

CoffeeScript

IN ACTION

Patrick Lee



MEAP



**MEAP Edition
Manning Early Access Program
CoffeeScript in Action version 4**

Copyright 2012 Manning Publications

For more information on this and other Manning titles go to
www.manning.com

Table of Contents

Part 1 Foundations

- 1 The road to CoffeeScript
- 2 Syntax
- 3 Functions
- 4 Dynamic objects

Part 2 Composition

- 5 Composing with objects
- 6 Composing with functions
- 7 Idioms and what not to write
- 8 Metaprogramming
- 9 Algorithms, time and event loops

Part 3 Writing programs

- 10 Server techniques
- 11 Driving with tests
- 12 Client and interface techniques
- 13 Builds and applications
- 14 The future

Appendixes

- A Reserved words
- B Answers to exercises

1

The road to CoffeeScript

This chapter covers:

- The evolution of JavaScript
- How CoffeeScript was invented
- Languages that compile to JavaScript
- The CoffeeScript way of thinking

CoffeeScript is JavaScript, the same language with a different accent. At the core JavaScript is an elegant programming language with simple but powerful features, but it has some gnarly edges that over time are making less and less sense to a growing number of language users. CoffeeScript, developed by Jeremy Ashkenas, takes JavaScript as it is used today and forges a path that removes as much syntactic noise and irregularity as possible, borrowing ideas from other popular dynamic languages such as Ruby and Python along the way. CoffeeScript is a programming language that compiles to JavaScript but in the end it's still just JavaScript.

Creating languages for humans to interact with computers has been a hit-and-miss affair for generations with the popularity of any given programming language often due to chance. Still, new programming languages appear every day because humans are language creators - a habit many thousands of years old. Beyond programming languages, people have made it their life's work to create a language that is better than the incumbent. Whether a language for world peace such as Zamenhof's Esperanto, a logical language such as Brown's Loglan or something else these constructed languages fail to get adopted. The evolved languages win every time.

Over the entire history of human languages, efforts to control language, to dictate the rules and usage of languages, have failed. Fifteen generations of native English speakers

©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=799>

shows that the language emerges through human interaction with that language. The languages that people speak are constantly evolving and changing through being used in the world that they are part of. The same is true of programming languages. It is not the best-designed languages that win but rather, the language that the people own.

JavaScript, being the language of the web, is the most widely distributed and used programming language in the history of programming languages. With the increasingly social nature of the web, and of programming for the web, the evolution for JavaScript is the closest to the evolution of natural languages of any programming language. JavaScript as a language and as a community embraces change. Change is happening. CoffeeScript is one way to take the change and start using some of it today — instead of waiting for a committee to invent and standardize the language that people might want.

This chapter starts with some of the history of JavaScript, and then looks generally at how languages evolve through time with the community of people that use them and how new languages are emerging that compile to JavaScript. Finally, this chapter looks at why CoffeeScript is a good option for a language that compiles to JavaScript. As most of this chapter is more about the why of CoffeeScript and not the how, if you want to get straight down to learning how to program in CoffeeScript without this detour then go straight to chapter 2 and revisit this chapter later. Otherwise, you start with a history lesson.

1.1 The nature of JavaScript

Remember, CoffeeScript is just JavaScript. To understand CoffeeScript you need to understand JavaScript. In this section you will see the languages that influenced JavaScript - from C to Scheme to Self - and how JavaScript's C-style syntax does not quite fit with some of the concepts it takes from Scheme and Self. You will then see how CoffeeScript fits into this picture. You begin with the curlyes.

1.1.1 Curly braces

JavaScript looks like C. That was deliberate. Most of the popular mainstream programming languages in history have looked like C. The most obvious visual attribute of a C-style programming language is that curly braces { and } are used as delimiters for blocks of code. If you don't know what a block of code is just know that programs in a C-style language have curly braces everywhere.

If you studied Computer Science when grunge music was popular all the cool kids were using C with curly braces. That was real programming. Real programming meant managing memory and manipulating strings as arrays of char pointers. Real programming was how computer games were made. The C programming language was the manliest think to write besides assembly. The Computer Science departments in universities around the world were full of boys. Nobody wanted to write assembly though. Years of standing in assemblies at high school probably didn't help.

The schools of Computer Science were motivated to produce graduates who could get jobs so the three most popular languages at the time: C, C++ and Java were usually taught.

©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=799>

If you missed the memo, C++ and Java also have curly braces. Programming in C or C++ generally required students to be familiar with Unix and be comfortable with memory management and not having a point-and-click development environment. These were contributing factors in many people migrating to Java or perhaps to other lesser-known programming languages such as Perl or Python. Those who did not find a path to Perl or Python gave up programming altogether and, as the popular sentiment went back in those days, took up web development instead.

Outside of the best sellers of pop programming there was a varied ecosystem of different languages and language paradigms. JavaScript, despite dressing in curly braces and a semicolon top hat like a C family language took two important ideas from less popular languages Scheme and Self - neither of which had curly braces. In order to fully understand JavaScript it is important to learn about Scheme and Self.

1.1.2 Scheme

Scheme is a Lisp dialect created by Guy Steele and Gerald Sussman. The late John McCarthy created Lisp when he was a young man. When Brendan Eich created JavaScript for NetScape's web browser all of McCarthy's hair was grey. Way back when men in rock bands had perms, dialects of Lisp were popular - they were not popular by the time Eich created JavaScript. Lisp's lack of popularity ended Eich's plan to put Scheme in the web browser... almost.

Lisp is based off a mathematical system called Lambda Calculus. This term is an intimidating one containing both a Greek letter and the name of something that you learned in high school mathematics. Shortly after John McCarthy created Lisp, Peter Landin showed that any imperative program, such as those written in C, could be expressed using Lambda Calculus.

Imperative programs give instructions to the machine, line by line, telling the machine what to do. An imperative program is one that looks a little bit like the list of chores that your mother used to leave you to do while you were a teenager at home from school during summer vacation:

1. Vacuum the lounge room
2. Hang out the washing
3. Unpack the dishwasher

Except that the machine actually carries out the instructions and in the order specified... usually. In Lambda Calculus, or in a purely functional program, instead of defining steps to do you define a function that describes the transformation from the initial state to the final state. What does Scheme look like?

```
(defun factorial (n)
  (if (<= n 1)
      1
      (* n (factorial (- n 1)))))
```

If you have never seen code written in a Lisp dialect before stay calm. Just look at the uniformity in the code. That uniformity can be confronting to beginners but at a deeper level

©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=799>

it leads to an important and powerful feature. In Scheme, as in Lisp, you can define all of the words and forms. You define the language.

How about JavaScript? Getting right to it, you are spared the eye-cutting curly braces of the JavaScript version in favor of the same factorial function written in CoffeeScript:

```
factorial = (n) ->
  if n <= 1
    1
  else
    n * factorial n - 1
```

That is elegant. There are few control characters. Compared to the JavaScript version thirty visible characters have been saved. Trivial? The characters that have been saved are not essential to the program. The characters removed do not matter to what the program does and importantly, do not matter to what you, the programmer, mean.

There are different dialects of Lisp and they have different standard libraries. The standard library for Common Lisp is large, whereas the standard library for Scheme is small. Scheme was developed as a minimalist dialect of Lisp with a small standard language core. Scheme is a small language. At heart, JavaScript too is a small language. Unfortunately it was created and standardized very quickly and some parts of it got burnt.

1.1.3 Self

Whilst the popularity of Lisp was waning object-oriented programming was starting to gain widespread acceptance. Object-oriented programming models things explicitly in the program. A standard, albeit contrived, example is that a program that did something related to cars would have car objects in it.

The Self Programming Language, developed as a research project by David Ungar and Randall Smith, was different from other object-oriented languages at the time because it did have a special way of dealing with classes. In most object-oriented languages classes are used to declare the basic building blocks from which the entire system is created. In an object-oriented language with classes to get a new car you define a Car class and then say, "give me a new one of those Car things." In Self, you instead just create something and call it a car. If you need another car you take the old car and say, "make another thing just like that one." Using classes is classical object-oriented. Using existing objects as prototypes for new objects is prototypical object-oriented. Classical is Plato to a Prototypical Heidegger¹.

Self not only has no concept of classes but the overall object model in self, the way that programs are put together, is based off only a few simple concepts: Prototypes, slots and behavior.

¹ Echoes of Plato's metaphysics can be seen in class-based languages where a chair object is created from an ideal Chair. In comparison, in a prototype-based language the prototype is a typical example, echoing the writings of Heidegger.

There were other prototype-based object languages being developed around the same time. One of them called Agora was referred to by Wolfgang De Meuter - one of its creators - as the Scheme of object-orientation.

Self and other prototypal languages never got much traction but research on those languages pioneered some of the techniques used to make JavaScript virtual machines run fast today. The programming techniques born in languages like Self are relevant to programming in CoffeeScript.

1.1.4 JavaScript to CoffeeScript

Brendan Eich wrote the first implementation of JavaScript in the time it takes to get over a cold. That resulted in some bad things and some good things. One of the good things is that it is a small language. A small language is a good place to start. Elaborate is desirable in a church or monument; it is not desirable in the design of a language.

In his seminal "Growing a language" talk, Guy Steele talks about how starting from a small language every time a program is written is not practical and that on the other hand designing a large language up front means that people will have too much to learn before they can be productive. Instead, it is better to start with a small language that can grow over time as people learn to use it.

JavaScript is a small and malleable language but the syntax noise makes it difficult to create new language constructs that feel natural because the new constructs will not look like what is built in to the language. In that way JavaScript is very much unlike Lisp. JavaScript needs to change.

JavaScript needs to grow but in part it is already a potted plant and there is danger of it being root-bound. As slave to a multitude of runtimes, as long as a browser holds market share code written in JavaScript needs to run on that browser. JavaScript programs have to maintain backwards compatibility in ways that other languages do not need to. This makes it especially difficult for JavaScript to grow as a language. By creating a language such as CoffeeScript that compiles to JavaScript it is possible to create any language. If you have time to create a grammar, lexer and parser for a language then great, do that. Otherwise you need a language that you can change - CoffeeScript allows this. It is important that you can change the language because that's how languages work, not just programming languages but all languages. Living languages change as the people using them change.

1.2 Language evolution

In Old Norse the word *happ* means luck. Twenty lives of man since the Vikings colonized Northern England there is - in Modern English - both happy and happen. Languages change and evolve over time.

Some time around the 5th Century a West Germanic tribe invaded Britain, bringing with them their language. As a result the existing Celtic and Cornish languages of the region were

replaced with a West Germanic dialect; only a hint of Celtic and Cornish remains today in Modern English.

A few hundred years later Vikings colonized parts of Northern Britain. They spoke a Scandinavian language and did not know how to speak the local West Germanic dialect very well but they still had to use it. The Vikings living in the area spoke a simplified, broken version of the local language. The way they spoke the language changed the way everybody spoke it.

In the eleventh century the Norman French conquered England and William the conqueror became king of England. During the Norman occupancy they decided that people would use their language of the region was Norman French. In the courts, in government and anything written down was Norman French. However, English was still by the commoners on the streets and on their farms. That the commoners used English on the farms is the reason English farm animals such as Cow and Pig have one name and the meat produced (used in the restaurants by people speaking Norman French) have other names such as Beef and Pork. For some other types of words there are three versions, the common name, the Norman French name and the really fancy Latin name. English is a diverse language.

When the Normans finally left the English language that emerged had changed dramatically from the language that had been there three hundred years before. Not only had it been influenced by Norman French but it had lost some more of the grammatical peculiarities of the West Germanic origins. Since nobody had been writing down much English for hundreds of years the way it was written had to change. Compounding that, when the printing press was invented it only have the Latin alphabet, lacking important characters such as the thorn **þ** and replacing their usage with hopefully equivalent digraphs (two letter combinations) such as **th**.

The result of all this was a language that had particular problems both in grammar and spelling. Grammarians spent some time trying to impose the grammatical structure of Latin onto what had become modern English but their grammatical structures didn't really fit English very well so far from simplifying English their input added more diversity. Spelling reform has been proposed for English almost since the 1500's. The only real success in changing the spelling of English was Webster when he published his *Dictionary of American English* and made it acceptable to use the spelling color as well as colour and standardize as well as standardise.

Throughout all of this human history where natural languages such as English had evolved there have been people trying to construct entire languages with the dream of creating a perfect language. A global language of peace, or a logically consistent language, or a language that was simple to learn.

In the history of constructed languages there are not many success stories. Volapük, one of the first to have an impact on an international scale fell after a schism between the inventor Johann Martin Schleyer and many of the active speakers of the language. Esperanto, perhaps the most well known constructed language has today somewhere between two hundred and one thousand native speakers. Constructed languages have also

©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=799>

been created out of natural languages such as Ogden Nash's Basic English, consisting of a set of only 850 English words meant to be easier to learn than English.

For the entire history of human languages people have tried to both create languages and control the direction of existing languages. The *Académie française* is the French learned body pertaining to issues on the French language. They publish the official dictionary of French. There is no single body or committee responsible for English.

Despite the difficulties, or perhaps because of them, English has emerged as a global language. Many languages have influenced English in the last fifteen hundred years. Being one of the primary languages of business and commerce today as well as being used in Western Culture media that is broadcast throughout the world English continues to be influenced by many speakers from different backgrounds and cultures. English is a truly global language.

This eventually leads back to JavaScript. As the unavoidable language of the web, as the language used to create experiences in the web browser, JavaScript has become a melting pot of different language ideas. JavaScript has changed a lot. The way it is used, the techniques and the types of things written in JavaScript have changed dramatically. What has not changed so much is the specification of the language. What has (almost) not changed at all is the syntax. This is unsurprising due to the way programming languages are perceived.

Programming languages feel fixed. You are taught the rules of a programming language and told that is all you can do. There are many books about programming languages that take a dictionary approach. These books document aspects of the language without actually telling you how to use it. A dictionary looks like a rulebook but, actually, dictionaries have to be updated every year to keep up with the way the language is actually used. A dictionary pretends it does not have an opinion on a language but it does. Just like Webster's dictionary that wanted to reform spelling.

This book is not a dictionary or reference book and it does not pretend to have no opinion. This book is about how to write CoffeeScript. The best way to learn CoffeeScript is by being exposed to it and using it. The best way to learn CoffeeScript is by being part of a community that uses it. This book is your companion in the world of CoffeeScript. You might think of it as Strider waiting to meet you at *The Prancing Pony*.

1.3 JavaScript as a language target

From birth into teenage years JavaScript was not a popular programming language. For a while, JavaScript was probably the most hated programming language in the world. However, it was the programming language of the browser and so was unavoidable. In order to at least avoid having to write lines of JavaScript many people wanted to find ways to either cross-compile their preferred language to JavaScript or two make JavaScript work more like their preferred language using a library. In this section you will see examples of both.

1.3.1 Libraries

In the Ruby community and in the Python community the first approach was to take JavaScript and create libraries that make it work more like Ruby or Python, writing libraries for creating classical object patterns in JavaScript, often by messing with `Object.prototype`. This was possible because JavaScript is a small and flexible language. JavaScript libraries such as *Prototype.js*, which came with the *Ruby on Rails* framework, were initially popular but they didn't completely with the larger body of people writing JavaScript - partly through clobbering other people's objects - and in the end jQuery won. These days jQuery code, both good and bad, is recognizable as idiomatic JavaScript. Consider the way chaining is done in jQuery:

```
$("#javascript").removeClass("c-style-syntax").addClass("significant
whitespace").append("fun");
```

Chaining function calls in that way is called a *fluent interface*. In JavaScript, it is about the best you can get for defining a language inside JavaScript without implementing an interpreter. The CoffeeScript directly translated version looks exactly the same and fluent interfaces are still a useful technique. What you will find with CoffeeScript is that there are additional ways to create readable code.

1.3.2 Languages that compile to JavaScript

Not everybody wrote libraries in JavaScript. Some preferred to write in a different language and then compile that language into JavaScript. From the Lisp community came Parenscript:

```
(defun next-image ()
  (when (< *current-image-index* (1- (getprop *images* 'length)))
    (show-image-number (1+ *current-image-index*))))
```

Parenscript allowed people to compile a subset of Common Lisp to JavaScript. That was nice but whilst JavaScript was influenced by Lisp it is not a Lisp.

Most other programming languages have their own version of this that compile JavaScript and sometimes other parts of the web stack - Red for Ruby, Pyjamas for Python, GWT for Java, Ocamljs for Ocaml, jshaskell for Haskell. These projects have some acceptance in their native source languages they have not been popular on a larger scale. Some have met with hostility from JavaScript developers. All of these solutions make the mistake of treating the problem as a technical one. They largely ignore the social and language aspects. Take Pyjamas for example, how much does this look like JavaScript?

```
from pyjamas.ui.HTML import HTML
from pyjamas.ui.RootPanel import RootPanel
from pyjamas.HTTPRequest import HTTPRequest
```

```
class SlideLoader:
    def __init__(self, panel):
        self.panel = panel
    def onCompletion(self, text):
        self.panel.setSlide(text)
```

If you've spent much time with JavaScript and you know and like Python, it's easy to feel uncomfortable seeing that instead of JavaScript. The same goes for Clamato, a Smalltalk that runs on JavaScript:

©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=799>

```

renderOn: html
  html h3 with: @counter.
  html button
    with: '+';
    onClick: [@counter := @counter + 1. self reload].
  html button
    with: '-';
    onClick: [@counter := @counter - 1. self reload].

```

What would work, in language terms, is to just take JavaScript, and occasionally mix in styles and techniques from other languages. Have different people bring different influences together and watch the languages merge. After all, that's the way languages evolve naturally.

The fourth edition of the ECMAScript specification (that was never cancelled) tried to make a lot of changes and take the language in a different direction. ActionScript 2 took some of that work and added types and class syntax was used in created Flash-based web applications. More recently, languages like Objective-J have added features to JavaScript and created a superset language.

CoffeeScript is the most natural language to compile to JavaScript because it is built from an understanding of how JavaScript works and how it is used. CoffeeScript is ecologically responsible.

1.4 *The CoffeeScript way*

You can't avoid JavaScript. You can avoid the syntax and possibly the language specification but you can't avoid the community of people who use the language. You would need a very large number of people to achieve language death for JavaScript. If you like JavaScript, or if you don't, it is here to stay. It will change, but it is not leaving any time soon. Understanding JavaScript and knowing good techniques is essential. As CoffeeScript is just JavaScript, many techniques that work in JavaScript are still applicable. CoffeeScript does some other things though. In this section you will see that CoffeeScript fixes some of the language problems in JavaScript whilst also having a simpler syntax. You will then see how this allows you to own the language. Finally, this section looks at the thought experiment that led to CoffeeScript being invented. Firstly though, JavaScript has some problems.

1.4.1 *Fixing language problems*

Some things in JavaScript that do not match the conceptual model are actually harmful to development. For one, JavaScript puts undeclared variables onto the global scope, like so:

```

holeInTheBucket = function () {
  thenFixIt = function () {
    console.log("With what shall I fix it?");
  };
};

```

```

thenFixIt(); // With what shall I fix it?

```

In JavaScript, when a variable cannot be found anywhere in a lexical lookup it is automatically defined on the global object. In a web browser this will be the Window object.

©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=799>

This is a bad thing. If you forget to declare a variable in JavaScript and you run your code in a browser that variable will be set on the `Window`. It may have been a convenient feature for novices trying to animate buttons in Netscape Navigator 4 but it is a bad thing. CoffeeScript fixes this global leakage. In JavaScript you can now `"use strict";` but in CoffeeScript global leakage is not possible *by default*.

Languages evolve over time and take influences but that to completely displace a language, to kill a language then the entire community of speakers needs to be wiped out. That's basically what happened to the Native America languages and what happened to Cornish and Celtic when the West Germanic tribes romped through Britain. Which is why JavaScript really had to be a curly-brace language, it had to survive in a world where all of the most popular languages had curly braces.

However, in the ensuing years Python and Ruby have made massive inroads in popularity and mindshare and dialects of Lisp are making something of a renaissance particularly with the growth in popularity of Clojure on the JVM². This is a good time for a language as simple and elegant as JavaScript but without the syntactic noise. That language is CoffeeScript. CoffeeScript removes curly braces from JavaScript. This is sensible as JavaScript only has function level lexical scoping of variables – it does not have block level variable scoping. Curly braces as block-delimiters made JavaScript look like it was supposed to have block level scoping. This code will not work in JavaScript the way a C-programmer who has never seen JavaScript might expect it to:

```
var i = 4;
for (var i = 0; i != 5; i++) {
  console.log(i);
}
console.log(i);
```

It is natural for a programmer coming from a C background to expect the variable to be scoped to the block whereas in JavaScript all variables are hoisted. By removing the curly braces the expectation that the language has block scope is gone. Even better, by removing curly braces the syntax can be made cleaner.

1.4.2 Cleaner syntax

CoffeeScript reduces the punctuation to user code ratio. Actually, code written in CoffeeScript can be some of the cleanest that you will find. It achieves this by removing things and by making things that are already there have a value other than zero. In the words of Edward Tufte, the data to ink ratio is maximized in CoffeeScript.

CoffeeScript removes curly braces and semicolons by making newlines and indentation mean something. This idea is well-known and loved to Python developers but in some circles even the mention of significant whitespace will send some developers into a diatribe involving tabs, spaces and make files.

² Java Virtual Machine

Java developers will be familiar with Ant, which uses XML to define project builds. Ruby developers will be familiar with Rake tasks, which use a Ruby DSL to define project builds. There is a much older Unix tool called make that is most commonly used to build C. If you've ever written much C you might remember your first Makefile and you probably remember that somebody else wrote it for you. One of the stumbling blocks for people learning make is that it uses tabs. The problem with tabs is that in the default editing-mode for most text editors the difference between a tab and a certain number of spaces is invisible. Significant whitespace though is fine if it's whitespace that users can see. Setup your editor so that it does not use tabs but inserts a series of spaces when the <TAB> key is used.

CoffeeScript also reduces the syntax of JavaScript by replacing commonly used programming patterns with more succinct versions and adds sugar such as ranges, splats and friendlier versions of some statements and operators.

CoffeeScript also makes parentheses optional for function calls. This is important for more than just removing a few characters. The biggest advantage is that it means user defined functions are able to look like language keywords. For example, a cleaner trait syntax like the following is possible, assuming we had two traits enumerable and comparable already defined:

```
interval = compose enumerable, comparable
  start: min
  end: max
  contains: (n) ->
    min <= n and n < max
```

The expressive power provided by the ability for you to easily extend the syntax this way, crafting a language that is your own is a powerful aspect of CoffeeScript that has not been used frequently enough in the early days of the language.

1.4.3 *You own the language*

It's important not to underestimate the power of a language that allows developers to escape the language, to create their own language. Humans, the language inventors, are very good at language - reading them, speaking them, writing them and inventing them. Humans invent languages and change languages, even as this book is being written, even as this book is being read, the languages spoken on the planet are evolving, changing. Not only that but in particular spheres of life particular languages are developing. Walk into the operating room whilst a surgery is going on and listen to the conversation. Humans are the language inventors.

Humans will invent the language they use. If they cannot change a language they grow bored of and invent a new one. Old languages are rarely fashionable. Consider this fragment of test code written using RSpec, a testing framework in Ruby:

```
describe Bowling, "#score" do
  it "returns 0 for all gutter game" do
    bowling = Bowling.new
    20.times do
      bowling.hit(0)
    end
  end
end
```

©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=799>

```

    bowling.score.should == 0
  end
end

```

And the same in JavaScript using Jasmine:

```

describe("Bowling", function() {
  it("returns 0 for all gutter games", function() {
    var bowling = new Bowling();
    for (var i = 0; i != 20; i++) {
      bowling.hit(0);
    }
    expect(bowling.score).toBe(0);
  });
});

```

Here's the same version of the Jasmine test, this time with CoffeeScript:

```

describe "Bowling", ->
  it "returns 0 for all gutter games", ->
    bowling = new Bowling()
    [0..20].forEach ->
      bowling.hit 0
    expect(bowling.score).toBe 0

```

The Ruby and CoffeeScript versions have less boilerplate. Whether you specifically like the different syntax doesn't matter. What does matter is that less built-in syntax means greater flexibility. Language users should not be tied to syntax rules determined by committee³. Consider these two quotes. One from from the inventor of Python and the other from the inventor of Ruby:

Don't bother users with details that the machine can handle

Guido van Rossum

For me the purpose of life is partly to have joy. Programmers often feel joy when they can concentrate on the creative side of programming; So Ruby is designed to make programmers happy.

Yukihiro Matsumoto

Python and ruby are the two languages that have had the most impact on the syntactic style of CoffeeScript. It started as a thought experiment by Jeremy Ashkenas to see what syntactic noise can be removed from JavaScript. You can try this yourself now.

1.4.4 The experiment

CoffeeScript starts with an experiment to remove as much non-essential syntax from JavaScript as possible. Consider the following JavaScript function:

³ Further, the rules of a language should not be determined by what is easier for a compiler. The language is for the humans, not for the computer.

```
var square = function (x) {
  return x * x;
};
```

Relax and follow along even if you don't know what this code means at this point. You will start to learn more about the language in the next chapter. For now it's just about syntax. How much of the syntax is really necessary to a human reader? For a start, the semicolons might be removed:

```
var square = function (x) {
  return x * x
}
```

The curly braces might be removed:

```
var square = function (x)
  return x * x
```

The var statement might be removed by making it implicit, instead of explicit:

```
square = function (x)
  return x * x
```

The same might apply to the return statement:

```
square = function (x)
  x * x
```

Finally, the function keyword is fundamental to the language and used frequently but it can be replaced with something just as clear - an arrow pointing from the function arguments to the function body:

```
square = (x) ->
  x * x
```

There you have CoffeeScript. It's just JavaScript. Depending on your background the syntax may feel more comfortable or less comfortable. What is important though is that the change in syntax leads to a change in thinking about how to program. This book is about learning CoffeeScript and about how learning CoffeeScript will help you learn new ways of programming.

1.5 Summary

For long-time JavaScript developers CoffeeScript is the language you're already using. For developers coming to JavaScript from other languages, CoffeeScript is the JavaScript you need. Remember that JavaScript is changing. Remember those Vikings? They came in and changed English forever because they didn't know or care how to handle the nuances of the Germanic language that was being spoken. English was changed forever. The same goes for JavaScript. The language simply cannot remain unchanged in the face of all the different people that are starting to use it.

Using a language that is entirely unlike JavaScript results in being isolated from the larger community that uses JavaScript and creates too much mental and administrative overhead. However, using a language that is close to JavaScript, that pays attention to JavaScript and how it is used allows developers to move forward with getting things done without being burdened by the baggage of the past. CoffeeScript, first released by Jeremy Ashkenas in 2008, is such a language.

If you're a long time JavaScript developer like I am, then you might have an inclination to not only be skeptical but to want to cling to the familiar syntax that you know. Some people might call that prudent; others might call it Stockholm syndrome. In the words of Bertrand Russell, I am firm. You are obstinate. He is a pig-headed fool. Willing to continue on your journey of learning CoffeeScript? Then read on; the next chapter introduces the syntax of the language and how to use it.