Extracted from:

CoffeeScript

Accelerated JavaScript Development

The Pragmatic Bookshelf

Dallas, Texas • Raleigh, North Carolina

The Pragmatic Programmers

CoffeeScript

Accelerated JavaScript Development

Trevor Burnham

Foreword by Jeremy Ashkenas

edited by Michael Swaine

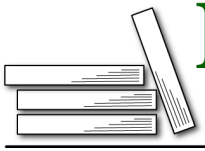
CoffeeScript

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

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.