The Pragmatic Programmers

CoffeeScript

Accelerated JavaScript Development

Trevor Burnham

Foreword by Jeremy Ashkenas

edited by Michael Swaine

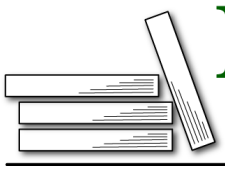
CoffeeScript

Accelerated JavaScript Development

Trevor Burnham

The Pragmatic Bookshelf

Dallas, Texas • Raleigh, North Carolina



Pragmatic Bookshelf

Our Pragmatic courses, workshops, and other products can help you and your team create better software and have more fun. For more information, as well as the latest Pragmatic titles, please visit us at *http://pragprog.com*.

The team that produced this book includes:

Michael Swaine (editor)
Potomac Indexing, LLC (indexer)
Kim Wimpsett (copyeditor)
David Kelly (typesetter)
Janet Furlow (producer)
Juliet Benda (rights)
Ellie Callahan (support)

Printed in the United States of America.
ISBN-13: 978-1-934356-78-4
Printed on acid-free paper.
Book version: P1.0—July 2011

JavaScript is born free, but until recently, everywhere it was in chains.

JavaScript had never been a very pleasant language to work in: terribly slow, implemented with different quirks in different browsers, stuck fast in the amber of time since the late 1990s. Perhaps you used it in the past to implement a dropdown menu or a reorderable list, but you probably didn't enjoy the experience.

Fortunately for us, the JavaScript of today is enjoying a well-deserved renaissance. Thanks to the tireless efforts of browser implementers, it's now the fastest mainstream dynamic language; it's present everywhere, from servers to Photoshop, and it's the only possible language you can use to program all angles of the web.

CoffeeScript is a little language that aims to give you easy access to the good parts of JavaScript: the first-class functions, the hash-like objects, even the much-misunderstood prototype chain. If we do our job right, you'll end up writing one-third less code in order to generate much the same JavaScript you would have written in the first place.

CoffeeScript places a high value on the readability of code and the elimination of syntactic clutter. At the same time, there's a fairly one-to-one correspondence between CoffeeScript and JavaScript, which means that there should be no performance penalty—in fact, many JavaScript libraries end up running faster after being ported to CoffeeScript due to some of the optimizations the compiler can perform.

You're fortunate to have picked up this book, because Trevor has been an enthusiastic contributor to CoffeeScript since the early days. Few people know more about the ins and outs of the language or the history of the debate behind language features and omissions than he does. This book is a gentle introduction to CoffeeScript led by an expert guide.

I'm looking forward to hearing about all of the exciting projects that I'm sure will come out of it, and—who knows—perhaps you'll be inspired to create a little language of your very own.

Jeremy Ashkenas, creator of CoffeeScript
April 2011

JavaScript was never meant to be the most important programming language in the world. It was hacked together in ten days, with ideas from Scheme and Self packed into a C-like syntax. Even its name was an awkward fit, referring to a language with little in common besides a few keywords.[1] But once JavaScript was released, there was no controlling it. As the only language understood by all major browsers, JavaScript quickly became the lingua franca of the Web. And with the introduction of Ajax in the early 2000s, what began as a humble scripting language for enhancing web pages suddenly became a full-fledged rich application development language.

As JavaScript's star rose, discontent came from all corners. Some pointed to its numerous little quirks and inconsistencies.[2] Others complained about its lack of classes and inheritance. And a new generation of coders, who had cut their teeth on Ruby and Python, were stymied by its thickets of curly braces, parentheses, and semicolons.

A brave few created frameworks for web application development that generated JavaScript code from other languages, notably Google's GWT and 280 North's Objective-J. But few programmers wanted to add a thick layer of abstraction between themselves and the browser. No, they would press on, dealing with JavaScript's flaws by limiting themselves to "the good parts" (as in Douglas Crockford's 2008 similarly titled book).

That is, until now.

The New Kid in Town

On Christmas Day 2009, Jeremy Ashkenas first released CoffeeScript, a little language he touted as "JavaScript's less ostentatious kid brother." The project quickly attracted hundreds of followers on GitHub as Ashkenas and other contributors added a bevy of new features each month. The language's compiler, originally written in Ruby, was replaced in March 2010 by one written in CoffeeScript.

After its 1.0 release on Christmas 2010, CoffeeScript became one of Github's "most-watched" projects. And the language attracted another flurry of attention in April 2011, when David Heinemeier Hansson confirmed rumors that CoffeeScript support would be included in Ruby on Rails 3.1.

1. See Peter Seibel's interview with Brendan Eich, the creator of JavaScript, in *Coders at Work* [Sei09].
2. http://wtfjs.com/

Why did this little language catch on so quickly? Three reasons come to mind: familiarity, safety, and readability.

The Good Parts Are Still There

JavaScript is vast. It contains multitudes. JavaScript offers many of the best features of functional languages while retaining the feel of an imperative language. This subtle power is one of the reasons that JavaScript tends to confound newcomers: functions can be passed around as arguments and returned from other functions; objects can have new methods added at any time; in short, *functions are first-class objects*.

All that power is still there in CoffeeScript, along with a syntax that encourages you to use it wisely.

The Compiler Is Here to Help

Imagine a language with no syntax errors, a language where the computer forgives you your typos and tries as best it can to comprehend the code you give it. What a wonderful world that would be! Sure, the program wouldn't always run the way you expected, but that's what testing is for.

Now imagine that you write that code once and send it out to the world, typos and all, and millions of computers work around your small mistakes in subtly different ways. Suddenly statements that your computer silently skipped over are crashing your entire app for thousands of users.

Sadly, that's the world we live in. JavaScript doesn't have a standard interpreter. Instead, hundreds of browsers and server-side frameworks run JavaScript in their own way. Debugging cross-platform inconsistencies is a huge pain.

CoffeeScript can't cure all of these ills, but the compiler tries its best to generate JavaScript Lint-compliant output[3], which is a great filter for common human errors and nonstandard idioms. And if you type something that just doesn't make any sense, such as `2 = 3`, the CoffeeScript compiler will tell you. Better to find out sooner than later.

It's All So Clear Now

Writing CoffeeScript can be highly addictive. Why? Take this piece of JavaScript:

```
function cube(num) {
```

3. http://www.javascriptlint.com/

```
  return Math.pow(num, 3);
}
var list = [1, 2, 3, 4, 5];
var cubedList = [];
for (var i = 0; i < list.length; i++) {
  cubedList.push(cube(list[i]));
}
```

Now here's an equivalent snippet of CoffeeScript:

```
cube = (num) -> Math.pow num, 3
list = [1, 2, 3, 4, 5]
cubedList = (cube num for num in list)
```

For those of you keeping score, that's half the character count and less than half the line count! Those kinds of gains are common in CoffeeScript. And as Paul Graham once put it, "Succinctness is power."[4]

Shorter code is easier to read, easier to write, and, perhaps most critically, easier to change. Gigantic heaps of code tend to lumber along, as any significant modifications require a Herculean effort. But bite-sized pieces of code can be revamped in a few swift keystrokes, encouraging a more agile, iterative development style.

It's worth adding that switching to CoffeeScript isn't an all-or-nothing proposition—CoffeeScript code and JavaScript code can interact freely. CoffeeScript's strings are just JavaScript strings, and its numbers are just JavaScript numbers; even its classes work in JavaScript frameworks like Backbone.js.[5] So don't be afraid of calling JavaScript code from CoffeeScript code or vice versa. As an example, we'll talk about using CoffeeScript with one of JavaScript's most popular libraries in Chapter 5, *Web Interactivity with jQuery*, on page ?.

But enough ancient history. Coding is believing, everything else is just meta, and as Jeff Atwood once said,"Meta is murder."[6] So let's talk a little bit about the book you're reading now, and then—in just a few pages, I promise!—we'll start banging out some hot code.

Who This Book Is For

If you're interested in learning CoffeeScript, you've come to the right place! However, because CoffeeScript is so closely linked to JavaScript, there are really two languages running through this book—and not enough pages to

4. http://www.paulgraham.com/power.html
5. http://documentcloud.github.com/backbone/
6. http://www.codinghorror.com/blog/2009/07/meta-is-murder.html

Embedding JavaScript in CoffeeScript

This is as good a place as any to mention that you can stick JavaScript inside of CoffeeScript code by surrounding it with backticks, like so:

```
console.log `impatient ? useBackticks() : learnCoffeeScript()`
```

The CoffeeScript compiler simply ignores everything between the backticks. That means that if, for instance, you declare a variable between the backticks, that variable won't obey conventional CoffeeScript scope rules.

In all my time writing CoffeeScript, I've never once needed to use backtick escapes. They're an eyesore at best and dangerous at worst. So in the immortal words of Troy McClure: "Now that you know how it's done—don't do it."

teach you both. Therefore, I'm going to assume that you know *some* JavaScript.

You don't have to be John "JavaScript Ninja" Resig. In fact, if you're only an amateur JavaScripter, great! You'll learn a lot about JavaScript as you go through this book. Check the footnotes for links to additional resources that I recommend. If you're new to programming entirely, you should definitely check out *Eloquent JavaScript* [Hav11], which is also available in an interactive online format.[7] If you've dabbled a bit but want to become an expert, head to the JavaScript Garden.[8] And if you want a comprehensive reference, no one does it better than the Mozilla Developer Network.[9]

You may notice that I talk about Ruby a lot in this book. Ruby inspired many of CoffeeScript's great features, like implicit returns, splats, and postfix `if`/`unless`. And thanks to Rails 3.1, CoffeeScript has a huge following in the Ruby world. So if you're a Rubyist, *great!* You've got a head start. If not, don't sweat it; everything will fall into place once you have a few examples under your belt.

If anything in the book doesn't make sense to you, I encourage you to post a question about it on the book's forum.[10] While I try to be clear, the only entities to whom programming languages are completely straightforward are computers—and they buy very few books.

7. http://eloquentjavascript.net/
8. http://javascriptgarden.info/
9. https://developer.mozilla.org/en/JavaScript/Guide
10. http://forums.pragprog.com/forums/169

How This Book Is Organized

We'll start our journey by discovering the various ways that we can compile and run CoffeeScript code. Then we'll delve into the nuts and bolts of the language. Each chapter will introduce concepts and conventions that tie into our ongoing project (see the next section).

To master CoffeeScript, you'll need to know how it works with the rest of the JavaScript universe. So after learning the basics of the language, we'll take brief tours of jQuery, the world's most popular JavaScript framework, and Node.js, an exciting new project that lets you run JavaScript outside of the browser. While we won't go into great depth with either tool, we'll see that they go with CoffeeScript like chocolate and peanut butter. And by combining their powers, we'll be able to write an entire multiplayer game in just a few hours.

No matter what level you're at, be sure to do the exercises at the end of each chapter. They're designed to be quick yet challenging, illustrating some of the most common pitfalls CoffeeScripters fall into. Try to solve them on your own before you check the answers in Appendix 1, *Answers to Exercises*, on page ?.

The code presented in this book, as well as errata and discussion forums, can be found on its PragProg page: http://pragprog.com/titles/tbcoffee/coffeescript.

About the Example Project: 5x5

The last section of each chapter applies the new concepts to an original word game called 5x5. As its name suggests, 5x5 is played on a grid five tiles wide and five tiles high. Each tile has a random letter placed on it at the start. Then the players take turns swapping letters on the grid, scoring points for all words formed as a result of the swap (potentially, this can be four words at each of the two swapped tiles: one running horizontally, one vertically, and two diagonally—only left-to-right diagonals count).

Scoring is based on the Scrabble point value of the letters in the formed words, with a multiplier for the number of distinct words formed. So, at the upper limit, if eight words are formed in one move, then the point value of each is multiplied by eight. Words that have already been used in the game don't count.

We'll build a command-line version of the game in Chapters 2–4, then move it to the browser in Chapter 5, *Web Interactivity with jQuery*, on page ?,

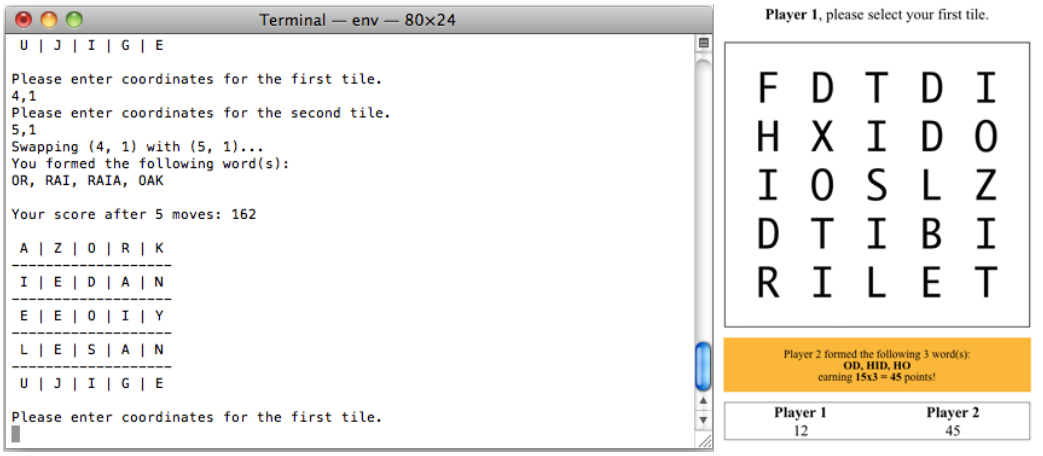


Figure 1—In the console and web versions of our project, the game logic code will be the same.

and finally add multiplayer capability in Chapter 6, *Server-Side Apps with Node.js*, on page ?. Moving the code from the command line to the browser to the server will be super-easy—they all speak the same language!

The CoffeeScript Community

A great language is of little use without a strong community. If you run into problems, who you gonna call?

Posting a question to StackOverflow (being sure to tag your question coffee-script) is a terrific way to get help, especially if you post a snippet of the code that's hassling you.[11] If you need a more immediate answer, you can usually find friendly folks in the #coffeescript channel on Freenode IRC. For relaxed discussion of CoffeeScript miscellany, try the Google Group.[12] For more serious problems, such as possible bugs, you should create an issue on GitHub.[13] You can also request new language features there. CoffeeScript is still evolving, and the whole team welcomes feedback.

What about documentation? You've probably already seen the snazzy official docs at http://coffeescript.org. There's also an official wiki at http://github.com/jashkenas/coffee-script/wiki. And now there's this book.

11. http://stackoverflow.com
12. http://groups.google.com/forum/#!forum/coffeescript
13. http://github.com/jashkenas/coffee-script/issues

Which brings us to me. I run @CoffeeScript on Twitter; you can reach me there, or by good old-fashioned email at trevorburnham@gmail.com.

These are exciting times for web development. Welcome aboard!

In the last chapter, we mastered functions. Now it's time to start applying those functions to collections of data.

We'll start by looking at objects in a new light as all-purpose storage. Then we'll learn about arrays, which give us a more ordered place to save our bits. From there, we'll segue into loops, the lingua franca of iteration. We'll also learn about building arrays directly from loops using comprehensions and about extracting parts of arrays using pattern matching. Finally, we'll complete the command-line version of 5x5 that we started in the last chapter and recap what we've learned with a fresh batch of exercises.

3.1 Objects as Hashes

Let's start by reviewing what we know about objects in JavaScript and then check out the syntactic additions that CoffeeScript provides.

Objects 101: A JavaScript Refresher

Every programming language worth its bits has some data structure that lets you store arbitrary named values. Whether you call them hashes, maps, dictionaries, or associative arrays, the core functionality is the same: you provide a key and a value, and then you use the key to fetch the value.

In JavaScript, every object is a hash. And just about everything is an object; the only exceptions are the *primitives* (booleans, numbers, and strings), and a few special constants like `undefined` and `NaN`.

The simplest object can be written like this:

```
obj = new Object()
```

Or (more commonly) you can use JSON-style syntax:

```
obj = {}
```

In JSON, objects are denoted by `{}`, arrays by `[]`. Note that JSON is a subset of JavaScript and can usually be pasted directly into CoffeeScript code. (The exception is when the JSON contains indentation that might be misinterpreted by the CoffeeScript compiler.)

But there are plenty of other ways of creating objects. In fact, we created a ton of them in the last chapter, because all functions are objects.

There are two ways of accessing object properties: dot notation and bracket notation. Dot notation is simple: `obj.x` refers to the property of `obj` named `x`. Bracket notation is more versatile: any expression placed in the brackets is evaluated and converted to a string, and then that string is used as the

property name. So `obj['x']` is always equivalent to `obj.x`, while `obj[x]` refers to the property whose name matches the (stringified) value of `x`.

Usually you want to use dot notation if you know a property's name in advance and bracket notation if you need to determine it dynamically. However, since property names can be arbitrary strings, you might sometimes need to use bracket notation with a literal key:

```
symbols.+ = 'plus'     # illegal syntax
symbols['+'] = 'plus'  # perfectly valid
```

We can create objects with several properties at once using JSON-style constructs, which separate keys from values using `:` like so:

```
father = {
  name: 'John',
  daughter: {
    name: 'Jill'
  },
  son: {
    name: 'Jack'
  }
}
```

Note that while curly braces have many uses in JavaScript, their *only* purpose in CoffeeScript is to declare objects.

Quotes are optional around the keys as long as they obey standard variable naming rules; otherwise, single- or double-quotes can be used:

```
symbols = {
  '+': 'plus'
  '-': 'minus'
}
```

Note that string interpolation is not supported in hash keys.

Streamlined JSON

CoffeeScript takes JSON and distills it to its essence. While full-blown JSON is perfectly valid, significant whitespace can be used in place of much of the "symbology": commas are optional between properties that are separated by new lines, and, best of all, curly braces are optional when an object's properties are indented. That means that the JSON above can be replaced with something more YAML-like:

```
father =
  name: 'John'
  daughter:
    name: 'Jill'
```

```
  son:
    name: 'Jack'
```

You can also use this streamlined notation inline:

```
fellowship = wizard: 'Gandalf', hobbits: ['Frodo', 'Pippin', 'Sam']
```

This code is equivalent to the following:

```
fellowship = {wizard: 'Gandalf', hobbits: ['Frodo', 'Pippin', 'Sam']}
```

The magic here is that every time the CoffeeScript compiler sees :, it knows that you're building an object. This technique is especially handy when a function takes a hash of options as its last argument:

```
drawSprite x, y, invert: true
```

Same-Name Key-Value Pairs

One handy trick that CoffeeScript offers is the ability to omit the value from a key-value pair when the value is a variable named by the key. For instance, the following two pieces of code are equivalent. Here's the short way:

```
delta = '\u0394'
greekUnicode = {delta}
```

This is a little longer:

```
delta = '\u0394'
greekUnicode = {delta: delta}
```

(Note that this shorthand only works with explicit curly braces.) We'll discover a common use for this trick in Section 3.6, *Pattern Matching (or, Destructuring Assignment)*, on page ?.

Soaks: 'a?.b'

Before we move on to arrays, there's one last CoffeeScript feature you should be aware of when accessing object properties: the existential chain operator, which has come to be known as the "soak."

Soaks are a special case of the existential operator we met in Section 2.5, *Default Arguments (arg =)*, on page ?. Recall that `a = b ? c` means "Set `a` to `b` if `b` exists; otherwise, set `a` to `c`." But let's say that we want to set `a` to a *property* of `b` if `b` exists. A naïve attempt might look like this:

```
a = b.property ? c  # bad!
```

The problem? If b doesn't exist when this code runs, we'll get a ReferenceError. That's because the code only checks that b.property exists, implicitly assuming that b itself does.

The solution? Put a ? before the property accessor:

```
a = b?.property ? c  # good
```

Now if either b or b.property doesn't exist, a will be set to c. You can chain as many soaks as you like, with both dots and square brackets, and even use the syntax to check whether a function exists before running it:

```
cats?['Garfield']?.eat?() if lasagna?
```

In one line, we just said that *if* there's lasagna and *if* we have cats and *if* one is named Garfield and *if* Garfield has an eat function, then run that function!

Pretty cool, right? But sometimes the universe is a little bit more orderly than that. And when I think of things that are ordered, a very special kind of object comes to mind.

3.2 Arrays

While you could use any old object to store an ordered list of values, arrays (which inherit the properties of the Array prototype) offer you several nice features.[1]

Arrays can be defined using JSON-style syntax:

```
mcFlys = ['George', 'Lorraine', 'Marty']
```

This is equivalent to the following:

```
mcFlys = new Array()
mcFlys[0] = 'George'
mcFlys[1] = 'Lorraine'
mcFlys[2] = 'Marty'
```

Remember that all object keys are converted to strings, so arr[1], arr['1'], and even arr[{toString: -> '1'}] are synonymous. (When an object has a toString method, its return value is used when the object is converted to a string.)

Because arrays are objects, you can freely add all kinds of properties to an array, though it's not a common practice. It's more common to modify the Array prototype, adding special methods to all arrays. For instance, the Pro-

1. http://developer.mozilla.org/en/JavaScript/Reference/Global_Objects/Array

totype.js framework does this to make arrays more Ruby-like, adding methods like flatten and each.

Ranges

Fire up the REPL, because the best way to get acquainted with CoffeeScript range syntax—and its close friends, the slice and splice syntaxes, introduced in the next section—is ('practice' for i in [1..3]).join(', ').

CoffeeScript adds a Ruby-esque syntax for defining arrays of consecutive integers:

```
coffee> [1..5]
[1, 2, 3, 4, 5]
```

The .. defines an *inclusive range*. But often, we want to omit the last value; in those cases, we add an extra . to create an *exclusive range*:

```
coffee> [1...5]
[1, 2, 3, 4]
```

(As a mnemonic, picture the extra . replacing the end value.) Ranges can also go backward:

```
coffee> [5..1]
[5, 4, 3, 2, 1]
```

No matter which direction the range goes in, an exclusive range omits the end value:

```
coffee> [5...1]
[5, 4, 3, 2]
```

This syntax is rarely used on its own, but as we'll soon see, it's essential to CoffeeScript's for loops.

Slicing and Splicing

When you want to tear a chunk out of a JavaScript array, you turn to the violent-sounding slice method:

```
coffee> ['a', 'b', 'c', 'd'].slice 0, 3
['a', 'b', 'c']
```

The two numbers given to slice are indices; everything from the first index up to *but not including* the second index is copied to the result. You might look at that and say, "That sounds kind of like an exclusive range." And you'd be right:

```
coffee> ['a', 'b', 'c', 'd'][0...3]
```

```
['a', 'b', 'c']
```

And you can use an inclusive range, too:

```
coffee> ['a', 'b', 'c', 'd'][0..3]
['a', 'b', 'c', 'd']
```

The rules here are *slightly* different than they were for standalone ranges, though, due to the nature of slice. Notably, if the first index comes after the second, the result is an empty array rather than a reversal:

```
coffee> ['a', 'b', 'c', 'd'][3...0]
[]
```

Also, negative indices count backward from the end of the array. While arr[-1] merely looks for a property named '-1', arr[0...-1] means "Give me a slice from the start of the array up to, but not including, its last element." In other words, when slicing arr, -1 means the same thing as arr.length - 1.

If you omit the second index, then the slice goes all the way to the end, whether you use two dots or three:

```
coffee> ['this', 'that', 'the other'][1..]
['that', 'the other']
coffee> ['this', 'that', 'the other'][1...]
['that', 'the other']
```

CoffeeScript also provides a shorthand for splice, the value-inserting cousin of slice. It looks like you're making an assignment to the slice:

```
coffee> arr = ['a', 'c']
coffee> arr[1...2] = ['b']
coffee> arr
['a', 'b']
```

The range defines the part of the array to be replaced. If the range is empty, a pure insertion occurs at the first index:

```
coffee> arr = ['a', 'c']
coffee> arr[1...1] = ['b']
coffee> arr
['a', 'b', 'c']
```

One important caveat: While negative indices work great for slicing, they fail completely when splicing. The trick of omitting the last index works fine, though:

```
coffee> steveAustin = ['regular', 'guy']
coffee> replacementParts = ['better', 'stronger', 'faster']
coffee> steveAustin[0..] = replacementParts
coffee> steveAustin
```

Slicing Strings

Curiously, JavaScript provides strings with a slice method, even though its behavior is identical to substring. This is handy, because it means you can use CoffeeScript's slicing syntax to get substrings:

```
coffee> 'The year is 3022'[-4..]
3022
```

However, don't get too carried away—while slicing works fine on strings, splicing doesn't. Once a JavaScript string is defined, it can never be altered.

```
['better', 'stronger', 'faster']
```

That does it for slicing and splicing. You should now consider yourself a wizard when it comes to extracting substrings and subarrays using ranges! But ranges have another, even more fantastical use in the for...in syntax, as we'll see in the next section.

3.3 Iterating over Collections

There are two built-in syntaxes for iterating over collections in CoffeeScript: one for objects and another for arrays (and other enumerable objects, but usually arrays). The two look similar, but they behave very differently:

To iterate over an object's properties, use this syntax:

```
for key, value of object
  # do things with key and value
```

This loop will go through all the keys of the object and assign them to the first variable named after the for. The second variable, named value above, is optional; as you might expect, it's set to the value corresponding to the key. So, value = object[key].

For an array, the syntax is a little different:

```
for value in array
  # do things with the value
```

Why have a separate syntax? Why not just use for key, value of array? Because there's nothing stopping an array from having extra methods or data. If you want the whole shebang, then sure, use of. But if you just want to treat the array as an array, use in—you'll only get array[0], array[1], etc., up to array[array.length - 1], in that order.

'hasOwnProperty' and 'for own'

JavaScript makes a distinction between properties "owned" by an object and properties inherited from its prototype. You can check whether a particular property is an object's own by using object.hasOwnProperty(key).

Because it's common to want to loop through an object's own properties, not those it shares with all its siblings, CoffeeScript lets you write for own to automatically perform this check and skip the properties that fail it. Here's an example:

```
for own sword of Kahless
  ...
```

This is shorthand for the following:

```
for sword of Kahless
  continue unless Kahless.hasOwnProperty(sword)
  ...
```

Whenever a for...of loop is giving you properties you didn't expect, try using for own...of instead.

No Scope for 'for'

When you write for x of obj or for x in arr, you're making assignments to a variable named x in the current scope. You can take advantage of this by using those variables after the loop. Here's one example:

```
for name, occupation of murderMysteryCharacters
  break if occupation is 'butler'
console.log "#{name} did it!"
```

Here's another:

```
countdown = [10..0]
for num in countdown
  break if abortLaunch()
if num is 0
  console.log 'We have liftoff!'
else
  console.log "Launch aborted with #{num} seconds left"
```

But this lack of scope can also surprise you, especially if you define a function within the for loop. So when in doubt, use do to capture the loop variable on each iteration:

```
for x in arr
  do (x) ->
    setTimeout (-> console.log x), 0
```

We'll review this issue in the Section 3.9, *Exercises*, on page ?.

Both styles of for loops can be followed by a when clause that skips over loop

iterations when the given condition fails. For instance, this code will run each function on obj, ignoring nonfunction properties:

```
for key, func of obj when typeof func is 'function'
  func()
```

And this code only sets highestBid to bid when bid is greater.

```
highestBid = 0
for bid of entries when bid > highestBid
  highestBid = bid
```

Of course, we could write continue unless condition at the top of these loops instead; but when is a useful syntactic sugar, especially for one-liners. It's also the only way to prevent any value from being added to the list returned by the loop, as we'll see in Section 3.5, *Comprehensions*, on page 14.

for...in supports an additional modifier not shared by its cousin for...of: by. Rather than stepping through an array one value at a time (the default), by lets you set an arbitrary step value:

```
decimate = (army) ->
  execute(soldier) for soldier in army by 10
```

Nor does the step value need to be an integer. Fractional values work great in conjunction with ranges:

```
animate = (startTime, endTime, framesPerSecond) ->
  for pos in [startTime..endTime] by 1 / framesPerSecond
    addFrame pos
```

And you can use negative steps to iterate backward through a range:

```
countdown = (max) ->
  console.log x for x in [max..0] by -1
```

Note, however, that negative steps are not supported for arrays. When you write for...in [start..end], start is the first loop value (and end is the last), so by step with a negative value works fine as long as start > end. But when you write for...in arr, the first loop index is always 0, and the last loop index is arr.length - 1. So if arr.length is positive, by step with a negative value will result in an infinite loop—the last loop index is never reached!

That's all you need to know about for...of and for...in loops. The most important thing to remember is that CoffeeScript's of is equivalent to JavaScript's in. Think of it this way: values live *in* an array, while you have keys *of* an array.

of and in lead double lives as operators: key of obj checks whether obj[key] is set, and x in arr checks whether any of arr's values equals x. As with for...in

loops, the in operator should only be used with arrays (and other enumerable entities, like arguments and jQuery objects). Here's an example:

```
fruits = ['apple', 'cherry', 'tomato']
'tomato' in fruits       # true
germanToEnglish: {ja: 'yes', nein: 'no'}
'ja' of germanToEnglish  #true
germanToEnglish[ja]?
```

What if you want to check whether a nonenumerable object contains a particular value? Let's save that for an exercise.

3.4 Conditional Iteration

If you're finding for...of and for...in a little perplexing, don't worry—there are simpler loops to be had. In fact, these loops are downright self-explanatory:

```
makeHay() while sunShines()

makeHay() until sunSets()
```

As you've probably guessed, until is simply shorthand for while not, just as unless is shorthand for if not.

Note that in both these syntaxes, makeHay() isn't run at all if the condition isn't initially met. There's no equivalent of JavaScript's do...while syntax, which runs the loop body at least once. We'll define a utility function for this in the exercises for this chapter.

In many languages, you'll see while true loops, indicating that the block is to be repeated until it forces a break or return. CoffeeScript provides a shorthand for this, the simply-named loop:

```
loop
  console.log 'Home'
  break if @flag is true
  console.log 'Sweet'
  @flag = true
```

Note that all loop syntaxes except loop allow both postfix and indented forms, just as if/unless does. loop is unique in that it's prefixed rather than postfixed, like so:

```
a = 0
loop break if ++a > 999
console.log a  # 1000
```

Though while, until and loop aren't as common as for syntax, their versatility should make them an invaluable addition to your repertoire.

Next up, we'll answer an ancient Zen koan: What is the value of a list?

3.5 Comprehensions

In functional languages like Scheme, Haskell, and OCaml, you rarely need loops. Instead, you iterate over arrays with operations like map, reduce, and compact. Many of these operations can be added to JavaScript through libraries, such as Underscore.js. [2] But to gain maximum succinctness and flexibility, a language needs array comprehensions (also known as list comprehensions).

Think of all the times you've looped over an array just to create another array based on the first. For instance, to negate an array of numbers in JavaScript, you'd write the following:

```
positiveNumbers = [1, 2, 3, 4];
negativeNumbers = [];
for (i = 0; i < positiveNumbers.length; i++) {
  negativeNumbers.push(-positiveNumbers[i]);
}
```

Now here's the equivalent CoffeeScript, using an array comprehension:

```
negativeNumbers = (-num for num in [1, 2, 3, 4])
```

You can also use the comprehension syntax with a conditional loop:

```
keysPressed = (char while char = handleKeyPress())
```

Do you see what's going on here? Every loop in CoffeeScript returns a value. That value is an array containing the result of every loop iteration (except

2. http://documentcloud.github.com/underscore/

those skipped by a continue or break or as a result of a when clause). And it's not just one-line loops that do this:

```
code = ['U', 'U', 'D', 'D', 'L', 'R', 'L', 'R', 'B', 'A']
codeKeyValues = for key in code
  switch key
    when 'L' then 37
    when 'U' then 38
    when 'R' then 39
    when 'D' then 40
    when 'A' then 65
    when 'B' then 66
```

(Do you see why we needed to use parentheses for the one-liners, but we don't here? Also, you're probably wondering about switch; it'll become clearer in *Polymorphism and Switching*, on page ?.)

Note that you can use comprehensions in conjunction with the for loop modifiers, by and when:

```
evens = (x for x in [2..10] by 2)

isInteger = (num) -> num is Math.round(num)
numsThatDivide960 = (num for num in [1..960] when isInteger(960 / num))
```

List comprehensions are the consequence of a core part of CoffeeScript's philosophy: everything in CoffeeScript is an expression. And every expression has a value. So what's the value of a loop? An array of the loop's iteration values, naturally.

Another part of CoffeeScript's philosophy is DRY: Don't Repeat Yourself. In the next section, we'll meet one of my favorite antirepetition features.

Running JavaScript on the server has long been a dream of web developers. Rather than switching back and forth between a client-side language and a server-side language, a developer using a JavaScript-powered server would only need to be fluent in that lingua franca of web apps—or in its twenty-first-century offshoot, CoffeeScript.

Now that dream is finally a reality. In this chapter, we'll take a brief tour of Node.js, starting with its module pattern (part of the CommonJS specification). Then we'll figure out just what an "evented architecture" is, with its implications for both server performance and our sanity. Finally, we'll add a Node back end to our 5x5 project from the last chapter, with real-time multiplayer support powered by WebSocket.

6.1 What Is Node.js?

Ignore the name: Node.js isn't a JavaScript library. Instead, Node.js is a JavaScript *interpreter* (powered by V8, the engine used by Google's Chrome browser) that interfaces with the underlying operating system. That way, JavaScript run by Node.js can read and write files, spawn processes, and—most enticingly—send and receive HTTP requests.

Like CoffeeScript, Node is a new project (dating to early 2009) that's taken off rapidly and attracted all kinds of excitement. Witness the Node.js Knockout, a Rails Rumble-inspired competition to develop the best Node app in forty-eight hours.[1]

A number of awesome projects have already been written with Node and CoffeeScript. The following is a small, select sampling. You might want to come back to this list after you've completed the book; reading real-world code is a great way to take your mastery to the next level:

- *Docco* [Ash11]: Uber-computer scientist Donald Knuth advocated "literate programming," in which code and comments are written so that someone encountering the program for the first time can understand it just by reading it once. Docco, written by Jeremy Ashkenas, supports this methodology by generating beautiful web pages in which comments and code are displayed side-by-side.
- *Eco* [Ste11]: Say you're writing a Node-based web application. You have all of these HTML skeletons and a mess of application code, but you're not sure how to combine the two. Eco lets you embed CoffeeScript *within* your markup, turning it into a server-side templating language.

1. http://nodeknockout.com/

- *Zappa* [NM11]: Creating web applications from scratch has never been simpler. Zappa is a layer on top of Node's popular Express framework that lets you succinctly define how your web server should respond to arbitrary HTTP requests.[2] Works great with Eco, too!
- *Zombie.js* [Ark11]: There's a new kid on the full-stack web app testing block: Zombie.js. Zombie lets you validate your application's behavior with the power of Sizzle, the same selection engine that powers jQuery. Not only is it easy to use, it's also insanely fast.

You can find a more comprehensive list of CoffeeScript-powered apps of all kinds at http://github.com/jashkenas/coffee-script/wiki/In-The-Wild.

6.2 Modularizing Code with 'exports' and 'require'

In past chapters, we've used global to put variables in an application-wide namespace. While global has its place, Noders generally prefer to keep their code nice and modular, with each file having its own namespace. How, then, do you share objects from one file with another?

The answer is a special object called exports, which is part of the CommonJS module standard. A file's exports object is returned when another file calls require on it. So, for instance, let's say that I have two files:

Download Nodejs/app.coffee

```
util = require './util'

console.log util.square(5)
```

Download Nodejs/util.coffee

```
console.log 'Now generating utility functions...'
exports.square = (x) -> x * x
```

When you run coffee app.coffee, require './util' executes util.coffee and then returns its exports object, giving you the following:

```
Now generating utility functions...
25
```

You might be wondering why we didn't need to specify a file extension. A .js file extension is always optional under Node.js. .coffee is also optional but only if the running application has loaded the coffee-script library, which is always implicitly done when we use coffee to run a file. coffee-script also tells Node how to handle CoffeeScript files. So if we compiled app to JavaScript but not util, then we'd have to write this:

2. http://expressjs.com

Download Nodejs/app.js

```
require('coffee-script');
var util = require('./util');
console.log(util.square(5));
```

When a library's name isn't prefixed with . or /, Node looks for a matching file in one of its library paths, which you can see by looking at require.paths.

By convention, a library's name for require is the same as its name for npm. Recall, for instance, that we used npm install -g coffee-script to install CoffeeScript. That gave us the coffee binary, but also the coffee-script library. We'll be using npm to install some more libraries for our project at the end of this chapter.

6.3 Thinking Asynchronously

One of the most common complaints about JavaScript has always been its lack of support for threading. While popular languages like Java, Ruby, and Python allow several tasks to be carried out simultaneously, JavaScript is strictly linear.

Yet what might seem on its surface to be JavaScript's greatest weakness is now widely seen as a blessing in disguise. Without threads, there are no mutexes, no race conditions, no endless sleep loops. Many of the most common sources of software bugs are banished. What's more, multithreading often adds significant overhead to an application, especially to web servers, which is one reason why Node.js has a reputation as an efficient alternative to frameworks in languages that typically rely on threads for concurrency.

(Of course, without threads there's no way to take advantage of multiple processors. The good news is that there are already projects out there, such as multi-node and cluster, that effectively bind multiple instances of your app to the same server port, giving you the performance advantages of parallel processing without the headaches of sharing data across threads.[3])

Because JavaScript is event-oriented rather than thread-oriented, events only run when all other execution has stopped. Imagine how frustrating it would be if every time your application made a request (say, to the file system or to an HTTP server), it froze up completely until the request was completed! For that reason, nearly every function in the Node.js API uses a callback: you make your request, Node.js quickly passes it along, and your application continues as if nothing happened. When your request is completed (or goes awry), the function you passed to Node.js gets called.

3. http://github.com/kriszyp/multi-node and http://github.com/learnboost/cluster, respectively.

For example, if you wanted to show the contents of the current directory, you would write the following:

```
fs = require 'fs'
fs.readdir '.', (err, files) ->
  console.log files
console.log 'This will happen first.'
```

Here's what happens:

1. We ask Node.js to read the current directory with fs.readdir, passing a callback.
2. Node.js passes the request along to the operating system, then immediately returns.
3. We print 'This will happen first.' to the console.
4. Once our code has run, Node.js checks to see if the operating system has answered our request yet. It has, so it runs our callback, and a list of files in the current directory is printed to the console.

You got that? This is very important to understand. *Your code is never interrupted.* No matter how many RPMs your hard drive has, that callback isn't getting run until after *all* of your code has run. JavaScript code never gets interrupted. Even the seemingly precise setTimeout and setInterval will wait forever if your code gets stuck in an infinite loop.

All of that is as true in the browser as it is in Node, but it's doubly important to understand in Node because your application logic *will* take the form of tangled chains of callbacks. Have no doubt about it. The challenge is to manage them in a way that humans can understand.

Consider how a simple form submission to a web application gets handled:

1. We get the user's information from the database to check that they have permission to make the request.
2. If so, we update the database accordingly.
3. We read a template from the file system, customize it appropriately, and send it to the user.

Then, at the very least, our application skeleton looks like this:

```
formRequestReceived = (req) ->
  checkDatabaseForPermissions req, ->
    updateDatabase req, ->
      renderTemplate req, (tmpl) ->
        sendResponse tmpl
```

And that's without error-handling at each step!

Unfortunately, that matryoshka doll feeling is never quite going to go away. The fact is, in most languages you'd rely on threads so that you could just write something like this:

```
formRequestReceived = (req) ->
  if checkDatabaseForPermissions req
    updateDatabase req
    tmpl = renderTemplate req
    sendResponse tmpl
```

But those languages are pretending to synchronize the asynchronous. Somewhere in each of those database-calling and file-reading functions, there's a sleep loop saying, "I hope someone else does something useful while I wait to hear from the database." It's simpler on the surface, but at a price in memory, CPU, and—more often than not—unpleasant surprises.

Note that many NodeJS API functions *do* offer a synchronous version for convenience. For instance, instead of using fs.readdir with a callback, you can call fs.readdirSync and simply get the list of filenames returned to you. If your application doesn't have any events waiting to fire, then there's no reason not to use these convenient alternatives.

Unfortunately, there's no way to implement a synchronous version of an arbitrary asynchronous function in JavaScript or CoffeeScript. It's only possible using native extensions (typically written in C++), which are beyond the scope of this book.

Scope in Loops

Remember what we learned in Section 2.2, *Scope: Where You See 'Em*, on page ?: *only functions create scope.* Expecting loops to create scope leads otherwise mild-mannered programmers to summon forth horrific bugs when dealing with asynchronous callbacks. In fact, this is probably the most common source of confusion in asynchronous code.

For instance, let's say that we have an application that loads numbers from some (synchronous) source and keeps a running tally of those numbers until the sum meets or exceeds limit. Each time a number is loaded, that number—and the sum thus far—needs to be saved. Also, due to overzealous security requirements, each save needs to be encrypted using a key unique to the given number. That key must be fetched *asynchronously* via the getEncryptionKey function.

A first attempt might look like this:

```
sum = 0
while sum < limit
  sum += x = nextNum()
  getEncryptionKey (key) ->
    saveEncrypted key, x, sum  # FAIL!
```

The problem here is that by the time the getEncryptionKey callback is called, x and sum have moved on—in fact, the entire loop has been run. So for each x the loop goes through, the values of x and sum after the loop has finished running will be saved (most likely with the wrong encryption key).

The solution is to *capture* the values of x and sum. The easiest way to do that is to use an anonymous function. The do keyword was added to CoffeeScript for precisely this purpose:

```
sum = 0
while sum < limit
  sum += x = nextNum()
  do (x, sum) ->
    getEncryptionKey (key) ->
      saveEncrypted key, x, sum  # Success!
```

If you're familiar with Lisp, this use of do should remind you of the let keyword. do (x, sum) -> ... is shorthand for ((x, sum) -> ...)(x, sum). So now the line saveEncrypted key, x, sum references the copies of x and sum created by the do instead of the x and sum used by the loop.

Note that this form shadows the outer x and sum, making them inaccessible. If you want to have access to the original variables while still capturing their values, then you might write something like this:

```
do ->
  capturedX = x; capturedSum = sum
  ...
```

Now it's time for a little project of our own, extending the jQuery version of 5x5 with a Node-powered back end.