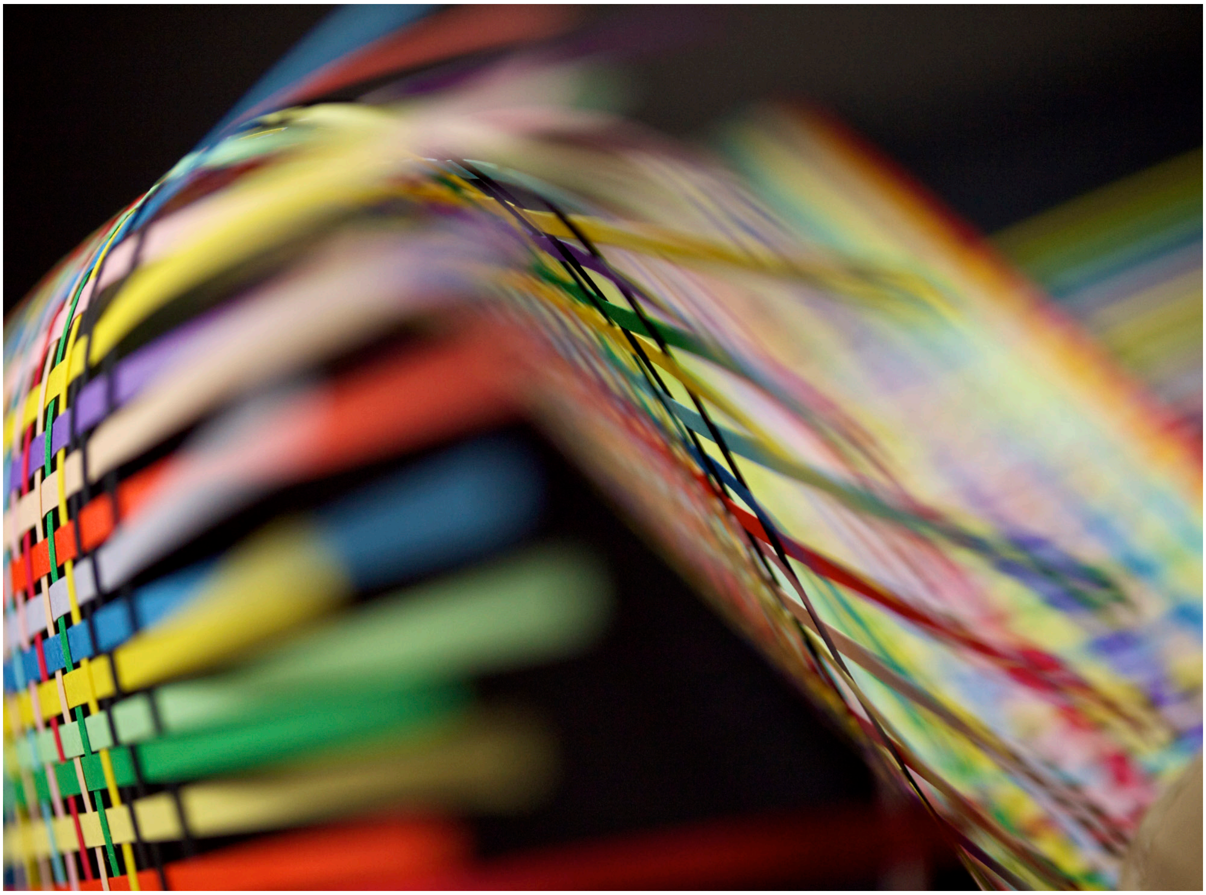
O'REILLY®

/ theory / in / practice

Beautiful JavaScript

Leading Programmers Explain How They Think

Anton Kovalyov



/ theory / in / practice

Beautiful JavaScript

JavaScript is arguably the most polarizing and misunderstood programming language in the world. Many have attempted to replace it as the language of the Web, but JavaScript has survived, evolved, and thrived. Why did a language created in such a hurry succeed where others failed?

This guide gives you a rare glimpse into JavaScript from people intimately familiar with it. Chapters contributed by domain experts such as Jacob Thornton, Ariya Hidayat, and Sara Chipps reveal what they love about their favorite language—whether it's turning the most feared features into useful tools, or how JavaScript can be used for self-expression.

Contributors include:

Jonathan Barronville
Sara Chipps
Angus Croll
Marijn Haverbeke
Ariya Hidayat
Daryl Koopersmith
Anton Kovalyov
Rebecca Murphey
Daniel Pupius
Graeme Roberts
Jenn Schiffer
Jacob Thornton
Ben Vinegar
Rick Waldron
Nicholas Zakas

About the editor:

Anton Kovalyov is a software engineer at Medium, creator of JSHint, and coauthor of *Third-Party JavaScript* (Manning).

"Reading this book is like sitting down with some of the masters of JavaScript for lunch and hearing them talk about what's on their mind at the moment. You'll leave with a new appreciation for the language, and with something you can use to make your next project better."

—Dave Camp, Director of Engineering, Firefox

US $39.99 CAN $45.99
ISBN: 978-1-449-37075-6

Twitter: @oreillymedia
facebook.com/oreilly
oreilly.com

Programming/JavaScript

Beautiful JavaScript

Edited by Anton Kovalyov

Beijing • Boston • Farnham • Sebastopol • Tokyo O'REILLY®

Beautiful JavaScript

edited by Anton Kovalyov

Published by O'Reilly Media, Inc., 1005 Gravenstein Highway North, Sebastopol, CA 95472.

O'Reilly books may be purchased for educational, business, or sales promotional use. Online editions are also available for most titles (*http://safaribooksonline.com*). For more information, contact our corporate/institutional sales department: 800-998-9938 or *corporate@oreilly.com*.

Acquisitions Editor: Simon St. Laurent
Editor: Allyson MacDonald
Production Editor: Matthew Hacker
Copyeditor: Rachel Head
Proofreader: Rachel Monaghan
Indexer: WordCo Indexing Services, Inc.
Interior Designer: David Futato
Cover Designer: Susan Thompson
Illustrator: Rebecca Demarest

August 2015: First Edition

Revision History for the First Edition

2015-08-07: First Release

See *http://oreilly.com/catalog/errata.csp?isbn=9781449370756* for release details.

978-1-449-37075-6

[LSI]

TABLE OF CONTENTS

Preface

FUNCTIONS ARE FIRST-CLASS CITIZENS, SYNTAX RESEMBLES JAVA, INHERITANCE is prototypal, and `(+"")` equals zero. This is JavaScript, arguably the most polarizing and misunderstood programming language in the world. It was created in 10 days and had a lot of warts and rough edges. Since then, there have been many attempts to replace it as the language of the Web. And yet, the language and the ecosystem around it are thriving. JavaScript is the most popular language in the world—and the only true language of the web platform. What made JavaScript special? Why did a language that was created in such a hurry succeed where others failed?

I believe the reasons why JavaScript (and the Web in general) survived lie in its omnipresence—it's practically impossible to find a personal computer that doesn't have some sort of JavaScript interpreter—and its ability to gain from disorder, to use its stressors for self-improvement.

JavaScript, like no other language, brought all kinds of different people to the platform. Anyone with a text editor and a web browser could get started with JavaScript, and many did. Its expressiveness and limited standard library prompted those people to experiment with the language and push it to its limits. People were not only making websites and applications; they were writing libraries and creating programming languages that could be compiled back into JavaScript. Those libraries competed with each other, and their approaches to solving problems often contradicted one another. The JavaScript ecosystem was a mess, but it was bursting with life.

Many of those libraries and languages are now forgotten. Their best ideas, however—the ones that proved themselves and stood the test of time—were absorbed into the language. They made their way into JavaScript's standard library and its syntax. They made the language better.

Then there were languages and technologies that were designed to replace JavaScript. But instead of succeeding, they unwillingly became its necessary stressors. Every time a new language or system to replace JavaScript emerged, browser vendors would find a way to make it faster, more powerful, and more robust. Once again, good ideas were absorbed into newer versions of the language, and the bad ones were discarded. These competing technologies didn't replace JavaScript; instead, they made it better.

Today, JavaScript is unbelievably popular. Will it last? I don't know. I cannot predict whether it will still be popular 5 or 10 years from now, but it doesn't really matter. For me, JavaScript will always be a great example of a language that survived not despite its flaws but because of them, and a language that brought people of so many different backgrounds into this wonderful world of computer programming.

About This Book

This book was written by people who are intimately familiar with the language. Each and every person who contributed a chapter is an expert in his or her domain. The authors highlight different sides of JavaScript, some of which you can discover only by writing lots of code, experimenting and making mistakes. As you make your way through this book, you'll get to see what JavaScript movers and shakers love about their favorite language.

You'll also learn a lot. I did. But do not mistake this book for a JavaScript tutorial, because it is much bigger than that. There are chapters that challenge the conventional wisdom and show how even the most feared features can be used as helpful tools. Some authors show that JavaScript can be a tool for self-expression and a form of art, while others share the wisdom of using JavaScript in codebases with hundreds of contributors. Some authors share personal stories, while others look into the future.

There's no common pattern that goes from one chapter to another—there's even a purely satirical chapter. This is intentional. I tried to give the authors as much freedom as possible to see what they would come up with, and they came up with something incredible. They came up with a book that resembles JavaScript itself, where each chapter is a reflection of its author.

Conventions Used in This Book

The following typographical conventions are used in this book:

Italic
: Indicates new terms, URLs, email addresses, filenames, and file extensions.

`Constant width`
: Used for program listings, as well as within paragraphs to refer to program elements such as variable or function names, databases, data types, environment variables, statements, and keywords.

TIP

This element signifies a tip or suggestion.

NOTE

This element signifies a general note.

Using Code Examples

Supplemental material (code examples, exercises, etc.) is available for download at *https://github.com/oreillymedia/beautiful_javascript*.

This book is here to help you get your job done. In general, if example code is offered with this book, you may use it in your programs and documentation. You do not need to contact us for permission unless you're reproducing a significant portion of the code. For example, writing a program that uses several chunks of code from this book does not require permission. Selling or distributing a CD-ROM of examples from O'Reilly books does require permission. Answering a question by citing this book and quoting example code does not require permission. Incorporating a significant amount of example code from this book into your product's documentation does require permission.

We appreciate, but do not require, attribution. An attribution usually includes the title, author, publisher, and ISBN. For example: "*Beautiful JavaScript*, edited by Anton Kovalyov (O'Reilly). Copyright 2015 Anton Kovalyov, 978-1-449-37075-6."

If you feel your use of code examples falls outside fair use or the permission given above, feel free to contact us at *permissions@oreilly.com*.

Safari® Books Online

How to Contact Us

Please address comments and questions concerning this book to the publisher:

O'Reilly Media, Inc.
1005 Gravenstein Highway North
Sebastopol, CA 95472
800-998-9938 (in the United States or Canada)
707-829-0515 (international or local)
707-829-0104 (fax)

We have a web page for this book, where we list errata, examples, and any additional information. You can access this page at *http://bit.ly/beautiful_javascript*.

To comment or ask technical questions about this book, send email to *bookquestions@oreilly.com*.

For more information about our books, courses, conferences, and news, see our website at *http://www.oreilly.com*.

Find us on Facebook: *http://facebook.com/oreilly*

Follow us on Twitter: *http://twitter.com/oreillymedia*

Watch us on YouTube: *http://www.youtube.com/oreillymedia*

CHAPTER ONE

Beautiful Mixins

Angus Croll

Developers love to create overly complex solutions to things that aren't really problems.

—Thomas Fuchs

In the beginning there was code, and the code was verbose, so we invented functions that the code might be reused. But after a while there were also too many functions, so we looked for a way to reuse those too. Developers often go to great lengths to apply "proper" reuse techniques to JavaScript. But sometimes when we try too hard to do the right thing, we miss the beautiful thing right in front of our eyes.

Classical Inheritance

Many developers schooled in Java, C++, Objective-C, and Smalltalk arrive at JavaScript with an almost religious belief in the necessity of the class hierarchy as an organizational tool. Yet humans are not good at classification. Working backward from an abstract superclass toward real types and behaviors is unnatural and restrictive—a superclass must be created before it can be extended, yet classes closer to the root are by nature more generic and abstract and are more easily defined *after* we have more knowledge of their concrete subclasses. Moreover, the need to tightly couple types *a priori* such that one type is always defined solely in terms of another tends to lead to an overly rigid, brittle, and often ludicrous model ("Is a button a rectangle or is it a control? Tell you what, let's make `Button` inherit from `Rectangle`, and `Rectangle` can inherit from `Control`…no, wait a minute…"). If we don't get it right early on, our system is forever burdened with a flawed set of relationships—and on those rare occasions that, by chance or genius, we do get it right, anything but a minimal tree structure usually represents too complex a mental model for us to readily visualize.

Classical inheritance is appropriate for modeling existing, well-understood hierarchies—it's okay to unequivocally declare that a `FileStream` is a type of `InputStream`. But if the primary motivation is function reuse (and it usually is), classical hierarchies can quickly become gnarly labyrinths of meaningless subtypes, frustrating redundancies, and unmanageable logic.

Prototypes

It's questionable whether the majority of behaviors can ever be mapped to objectively "right" classifications. And indeed, the classical inheritance lobby is countered by an equally fervent band of JavaScript *loyalists* who proclaim that JavaScript is a prototypal, not classical, language and is deeply unsuited to any approach that includes the word *class*. But what does "prototypal" mean, and how do prototypes differ from classes?

In generic programming terms, a prototype is an object that supplies base behavior to a second object. The second object can then extend this base behavior to form its own specialization. This process, also known as *differential inheritance*, differs from classical inheritance in that it doesn't require explicit typing (static or dynamic) or attempt to formally define one type in terms of another. While classical inheritance is planned reuse, true prototypal inheritance is opportunistic.

> In general, when working with prototypes, one typically chooses not to categorize but to exploit alikeness.
>
> —Antero Taivalsaari, Nokia Research Center

In JavaScript, every object references a prototype object from which it can inherit properties. JavaScript prototypes are great instruments for reuse: a single prototype instance can define properties for an infinite number of dependent instances. Prototypes may also inherit from other prototypes, thus forming prototype chains.

So far, so good. But, with a view to emulating Java, JavaScript tied the `prototype` property to the constructor. As a consequence, more often than not, multilevel object inheritance is achieved by chaining constructor-prototype couplets. The standard implementation of a JavaScript prototype chain is too grisly to appear in a book about beautiful JavaScript, but suffice it to say, creating a new instance of a base prototype in order to define the initial properties of its inheritor is neither graceful nor intuitive. The alternative—manually copying properties between prototypes and then meddling with the `constructor` property to fake real prototypal inheritance—is even less becoming.

Syntactic awkwardness aside, constructor-prototype chaining requires upfront planning and results in structures that more closely resemble the traditional hierarchies of classical languages than a true prototypal relationship: constructors represent types (classes), each type is defined as a subtype of one (and only one) supertype, and all properties are inherited via this type chain. The ES6 `class` keyword merely formalizes the existing semantics. Leaving aside the gnarly and distinctly unbeautiful syntax characteristic in constructor-prototype chains, traditional JavaScript is clearly less prototypal than some would claim.

In an attempt to support less rigid, more opportunistic prototypes, the ES5 specification introduced `Object.create`. This method allows a prototype to be assigned to an object directly and therefore liberates JavaScript prototypes from constructors (and thus categorization) so that, in theory, an object can acquire behavior from any other arbitrary object and be free from the constraints of typecasting:

```
var circle = Object.create({
  area: function() {
    return Math.PI * this.radius * this.radius;
  },
  grow: function() {
    this.radius++;
  },
  shrink: function() {
    this.radius--;
  }
});
```

`Object.create` accepts an optional second argument representing the object to be extended. Sadly, instead of accepting the object itself (in the form of a literal, variable, or argument), the method expects a full-blown `meta` property definition:

```
var circle = Object.create({
  /*see above*/
}, {
  radius: {
    writable:true, configurable:true, value: 7
  }
});
```

Assuming no one actually uses these unwieldy beasts in real code, all that remains is to manually assign properties to the instance after it has been created. Even then, the `Object.create` syntax still only enables an object to inherit the properties of a single prototype. In real scenarios, we often want to acquire behavior from multiple prototype objects: for example, a person can be an employee and a manager.

Mixins

Fortunately, JavaScript offers viable alternatives to inheritance chaining. In contrast to objects in more rigidly structured languages, JavaScript objects can invoke any function property regardless of lineage. In other words, JavaScript functions don't need to be inheritable to be visible—and with that simple observation, the entire justification for inheritance hierarchies collapses like a house of cards.

The most basic approach to function reuse is manual delegation—any public function can be invoked directly via `call` or `apply`. It's a powerful and easily overlooked feature. However, aside from the verbosity of serial `call` or `apply` directives, such delegation is so convenient that, paradoxically, it sometimes actually works against structural discipline in your code—the invocation process is sufficiently ad hoc that in theory there is no need for developers to organize their code at all.

Mixins are a good compromise: by encouraging the organization of functionality along thematic lines they offer something of the descriptive prowess of the class hierarchy, yet they are light and flexible enough to avoid the premature organization traps (and head-spinning dizziness) associated with deeply chained, single-ancestry models. Better still, mixins require minimal syntax and play very well with unchained JavaScript prototypes.

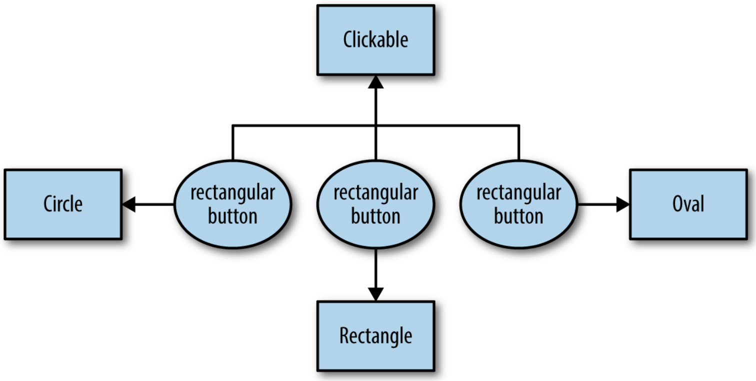
The Basics

Traditionally, a mixin is a class that defines a set of functions that would otherwise be defined by a concrete entity (a person, a circle, an observer). However, mixin classes are considered abstract in that they will not themselves be instantiated—instead, their functions are copied (or *borrowed*) by concrete classes as a means of inheriting behavior without entering into a formal relationship with the behavior provider.

Okay, but this is JavaScript, and we have no classes per se. This is actually a good thing because it means we can use objects (instances) instead, which offer clarity and flexibility: our mixin can be a regular object, a prototype, a function, whatever, and the mixin process becomes transparent and obvious.

The Use Case

I'm going to discuss a number of mixin techniques, but all the coding examples are directed toward one use case: creating circular, oval, or rectangular buttons (something that would not be readily possible using conventional classical inheritance techniques). Here's a schematic representation: square boxes represent mixin objects, and rounded boxes represent the actual buttons.



Classic Mixins

Scanning the first two pages returned from a Google search for "javascript mixin," I noticed the majority of authors define the mixin object as a full-blown constructor type with its function set defined in the prototype. This could be seen as a natural progression—early mixins were classes, and this is the closest thing JavaScript has to a class. Here's a circle mixin modeled after that style:

```
var Circle = function() {};
Circle.prototype = {
  area: function() {
    return Math.PI * this.radius * this.radius;
  },
  grow: function() {
    this.radius++;
  },
  shrink: function() {
    this.radius--;
  }
};
```

In practice, however, such a heavyweight mixin is unnecessary. A simple object literal will suffice:

```
var circleFns = {
  area: function() {
    return Math.PI * this.radius * this.radius;
  },
  grow: function() {
    this.radius++;
  },
  shrink: function() {
    this.radius--;
  }
};
```

Here's another mixin defining button behavior (for the sake of demonstration, I've substituted a simple log call for the working implementation of some function properties):

```
var clickableFns = {
  hover: function() {
    console.log('hovering');
  },
  press: function() {
    console.log('button pressed');
  },
  release: function() {
    console.log('button released');
  },
  fire: function() {
    this.action.fire();
  }
};
```

The extend Function

How does a mixin object get mixed into your object? By means of an `extend` function (sometimes known as *augmentation*). Usually `extend` simply copies (not clones) the mixin's functions into the receiving object. A quick survey reveals some minor variations in this implementation. For example, the Prototype.js framework omits a `hasOwnProperty` check (suggesting the mixin is not expected to have enumerable properties in its prototype chain), while other versions assume you want to copy only the mixin's prototype object. Here's a version that is both safe and flexible:

```
function extend(destination, source) {
  for (var key in source) {
    if (source.hasOwnProperty(key)) {
      destination[key] = source[key];
    }
  }
  return destination;
}
```

Now let's extend a base prototype with the two mixins we created earlier to make a `RoundButton.prototype`:

```
var RoundButton = function(radius, label) {
  this.radius = radius;
  this.label = label;
};

extend(RoundButton.prototype, circleFns);
extend(RoundButton.prototype, clickableFns);

var roundButton = new RoundButton(3, 'send');
```

```
roundButton.grow();
roundButton.fire();
```

Functional Mixins

If the functions defined by mixins are intended solely for the use of other objects, why bother creating mixins as regular objects at all? Isn't it more intuitive to think of mixins as processes instead of objects? Here are the circle and button mixins rewritten as functions. We use the context (`this`) to represent the mixin's target object:

```
var withCircle = function() {
  this.area = function() {
    return Math.PI * this.radius * this.radius;
  };
  this.grow = function() {
    this.radius++;
  };
  this.shrink = function() {
    this.radius--;
  };
};

var withClickable = function() {
  this.hover = function() {
    console.log('hovering');
  };
  this.press = function() {
    console.log('button pressed');
  };
  this.release = function() {
    console.log('button released');
  };
  this.fire = function() {
    this.action.fire();
  };
}
```

And here's our `RoundButton` constructor. We'll want to apply the mixins to `RoundButton.prototype`:

```
var RoundButton = function(radius, label, action) {
    this.radius = radius;
    this.label = label;
    this.action = action;
};
```

Now the target object can simply inject itself into the functional mixin by means of `Function.prototype.call`, cutting out the middleman (the `extend` function) entirely:

```
withCircle.call(RoundButton.prototype);
withClickable.call(RoundButton.prototype);

var button1 = new RoundButton(4, 'yes!', function() {return 'you said yes!'});
button1.fire(); //'you said yes!'
```

This approach feels right. Mixins as verbs instead of nouns; lightweight one-stop function shops. There are other things to like here too. The programming style is natural and concise: `this` always refers to the receiver of the function set instead of an abstract object we don't need and will never use; moreover, in contrast to the traditional approach, we don't have to protect against inadvertent copying of inherited properties, and (for what it's worth) functions are now cloned instead of copied.

Adding Options

This functional strategy also allows mixed in behaviors to be parameterized by means of an `options` argument. The following example creates a `withOval` mixin with a custom grow and shrink factor:

```
var withOval = function(options) {
  this.area = function() {
    return Math.PI * this.longRadius * this.shortRadius;
  };
  this.ratio = function() {
    return this.longRadius/this.shortRadius;
  };
  this.grow = function() {
    this.shortRadius += (options.growBy/this.ratio());
    this.longRadius += options.growBy;
  };
  this.shrink = function() {
    this.shortRadius -= (options.shrinkBy/this.ratio());
    this.longRadius -= options.shrinkBy;
  };
}

var OvalButton = function(longRadius, shortRadius, label, action) {
  this.longRadius = longRadius;
  this.shortRadius = shortRadius;
  this.label = label;
  this.action = action;
};

withButton.call(OvalButton.prototype);
withOval.call(OvalButton.prototype, {growBy: 2, shrinkBy: 2});

var button2 = new OvalButton(3, 2, 'send', function() {return 'message sent'});
button2.area(); //18.84955592153876
button2.grow();
button2.area(); //52.35987755982988
button2.fire(); //'message sent'
```

Adding Caching

You might be concerned that this approach creates additional performance overhead because we're redefining the same functions on every call. Bear in mind, however, that when we're applying functional mixins to prototypes, the work only needs to be done once: during the definition of the constructors. The work required for instance creation is unaffected by the mixin process, since all the behavior is preassigned to the shared prototype. This is how we support all function sharing on the *twitter.com* site, and it produces no noticeable latency. Moreover, it's worth noting that performing a classical mixin requires property getting as well as setting, and in fact functional mixins appear to benchmark quicker in the Chrome browser than traditional ones (although this is obviously subject to considerable variance).

That said, it is possible to optimize functional mixins further. By forming a closure around the mixins we can cache the results of the initial definition run, and the performance improvement is impressive. Functional mixins now easily outperform classic mixins in every browser.

Here's a version of the `withRectangle` mixin with added caching:

```
var withRectangle = (function() {
  function area() {
    return this.length * this.width;
  }
  function grow() {
    this.length++, this.width++;
  }
  function shrink() {
    this.length--, this.width--;
  }
  return function() {
    this.area = area;
    this.grow = grow;
    this.shrink = shrink;
    return this;
  };
})();

var RectangularButton = function(length, width, label, action) {
  this.length = length;
  this.width = width;
  this.label = label;
  this.action = action;
}

withClickable.call(RectangularButton.prototype);
withRectangle.call(RectangularButton.prototype);

var button3 =
  new RectangularButton(4, 2, 'delete', function() {return 'deleted'});
```

```
button3.area(); //8
button3.grow();
button3.area(); //15
button3.fire(); //'deleted'
```

Advice

One danger with any kind of mixin technique is that a mixin function will accidentally overwrite a property of the target object that, coincidentally, has the same name. Twitter's Flight framework, which makes use of functional mixins, guards against clobbering by temporarily locking existing properties (using the `writable` meta property) during the mixin process.

Sometimes, however, instead of generating a collision error we might want the mixin to augment the corresponding method on the target object. `advice` redefines a function by adding custom code before, after, or around the original implementation. The Underscore framework implements a basic function wrapper that enables `advice`:

```
button.press = function() {
  mylib.appendClass('pressed');
};

//after pressing button, reduce shadow (using underscore)
button.pressWithShadow = _.wrap(button.press, function(fn) {
  fn();
  button.reduceShadow();
}
```

The Flight framework takes this a stage further: now the `advice` object is itself a functional mixin that can be mixed into target objects to enable advice for subsequent mixins.

Let's use this `advice` mixin to augment our rectangular button actions with shadow behavior. First we apply the `advice` mixin, followed by the two mixins we used earlier:

```
withAdvice.call(RectangularButton.prototype);
withClickable.call(RectangularButton.prototype);
withRectangle.call(RectangularButton.prototype);
```

And now the `withShadow` mixin that will take advantage of the `advice` mixin:

```
var withShadow = function() {
  this.after('press', function() {
    console.log('shadow reduced');
  };
  this.after('release', function() {
    console.log('shadow reset');
  };
};

withShadow.call(RectangularButton.prototype);
```

```
var button4 = new RectangularButton(5, 4);
button4.press(); //'button pressed' 'shadow reduced'
button4.release(); //'button released' 'shadow reset'
```

The Flight framework sugarcoats this process. All flight components get `withAdvice` mixed in for free, and there's also a `defineComponent` method that accepts multiple mixins at a time. So, if we were using Flight we could further simplify the process (in Flight, constructor properties such as rectangle dimensions are defined as `attr` properties in the mixins):

```
var RectangularButton =
  defineComponent(withClickable, withRectangle, withShadow);
var button5 = new RectangularButton(3, 2);
button5.press(); //'button pressed' 'shadow reduced'
button5.release(); //'button released' 'shadow reset'
```

With `advice` we can define functions on mixins without having to guess whether they're also implemented on the target object, so the mixin can be defined in isolation (perhaps by another vendor). Conversely, `advice` allows us to augment third-party library functions without resorting to monkey patching.

Wrapup

> When possible, cut with the grain. The grain tells you which direction the wood *wants* to be cut. If you cut against the grain, you're just making more work for yourself, and making it more likely you'll spoil the cut.
>
> —Charles Miller[1]

As programmers, we're encouraged to believe that certain techniques are indispensable. Ever since the early 1990s, object-oriented programming has been hot, and classical inheritance has been its poster child. It's not hard to see how a developer eager to master a new language would feel under considerable pressure to fit classical inheritance under the hood.

But peer pressure is not an agent of beautiful code, and neither is serpentine logic. When you find yourself writing `Circle.prototype.constructor = Circle`, ask yourself if the pattern is serving you, or you're serving the pattern. The best patterns tread lightly on your process and don't interfere with your ability to use the full power of the language.

By repeatedly defining an object solely in terms of another, classical inheritance establishes a series of tight couplings that glue the hierarchy together in an orgy of mutual dependency. Mixins, in contrast, are extremely agile and make very few organizational

1 See Charles Miller's entire post at his blog, The Fishbowl.

demands on your codebase—mixins can be created at will, whenever a cluster of common, shareable behavior is identified, and all objects can access a mixin's functionality regardless of their role within the overall model. Mixin relationships are entirely ad hoc: any combination of mixins can be applied to any object, and objects can have any number of mixins applied to them. Here, at last, is the opportunistic reuse that prototypal inheritance promised us.

CHAPTER TWO

eval and Domain-Specific Languages

Marijn Haverbeke

`eval` is a language construct that takes a string and executes it as code.

This means that in a language with an `eval` construct, the code that is being executed can come not just from input files, but also from the running code itself.

There are several reasons why this is interesting and useful. In this chapter, I will explore the degree to which JavaScript's `eval` can be used to create simple language-based abstractions.

What About "eval Is Evil"?

I know that some of my readers, at the mention of the word `eval`, are feeling the adrenaline shoot into their veins, and hearing the solemn voice of a certain bearded JavaScript evangelist boom in the back of their heads. *"eval is evil!"* this voice proclaims.

I've never found absolute moral judgments very applicable in engineering. But if you do, and don't want to reevaluate your faith, feel free to skip this chapter.

Practically speaking, there are a number of problematic issues that come up when `eval` is used. Its semantics are confusing and error-prone, and its impact on performance is not always obvious. I'm going to approach it as a tool, and try to clarify and study these issues, in order to help you use the tool effectively.

History and Interface

An interpreter (in the broad sense of the word) for a language is a program that takes text and executes it as code. When you have an interpreter available, exposing it as an `eval` construct, which does pretty much the same thing, is easy and obvious.

The first language to do this was an early dialect of Lisp. More recent dynamic languages—Perl, Python, PHP, Ruby, and of course JavaScript—followed suit. Most of these languages went through a similar process, where they initially introduced a straightforward, naive evaluation construct, and later tried to refine, extend, or disable it as a form of damage control.

The subtlety in designing an interface for code execution lies in the environment in which the code is to be interpreted—the question of which variables it can see. In a primitive interpreter, which often represents variables in a way that makes it easy to inspect and manipulate them, it is no problem to give evaluated code full access to all the variables that are visible at the point where the `eval` construct is used. The initial design of a dynamic language is often intertwined with the first implementation of its interpreter, and this makes it tempting to go with the model where the evaluated code has access to the local environment.

There are two reasons why this is problematic. Firstly, there's rarely a reason to *want* to access local scope. You'll occasionally see some confused JavaScript programmers do something like `eval("obj." + propertyName)` because they fail to realize that the language allows dynamic property access, or `eval("var result = " + code)` because they are ignorant of the fact that `eval` already returns the result of the evaluation, and the `var result =` part could be lifted out. When the code string comes from an external source, there's also the risk of a variable in the string accidentally using a variable name that is also defined locally, causing a conflict between the two uses. The one case where access to a local scope is not completely wrongheaded is when evaluated code needs to have access to utility functions defined in the module that evaluates it. We'll see a decent way to work around that later.

The second reason that evaluating in the local scope is not a good idea is that it makes life quite a bit harder for the compiler. Knowing exactly what the code it's compiling looks like enables a compiler to make a lot of decisions at compile time (rather than runtime), which makes the code it produces faster. Most importantly, if it knows a variable `x` refers to a specific `x` variable defined either globally or in one of its enclosing scopes, it can generate very simple code to access this `x`. An `eval` could introduce a new variable `x`, forcing the compiler to represent its environment in a more complex way and to output more expensive code for each variable access.

And this last point is the reason for the *very* odd way in which JavaScript `eval` behaves—the distinction between local and global evaluation.

`eval` is, historically, a regular global variable that holds a function. That means you can do everything with it that you can do with other values—store it in another variable or in a data structure, pass it to a function, and so on. But because the people trying to optimize JavaScript execution did not want to represent all environments and variable accesses in the expensive, dynamic way I described previously, they introduced a subtle rule, probably initially as a hack, that was later standardized into ECMAScript.

This rule is: the `eval` is only done in the local scope if we can see, during compilation, that a call to `eval` takes place—there has to be a function call to the actual global variable named `eval` in the code (and this global must still have its original value). If you call `eval` in any other, more indirect way, it will not have access to the local scope, and thus will be a global evaluation.

For example, `eval("foo")` is local, while `(0 || eval)("foo")` is global, and so is `var lave = eval; lave("foo")`.

Though this was conceived purely as an efficiency kludge, not as an attempt to provide a better interface, people have been intentionally making use of it, since global evaluation is often more useful and less error-prone than local evaluation.

Another variant of global evaluation is the `Function` constructor. It takes strings for the argument names and function body as arguments, and returns a function in the context of the global scope (it does not close over variables in the scope where it was created). Note that the argument names can be passed either as separate arguments (`new Function("a", "b", "return a + b")`) or as a single comma-separated string (`new Function("a, b", "return a + b")`). For most purposes, this is the preferred way to evaluate code.

Performance

Evaluating code is expensive. Not only does the JavaScript compiler have to be invoked to compile the code, but modern JavaScript engines also tend to perform analysis on the loaded program in order to perform certain optimizations. Introducing new code can invalidate the results of such analysis, and cause recompilation of other parts of the program.

Evaluation in local scope is extra worrying, for the reasons discussed before. I ran a number of benchmarks on modern JavaScript engines, and found that variable access that goes through a scope that can be accessed by a local `eval` form is significantly slower. This means that if you're using the closure module pattern (an anonymous function as module scope), having a local evaluation anywhere in your module will incur a cost for all code in the module. The scope just needs to *have* such a call—it doesn't even have to execute it—to incur this cost.

On the other hand, the speed of a function created by `new Function` or a global `eval` is not adversely affected by the fact that it was created dynamically.

So, a desirable pattern is one where the evaluation happens once (at program startup), or outside of hot loops (we're talking about few-millisecond delays here, not interface-freezing disasters). The functions generated by the evaluation can then be used as intensively as needed.

Common Uses

The most obvious use of `eval` is dynamically running code from an external source: for example, in a module-manager library that fetches code from somewhere and then uses a global `eval` to inject it into the environment, or an interactive *repl* (read-eval-print loop) that executes code that the user types.

In the past, `eval` was the easiest way to parse strings of JSON data, whose representation is a subset of JavaScript's own syntax. In modern implementations we have `JSON.parse` for that, which has the significant advantage of not enabling code injection attacks when parsing untrusted data.

Most JavaScript-based text templating systems use some form of `eval` to precompile templates. They parse the template text once, produce a program that instantiates the template, and use `eval` to have the JavaScript compiler compile that. In some cases this is simply an optimization, but in others the templates may contain JavaScript code, so some form of `eval` *has* to be involved. We'll go over the compiler for a simple JavaScript-based templating language in the next section.

A template is a kind of domain-specific language (DSL), a language designed to solve a specific problem (in this case, building up strings) by being specialized to express the elements of that problem more directly than plain JavaScript. Domain-specific languages are a more interesting application of `eval`. We'll cover another one, a compact and efficient notation for matching and extracting binary data, later on in this chapter.

A Template Compiler

Before you look at the code that follows, I should warn you. You opened a book called *Beautiful JavaScript*, and I'm about to confront you with some rather *ugly* code. That may seem disingenuous.

Code that builds up strings of code tends to look bad. If we had string interpolation, a code-oriented templating system, or even a data structure that represented code, things might be slightly better. But as it is, we'll be crudely concatenating lots of strings, many of them containing the same keywords and syntactic patterns as the code around them. This does not make for very elegant or readable code.

The function shown here accepts a template string as an argument and returns a function that represents a compiled version of this template. It recognizes templating directives written between hash signs. Here's an example of a trivial template that it parses:

```
#$in.title#
==============

Items on today's list:
#for item in $in.items#
  * #item.name##if item.note# (Note: #item.note#) #end#
#end#
```

A directive starting with `for` opens a loop (over an array). An `if` directive opens a conditional. Both are closed by an `end` directive. Anything else is interpreted as a value that should simply be inserted as text into the output. The variable `$in` is used to refer to the value passed into the template.

For brevity, the code does no input checking whatsoever. Here's the implementation of that function:

```
function compile(template) {
  var code = "var _out = '';", uniq = 0;
  var parts = template.split("#");
  for (var i = 0; i < parts.length; ++i) {
    var part = parts[i], m;
    if (i % 2) { // Odd elements are templating directives
      if (m = part.match(/^for (\S+) in (.*)/)) {
        var loopVar = m[1], arrayExpr = m[2];
        var indexVar = "_i" + (++uniq), arrayVar = "_a" + uniq;
        code += "for (var " + indexVar + " = 0, " + arrayVar + " = " +
          arrayExpr + ";" + indexVar + "<" + arrayVar + ".length; ++" +
          indexVar + ") {" + "var " + loopVar + " = " + arrayVar +
          "[" + indexVar + "];";
      } else if (m = part.match(/^if (.*)/)) {
        code += "if (" + m[1] + ") {";
      } else if (part == "end") {
        code += "}";
      } else {
        code += "_out += " + part + ";";
      }
    } else if (part) { // Even elements are plain text
      code += "_out += " + JSON.stringify(part) + ";";
    }
  }
  return new Function("$in", code + "return _out;");
}
```

To locate the directives, the function simply splits the template on hash characters, and considers the even-numbered parts to be plain text and the odd-numbered elements (the parts that appear between hash characters) as templating directives. Regular expressions are used to recognize the `if` and `for` directives.

The _out variable in the generated code is used to build up the output string. The underscore is an attempt to avoid name clashes, since we'll be mixing generated code with code found in the template.

To build a loop for a for directive, we need to introduce two additional variables into the generated code—one for the index and one to hold the array. We need a variable that holds the array to ensure that whatever expression is used to produce it is not evaluated repeatedly, since it might be expensive to compute or have side effects. In order to make sure that these variable names do not clash, even for nested loops, a counter (uniq) is added to the variable name (_i1, _i2, etc.).

Finally, the Function constructor is used to create a function with our generated code as the body and a single argument, $in.

If we feed the template compiler the example template, it will spit out a function like this (whitespace added):

```
function($in) {
  var _out = '';
  _out += $in.title;
  _out += "\n=============\n\nItems on today's list:\n";
  for (var _i1 = 0, _a1 = $in.items; _i1 < _a1.length; ++_i1) {
    var item = _a1[_i1];
    _out += "\n  * ";
    _out += item.name;
    if (item.note) {
      _out += " (Note: ";
      _out += item.note;
      _out += ") "
    }
  }
  return _out;
}
```

We could make that code cleaner by adding some intelligence to the compiler (for example, it could combine subsequent += statements to simply use +), but you can see how it expresses the steps needed to instantiate the template.

With a few extensions, such as the option to escape the inserted strings for your output format of choice (HTML, for example), and some error checking, this code can be built into a practical templating engine.

Speed

It is always possible to interpret a domain-specific language on demand. But just as compilers tend to run programs faster than interpreters, precompiling a template leads to faster instantiation than interpreting it from its source every time it is instantiated.

If we forget for a second that the templating language contains JavaScript code, it would be possible to do a form of compilation without `new Function`—we could parse the template, and build up a data structure that allows us to instantiate it quickly with little repeated work. But it'd take a lot of effort to come close to the speed of the preceding approach that way.

The JavaScript compiler is much more powerful (and has more direct access to the machine) than our puny compiler, so by first translating to JavaScript and then handing off the rest of the work to its more advanced peer, we can get good results with very little work.

This idea of building on top of a compiler for another language in order to run your own language or notation is widely applicable. The various compile-to-JavaScript languages make use of it. But it also works well on a smaller scale, such as for writing a tiny compiler for a simple language to solve a very specific problem.

Mixing Languages

Let's look a bit more at the fact that the templates in the toy templating language *contain* JavaScript code. They are, in a way, JavaScript programs with a syntactic extension that optimizes them for text expansion.

Whether this is a good idea is a question that can be answered in several ways. If you don't trust the source of your templates, or you want to expand the templates in an environment that doesn't run JavaScript, then it is definitely a bad idea. The authors of the templates can inject arbitrary code into your program, and expanding these templates in, for example, a Ruby program would be awkward.

But we do get the full expressive power of a real programming language in our templates. The alternative would be to define a simple expression language as part of the templating language, parse that, and either interpret it during expansion or convert it to the output language (JavaScript, in our case). This approach has its own problems, though. It's more work, obviously. But it is also hard to find a balance between offering enough features to allow people to do what they need to do without the language becoming huge and complex.

We already know JavaScript, so if we wanted, in the example template, to render only items whose `category` property contains the string `important`, we could simply type `#if /\bimportant\b/.test(item.category)#`. If we had to express that in a sublanguage, we'd either be out of luck if the language didn't have string search, or need to first spend 10 minutes digging through documentation to figure out how to express string search in the language.

(Tangentially related is the argument that templating languages should be weak because they should contain presentation logic only. My take on that is that, firstly, presentation logic can get quite complicated, and secondly, taking away my hammer to ensure that I don't use it on screws is a lousy way of enforcing good style.)

A tricky issue that comes up when you're mixing languages is "hygiene." The generated code and the code that appeared in the template both run in the same scope. Thus, there is a danger that the two sources of code will disagree on what a certain variable name refers to. The toy template compiler generates variables like `_a3` to avoid accidentally clashing with variables from the included code. This mostly works, but is of course far from perfect (`#for _a1 in [1, 2, 3]#` causes a clash). You could use more obscure variable names (`_$$_o_O_a3`) to further reduce the chance of clashes, but it'll never be elegant. Languages that use this kind of metaprogramming more intensively have mechanisms to cope with these kinds of problems. JavaScript doesn't, but because its metaprogramming support is so minimal, that's usually not a problem.

Dependencies and Scopes

Since the toy template compiler used `new Function` to evaluate its code, that code will only be able to see the global scope.

What if the code that sits in the template needs access to, for example, a date formatting function? Or what if the generated part of the code needs an HTML escaping function to escape the dynamic parts of the output? You could put them in the global scope, but if you're using modern, disciplined scoping in the style of CommonJS (Node.js) or RequireJS modules, that would be unfortunate.

The key to a workable solution to this problem is that, though we can't control what the generated function itself closes over, we *can* wrap our result function in an additional function, and thus inject stuff for it to close over.

Here's a crude utility that does this:

```
function newFunctionWith(env, args, body) {
  var code = "";
  for (var prop in env)
    code += "var " + prop + " = $$env." + prop + ";";
  code += "return function(" + args + ") {" + body + "};";
  return new Function("$$env", code)(env);
}

console.log(newFunctionWith({x: 10}, "y", "return x + y;")(20));
// → 30
```

Given an object mapping variables to values, an argument list string, and a function body string, this helper acts like `new Function(args, body)`, except that it makes sure that all the properties in the `env` object are visible as closed-over variables to the body of the function.

It does this by generating a wrapping function that *unpacks* its argument into local variables, and then, immediately after evaluating this function, calling it. For simple values like integers, it could also have inserted the string form of the value directly into the wrapping function (`var x = 10;`). However, that doesn't work for complex values, so we need to pass the environment object to the evaluated code, allowing it to extract the actual values from that object.

Using this utility, the templating system could do something like allowing templates to declare their dependencies and `require`-ing those in, making the code close over them.

Debugging Generated Code

Debugging generated code is rarely a pleasant experience. When you write a compiler like the one we just looked at, and try it out, you will most likely be greeted by some kind of syntax error. Details differ between JavaScript engines, but if this error has origin information at all, it'll often point to the line that did the evaluation, not to the generated code.

So what now? Unfortunately, there's no good answer that I know of. One approach is to make your compiler function log the code before it evaluates it, autoformat it, put it in a file, and try to load it. Then, the error will at least point to the actual place where the code is broken.

If it's not a syntax error but a logic error, this might not be necessary—you might just be able to insert `console.log` or `debugger` statements into your generated code.

Where it gets really bad is when, as in the templating system I discussed, code from the input is mixed into the generated code. Debugging a compiler once is one thing. Getting strange, contextless exceptions whenever you make a typo in your template can ruin your whole day. For production-strength systems, you probably want serious syntax checking of your templates. There are a variety of good JavaScript parsers (written in JavaScript) available nowadays, and they can be used to properly parse the expressions or statements you expect in your template, at compile time. This also helps to determine their extent in a reliable way (a directive like `#if $in.type == "#" #` would not parse in the code shown earlier, because it doesn't understand that the second hash sign is quoted), and would make it possible to emit a meaningful error (including the template name and line offset) when nonsense is encountered.

Binary Pattern Matches

The second example I want to show you largely follows the same pattern as the first: we compile a domain-specific language down to JavaScript, in order to gain both speed and expressivity.

There is a feature in the Erlang programming language that allows you to pattern-match against binary data by specifying a sequence of fields and, for each field, a variable name or constant. Variables will be bound to the content of the field, and constants will be compared to the content of the field in order to determine whether the pattern matches. This provides a very convenient way of checking and extracting data from binary blobs.

Let's say we want something like this in JavaScript. Ideally, it'd look like this:

```
function gifSize(bytes) {
  binswitch (bytes) {
    case <<"GIF89a" width:uint16 height:uint16>>:
      return {width: width, height: height};
    default:
      throw new Error("not a GIF file");
  }
}
```

where `binswitch` is like `switch`, except that it matches a series of fields in the given chunk of binary data (a typed array, presumably). This pattern would mean "first the bytes corresponding to the string `"GIF89a"`, then a two-byte unsigned integer, which is bound to `width`, and finally another unsigned integer bound to `height`." Patterns that bind variables like that are found in many modern programming languages, and are a very pleasant feature.

If you're willing to do heavyweight full-file preprocessing, you could write your own JavaScript dialect in which this code is valid. But in this chapter, we're looking for lightweight tricks, not alternative languages. We need to find some kind of operator that gets us close enough to this goal, but can be expressed in the existing syntax of the language.

Here's what I came up with:

```
var pngHead = binMatch("'\x89PNG\\r\\n\x1a\\n':str8 _:uint4 'IHDR':str4 " +
                       "width:uint4 height:uint4 depth:uint1");

function pngSize(bytes) {
  var match;
  if (match = pngHead(bytes, 0))
    return {width: match.width, height: match.height};
  else
    throw new Error("Not a PNG file.");
}
```

Patterns are precompiled from strings to functions, much like in the template example. The pattern string contains any number of `binding:type` pairs, where `type` is a word like `str` or `uint` followed by a byte size, and `binding` can be _ (an underscore) to ignore a field, a literal (in which case the pattern matches only when the value is equal to the literal), or a field name in which to store the value.

The very ugly string at the start of the pattern contains the first eight bytes of the PNG header. The double backslashes are needed because the content of the string is interpreted as a string literal (again) in the generated code, so it may not contain raw newlines. After the file-identifying string, a four-byte field is found, which we ignore. Next, the string `'IHDR'` announces the start of the image header, which starts with width, height, and color depth fields.

A function produced by `binMatch` takes a `Uint8Array` and an offset integer, and returns `null` for failed matches and an object containing the matched values when the match succeeds. The return object will have an additional field, `end`, which indicates the byte offset of the end of the match.

Here is the core of the match compiler. It is pleasantly small:

```
function binMatch(spec) {
  var totalSize = 0, code = "", match;
  while (match = /^([^:]+):(\w+)(\d+)\s*/.exec(spec)) {
    spec = spec.slice(match[0].length);
    var pattern = match[1], type = match[2], size = Number(match[3]);
    totalSize += size;

    if (pattern == "_") {
      code += "pos += " + size + ";";
    } else if (/^[\w$]+$/.test(pattern)) {
      code += "out." + pattern + " = " + binMatch.read[type](size) + ";";
    } else {
      code += "if (" + binMatch.read[type](size) + " !== " +
        pattern + ") return null;";
    }
  }
  code = "if (input.length - pos < " + totalSize + ") return null;" +
    "var out = {end: pos + " + totalSize + "};" + code + "return out;";
  return new Function("input, pos", code);
}
```

It does a (crude, non-error-checking) parse of the input string using a regular expression that matches a single `pattern:type` element. For wildcard (_) patterns, it simply generates code to move the offset (`pos`) forward. For other patterns, it uses a helper from `binMatch.read` (which we'll look at momentarily) to generate an expression that builds up a JavaScript value from the bytes at the current position. For literals, it generates an `if` that returns `null` when the value found doesn't match the literal.

Finally, an extra conditional is generated at the start of the function, which verifies that there are enough bytes in the array to match the pattern, and code that initializes and returns the output object is added.

These are the type-parsing functions needed for the example:

```
binMatch.read = {
  uint: function(size) {
    for (var exprs = [], i = 1; i <= size; ++i)
      exprs.push("input[pos++] * " + Math.pow(256, size - i));
    return exprs.join(" + ");
  },
  str: function(size) {
    for (var exprs = [], i = 0; i < size; ++i)
      exprs.push("input[pos++]");
    return "String.fromCharCode(" + exprs.join(", ") + ")";
  }
};
```

Given a size, they return a string that contains the expression that will advance the `pos` variable and produce a value of the specified type. Note that `uint` is big-endian (network byte order). Obvious extensions would be to write a little-endian type (`uintL`), which we'd need when parsing our earlier GIF example, and of course signed types (`int`, `intL`).

Further optimizations are possible. For example, we could pick literal strings and integers apart into bytes at compile time, and compare those bytes one by one instead of building up the composite value and comparing that. Or, we could first check all literals in a pattern and only then extract the output fields, so that the match does as little work as possible if it fails. This is a nice property of static metaprogramming—the static part of the input (in this case, the pattern string) gives us a rather high-level view of the desired dynamic behavior, and we can pick a compilation strategy based on that information. If you were to interpret such a pattern at runtime, there would be less room for such decisions.

If you want to test this code out, here's a tiny HTML page that, using the code shown previously, allows you to pick a PNG file and will `console.log` its size:

```
<!doctype html>
<script src="binMatch.js"></script>
<input type="file" id="file">
<script>
  var pngHead = binMatch("'\x89PNG\\r\\n\x1a\\n':str8 _:uint4 " +
                         "'IHDR':str4 width:uint4 height:uint4 depth:uint1");
  document.getElementById("file").addEventListener("change", function(e) {
    var reader = new FileReader();
    reader.addEventListener("loadend", function() {
      var match = pngHead(new Uint8Array(reader.result), 0);
      if (match)
        console.log("Your image is ", match.width, "x", match.height, "pixels.");
      else
        console.log("That is not a PNG image.");
    });
    reader.readAsArrayBuffer(e.target.files[0]);
  });
</script>
```

The binary pattern compiler, by putting pieces of code (literals) from the input string directly in the generated code (without sanity-checking them), could, in slightly contrived situations such as building up the pattern string from user input, be used to inject code into a system. Always take a moment to consider this angle when you use `eval`-like constructs. For some tools, like the template compiler, giving the sublanguage the ability to run arbitrary code is part of the design. For others, like this one, it isn't, and it is a good idea to make sure they can't be used for that purpose. We could fix this by checking whether the syntax of the literals actually conforms to JavaScript literals, or by defining and parsing our own string and number syntax (which could also get rid of the double backslash problem) and not inserting any raw, unparsed code from the template at all.

Closing Thoughts

There is a major convenience gap between my fantasy syntax for pattern matching and the reality of what I came up with. Instead of elegantly expressing our pattern inline, we have to build it up beforehand, in order to ensure that it is built only once—reparsing and recompiling it every time it gets run would, in a situation where the matching happens multiple times, be embarrassingly wasteful. Instead of simply *binding* the variables in the pattern to local variables, we have to store them in an object.

In this case, I think that if you are doing actual binary parsing, the abstraction is helpful enough to live with the not-quite-ideal interface. But the case is representative of a wall that you hit when trying to push `eval`-based abstractions beyond a certain point.

There's a pattern that works well—compiling a domain-specific language down to a piece of code. Some languages can be expressed as JSON-like composite data, rather than plain strings (for example, a decision tree modeled as nested objects).

The awkward part lies in the interaction between the domain-specific language and the code around it. They can't be mixed, due to the requirement that the compilation happens only once, whereas the code that makes *use* of the domain-specific functionality will typically run many times.

Small snippets of code with little external dependencies can be made part of the domain language. In some cases, you might even decide to include closures in your source data structure, in order to be able to access the local environment—yet even those won't be able to close over the incoming data for a specific *invocation* of the functionality, but only over data that has the same lifetime as the compiled artifact.

For this reason, many domain-specific languages are better expressed using interpretation rather than compilation. jQuery is a good example of a successful interpreted domain language in JavaScript—it hacks method chaining in a way that allows for succinct DOM operations. This abstraction would be completely unpractical (though probably faster) when executed as a compiled language.

The pattern where you should consider reaching for a compiled domain-specific language is:

- You're writing chunks of repetitive, low-density code.
- Performance is important.
- The code chunks can conveniently be isolated in functions.
- You can think of a shorter, more elegant notation.

CHAPTER THREE

How to Draw a Bunny

Jacob Thornton

This chapter is not about rendering rabbits with JavaScript.

This chapter is about language and the difference between what it means to draw a "rabbit" and what it means to draw a "bunny."

This chapter is not a tutorial. It's an exegesis. This chapter is at play.

What Is a Rabbit?

> So she was considering, in her own mind (as well as she could, for the hot day made her feel very sleepy and stupid), whether the pleasure of making a daisy-chain would be worth the trouble of getting up and picking the daisies, when suddenly a White Rabbit with pink eyes ran close by her.
>
> —Lewis Carroll, *Down the Rabbit Hole*

A "rabbit" is an animal you might find in a field, forest, or pet shop. It is a gregarious plant-eater with a short tail and floppy ears. It is an actual rabbit existing in reality. A "rabbit" cannot talk to itself. A "rabbit" does not run late. From this point forward, when we speak of rabbits, we speak of these *ordinary*, *everyday* rabbits.

For the purposes of this chapter, to "draw a rabbit" is to apply various drawing techniques in such a way as to render an image of a rabbit indistinguishable from the actual rabbit itself. It is to approach a level of realism on par with that of a photograph. A rabbit drawing is strictly referential. It strives to be a copy.

Drawing a rabbit is mechanical and spec-based. There is a correct way to draw a rabbit and an incorrect way to draw a rabbit.

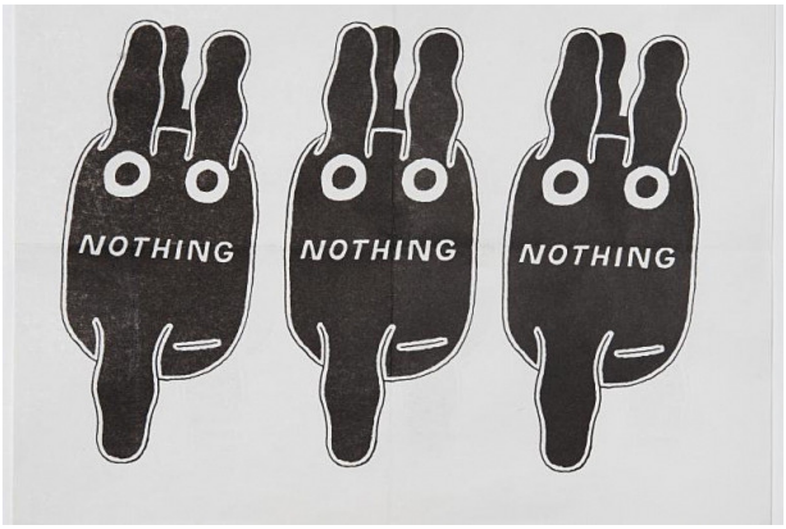
When you draw a rabbit, you are always drawing a very particular rabbit. Deviations from the rabbit model should be regarded as errors. The more your rabbit rendering stays on model, the better.

What Is a Bunny?

> After a time she heard a little pattering of feet in the distance, and she hastily dried her eyes to see what was coming. It was the White Rabbit returning, splendidly dressed, with a pair of white kid gloves in one hand and a large fan in the other: he came trotting along in a great hurry, muttering to himself as he came, "Oh! the Duchess, the Duchess! Oh! won't she be savage if I've kept her waiting!"
>
> —Lewis Carroll, *Down the Rabbit Hole*

A "bunny" is not just a young, cute rabbit.

A bunny is a splendidly dressed abstraction. A playful resemblance that prioritizes an identity other than the rabbit. It is a symbol.



There are several examples from pop culture of bunnies: Bugs Bunny, the Energizer Bunny, etc. These icons are always characters first and rabbits second (or third). Here, the rabbit identity is hijacked and subjugated to serve a new ruling identity.

To "draw a bunny" is to play within the loose constraints of an already existing identity (the *rabbit*) to create something entirely new. The connotation of the word "bunny" itself invokes a lack of seriousness which serves to disarm and undermine the rigid structure of the rabbit, promoting both creative exploration and expression.

Consider the bunny heads of Ray Johnson (pictured above), a correspondence artist from New York.

> In January 1964, Ray Johnson signed a letter to his friend William (Bill) S. Wilson with a small picture of a bunny head next to his name. This image rapidly proliferated, primarily becoming Johnson's signature and "self portrait" as personifications of how he felt on a given day. Johnson also used the bunny head to represent other "characters" who populate his works, as well as the subject of one of his "How to draw" series.
>
> —Frances F.L. Beatty, *Ph.D. The Ray Johnson Estate*

When you draw bunnies, their proximity to a real image of a rabbit isn't called into question. For Johnson, the bunnies ceased to be rabbits, instead becoming a vehicle for alternative expression; a means to creativity; and an exercise in play, imagination, inventiveness, and originality.

What Does This Have to Do with JavaScript?

JavaScript is an expressive language.

Expressions are what lie beyond the literal compiled logic of a program. They are what we as humans read and interpret. The expressiveness of JavaScript is a vehicle through which software developers *speak*. It is a way for developers to infuse their code with semantic value: different styles, dialects, and character. And this potential for linguistic play inherent in JavaScript is precisely where we begin to see "bunnies."

To draw a rabbit in JavaScript is to copy patterns out of books and slides, to mimic specific styles from blogs, and more generally to reproduce already established forms and expressions. Alternatively, to draw a bunny here is to undertake an exercise in experimentation. It is to unearth alternative forms from within the language and then combine these forms in functional yet inventive ways.

In drawing JavaScript bunnies, you're *playing*. It's fun. It challenges and evolves both your individual and the community's understanding of the language. It opens up new potential solutions to old problems, and exposes flaws in old assumptions. It establishes a personal relationship between you and the code you produce. It makes writing JavaScript a craft. An art. It makes reading software personal and purposeful. It establishes an audience for your program other than just the compiler. Intent becomes clearer. Code becomes more consistent. And you grow as a developer.

With this in mind, consider the following conditional statement, which checks to see if a property exists; if the property doesn't exist, it calls a method to set it. Traditionally, this logic might have looked something like this:

```
if (!this.username) {
  this.setUsername();
}
```

As an expression, this logic reads: if not a username, then set a username. However, using the logical OR operator you could express this same statement in a more minimalist way:

```
this.username || this.setUsername()
```

The expression: a username exists, or set a username.

These two code blocks are functionally equivalent, yet their expressions are different. They read differently. Where the former has a sort of exactness and formality, the latter is pithy and short. Exploring these variations in expression is precisely what drawing bunnies is all about. And what's more, by using expressions in conjunction with other like expressions a developer can begin to architect an overarching voice or tone in a program.

Let's consider a second reduced example. Imagine looking inside an array for a username. If the username is not present, you want to add the username to the array. The logic for this might be expressed as follows:

```
if (users.indexOf(this.username) === -1) {
  users.push(this.username)
}
```

This code reads: if the username has an index in the `users` array that is equal to `-1`, then push the username into the `users` array.

An alternative way to express this statement might be to make use of the bitwise NOT operator. The bitwise NOT operator inverts the bits of its operand, turning a `-1` into a `0` (or *falsy*). The preceding logic might then be rewritten simply as:

```
~users.indexOf(this.username) || users.push(this.username)
```

The expression: the username is in the array, or add it.

As you begin to build up these expressions into programs, a certain rhythm and time signature emerges. And as you improve as an engineer, you can begin to orchestrate different phrasings and melodies into your software as well. This establishes a consistent rhythm at the project level, which will make it *much* easier to flow from one piece of a program to another.

The following is a simple function that, given `x`, `y`, `w`, `h`, and `placement` arguments, returns an offset object with a top and left value. It is written in a decidedly slow manner, with a very deliberate, heavy rhythm (switch > case… case… case… case… return):

```
function getOffset (x, y, w, h, placement) {
  var offset
  switch (placement) {
    case 'bottom':
      offset = {
        top: y + h,
        left: x + w/2
      }
      break
    case 'top':
      offset = {
        top: y,
        left: x + w/2
      }
      break
    case 'left':
      offset = {
        top: y + h/2,
        left: x
      }
      break
    case 'right':
      offset = {
        top: y + h/2,
        left: x + w
      }
      break
  }
  return offset
}
```

Notice the difference between this function and the following function, not in terms of computing performance (where the difference is inconsequential), but rather in pure *cognitive* pacing. The next function returns the same result, but with a quicker, more succinct rhythm (return > this/that, this/that, this/that):

```
function getOffset (x, y, w, h, placement) {
  return placement == 'bottom' ? { top: y + h,   left: x + w/2 } :
         placement == 'top'    ? { top: y,       left: x + w/2 } :
         placement == 'left'   ? { top: y + h/2, left: x       } :
                                 { top: y + h/2, left: x + w   }
}
```

A third function might even exaggerate the pacing further, focusing in on the return object itself—clearly calling out expected properties "top" and "left"—but with a more complex rhythm, forking the conditions at the object's properties:

```
function getOffset (x, y, w, h, placement) {
  return {
    top  : placement == 'bottom' ? y + h :
           placement == 'top'    ? y     : y + h/2,
    left : placement == 'right'  ? x + w :
            placement == 'left'   ? x     : x + w/2
    }
  }
```

As you've begun to see, expressions guide our reading of software. In JavaScript, the potential for this sort of variation both enables and is enabled by experimentation and play—which therefore should be championed and not discouraged.

With So Much Variation, Which Way Is Correct?

Imagine sitting several adults down in a room and providing them with an actual image of a rabbit and adequate drawing supplies. Imagine asking them each to draw a rabbit.

Depending on the group's exposure to various drawing techniques, you'd likely receive a variety of renderings, ranging from rather crude to rather capable.

Variety here becomes a metric for the lack of experience in drawing amongst the group. Which is to say, if everyone were perfect at illustration they would each have rendered a photorealistic image, indistinguishable from the image of the rabbit; there wouldn't have been any variety at all.

This is because to draw a rabbit is to exercise one's ability to duplicate. It is an exercise in experience and mimicry. There is a *right answer*, and thus, there isn't room for creativity.

But what if you had asked the same group to draw a bunny?

Arguably the request is at once less threatening, less rigid, and less scientific. To draw a bunny is to draw a rabbit-like thing. It is exceedingly difficult to be critical of a bunny drawing because at most it's only ever a resemblance.

Following this, you could expect the variety in the group's images to be even more exaggerated. To draw a bunny is to celebrate and to lean on variety. *Here, however, variety no longer takes a negative form.* Instead, it is symptomatic of the potential for creative expression implicit in the act of drawing without bounds. It is a positive metric for inventiveness and imagination.

To draw a bunny is to engage with variety. It serves to challenge the image of the rabbit by introducing new means of achieving likeness.

Consider immediately invoked function expressions (IIFEs). By convention, an IIFE takes one of the two following forms:

```
(function (){})()
(function (){}())
```

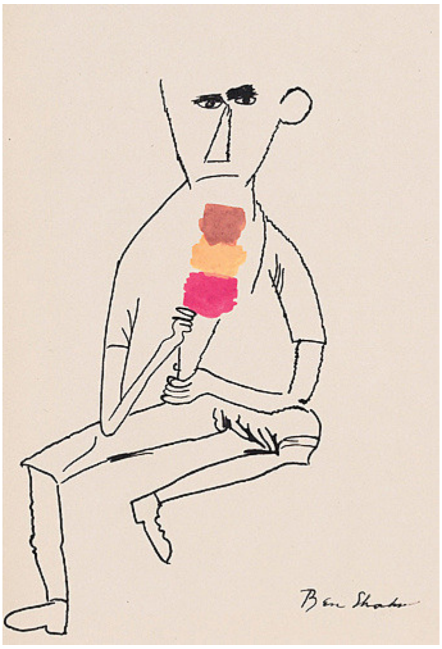
But drawing bunnies is not about convention. Rather, it's an exercise in upsetting convention. And yet at the same time it's about positive variation—one manifestation of an expression not being absolutely superior to another. With this in mind, here are a few other ways you may write an IIFE:

```
!function (){}()
~function (){}()
+function (){}()
-function (){}()
new function (){}
1,function (){}()
1&&function (){}()
var i=function (){}()
```

Each manifestation has its own unique qualities and advantages—some with fewer bytes, some safer for concatenation, each valid and each executable.

How Does This Affect the Classroom?

Because school is limited by grades, it spends much of its time propagandizing the drawing of rabbits.



If you've taken a drawing class, you've almost certainly drawn a block of wood. You've spent hours shading a piece of fruit. You've studied proportions. You've been lectured on perspective. You've been given tools to break things down to a grid. And, after a few months of intense studying, your apple does begin to look a bit more like the apple sitting in front of you.

To be sure, this isn't a bad thing. In fact, quite the opposite. These practices give you foundational knowledge on top of which you can build more complex structures. Furthermore, you can turn the tools in on themselves and exploit them in very interesting ways. And perhaps best of all, they introduce conventions and a new language through which you can engage with your peers.

The problem emerges when students think of these tools in absolute ways. This is the right way to do X; this is the only way to do Y. As you might imagine, this absolutism breeds arrogance, narcissism, and an environment rooted in peer opposition.

Is This Art? And Why Does That Matter?

> It's true to say that when you paint anything, you are also painting not only the subject, but you are painting yourself as well as the object that you are trying to record. Because painting is a dual performance. Because, for instance, if you look at a Rembrandt painting, I feel like I know very much more about Rembrandt than I do about the sitter.
>
> —Francis Bacon, interview with David Sylvester

Briefly consider two libraries I've contributed to this past year: Ratchet and Bootstrap.

Functionally, the content of both libraries is as it should be. What's interesting are the undertones—or rather, the potential for the same sort of undertones you would expect to find in painting, music, or creative writing. Which is to say, the differences in style between these two projects aren't just arbitrary preferences. They're very definite, derived expressions, representative of a certain mood over time.

Bootstrap reads very fun, not serious—nearly every line is a joke. It's trying to provoke you. Taking shortcuts. Demanding that you reread it. Reread it again. It's very pop. Very optimistic. Forward. Playful.

The code for Ratchet is very different. It's very conservative. It's not meant to draw attention to itself. It's very explicit. Assertive, necessary. It's easy to approach. It's a vanilla milkshake.

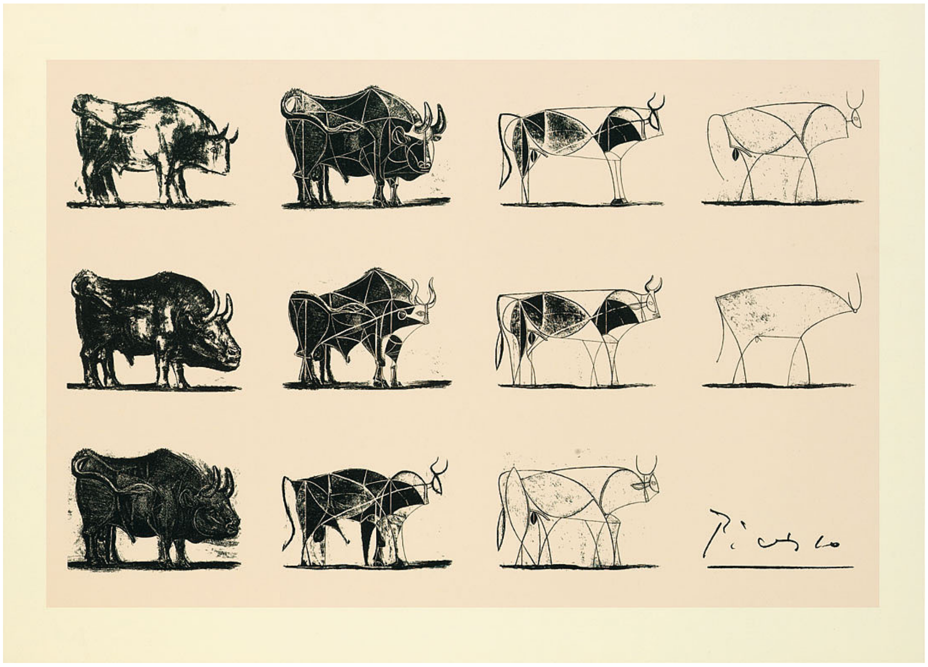
Insofar as art has been characterized in terms of mimesis, expression, communication of emotion, and other such values, it follows that software, when written expressively, is also an artistic gesture. What's more, this realization reinforces our insistence on the importance of drawing bunnies inasmuch as the exercise stretches one's creative and expressive capacities, enabling the formation of opinions and development of style, while also helping to strengthen communication, exploration, and imaginative faculties in the programmer.

Along these lines, my good friend Angus Croll has been exploring further creative manifestations of code with his great articles on literary figures writing JavaScript. In his articles, he writes several functions to return a Fibonacci series of a given length, each program in the style of a different literary figure: Hemingway, Breton, Shakespeare, Poe. The results are comedic, but the point is consistent:

> The joy of JavaScript is rooted in its lack of rigidity and the infinite possibilities that this allows for. Natural languages hold the same promise. The best authors and the best JavaScript developers are those who obsess about language, who explore and experiment with language every day and in doing so develop their own style, their own idioms, and their own expression.
>
> —Angus Croll, *If Hemingway wrote JavaScript*

Beautiful JavaScript is an art. Reading through it should feel uniform; it should allow you to flow from expression to expression. It's not just about executing logic; it's about establishing pace and reflecting a little bit of yourself. It's about taking pride in what you create.

What Does This Look Like?



In 1945, Picasso released a suite of 11 lithographs entitled "Bull." In this series he deconstructs the image of the bull, from realist rendering to hyperreduced line drawing, progressively subtracting from and reimagining its form with each plate.

What's of particular interest here is the progression. Beginning with the realistic brush drawing, Picasso bulks the form up, increasing its expression of power before dissecting it with lines of force, following the contours of its muscles and skeleton, ultimately reducing and simplifying the image into a line. This study is considered the ultimate master class in abstraction, and what's more, it's a classic example of Picasso *drawing bunnies*.

This same exercise in abstraction can be applied to JavaScript.

I had the privilege of working with Alex Maccaw during my time at Twitter. There, we had a number of conversations about interview philosophies and code challenges.

During one of our discussions he mentioned that he had always asked the same introductory interview question during phone screens—and since then, I have adopted it as my first question as well.

The question goes, given the following condition, define `explode`:

```
if ('alex'.explode() === 'a l e x') interview.nextQuestion()
else interview.terminate()
```

There are a number of ways to answer this question. Let's begin with the most verbose:

```
String.prototype.explode = function () {
  var i
  var result = ''
  for (i = 0; i < this.length; i++) {
    result = result + this[i]
    if (i < result.length - 1) {
      result = result + ' '
    }
  }
  return result
}
```

This block is swollen and distended, yet deliberate. There's nothing clever. It's by the book. And it's easily the most common response to the question.

Simply put, we declare variables `i` and `result`, iterating over the string's value, pushing its characters to `result` and conditionally adding a space between each character until eventually we return.

Fine. But now let's try something a bit cleverer:

```
String.prototype.explode = function (f,a,t) {
  for (f = a = '', t = this.length; a++ < t;) {
    f += this[a-1]
    a < t && (f += ' ')
  }
  return f //ollow @fat
}
```

If you write code like this, people will hate you. *Without question*. It's playful. It looks to trick you. To trick the language. It assaults the reader. It's concerned with everything, except its own logic. It's vain. But it's beautiful (to me).

In this block, we're scoping the variables to the function by including them as pseudoarguments (which spell my Twitter handle). The `for` loop saves some characters by setting both `f` and `a` to *new string*, and the `a` is then coerced in the next expression to `1` by the `++` increment operator_, just in time to be used in the equality comparison. On the next line the program subtracts 1 from `a` before indexing the string to make up for starting the loop at 1 (rather than 0). It then conditionally adds a space to the end, before completing the loop and returning the result.

The next iteration of the solution is by far the simplest, leaning heavily on the language's tool belt. Perhaps surprisingly, this response is actually very uncommon to receive in real interviews:

```
String.prototype.explode = function () {
  return this.split('').join(' ')
}
```

This solution is about getting to the next question. It's clever, but not overly so. It's blunt. It's mature. If the previous solution was crass, this one is urbane.

And finally, the absolute simplest:

```
String.prototype.explode = function (/*smart a$$*/) {
  return 'a l e x'
}
```

Which I've never gotten.

What Did I Just Read?

If drawing rabbits in JavaScript means copying patterns out of books or mimicking specific styles from blogs, drawing bunnies is about experimentation and creative expression.

To draw a bunny is to pervert the conventions of the language. To draw your breath or to get it all out as fast as possible. It's an exercise in discovering and pushing the bounds of your understanding of the language. It's about reinforcing and challenging JavaScript as a craft.

In drawing JavaScript bunnies, you're always at *play*. And you're getting better.

CHAPTER FOUR

Too Much Rope, or JavaScript for Teams

Daniel Pupius

Beauty is power and elegance, right action, form fitting function, intelligence, and reasonability.

—Kim Stanley Robinson, *Red Mars*

JavaScript is a flexible language. In fact, this entire book is a testament to its expressiveness and dynamism. Within these pages you'll hear stories of how to bend the language to your will, descriptions of how to use it to experiment and play, and suggestions for seemingly contradictory ways to write it.

My job is to tell a more cautionary tale.

I'm here to ask the question: what does it mean to write JavaScript in a team? How do you maintain sanity with 5, 10, 100 people committing to the same codebase? How do you make sure new team members can orient themselves quickly? How do you keep things DRY without forcing broken abstractions?

Know Your Audience

In 2005 I joined the Gmail team in sunny Mountain View, California. The team was building what many considered at the time to be the pinnacle of web applications. They were awesomely smart and talented, but across Google, JavaScript wasn't considered a "real programming language"—you engineered backends, you didn't engineer web UIs—and this mentality affected how they thought about the code.

Furthermore, even though the language was 10 years old, JavaScript engines were still limited: they were designed for basic form validation, not building applications. Gmail was starting to hit performance bottlenecks. To get around these limitations much of the application was implemented as global functions, anything requiring a dot lookup was avoided, sparse arrays were used in place of templates, and string concatenation was a no-no.

The team was writing first and foremost for the JavaScript engine, not for themselves or others. This led to a codebase that was hard to follow, inconsistent, and sprawling.

Instead of optimizing by hand, we transitioned to a world where code was written for humans and the machine did the optimizations. This wasn't a new language, mind you—it was important that the raw code be valid JavaScript, for ease of understanding, testing, and interoperability. Using the precursor to the Closure Compiler, we developed optimization passes that would collapse namespaces, optimize strings, inline functions, and remove dead code. This is work much better suited to a machine, and it allowed the raw code to be more readable and more maintainable.

TIP

Lesson 1: Code for one another, and use tools to perform mechanical optimizations.

Stupid Good

As the old adage goes, debugging is harder than writing code, so if you write the cleverest code you can, you'll never be clever enough to debug it.

It can be fun to come up with obscure and arcane ways of solving problems, especially since JavaScript gives you so much flexibility. But save it for personal projects and JavaScript puzzlers.

When working in a team you need to write code that everyone is going to understand. Some parts of the codebase may go unseen for months, until a day comes when you need to debug a production issue. Or perhaps you have a new hire with little JavaScript experience. In these types of situation, keeping code simple and easy to understand will be better for everyone. You don't want to spend time decoding some bizarro, magical incantation at two in the morning while debugging production issues.

Consider the following:

```
var el = document.querySelector('.profile');
el.classList[['add','remove'][+el.classList.contains('on')]]('on');
```

And an alternative way of expressing the same behavior:

```
var el = document.querySelector('.profile');
if (el.classList.contains('on')) el.classList.remove('on');
else el.classList.add('on');
```

Saying that the second snippet is better than the first may seem in conflict with the concept that "succinctness = power." But I believe there is a disconnect that stems from the common synonyms for succinct: compact, brief.

I prefer *terse* as a synonym:

> using few words, devoid of superfluity, smoothly elegant

The first snippet *is* more compact than the second snippet, but it is denser and actually includes more symbols. When reading the first snippet you have to know how coercion rules apply when using a numeric operator on a Boolean, you have to know that methods can be invoked using subscript notation, and you have to notice that square brackets are used for both defining an array literal and method lookup.

The second snippet, while longer, actually has less syntax for the reader to process. Furthermore, it reads like English: "If the element's class list contains 'on', then remove 'on' from the class list; otherwise, add 'on' to the class list."

All that said, an even better solution would be to abstract this functionality and have the very simple, readable, and succinct:

```
toggleCssClass(document.querySelector('.profile'), 'on');
```

TIP

Lesson 2: Keep it simple; compactness != succinctness.

Keep It Classy

When I'm talking with "proper programmers," they often complain about how terrible JavaScript is. I usually respond that JavaScript is misunderstood, and that one of the main issues is that it gives you too much rope—so inevitably you end up hanging yourself.

There were certainly questionable design decisions in the language, and it is true that the early engines were quite terrible, but many of the problems that occur as JavaScript codebases scale can be solved with pretty standard computer science best practices. A lot of it comes down to code organization and encapsulation.

Unfortunately, until we finally get ES6 we have no standard module system, no standard packaging mechanisms, and a prototypal inheritance model that confuses a lot of people and begets a million different class libraries.

While JavaScript's prototypal inheritance allows instance-based inheritance, I generally suggest when working in a team that you simulate classical inheritance as much as possible, while still utilizing the prototype chain. Let's consider an example:

```
var log = console.log.bind(console);
var bob = {
  money: 100,
  toString: function() { return '$' + this.money }
};
var billy = Object.create(bob);

log('bob:' + bob, 'billy:' + billy); // bob:$100 billy:$100
bob.money = 150;
log('bob:' + bob, 'billy:' + billy); // bob:$150 billy:$150
billy.money = 50;
log('bob:' + bob, 'billy:' + billy); // bob:$150 billy:$50
delete billy.money;
log('bob:' + bob, 'billy:' + billy); // bob:$150 billy:$150
```

In this example, `billy` inherits from `bob`. What that means in practice is that `billy.prototype = bob`, and nonmatching property lookups on `billy` will delegate to `bob`. In other words, to begin with `billy`'s $100 *is* `bob`'s $100; `billy` isn't a copy of `bob`. Then, when `billy` gets his own money, it essentially overrides the property that was being inherited from `bob`. Deleting `billy`'s money doesn't set it to `undefined`; instead, `bob`'s money becomes `billy`'s again.

This can be rather confusing to newcomers. In fact, developers can go a long time without ever knowing precisely how prototypes work. So, if you use a model that simulates classical inheritance, it increases the chances that people on your team will get on board quickly and allows them to be productive without necessarily needing to understand the details of the language.

Both the Closure library's `goog.inherits` and Node.js's `util.inherits` make it easy to write class-like structures while still relying on the prototype for wiring:

```
function Bank(initialMoney) {
  EventEmitter.call(this);
  this.money = money;
}
util.inherits(Bank, EventEmitter);

Bank.prototype.withdraw = function (amount) {
  if (amount <= this.money) {
    this.money -= amount;
    this.emit('balance_changed', this.money); // inherited
    return true;
  } else {
    return false;
  }
}
```

This looks very similar to inheritance in other languages. `Bank` inherits from `EventEmitter`; the superclass's constructor is called in the context of the new instance; `util.inher` `its` wires up the prototype chain just like we saw with `bob` and `billy` earlier; and then the property lookup for `emit` falls to the `EventEmitter` "class."

A suggested exercise for the reader is to create instances of a class without using the `new` keyword.

TIP

Lesson 3: Just because you can doesn't mean you should.

TIP

Lesson 4: Utilize familiar paradigms and patterns.

Style Rules

The need for consistent style as codebases and teams grow is nothing unique to JavaScript. However, where many languages are opinionated about coding style, JavaScript is lenient and forgiving. This means it's all the more important to define a set of rules the team should stick to.

Good style is subjective and can be difficult to define, but there are many cases where certain style choices are quantifiably better than others. In the cases where there isn't a quantifiable difference, there is still value in making an arbitrary choice one way or the other.

TIP

Style guides provide a common vocabulary so people can concentrate on what you're saying instead of how you're saying it.

A good style guide should set out rules for code layout, indentation, whitespace, capitalization, naming, and comments. It is also good to create usage guides that explain best practices and provide guidance on how to use common APIs. Importantly, these guides should explain *why* a rule exists; over time you will want to reevaluate the rules and should avoid them becoming cargo cults.

Style guides should be enforced by a linter and if possible coupled with a formatter to remove the mechanical steps of adhering to the guide. You don't want to waste cycles correcting style nits in code reviews.

The ultimate goal is to have all code look like it was written by the same person.

TIP

Lesson 5: Consistency is king.

Evolution of Code

When I was first working on Google Closure there was no simple utility for making `XMLHttpRequests`; everything was rolled up in large, application-specific request utilities.

So, in my naiveté XhrLite was born.

XhrLite became popular—no one wants to use a "heavy" implementation—but its users kept finding features that were missing. Over time small patches were submitted, and XhrLite accumulated support for form encoded data, JSON decoding, XSSI handling, headers, and more—even fixes for obscure bugs in FF3.5 web workers.

Needless to say, the irony of "XhrLite" becoming a distinctly heavy behemoth was not lost, and eventually it was renamed "XhrIo." The API, however, remained bloated and cumbersome.

TIP

Small changes–reasonable in isolation–evolve into a system that no one would *ever* design if given a blank canvas.

Evolutionary complexity is almost a force of nature in software development, but it has always seemed more pronounced with JavaScript. One of the strengths that helped spur JavaScript's popularity is that you can get up and running quickly. Whether you're creating a simple web app or a Node.js server, a minimal dev environment and a few lines of code yields something functional. This is great when you're learning, or prototyping, but can lead to fragile foundations for a growing team.

You start out with some simple HTML and CSS. Perhaps you add some event handlers using jQuery. You add some XHRs, maybe you even start to use `pushState`. Before long you have an actual single-page application, something you never intended at first. Performance starts to suffer, there are weird race conditions, your code is littered with `setTimeouts`, there are hard-to-track-down memory leaks…you start wondering if a traditional web page would be better. You have the duck-billed platypus of applications.

TIP

Lesson 6: Lay good foundations. Be mindful of evolutionary complexity.

Conclusion

JavaScript's beauty is in its pervasiveness, its flexibility, and its accessibility. But beauty is also contextual. What started as a "scripting language" is now used by hundred-plus-person teams and forms the building blocks of billion-dollar products. In such situations you can’t write code in the same way you would hacking up a one-person website. So…

1. Code for one another, and use tools to perform mechanical optimizations.
2. Keep it simple; compactness != succinctness.
3. Just because you can doesn’t mean you should.
4. Utilize familiar paradigms and patterns.
5. Consistency is king.
6. Lay good foundations. Be mindful of evolutionary complexity.

CHAPTER FIVE

Hacking JavaScript Constructors for Model Harmony

Ben Vinegar

JavaScript MVC—or MVW (Model, View, "Whatever")—frameworks come in many flavors, shapes, and sizes. But by virtue of their namesake, they all provide developers with a fundamental component: models, which "model" the data associated with the application. In client-side web apps, they typically represent a database-backed object.

Last year at Disqus, we rewrote our embedded client-side application in Backbone, a minimal MVC framework. Backbone is often criticized for having an unsophisticated "view" layer, but one thing it does particularly well is managing models.

Defining a new model in Backbone looks like this:

```
var User = Backbone.Model.extend({
  defaults: {
    username: '',
    firstName: '',
    lastName: ''
  },

  idAttribute: 'username',

  fullName: function () {
    return this.get('firstName') + this.get('lastName');
  }
});
```

Here's some sample code that initializes a new model, and demonstrates how that model instance might be used in an application:

```
var user = new User({
  username: 'john_doe',
  firstName: 'John',
  lastName: 'Doe'
});

user.fullName(); // John Doe

user.set('firstName', 'Bill');

user.save(); // PUTs changes to server endpoint
```

These are simple examples, but client-side models can be very powerful, and they are typically—ahem—the backbone of any nontrivial MVC app.

Additionally, Backbone provides what are called "collection" classes, which help developers easily manipulate common sets of model instances. You can think of them as superpowered arrays, loaded with helpful utility functions:

```
var UserCollection = Backbone.Collection.extend({
  model: User,
  url: '/users'
});

var users = new UserCollection();

users.fetch(); // Fetches user records via HTTP

var johndoe = users.get('john_doe'); // Find by primary idAttribute
```

Not all MVC frameworks implement a `Collection` class exactly like Backbone does. For example, Ember.js defines a `CollectionView` class, which similarly maintains a set of common models, but tied to a DOM representation. API differences aside, it's clear that developers commonly manipulate and render sets of objects, and frameworks provide different facilities for doing so.

Doppelgangers

When you're working with large or even medium-sized client applications, it's common to have multiple model instances representing the same database-backed object. This usually happens when you have multiple views of some data, such that a model appears in two or more views.

Consider this example, which introduces two new collections of users: `Followers`, for users that are following a given user (say, on a social network), and `Following`, for users whom a given user happens to be following. A user who is both a follower *and* being followed will appear in both collections, in which case we will have duplicate instances of the same database-backed model:

```
var FollowingCollection = UserCollection.extend({
  url: '/following'
});

var FollowersCollection = UserCollection.extend({
  url: '/followers'
});

var following = new FollowingCollection();
var followers = new FollowersCollection();

following.fetch();
followers.fetch();

var user1 = following.get('johndoe');
var user2 = followers.get('johndoe');

user1 === user2; // false
```

Having multiple instances of the same model has two major downsides.

First, you are using additional memory to represent the same object. Depending on the complexity of the model and the sizes of the attributes it holds, it's not unreasonable for a single instance to consume kilobytes of memory. If instances are duplicated dozens or hundreds of times—a very possible scenario for long-lived single-page applications—they can quickly become a memory sink.

Secondly, if you or the user modifies the state of one of these models on the client, other instances of that model will fall out of sync. This can happen through a number of means, like if the user changes the state of the object via the UI, or an update created by another user is sent to the client via a real-time service:

```
user1.set('firstName', 'Johnny');

user2.get('firstName'); // still John
```

In this simple example, where the same user appears in only two different collections, it might seem trivial to update both instances manually with the new property. But it's easy to imagine how in a complex application the same user object might exist across dozens of different collections—not just follower/following lists, but also notifications, feed items, logs, and so on.

It would be terrific if, instead of having to track down every instance of a given model, we could have each instance update itself intelligently. Or better yet, if we never had duplicated instances to begin with.

Miniature Models of Factories

A common solution for handling duplicate instances is to use a factory function when you create a new model instance. If the factory detects that a model instance already exists, it will just return the existing instance instead:

```
var userCache = {};

function UserFactory(attrs, options) {
  var username = attrs.username;

  return userCache[username] ?
    userCache[username] :
    new User(attrs, options);
}

var user1 = UserFactory({ username: 'johndoe' });
var user2 = UserFactory({ username: 'johndoe '});

user1 === user2; // true
```

In order to use this pattern effectively, you must always use this factory function when creating new instances. This is a simple enough chore when managing your own code. But difficulty arises when you try to enforce this pattern in codebases you aren't responsible for, like third-party libraries and plugins.

Consider, for example, the `Collection.prototype._prepareModel` function from Backbone's source code. Backbone uses this function to "prepare" and ultimately create a new model instance to add to a collection. It is invoked by a variety of means, such as when you're populating a collection with models returned from an HTTP resource:

```
// Prepare a hash of attributes (or other model) to be added to this
// collection.
Backbone.Collection.prototype._prepareModel = function(attrs, options) {
  if (attrs instanceof Model) {
    if (!attrs.collection) attrs.collection = this;
    return attrs;
  }
  options || (options = {});
  options.collection = this;
  var model = new this.model(attrs, options);
  if (!model._validate(attrs, options)) {
    this.trigger('invalid', this, attrs, options);
    return false;
  }
  return model;
};
```

Of particular importance is this line:

```
var model = new this.model(attrs, options);
```

This is what actually creates a new instance of the model associated with this collection.

`this.model` is a reference to the constructor of the model class the collection wraps. It's specified when you define a new collection class, like we did earlier:

```
var UserCollection = Backbone.Collection.extend({
  model: User,
  url: '/users'
});
```

What's pretty cool is that instead of passing the `User` class to the collection definition, we can pass the `UserFactory` class (our factory function that returns unique model instances):

```
var UserCollection = Backbone.Collection.extend({
  model: UserFactory,
  url: '/users'
});
```

`UserFactory` will then be assigned to `this.model`, and will be invoked by the `new` operator when the collection creates a new instance:

```
var model = new this.model(attrs, options); // this.model is UserFactory
```

But wait a minute. Now we're invoking `UserFactory` via the `new` operator. We weren't doing that earlier; we were calling the function directly. Does this even work?

It turns out it does.

Constructor Identity Crisis

What exactly happens when you use the `new` operator on a function? A few things:

1. It creates a new object.
2. It sets that object's `prototype` property to be the `prototype` property of the constructor function.
3. It invokes the constructor function, with `this` assigned to the newly created object.
4. It returns the object, *unless* the constructor function returns a *nonprimitive* value. In that case, the nonprimitive value is returned instead.

That last one is the neat part. If your constructor function returns a nonprimitive value, that becomes the result of the `new` operation.

Since `UserFactory` returns a nonprimitive, that means that these two operations return the same value:

```
var user1 = UserFactory({ username: 'johndoe' });
var user2 = new UserFactory({ username: 'johndoe '});

user1 === user2; // true
```

This property of the `new` operator is pretty handy. It means that you can essentially discard the object created by `new`, and return what you want—in our case, a unique user model instance.

Making It Scale

In the examples so far, `UserFactory` has been a single-purpose factory function; it only guarantees uniqueness of `User` instances. While that's super handy, there are probably other models for which we'll want to guarantee uniqueness. So, it would be nice to have a general-purpose wrapper that can work for any model class.

In a moment we'll look at a function called `UniqueFactory`. It's actually a constructor function that is invoked with the `new` operator, and takes as input a normal Backbone model class. It returns a wrapped constructor function that generates unique instances of that class.

For example, it can actually generate a `UserFactory` class:

```
var UserFactory = new UniqueFactory(User);

var user1 = UserFactory({ username: 'johndoe' });
var user2 = new UserFactory({ username: 'johndoe '});

user1 === user2; // true
```

The `UniqueFactory` implementation is shown here:

```
/**
 * UniqueFactory takes a class as input, and returns a wrapped version of
 * that class that guarantees uniqueness of any generated model instances.
 *
 * Example:
 *   var UniqueUser = new UniqueFactory(User);
 */

function UniqueFactory (Model) {
  var self = this;

  // The underlying Backbone Model class
  this.Model = Model;

  // Tracked instances of this model class
  this.instances = {};

  // Constructor to return that will be used for creating new instances
  var WrappedConstructor = function (attrs, options) {
    return self.getInstance(attrs, options);
  };

  // For compatibility with Backbone collections, our wrapped
  // model prototype should point to the *actual* Model prototype
  WrappedConstructor.prototype = this.Model.prototype;

  return WrappedConstructor;
}

UniqueFactory.prototype.getInstance = function (attrs, options) {
  options = options || {};

  var id = attrs && attrs[this.Model.prototype.idAttribute];

  // If there's no ID, this model isn't being tracked, and
  // cannot be tracked; return a new instance
  if (!id)
    return new this.Model(attrs, options);

  // Attempt to restore a cached instance
  var instance = this.instances[id];
  if (!instance) {
    // If we haven't seen this instance before, start caching it
    instance = this.createInstance(id, attrs, options);
  } else {
    // Otherwise update the attributes of the cached instance
    instance.set(attrs);
  }
  return instance;
};

UniqueFactory.prototype.createInstance = function (id, attrs, options) {
  var instance = new this.Model(attrs, options);
  this.instances[id] = instance;

  return instance;
};
```

Let's take a closer look at the `UniqueFactory` constructor, because it's doing some tricky stuff.

First recall that `UniqueFactory` is intended to be invoked with the `new` operator, which creates a new object and assigns it to `this` (which is immediately aliased to `self`). The constructor creates a new function, `WrappedConstructor`, whose signature matches that of a `Backbone.Model` constructor function. But instead of invoking the actual constructor, it calls the `getInstance` prototype method of the `UniqueFactory` instance we just created:

```
var WrappedConstructor = function (attrs, options) {
  return self.getInstance(attrs, options);
};
```

Then, on the last line of this function, `UniqueFactory` returns `WrappedConstructor`. Once again, we've decided to ignore the object created by the `new` operator, and instead return an entirely different object—a function, even.

This means that when we invoke `UniqueFactory`, the return value is actually our wrapped constructor:

```
var UserFactory = new UniqueFactory(User); // WrappedConstructor
```

However, this time we actually used the object created by the `new` operator. We just didn't return it. And it still exists: in the closure created by the `WrappedConstructor` function (`self`).

Phew. Did you catch all that?

This is kind of a funny implementation. It's not necessarily ideal, but I presented it to you to demonstrate how the `new` operator can be abused in an interesting—if somewhat confusing—way. Namely, a constructor function can both make use of the object created by `new` and return an entirely new value, at the same time.

Beware of Memory Leaks

In the example factory implementations here, I've glossed over an important detail: they maintain an ever-growing global cache of model instances. Since instances are never removed from the cache even when they're no longer needed, they continue occupying memory forever (or at least, until the page refreshes).

For example, suppose a unique model instance is destroyed via `Model.prototype.destroy`:

```
(function () {
  var user = UserFactory({ username: 'johndoe' });

  user.destroy(); // sends HTTP DELETE to API server
})();
```

Despite the `user` variable not existing outside the functional scope in which it is declared, and despite the *johndoe* record being destroyed on the server, the instance lives on inside our `UserFactory` instance cache.

This is particularly bad in long-lived single-page applications. A proper implementation would "track" instance creation and dismissal, and remove the instance from the cache when it is no longer required to be there.

Conclusion

In this chapter, we've identified the "uniqueness" problem that affects applications where the same database-backed object appears in multiple collections. We explored a powerful solution for this problem: functions that wrap a class constructor, and guarantee the uniqueness of any returned objects. Lastly, we introduced a utility, `UniqueFac tory`, that generates model classes that similarly guarantee uniqueness.

What we've covered isn't necessarily unique to JavaScript. Factory methods that return unique instances are tried-and-true patterns that can be—and certainly have been—implemented in any number of languages.

But one clever trick that JavaScript has up its sleeve is the `new` operator. Specifically, the function on which `new` is called can ignore the newly created object (`this`) and return what it pleases. This little quirk is deceptively powerful, because it allows you to emulate object creation when object creation is expected—for instance, when you're working with external libraries like Backbone.

In my experience, JavaScript has never been accused of being a particularly flexible language. It still bears the marks of being designed in 10 days. But for all its warts, occasionally I discover new things about it that particularly please me. This small property of the `new` operator is one of them. Hopefully, having read this chapter, you'll feel similarly.

CHAPTER SIX

One World, One Language

Jenn Schiffer

There sure are a lot of languages.

—Jenn Schiffer

It was September 2003 when I began my undergraduate studies in computer science. Having chosen a liberal arts school, I was required to select a number of general education course requirements that lived outside the realm of my major. One of those requirements was two foreign language courses. When I inquired about using Java to fulfill that sequence, my request was immediately shut down. "You have to pick a real foreign language, like Spanish or French," my undergraduate advisor told me.

Perhaps I should have asked about JavaScript.

To be multilingual, or a polyglot, has always been presented as superior to being able to speak one's native language only. I have never understood why people believe this. Living under one roof, having one job for an extended amount of time, and being in a long-term monogamous relationship: these are seen as qualities of a stable life. Being an expert in a single subject, as opposed to knowing a little bit about a lot, is championed. So should be the case with programming.

JavaScript is a single, stable language that is powerful enough to build the World Wide Web, make robots move, and convince publishers to print entire books about it. If we were required to pick a single "best" programming language, JavaScript seems like a no-brainer.

It is understandably controversial to say that a specific language is better than the rest and that it should, therefore, become the official language of programming. Who am I to decide which language every other programmer should learn and build with? In my favor, one of the greatest aspects of web development in the 21st century is the expression of opinions so strong they are worthy of becoming web standards.

An Imperative, Dynamic Proposal

Imagine you are an academic advisor at a liberal arts college and are tasked with defining the choices given to students for their foreign language requirements. A language called "JavaScript" comes up in a proposal, and you need to study it and determine if it is a viable option. Naturally, you just so happen to be a fluent JavaScript expert, yet you are not sure it would be more useful than, say, Java.

Java is notoriously simple to learn at the college freshman level, regardless of the student's experience:

```
/**
* Hello World in Java
*/

class Example {
  public static void main(String[] args) {
    System.out.println("Hello World.");
  }
}
```

To run Java, though, the client must also be running the Java virtual machine (J.V.M.). It would be silly to ask students to carry multiple machines around to all of their classes, so a language that does not require a JVM would be a better option. You might be thinking, "Maybe this is a weird joke I just don't get?" Perhaps the author, yours truly, is trying to make a joke, and you feel like there are much better ones she could make. But this is no joke: JavaScript does *not* require a Java virtual machine.

Neither does Haskell:

```
-- Hello World in Haskell
main = putStrLn "Hello World."
```

The problem with Haskell is that, unlike JavaScript, it requires installation of a compiler. It is also a functional programming language that, like Latin, is considered "dead" and referenced only in historical texts. Yes, it is useful to learn Haskell in order to understand the context of programming today, but not for making useful products. It would be irresponsible to require students to learn something that would not help them build client-side web applications.

Ruby happens to be quite useful in building web applications:

```
# Hello World in Ruby
puts "Hello World."
```

One of the features of Ruby is flexibility in the form of having dozens of different versions, the most popular of which is called Rails. Rails itself has many versions—dialects, if you will—which causes communication breakdowns between apps. Multiple versions works for operating system releases, but not for web development. JavaScript versions do not matter to the user or developer because it is not server-side, and removing that headache makes it a better option for teaching.

Cascading Style Sheets (C.S.S.) is also not server-side and does not require a compiler or virtual machine:

```
/* Hello World in C.S.S. */
#example { content:'Hello World.'}
```

But much like hardware does not work without software, C.S.S. does not work without other languages. In the previous example, the browser looks for an element on the page with the ID “example.” If the developer did not use another language to create that element, the C.S.S. cannot do anything. The professor teaching the foreign language course would have to teach another language in addition to C.S.S., and that is asking a lot of the staff. JavaScript does not need other languages to work. It just works.

How about HyperText Markup Language (H.T.M.L.)? It works on its own and does not need a compiler installed:

```
<!-- Hello World in H.T.M.L. -->
<!DOCTYPE html>
<html ng-app>
  <head>
    <script src="angular.js"></script>
  </head>
  <body ng-controller="ExampleController">

    <script type="text/javascript">
      function ExampleController($scope) {
        $scope.printText = "Hello World";
      }
    </script>

    <h1></h1>

  </body>
</html>
```

Actually, H.T.M.L. *does* need another language to work, and it is JavaScript. Sure, in the past, H.T.M.L. used to be all you needed to create a web page. In the current state of the Semantic Web, though, the use of frontend JavaScript frameworks like Ember.js is required to bind text to a document.

JavaScript does not need a JavaScript framework to run, because it is JavaScript already:

```
// Hello World in JavaScript
alert('Hello World');
```

And there you have it. Simple, pure, vanilla, untouched, beautiful JavaScript. Short, effective, and simple to teach. You can rightfully count JavaScript among the options for teaching foreign languages to your college's student body.

The Paradox of Choice

As hard as it is to choose the options of foreign language courses a student can take, it is even harder for the student to decide among those options. One of the hardest problems in computer science is choosing the right tool to use, and the same certainly goes for communication. It is an impossible question to ask: "German or JavaScript?" Why can a student not learn both?

This may seem like an NP-complete problem. You cannot teach JavaScript in German, because JavaScript syntax is in American English:

```
Benachrichtigung('Hello World');
```

Although semantically, factually, and tactfully correct, the preceding code is syntactically incorrect:

```
>> ReferenceError: Benachrichtigung is not defined
```

It turns out, though, that you *can* teach German in JavaScript:

```
alert('Hallo Welt');
```

If one can learn a language within JavaScript, then it is clear that JavaScript can be the only foreign language course offered that will not prevent students from learning how to communicate in foreign countries.

Globalcommunicationscript

College is the basis of learning for all web developers, as is evident with the current education revolution within the software industry. As more programming jobs are created, educators grow more responsible for fostering the growth of new developers. To make this job easy, it only makes perfect sense to choose a language that everyone can communicate and learn with. As we discovered in our foreign language course narrative, that language is JavaScript.

Simple, pure, vanilla, untouched, beautiful JavaScript.

CHAPTER SEVEN

Math Expression Parser and Evaluator

Ariya Hidayat

Domain-specific languages (DSLs) are encountered in many aspects of a software engineer's life: configuration file formats, data transfer protocols, model schemas, application extensions, interface definition languages, and many others. Because of the nature of such languages, the language expression needs to be straightforward and easy to understand.

In this chapter, we will explore the use of JavaScript to implement a simple language that can be used to evaluate a mathematical expression. In a way, it is very similar to a classic handheld programming calculator. Besides the typical math syntax, our JavaScript code should handle operator precedence and understand predefined functions.

Given a math expression as a string, this is the series of processing applied to that string:

- The string is split into a stream of tokens.
- The tokens are used to construct the syntax tree.
- The syntax tree is traversed to evaluate the expression.

Each step will be described in the following sections.

Lexical Analysis and Tokens

The first important thing to do to a string representing a math expression is *lexical analysis*—that is, splitting the string into a stream of tokens. Quite expectedly, a function that does this is often called a *tokenizer*. Alternatively, it is also known as a *lexer* or a *scanner*.

We first need to define the types of the tokens. Since we'll be dealing with simple math expressions, all we really need are number, identifier, and operator. Before we can identify a portion of a string as one of these tokens, we need some helper functions (they are self-explained):

```
function isWhiteSpace(ch) {
    return (ch === 'u0009') || (ch === ' ') || (ch === 'u00A0');
}

function isLetter(ch) {
    return (ch >= 'a' && ch < = 'z') || (ch >= 'A' && ch < = 'Z');
}

function isDecimalDigit(ch) {
    return (ch >= '0') && (ch < = '9');
}
```

Another very useful auxiliary function is the following `createToken`, used mostly to avoid repetitive code in the later stages. It basically creates an object for the given token `type` and `value`:

```
function createToken(type, value) {
    return {
        type: type,
        value: value
    };
}
```

As we iterate through the characters in the math expression, we will need a way to advance to the next character and another method to have a peek at the next character without advancing our position:

```
function getNextChar() {
    var ch = 'x00',
        idx = index;
    if (idx < length) {
        ch = expression.charAt(idx);
        index += 1;
    }
    return ch;
}

function peekNextChar() {
    var idx = index;
    return ((idx < length) ? expression.charAt(idx) : 'x00');
}
```

In our expression language, spaces do not matter: `40 + 2` is treated the same as `40+2`. Thus, we need a function that ignores whitespace and continues to move forward until there is no whitespace anymore:

```
function skipSpaces() {
    var ch;

    while (index < length) {
        ch = peekNextChar();
        if (!isWhiteSpace(ch)) {
            break;
        }
        getNextChar();
    }
}
```

Suppose we want to support standard arithmetic operations, brackets, and simple assignment. The operators we need to support are `+`, `-`, `*`, `/`, `=`, `(`, and `)`. A method to scan such an operator can be constructed as follows. Note that rather than checking the character against all possible choices, we just use a simple trick utilizing the `String.indexOf` method. By convention, if this `scanOperator` function is called but no operator is detected, it returns `undefined`:

```
function scanOperator() {
    var ch = peekNextChar();
    if ('+-*/()='.indexOf(ch) >= 0) {
        return createToken('Operator', getNextChar());
    }
    return undefined;
}
```

Deciding whether a series of characters is an identifier or not is slightly more complex. Let us assume we allow the first character to be a letter or an underscore. The second, third, and subsequent characters can each be another letter or a decimal digit. We disallow a decimal digit to start an identifier to avoid confusion with a number. Let's begin with two simple helper functions that do these checks:

```
function isIdentifierStart(ch) {
    return (ch === '_') || isLetter(ch);
}

function isIdentifierPart(ch) {
    return isIdentifierStart(ch) || isDecimalDigit(ch);
}
```

The identifier check can now be written as a simple loop like this:

```
function scanIdentifier() {
    var ch, id;

    ch = peekNextChar();
```

```
        if (!isIdentifierStart(ch)) {
            return undefined;
        }

        id = getNextChar();
        while (true) {
            ch = peekNextChar();
            if (!isIdentifierPart(ch)) {
                break;
            }
            id += getNextChar();
        }

        return createToken('Identifier', id);
    }
```

Since we want to process math expressions, it would be absurd not to be able to recognize numbers. We want to support simple integers such as 42, floating points like 3.14159, and also numbers written in scientific notation like 6.62606957e-34. A skeleton for such a function is:

```
    function scanNumber() {
        // return a token representing a number
        // or undefined if no number is recognized
    }
```

And here is the breakdown of the function implementation.

First and foremost, we need to detect the presence of a number. It's rather easy—we just check whether the next character is a decimal digit or a decimal point (because .1 is a valid number):

```
    ch = peekNextChar();
    if (!isDecimalDigit(ch) && (ch !== '.')) {
        return undefined;
    }
```

And if that is the case, we need to process each following character as long as it is a decimal digit:

```
    number = '';
    if (ch !== '.') {
        number = getNextChar();
        while (true) {
            ch = peekNextChar();
            if (!isDecimalDigit(ch)) {
                break;
            }
            number += getNextChar();
        }
    }
```

Since we want to support floating-point numbers, potentially we will see a decimal point coming (for example, for 3.14159, up to now only we're processing the 3). If that is the case, we need to loop again and process all the digits after the decimal point:

```
    if (ch === '.') {
        number += getNextChar();
        while (true) {
            ch = peekNextChar();
            if (!isDecimalDigit(ch)) {
                break;
            }
            number += getNextChar();
        }
    }
```

Supporting scientific notation with exponents means we may see an "e" after those digits. For example, if we are supposed to scan 6.62606957e-34, the previous code will get us only up to 6.62606957. We need to process the "e," and more digits after the exponent sign. Note that there can be a plus or a minus sign as well:

```
    if (ch === 'e' || ch === 'E') {
        number += getNextChar();
        ch = peekNextChar();
        if (ch === '+' || ch === '-' || isDecimalDigit(ch)) {
            number += getNextChar();
            while (true) {
                ch = peekNextChar();
                if (!isDecimalDigit(ch)) {
                    break;
                }
                number += getNextChar();
            }
        } else {
            throw new SyntaxError('Unexpected character after exponent sign');
        }
    }
```

The exception is needed because we want to tackle invalid numbers such as 4e.2 (there cannot be a decimal point after the exponent sign) or even just 4e (there must be some digits after the exponent sign).

If we want to consume a math expression and produce a list of tokens represented by the expression, we need a function that recognizes and gets the next token. This is easy, since we have three individual functions that can handle a number, an operator, or an identifier:

```
function next() {
    var token;

    skipSpaces();
    if (index >= length) {
            return undefined;
        }

        token = scanNumber();
        if (typeof token !== 'undefined') {
            return token;
        }

        token = scanOperator();
        if (typeof token !== 'undefined') {
            return token;
        }

        token = scanIdentifier();
        if (typeof token !== 'undefined') {
            return token;
        }

        throw new SyntaxError('Unknown token from character ' + peekNextChar());
    }
```

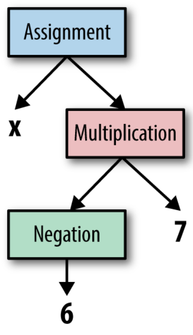
Syntax Parser and Syntax Tree

The stream of tokens produced by the lexer does not give us enough information to compute the math expression. Before we can evaluate the expression, an abstract syntax tree (AST) corresponding to the expression needs to be constructed. This procedure is commonly known as *syntactic analysis*, and it is usually carried out by a *syntax parser*.

Consider the following expression:

```
x = -6 * 7
```

The associated syntax tree for this expression is depicted in the following diagram.



A popular technique to construct the syntax tree is *recursive-descent parsing*. In such a parsing strategy, we go top down and match the possible parse tree from the highest level. For this particular problem, the simplified grammar of the math expression we want to handle is written as the following (in Backus-Naur Form):

```
Expression ::= Assignment

Assignment ::= Identifier '=' Assignment | Additive

Additive ::= Multiplicative | Additive '+' Multiplicative |
  Additive '-' Multiplicative

Multiplicative ::= Unary | Multiplicative '*' Unary | Multiplicative '/' Unary

Unary ::= Primary | '-' Unary

Primary ::= Identifier | Number | '(' Assignment ')' | FunctionCall

FunctionCall ::= Identifier '(' ')' | Identifier '(' ArgumentList ')'

ArgumentList := Expression | Expression ',' ArgumentList
```

The following code walkthrough illustrates the process of matching the expression from the topmost level (`Expression`). The lexer itself comes from the implementation of the lexical analyzer shown earlier. The main entry point for the parsing looks like this:

```
function parse(expression) {
    var expr;

    lexer.reset(expression);
    expr = parseExpression();

    return {
        'Expression': expr
    };
}
```

From this, we go to the main `parseExpression` function, which is surprisingly simple. This is because our syntax implies only a variable assignment as an expression. For other languages with more elaborate control flow (branching, loops, etc.) or some form of DSL, assignment may not be the only form of expression:

```
function parseExpression() {
    return parseAssignment();
}
```

For the subsequent `parseFoo` variants, we need a function that can match an operator. If the incoming operator is the same as the expected one, then it returns `true`:

```
function matchOp(token, op) {
    return (typeof token !== 'undefined') &&
        token.type === T.Operator &&
            token.value === op;
    }
```

An example form of assignment is `x = 42`. However, we also want to tackle cases where the expression is as plain as `42`, or a nested assignment such as `x = y = 42`. See if you can understand how the following implementation of `parseAssignment` handles all the three cases (hint: recursion is a possibility):

```
    function parseAssignment() {
        var token, expr;

        expr = parseAdditive();

        if (typeof expr !== 'undefined' && expr.Identifier) {
            token = lexer.peek();
            if (matchOp(token, '=')) {
                lexer.next();
                return {
                    'Assignment': {
                        name: expr,
                        value: parseAssignment()
                    }
                };
            }
            return expr;
        }

        return expr;
    }
```

The function `parseAdditive` processes both addition and subtraction—that is, it creates a binary operator node. There will be two child nodes, the left and right ones. They represent the two subexpressions, further handled by `parseMultiplicative`, to be added or subtracted:

```
    function parseAdditive() {
        var expr, token;

        expr = parseMultiplicative();
        token = lexer.peek();
        while (matchOp(token, '+') || matchOp(token, '-')) {
            token = lexer.next();
            expr = {
                'Binary': {
                    operator: token.value,
                    left: expr,
                    right: parseMultiplicative()
                }
            }
            token = lexer.peek();
        };
        return expr;
    }
```

The same logic follows for `parseMultiplicative`. It handles both multiplication and division:

```
    function parseMultiplicative() {
        var expr, token;

        expr = parseUnary();
        token = lexer.peek();
        while (matchOp(token, '*') || matchOp(token, '/')) {
            token = lexer.next();
            expr = {
                'Binary': {
                    operator: token.value,
                    left: expr,
                    right: parseUnary()
                }
            };
            token = lexer.peek();
        }
        return expr;
    }
```

Before we check the details of `parseUnary`, you may wonder why `parseAdditive` is called first, and then `parseMultiplicative`. This is done in order to satisfy the operator precedence requirement. Consider the expression `2 + 4 * 10`, which actually evaluates to `42` (multiply 4 by 10, then add 2) rather than `60` (add 2 to 4, then multiply by 10). This is possible only if the topmost node in the syntax tree is the *binary operator* `+`, which has two child nodes: the left one is just the number 2, and the right one is actually another binary operator, `*`. The latter holds two numbers as the corresponding child nodes, 4 and 10.

To handle a negation, like `-42`, we use the concept of unary operation. In the syntax tree, this is represented by a unary operator node and it has only one child node (hence the name). While negation is one form of unary operation, we also need to take into account the unary positive operator, as in `+42`. Thanks to the function's recursive nature, expressions like `----42` or even `-+-+42` can be handled without any problem as well. The code to handle the unary operation is as simple as the following:

```
    function parseUnary() {
        var token, expr;

        token = lexer.peek();
        if (matchOp(token, '-') || matchOp(token, '+')) {
            token = lexer.next();
            expr = parseUnary();
            return {
                'Unary': {
```

```
                operator: token.value,
                expression: expr
            }
        };
    }

    return parsePrimary();
}
```

Now here comes one of the most important functions of all: `parsePrimary`. First of all, let's consider the four possible forms of primary node:

- An identifier (basically referring to a variable in this context)--for example, `x`
- A number—for example, `3.14159`
- A function call—for example, `sin(0)`
- Another expression enclosed in brackets—for example, `(4 + 5)`

Fortunately, deciding whether the incoming tokens will form one of these possibilities is rather easy, as we just need to examine the token type. There is only ambiguity between an identifier and a function call, which can be solved if we peek at the next token (i.e., whether it is an open bracket or not). Without further ado, here is the code:

```
function parsePrimary() {
    var token, expr;

    token = lexer.peek();

    if (token.type === T.Identifier) {
        token = lexer.next();
        if (matchOp(lexer.peek(), '(')) {
            return parseFunctionCall(token.value);
        } else {
            return {
                'Identifier': token.value
            };
        }
    }

    if (token.type === T.Number) {
        token = lexer.next();
        return {
            'Number': token.value
        };
    }

    if (matchOp(token, '(')) {
        lexer.next();
        expr = parseAssignment();
```

```
            token = lexer.next();
            if (!matchOp(token, ')')) {
                throw new SyntaxError('Expecting )');
            }
            return {
                'Expression': expr
            };
        }

        throw new SyntaxError('Parse error, can not process token ' + token.value);
    }
```

Now the remaining part is `parseFunctionCall`. If we see an example of a function call like `sin(0)`, it basically consists of a function name, open bracket, function argument, and close bracket. It is important to realize that there can be more than one argument (`foo(1, 2, 3)`) or no argument at all (`random()`), depending on the function itself. For simplicity, we split out the handling of the function argument to `parseArgumentList`. Here are both functions for your pleasure:

```
    function parseArgumentList() {
        var token, expr, args = [];

        while (true) {
            expr = parseExpression();
            if (typeof expr === 'undefined') {
                break;
            }
            args.push(expr);
            token = lexer.peek();
            if (!matchOp(token, ',')) {
                break;
            }
            lexer.next();
        }

        return args;
    }

    function parseFunctionCall(name) {
        var token, args = [];

        token = lexer.next();
        if (!matchOp(token, '(')) {
            throw new SyntaxError('Expecting ( in a function call "' + name + '"');
        }

        token = lexer.peek();
        if (!matchOp(token, ')')) {
            args = parseArgumentList();
        }

        token = lexer.next();
        if (!matchOp(token, ')')) {
            throw new SyntaxError('Expecting ) in a function call "' + name + '"');
        }

        return {
            'FunctionCall' : {
                'name': name,
                'args': args
            }
        };
    }
```

Voilà! That's all our parser code. When combined properly into a functional object, it is just about 200 lines of code, supporting various math operations with proper precedences, brackets, variables, and function calls.

Tree Walker and Expression Evaluator

Once a syntax tree is obtained, evaluating the expression associated with it is surprisingly easy. It is simply a matter of walking the tree, from the topmost syntax node through all children, and executing a specific instruction related to the type of each syntax node. For example, a binary operator node means that we need to add (or subtract, or multiply, or divide) the two values obtained from each child node. Looking at the previous example:

```
x = -6 * 7
```

the generated syntax tree as a JavaScript object is:

```
{
    "Expression": {
        "Assignment": {
            "name": {
                "Identifier": "x"
            },
            "value": {
                "Binary": {
                    "operator": "*",
                    "left": {
                        "Unary": {
                            "operator": "-",
                            "expression": {
                                "Number": "6"
                            }
                        }
                    },
                    "right": {
                        "Number": "7"
                    }
                }
            }
            }
        }
    }
```

The code to interpret this JSON-formatted tree is quite straightforward. Let's start from the leaf, such as a number (we assume from here on that `node` points to the current node we need to evaluate):

```
    if (node.hasOwnProperty('Number')) {
        return parseFloat(node.Number);
    }
```

For a unary operation node, we need to evaluate the child node first and then apply the unary operation, either `+` or `-`:

```
    if (node.hasOwnProperty('Unary')) {
        node = node.Unary;
        expr = exec(node.expression);
        switch (node.operator) {
        case '+':
            return expr;
        case '-':
            return -expr;
        default:
            throw new SyntaxError('Unknown operator ' + node.operator);
        }
    }
```

A binary node is handled similarly—we just need to process both child nodes for the left and right side of the operator:

```
    if (node.hasOwnProperty('Binary')) {
        node = node.Binary;
        left = exec(node.left);
        right = exec(node.right);
        switch (node.operator) {
        case '+':
            return left + right;
        case '-':
            return left - right;
        case '*':
            return left * right;
        case '/':
            return left / right;
        default:
            throw new SyntaxError('Unknown operator ' + node.operator);
        }
    }
```

Before we continue to tackle variable assignment, let's take a step back and consider the concept of *evaluation context*. For this purpose, we define the context as an object that holds the variables, constants, and function definitions. When we evaluate an expression, we also need to pass a context so that the evaluator knows where to fetch the value of a variable, store a value to a variable, and invoke a certain function. Keeping the context as a different object promotes the separation of logic: the interpreter knows nothing about the context, and the context does not really care how the interpreter works.

In our evaluator, the simplest possible context is:

```
context = {
    Constants: {},
    Functions: {},
    Variables: {}
}
```

A slightly more useful context (that can be used as a default) is:

```
context = {

    Constants: {
        pi: 3.1415926535897932384,
        phi: 1.6180339887498948482
    },

    Functions: {
        abs: Math.abs,
        acos: Math.acos,
        asin: Math.asin,
        atan: Math.atan,
        ceil: Math.ceil,
        cos: Math.cos,
        exp: Math.exp,
        floor: Math.floor,
        ln: Math.ln,
        random: Math.random,
        sin: Math.sin,
        sqrt: Math.sqrt,
        tan: Math.tan
    },

    Variables: {}
}
```

We still do not have any variables (because the context is freshly created), but there are two common constants ready to use. The difference between a constant and a variable in this example is very simple and obvious: you cannot change a constant or create a new one, but you can do both with a variable.

With the context and its variables and constants ready, now we can handle identifier lookup (e.g., in an expression like `x + 2`):

```
if (node.hasOwnProperty('Identifier')) {
    if (context.Constants.hasOwnProperty(node.Identifier)) {
        return context.Constants[node.Identifier];
    }
    if (context.Variables.hasOwnProperty(node.Identifier)) {
        return context.Variables[node.Identifier];
    }
    throw new SyntaxError('Unknown identifier');
}
```

Assignment (like `x = 3`) works the other way around, though we have to ensure that we process only variable assignment and not constant override:

```
if (node.hasOwnProperty('Assignment')) {
    right = exec(node.Assignment.value);
    context.Variables[node.Assignment.name.Identifier] = right;
    return right;
}
```

Finally, the remaining function node is handled as follows. Basically, the function arguments (if any) are prepared in an array and then passed to the actual function. Note that in our default context, we simply wire a bunch of functions to the methods of the built-in `Math` object:

```
if (node.hasOwnProperty('FunctionCall')) {
    expr = node.FunctionCall;
    if (context.Functions.hasOwnProperty(expr.name)) {
        args = [];
        for (i = 0; i < expr.args.length; i += 1) {
            args.push(exec(expr.args[i]));
        }
        return context.Functions[expr.name].apply(null, args);
    }
    throw new SyntaxError('Unknown function ' + expr.name);
}
```

What if we want to have a custom function, maybe because it is not supported by the `Math` object? It can’t be easier: all we have to do is define the function for the context. As an example, let’s implement `sum`, which adds all the numbers passed in the argument. Since we’re dealing with a function that may have a variable number of arguments, we use a special `arguments` object instead of named parameters:

```
context.Functions.sum = function () {
    var i, total = 0;
    for (i = 0; i < arguments.length; i += 1) {
        total += arguments[i];
    }
    return total;
}
```

Final Words

The simple example presented here can be easily extended or modified for a wide range of domain-specific languages. For a simpler language, the lexer can be implemented as a collection of regular expressions. Alternatively, a simple state machine is often suitable in many cases. On the other hand, a language with a complex grammar may require a deeper recursive-descent parsing. In some cases, it is more convenient to handle some of the recursive aspect by using a stack-based shift and reduce.

Some languages are known to have peculiar cases that complicate both the lexer and the parser. For example, doing lexical analysis on JavaScript code is notoriously difficult because the symbol / is ambiguous: it can signify either a division operator or the beginning of a regular expression. In addition to that, the famous automatic semicolon insertion feature requires various parts of the parser to take that into account wherever it is mandated by the language specification. It is instructive to learn how various parsers handle these types of edge cases.

Happy parsing!

CHAPTER EIGHT

Evolution

Rebecca Murphey

In March 2009, Paul Irish published a blog post, "Markup-based Unobtrusive Comprehensive DOM-ready Execution," describing a solution to a pesky problem familiar to every newcomer to the world of client-side JavaScript at the time: executing only the code that was required for a given page.

Back in 2009, it was common for client-side JavaScript developers to just put all of their code—for all of their pages—inside one giant `$(document).ready()` callback; some were a bit cleverer, and tested for the presence of an element with a certain ID in order to determine the page they were on. A newcomer to such code struggled mightily to mentally parse hundreds of lines where function declarations, anonymous functions, and long chains of jQuery methods intermingled.

The method proposed in this blog post was simple: put a class on the `<body>` element, and then use a simple helper function to look up a corresponding initialization method in an application object:

```
UTIL = {
  loadEvents : function () {
    var bodyId = document.body.id;

    $.each(document.body.className.split(/\s+/), function (i, className) {
      UTIL.fire(className);
      UTIL.fire(className,bodyId);
    });
  },

  fire : function (func, funcname, args) {
    var namespace = APP;  // indicate your obj literal namespace here

    funcname = (funcname === undefined) ? 'init' : funcname;
```

```
      if (
        func !== '' &&
        namespace[func] &&
        typeof namespace[func][funcname] == 'function'
      ) {
        namespace[func][funcname](args);
      }
    }
  };

  $(document).ready(UTIL.loadEvents);
```

The code, written by a self-taught and largely unknown frontend developer with a degree in technical communications, was mediocre. The idea, though, was transformative, especially for a community with lots of similarly self-taught developers: if we could organize our code somehow, maybe writing ever-bigger JavaScript applications didn't have to be such a messy affair.

A few months after Paul wrote his post, I published "Using Objects to Organize Your Code" and gave a talk on the same topic at the 2009 jQuery Conference. My post suggested having one object per "feature" (a piece of functionality on a page), and encapsulating all of the functionality of that feature in methods on that object. For example, a list of email messages might be one feature; a list of mailboxes might be another.

I know for a fact that I had only the most cursory understanding of `.call()` and `.apply()` at the time, and though `$.proxy` didn't exist yet, I'm not sure I'd have fully understood it if it did. John Resig had posted his micro-templating snippet a year before, and I'd read *JavaScript: The Good Parts*, yet the post contained no consideration of client-side templating or being able to create *instances* of these feature objects.

"If I tried to think of the simplest JavaScript thing I could write a post about," my friend Alex Sexton said to me recently (in the nicest way possible, because he is Alex), "I'd still never come up with something as simple as that."

And yet this too seemed to have a transformative effect in the still largely self-taught JavaScript community of the time. Not only could we break our code down per page; we could also break it down *per component*, and those components could be clearly represented by distinct pieces of code.

We could even…not to get crazy here, but…we could put those pieces of code in separate files, using a global object as a namespace, right? Granted, loading all of those separate files as `<script>` tags on our page during development would be a pain, and every time we added a new file we'd have to update our list of `<script>` tags, but our server-side code could probably help us out there. Really, though, it sure would be nice if JavaScript had a module system for asynchronously loading these features, wouldn't it? Especially because we'd need to concatenate all of these files for production.

Backbone

Dojo changed everything for me; Backbone changed everything for everyone else. While there are plenty of criticisms to be made of Backbone, to be sure, the 619 unminified lines that made up version 0.1.0 once again transformed the way that we thought about JavaScript application development. It gave us easy-to-understand building blocks without trying to provide answers to every problem under the sun—perhaps Dojo's major failing.

Backbone's tiny file size ensured there would be few accusations of bloat; its utter simplicity and the way it embraced jQuery paradigms made it an easy leap for moderately skilled jQuery developers. Its unopinionated approach meant it was as easy to sprinkle some Backbone onto an existing app as it was to start a new Backbone app from scratch (and equally easy to get yourself into trouble if your own opinions turned out to be bad). Its inclusion of a router,[1] a mainstay of server-side frameworks like Rails and Django—well, let's just say it took Paul's "Markup-based Unobtrusive Comprehensive DOM-ready Execution" to a whole new level. It also, interestingly, provided a gateway for traditionally server-side developers who had long been turned off by the tangled mess of "get some elements and do something with them."

Perhaps the happiest thing of all for me, though, was that Backbone made it normal and easy to `new` up an instance of a `View`, largely throwing the misguided jQuery plugin paradigm out the window. Sure, you had to bring your own templating (and rendering) solution to the `Backbone.View` party, but Underscore was there to help by default; composition wasn't as straightforward as in Dojo, but it wasn't hard either.

Adding things like attach points and lifecycle methods and memory-safe teardown was straightforward enough, too. For those ready to make the leap, Backbone became something of a framework-building library. Indeed, on my current project, Backbone serves as the scaffolding upon which we've built a much more elaborate client-side application development framework, without having to labor over the basics.[2]

New Possibilities

Love it or otherwise—and my opinions are definitely mixed—the arrival of Backbone left no doubt about two things: one, JavaScript's days as a toy language were firmly

1 Sammy.js included a router years before Backbone, but its more opinionated approach meant that it never gained widespread adoption.

2 Is it good that lots of projects are building their own frameworks on top of a 6.4 KB library? For now I think yes; we are still learning what we need from frameworks, and we're a long way from a one-size-fits-all answer (or even a three-sizes-fit-most answer). I'm hopeful that this will change over the next 12–18 months, especially considering everything that has transpired in the few years covered by this chapter.

behind it; and two, it was time for JavaScript developers at all levels to understand and embrace the client-side JavaScript app.

That impossible situation of arbitrary components interacting in arbitrary ways? That's actually been a core requirement of the two major projects I've worked on in the last couple of years.

At Toura, we were creating configuration-driven, offline-capable PhoneGap apps. Customers would use a content management system to design their application, and the content management system would spit out a JSON config spelling out what was on each page. A page might include a photo gallery, a caption area, a text area, the ability to favorite things, or any number of other features. Every application ran the same JavaScript code; that code would, at runtime, read the config file to figure out how to set itself up, and what to show users as they moved through the application.

Our solution there was what I dubbed a "capability"; a page could have any number of capabilities, and each capability dictated how a set of components would interact with one another. The controller for a page was essentially generated dynamically based on the capabilities that the config said the page should have; the code within a capability handled passing messages from one component to another.

At Bazaarvoice, the situation is similar: our customers use a configuration tool to decide how they want their application to behave and which features should be enabled, and that configuration tool generates a JSON config. We use that config to figure out exactly what to put in the built JavaScript for that customer—a big improvement over the approach we took at Toura—and we also use that config to wire up the relationship between components at runtime, using what we call "outlets." A component's configuration might look something like this:

```
"reviewSummary" : {
  "features" : {
    // an object describing the features that are enabled for the component
  },
  "outlets" : {
    "showreviews" : [{
      "component" : "reviewContentList",
      "event" : "scrolltocontent"
    }],
    "showquestions" : [{
      "component" : "questionContentList",
      "event" : "scrolltocontent"
    }],
    "filtercontent" : [{
      "component" : "reviewContentList",
      "event" : "filtercontent"
    }]
  }
}
```

At runtime, we read the configuration for the component and wire up its relationship with the other components; for this example, we initialize an `Outlet` so that when the `reviewSummary` component triggers its `showReviews` method, we ensure the `scrolltocontent` method is triggered on the `reviewContentList`:

```
var Outlet = function (options) {
  this.targetComponent = options.targetComponent;
  this.originatingComponent = options.originatingComponent;
  this.target = options.target;
  this.key = options.key;

  var event = this.event = 'outlet:' + this.key;

  if (this.target.event) {
    this.originatingComponent.on(event, this._eventHandler());
  }
};

Outlet.prototype._eventHandler = function () {
  var targetComponent = this.targetComponent;

  if (!targetComponent) {
    return;
  }

  return function () {
    var args = [ targetComponent.scopeEvent(target.event) ].concat(
      [].slice.call(arguments)
    );

    targetComponent.trigger.apply(targetComponent, args);

    return;
  };
};
```

This *could* be considered a variation on the dreaded direct communication between components, but realistically, it's more of a mini-controller that's created on the fly at runtime, brokering communication between components without either component requiring direct knowledge of the other.

I'm not just mentioning this because it's what I've been working on of late; I think it's the next thing we'll hash out as a JavaScript community, once we get done with or bored of fighting about which framework is The Best.

Imagine a page where you use a calendar component written by Jenn, and an invitation list component written by Adam, and a DSL with which you can dictate that when an item in the invitation list triggers its `accept` event, the calendar's `schedule` method gets called with data about the invitation—and neither component needs direct knowledge of the other. Web components, inspired directly by Dojo's templated widgets of yore, are a baby step in that direction. I hope we take more steps, and bigger ones, and soon.

CHAPTER NINE

Error Handling

Nicholas Zakas

If you're like me, you probably don't think much about how you'll handle errors until they start popping up on a regular basis. Programmers tend to write code as if there will never be any errors, and then spend the rest of their time tracking down errors they've caused. This inclination is totally natural. No one starts out on a project thinking about all the ways they will do something wrong. You start out believing you know the right way to do it and then are unpleasantly surprised as errors start to pop up.

But what if you changed the thought process? Instead of assuming that errors won't happen, assume that they will. How would that change your approach to writing code? That's precisely what this chapter is about: thinking about and planning for the errors that will inevitably occur in your JavaScript.

Assume Your Code Will Fail

> If an error is possible, someone will make it. The designer must assume that all possible errors will occur and design so as to minimize the chance of the error in the first place, or its effects once it gets made.
>
> —Donald A. Norman, *The Design of Everyday Things*

The first step to good error handling is to accept that your code will fail at some point. That may be because of improper use, or proper use that you didn't plan for. Regardless, your code will fail at some point in time. Given that, what can you do to make your code more robust? What are the things you can do, right now, to make your code easier to deal with when it fails?

Throwing Errors

When I was younger, the most befuddling part of programming languages was the ability to create errors. My first reaction to the `throw` operator in Java was, "Well, that's stupid, why would you ever want to cause an error?" Errors were the enemy to me, something I sought to avoid, so the ability to cause an error seemed like a useless and dangerous aspect of the language. I thought it was dumb to include the same operator in JavaScript, a language that people just didn't understand in the first place. Now, with a great deal of experience under my belt, I'm a big fan of throwing my own errors. When done properly, this can lead to easier debugging and code maintenance.

In programming, an error occurs when something unexpected happens. Maybe the incorrect value was passed into a function, or a mathematical operation had an invalid operand. Programming languages define a base set of rules that, when deviated from, result in errors so that you can fix the code. Debugging would be nearly impossible if errors weren't thrown and reported back to you. If everything failed silently, it would take you a long time to notice that there was an issue in the first place, let alone to isolate and fix it. Errors are a developer's friends, not enemies.

The problem with errors is that they tend to pop up in unexpected places and at unexpected times. To make matters worse, the default error messages are usually too terse to really explain what's gone wrong. JavaScript error messages are notoriously uninformative and cryptic (especially in old versions of Internet Explorer), which only compounds the problem. Imagine if an error popped up with a message that said, "This function failed because this happened." Instantly, your debugging task would become easier. This is the advantage of throwing your own errors.

It helps to think of errors as built-in failure cases. It's always easier to plan for a failure at a particular point in code than it is to anticipate failure everywhere. This is a very common practice in product design, not just in code. Cars are built with crumple zones, areas of the frame that are designed to collapse in a predictable way when impacted. Knowing how the frame will react in a crash—which parts will fail—allows the manufacturers to ensure passenger safety. Your code can be constructed in the same way.

You can throw an error by using the `throw` operator and providing an object to throw. Any type of object can be thrown, but an `Error` object is the most typical to use:

```
throw new Error("Something bad happened.")
```

When you throw an error in this way, and the error isn't caught via a `try-catch` statement, the browser will display the error text in its typical way. For Internet Explorer, this means a little icon appears in the lower-left corner of the browser window, and a dialog with the error text is displayed when that icon is double-clicked; Firefox will show the error in the Web Console; Safari, Chrome, and Opera output the error into the Web Inspector. In other words, it's treated the same way as an error that you didn't throw.

The difference is that you get to provide the exact text to be displayed by the browser. Instead of just line and column numbers, you can include any information that you'll need to successfully debug the issue. I recommend that you always include the function name in the error message, as well as the reason why the function failed. Consider the following function:

```
function addClass(element, className){
    element.className += " " + className;
}
```

This function's purpose is to add a new CSS class to the given element (a very common method in JavaScript libraries). But what happens if `element` is `null`? You'll get a cryptic error message such as "object expected." Then, you'll need to look at the execution stack (if your browser supports it) to actually locate the source of the problem. Debugging becomes much easier if you throw your own error:

```
function addClass(element, className){
    if (element !== null && typeof element.className === "string"){
        element.className += " " + className;
    } else {
        throw new Error("addClass(): First argument must be a DOM element.");
    }
}
```

Discussions about accurately detecting whether an object is a DOM element aside, this method now provides better messaging when it fails due to an invalid `element` argument. Seeing such a verbose message in your error console immediately leads you to the source of the problem. I like to think of throwing errors as leaving Post-it notes for myself as to why something has failed.

As a bonus, JavaScript engines add a `stack` property to any `Error` object that is thrown. The `stack` property is a string containing a formatted stack trace leading up to the error being thrown. Here's an example value for `stack`:

```
Error
    at foo (test.js:2:24)
    at test.js:2:7
```

While each JavaScript engine has a slightly different representation of stack information in the `stack` property, the information available inside is roughly the same: the type of error, the filename in which the error originated, line and column numbers, and function names. This information is very useful should you decide to log your JavaScript errors for later investigation.

When to Throw Errors

Understanding how to throw errors is just one part of the equation; understanding when to throw errors is the other. Since JavaScript doesn't have type or argument checking, a lot of developers incorrectly assume that they should implement that for every function. Doing so is impractical and can adversely affect the overall script's performance. The key is to identify parts of the code that are likely to fail in a particular way and throw errors only there. In short, throw errors only where errors will already occur.

If a function is only ever going to be called by known entities, error checking is probably not necessary (this is the case with private functions); if you cannot identify all the places where a function will be called ahead of time, then you'll likely need some error checking and will be more likely to benefit from throwing your own errors. The best place for throwing errors is in utility functions: those functions that are a general part of the scripting environment and may be used in any number of places. This is precisely the case with JavaScript libraries.

All JavaScript libraries should throw errors from their public interfaces for known error conditions. YUI/jQuery/Dojo/etc. can't possibly anticipate when and where you'll be calling their functions. It's their job to tell you when you're doing stupid things. Why? Because you shouldn't have to debug into their code to figure out what went wrong. The call stack for an error should terminate in the library's interface, no deeper. There's nothing worse than seeing an error that's 12 functions deep into a library; library developers have a responsibility to prevent this from happening.

This also goes for private JavaScript libraries. Many web applications have their own proprietary JavaScript libraries, built either with or in lieu of the well-known public options. The goal of libraries is to make developers' lives easier, and they do so by providing an abstraction away from the dirty implementation details. Throwing errors helps to keep those dirty implementation details hidden safely away from developers.

Types of Errors

ECMA-262 specifies seven error object types. These are used by the JavaScript engine when various error conditions occur and can also be manually created:

`Error`
: Base type for all errors. Never actually thrown by the engine.

`EvalError`
: Thrown when an error occurs during execution of code via `eval()`.

`RangeError`
: Thrown when a number is outside the bounds of its range—for example, trying to create an array with –20 items (`new Array(-20)`). These occur rarely during normal execution.

`ReferenceError`
: Thrown when an object is expected but not available—for instance, trying to call a method on a `null` reference.

`SyntaxError`
: Thrown when the code passed into `eval()` has a syntax error.

`TypeError`
: Thrown when a variable is of an unexpected type—for example, `new 10` or `"prop" in true`.

`URIError`
: Thrown when an incorrectly formatted URI string is passed into `encodeURI`, `encodeURIComponent`, `decodeURI`, or `decodeURIComponent`.

You can create and throw each of these error types at any time in JavaScript by invoking the constructor of the same name, such as:

```
throw new TypeError("Unexpected type.");

throw new ReferenceError("Bad reference.");

throw new RangeError("That's out of range.");
```

The error types thrown most frequently by developers are `Error`, `RangeError`, `ReferenceError`, and `TypeError`. The other error types are very specific to use cases inside of the JavaScript engine, so it doesn't make sense to use them in your code (even though there's nothing stopping you).

All error types inherit from `Error`, so checking the type with `instanceof Error` doesn't give you any useful information. By checking for the more specific error types, you get more robust error handling:

```
var error = new TypeError("Not my type.");

console.log(error instanceof Error);        // true
console.log(error instanceof TypeError);    // true
```

Of course, if you only ever throw errors using the built-in JavaScript error types, it becomes difficult to distinguish between errors thrown by the engine and errors you threw intentionally. That's where custom errors come in.

Custom Errors

In large applications, it's useful to create your own error type. Using a custom error type allows you to easily tell the difference between an error that was thrown intentionally and an unexpected error that the browser throws. You can create a custom error type easily by inheriting from `Error` and following a simple pattern:

```
function MyError(message){
    this.message = message;
}

MyError.prototype = Object.create(Error.prototype);
```

There are two important parts of this code: 1) the `message` property, which is necessary for browsers to know the actual error string, and 2) setting the `prototype` to an instance of `Error`, which identifies the object as an error to the JavaScript engine. Now, you can throw an instance of `CustomError` and have the browser respond as if it were a native error:

```
throw new MyError("Something really bad happened!");
```

If you want to throw a lot of different types of error but still want to distinguish between the errors you throw and the native errors, then you can use your custom error as a base for other custom error types, such as:

```
function MyError(message)
    this.message = message;
}

MyError.prototype = Object.create(Error.prototype);

function MissingArgumentError(message) {
    this.message = message;
}

MissingArgumentError.prototype = Object.create(MyError.prototype);

function NotFunnyError(message) {
    this.message = message;
}

NotFunnyError.prototype = Object.create(MyError.prototype);
```

In this example, `MissingArgumentError` and `NotFunnyError` both inherit from `MyError` (which, in turn, inherits from `Error`). Due to this inheritance, you can easily separate out error handling using an `if` statement:

```
if (error instanceof MyError) {
    // handle MissingArgumentError and NotFunnyError
} else {
        // handle native error types
    }
```

Distinguishing between the errors you threw and the errors thrown by the JavaScript engine is important, because you frequently want to treat them differently. As discussed earlier, throwing your own error indicates that this condition is a known possibility (unlike native errors, which are frequently unexpected).

Handling Errors

Errors should be easy to detect, they should have minimal consequences, and, if possible, their effects should be reversible.

—Donald A. Norman, *The Design of Everyday Things*

ECMA-262 defines a `try-catch-finally` construct similar to those found in other languages. The basic idea is to place code that might throw an error in the `try` clause and code to handle that error in the `catch` clause. The optional `finally` clause runs in either case. The basic syntax is:

```
try {
    // some code that might throw an error
} catch(error) {
    // handle an error that was thrown
} finally {
    // optionally run code regardless of error
}
```

When an error occurs inside of the `try` clause, execution stops and is resumed inside of the `catch` clause. The thrown error is passed into the `catch` clause as an additional variable. This happens regardless of the error type, and it's up to you to look at the error object to determine what type of error occurred and how to respond appropriately. For example:

```
try {
    functionThatMightThrowError();
} catch(error) {
    if (error instanceof MyError) {
        // handle custom error
    } else {
        // handle native error
    }
}
```

It's also possible to omit the `catch` clause completely and just use a `finally` clause, such as:

```
try {
    functionThatMightThrowError();
} finally {
        // do whatever you want
    }
```

In this case, an error will cause execution to stop inside of the `try` clause and go immediately into the `finally` clause. If an error doesn't occur, then all of the statements inside of the `try` clause are executed, and then the statements in the `finally` clause are executed. In either case, you are saying that there is no special functionality when an error occurs.

Realistically, you typically want a `catch` clause along with `try`, but you may also want `finally`. The `finally` clause runs no matter what, and that is true even if the `try` or `catch` clauses contain a `return` statement. Consider the following two functions:

```
function doSomething() {
    try {
        functionThatMightThrowError();
        return "success";
    } catch(error) {
        return "failure";
    } finally {
        return "finally";
    }
}

function doSomethingElse() {
    try {
        functionThatMightThrowError();
        return "success";
    } catch(error) {
        return "failure";
    }

    return "finally";
}

var result1 = doSomething();
var result2 = doSomethingElse();
```

The functions `doSomething` and `doSomethingElse` contain the same code, except that the former uses a `finally` clause and the latter does not. The difference in the behavior of the two functions is striking. The value of `result1` is always `"finally"`, regardless of whether an error occurs. That's because the `return` statement is skipped over in the `try` and `catch` clauses in favor of the one in the `finally` clause. The value of `result2`, on the other hand, will never be `"finally"`. That's because in the case of no error, the `return` statement in the `try` clause is used, while the `return` statement in the `catch` clause is used when an error happens. Those are the only two options in the function, and so the last `return` statement outside of the `try-catch` is not reachable. The value of `result2` will be `"success"` if there is no error and `"failure"` if there is an error.

There are some downsides to using `try-catch-finally`. First, you must know ahead of time whether or not some piece of code could potentially throw an error. While this may be easy to determine in some cases, it may not be so easy in other cases. Using `try-catch-finally` effectively, therefore, requires some upfront planning. Second, there is a performance hit for wrapping code in a `try-catch-finally` even when an error doesn't occur. As with many performance tips in JavaScript, however, this becomes important only if you find code that is running millions of times in a row—for code that is run a nominal number of times, the difference in execution time will not be apparent.

Global Error Handling in Browsers

In a web browser, all uncaught errors bubble up to a top-level event handler called `window.onerror`. This event handler receives four arguments: the error message, the URL that raised the error, a line number, and a column number. As an added feature, returning `true` from `window.onerror` tells the browser that the error was handled and there's no need to show it to the user. For example:

```
window.onerror = function(message, url, line, col) {
    logError(message, url, line, col);
    return true;
};
```

In this example, the error message is being logged and `true` is returned to indicate that the error has been handled properly.

In late 2013, the HTML5 specification was changed to specify a fifth argument to `window.onerror`, which is the actual error object. Prior to that point, there was no access to the error object inside of `window.onerror`. At the time of writing only Chrome and Firefox have implemented this change, but it should be making its way into other browsers. With the error object being passed in, you are now free to look at the additional information attached to it:

```
window.onerror = function(message, url, line, col, error) {
    logError(message, url, line, col, error.stack);
    return true;
};
```

This example also extracts the `stack` information from the error that was thrown.

The `window.onerror` event handler should be used in web applications to ensure that you always know when any JavaScript error occurs. Since it's unlikely that you'll be aware of all possible combinations that could cause a JavaScript error in your application, using this event handler gives you a safe way to monitor errors without being overly intrusive to developers.

Global Error Handling in Node.js

Node.js has a similar mechanism for catching errors globally. The `process` object fires an event called `uncaughtException` whenever a JavaScript error occurs that is not handled in some other way. You can listen for the event and receive the JavaScript error object using code such as:

```
process.on("uncaughtException", function(err) {
    log(err);
});
```

If an error is handled by this event handler, then the Node.js process will not automatically exit (any uncaught exceptions will cause such an exit). Some suggest that you should always call `process.exit` inside of this event handler; however, whether or not you choose to do so depends largely on your application and how easy it is to recover from such an error without affecting the overall state of the application. You should use your best judgment in determining the correct course of action when an uncaught error occurs, whether that be to log the error, exit the process, restart the process, or something completely different.

Node.js also has a feature called *domains* that allows you to set up an error handler for uncaught exceptions that occur during the execution of specific code. To do so, use code such as:

```
var d = require("domain").create();
d.on("error", function(err){
    log(err);
});

d.run(function(){
    /* some code that might throw an error */
});
```

The basic idea of this example is that you can place some code that might cause an error within the call to `run` on a domain. Then, any errors that occur within that code will cause the `error` event to fire on that domain. You can listen for the `error` event and respond appropriately to the error.

Domains are a fairly new concept in Node.js and so may change considerably in the future. Best practices around domain usage are still being developed and discussed, so make sure you take the time to explore whether domains fit your error handling strategy before committing to their use.

Summary

Errors and error handling aren't topics that developers love to talk about, but ultimately the job comes down to finding and eliminating sources of error. The first step in the process is always to assume that your code will fail and plan to deal with that failure. Figure out how you will know when a particular type of error has occurred and what you should do to resolve it (if anything).

Throwing your own errors can be a powerful tool in this regard. When you throw an error, you can specify the exact information that you need to track down its source. Creating a custom error type as a base allows you to easily tell the difference between a JavaScript error thrown by the engine and one thrown by you (or your teammates). You can then use constructs like `try-catch-finally` to monitor for errors.

In larger applications, you should also listen for uncaught exceptions. Both browsers and Node.js allow you to listen for these exceptions in one location, allowing you to log or otherwise handle the errors as they occur.

Remember, most errors are not appropriate to be shown to your users, so be sure to have user-friendly error messages (or no error messages at all, if you can recover easily).

CHAPTER TEN

The Node.js Event Loop

Jonathan Barronville

If you're using Node.js, chances are that you started tinkering with it after getting tired of hearing everyone rave about this platform for building fast servers using JavaScript.

You went on the Node.js website and read this: "Node.js uses an event-driven, non-blocking I/O model that makes it lightweight and efficient, perfect for data-intensive real-time applications that run across distributed devices."

Now, if you have experience building event-driven servers, this will already make enough sense to you. (And this chapter is probably not for you!) However, if you're like me, when you read this you probably decided to quit programming, because here's a platform for developers and I'm a developer, but I'm too stupid to understand why I should care!

Okay, maybe that was a little bit of an exaggeration. You didn't quit programming, and you're not stupid for not understanding why you should care about Node.js.

My goal is that, by the end of this chapter, you will be proud to tell the world you understand how the Node.js event loop works and start receiving your much-deserved LinkedIn endorsements for "Node.js event loop."

Event-Driven Programming

At a high level, event-driven programming is when a system expresses its interest in a particular set of events, provides a way to be alerted when said events happen, and responds to them using callbacks.

What do these terms mean, though? An *event* is some change in a system's state. The term *callback* can mean different things depending on the type of system, but in the case of JavaScript, it simply means a closure whose function will be invoked once a particular event happens.

Under the hood, Node.js uses a native library called *libuv* for listening to events and invoking the necessary callbacks. To do this, libraries and frameworks like libuv have an *event loop*, which is essentially a loop for handling events that usually runs forever.

To make some of this a little bit more concrete, here's a snippet of the underlying C++ code (lines 3761–3773 in */src/node.cc*) that handles starting and managing Node.js's event loop (as of commit 0df5e1c049 of Node.js):

```
bool more;
do {
  more = uv_run(env->event_loop(), UV_RUN_ONCE);
  if (more == false) {
    EmitBeforeExit(env);

    // Emit `beforeExit` if the loop became alive either after emitting
    // event, or after running some callbacks.
    more = uv_loop_alive(env->event_loop());
    if (uv_run(env->event_loop(), UV_RUN_NOWAIT) != 0)
      more = true;
  }
} while (more == true);
```

Let's quickly break down the two parts of this code you should care about right now. First:

```
do {...} while(...);
```

This says to execute everything in the `do` block and continue to do so until the condition in `while(...)` evaluates to `false`. And second:

```
more = uv_run(env->event_loop(), UV_RUN_ONCE);
```

`uv_run(...)` can be considered to be the most important function in libuv, because it's actually what starts and runs the event loop. Without going too deep into the technical aspects of libuv from a C++ standpoint, all you need to know for now is that this invocation of `uv_run` returns `0` (which is a "falsy" value in C++) when there are no more things to do, which would make `more` be `false`. If it returns a "truthy" value, `more` will be `true`.

Whoa, a couple of paragraphs into the chapter and I've already thrown C++ code at you! Well, as it turns out, when we talk about Node.js's event loop, what we're really talking about is a libuv loop, so I think it helps to show a little bit of the low-level implementation. The rest of this chapter will be more high-level, I promise!

Asynchronous, Nonblocking I/O

All modern operating systems have event notification systems built in. These event notification systems tend to work differently across platforms. This is one of the main issues libuv solves. It provides cross-platform high-level abstractions to handle events, while handling all of the crazy not-so-fun platform differences under the hood.

Often, you'll hear the words *asynchronous* and *nonblocking* being used in discussions about Node.js and its scalability—but what do they mean, exactly?

Let's say you're writing a TCP server. You create a simple loop that accepts and processes new connections on every iteration. You then realize you have a problem: every time your server is handling a connection, it blocks until data is available to be read from the connection. This is bad, because you can't process any other connections! One way to fix this problem is to instead use an operating system hook to request that the operating system let you know when data is available. This is *asynchronous* because when data is available, you'll be notified by the event notification system, and it's *nonblocking* because your loop will never be blocked from processing other connections.

Although we won't touch on this, keep in mind that another common model for building servers is using *operating system threads*, which usually means creating a thread for every client/connection. Using operating system threads is not only difficult to scale, but actually pretty hard to understand and do right.

One of the things I find really cool about asynchronous, nonblocking I/O is how the model is easy to explain when compared to many real-life examples. The best example of this, in my opinion, is ordering food at a restaurant. You go to your favorite fast food restaurant and you get in line. Once it's your turn, your server takes your order. Your order goes through and your server gives you a number, so that they can call you back when your burger is ready. This is an extremely efficient model because the server can quickly process many orders, whereas the other option would've been for the server to take your order, wait for it to be prepared while other customers wait in line, and finally move to the next person in line once your burger is ready.

Node.js programs work similarly to the restaurant ordering example. Let's look at an example:

```
'use strict'

var http = require('http')

function serverRequestHandler (serverRequest, serverResponse) {
  serverResponse.writeHead(200, {'content-type': 'text/plain'})
  endServerResponse(serverResponse)
}

function endServerResponse(serverResponse) {
      serverResponse.end('Hello, world!\n')
    }

    var httpServer = http.createServer(serverRequestHandler)

    httpServer.listen(3620, '127.0.0.1')

    console.log('Server running at http://127.0.0.1:3620.')
```

Let's break this example down. First, we import the `http` model:

```
    var http = require('http')
```

Next up are the `serverRequestHandler` and `endServerResponse` functions:

```
    function serverRequestHandler (serverRequest, serverResponse) {
      serverResponse.writeHead(200, {'content-type': 'text/plain'})
      endServerResponse(serverResponse)
    }

    function endServerResponse(serverResponse) {
      serverResponse.end('Hello, world!\n')
    }
```

`serverRequestHandler` is the callback for handling requests to our server. When called, it will be passed a "request" object and a "response" object. The request object contains all the necessary data about the current request and provides facilities for accessing that data. The response object provides facilities for constructing and sending responses. One interesting thing to note here is that `serverRequestHandler` calls `endServerResponse`. This is interesting because when `serverRequestHandler` is called, it won't be running in the same environment it was defined in. `endServerResponse` shouldn't be available, but because of closures in JavaScript, all of the state available to `serverRequestHandler` where it was defined will continue to be available to it no matter where it is called.

Next, we create a new server and pass in the handler to use for requests. When we express handler will be cached, and every time a request is sent to our server it will be pushed onto the queue of callbacks to call:

```
    var httpServer = http.createServer(serverRequestHandler)
```

Finally, we express our interest to begin accepting connections on the port 3620 and hostname 127.0.0.1:

```
    httpServer.listen(3620, '127.0.0.1')
```

This is the most important piece of the code. By expressing this interest, the system will start watching for requests, triggering the appropriate events when necessary and invoking the necessary callbacks.

Concurrency

Node.js is single-threaded. Before I talk about what that means, let's talk about concurrency. It seems a lot of folks get the wrong idea about concurrency, assuming that it's exactly the same thing as parallelism. The terms are actually related, but they're not the same.

Concurrency is when a set of tasks can start, run, and complete in overlapping time periods. The tasks may never run at the same time, but they could. *Parallelism*, on the other hand, is when a set of tasks are running at the same time.

When I say that Node.js is *single-threaded*, what I mean is that the Node.js event loop is managing at most one thread at any point in time, which of course means a single call stack. By that same logic, an important thing to note is that the event loop can only ever do one thing at once.

So while you're able to write highly concurrent servers with Node.js, your servers can process only one request at a time. This is a difficult but important distinction to understand when thinking about Node.js concurrency.

Let's use the earlier HTTP server example to understand what I mean by concurrency here. When a request comes to our server, the "request" event is triggered with the request data, which causes our request handler to be pushed onto the task queue. Once the call stack is free and the event loop is free of things to process, our request handler will be invoked. The server is able to handle many requests concurrently, because every request is processed quickly and independently without blocking.

Adding Tasks to the Event Loop

So let's say you want to give the event loop a little bit of work to do. Is there a way to do that efficiently? Yes!

`process.nextTick` to the rescue! `process.nextTick` enables you to provide the Node.js event loop a callback to invoke immediately in the next iteration, or *tick*, of the event loop:

```
function runCPUIntensiveTask(data) {
  if (data === null) {
    return
  }
  // Do some CPU-intensive work ...
  process.nextTick(function () {
    runCPUIntensiveTask(newData)
  })
}
```

In this example, `runCPUIntensiveTask` is a function that does something CPU-intensive recursively. However, rather than simply calling the function recursively, which would essentially block the event loop, the recursion is handled in the event loop instead. This allows the event loop to do whatever it has to do, invoke `runCPUIntensiveTask`, do anything else it has to do, and repeat the process, without ever being blocked.

Those are the basics. Understanding the Node.js event loop is key to being effective with Node.js, so I hope I was able to clarify the confusing parts for you!

CHAPTER ELEVEN

JavaScript Is...

Sara Chipps

> Science is what we understand well enough to explain to a computer. Art is everything else we do.
>
> —Donald Knuth

I understand there are people in this world who do not like JavaScript. I've gotten into enough late-night battles about CoffeeScript to have heard all the vitriol. I've been generally unmovable about this. I think curly braces are elegant, I think semicolons are enchanting, and I think duck typing is adorable. I thought writing this chapter would be a good opportunity to share my favorite things about JavaScript.

JavaScript Is Dynamic

JavaScript takes advantage of virtual machines for just-in-time (JIT) compilation (see V8, Node.js, and Spider Monkey for some examples). JavaScript makes excellent use of closures by having variables that exist both on the global level and the functional level.

Consider this function:

```
function CheckForPrefix(name){
  var prefix = "Dr.";

  if (name.indexOf(prefix) == -1)
    return function AddPrefix(){
      return prefix + name;
    }
}
```

JavaScript's closures give us the ability to reference variables defined in the containing function. In this instance it enables us to keep the separation of concerns while not repeating ourselves at the same time.

JavaScript has an `eval` function that allows us to concatenate values and evaluate them at runtime. `eval` makes things slower, as it adds a compilation step, so it's to be used sparingly; however, it does allow us to create macros, which are another dynamic language feature.

JavaScript Can Be Static

I feel like JavaScript gives us great luxury with dynamic types, but let's not get ahead of ourselves. Recently I have been pushing a lot of C++ because of a project, and just today I was complaining about types. The person I was talking to said, "I love statically typed languages because you don't have to write tests." Joking about tests aside, static languages offer a lot of safety around compile time. Many people have written static wrappers around JavaScript. Type checking with JavaScript is an abstraction—if you were to write it into your program, it would look something like this:

```
switch(typeof input) {
  case('number'):
    if(Math.Floor(foo) == foo)
      console.log("This variable is an integer");
    else
      console.log("This variable is a floating point");
    break;
  case('string'):
    console.log("This variable is a string");
    break;
  case('object'):
    if(foo instanceof Array)
      console.log("This variable is an array");
}
```

There is a page that lives on the CoffeeScript Wiki that has a list of languages and libraries that compile to JavaScript. At the time I'm writing there are 16 languages and libraries listed in the statically typed section, including Dart, as Google continues a static language => JavaScript path. Libraries like asm.js use a compiler that runs before the JIT to make sure that the library doesn't interfere with web performance.

JavaScript Is Functional

When I meet developers who say they specialize in Scala, or Haskell, or even F#, my reaction is always "Wow, that is legit." Functional languages have a reputation for attracting brilliant developers who solve difficult problems, like managing millions of stock market trades and making sure all the tweets get to the other side.

JavaScript is just as legit, though, since it incorporates first-class functions, and that makes it functional. I love them because they're like, "Surprise, function!"—you think they're variables, but they're not.

I personally like this new take on a switch function as a cool implementation of a first-class function (monetary conversions are from the time of writing):

```
function getLocalTotal(country, price) {
  var currency = {
    'dollar': function(price) {
      return price;
    },
    'pound': function(price) {
      return price * 0.61      // Else
    },
    'peso': function(price) {
      return price * 13.27
    },
    'kroner': function(price) {
      return price * 5.48    }
  };

  if (currency[country.currancy])
    return currency[country.currency](price);
  else
    return currency.dollar(price);
}
```

JavaScript Does Everything

When I was first introduced to JavaScript, it was a functional language that animated your website. There were no libraries like jQuery; there was no in-browser debugging; there were no web servers or desktop applications. When I asked people to share the coolest thing they had seen powered by JavaScript, my mind was blown. The answers were everything from a box that analyzes liquid you put in it to a WebGL version of the game Quake, quadcopters that double as web servers, a library that lets you query your genome, and, last but not least, the inclusive, innovative and ever-expanding Node.js community. JavaScript helped me find hardware, which is my latest passion since JavaScript. I look forward to seeing where it takes us next.

CHAPTER TWELVE

Coding Beyond Logic

Daryl Koopersmith

0. The Basement

“Check it out!” I sat in a beat-up beige armchair in the basement of a college apartment building, staring at a jumble of charts, icons, and code. Two physicists-in-training beamed down at me. One of them said, “It’s pretty simple. You know what it does, don’t you?”

I paused, eyebrows raised, scanning the lines back and forth. The code read like a chalkboard full of high school algebra.

“No,” I shrugged. “You wrote a few loops and built a graph, but I have absolutely no idea what this code actually does.”

As they explained the graph, I couldn’t stop thinking. Where were the well-named variables? Where were the comments? Who taught them to code?

1. Quine’s Paradox

William Van Orman Quine was a logician who explored the limits of self-reference (along with many other philosophical and logical concepts) throughout the 20th century. In his essay “The Ways of Paradox,” he explores how indirect self-reference can be applied to the liar’s paradox (“The following statement is false. The preceding statement is true.”). Aside from reading like a convoluted interview question, Quine’s paradox unintentionally laid the foundation for a programming puzzle that has persisted for decades:

> “Yields a falsehood when appended to its own quotation” yields a falsehood when appended to its own quotation.

> This sentence specifies a string of nine words and says of this string that if you put it down twice, with quotation marks around the first of the two occurrences, the result is false. But that result is the very sentence that is doing the telling. The sentence is true if and only if it is false.

The paradox, in turn, was the subject of a conversation between the Tortoise and Achilles in *Gödel, Escher, Bach: an Eternal Golden Braid*, written by Douglas Hofstadter in 1979. Within the conversation, the Tortoise coins the verb *quine*:

> Tortoise: It's very earnest stuff, in my opinion. In fact this operation of preceding some phrase by its quotation is so overwhelmingly important that I think I'll give it a name.
>
> Achilles: You will? What name will you dignify that silly operation by?
>
> Tortoise: I believe I'll call it "to quine a phrase", to quine a phrase.

Hofstader went on to win a Pulitzer, and sometime in the decade between 1988 and 1998 the conversation inspired the definition of *quine* (the noun), which was added to the Jargon File, a comprehensive guide to programmer's slang:

> quine: /kwi:n/, n. A program that generates a copy of its own source text as its complete output. Devising the shortest possible quine in some given programming language is a common hackish amusement.

In time, programmer Gary P. Thompson II stumbled upon this definition and The Quine Page was born: a website boasting a collection of quines in over 50 languages (and beveled badges that touted "Lynx enhanced" and "vi powered" to boot). Thompson credits the entry in the Jargon File as his inspiration, and in a delightfully circular turn, the entry in the Jargon File now links to The Quine Page as "amusing."

A quine is the ouroboros of programs: its stated purpose is to replicate itself as output. But as a form, quines are an artistic puzzle, an outlet for unabashed creative expression. A quine is focused purely on the code itself. It lives to be dissected.

Here is a quine inspired by Geoffrey A. Swift's entry from The Quine Page. It initially reads as a bit of a jumble, so we'll step through it together:

```
var a = []; a[0] = 'var a = []; ';
a[1] = 'a[';
a[2] = '] = ';
a[3] = '\'';
a[4] = '\\';
a[5] = ';';
a[6] = '';
a[7] = 'for(var i = 0; i < a.length; i++) console.log((i == 0 ? a[0] : a[6])\
+ a[1] + i + a[2] + a[3] + ((i == 3 || i == 4) ? a[4] : a[6]) + a[i] + a[3] \
+ a[5] + (i == 7 ? a[7] : a[6]))'; for(var i = 0; i < a.length; i++) \
console.log((i == 0 ? a[0] : a[6]) + a[1] + i + a[2] + a[3] + \
((i == 3 || i == 4) ? a[4] : a[6]) + a[i] + a[3] + a[5] + (i == 7 ? a[7] : \
a[6]))
```

This is a pure quine: a JavaScript program that rebuilds itself using simple constructs. First, we declare an array whose indices are mapped to the strings necessary to output the program. Then, we iterate over a loop to actually produce the output.

Consider the body of the loop when `i` equals `1`:

```
console.log((i == 0 ? a[0] : a[6]) + a[1] + i + a[2] + a[3] +
  ((i == 3 || i == 4) ? a[4] : a[6]) + a[i] + a[3] + a[5] +
  (i == 7 ? a[7] : a[6]))
```

Note that the failure case of each ternary conditional is `a[6]`, which maps to the empty string. Since the ternaries evaluate to the empty string when `i` equals `1`, we can remove them:

```
console.log(a[1] + i + a[2] + a[3] + a[i] + a[3] + a[5])
```

Substituting in strings yields:

```
console.log('a[' + i + '] = '+ '\'' + a[i] + '\'' + ';')
```

which evaluates to the second line:

```
a[1] = 'a[';
```

The three conditionals inside the loop allow us to print the initial array declaration, escape slashed values, and print the `for` loop, respectively.

All JavaScript quines aren't that complicated, though. JavaScript lets us cheat a little. Consider this quine by James Halliday:

```
(function f() { console.log('(' + f.toString() + ')()') })()
```

Much simpler. The crux of the line lies in `f.toString`. Calling `toString` on a function returns the source of that function as a string (and maintains identical spacing). To produce a function that outputs its source when called, we would write:

```
function f() { console.log(f.toString()) }
```

However, running this as a program would produce no output because the function still needs to be invoked. We wrap the function in parentheses to indicate to the interpreter that the function should be treated as an expression, and invoke it with the following set of parentheses:

```
(function f() { console.log(f.toString()) })()
```

However, this still outputs the source of the function without the added parentheses. To make this program a quine, we have to account for the parentheses when printing the output as well:

```
(function f() { console.log('(' + f.toString() + ')()') })()
```

This example also reveals self-reference as a form of recursion. We can transform the quine into an infinite loop with the slightest change. Transform `console.log` into `eval`, and suddenly it runs forever:

```
(function f() { eval('(' + f.toString() + ')()') })()
```

Not all implementations of the quine are so accessible. This dense quine by Ben Alman originally fit into a tweet (followed by "#quine," of course):

```
!function $(){console.log('!'+$+'()')}()
```

Ben's quine is conceptually identical to the quine we just analyzed, but compacted and obscured as much as possible. The parentheses wrapping the function are traded for a leading `!` operator—both ensure the function is interpreted as an expression. We then rename the function to `$` simply to throw a little spice into the mix—something that feels like it should be an operator, but really isn't. Inside the `console.log` statement, we reuse the `!` operator (this is a quine, after all) and concatenate it with the `$` method and trailing invocation parentheses. The `toString` method is nowhere to be found: concatenating a string with a function implicitly calls the function's `toString` method.

While the techniques behind the previous two quines were identical, each program's tone is considerably different. The first quine is utilitarian and accessible. The second is sparse and coy. But when it comes to the art of the quine, these programs are just the tip of the iceberg when compared to the work of Yusuke Endoh.

Yusuke Endoh is a self-described "Quine programmer," and a contributor to the Ruby programming language. His best-known quine is the Quine Relay, a Ruby program that circles through 50 programming languages before arriving back at its Ruby origins. Another program is a quine with a twist—the Radiation Hardened Quine will regenerate the original program even when a single character is removed from the source at random. He has written a quine with an embedded rotating globe (the "Qlobe") and another in Piet, a language in which programs take on the appearance of abstract art.

Endoh's work pushes the boundaries of the quine as a form, subverting it into a vessel for creative expression. His quines are maximalist, each one a mysteriously self-supporting house of cards. The rigidity of the form aids the reading, because the intent and structure of a quine are limited and nonnegotiable. The program is designed to replicate itself. Code goes in, code comes out. As in a scientific experiment, establishing these constant controls facilitates a deeper, more focused analysis of the variables that remain. In the case of quines, this allows us to focus on the author's intent, and how the quine fulfills its purpose.

While quines themselves aren't particularly useful in everyday programming (unless, of course, you're Yusuke Endoh), they are an elegant illustration of programming within constraints. Despite the quine's simple requirements, satisfying the demands of the form often forces the programmer to shirk best practices, contorting the code until it can spit itself out again.

Sometimes, optimizing for constraints will violate some tenet of conventional wisdom. You might balk at first (and you certainly might feel dirty writing the code), but it might be the most effective way to solve the problem at hand. Every program makes trade-offs.

A classic example is unrolling a loop:

```
for (var i = 0; i < 100; i++) {
  doSomething(i)
}
```

In an ideal environment, the cost of each iteration of the loop (increasing `i` and inspecting whether `i` is less than `100`) would be negligible compared to the cost of executing the `doSomething`. This *should* be optimal. But if, for some reason, iterating over a loop is expensive, then you need to come up with alternatives:

```
for (var i = 0; i < 100; i += 4) {
  doSomething(i)
  doSomething(i + 1)
  doSomething(i + 2)
  doSomething(i + 3)
}
```

This is not nearly as graceful as the previous loop: we've squashed four iterations of the previous loop together to form a single unrolled iteration. But if this loop were considerably more performant than the former (in our hypothetical scenario), we would opt for the unrolled code.

Thankfully, this type of problem can often be transparently solved without your knowledge: if unrolled loops are more efficient, when a compiler or interpreter encounters the first loop, it will be unrolled behind the scenes.

In JavaScript, a more practical example is looping over an array:

```
function loop(items) {
  for (var i = 0; i < items.length; i++) {
    doSomething(i)
  }
}
```

For a time, the Web was filled with articles with titles like "You'll Never Believe How This Web Developer Loops Over Arrays" that advocated for storing the array length in a variable first:

```
function loop(items) {
  for (var i = 0, len = items.length; i < len; i++) {
    doSomething(i)
  }
}
```

For a little while, this change was a micro-optimization in most browsers. But now performance swings the other way: browser engines recognize the patterns in the first example, and optimize accordingly.

Now imagine JavaScript arrays didn't have a `length` property. Imagine a `count` method instead that iterated over the entire array every time it was called (don't imagine too hard—this is how it works in PHP):

```
function loop(items) {
  for (var i = 0, len = items.count(); i < len; i++) {
    doSomething(i)
  }
}
```

In this case, storing the length of the array in a variable is significantly more efficient. It's worth the lower readability.

If we consider the broader picture, though, in almost every case this micro-optimization was unnecessary and only complicated the code. Our community preached patterns without understanding or explaining the thought processes behind them.

2. The Conjecture

Shinichi Mochizuki is both the world's only "inter-universal geometer" and the only person who currently understands what that means. To the rest of us, Mochizuki is a mathematician. For almost two decades, Mochizuki worked to solve the *abc* conjecture, a proposition that, if proven, would establish unknown fundamental properties of prime numbers. In August 2012, he released a 512-page solution to the conjecture.

Three years later his solution remains unverified, and not for lack of trying. This might be partially due to the fact that Mochizuki invented an entirely new branch of mathematics, "inter-universal geometry," to write the proof—which, in turn, is built atop concepts from a complex, little-known branch of mathematics called anabelian geometry. And if you were hoping for any inroads into the thousands of pages of mathematical literature, you're out of luck. Mochizuki practically refuses to lecture on the topic, with only a handful of seminars offered at his home university in Japan.

It's no wonder the proof is yet to be deciphered. To put this in perspective, this is akin to an engineer requesting to merge a single commit that rewrites the Linux kernel in a new programming language that he invented solely for that commit with no explanation or comment. Even if appears to run perfectly, it's not getting merged.

To Mochizuki, the *abc* conjecture has been proven. To the rest of the world, it remains unsolved. When asked about Mochizuki's proof, math professor Cathy O'Neil said, "You don't get to say you've proved something if you haven't explained it. A proof is a social construct. If the community doesn't understand it, you haven't done your job."

While JavaScript doesn't have the same burden of proof as mathematics (and we're lucky that's the case), software operates in much the same fashion. As an author, you must identify your audience: the maintainers, the contributors, the readers. If they don't understand your code, how effective can it be in the long run?

Software is a social construct. A pull request requires understanding and approval from project maintainers before it can be merged. Documentation is only useful if it's comprehensible. An API must be explained before it can be used.

Even if you're the only author of the code, the same needs apply—they're just easier to ignore. While you have more insight into your own thought process than anyone else, memory degrades over time. It's not about when the code is written, but the weeks, months, and years that follow. When you inevitably decide to refactor in six months, you'll be glad you added that documentation.

We create social conventions to govern our code: common design patterns, style guides, and shared philosophies. The desire for understanding drives the unrelenting march toward the consistent and thorough, fuels the fires of style guides, and ensures that every name endures just enough bikeshedding and pedantry to emerge slightly more sensible.

But in a codebase where each line of code perfectly adheres to a guide, it's still apparent when code is written by multiple authors. The giveaways are the snippets the guide doesn't specify—whether it's the whitespace between operators or when methods should return early. Everyone has a calling card. It's the broadest strokes that are the most telling: we each think about problems differently. Certain patterns are our crutches. Maybe it's using factories, or a preferred method of inheritance. Maybe it's taking a more functional approach.

The code is just the surface—a reflection of its contributors, of their ideas and culture. The thought processes behind those contributions are easily lost, archived in code reviews and meeting notes, cobwebbed in the corners of institutional memory. In 1994, mathematician William Thurston published "On Proof and Progress in Mathematics", a survey of the culture of mathematics in the form of a scholarly paper. Thurston observed how formalism drowned out institutional thought processes as published work spread:

> There is another effect caused by the big differences between how we think about mathematics and how we write it. A group of mathematicians interacting with each other can keep a collection of mathematical ideas alive for a

> period of years, even though the recorded version of their mathematical work differs from their actual thinking, having much greater emphasis on language, symbols, logic and formalism. But as new batches of mathematicians learn about the subject they tend to interpret what they read and hear more literally, so that the more easily recorded and communicated formalism and machinery tend to gradually take over from other modes of thinking.

So it is in JavaScript. Shared knowledge is the bedrock of programming. Every program is built atop an ever-growing mountain of abstractions. The most accessible programs are those that leverage and extend our collective knowledge, utilizing familiar patterns. They're aware of the audience.

As engineers, our goal is to minimize the distance between the thought process and the final result. Why was this solution selected? Where are the pitfalls? Institutional memory is an inevitable byproduct of writing code. It's impossible to perfectly express our thinking, but it's important to try.

3. Peer Review

It may seem that Mochizuki could fill the role of a human Quine's paradox. His work is largely self-referential, and his it's-proven-because-I-say-so attitude is directly in conflict with a community that prides itself on correctness, formality, and peer review. And if he cannot convince the world of his proof, he risks his work becoming as ornamental as a quine. But time passes on, and there's hope for him yet.

In December 2014, Mochizuki posted a progress report on his website (which is truly glorious and worth visiting in its own right, with a very serious Mochizuki gazing into the distance, surrounded by bubbles and animated GIFs of clip art textbooks, phi glyphs, and lightbulbs). In the report he repeatedly praises his three collaborators, criticizes every other practicing mathematician, and liberally uses italics.

After claiming that "the verification of [the proof] is, for all practical purposes, complete" and underscoring "the quite essential importance of reading through the papers carefully," Mochizuki dismisses every other mathematician as "a complete novice with respect to the mathematics" surrounding the proof, and "simply not qualified to issue a definitive (i.e., mathematically meaningful) judgment."

He's a little prickly. With three assenting reviewers, his work will likely be verified. But even once his work is verified, Mochizuki risks the same problem. If no one understands his work, the theory is purely decorative—an elaborate exercise in self-reference. Despite his repeated admonishments and sardonic tone, it appears Mochizuki understands this, and has settled on a strategy that will ensure his work becomes a part of the mathematical canon:

> In light of the present state of affairs, the only reasonable course of action lies in taking a long-term approach to promoting the dissemination of [the proof] by cultivating a collection of researchers, one by one.

Now he's being reasonable.

The arbiter of successful code is not the author, but the reader and the passage of time. What matters are the people who interact with your code, including you.

Over the past two decades, JavaScript has exploded into a sprawling ecosystem of programmers, browsers, libraries, servers, frameworks, and standards bodies. It's through that chaos—through sharing and building off of one another's ideas—that JavaScript has flourished. As Paul Ford wrote,

> Making a new language is hard. Making a popular language is much harder still and requires the smile of fortune. And changing the way a popular language works appears to be one of the most difficult things humans can do, requiring years of coordination to make the standards align. Languages are large, complex, dynamic expressions of human culture.

Through our collective work we've overcome pedantry and dead ends and comments that begin with "Actually…" to create an ecosystem where our language and our thought processes can evolve. But we only grow when we listen to one another.

Which brings me back to the basement. I smile when I remember how my physicist friends thought I could instantly know what they were trying to accomplish just from seeing a graph and a handful of variables.

What I initially experienced was the disconnect between the program as a logical construct and the program as an expression of culture. My eyes parsed loops, variables, and methods, but I failed to understand the purpose and context of the code. The expectation that I could understand the intent behind code simply because I knew how to program was flattering, but impossible.

And who taught them to code? As it turns out, it was mathematicians and scientists. Over time, I came to realize that just because their program didn't conform to my expectation of what a program should look like didn't mean it was wrong or ineffective. It just meant that I was not the intended reader, and in my arrogance I judged them for it. Maybe I shouldn't have been so hasty. The fact that they shared their work and I understood their explanation was success enough.

Even quines aren't meant to sit in isolation. The quine is not only an exercise for the author, but an exercise for the reader as well. Quines are meant to be shared. That is the true purpose of The Quine Page—as Thompson writes, "These programs are written for educational purposes, to further one's computer science skills. It seems rather paradoxical to create such a program and not share your unique solution."

As engineers, we continually reinforce the notion that a program is a logical construct, but a program is also a means of communication, and no means of communication can perfectly convey intent. There's more to code than just logic. Programming is lossy, and therein lies its beauty.

CHAPTER THIRTEEN

JavaScript Is Cutieful

Graeme Roberts

All This Loose Beauty

JavaScript is beautiful, and I can say that with certainty, for beauty is in the eye of the beholder, and in the hands of that beholder is a language soft as plasticine that will mold and change at will and not protest against maltreatment, and will trust the code it's given as though it were the word of God.

Beauty, I suppose, is a rather personal thing. Some detest the rain, and others cry in it and feel the force of life itself in drops as it hammers on their skin.

And so, in JavaScript.

Some fear and remonstrate against a single use of anything not sanctioned by ancestral coders in whom they've learned to place their trust. As though experimentation, thought, invention, play, discovery, and learning were some capital offense.

Some of the authors in this book have quite deftly shown the beauty in the structure and the safety of those parts of the language we've accepted as permissible.

Others have, with expertise and confidence and thorough knowledge of their craft, demonstrated elegance and let us see the power and succinctness that these oft lambasted features can grant a learned user.

The Absurdity of Dalí

The work of Dalí is often absurd, disturbing, strange, enchanting, comical, and surprising—and it is beautiful.

I want to explore a part of JavaScript that captures that same kind of beauty for me.

Dalí’s JavaScript

```
Array.apply(null, { length: 10 }).map(eval.call, Number)
// → [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

Before reading on, just look at that for a little while. Really drink it in. Just look at what is going on there, until soft clocks inside your head melt all down your repl.

Is This Beauty Just Ugly?

I can see why detractors might say that this is quite the opposite of beauty. They may know that in other languages they would be treated to something friendly like `range(10)`.

But try to let that feeling pass. Or, better yet, let’s pretend we really do have such a thing as `Math.range(10)`, and let that free us to appreciate the details of the true beauty here.

An Unfortunate Necessity

`Array.apply(null, { length: 10 } )` and `Array.apply(null, Array(10))` are two versions of an unfortunate, and slightly offensive, necessity in order to generate an array of 10 `undefined`s rather than an array of length 10 without defined contents.

The Beauty Is in the Madness

So now we’ve got ourselves a tasty array like this:

```
[undefined, undefined, undefined, undefined, undefined, undefined, undefined,
 undefined, undefined, undefined]
```

Nothing rousing about that. “I’m not roused,” I pretend to imagine that I hear you think.

But, my dear, the beauty that I promised is to be found within the madness of `map(eval.call, Number)`.

The use of `eval` is not actually relevant. I chose it because it’s short (and because typing `eval` makes me feel mischievous). Any function will do; `function(){}.call` works just as well. It’s the `call` we want!

Let’s Have a Wee Look at map

You probably already know all this. Sorry.

`Array.prototype.map` expects a `callback` function as an argument, which it will call in turn for each value in the array. To `callback` it passes arguments `value`, `index`, and the full `array`. In the end, it returns a new array with the result of `callback` in place, or each `item`.

There's a beautiful JS goldmine in passing native constructors to `[].map`:

```
["10", "20", "lol"].map(Number)
// → [10, 20, NaN]
```

Adorable, right? I love this with `[].filter` too; it's just some real cute-as-a-box-of-buttons JavaScript:

```
["10", "20", "lol"].map(Number).filter(Boolean)
// → [10, 20]
```

Hello, thisArg

`[].map` also takes a second argument, and that's a calling context, or `thisArg`. This is the object that will serve as `this` when `callback` is running on the values:

```
["lol", "wow", "ok"].map(function(string) {
  return this[string];
}, {
  lol: "yeah!",
  wow: "alright!",
  ok: "cool!"
});
// → ["yeah!", "alright!", "cool!"]
```

So it's essentially like `[].map(fn.bind(obj))`.

Okay! So That's a Bunch of Stuff I Already Knew About [].map—Now What?

Well, let's have another look at Dalí's `map`:

```
Array(null, { length: 10 }).map(eval.call, Number)
```

Rad. We're covered up to about character 38 now. So what else is going on here?

calling All Cars

So, `Function.prototype.call` takes a `thisArg`. It's thanks to it that we are able to indulge in such pleasures as the ever-merciful `var args = [].slice.call(arguments)`. Supplemental arguments will be passed to the function being called.

Number

`Number` takes one argument, and it tries earnestly to coax that argument into becoming a number:

```
Number(" 47 ")
// → 47
Number(true)
// 1
```

Now I Know Everything

One last look at this:

```
Array(null, { length: 10 }).map(eval.call, Number)
```

For each `undefined` in our array, `Function.prototype.call` is being called with a `this` context of `Number`, like:

```
Function.prototype.call.bind(Number, undefined, index, array);
```

which is like:

```
Number.call(undefined, index, array)
```

which equates to `Number` with a `this` context of `undefined` (which has no effect because `Number` doesn't utilize `this`) being passed the arguments `index` and `array`:

```
Number(index, array)
```

`Number` ignores the array and returns the number, delivering us a new array consisting of the indexes of the old array.

Wild

Now, this is clearly *completely insane*.

Like, *really mental*.

But it does make a quite flamboyant display of what I think is one of the most gorgeous and powerful mechanisms in sweet JavaScript: the ability to call any function you find lying about, in the context of any object you choose. And I really think that that is beautiful JavaScript.

But I'm one of the ones outside crying in the rain.

CHAPTER FOURTEEN

Functional JavaScript

Anton Kovalyov

Is JavaScript a functional programming language? This question has long been a topic of great debate within our community. Given that JavaScript's author was recruited to do "Scheme in the browser," one could argue that JavaScript was designed to be used as a functional language. On the other hand, JavaScript's syntax closely resembles Java-like object-oriented programming, so perhaps it should be utilized in a similar manner. Then there might be some other arguments and counterarguments, and the next thing you know the day is over and you didn't do anything useful.

This chapter is not about functional programming for its own sake, nor is it about altering JavaScript to make it resemble a pure functional language. Instead, this chapter is about taking a pragmatic approach to functional programming in JavaScript, a method by which programmers use elements of functional programming to simplify their code and make it more robust.

Functional Programming

Programming languages come in several varieties. Languages like Go and C are called *procedural*: their main programming unit is the procedure. Languages like Java and SmallTalk are object oriented: their main programming unit is the object. Both these approaches are imperative, which means they rely on commands that act upon the machine state. An imperative program executes a sequence of commands that change the program's internal state over and over again.

Functional programming languages, on the other hand, are oriented around expressions. Expressions—or rather, pure expressions—don't have a state, as they merely compute a value. They don't change the state of something outside their scope, and they don't rely on data that can change outside their scope. As a result, you should be able to substitute a pure expression with its value without changing the behavior of a program. Consider an example:

```
function add(a, b) {
  return a + b
}

add(add(2, 3), add(4, 1)) // 10
```

To illustrate the process of substituting expressions, let's evaluate this example. We start with an expression that calls our `add` function three times:

```
add(add(2, 3), add(4, 1))
```

Since `add` doesn't depend on anything outside its scope, we can replace all calls to it with its contents. Let's replace the first argument that is not a primitive value—`add(2, 3)`:

```
add(2 + 3 , add(4, 1))
```

Then we replace the second argument:

```
add(2 + 3, 4 + 1)
```

Finally, we replace the last remaining call to our function and calculate the result:

```
(2 + 3) + (4 + 1) // 10
```

This property that allows you to substitute expressions with their values is called *referential transparency*. It is one of the essential elements of functional programming.

Another important element of functional programming is *functions as first-class citizens*. Michael Fogus gave a great explanation of functions as first-class citizens in his book, *Functional JavaScript*. His definition is one of the best I've seen:

> The term "first-class" means that something is just a value. A first-class function is one that can go anywhere that any other value can go—there are few to no restrictions. A number in JavaScript is surely a first-class thing, and therefore a first-class function has a similar nature:
>
> - A number can be stored in a variable and so can a function:
>
>   ```
>   var fortytwo = function() { return 42 };
>   ```
>
> - A number can be stored in an array slot and so can a function:
>
>   ```
>   var fortytwos = [42, function() { return 42 }];
>   ```
>
> - A number can be stored in an object field and so can a function:
>
>   ```
>   var fortytwos = {number: 42, fun: function() { return 42 }};
>   ```
>
> - A number can be created as needed and so can a function:
>
>   ```
>   42 + (function() { return 42 })(); // => 84
>   ```

- A number can be passed to a function and so can a function:

  ```
  function weirdAdd(n, f) { return n + f() }

  weirdAdd(42, function() { return 42 }); // => 84
  ```

- A number can be returned from a function and so can a function:

  ```
  return 42;

  return function() { return 42 };
  ```

Having functions as first-class citizens enables another important element of functional programming: higher-order functions. A *higher-order function* is a function that operates on other functions. In other words, higher-order functions can take other functions as their arguments, return new functions, or do both. One of the most basic examples is a higher-order `map` function:

```
map([1, 2, 3], function (n) { return n + 1 }) // [2, 3, 4]
```

This function takes two arguments: a collection of values and another function. Its result is a new list with the provided function applied to each element from the list.

Note how this `map` function uses all three elements of functional programming described previously. It doesn't change anything outside of its scope, nor does it use anything from the outside besides the values of its arguments. It also treats functions as first-class citizens by accepting a function as its second argument. And since it uses that argument to compute the value, one can definitely call it a higher-order function.

Other elements of functional programming include recursion, pattern matching, and infinite data structures, although I will not elaborate on these elements in this chapter.

Functional JavaScript

So, is JavaScript a truly functional programming language? The short answer is no. Without support for tail-call optimization, pattern matching, immutable data structures, and other fundamental elements of functional programming, JavaScript is not what is traditionally considered a truly functional language. One can certainly try to treat JavaScript as such, but in my opinion, such efforts are not only futile but also dangerous. To paraphrase Larry Paulson, author of the *Standard ML for the Working Programmer*, a programmer whose style is "almost" functional had better not be lulled into a false sense of referential transparency. This is especially important in a language like JavaScript, where one can modify and overwrite almost everything under the sun.

Consider `JSON.stringify`, a built-in function that takes an object as a parameter and returns its JSON representation:

```
JSON.stringify({ foo: "bar" }) // -> "{"foo":"bar"}"
```

One might think that this function is pure, that no matter how many times we call it or in what context we call it, it always returns the same result for the same arguments. But what if somewhere else, most probably in code you don't control, someone overwrites the `Object.prototype.toJSON` method?

```
JSON.stringify({ foo: "bar" })
// -> "{"foo":"bar"}"

Object.prototype.toJSON = function () {
  return "reality ain't always the truth"
}

JSON.stringify({ foo: "bar" })
// -> ""reality ain't always the truth""
```

As you can see, by slightly modifying a built-in `Object`, we managed to change the behavior of a function that looks pretty pure and functional from the outside. Functions that read mutable references and properties aren't pure, and in JavaScript, most nontrivial functions do exactly that.

My point is that functional programming, especially when used with JavaScript, is about reducing the complexity of your programs and not about adhering to one particular ideology. Functional JavaScript is not about eliminating all the mutations; it's about reducing occurrences of such mutations and making them very explicit. Consider the following function, `merge`, which merges together two arrays by pairing their corresponding members:

```
function merge(a, b) {
  b.forEach(function (v, i) { a[i] = [a[i], b[i]] })
}
```

This particular implementation does the job just fine, but it also requires intimate knowledge of the function's behavior: does it modify the first argument, or the second?

```
var a = [1, 2, 3]
var b = ["one", "two", "three"]

merge(a, b)
a // -> [ [1, "one"], [2, "two"],.. ]
```

Imagine that you're unfamiliar with this function. You skim the code to review a patch, or maybe just to familiarize yourself with a new codebase. Without reading the function's source, you have no information regarding whether it merges the first argument into the second, or vice versa. It's also possible that the function is not destructive and someone simply forgot to use its value.

Alternatively, you can rewrite the same function in a nondestructive way. This makes the state change explicit to everyone who is going to use that function:

```
function merge(a, b) {
  return a.map(function (v, i) { return [v, b[i]] })
}
```

Since this new implementation doesn't modify any of its arguments, all mutations will have to be explicit:

```
var a = [1, 2, 3]
var b = ["one", "two", "three"]

merge(a, b) // -> [ [1, "one"], [2, "two"],.. ]

// a and b still have their original values.
// Any change to the value of a will have to
// be explicit through an assignment:
a = merge(a, b)
```

To further illustrate the difference between the two approaches, let's run that function three times without assigning its value:

```
var a = [1, 2]
var b = ["one", "two"]

merge(a, b)
// -> [ [1, "one"], [2, "two"] ]; a and b are the same
merge(a, b)
// -> [ [1, "one"], [2, "two"] ]; a and b are the same
merge(a, b)
// -> [ [1, "one"], [2, "two"] ]; a and b are the same
```

As you can see, the return value never changes. It doesn't matter how many times you run this function; the same input will always lead to the same output. Now let's go back to our original implementation and perform the same test:

```
var a = [1, 2]
var b = ["one", "two"]

merge(a, b)
// -> undefined; a is now [ [1, "one"], [2, "two"] ]
merge(a, b)
// -> undefined; a is now [ [[1,"one"], "one"], [[2, "two"],"two"] ]
merge(a, b)
// -> undefined; a is even worse now; the universe imploded
```

Even better is that this version of `merge` allows us to use it *as a value itself*. We can return the result of its computation or pass it around without creating temporary variables, just like we would do with any other variable holding a primitive value:

```
function prettyTable(table) {
  return table.map(function (row) {
    return row.join(" ")
  }).join("\n")
}
```

```
console.log(prettyTable(merge([1, 2, 3], ["one", "two", "three"])))
// prints:
//   1 "one"
//   2 "two"
//   3 "three"
```

This type of function, known as a *zip* function, is quite popular in the functional programming world. It becomes useful when you have multiple data sources that are coordinated through matching array indexes. JavaScript libraries such as Underscore and LoDash provide implementations of `zip` and other useful helper functions so you don't have to reinvent the wheel within your projects.

Let's look at another example where explicit code reads better than implicit. JavaScript —at least, its newer revisions—allows you to create constants in addition to variables. Constants can be created with a `const` keyword. While everyone else (including yours truly) primarily uses this keyword to declare module-level constants, my friend Nick Fitzgerald uses `const`s virtually everywhere to make clear which variables are expected to be mutated and which are not:

```
function invertSourceMaps(code, mappings) {
  const generator = new SourceMapGenerator(...)

  return DevToolsUtils.yieldingEach(mappings, function (m) {
    // ...
  })
}
```

With this approach, you can be sure that a `generator` is always an instance of `SourceMapGenerator`, regardless of where it is being used. It doesn't give us immutable data structures, but it *does* make it impossible to point this variable to a new object. This means there's one less thing to keep track of while reading the code.

Here's a bigger example of a functional approach to programming: a few weeks ago, I wrote a static site generator in JavaScript for the JSHint website and my personal blog. The main module that actually reads all the templates, generates a new site, and writes it back to disk consists of only three small functions. The first function, `read`, takes a path as an argument and returns an object that contains the whole directory tree plus the contents of the source files. The second function, `build`, does all the heavy work: it compiles all the templates and Markdown files into HTML, compresses static files, and so on. The third function, `write`, takes the site structure and saves it to disk.

There's absolutely no shared state between those three functions. Each has a set of arguments it accepts and some data it returns. An executable script I use from my command line does precisely the following:

```
#!/usr/bin/env node

var oddweb = require("./index.js")
var args   = process.argv.slice(2)

oddweb.write(oddweb.build(oddweb.read(args[1])))
```

I also get plug-ins for free. If I need a plug-in that deletes all files with names ending with *.draft*, all I do is write a function that gets a site tree as an argument and returns a new site tree. I then plug in that function somewhere between `read` and `write`, and I'm golden.

Another benefit of using a functional programming style is simpler unit tests. A pure function takes in some data, computes it, and returns the result. This means that all that's needed in order to test that function is input data and an expected return value. As a simple example, here's a unit test for our function `merge`:

```
function testMerge() {
  var data = [
    { // Both lists have the same size
      a: [1, 2, 3],
      b: ["a", "b", "c"],
      ret: [ [1, "a"], [2, "b"], [3, "c"] ]
    },

    { // Second list is larger
      a: [1, 2],
      b: ["a", "b", "c"],
      ret: [ [1, "a"], [2, "b"] ]
    },

    { // Etc.
      ...
    }
  ]

  data.forEach(function (test) {
    isEqual(merge(test.a, test.b), test.ret)
  })
}
```

This test is almost fully declarative. You can clearly see what input data is used and what is expected to be returned by the `merge` function. In addition, writing code in a functional way means you have less testing to do. Our original implementation of `merge` was modifying its arguments, so that a proper test would have had to cover cases where one of the arguments was frozen using `Object.freeze`.

All functions involved in the preceding example—`forEach`, `isEqual`, and `merge`—were designed to work with only simple, built-in data types. This approach, where you build your programs around composable functions that work with simple data types, is called *data-driven programming*. It allows you to write programs that are clear and elegant and have a lot of flexibility for expansion.

Objects

Does this mean you shouldn't use objects, constructors, and prototype inheritance? Of course not! If something makes your code easier to understand and maintain, it'd be silly not to use it. However, JavaScript developers often start making overcomplicated object hierarchies without even considering whether there are simpler ways to solve the problem.

Consider the following object that represents a robot. This robot can walk and talk, but otherwise it's pretty useless:

```
function Robot(name) {
  this.name = name
}

Robot.prototype = {
  talk: function (what)  { /* ... */ },
  walk: function (where) { /* ... */ }
}
```

What would you do if you wanted two more robots: a guard robot to shoot things and a housekeeping robot to clean things? Most people would immediately create child objects `GuardRobot` and `HousekeeperRobot` that inherit methods and properties from the parent `Robot` object and add their own methods. But what if you then decided you wanted a robot that can both clean and shoot things? This is where hierarchy gets complicated and software fragile.

Consider the alternative approach, where you extend instances with functions that define their behavior and not their type. You don't have a `GuardRobot` and a `HousekeeperRobot` anymore; instead, you have an instance of a `Robot` that can clean things, shoot things, or do both. The implementation will probably look something like this:

```
function extend(robot, skills) {
  skills.forEach(function (skill) {
    robot[skill.name] = skill.fn.bind(null, rb)
  })

  return robot
}
```

To use it, all you have to do is to implement the behavior you need and attach it to the instance in question:

```
function shoot(robot) { /* ... */ }
function clean(robot) { /* ... */ }
```

```
var rdo = new Robot("R. Daniel Olivaw")
extend(rdo, { shoot: shoot, clean: clean })

rdo.talk("Hi!") // OK
rdo.walk("Mozilla SF") // OK
rdo.shoot() // OK
rdo.clean() // OK
```

NOTE

My friend Irakli Gozalishvili, after reading this chapter, left a comment saying that his approach would be different. What if objects were used only to store data?

```
function talk(robot)  { /* ... */ }
function shoot(robot) { /* ... */ }
function clean(robot) { /* ... */ }

var rdo = { name: "R. Daniel Olivaw" }

talk(rdo, "Hi!") // OK
walk(rdo, "Mozilla SF") // OK
shoot(rdo) // OK
clean(rdo) // OK
```

With his approach you don't even need to extend anything: all you need to do is pass the correct object.

At the beginning of this chapter, I warned JavaScript programmers against being lulled into the false sense of referential transparency that can result from using a pure functional programming language. In the example we just looked at, the function `extend` takes an object as its first argument, modifies it, and returns the modified object. The problem here is that JavaScript has a very limited set of immutable types. Strings are immutable. So are numbers. But objects—such as an instance of `Robot`—are mutable. This means that `extend` is not a pure function, since it mutates the object that was passed into it. You can call `extend` without assigning its return value to anything, and `rdo` will still be modified.

Now What?

The major evolution that is still going on for me is towards a more functional programming style, which involves unlearning a lot of old habits, and backing away from some OOP directions.

—John Carmack

JavaScript is a multiparadigm language supporting object-oriented, imperative, and functional programming styles. It provides a framework in which you can mix and match different styles and, as a result, write elegant programs. Some programmers, however, forget about all the different paradigms and stick only with their favorite one. Sometimes this rigidity is due to fear of leaving a comfort zone; sometimes it's caused by relying too heavily on the wisdom of elders. Whatever the reason, these people often limit their options by confining themselves to a small space where it's their way or the highway.

Finding the right balance between different programming styles is hard. It requires many hours of experimentation and a healthy number of mistakes. But the struggle is worth it. Your code will be easier to reason about. It will be more flexible. You'll ultimately find yourself spending less time debugging, and more time creating something new and exciting.

So don't be afraid to experiment with different language features and paradigms. They're here for you to use, and they aren't going anywhere. Just remember: there's no single true paradigm, and it's never too late to throw out your old habits and learn something new.

CHAPTER FIFTEEN

Progress

Rick Waldron

This is a risky chapter to write—it's possible that some of the code it contains will be wrong by the time this book goes to print. In June 2015 the sixth edition of the ECMAScript standard is scheduled to be published, which means that by the time you're reading it, this chapter is either a goldmine or a dud. While being rational certainly has its merits, being adventurous and perhaps a little irrational is also valuable, so here we go.

To be clear: I really love JavaScript—as a general-purpose programming language, it is a truly unique and beautiful creation. When I say beautiful, I don't necessarily mean it in the classical sense: sometimes a beast can also be beautiful—that beast just needs someone to love it and dress it in nice clothes for the ball.

> JavaScript has an easy-to-use mechanism for inheritance.
>
> —No one

If I had a nickel for every mention of "JavaScript inheritance" on the Internet-according-to-Google, at the time of this writing I would have over $2,000.[1] The top results[2] all disagree with each other, but in interesting and not-incorrect ways. The problem is that JavaScript's built-in mechanism for defining a class of object and its behavior is the same mechanism used for defining *any* encapsulated operation: a function. Additionally, veterans of other programming languages like C++, C#, Java, Python, Ruby, and so on tend to get hung up on JavaScript's lack of inheritance as

1 As this book entered production, Google showed over 45,000 hits for the quoted term "JavaScript inheritance."

2 "Classical Inheritance in JavaScript—Douglas Crockford", "Understanding JavaScript Inheritance—Alex Sexton", "Inheritance and the prototype chain—JavaScript|MDN", and "John Resig—Simple JavaScript Inheritance".

they understand it in those languages. JavaScript's prototypal inheritance model is often the target of unfair name calling, and the truth is, while beautiful and powerful, it's a bit confusing to learn and more so to master. As an added kick in the pants, JavaScript's built-in classes weren't designed with subclassing in mind.

Thwarted at every turn!

Early in the history of jQuery, John Resig attempted to make it a subclass of the built-in `Array` class. He ultimately moved on after discovering that not only is `length` property assignment "magic" lost for `Array` subclasses, but older versions of Internet Explorer would always report a value of `0`, regardless of the length of items in the instance of the `Array` subclass. This was before the fifth edition of ECMAScript had been published, so the `Array` built-in hadn't grown the familiar API that it sports today—which also meant that in addition to `length` property issues, John had to design and implement his own API for interacting with array-like collections. These obstacles led to the current jQuery design: a class that implements collection-centric iterative operation methods and produces instances that are array-like. In this chapter, I'm going to show you what this means in terms of code, by implementing a class whose instances are an array-like list of elements with a few simple but useful methods, and then evolve the code by refactoring it several times using modern, then future, language features. The "test suite" will assert the basic functionality of the library—none of the refactorings will violate their expectations. The full code can be found at *http://bit.ly/jquery_test_suite*.

This is approximately what the output will look like:

```
// file:criteria.js
(Result) The Elements class prototype
(Result) Zero length instance
(Result) Elements from Elements
(Result) One match will have a length of 1 (no context)
(Result) One match will have a length of 1
(Result) Two matches will have a length of 2
(Result) Add a class
(Result) Set and get an attribute
(Result) Set and get a css style property
(Result) Set and get some html
(Result) Filtering produces a new instance
(Result) Filter with a dummy predicate
(Result) Filter with a predicate
(Result) Invocation forEach item in the list
(Result) Find the indexOf an element
(Result) Push an element onto the list
(Result) Push returns the instance, not the length
(Result) Slicing produces a new instance
(Result) Slice a list of elements
(Result) Sort a list of elements by nodeName
```

NOTE

`(Result)` will be either `Pass` or `Fail`.

Engineers often dismiss testing and the value it provides to their code, but I believe that in order to write truly beautiful code in any language, you must prove that code through tests. A good way to think about writing tests is as an agreement, where the agreement is that the code being written behaves a certain way and produces a certain result and will do so for all time (or until the requirements change). Tests will therefore help guide the process of refactoring the code several times over by forcing us to uphold our side of the agreement.

The following library code is the first iteration of the `Elements` class. This implementation does not attempt to subclass the built-in `Array` class.

NOTE

Every call to `context.querySelectorAll(selector)` will be wrapped in a `try-catch` to suppress exceptions thrown by invalid selectors.

```
// file:elements-r1.js
function Elements(selector, context) {
  var elems, elem, k;

  selector = selector || "";

  this.context = context || document;

  if (Array.isArray(selector) || selector instanceof Elements) {
    elems = selector;
  } else {
    try {
      elems = this.context.querySelectorAll(selector);
    } catch (e) {
      elems = [];
    }
  }

  if (!elems) {
    // elems is either:
    //   - undefined because the selector was invalid
    //      resulting in a thrown exception
    //   - null because the querySelectorAll returns
    //      null instead of an empty object when no
    //      matching elements are found.
    elems = []
  }

  if (elems.length) {
    k = -1;
```

```
      while (elem = elems[++k]) {
        this[k] = elem;
      }
    }

    this.length = elems.length;
  }

  Elements.prototype = {
    constructor: Elements,
    addClass: function(value) {
      this.forEach(function(elem) {
        elem.classList.add(value);
      });

      return this;
    },
    attr: function(key, value) {
      if (typeof value !== "undefined") {
        this.forEach(function(elem) {
          elem.setAttribute(key, value);
        });

        return this;
      } else {
        return this[0] && this[0].getAttribute(key);
      }
    },
    css: function(key, value) {
      if (typeof value !== "undefined") {
        this.forEach(function(elem) {
          elem.style[key] = value;
        });

        return this;
      } else {
        return this[0] && this[0].style[key];
      }
    },
    html: function(html) {
      if (typeof html !== "undefined") {
        this.forEach(function(elem) {
          elem.innerHTML = html;
        });
        return this;
      } else {
        return this[0] && this[0].innerHTML;
      }
    },
    filter: function() {
      return new Elements([].filter.apply(this, arguments));
    },
    forEach: function() {
      [].forEach.apply(this, arguments);
      return this;
    },
    indexOf: function() {
      return [].indexOf.apply(this, arguments);
    },
    push: function() {
      [].push.apply(this, arguments);
      return this;
    },
    slice: function() {
      return new Elements([].slice.apply(this, arguments));
    },
    sort: function() {
      return [].sort.apply(this, arguments);
    }
  };
```

While certainly correct, this code is a technical debt nightmare. The six array allocations could be replaced with a shared reference to `Array.prototype`, but this would require wrapping the entire declaration and prototype definition inside an immediately invoked function expression to avoid leaking that binding into the global object. It's also safe to assume that the explicit reboxing of `Array` instances into `Elements` instances will have performance penalties. Despite these drawbacks, the first implementation is functional and has revealed that `length` property assignment semantics, as they are defined for a built-in `Array` instance, are not a requirement for this object. With that understanding, the example can be naively refactored as a rudimentary `Array` subclass:

```
  // file:elements-r2.js
  function Elements(selector, context) {
    Array.call(this);

    var elems;

    this.context = context || document;

    if (Array.isArray(selector) || selector instanceof Elements) {
      elems = selector;
    } else {
      try {
        elems = this.context.querySelectorAll(selector || "");
      } catch (e) {
        elems = [];
      }
    }

    if (!elems) {
      // elems is either:
      //   - undefined because the selector was invalid
      //      resulting in a thrown exception
    //   - null because the querySelectorAll returns
    //      null instead of an empty object when no
    //      matching elements are found.
    elems = []
  }

  this.push.apply(this, elems);
}

Elements.prototype = Object.create(Array.prototype);
Elements.prototype.constructor = Elements;

Elements.prototype.addClass = function(value) {
  this.forEach(function(elem) {
    elem.classList.add(value);
  });

  return this;
};
Elements.prototype.attr = function(key, value) {
  if (typeof value !== "undefined") {
    this.forEach(function(elem) {
      elem.setAttribute(key, value);
    });

    return this;
  } else {
    return this[0] && this[0].getAttribute(key);
  }
};
Elements.prototype.css = function(key, value) {
  if (typeof value !== "undefined") {
    this.forEach(function(elem) {
      elem.style[key] = value;
    });

    return this;
  } else {
    return this[0] && this[0].style[key];
  }
};
Elements.prototype.html = function(html) {
  if (typeof html !== "undefined") {
    this.forEach(function(elem) {
      elem.innerHTML = html;
    });
    return this;
  } else {
    return this[0] && this[0].innerHTML;
  }
};
Elements.prototype.filter = function() {
  return new Elements([].filter.apply(this, arguments));
```

```
};
Elements.prototype.slice = function() {
  return new Elements([].slice.apply(this, arguments));
};
Elements.prototype.push = function() {
  [].push.apply(this, arguments);
  return this;
};
```

A number of changes have occurred, and the design no longer specifies an explicit definition of `forEach`, `indexOf`, and `sort`: these are now inherited directly from the built-in `Array.prototype`. This refactored version is still functionally correct and will pass all of the assertions defined for the `Elements` class. Unfortunately, refactoring the design so that it is an `Array` subclass has cost the library any sense of cohesion: assigning `Elements.prototype` the value of `Object.create(Array.prototype)` means that the design must trade readability of `Elements.prototype = {...};` for the forced one-property-assignment-at-a-time form for all of the subclass's own prototype methods. This further exposes a pathological problem: the source doesn't visually express the intent, which is to define a *class* of thing and its behavior.

A reasonably large portion of readers may cringe at the word *class*, but it's important to remember that *class* is not about any specific language's implementation of object-oriented programming paradigms. Remember where the terms "object" and "class" came from, with regard to computer programming languages:

> A central new concept in Simula 67 is the "object". An object is a self-contained program (block instance), having its own local data and actions defined by a "class declaration". The class declaration defines a program (data and action) pattern, and objects conforming to that pattern are said to "belong to the same class."
>
> —Ole-Johanv Dahl, Bjorn Myhrhaug, and Kristen Nygaard, "SIMULA 67 Common Base Language"

With this definition in mind, the piece of code I'm designing is undoubtedly a "class." That it is a function declaration with prototype definition is nothing more than an implementation detail. Once this is accepted, the library may embrace the changes it will undergo in the final refactoring. Before we can move to the third and final revision, it's valuable to revisit patterns that have emerged and gained the most traction. The constructor function and prototype definition pattern has, unfortunately, proven itself to be less than intuitive, and for almost a decade JavaScript programmers have been pursuing a mechanism that will allow them to write program code that includes some form of class-like semantics. This has led to the proliferation of API-bound library code that attempts to paper over the lack of syntactic forms, where simplicity is in the eye of the author.[3] In more recent years, languages that transpile to JavaScript have been created to allow for even simpler syntactic forms to fill in the role of the missing mechanism.[4] What follows is a collection of various class-like APIs and syntactic class declaration forms, all of which define a `List` class and a `People` class: the latter is a subclass of the former whose `push` method will reject nonstring entries. These criteria represent a common, not-quite-trivial programming task that will serve to illustrate, by example, the approaches taken and the patterns that have emerged.

So, for the following specified criteria:

- Define a `List` class.
- Define a `People` subclass of `List`.
- Define a `push` method that will reject nonstring entries.

these are the assertions that we'll write tests for:

```
(Result) An object named List exists, its type is "function"
(Result) An object named People exists, its type is "function"
(Result) Initialization arguments length equals List object length
(Result) Initialization arguments length equals People object length for
  strings only
(Result) Any value may be pushed into a List object.
(Result) String values may be pushed into People object.
```

A simplified version of these assertions might look like this:

```
var l = new List(1, "foo", []);
console.log(l.length === 3);
console.log(l.push(42) === 4);

var p = new People("Alice", "Bob", "Carol");
console.log(p.length === 3);
console.log(p.push(42) === 3);
console.log(p.push("Dennis") === 4);
```

This should be run after each of the following examples (in some cases, after they've been transpiled).

3 See, for example, *http://api.prototypejs.org/language/Class/*, *http://ejohn.org/blog/simple-javascript-inheritance/*, *http://dean.edwards.name/weblog/2006/03/base/*, and *http://dojotoolkit.org/documentation/tutorials/1.7/declare/*.

4 See *http://coffeescript.org/#classes*.

JavaScript:

```
function List() {
  this.push.apply(this, arguments);
}

List.prototype.push = function() {
  return [].push.apply(this, arguments);
};

function People() {
  List.call(this);
  this.push.apply(this, arguments);
}

People.prototype.push = function() {
  return List.prototype.push.apply(
    this, [].filter.call(arguments, function(peep) {
      return typeof peep === "string";
    })
  );
};
```

Prototype.js:

```
var List = Class.create({
  initialize: function() {
    this.push.apply(this, arguments);
  },
  push: function() {
    return [].push.apply(this, arguments);
  }
});

var People = Class.create(List, {
  push: function($super) {
    return $super.apply(
      this, [].slice.call(arguments, 1).filter(function(peep) {
        return typeof peep === "string";
      })
    );
  }
});
```

Simple JavaScript Inheritance:

```
var List = Class.extend({
  init: function() {
    this.push.apply(this, arguments);
  },
  push: function() {
    return [].push.apply(this, arguments);
  }
});
```

```
var People = List.extend({
  init: function() {
    this._super.apply(this, arguments);
  },
  push: function() {
    return this._super.apply(
      this, [].filter.call(arguments, function(peep) {
        return typeof peep === "string";
      })
    );
  }
});
```

Dojo:

```
// Thanks to Brian Arnold @brianarn for this one.
// Note that this approach is deprecated and is shown
// here only as a means to illustrate a point.
var List = dojo.declare(null, {
  constructor: function() {
    this.push.apply(this, arguments);
  },
  push: function() {
    return [].push.apply(this, arguments);
  }
});

var People = dojo.declare(List, {
  push: function() {
    return this.inherited(
      arguments, [].filter.call(arguments, function(peep) {
        return typeof peep === "string";
      })
    );
  }
});
```

Ext.js:

```
Ext.define("List", {
  constructor: function () {
    this.push.apply(this, arguments);
  },
  push: function() {
    return [].push.apply(this, arguments);
  }
});

Ext.define("People", {
  extend: "List",
  push: function() {
    return this.callParent(
      [].filter.call(arguments, function(peep) {
        return typeof peep === "string";
      })
    );
  }
});
```

CoffeeScript:

```
class List
  constructor: (args...) ->
    this.push args...

  push: (args...) ->
    [].push args...


class People extends List
  push: (args...) ->
    super (args.filter (peep) -> typeof peep is "string")...
```

…which transpiles to…

```
var List, People, _ref,
  __slice = [].slice,
  __hasProp = {}.hasOwnProperty,
  __extends = function(child, parent) {
    for (var key in parent) {
      if (__hasProp.call(parent, key)) child[key] = parent[key];
    }
    function ctor() {
      this.constructor = child;
    }
    ctor.prototype = parent.prototype;
    child.prototype = new ctor();
    child.__super__ = parent.prototype;
    return child;
  };

List = (function() {
  function List() {
    var args;
    args = 1 <= arguments.length ? __slice.call(arguments, 0) : [];
    this.push.apply(this, args);
  }

  List.prototype.push = function() {
    var args, _ref;
    args = 1 <= arguments.length ? __slice.call(arguments, 0) : [];
    return (_ref = []).push.apply(_ref, args);
  };

  return List;

})();
```

```
People = (function(_super) {
  __extends(People, _super);

  function People() {
    _ref = People.__super__.constructor.apply(this, arguments);
    return _ref;
  }

  People.prototype.push = function() {
    var args;
    args = 1 <= arguments.length ? __slice.call(arguments, 0) : [];
    return People.__super__.push.apply(this, args.filter(function(peep) {
      return typeof peep === "string";
    }));
  };

  return People;

})(List);
```

TypeScript:

```
class List {
  constructor(...args) {
    this.push.apply(this, args);
  }
  push(...args) {
    return [].push.apply(this, args);
  }
}

class People extends List {
  push(...args) {
    return super.push.apply(
      this, args.filter(peep => typeof peep === "string")
    );
  }
}
```

…which transpiles to…

```
var __extends = this.__extends || function (d, b) {
  for (var p in b) if (b.hasOwnProperty(p)) d[p] = b[p];
  function __() { this.constructor = d; }
  __.prototype = b.prototype;
  d.prototype = new __();
};
var List = (function () {
  function List() {
    var args = [];
    for (var _i = 0; _i < (arguments.length - 0); _i++) {
      args[_i] = arguments[_i + 0];
    }
```

```
      this.push.apply(this, args);
    }
    List.prototype.push = function () {
      var args = [];
      for (var _i = 0; _i < (arguments.length - 0); _i++) {
        args[_i] = arguments[_i + 0];
      }
      return [].push.apply(this, args);
    };
    return List;
  })();

  var People = (function (_super) {
    __extends(People, _super);
    function People() {
      _super.apply(this, arguments);
    }
    People.prototype.push = function () {
      var args = [];
      for (var _i = 0; _i < (arguments.length - 0); _i++) {
        args[_i] = arguments[_i + 0];
      }
      return _super.prototype.push.apply(
        this, args.filter(function (peep) {
          return typeof peep === "string";
        })
      );
    };
    return People;
  })(List);
```

Nearly all of these examples have one particular thing in common: a `class` definition or some other declaration mechanism that actually uses the word "class." The compile-to-JavaScript examples provide a syntactic form, while the already-JavaScript examples offer API-based implementations. In Dojo's case, the authors have chosen to name the API `declare`, but the documentation for `dojo.declare`, and the AMD replacement that supersedes it, describes defining a "Class." The only outlier here is the example written JavaScript itself—but don't be fooled, the language has held the word `class` as a *FutureReservedWord* since ECMAScript 1.0 and has used `[[Class]]` as an internal specification mechanism for just as long.

The "super" mechanisms are completely different in each of the API-based examples, but in all cases the user is expected to know that `$super`, `_super`, `inherited`, and `callParent` are designed to allow calling code to reach the superclass's method of the same name as the method from which the call originates. Arguably, if these were all consistently named, this semantic relationship would be more intuitive (as it is in other languages with similar mechanisms).

I applaud the authors of those projects for their impactful creativity and ingenuity; however, the repetition and boilerplate shown in the API examples, compounded by the dramatic output of the transpiled examples, should lead a critical mind to the conclusion that a `class` mechanism is desired, and that for it to be powerful enough to meet the most common needs it must exist at the language level.

The following example is what the JavaScript example will become, in the very near future:

```
class List extends Array {
  constructor(...args) {
    this.push(...args);
  }
}

class People extends List {
  push(...args) {
    return super.push(
      ...args.filter(peep => typeof peep === "string")
    );
  }
}
```

Subjectively, that's much nicer to look at than all of the API-based examples and visually on par with compile-to-JavaScript examples. Aesthetics aside, this program is technically superior to all of the preceding examples. The most obvious change is that `function` is no longer used to declare a class, having been replaced with a new declarative form, appropriately named `class`. Instead of four statement boundaries, as in the present-day JavaScript example (`List`, `List.prototype`, `People`, `People.prototype`), there are now two encapsulated class definitions: `List` and `People`. `List` no longer provides an explicit declaration of its `push` method because it's now inheriting the method (and correct `length` semantics) from `Array`—which can now be safely subclassed. The `extends` clause is obviously not limited to built-ins, as we see that the `People` class is itself a subclass of `List`, which is a subclass of `Array`. Inside of the `People` class, there is now a call to a qualified `super.push`, which is a call to the `push` method of this class's `super` class (in this case, the call goes up the prototype two steps to `Array.prototype.push`). A lot of the ceremonial boilerplate, in most cases irrelevant to what the program is expressing, has been removed. This is most evident in the lack of any occurrences of the word "function," having been replaced by the semantically meaningful "class" and removed in favor of method shorthand notation. Clumsy `arguments` objects and verbose parameter handling have been completely replaced by elegantly simple rest parameters and spread arguments.

With new syntactic forms and language-level mechanisms, we can revisit the `Elements` class from earlier in the chapter and apply the same changes to that code—while abiding by the regression tests—for truly dramatic improvements to the code:

```
// file:elements-r3.js
class Elements extends Array {
  constructor(selector = "", context = document) {
    super();

    let elems;

    this.context = context;

    if (Array.isArray(selector) ||
        selector instanceof Elements) {
      elems = selector;
    } else {
      try {
        elems = this.context.querySelectorAll(selector);
      } catch (e) {
        // Thrown Exceptions caused by invalid selectors
        // are a nuisance.
      }
    }

    if (!elems) {
      // elems is either:
      //   - undefined because the selector was invalid
      //      resulting in a thrown exception
      //   - null because the querySelectorAll returns
      //      null instead of an empty object when no
      //      matching elements are found.
      elems = []
    }

    this.push(...elems);
  }
  addClass(value) {
    return this.forEach(elem => elem.classList.add(value));
  }
  attr(key, value) {
    if (typeof value !== "undefined") {
      return this.forEach(elem => elem.setAttribute(key, value));
    } else {
      return this[0] && this[0].getAttribute(key);
    }
  }
  css(key, value) {
    if (typeof value !== "undefined") {
      return this.forEach(elem => elem.style[key] = value);
    } else {
      return this[0] && this[0].style[key];
    }
  }
  html(html) {
    if (typeof html !== "undefined") {
      return this.forEach(elem => elem.innerHTML = html);
```

```
        } else {
          return this[0] && this[0].innerHTML;
        }
      }
      filter(callback) {
        return new Elements(super.filter(callback, this));
      }
      slice(...args) {
        return new Elements(super.slice(...args));
      }
      forEach(callback) {
        super.forEach(callback, this);
        return this;
      }
      push(...elems) {
        super.push(...elems);
        return this;
      }
    }
```

A lot has changed, but it's very important to remember that despite these changes, this code produces the same class as the previous version (it's not identical, but for our purposes it meets the requirements) version and will pass the test suite written for the implementation prior to this refactoring.

To more accurately express the intention of this code, the function declaration and explicit `prototype` definition have been replaced by a single class declaration. Where the previous examples required two or three different syntactic forms (function declaration statements, assignment expressions coupled to make expression statements, etc.), the refactored form provides a distinct boundary (the class body) that encapsulates the constructor and all of the class's `prototype` object method definitions—which use the elegantly succinct concise method syntax. Of course, by using the class form, the declaration can now take advantage of true subclassing via the `extends` clause. Potential "falsy-positive" footguns created by logical OR operations to determine the default values of the `selector` and `context` parameters have been mitigated, and these are now more clearly expressed in the form of default parameter assignments. The entirely unobvious use of the `Array` constructor as a pseudo-`super` call mechanism has been replaced by the unmistakably obvious `super` call in the constructor. All of the anonymous function expressions have been replaced by arrow functions, eliminating the clutter incurred by `function() { return ...; }`.

The most important aspect to consider is the overall removal of "distraction." This revision from the future takes the focus away from the esoteric inheritance, method borrowing incantations, and repetitious boilerplate, to bring the semantics of the program itself into view—and does so in a completely compatible way, as evidenced by the passing of our regression tests.

As we've seen throughout this chapter, JavaScript is a powerful language that has always been flexible and expressive enough to empower its users to define the evolution of the language well ahead of its time. Through this real-world inspiration, the language itself has been able to progress in ways that directly correspond to the works of its practitioners.

Index

About the Authors

Anton Kovalyov (@valueof) was born and raised in Tashkent, Uzbekistan. Back in the day, he was mostly writing Python and (re-)compiling Gentoo. In 2008, he moved to the United States where he joined Disqus. Around the same time, he discovered JavaScript and the two have been inseparable ever since. While at Disqus, Anton authored JSHint, a JavaScript linting tool, and coauthored *Third-Party JavaScript* (Manning). After Disqus, Anton moved to Mozilla, where he worked on the Firefox Developer Tools team. Today, Anton works at Medium and lives in Oakland, California.

Jonathan Barronville (@jonathanmarvens) is a 21-year-old Haitian hacker. He enjoys learning new things and then force-teaching them to you. Although most of his experience is in web development, Jonathan enjoys low-level systems hacking, database theory, and distributed systems problems.

Sara Chipps (@sarajchipps) is a JavaScript developer based in New York City. She has been working on software and in the open source community since 2001. She's been obsessed with hardware and a fan of Nodebots.com since 2012.

Sara is the CEO of Jewelbots.com, a company dedicated to drastically changing the number of girls entering STEM (science, technology, engineering, and mathematics) fields using hardware.

In 2010, Sara cofounded Girl Develop It, a nonprofit focused on helping more women become software developers. Girl Develop It is in 45 cities, and has taught over 17,000 women how to build software.

Angus Croll (@angustweets) is obsessed with JavaScript and literature in equal measure. He works as a frontend engineer at Twitter and is the author of *If Hemingway Wrote JavaScript* (No Starch Press).

Marijn Haverbeke (@marijnjh) is the author of *Eloquent JavaScript* (No Starch Press) and creator of CodeMirror and Tern. He's an independent open source critter and JavaScript code cowboy.

Ariya Hidayat (@ariyahidayat), currently working for Shape Security, is a passionate engineer interested in bleeding-edge technologies. He is known as the author of PhantomJS and Esprima. These days, his focus is mostly on software craftsmanship around web technologies.

Daryl Koopersmith (@koop) is an engineer at Medium, where he leads the web client guild. Previously at Automattic, he was a core committer to the WordPress open source project. Some days, he pretends to be a barista.

Rebecca Murphey (@rmurphey) is a staff software engineer at Bazaarvoice. She has played a key role in the software design and development of high-traffic client-side web applications, and is known for her expertise in best practices for organizing, testing, refactoring, and maintaining JavaScript application code. Rebecca developed the JS Assessment project, an open source tool used by individuals, companies, and code schools to evaluate a developer's JavaScript skills. She was instrumental in getting promises introduced to jQuery 1.5 and has contributed to several open source projects. She authored the online book *jQuery Fundamentals*, contributed to the *jQuery Cookbook* (O'Reilly), and served as a technical reviewer for Garann Means's *Node for Front-End Developers* (O'Reilly) and David Herman's *Effective JavaScript* (Addison-Wesley Professional). She lives in Austin with her partner and their son.

Daniel Pupius (@dpup) is the head of engineering at Medium. Previously at Google, he worked on Google+ and Gmail, and cofounded the Closure library.

He cut his teeth fighting version 4 browsers and was involved in the early DHTML community, before AJAX was a thing. In other lives Dan has raced snowboards, jumped out of planes, and lived in the jungle.

Graeme Roberts (@cheedear), "chee"; recently made redundant by lonely planet; cyrano de bergerac of svg icons; five foot ten and a half; drinks tom collins; essentially useless; made of fire and tears.

Jenn Schiffer (@jennschiffer) is an engineer and artist who focuses on open web technology, open source development, and getting yelled at on Twitter. She is the creator of make8bitart.com, among other code/art projects, and writes tech satire on a number of media. You can find everything she has ruined ever online at *http://jennmoney.biz*.

Jacob Thornton (@fat), creator of bootstrap, bower, ratchet, and a handful of other open source technologies. Twitter, Medium, Obvious, chill tech enterprises. 6′ 3″. Aries. Grows parsley. Emotional. Worst engineer at the company, but third coolest.

Ben Vinegar (@bentlegen) is a software engineer based in San Francisco, and the coauthor of *Third-Party JavaScript* (Manning). He was formerly lead frontend engineer at Disqus.

Rick Waldron (@rwaldron) is an open web engineer at Bocoup, Ecma/TC39 representative for the jQuery Foundation, and creator of Johnny-Five, a JavaScript robotics programming framework.

Nicholas Zakas (@slicknet) is a frontend engineer, author, and speaker. He currently works at Box making the web application awesome. Prior to that, he worked at Yahoo! for almost five years, where he was frontend tech lead for the Yahoo! home page and a contributor to the YUI library. He is the author of *Maintainable JavaScript* (O'Reilly), *Professional JavaScript for Web Developers* (Wrox), *High Performance JavaScript* (O'Reilly), and *Professional Ajax* (Wrox). Nicholas is a strong advocate for development best practices including progressive enhancement, accessibility, performance, scalability, and maintainability. He blogs regularly at NCZOnline.

Colophon

The cover fonts are URW Typewriter and Guardian Sans. The text font is Adobe Meridien; the heading fonts are Akzidenz-Grotesk and Adobe Minion Pro; and the code font is Dalton Maag's Ubuntu Mono.