's an example that implements a simplified form of [Aspect-Oriented Programming](#)<sup>53</sup>

Consider a `businessObject` and a `businessObjectCollection`. Never mind what they are, that isn't important. We start with the idea that a `businessObject` can be `valid` or `invalid`, and there's a method for querying this:

---

<sup>53</sup>[https://en.wikipedia.org/wiki/Aspect-oriented\\_programming](https://en.wikipedia.org/wiki/Aspect-oriented_programming)

```
var businessObject = Object.create(null, {
  // ...
  isValid: {
    enumerable: false,
    value: function () {
      // ...
    }
  }
});
```

Obviously, the `businessCollection` has methods for adding business objects and for finding business objects:

```
var businessCollection = Object.create(null, {
  // ...
  add: {
    enumerable: false,
    value: function (bobj) {
      // ...
    }
  },
  find: {
    enumerable: false,
    value: function (fn) {
      // ...
    }
  }
});
```

Our `businessCollection` is just a collection, it doesn't actually do anything to the business objects it holds.

One day, we decide that it is an error if an invalid business object is placed in a business collection, and also an error if an invalid business object is returned when you call `businessCollection.find`. To "fail fast" in these situations, we decide that an exception should be thrown when this happens.

Should we add some checking code to `businessCollection.add` and `businessCollection.find`? Yes, but we shouldn't modify the methods themselves. Since `businessCollection`'s single responsibility is storing objects, business rules about the state of the objects being stored should be placed in some other code.

What we'd like to do is "advise" the `add` method with code that is to be run before it runs, and advise the `find` method with code that runs after it runs. If our methods had methods, we could write:

```
businessCollection.add.beforeAdvice(function (bobjProvided) {
  if (!bobjProvided.isValid())
    throw 'bad object provided to add';
});

businessCollection.find.afterAdvice(function (bobjReturned) {
  if (bobjReturned && !bobjReturned.isValid())
    throw 'bad object returned from find';
});
```

As you can see, we've invented a little protocol. `.beforeAdvice` adds a function "before" a method, and `.afterAdvice` adds a function "after" a method. We'll need a function to make method objects out of the desired method functions:

```
function advisable (methodBody) {

  function theMethod () {
    var args = [].slice.call(arguments);

    theMethod.befores.forEach( function (advice) {
      advice.apply(this, args);
    }, this);

    var returnValue = theMethod.body.apply(this, arguments);

    theMethod.afters.forEach( function (advice) {
      advice.call(this, returnValue);
    }, this);

    return returnValue;
  }

  Object.defineProperties(theMethod, {
    befores: {
      enumerable: true,
      writable: false,
      value: []
    },
    body: {
      enumerable: true,
      writable: false,
      value: body
    }
  });
}
```

```
        },
        afters: {
            enumerable: true,
            writable: false,
            value: []
        },
        beforeAdvice: {
            enumerable: false,
            writable: false,
            value: function (fn) {
                this.befores.unshift(fn);
                return this;
            }
        },
        afterAdvice: {
            enumerable: false,
            writable: false,
            value: function (fn) {
                this.afters.push(fn);
                return this;
            }
        }
    });

    return theMethod;
}
```

Let's rewrite our `businessCollection` to use our `advisable` function:

```
var businessCollection = Object.create(null, {
    // ...
    add: {
        enumerable: false,
        value: advisable(function (bobj) {
            // ...
        })
    },
    find: {
        enumerable: false,
        value: advisable(function (fn) {
            // ...
        })
    }
});
```

```
    }  
});
```

And now, exactly as above, we can write `businessCollection.add.beforeAdvice` and `businessCollection.find.afterAdvice`, separating the responsibility for error checking from the responsibility for managing a collection of business objects.

`beforeAdvice` and `afterAdvice` are methods of methods, or more simply, *meta-methods*.

# Metaobjects



54

In computer science, a metaobject is an object that manipulates, creates, describes, or implements other objects (including itself). The object that the metaobject is about is called the base object. Some information that a metaobject might store is the base object's type, interface, class, methods, attributes, parse tree, etc.—[Wikipedia](#)<sup>55</sup>

It is technically possible to write software just using objects. When we need behaviour for an object, we can give it methods by binding functions to keys in the object:

---

<sup>54</sup>[Krups Machines](#) (c) 2010 Shadow Becomes White, some rights reserved

<sup>55</sup><https://en.wikipedia.org/wiki/Metaobject>

```
var sam = {
  firstName: 'Sam',
  lastName: 'Lowry',
  fullName: function () {
    return this.firstName + " " + this.lastName;
  },
  rename: function (first, last) {
    this.firstName = first;
    this.lastName = last;
    return this;
  }
}
```

We call this a “naïve” object. It has state and behaviour, but it lacks division of responsibility between its state and its behaviour.

This lack of separation has two drawbacks. First, it intermingles properties that are part of the model domain (such as `firstName`) with methods (and possibly other properties, although none are shown here) that are part of the implementation domain. Second, when we needed to share common behaviour, we could have objects share common functions, but does it not scale: There’s no sense of organization, no clustering of objects and functions that share a common responsibility.

Metaobjects solve this problem by separating the domain-specific properties of objects from their behaviour and implementation-specific properties. In classic JavaScript, objects have *prototypes*, and the prototypes are the objects’ metaobjects.

The basic principle of the metaobject is that we separate the mechanics of behaviour from the domain properties of the base object. There are many ways to implement metaobjects, and we will explore some of these as a mechanism for exploring some of the ideas in Object-Oriented Programming.

## Templates and Prototypes

We've already seen extend:

```
var base = {  
  all_your: 'base are belong to us'  
};  
  
var extended = {};  
  
extended.all_your  
//=> undefined  
  
extend(extended, base);  
  
extended.all_your  
//=> 'base are belong to us'
```

extend *copies* the properties from one or more objects into another. This allows us to separate behaviour from domain properties in our code. The behaviour goes in a *template* object, while the domain properties go in a *concrete* object:

```
var RectangleTemplate = {  
  area: function () {  
    return this.length * this.width;  
  }  
};  
  
var twoByThree = extend({  
  length: 2,  
  width: 3  
}, RectangleTemplate);  
  
twoByThree.area();  
//=> 6
```

As noted earlier, JavaScript is an **object-1**, so our methods are as visible as our properties when we do things like enumerate over them. Assuming that our dimensions are in meters, the following (tremendously bad) code won't work:

```
var RectangleTemplate = {
  area: function () {
    return this.length * this.width;
  },
  imperialize: function () {
    for (var dimension in this) {
      this[dimension] = this[dimension] * 3.281;
    }
    return this;
  }
};

var twoByThree = extend({
  length: 2,
  width: 3
}, RectangleTemplate);

twoByThree.imperialize().area()
 //=> TypeError: Property 'area' of object #<Object> is not a function
```

What went wrong?

```
twoByThree
 //=>
 { length: 6.562,
   width: 9.843,
   area: NaN,
   imperialize: NaN }
```

Our methods were enumerated alongside our domain properties. We learned earlier that we can use `Object.defineProperty` to make them non-enumerable. Let's rewrite `extend` to make everything in our template non-enumerable:

```
var naiveExtendWithTemplates = variadic( function (concrete, templates) {
  var key,
    i,
    template;

  for (i = 0; i < templates.length; ++i) {
    template = templates[i];
    for (key in template) {
      if (template.hasOwnProperty(key)) {
        Object.defineProperty(concrete, key, {
          writable: true,
          enumerable: false,
          value: template[key]
        });
      }
    }
  }
  return concrete;
});
```

Let's try it now:

```
var RectangleTemplate = {
  area: function () {
    return this.length * this.width;
  },
  imperialize: function () {
    for (var dimension in this) {
      this[dimension] = this[dimension] * 3.281;
    }
    return this;
  }
};

var twoByThree = naiveExtendWithTemplates({
  length: 2,
  width: 3
}, RectangleTemplate);

twoByThree.imperialize().area()
//=> 64.589766
```

```
twoByThree
//=> { length: 6.562, width: 9.843 }
```

## this does not commute

One problem with our first cut is that you cannot use `naiveExtendWithTemplates` to make a template out of a template:

```
var SquareAwareTemplate = naiveExtendWithTemplates({
  isaSquare: function () {
    return this.length === this.width;
  }
}, RectangleTemplate);

var fourByFour = naiveExtendWithTemplates({
  length: 4,
  width: 4
}, SquareAwareTemplate);

fourByFour.area()
//=> TypeError: Object #<Object> has no method 'area'
```

The problem arises from the fact that we iterate over properties in the template, so we are only looking at *enumerable* properties. But when we copy properties from `RectangleTemplate` into `SquareAwareTemplate`, we made the `RectangleTemplate` properties non-enumerable.

So we need to change things such that we get all the properties, including the non-enumerable ones:

```
var extendWithTemplates = variadic( function (concrete, templates) {
  var key,
    i,
    template;

  for (i = 0; i < templates.length; ++i) {
    template = templates[i];
    Object.getOwnPropertyNames(template).forEach(function (key) {
      Object.defineProperty(concrete, key, {
        writable: true,
        enumerable: false,
        value: template[key]
      });
    });
  };
});
```

```
    }
    return concrete;
});
```

Let's try it again:

```
var SquareAwareTemplate = extendWithTemplates({
  isaSquare: function () {
    return this.length === this.width;
  }
}, RectangleTemplate);

var fourByFour = extendWithTemplates({
  length: 4,
  width: 4
}, SquareAwareTemplate);

fourByFour.area()
//=> 16
```

## prototypes

JavaScript does not normally use templates to separate behaviour, especially shared behaviour, from domain properties. Instead, it uses prototypes. Our example with prototypes, `Object.create` and `extend` looks like this:

```
var RectanglePrototype = extend(Object.create(null), {
  area: function () {
    return this.length * this.width;
  },
  imperialize: function () {
    for (var dimension in this) {
      this[dimension] = this[dimension] * 3.281;
    }
    return this;
  }
});

var SquareAwarePrototype = extend(Object.create(RectanglePrototype), {
  isaSquare: function () {
    return this.length === this.width;
  }
});
```

```
});  
  
var fourByFour = extend(Object.create(SquareAwarePrototype), {  
    length: 4,  
    width: 4  
});
```

In each case, we're creating a new, empty object and copying our properties into it with `extend`. Thanks to the prototype chain:

```
fourByFour.area()  
//=> 16  
fourByFour.isaSquare()  
//=> true  
fourByFour  
//=> { length: 4, width: 4 }
```

We don't actually *need* `extend`, of course, we can always write things like:

```
var SquareAwarePrototype = Object.create(RectanglePrototype);  
SquareAwarePrototype.isaSquare = function () { return this.length === this.width; },
```

## templates vs prototypes

So why use templates instead of prototypes? Or rather, when would it be appropriate? This is a poignant question, especially since prototypes are the standard way to separate behaviour from domain responsibilities in JavaScript.

The decision comes down to this observation: Templates share via shallow copying, prototypes share via reference. This means that changes to a template do *not* propagate after-the-fact:

```

var RectangleTemplate = {
  area: function () {
    return this.length * this.width;
  },
  imperialize: function () {
    for (var dimension in this) {
      this[dimension] = this[dimension] * 3.281;
    }
    return this;
  }
};

var fourByFour = extendWithTemplates({
  length: 4,
  width: 4
}, RectangleTemplate);

RectangleTemplate.isaSquare = function () {
  return this.length === this.width;
};

fourByFour.isaSquare()
//=> TypeError: Object #<Object> has no method 'isaSquare'

```

Whereas:

```

var RectanglePrototype = extend(Object.create(null), {
  area: function () {
    return this.length * this.width;
  },
  imperialize: function () {
    for (var dimension in this) {
      this[dimension] = this[dimension] * 3.281;
    }
    return this;
  }
});

var fourByFour = extend(Object.create(RectanglePrototype), {
  length: 4,
  width: 4
});

```

```
RectanglePrototype.isaSquare = function () {
  return this.length === this.width;
};

fourByFour.isaSquare()
//=> true
```

Making changes to a prototype *after* we create objects with that prototype has the effect of changing the behaviour of those objects. Usually, that's exactly what we want: As a rule of thumb, *we prefer late-binding to early-binding*.

K> Prototypes aren't *always* superior to templates, so we should keep our minds open. JavaScript's prototypes were inspired by the language [Self<sup>56</sup>](#). Self has prototypes, but they are copied like our templates rather than maintaining references. Designers David Ungar and Randall Smith had been working with large Smalltalk applications, and one of their concerns was that large hierarchies of classes became very fragile: Changes to classes would ripple through the system, making it very hard to refactor. Switching to prototypes that were copied like templates made it easier to handle changes to the code.

---

<sup>56</sup><https://en.wikipedia.org/wiki/Self>

## The “I” Word

In [Caffé Lungo](#), our recap of well-known practices, we used the word “inheritance” because that’s the word most people use, and we wanted to focus on JavaScript, not terminology. However in [Prototype Chains and Trees](#), we were very careful not to use the word “inheritance.” What gives?

The answer is simple. Inheritance, it turns out, is a very imprecise term for describing how metaobjects define behavior for domain objects. Everyone “knows what it means,” and what it means varies from language to language and even from library to library within a language.

### a broken metaphor

At its most fundamental level, inheritance in organisms occurs by passing discrete heritable units, called genes, from parents to progeny.—Wikipedia, [Features of Inheritance](#)<sup>57</sup>

In living creatures, inheritance is 100% copying. My son Thomas is tall. Both his mother and I are tall. We have copied parts of our programs—DNA—into his DNA, and thus he is tall. That copying occurred at conception, and thereafter my height has no effect on his height.

As I age, I am actually growing shorter, but he does not grow shorter. Likewise, although my hair turns silver, his hair remains brown. We maintain a social bond, but purely physical traits like height and hair colour are now independent. This is exactly how the [templates](#) work: Behaviour is shared from one object to another with copying, and thereafter changes to one object cannot affect the other object.

This is not how [prototypes](#) work at all. If you change a prototype, the behaviour of all objects “inheriting” from it is changed instantly. There is a dynamic relationship between every domain objects and the prototypes in its prototype chain.

The difference between templates and prototypes is crucial, and the word “inheritance” does a very poor job of disambiguating the two cases.

### a word we’ve been saving for just the proper occasion

There is a word that describes the relationship between an object and a prototype: [delegation](#):<sup>[^]</sup>

Most commonly, [delegation] refers to a programming language feature making use of the method lookup rules for dispatching so-called self-calls as defined by Lieberman in his 1986 paper “Using Prototypical Objects to Implement Shared Behavior in Object-Oriented Systems”. Delegation as a language feature supports the prototype-based programming model.

---

<sup>57</sup>[https://en.wikipedia.org/wiki/Genetics#Features\\_of\\_inheritance](https://en.wikipedia.org/wiki/Genetics#Features_of_inheritance)

If I say that “I’m delegating all questions about my finances to my accountant,” I am not saying that my accountant is my mother and that I inherited a talent for totting up columns of figures from her. I’m saying that at the moment you ask me a question, I pass the question on to my accountant, who answers the question in the context of my finances. The word “delegation” implies several things:

1. The behaviour for one entity, the delegator, is handled by another entity, the delegate.
2. The delegator delegates handling the behaviour to the delegate at the time the behaviour is needed.
3. Messages are handled in the context of the delegator, not the delegate.

The third point is important: If the Canada Revenue Agency demands my annual income and I delegate that question to my accountant, the answer that comes back is going to be my annual income, not my accountant’s annual income.<sup>58</sup>

*Delegation* is the correct term for the relationship between an object and its prototype. An object doesn’t inherit behaviour from a prototype, it delegates handling *to* a prototype. And that prototype may in turn delegate to its prototype, and so forth along a prototype chain, with each object passing the context along.

## asymmetry in physical laws

Delegation implies a certain symmetry between the delegator and delegate. If I delegate work to my accountant, we are discussing a relationship of equals. I am asked to do some work, I delegate to someone else who will do the work.

That is very different than how template inheritance works. My son doesn’t delegate his height to me, he has his own height that is a consequence of some information I passed along to him at conception. This is more like what happens if we have a set of blueprints and we build houses from them. The blueprints define the characteristics of the houses, but if you want to know how many rooms a house has, you look at the house. You don’t ask it how many rooms it has and wait for it to delegate the question to the architecture firm that drew up the plans.

Template inheritance implies an asymmetry between an object and the object(s) that determine its behaviour. In common JavaScript, constructor functions act as blueprints, but not delegates. for example:

---

<sup>58</sup>A similar, yet different relationship is called “forwarding:” If I forward questions about my old koi pond to the people who bought the house and pond from me, they are going to answer the questions in their own context, not mine. Objects that forward behaviour to other objects relinquish the context.

```
var Counter = (function () {
  var i = 0;
  return {
    decrement: function count () {
      return i--;
    },
    increment: function count () {
      return i++;
    },
    peek: function peek () {
      return i;
    },
    reset: function reset () {
      i = 0;
    }
  };
});

var c = Counter();
```

Counter obviously defines the behaviour of c, but it isn't a delegate.

## Constructing Objects

As we discussed with respect to prototypes, a desire to create small, focused elements with clear responsibilities leads to creating lots of small prototypes. The desire to eliminate duplication leads to the prototypes being maximally shared, and the result is a thickly branching tree of prototypes, with each object delegating behaviour a chain of prototypes.

This has two drawbacks. The first drawback has already been mentioned: A tree is a fragile architecture for sharing prototypes. Changes to any one prototype affect large numbers of other prototypes and by extension, the objects that depend on the prototype being changed.

The second drawback of a tree is that non-trivial domains cannot usually be represented as trees of functionality. Consider our hypothetical representation of DOM elements: Some elements—like paragraphs and lists—are block elements. Others—like spans and emphasized text—are inline elements.

Obviously, at some point in a paragraph's prototype chain there will be a prototype with block element behaviour. Likewise, at some point in a span's prototype chain there will be a prototype with inline element behaviour. But the span won't delegate to the block prototype, and the paragraph won't delegate to the inline prototype.

All this falls apart when we consider images: An image may appear either as a block element or as an inline element. How does the image's prototype chain include both the block and inline prototypes?

This isn't an isolated case, either. Some elements, like paragraphs, are containers. Some, like hard rules ("HR"), are not containers. How do we resolve the problem that containers can be both blocks (like paragraphs) and inline (like emphasized text) while non-containers can be blocks (like hard rules) or inline (like line breaks).

JavaScript's prototypical delegation can only form trees, and what we've just described is a directed acyclic graph ("DAG"). A tree simply can't model the behaviour we want to define for DOM elements unless we duplicate some behaviour.

## duplication

One option is to duplicate behaviour in prototypes. If our chains look like this:

```
var ElementPrototype = {};  
  
var BlockPrototype = Object.create(ElementPrototype);  
var InlinePrototype = Object.create(ElementPrototype);  
  
var P = Object.create(BlockPrototype);  
var EM = Object.create(InlinePrototype);
```

We can create:

```
var HR = Object.create(BlockPrototype);
```

And then we can *copy* the properties from `InlinePrototype` into `HR`:

```
extend(HR, InlinePrototype);
```

Now our `HR` has all the properties of `BlockPrototype` through delegation, and all the properties of `InlinePrototype` through duplication.

This scheme has two drawbacks of its own.

First, if we subsequently add or remove properties from `InlinePrototype`, those changes will not be reflected in `HR`. It may seem fanciful to imagine a domain program making these kinds of changes, but sometimes we load libraries that make these kinds of changes for us. If we program by copying elements from prototypes, we could create a situation where `P` block elements behave differently than `HR` block elements, and worse, we could have a situation where our program becomes very dependent upon the order that libraries and modules are loaded.

Second, we lose our ability to reflect on our prototypes:

```
var p = Object.create(P);
var em = Object.create(EM);
var hr = Object.create(HR);

BlockPrototype.isPrototypeOf(p)
 //=> true
BlockPrototype.isPrototypeOf(hr)
 //=> true
```

However:

```
InlinePrototype.isPrototypeOf(em)
 //=> true
InlinePrototype.isPrototypeOf(hr)
 //=> false
```

JavaScript is very strict about the way that `.isPrototypeOf` behaves.

## Metaobject-1s and Metaobject-2s

*TODO: Begin with the transition from function constructor to .create*

## Singleton Prototypes

In classic JavaScript, we delegate object behaviour to prototypes. Typically, prototypes are used for *shared* object behaviour:

```
var RectanglePrototype = extend(Object.create(null), {
    area: function () {
        return this.length * this.width;
    },
    isaSquare: function () {
        return this.length === this.width;
    }
});

var twoByThree = extend(Object.create(RectanglePrototype), {
    length: 2,
    width: 3
});

var fourByFour = extend(Object.create(RectanglePrototype), {
    length: 4,
    width: 4
});

twoByThree.area()
//=> 6

fourByFour.area()
//=> 16
```

There is a problem with this approach. What happens when we want to customize the behaviour of one object?

```
fourByFour.isaJeep = function () { return false; }

fourByFour
//=>
{ length: 4,
  width: 4,
  isaJeep: [Function] }
```

Once again, we're mixing behaviour with domain properties. We can make `isaJeep` non-enumerable, but having embraced the idea of separating behaviour from domain properties with a prototype, why not use a prototype here? The trick is to create a unique prototype for *every* object.

Like this:

```
var fourByFour = (function () {
  var SingletonPrototype = Object.create(RectanglePrototype);

  return extend(Object.create(SingletonPrototype), {
    length: 4,
    width: 4
  });
})();
```

`SingletonPrototype` is an empty object that delegates to `RectanglePrototype`, but no object except `fourByFour` delegates to `SingletonPrototype`. We can rewrite it more simply:

```
var fourByFour = extend(
  Object.create(Object.create(RectanglePrototype)),
  {
    length: 4,
    width: 4
  }
);
```

Now when we want to bind behaviour to `fourByFour`, we can write:

```
Object.getPrototypeOf(fourByFour).isaJeep = function () { return false; }

fourByFour
//=> { length: 4, width: 4 }

fourByFour.isaJeep()
//=> false
```

When a prototype exists solely to manage behaviour for a single object, we call it a *singleton prototype*. It's a specific case of a *singleton metaobject*. JavaScript is agnostic about singleton metaobjects, it's up to you whether to use them.

Other languages have support for singleton metaobjects baked in. For example, in Ruby you can define a method specific to an object:

```
obj = Object.new
def obj.ect
  'Object'
end
```

Behind the scenes, Ruby creates a “singleton class” just for `obj` and then defines the method in the singleton class. We can see this:

```
obj.singleton_class.instance_methods - Object.methods
#=> [:ect]
```

## To Do

**metaclasses and meta-metaobjects**

**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**

**prototype chaining**

**single inheritance**

**mixins and multiple inheritance**

**resolution: merge, override, and final**

**method guards and contracts**

**early and late method composition**

**multiple dispatch and generic functions**

**pattern matching protocols**

# **Appendices**

## Utility Functions

Throughout this book, we have used utility functions in our code snippets and examples. Here is a list of functions we've borrowed from other sources. Most are from [underscore<sup>59</sup>](#) or [allong.es<sup>60</sup>](#).

```
var allong = require('allong.es');
var _ = require('underscore');
```

### extend

extend can be found in the [underscore<sup>61</sup>](#) library:

```
var extend = _.extend;
```

Or you can use this formulation:

```
var __slice = [] .slice;

function extend () {
  var consumer = arguments[0],
    providers = __slice.call(arguments, 1),
    key,
    i,
    provider,
    except;

  for (i = 0; i < providers.length; ++i) {
    provider = providers[i];
    except = provider['except'] || [];
    except.push('except');
    for (key in provider) {
      if (except.indexOf(key) < 0 && provider.hasOwnProperty(key)) {
        consumer[key] = provider[key];
      };
    };
  };
  return consumer;
};
```

---

<sup>59</sup><http://underscorejs.org>

<sup>60</sup><http://allong.es>

<sup>61</sup><http://underscorejs.org>

## pipeline

`pipeline` composes functions in the order they are applied:

```
var pipeline = allong.es.pipeline;

function square (n) { return n * n; }
function increment (n) { return n + 1; }

pipeline(square, increment)(6)
//=> 37
```

## tap

`tap` passes a value to a function, discards the result, and returns the original value:

```
var tap = allong.es.tap;

tap(6, function (n) {
  console.log("Hello there, " + n);
});
//=>
Hello there, 6
6
```

## map

`map` applies function to an array, returning an array of the results:

```
var map = allong.es.map;

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

## variadic

`variadic` takes a function and returns a “variadic” function, one that collects arguments in an array and passes them to the last parameter:

```
var variadic = allong.es.variadic;

var a = variadic(function (args) {
  return { args: args };
});

a(1, 2, 3, 4, 5)
//=> { args: [1, 2, 3, 4, 5] }

var b = variadic(function (first, second, rest) {
  return { first: first, second: second, rest: rest };
});

b(1, 2, 3, 4, 5)
//=> { first: 1, second: 2, rest: [3, 4, 5] }
```

## unvariadic

unvariadic takes a variadic function and turns it into a function with a fixed arity:

```
var unvariadic = allong.es.unvariadic;

function ensuresArgumentsAreNumbers (fn) {
  return unvariadic(fn.length, function () {
    for (var i in arguments) {
      if (typeof(arguments[i]) !== 'number') {
        throw "Ow! NaN!!"
      }
    }
    return fn.apply(this, arguments);
  });
}

function myAdd (a, b) {
  return a + b;
}

var myCheckedAdd = ensuresArgumentsAreNumbers(myAdd);

myCheckedAdd.length
//=> 2
```