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 2014-01-10

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

Also By Reginald Braithwaite

Kestrels, Quirky Birds, and Hopeless Egocentricity

What I've Learned From Failure

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

CoffeeScript Ristretto

JavaScript Allongé

Contents

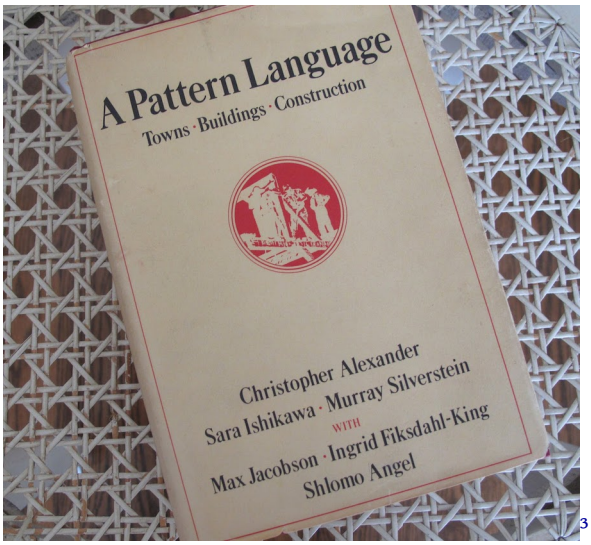
Prefaces

[1]

[1] Group (c) 2013 J MacPherson, some rights reserved

Foreword

There are many ways to write books about Architecture. One might attempt to catalogue a broad variety of problems and their solutions, at scales ranging from designing communities down to items of furniture, as Christopher Alexander did in A Pattern Language[2]:



[3]

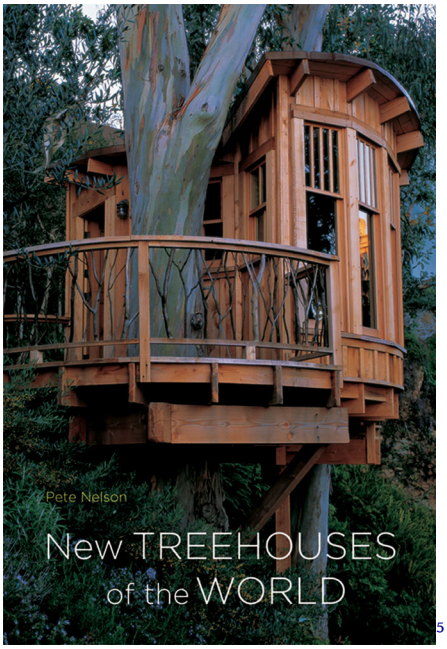
New Treehouses of the World

One might also pick a particular domain and then survey how different architects approached solving a common set of problems, highlighting the variety of possible styles. New Treehouses of the World[4] takes this approach:

[2] http://www.amazon.com/gp/product/0195019199/ref=as_li_ss_tl?ie=UTF8&camp=1789&creative=390957&creativeASIN=0195019199&linkCode=as2&tag=raganwald001-20

[3] http://www.amazon.com/gp/product/0195019199/ref=as_li_ss_tl?ie=UTF8&camp=1789&creative=390957&creativeASIN=0195019199&linkCode=as2&tag=raganwald001-20

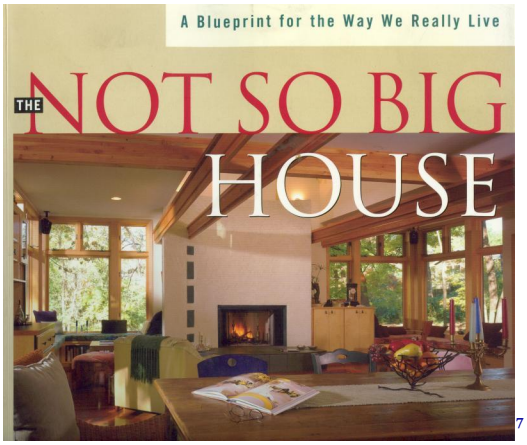
[4] http://www.amazon.com/gp/product/0810996324/ref=as_li_ss_tl?ie=UTF8&camp=1789&creative=390957&creativeASIN=0810996324&linkCode=as2&tag=raganwald001-20



[5]

The Not So Big House

Or one might approach architecture from a single viewpoint, describing a highly specific and coherent style backed by personal experience, as Sarah Susanka did in The Not So Big House[6]:



[7]

[5] http://www.amazon.com/gp/product/0810996324/ref=as_li_ss_tl?ie=UTF8&camp=1789&creative=390957&creativeASIN=0810996324&linkCode=as2&tag=raganwald001-20

[6] http://www.amazon.com/gp/product/1600851509/ref=as_li_ss_tl?ie=UTF8&camp=1789&creative=390957&creativeASIN=1600851509&linkCode=as2&tag=raganwald001-20

[7] http://www.amazon.com/gp/product/1600851509/ref=as_li_ss_tl?ie=UTF8&camp=1789&creative=390957&creativeASIN=1600851509&linkCode=as2&tag=raganwald001-20

JavaScript Spessore

JavaScript Spessore[8] follows this path. This book will not attempt to dictate the "one true definition" of object-oriented programming. It will not survey and catalog the many different programming languages and libraries people have used to solve software problems with objects.

Instead, JavaScript Spessore describes one way to design programs with objects as the central idea: Building software on top of a metaobject protocol. A metaobject protocol is a kind of abstraction for implementing object-oriented programming semantics. One program might be developed with pattern matching for methods, another might use state machines, a third might be heavily factored into aspects.

By implementing this metaobject protocol and semantics on top of it, we gain insight into how these semantics work and how they might be applied to solving problems we encounter as software developers.

[8] https://leanpub.com/javascript-spessore

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[9] ("**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[10], 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[11], you can't help but feel comfortable using JavaScript callbacks in Node.js[12].

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.

---

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

[10] 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.

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

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

JavaScript Allongé and allong.es

JavaScript Spessore[13] is written for the reader who has read JavaScript Allongé[14] 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

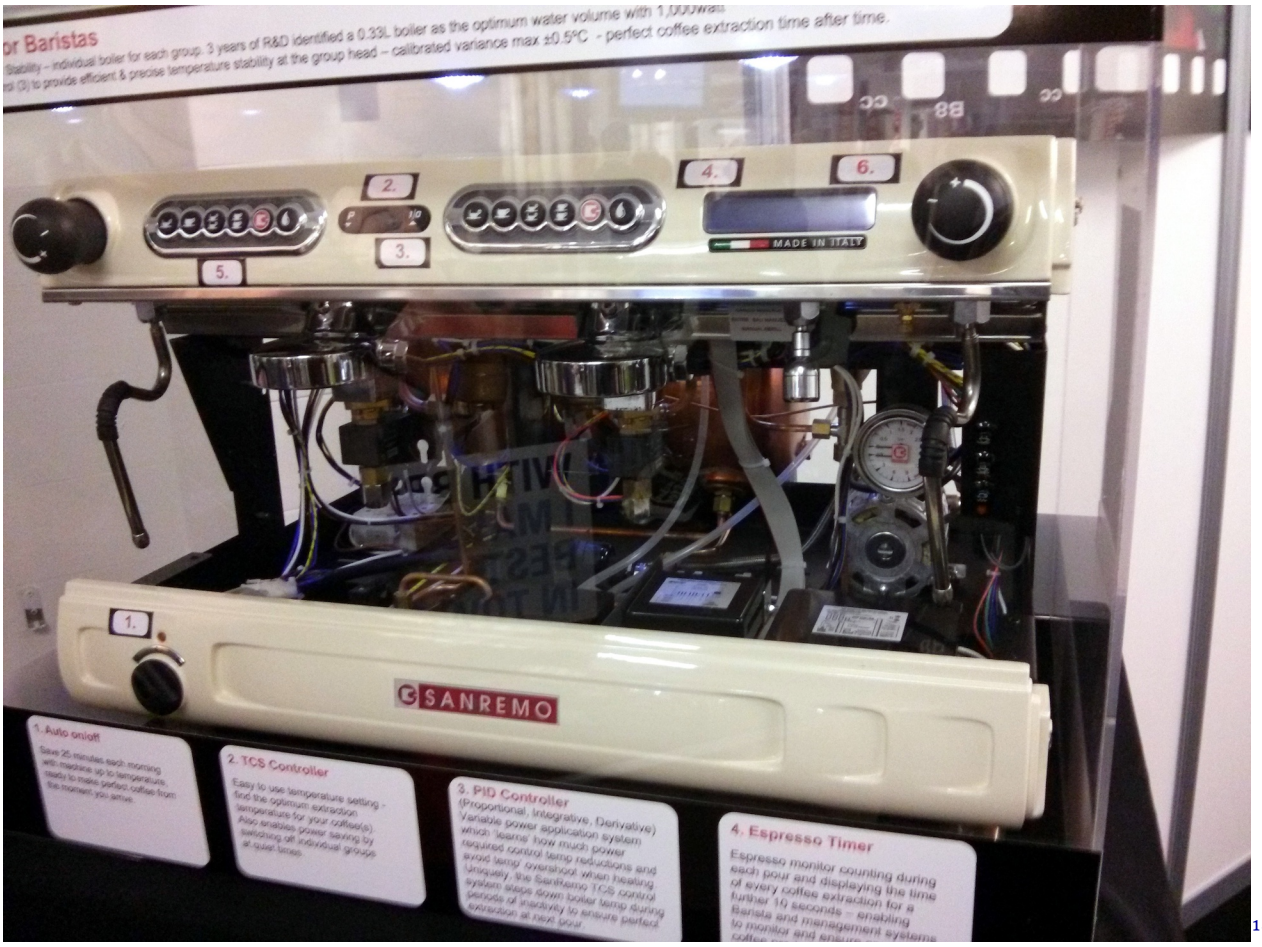
Even if you know the material, you may want to read JavaScript Allongé to familiarize yourself its approach to functional combinators. You can read it for free online[15].

allong.es

allong.es[16] 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[17] page and see them work.

[13] https://leanpub.com/javascript-spessore
[14] https://leanpub.com/javascript-allonge
[15] https://leanpub.com/javascript-allonge/read
[16] http://allong.es
[17] http://allong.es/try/

Preamble



[18]

[18] Transparent Sanremo espresso machine, London Coffee Festival, Truman Brewery, Brick Lane, Hackney, London, UK (c) 2013 Cory Doctorow, some rights reserved

Objects and References

The material in this preamble should be familiar to the working JavaScript programmer, so feel free to skip it if you are in a hurry. However, it never hurts to review the "obvious:" Prefetching the basics into your L2 cache can improve learning performance when tackling more advanced material.

This expression evaluates to an object:

```
{}
  //=> {}
```

So does this:

```
Object.create(null)
  //=> {}
```

This one lets us define properties in a familiar way:

```
var acct = {
  transitNumber: '129131673',
  accountNumber: '0114584906'
}
```

There is a syntax for accessing property contents:

```
acct.accountNumber
  //=> '0114584906'
```

We can also use it to assign properties directly:

```
var cheque = {};
cheque.number = 1;
```

We can write a function to extend one object with the properties of another:

```
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;
};


var account = {
  transitNumber: '129131673',
  accountNumber: '0114584906'
};

extend(account, {
  type: 'chequing',
  name: 'slush fund'
});

account.name
  //=> 'slush fund'
```

JavaScript properties must be named with strings, but can have any value:

```
{
  one: 1
}
  //=> { one: 1 }
```

If a name isn't a string, it will be converted to a string:

```
{
  1: 'one'
}
  //=> { '1': 'one' }
```

Since objects are values, objects can have objects as values:

```
{
  transitNumber: {
    federalReserveRouting: {
      bank: '12',
      processingCentre: '9',
      location: '1'
    },
    abaInstitution: '3167',
    checkDigit: '3'
  },
  accountNumber: '0114584906'
}
```

Object values are references:

```
var reserveRouting = {
  bank: '12',
  processingCentre: '9',
  location: '1'
};

var transitNumber = {
  federalReserveRouting: reserveRouting,
  abaInstitution: '3167',
  checkDigit: '3'
};
```

```
var bankAccount = {
  transitNumber: transitNumber,
  accountNumber: '0114584906'
};

var amount = {
  dollars: 2,
  cents: 56
}

var cheque = {
  bankAccount: bankAccount,
  number: '1',
  amount: amount
}

cheque
  //=>
    { bankAccount:
       { transitNumber:
          { federalReserveRouting:
             { bank: '12',
               processingCentre: '9',
               location: '1' },
            abaInstitution: '3167',
            checkDigit: '3' },
         accountNumber: '0114584906' },
      number: '1',
      amount: { dollars: 2, cents: 56 } }
```

Since they are references, mutating an object means that its value in one place mutates its value everywhere:

```
var cheque2 = {
  bankAccount: bankAccount,
  number: '2',
  amount: {
    dollars: 1,
    cents: 0
  }
}
```

```
cheque2.bankAccount.transitNumber
  //=>
    { federalReserveRouting:
       { bank: '12',
         processingCentre: '9',
         location: '1' },
      abaInstitution: '3167',
      checkDigit: '3' }

extend(reserveRouting, {
  processingCentre: '1',
  location: '0'
});

cheque.bankAccount.transitNumber
  //=>
    { federalReserveRouting:
       { bank: '12',
         processingCentre: '1',
         location: '0' },
      abaInstitution: '3167',
      checkDigit: '3' }

cheque2.bankAccount.transitNumber
  //=>
    { federalReserveRouting:
       { bank: '12',
         processingCentre: '1',
         location: '0' },
      abaInstitution: '3167',
      checkDigit: '3' }

      month: month,
      day: 1
    },
    amount: extend({}, rentAmount)
  }
});

rentCheques[7].amount.dollars
  //=> 420
```

```
rentCheques[0].amount.dollars
  //=> 420
```

The expression `extend({}, rentAmount)` copies all the properties of `rentAmount` into a new, empty object. It is evaluated every time we create a new cheque, so we get a new amount object for each cheque. Thus, when we change some of them, they are the only ones to change:

```
[4, 5, 6, 7, 8, 9, 10, 11, 12].forEach( function (month) {
  rentCheques[month - 1].amount.dollars = 600;
});

rentCheques[7].amount.dollars
  //=> 600

rentCheques[0].amount.dollars
  //=> 420
```

The drawback of this approach is that creating new objects takes both time and space, and it becomes very expensive. You might decide that the flaw was in assigning a new dollar amount instead of changing the entire amount:

```
[4, 5, 6, 7, 8, 9, 10, 11, 12].forEach( function (month) {
  rentCheques[month - 1].amount = {
    dollars: 600,
    cents 0
  }
});
```

This would also have worked. If these were our only options, we would balance their respective considerations before making a choice:

1. Making separate amount objects when we crate cheques is slow and takes up space, but it is resistant to programmers making mistakes later.
2. Making new amount objects when we need to make changes is faster and tighter, but requires discipline from programmers to know when to make copies.

When we discuss defined properties, we will look at some techniques for making code more resistant to errors.

Summary

coming soon

What is a Metaobject Protocol?

[19]

> A metaobject protocol provides the vocabulary to access and manipulate the structure and behavior of objects. Typical functions of a metaobject protocol include: Creating and deleting new classes; Creating new methods and properties; Changing the class structure so that classes inherit from different classes; Generating or modifying the code that defines the methods for the class.–Wikipedia[20]

Languages with support for object-oriented programming typically provide their own "Vocabulary to access and manipulate the structure and behavior of objects" at runtime. Ruby, for example, provides keywords like `def` and `class`, as well as imbuing objects and modules with a variety of methods for inspecting and modifying behaviour while the program is running.

[19] Normalized Pump Pressure Reading (c) 2009 Nicholas Lundgaard, some rights reserved

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

JavaScript can be said to have an extremely minimalist metaobject protocol. In this book, we'll look at building our own, richer metaobject protocols on top of JavaScript. Our metaobject protocols will be guided by the philosophy that object behaviour should be divied amongst three entities:

- Objects are responsible for the domain-specific properties of entities we're modeling.
- Metaobjects are responsible for encapsulating the behaviour of base objects.
- Protocols are façades for insulating code that defines behaviour from the implementation of metaobjects.

What is an Object?

> A Smalltalk object can do exactly three things: Hold state (references to other objects), receive a message from itself or another object, and in the course of processing a message, send messages to itself or another object.–Smalltalk on Wikipedia[21]

Objects seem simple enough: They hold state, they receive messages, they process those messages, and in the course of processing messages, they also send messages. This brief definition implies an important idea: Objects can *change state* in the course of processing messages. They do this directly by removing, changing, or adding to the references they hold.

messages and method invocations

A nine year-old messenger boy

Smalltalk speaks of "messages." The metaphor of a message is very clear. A message is composed and sent from one entity (the "sender") to one entity (the "recipient"). The sender specifies the identity of the recipient. The message may contain information for the recipient, instructions to perform some task, or a question to be answered. An immediate reply may be requested, or the sender may trust the message's recipient to act appropriately. The metaphor of the "message" emphasizes the arms-length relationship between sender and recipient.[22]

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

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

[22] 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.

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.

references and state

You can imagine sending a message to a mathematician: "What's the biggest number: one, five, or four?" You can do that is 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[23] 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 use handlers to respond to**

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

messages. It maintains internal state, and its handlers are responsible for querying and/or updating its state.

We'll have a closer look at implementing objects in Javascript a little later in this book.

What is a Metaobject?

> 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[24]

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:

```
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.

singleton metaclasses

Metaobjects solve this problem by separating the domain-specific properties of objects from their behaviour and implementation-specific properties. In classic JavaScript, we can use a prototype for the behaviour. We call this a "singleton metaobject:"

[24] https://en.wikipedia.org/wiki/Metaobject

```
var singletonMetaobject = {};
var sam = Object.create(singletonMetaobject);

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

Notice that a prototype provides value even when we have no intention of sharing behaviour: Its value is that it separates behaviour from our model's domain properties very neatly:

```
Object.getOwnPropertyNames(sam)
  //=> [ 'firstName', 'lastName' ]
```

Insulating our base objects from our metaobjects is important for hiding implementation details, a core "OO" value. When we provide a method like `fullName`, we're hiding details of how to compute a full name from other entities, all they need to know is the name of the message to send.

hiding implementation details

But we need to hide implementation details within an object as well. Consider logging.[25] If we wish to write to the console every time an object is renamed, we have many choices. Here are three:

First, we can rewrite our function:

[25] The canonical example of a cross-cutting concern. We could also consider implementing undo and transactions, they have similar characteristics.

```
singletonMetaobject.rename = function (first, last) {
  console.log(this.fullName() + " is being renamed " + first + " " + last);
  this.firstName = first;
  this.lastName = last;
  return this;
};
```

Second, we can use a method combinator[26]:

```
var logsNameChange = before( function (first, last) {
  console.log(this.fullName() + " is being renamed " + first + " " + last);
});

singletonMetaobject.rename = logsNameChange( function (first, last) {
  this.firstName = first;
  this.lastName = last;
  return this;
});
```

Third, we could use an aspect-oriented programming library like YouAreDaChef[27]:

```
YouAreDaChef(sam).before('rename', function (first, last) {
  console.log(this.fullName() + " is being renamed " + first + " " + last);
});
```

Never mind which choice we should exercise. The important thing is that whatever we do should be hidden from the `sam` object itself! All it "knows" is that it has a prototype. What happens inside that prototype ought to be opaque. Is the `rename` function rewritten? Is it recombined? Is there a `before_rename` property with list of functions to execute? That should be irrelevant to the `sam` object itself.

The basic principle of the metaobject is that we separate the mechanics of behaviour from the domain properties of the base object. Everything else, like how inheritance is implemented, or whether a method is being logged, is hidden from the object.

[26] https://github.com/raganwald/method-combinators

[27] https://github.com/raganwald/YouAreDaChef

What is a Protocol?

We've seen that a metaobject separates an object's domain-specific properties from the implementation details of its behaviour. The implementation details might be simple, such as a few methods that are expressed as functions, or complex, such as an inheritance hierarchy with composite methods.

A metaobject *protocol* (or just "protocol") is an interface for managing metaobjects. While the metaobject is responsible for the object's behaviour, the protocol is responsible for creating, changing, and removing behaviour from metaobjects.

protocols and object creation

Previously, we showed how to use a prototype as a singletonMetaobject:

```
var singletonMetaobject = {};
var sam = Object.create(singletonMetaobject);

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

A protocol can hide this implementation detail behind a function:

```
function object(superprototype, propertyDescriptors) {
  if (arguments.length == 0 || typeof(superprototype) == 'undefined') {
    superprototype = Object.prototype;
  }
  var singletonMetaobject = Object.create(superprototype)
  return Object.create(singletonMetaobject, propertyDescriptors || {});
}

var sam = object();
extend(sam, {
  firstName: 'Sam',
  lastName: 'Lowry'
});
```

Adding methods to the singleton metaobject is also hidden behind a function. We'll elide the details for the moment:

```
function extendWithMethods (object, methodDescriptions) {
  for (var methodName in methodDescriptions) {
    // ...
  }
  return object;
}

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

protocols and advice

In the previous section, we gave three different implementations for adding logging to the `rename` method. The base object is insulated from the details by its metaobject, and the code that attaches the logging should likewise be insulated from the implementation choice.

Our protocol provides the insulation for us:

```
function method (object, methodName, optionalOverride) {
  // ...
}

method(sam, 'rename').unshift( function (firstName, lastName) {
  console.log(this.fullName() + " is being renamed " + firstName + " " + lastName\
);
});

sam.rename('Samuel', 'Lowrie')
  //=> Sam Lowry is being renamed Samuel Lowrie
  //   { firstName: 'Samuel', lastName: 'Lowrie' }
```

The metaobject hides the implementation of "advice" from the base object. However, the implementation is exposed to the code responsible for adding logging to base objects. This is where the metaobject comes in.

To be specific, we meant that a base object should be insulated from the implementation consequences of rewriting a method, applying a combinator to it, or applying "before advice" to it. But we go further: In addition to insulating the base object from the implications of this choice, we must insulate the programmer from needing to think through and possibly re-invent the implementation of any of these three choices.

Summary

Metaobject protocols are vocabularies for accessing and manipulating the structure and behavior of objects. The protocols in this book will be implemented with three main entities:

- *Objects*: Objects are responsible for domain-specific properties.
- *Metaobjects*: Metaobjects are responsible for the insulating objects from the implementation of their behaviour.
- *Protocols*: Protocols are responsible for insulating other code from the implementation of metaobject behaviour.

Objects

[28]

introductory remarks coming soon

[28] Londinium 1 Lever Espresso Machine (c) 2013 Alejandro Erickson, some rights reserved

Objects, Revisited

coming soon

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:

```
var allong = require('allong.es');
var tap = allong.es.tap;

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

Objects and Functions

Since functions are values, objects can have functions as values:

```
var allong = require('allong.es');
var variadic = allong.es.variadic;
var map = allong.es.map;

var getAll = variadic( function (object, propertyNames) {
  return map(propertyNames, function (name) { return object[name]; });
});

var federalReserve1 = {
  bank: '12',
  processingCentre: '9',
  location: '1',
  toString: function () {
    return getAll(federalReserve1, 'bank', 'processingCentre', 'location').join('\
');
  }
};

var transitNumber1 = {
  federalReserveRouting: federalReserve1,
  abaInstitution: '3167',
  checkDigit: '3',
  toString: function () {
    return getAll(transitNumber1, 'federalReserveRouting', 'abaInstitution', 'che\
ckDigit').join('');
  }
};

var account1 = {
  transitNumber: transitNumber1,
  accountNumber: '0114584906',
  toString: function () {
    return account1.transitNumber + '-' + account1.accountNumber;
  }
}
```

You can call those functions by referring to them with `.`, and then invoking them with `()`:

```
account1.toString()
  //=> '129131673-0114584906'
```

Functions that are associated with properties are values, we can do whatever we want with them:

```
var accountStringifier = account1.toString;

accountStringifier
  //=> [function]

accountStringifier()
  //=> 129131673-0114584906'
```

You have to be careful when you share a single function amongst multiple objects. Watch the 'copypasta' error:

```
var amount = {
  dollars: 2,
  cents: 56,
  toString: function () {
    return '$' + amount.dollars + '.' + (amount.cents < 10 ? ('0' + amount.cents)\
 : amount.cents)
  }
}

var amount2 = {
  dollars: 1,
  cents: 0,
  toString: function () {
    return '$' + amount.dollars + '.' + (amount.cents < 10 ? ('0' + amount.cents)\
 : amount.cents)
  }
}

amount.toString()
  //=> '$2.56'

amount2.toString()
  //=> '$2.56'
```

Some simple factoring will help.

```
function inDollarsAndCents (amount) {
  return '$' + amount.dollars + '.' + (amount.cents < 10 ? ('0' + amount.cents) :\
 amount.cents);
};

var amount = {
  dollars: 2,
  cents: 56,
  toString: function () {
    return inDollarsAndCents(amount);
  }
}

var amount2 = {
  dollars: 1,
  cents: 0,
  toString: function () {
    return inDollarsAndCents(amount2);
  }
}

amount.toString()
  //=> '$2.56'

amount2.toString()
  //=> '$1.00'
```

This is a general principle: *functions that belong to objects really need a reference to the current object to be useful.*

this

JavaScript provideth:

```
function inDollarsAndCents () {
  "use strict";
  return '$' + this.dollars + '.' + (this.cents < 10 ? ('0' + this.cents) : this.\
cents);
};

var amount = {
  dollars: 2,
  cents: 56,
  toString: inDollarsAndCents
}

var amount2 = {
  dollars: 1,
  cents: 0,
  toString: inDollarsAndCents
}

amount.toString()
  //=> '$2.56'

amount2.toString()
  //=> '$1.00'
```

Within a function that is invoked as a property of an object, the environment contains a magic variable, `this`, that refers to the object.

`this` does not refer to the object if we invoke the function in another way. Recall that functions are values we can extract from an object without calling them:

```
var stringifier = amount.toString;

stringifier
  //=> [Function: inDollarsAndCents]
```

However, invoking a function without a property reference means that we have lost the context provided by `this`:[^global]

```
stringifier()
  //=>
    TypeError: Cannot read property 'dollars' of undefined
```

[^global] If we run without “strict” mode, we get a very unhelpful “global” context that often delays or obfuscates the source of bugs.

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[29] ("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:

[29] 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 allong = require('allong.es');
map = allong.es.map;

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.

```
var allong = require('allong.es');
var tap = allong.es.tap;

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.[30]

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`.[31]

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:

[30] 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.

[31] 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 t\
erm")

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 metaobject,s 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

[32]

introductory remarks coming soon

[32] Vacuum Pots (c) 2012 Olin Viydo, some rights reserved

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 allong = require('allong.es');
map = allong.es.map;

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 form 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[33], 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:[34]

[33] https://en.wikipedia.org/wiki/Observer_pattern

[34] 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;
    }
  }
});
```

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[35]

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[36]?

[35] With the greatest respect to Chongo, author of "The Homeless Interpretation of Quantum Mechanics."

[36] https://en.wikipedia.org/wiki/Gabriel_Garc%C3%ADa_Márquez

4. What do we do with maiden names[37] 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.

[37] 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.[^jargon]

Both methods and objects can and should be factored into reusable components that have a single, well-defined responsibility[38].[39]

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;
      }
    }
  }
});
```

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:"

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

[39] 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.

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

  // ...

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

The methods `setFirstName` and `setLastName` now call the helper method `isaValidName`. The usual motivation for this is known as DRY[40] 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 `isaValidString` helper method ensures that now there is just the 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:

[40] https://en.wikipedia.org/wiki/Don't_repeat_yourself

```
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[41] ("IIFE"). If our four helpers exist only to decompose `doSomethingComplicated`, we can write:

[41] 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 function 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 allong = require('allong.es');
var tap = allong.es.tap;

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.

Method Objects

If functions are objects, and functions can have properties, then functions can have methods:

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

K.length
  //=> 1

K.call(null, 'hello')
  //=> [Function]
```

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: []
  },
  push: {
    enumerable: false,
    writable: false,
    value: function (fn) {
      return this._functions.push(fn);
    }
  },
  unshift: {
    enumerable: false,
    writable: false,
    value: function (fn) {
      return this._functions.unshift(fn);
    }
  }
});
```

`sequencer` is an object-oriented way to implement the `sequence` function from function-oriented libraries like underscore[42] and allong.es[43]. Instead of writing something like:

[42] http://underscorejs.org
[43] http://allong.es

```
var allong = require('allong.es');
var sequence = allong.es.sequence;

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

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

We can write:

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

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

This gives us some additional flexibility that we will exploit more fully later on. But for now, the important idea is that a function can have properties of its own that it calls like helpers, and some of those properties can be collections. The collection properties can by dynamically updated after the fact.

What can we do with this?

Garnished Functions

Let's expand our sequencer to have *two* lists of functions and a "body:"

```
function garnished (arg) {
  var args = [].slice.call(arguments);

  garnished.befores.forEach( function(garnishing) {
    garnishing.apply(this, args);
  }, this);

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

  garnished.afters.forEach( function(garnishing) {
    garnishing.call(this, returnValue);
  }, this);

  return returnValue;
```

```
}

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

Our `garnished` function has an `unshift` and `push` method just like our `sequencer`, but it is arranged such that unshifting functions puts them on the head of a list of before functions, while pushing functions puts them on the tail of a list of after functions. right in the middle is a `body` function that defaults to doing nothing, but we can assign it like any other property:

```
garnished.body = function () { return 'i am a garnished function'; };
garnished.unshift(function () { console.log('before the body'); });
garnished.push(function () { console.log('after the body'); });

garnished()
  //=>
    before the body
    after the body
    'i am a garnished function'
```

Our function has a basic body that is responsible for the return value when called. It also can be "garnished" with functions to call before the body is evaluated and functions to call after the body has been evaluated.

Before we make another example, let's not type out all that code again. Of course we are going to discuss prototypes and classes and sharing behaviour later, but for now let's simply write a function that gives us garnished functions:

```
function garnishize (body) {

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

    garnished.befores.forEach( function (garnishing) {
      garnishing.apply(this, args);
    }, this);

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

    garnished.afters.forEach( function (garnishing) {
      garnishing.call(this, returnValue);
    }, this);

    return returnValue;
  }

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

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

  return garnished;
}

var double = garnishize(function (n) { return n * 2; });

double(2)
  //=> 4
```

One day we discover a bug in our code, someone seems to be passing something that isn't a number to `double`:

```
double('two')
  //=> NaN
```

It would be nice to have strong, static typing for such a problem, but the next best thing is to check our arguments. Here's a function that checks all of its arguments and throws an exception if any of them are not numbers:

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

  args.forEach(function (arg) {
    if (typeof(arg) !== 'number') throw ('Argument Error, "' + arg + '" is not a \
number');
  });
}
```

Instead of rewriting the code for `double`, we can just garnish it:

```
double.unshift(mustBeNumericArguments);

double(2)
  //=> 4

double('two')
  //=> Argument Error, "two" is not a number
```

This general pattern of "garnishing" functions with before and after functions is a long-standing pattern, going back to Lisp Machine Flavours[44], an early object-oriented system that featured before-, after-, and default "daemons."

Implementing garnishing with properties has more moving parts than wrapping functions with combinators, however it has the advantage of being reflective: We will see later on how to make garnishes play well with prototype chains.

[44] http://www.definitions.net/definition/Flavors

Metaobjects

 [45]

introductory remarks coming soon

[45] Krups Machines (c) 2010 Shadow Becomes White, some rights reserved

JavaScript's Constructors

Metaobjects have one key responsibility: *Defining base object behaviour*. Although it's not strictly required, most object-oriented languages accomplish this by having metaobjects also construct each object.

In classic JavaScript, any function can be used to construct a new object with the use of the `new` keyword. All functions have a `prototype` property even if they aren't intended to be used as constructors. You can change a function's prototype is you wish.

Functions used to create objects are called *constructors*, and in classic JavaScript the *constructor* is responsible for initializing new objects, while the *prototype* is responsible for the behaviour of the object. When we use JavaScript's `new` operator on a function, the prototype of the object is initialized automatically to be the prototype of the constructor. Since initialization and the prototype flow from the constructor, it is tempting to think of the constructor as the "Queen of all Things" in JavaScript.

```
var ClassicJSConstructor = function () {};
ClassicJSConstructor.prototype.identity = 'classic';
var classicObject = new ClassicJSConstructor();
Object.getPrototypeOf(classicObject)
  //=> { identity: 'classic' }
classicObject instanceof ClassicJSConstructor
  //=> true
```

If we follow that reasoning, we think of constructors as our metaobjects, and consider the prototype as part of the constructor. JavaScript encourages this perspective by providing an `instanceof` operator that appears to test whether an object was created by a particular constructor.[46]

constructors are not metaobjects

This thinking is mistaken. Constructors are not metaobjects. The `new` operator isn't even necessary to create objects:

```
var NakedPrototype = {
  identity: 'naked'
};
var unconstructedObject = Object.create(NakedPrototype);
Object.getPrototypeOf(unconstructedObject)
  //=> { identity: 'naked' }
```

[46] Most OO programmers prefer using polymorphism to explicitly testing `instanceof`. Wide use of explicit type testing is generally a design smell, but nevertheless it is a useful tool in some circumstances.

Furthermore, the `instanceof` operator doesn't do what it advertises. It appears to test whether a constructor created an object. But it actually tests whether an object and a function have compatible `prototype` properties.

Here we are fooling the operator:

```
var NeverConstructedAnything = function () {};
NeverConstructedAnything.prototype = NakedPrototype;
unconstructedObject instanceof NeverConstructedAnything
  //=> true
```

It turns out that `instanceof` is fine when there is a 1:1 correspondence between constructors and prototypes, when we do not change prototypes dynamically, and when we use `new` for all objects, eschewing `Object.create`. But the moment we venture into deeper waters, they the required workarounds outweigh their convenience.

If we want to test compatibility between an object and a prototype, we can and should do so directly:

```
NakedPrototype.isPrototypeOf(unconstructedObject)
  //=> true
```

The prototype always defines the behaviour of an object. JavaScript's "constructors," `new` operator, and its `instanceof` operator are convenient for programming within a narrow band of conventions, but are unreliable in the general case.

Therefore, although constructors can be used create objects, we consider the object's prototype to be central to the idea of a metaobject. We do not rely on the `instanceof` operator and if we wish to test for prototype compatibility, we use the `isPrototypeOf` method instead.

Classes and Prototypes

Although classes and prototypes both are responsible for creating objects and managing their shared behaviour, classes aren't prototypes and prototypes aren't classes. The simplest way to see the difference is to think about their methods. An object's methods are the surest clue to its function and responsibilities.

Let's revisit an example prototype:

```
function MovieCharacter (firstName, lastName) {
  this.firstName = firstName;
  this.lastName = lastName;
};

MovieCharacter.prototype.fullName = function () {
  return this.firstName + " " + this.lastName;
};
```

What are the prototype's methods?

```
Object.keys(MovieCharacter.prototype).filter(function (key) {
  return typeof(MovieCharacter.prototype[key]) === 'function'
});
  //=> [ 'fullName' ]
```

In JavaScript, `fullName` is a method of `MovieCharacter`'s prototype. The prototype's methods are the behaviour we're defining for `MovieCharacter` objects.

Now let's compare this to a "class." JavaScript doesn't have classes right out of the box, so we'll compare the prototype's methods to the methods of an equivalent Ruby class as an example:

```
class MovieCharacter

  def initialize(first_name, last_name)
    @first_name, @last_name = first_name, last_name
  end

  def full_name
    "#{first_name} #{last_name}"
  end

end
```

```
MovieCharacter.methods - Object.instance_methods
#=> [ :allocate, :new, :superclass, :<, :<=, :>, :>=, :included_modules, :include\
?,
      :name, :ancestors, :instance_methods, :public_instance_methods,
      :protected_instance_methods, :private_instance_methods, :constants, :const_\
get,
      :const_set, :const_defined?, :const_missing, :class_variables,
      :remove_class_variable, :class_variable_get, :class_variable_set,
      :class_variable_defined?, :public_constant, :private_constant, :module_exec,
      :class_exec, :module_eval, :class_eval, :method_defined?, :public_method_de\
fined?,
      :private_method_defined?, :protected_method_defined?, :public_class_method,
      :private_class_method, :autoload, :autoload?, :instance_method,
      :public_instance_method ]
```

In Ruby, `full_name` isn't a method of the `MovieCharacter` class, and unlike JavaScript, the class has lots and lots of methods that are specific to the business of being a meta-class that aren't shared by other objects.

JavaScript prototypes look just like ordinary objects, while Ruby classes don't look anything like ordinary objects. They're both metaobjects, but the two languages use completely different approaches. This is not surprising when you learn that Ruby was inspired by Smalltalk[47], a language that emphasized classes, while JavaScript was inspired by Self[48], a successor to Smalltalk that used prototypes instead of classes.

In some languages the difference between a class and a metaobjetc liek a prototype is even more pronounced. They have the notion of classes, but they don't have metaobjects you can access at runtime. C++[49], for example, allows you to define classes, the definitions are compiled into protocols for virtual functions that are late-bound, but there are no class objects in the system at run time. Such classes aren't metaobjects at all, so those classes are even more different than JavaScript's prototypes.

The takeaway is that there is no one "correct" design for metaobjects. The fundamental idea is that metaobjects manage creation and/or behaviour of a common set of objects, and that they are themselves objects a program can access and manipulate at runtime.

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

[48] https://en.wikipedia.org/wiki/Self_programming_language

[49] https://en.wikipedia.org/wiki/C%2B%2B

To Do

templates and creation by value

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

Protocols

[50]

introductory remarks coming soon

[50] Cime Espresso Machine Factory (c) 2012 Jong Hoon Lee, some rights reserved

To Do

prototype chaining

single inheritance

mixins and multiple inheritance

resolution: merge, override, and final

template method protocols

method guards and contracts

early and late method composition

multiple dispatch and generic functions

pattern matching protocols