Mark Bates

Programming in CoffeeScript

Developer's Library

Programming in CoffeeScript

Programming in CoffeeScript

Mark Bates

Addison-Wesley

Upper Saddle River, NJ • Boston • Indianapolis • San Francisco
New York • Toronto • Montreal • London • Munich • Paris • Madrid
Cape Town • Sydney • Tokyo • Singapore • Mexico City

Programming in CoffeeScript
Copyright © 2012 by Pearson Education, Inc.

ISBN-13: 978-0-32-182010-5
ISBN-10: 0-32-182010-X

Library of Congress Cataloging-in-Publication Data is on file

Editor-in-Chief
Mark Taub

Acquisitions Editor
Debra Williams Cauley

Senior Development Editor
Chris Zahn

Managing Editor
Kristy Hart

Project Editor
Andy Beaster

Copy Editor
Barbara Hacha

Indexer
Tim Wright

Proofreader
Debbie Williams

Technical Editors
Stuart Garner
Dan Pickett

Publishing Coordinator
Olivia Basegio

Book Designer
Gary Adair

Compositor
Nonie Ratcliff

❖

Rachel, Dylan, and Leo: My life for you.

❖

Contents at a Glance

Table of Contents

Acknowledgments[1]

I said it in my first book, and I'll say it again here: Writing a book is incredibly hard work! Please make sure no one ever tells you differently. If they do, they are either an incredible liar or Stephen King. Fortunately for me I fall somewhere in the middle.

Writing a book is simultaneously a very independent and solitary activity, as well as a team effort. After I put the kids to bed, I head down to my office, crack open a few Guinesses (is the plural Guinei?), crank up the tunes, and work, by myself, into the wee hours of the morning. When I finish a chapter, I send it off to my editor, who then sends it off to a bunch of people who take what I have written and improve it in ways that I didn't know possible. Whether it's as simple as correcting grammar or spelling mistakes, to something more complex such as helping to improve the flow of the book, or point out where example code could be improved to further clarify a point. So, while the writing may be done alone in a dark room by yours truly, the final product is the culmination of many people's hard work.

In this section of the book, I get the chance to say thank you to those who help shape, define, and otherwise ensure that the book you are currently holding (or downloading) is of the highest quality it can be. So without further adieu I'm going to thank people Academy Awards style, knowing that I'm sure I've left someone off the list, for which I am incredibly sorry.

First and foremost I have to thank my beautiful wife, Rachel. Rachel is one of the most supportive and strong people I have ever met. Each night I get to crawl into bed beside her and each morning I get the joy of waking up next to her. I have the pleasure of staring into her eyes and seeing unconditional love there. I also get the encouragement to write books, start my own business, and to do whatever it is that will make me happiest in life. She gave me two handsome sons and in return I've given her bad jokes and my used cell phones. I clearly got the better end of the bargain in this marriage, and for that I am eternally grateful.

Next, I would like to thank my sons, Dylan and Leo. While neither of them directly contributed to this book, they do make life worth living and they give my life an energy and excitement that only children can. I love you boys both so very much.

Before moving off the subject of my family, I would like to thank my parents (especially you Mom!) and the rest of my family for always being there to both simultaneously support me and cut me down to size. I love you all.

Next I have to thank Debra Williams Cauley. Debra was my editor, handler, and psychiatrist on my first book, *Distributed Programming with Ruby*. I can only hope that other authors have the fortune to work with an editor as good as Debra. She truly has the patience of a saint.

[1] Many at my publishing house thought that my acknowledgments section, as well as other parts of this book, were a bit risqué, so the original has been edited down to what you see here. I apologize if you are offended by anything I wrote, that was never my intention. Apparently, I've been told, my sense of humor is not appreciated by all. If you do like bad fart jokes, then please follow me on Twitter @markbates.

I hope that should I ever write another book, Debra will be right there with me. I can’t imagine writing a book without her. Thank you, Debra.

When writing a technical book, there are people that are very important to the process; they are the technical reviewers. A technical reviewer’s job is to read each chapter and critique it from a technical standpoint, as well as answer the question, “Does it make sense to learn this here?” These reviewers are there to act as your audience. They are technically minded and know their subject. Therefore, the feedback that you get from them is incredibly important. On this book there have a been a few technical reviewers. But the two I really want to call out are Stuart Garner and Dan Pickett. Stuart and Dan went way above the call of duty on this book and were by no means afraid of telling me when I did or said something boneheaded. They received frantic phone calls and emails from me at all hours of the day and night and responded with amazing feedback. If I didn’t want all those sweet royalty checks all to myself I might’ve been tempted to cut them in. (Don’t worry, they got paid for their work. They each received a coupon for one free hour of “Mark” time.) Thank you Dan and Stuart, and the rest of the technical reviewers, for all of your hard work.

There are people who contribute to a book like this in different ways. Someone has to design the cover, index the book, write the language (CoffeeScript), or do any of the other countless jobs involved in something like this. Here is a list of some of those people (that I know about), in no particular order: Jeremey Ashkenas, Trevor Burnham, Dan Fishman, Chris Zahn, Gregg Pollack, Gary Adair, Sandra Schroeder, Obie Fernandez, Kristy Hart, Andy Beaster, Barbara Hacha, Tim Wright, Debbie Williams, Brian France, Vanessa Evans, Dan Scherf, Gary Adair, Nonie Ratcliff, and Kim Boedigheimer.

I would also like to thank everyone I have seen since I first starting writing this book who have heard me blather on for hours about it. I know it’s not that interesting to most people, but damn, do I love to hear the sound of my voice. Thank you all for not punching me in the mouth, like I probably deserve.

Finally, I would like to say thank you to you, the reader. Thank you for purchasing this book and helping to support people such as myself who, at the end of the day, really just want to help out our fellow developers by sharing the knowledge we have with the rest of the world. It’s for you that I have put the countless hours of work and toil into this book. I hope that by the time you close the cover, you will have gained a better understanding of CoffeeScript and how it can impact your development. Good luck.

About the Author

Mark Bates is the founder and chief architect of the Boston-based consulting company Meta42 Labs. Mark spends his days focusing on new application development and consulting for his clients. At night he writes books, raises kids, and occasionally he forms a band and “tries to make it.”

Mark has been writing web applications, in one form or another, since 1996. His career started as a UI developer writing HTML and JavaScript applications before moving toward the middle(ware) with Java and Ruby. Nowadays, Mark spends his days cheating on Ruby with his new mistress, CoffeeScript.

Always wanting to share his wisdom, or more correctly just wanting to hear the sound of his own voice, Mark has spoken at several high-profile conferences, including RubyConf, RailsConf, and jQueryConf. Mark has also taught classes on Ruby and Ruby on Rails. In 2009 Mark’s first (surprisingly not his last!) book, *Distributed Programming with Ruby*, was published by Addison-Wesley.

Mark lives just outside of Boston with his wife, Rachel, and their two sons, Dylan and Leo. Mark can be found on the web at: http://www.markbates.com, http://twitter.com/markbates, and http://github.com/markbates.

Preface

I started my professional development career in 1999, when I first was paid a salary to be a developer. (I don't count the few years before that when I was just having fun playing around on the Web.) In 1999 the Web was a scary place. HTML files were loaded down with `font` and `table` tags. CSS was just coming on the scene. JavaScript[1] was only a few years old, and a battlefield of various implementations existed across the major browsers. Sure, you could write some JavaScript to do something in one browser, but would it work in another browser? Probably not. Because of that, JavaScript got a bad name in the early 2000s.

In the middle of the 2000s two important things happened that helped improve JavaScript in the eyes of web developers. The first was AJAX.[2] AJAX enabled developers to make web pages more interactive, and faster, by making remote calls back to the server in the background without end users having to refresh their browsers.

The second was the popularity of JavaScript libraries, such as Prototype,[3] that made writing cross-browser JavaScript much simpler. You could use AJAX to make your applications more responsive and easier to use and a library like Prototype to make sure it worked across major browsers.

In 2010, and certainly in 2011, the Web started evolving into "single page" applications. These applications were driven through the use of JavaScript frameworks, such as Backbone.js.[4] These frameworks allowed the use of an MVC[5] design pattern using JavaScript. Whole applications would be built in JavaScript and then downloaded and executed in the end user's browser. This all made for incredibly responsive and rich client-side applications.

On the developer's side, however, things weren't all roses. Although the frameworks and tools made writing these sorts of applications easier, JavaScript itself proved to be the pain point. JavaScript is at times both an incredibly powerful language and an incredibly frustrating one. It is full of paradoxes and design traps that can quickly make your code unmanageable and bug ridden.

So what were developers to do? They want to build these great new applications, but the only universally accepted browser language is JavaScript. They could certainly write these applications in Flash,[6] but that would require plug-ins, and it won't work on some platforms, such as iOS[7] devices.

I first discovered CoffeeScript[8] in October 2010. CoffeeScript promised to help tame JavaScript and to expose the best parts of the quirky language that is JavaScript. It presented a cleaner syntax, like forgoing most punctuation in favor of significant whitespace and protection from those design traps that awaited JavaScript developers at every turn, such as poor scoping and misuse of the comparison operators. Best of all, it did all this while compiling to standard JavaScript that could then be executed in any browser or other JavaScript runtime environment.

When I first used CoffeeScript, the language was still very rough around the edges, even at version 0.9.4. I used it on a project for a client to try it out to see whether it was worth the little bit of hype I was hearing. Unfortunately, at the time two things caused me to push it aside. The first was that it was still not quite ready for prime time. There were too many bugs and missing features.

The second reason why I didn't use CoffeeScript was because the app I was trying it out on wasn't a very JavaScript-heavy application. I was mostly doing a few validation checks and an occasional bit of AJAX, which Ruby on Rails[9] helped me do with very little, if any, JavaScript code.

So what made me come back to CoffeeScript? Some six months after I had tried out Coffee-Script for the first time, it was announced[10] that Rails 3.1 would ship with CoffeeScript as the default JavaScript engine. Like most developers I was taken aback by this. I had tried Coffee-Script and didn't think it was that great. What were they thinking?

Unlike a lot of my fellow developers, I took the time to have another look at CoffeeScript. Six months is a very long time in the development of any project. CoffeeScript had come a long, long way. I decided to try it again, this time on an application that had some pretty heavy JavaScript. Within a few days of using CoffeeScript again, I became not just a convert but an advocate of the language.

I'm not going to tell you exactly what it was that converted me, or try to tell you why I love it. I want you to form your own opinion. Over the course of this book I hope to both convert you and make you an advocate of this wonderful little language for reasons that are all your own. But I will give you a little sneak peak at what's to come. Here's a bit of CoffeeScript, from an actual application, followed by its equivalent JavaScript. Enjoy!

Example: **(source: sneak_peak.coffee)**

```
@updateAvatars = ->
  names = $('.avatar[data-name]').map -> $(this).data('name')
  Utils.findAvatar(name) for name in $.unique(names)
```

Example: **(source: sneak_peak.js)**

```
(function() {

  this.updateAvatars = function() {
    var name, names, _i, _len, _ref, _results;
    names = $('.avatar[data-name]').map(function() {
      return $(this).data('name');
    });
    _ref = $.unique(names);
    _results = [];
    for (_i = 0, _len = _ref.length; _i < _len; _i++) {
      name = _ref[_i];
      _results.push(Utils.findAvatar(name));
    }
    return _results;
  };

}).call(this);
```

What Is CoffeeScript?

CoffeeScript is a language that compiles down to JavaScript. Not very informative, I know, but it's what it does. CoffeeScript was developed to closely resemble languages such as Ruby[11] and Python.[12] It was designed to help developers write their JavaScript more efficiently. By removing unnecessary punctuation like braces, semicolons, and so on, and by using significant whitespace to replace those characters, you can quickly focus on the code at hand—and not on making sure you have all your curly braces closed.

Chances are you would write the following JavaScript like this:

Example: **(source: punctuation.js)**

```
(function() {

  if (something === something_else) {
    console.log('do something');
  } else {
    console.log('do something else');
  }

}).call(this);
```

So why not write it like this:

Example: **(source: punctuation.coffee)**

```
if something is something_else
  console.log 'do something'
else
  console.log 'do something else'
```

CoffeeScript also gives you several shortcuts to write rather complicated sections of JavaScript with just a short code snippet. Take, for example, this code that lets you loop through the values in an array, without worrying about their indices:

Example: **(source: array.coffee)**

```
for name in array
  console.log name
```

Example: **(source: array.js)**

```
(function() {
  var name, _i, _len;

  for (_i = 0, _len = array.length; _i < _len; _i++) {
    name = array[_i];
    console.log(name);
  }

}).call(this);
```

In addition to the sugary sweet syntax improvements CoffeeScript gives you, it also helps you write better JavaScript code by doing things such as helping you scope your variables and classes appropriately, making sure you use the appropriate comparison operators, and much more, as you'll see during the course of reading this book.

CoffeeScript, Ruby, and Python often get mentioned together in the same breath, and for good reason. CoffeeScript was directly modeled on the terseness and the simple syntax that these languages have to offer. Because of this, CoffeeScript has a much more modern "feel" than JavaScript does, which was modeled on languages such as Java[13] and C++.[14] Like JavaScript, CoffeeScript can be used in any programming environment. Whether you are writing your application using Ruby, Python, PHP,[15] Java, or .Net,[16] it doesn't matter. The compiled JavaScript will work with them all.

Because CoffeeScript compiles down to JavaScript, you can still use any and all of the JavaScript libraries you currently use. You can use jQuery,[17] Zepto,[18] Backbone,[19] Jasmine,[20] and the like, and they'll all just work. You don't hear that too often when talking about new languages.

This all sounds great, I hear you saying, but what are the downsides of using CoffeeScript over just plain old JavaScript? This is a great question. The answer is, not much. First, although CoffeeScript is a really nice way to write your JavaScript, it does not let you do anything you couldn't already do with JavaScript. I still can't, for example, create a JavaScript version of Ruby's famous *method_missing*.[21] The biggest downside would have to be that it's another language for you or the members of your team to learn. Fortunately, this is easily rectified. As you'll see, CoffeeScript is incredibly easy to learn.

Finally, should CoffeeScript, for whatever reason, not be right for you or your project, you can take the generated JavaScript and work from there. So really, you have nothing to lose by giving CoffeeScript a try in your next project, or even in your current project (CoffeeScript and JavaScript play very well with each other).

Who Is This Book For?

This book is for intermediate- to advanced-level JavaScript developers. There are several reasons why I don't think this book is good for those unfamiliar with JavaScript, or for those who only have a passing acquaintance.

First, this book is not going to teach you about JavaScript. This is a book about CoffeeScript. Along the way you are certainly going to learn a few bits and bobs about JavaScript (and CoffeeScript has a knack for making you learn more about JavaScript), but we are not going to start at the beginning of JavaScript and work our way up.

Example: **What does this code do? (source: example.js)**

```
(function() {
  var array, index, _i, _len;

  array = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10];

  for (_i = 0, _len = array.length; _i < _len; _i++) {
    index = array[_i];
    console.log(index);
  }

}).call(this);
```

If you don't know what the preceding code example does, I recommend that you stop reading here. Don't worry, I really want you to come back and keep reading. I just think that you will get the most out of this book if you already have a good understanding of JavaScript. I will be covering certain basic areas of JavaScript as we go along, usually to help illustrate a point or help you to better understand what CoffeeScript is doing. Despite covering certain areas of JavaScript for clarity, it really is important that you have a fundamental grasp of JavaScript before we continue. So please, go find a good book on JavaScript (there are plenty out there), read it, and then join me along our journey to become CoffeeScript gurus.

For those of you who are already JavaScript rock stars, let's step up your game. This book is going to teach you how to write cleaner, more succinct, and better JavaScript using the sweet sugary goodness that is CoffeeScript.

How to Read This Book

I have to tried to present the material in this book to help you form building blocks to learning CoffeeScript. The chapters, in Part I, should be read in order because each chapter will build on the concepts that we have learned in previous chapters—so please, no jumping around.

As we go through each chapter, you'll notice a few things at work.

First, whenever I introduce some outside library, idea, or concept, I include a footnote to a website where you can learn further information about that subject. Although I would love to be able to talk your ear off about things like Ruby, there is not enough space in this book to do that. So if I mention something and you want to find out more about it before continuing, go to the bookmarked site, quench your thirst for knowledge, and come back to the book.

Second, as we go through each chapter I will sometimes walk you through the wrong solution to a problem first. After you see the wrong way to do something, we can then examine it, understand it, and then work out the correct solution to the problem at hand. A great example of this is in Chapter 1, “Getting Started,” when we talk about the different ways to compile your CoffeeScript to JavaScript.

At some points in the book you will come across something like this:

Tip Some helpful tip here.

These are little tips and tricks that I think might be of value to you.

Finally, throughout the book I will present you with two or three code blocks at a time. I will first give you the CoffeeScript example followed by the compiled (JavaScript) version of the same example. If there is any output from the example (and if I think it’s worth showing) I will include the output from the example, as well. Here’s what that looks like:

Example: **(source: example.coffee)**

```
array = [1..10]

for index in array
  console.log index
```

Example: **(source: example.js)**

```
(function() {
  var array, index, _i, _len;

  array = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10];

  for (_i = 0, _len = array.length; _i < _len; _i++) {
    index = array[_i];
    console.log(index);
  }

}).call(this);
```

Output: **(source: example.coffee)**

```
1
2
3
4
5
6
7
8
9
10
```

Sometimes there are errors that I want to show you. Here is an example:

Example: **(source: oops.coffee)**

```
array = [1..10]

oops! index in array
  console.log index
```

Output: **(source: oops.coffee)**

```
Error: In content/preface/oops.coffee, Parse error on line 3: Unexpected 'UNARY'
    at Object.parseError (/usr/local/lib/node_modules/coffee-script/lib/coffee-script/
➥parser.js:470:11)
    at Object.parse (/usr/local/lib/node_modules/coffee-script/lib/coffee-script/
➥parser.js:546:22)
    at /usr/local/lib/node_modules/coffee-script/lib/coffee-script/coffee-script.
➥js:40:22
    at Object.run (/usr/local/lib/node_modules/coffee-script/lib/coffee-script/
➥coffee-script.js:68:34)
    at /usr/local/lib/node_modules/coffee-script/lib/coffee-script/command.js:135:29
    at /usr/local/lib/node_modules/coffee-script/lib/coffee-script/command.js:110:18
    at [object Object].<anonymous> (fs.js:114:5)
    at [object Object].emit (events.js:64:17)
    at afterRead (fs.js:1081:12)
    at Object.wrapper [as oncomplete] (fs.js:252:17)
```

How This Book Is Organized

In an effort to help you get the most from this book, I have split it into two distinct parts.

Part I: Core CoffeeScript

The first part of this book is designed to cover the entire CoffeeScript language from top to bottom. By the end of this part of the book, you should be fully armed to attack any CoffeeScript project that comes your way, including those in the second part of this book.

Chapter 1, “Getting Started,” introduces the various ways CoffeeScript can be compiled and run. It also introduces the powerful coffee command-line utility and REPL that ships with CoffeeScript.

In Chapter 2, “The Basics,” we start to explore what makes CoffeeScript different from JavaScript. Talk of syntax, variables, scope, and more will lay a strong foundation for the rest of the book.

Chapter 3, “Control Structures,” focuses on an important part of any language, control structures such as `if` and `else`. You will also learn the differences between some operators in CoffeeScript and those in JavaScript.

Chapter 4, “Functions and Arguments,” covers the ins and outs of functions in CoffeeScript. We’ll talk about defining functions, calling functions, and a few extras such as default arguments and splats.

From arrays to objects, Chapter 5, “Collections and Iterations,” shows you how to use, manipulate, and iterate over collection objects in CoffeeScript.

Chapter 6, “Classes,” ends the first part of the book by covering classes in CoffeeScript. Define new classes, extend existing classes, override functions in super classes, and more.

Part II: CoffeeScript in Practice

The second part of this book focuses on using CoffeeScript in practical examples. Through learning about some of the ecosystem that surrounds CoffeeScript, as well as building a full application, by the end of Part II your CoffeeScript skills should be well honed.

Chapter 7, “Cake and Cakefiles,” covers the Cake tool that ships with CoffeeScript. You can use this little tool for creating build scripts, test scripts, and more. We’ll cover all that it has to offer.

Testing is a very important part of software development, and Chapter 8, “Testing with Jasmine,” gives a quick tour through one of the more popular CoffeeScript/JavaScript testing libraries, Jasmine. This chapter will exercise the popular pattern of test-driven development by writing tests first for a calculator class.

Chapter 9, “Intro to Node.js,” is a quick introduction to the event-driven server-side framework, Node.js. In this chapter we will use CoffeeScript to build a simple HTTP server that will automatically compile CoffeeScript files into JavaScript files as they are requested by the web browser.

In Chapter 10, “Example: Todo List Part 1 (Server-side),” we will be building the server-side part of a todo list application. Building on Chapter 9, we will build an API using the Express.js web framework and the Mongoose ORM for MongoDB.

In Chapter 11, “Example: Todo List Part 2 (Client-side w/ jQuery),” we will build a client for the todo list API we built in Chapter 10 using the popular jQuery libary.

Finally, in Chapter 12, “Example: Todo List Part 3 (Client-side w/ Backbone.js),” we will rebuild the client for the todo list application, this time forsaking jQuery in favor of the client-side framework, Backbone.js.

Installing CoffeeScript

I am not a big fan of having installation instructions in books, mostly because by the time the book hits the shelf, the installation instructions are out of date. However, people—and by people, I mean those who publish books—believe that there should be an installation section for books. So this is mine.

Installing CoffeeScript is pretty easy. The easiest way to install it is to go to http://www.coffeescript.org and look at the installation instructions there.

I believe the maintainers of projects like CoffeeScript and Node[22] are the best people to keep the installation instructions up to date for their projects, and their websites are a great place to find those instructions.

At the time of writing, CoffeeScript was at version: 1.2.0. All examples in this book should work on that version.

How to Run the Examples

You will be able to find and download all the original source code for this book at https://github.com/markbates/Programming-In-CoffeeScript. As you’ll see, all the examples tell you which example file to look to. The example files will be in a folder relevant to their respective chapter.

Unless otherwise indicated, you should be able to run the examples in your terminal, like so:

```
> coffee example.coffee
```

So now that you know how to run the examples in this book, as soon as you have CoffeeScript installed, why don’t you meet me at Chapter 1 and we can get started? See you there.

Notes

1. http://en.wikipedia.org/wiki/JavaScript
2. http://en.wikipedia.org/wiki/Ajax_(programming)
3. http://www.prototypejs.org/
4. http://documentcloud.github.com/backbone/
5. http://en.wikipedia.org/wiki/Model–view–controller
6. http://www.adobe.com/
7. http://www.apple.com/ios/
8. http://www.coffeescript.org
9. http://www.rubyonrails.org
10. http://www.rubyinside.com/rails-3-1-adopts-coffeescript-jquery-sass-and-controversy-4669.html
11. http://en.wikipedia.org/wiki/Ruby_(programming_language)
12. http://en.wikipedia.org/wiki/Python_(programming_language)
13. http://en.wikipedia.org/wiki/Java_(programming_language)
14. http://en.wikipedia.org/wiki/C%2B%2B
15. http://en.wikipedia.org/wiki/Php
16. http://en.wikipedia.org/wiki/.NET_Framework
17. http://www.jquery.com
18. https://github.com/madrobby/zepto
19. http://documentcloud.github.com/backbone
20. http://pivotal.github.com/jasmine/
21. http://ruby-doc.org/docs/ProgrammingRuby/html/ref_c_object.html#Object.method_missing
22. http://nodejs.org

4 Functions and Arguments

In this chapter we are going to look at one of the most essential parts of any language, the function. Functions allow us to encapsulate reusable and discrete code blocks. Without functions our code would be one long, unreadable, and unmaintainable mess.

I wanted to give you an example of what JavaScript would look like if we were not able to use or write functions, but I was unable to. Even the simplest example of taking a string and making it lowercase requires the use of functions in JavaScript.

Because I can't show you an example devoid of functions, I'll show you an example of some CoffeeScript code that could use the help of a function or two, so you can see how important functions are to helping you keep your code manageable.

Example: **(source: no_functions_example.coffee)**

```
tax_rate = 0.0625

val = 100
console.log "What is the total of $#{val} worth of shopping?"
tax = val * tax_rate
total = val + tax
console.log "The total is #{total}"

val = 200
console.log "What is the total of $#{val} worth of shopping?"
tax = val * tax_rate
total = val + tax
console.log "The total is #{total}"
```

Example: **(source: no_functions_example.js)**

```
(function() {
  var tax, tax_rate, total, val;

  tax_rate = 0.0625;

  val = 100;

  console.log("What is the total of $" + val + " worth of shopping?");

  tax = val * tax_rate;

  total = val + tax;

  console.log("The total is " + total);

  val = 200;

  console.log("What is the total of $" + val + " worth of shopping?");

  tax = val * tax_rate;

  total = val + tax;

  console.log("The total is " + total);

}).call(this);
```

Output: **(source: no_functions_example.coffee)**

```
What is the total of $100 worth of shopping?
The total is 106.25
What is the total of $200 worth of shopping?
The total is 212.5
```

In our example, we are calculating the total value of goods purchased in-state with certain sales tax. Apart from the banality of the example, you can see that we are repeating our code to calculate the total value with tax several times.

Let's refactor our code a bit, add some functions, and try to clean it up.

Example: **(source: with_functions_example.coffee)**

```
default_tax_rate = 0.0625

calculateTotal = (sub_total, rate = default_tax_rate) ->
  tax = sub_total * rate
  sub_total + tax

val = 100
console.log "What is the total of $#{val} worth of shopping?"
console.log "The total is #{calculateTotal(val)}"

val = 200
console.log "What is the total of $#{val} worth of shopping?"
console.log "The total is #{calculateTotal(val)}"
```

Example: **(source: with_functions_example.js)**

```
(function() {
  var calculateTotal, default_tax_rate, val;

  default_tax_rate = 0.0625;

  calculateTotal = function(sub_total, rate) {
    var tax;
    if (rate == null) rate = default_tax_rate;
    tax = sub_total * rate;
    return sub_total + tax;
  };

  val = 100;

  console.log("What is the total of $" + val + " worth of shopping?");

  console.log("The total is " + (calculateTotal(val)));

  val = 200;

  console.log("What is the total of $" + val + " worth of shopping?");

  console.log("The total is " + (calculateTotal(val)));

}).call(this);
```

Output: **(source: with_functions_example.coffee)**

```
What is the total of $100 worth of shopping?
The total is 106.25
What is the total of $200 worth of shopping?
The total is 212.5
```

You probably don't understand everything we just did there, but don't worry, that's what this chapter is for. However, even without knowing the specifics of how functions are defined, and work, in CoffeeScript, you can see how much cleaner our code is between the two examples. In the refactored code, we are even able to pass in a different tax rate, should we need to. This also helps us keep our code DRY[1]: Don't Repeat Yourself. Not repeating your code makes for an easier-to-manage code base with, hopefully, fewer bugs.

Function Basics

We'll start with the very basics on how to define a function in CoffeeScript. The anatomy of a very simple function looks like this:

Example: **(source: simple_function.coffee)**

```
myFunction = ()->
  console.log "do some work here"

myFunction()
```

Example: **(source: simple_function.js)**

```
(function() {
  var myFunction;

  myFunction = function() {
    return console.log("do some work here");
  };

  myFunction();

}).call(this);
```

In that example we gave the function a name, `myFunction`, and a code block to go with it. The body of the function is the code that is indented below the `->`, following the significant whitespace rules we learned about in Chapter 2, "The Basics."

The function does not accept any arguments. We know that by the empty parentheses prior to the `->`. When calling a function in CoffeeScript that has no arguments, we are required to use parentheses, `myFunction()`.

Because our function has no arguments, we can drop the parentheses entirely when defining it, like so:

Example: **(source: simple_function_no_parens.coffee)**

```
myFunction = ->
  console.log "do some work here"

myFunction()
```

Example: **(source: simple_function_no_parens.js)**

```
(function() {
  var myFunction;

  myFunction = function() {
    return console.log("do some work here");
  };

  myFunction();

}).call(this);
```

There is one more way we can write this simple function. Because the body of our function is on only one line, we can collapse the whole function definition to one, like this:

Example: **(source: simple_function_one_line.coffee)**

```
myFunction = -> console.log "do some work here"

myFunction()
```

Example: **(source: simple_function_one_line.js)**

```
(function() {
  var myFunction;

  myFunction = function() {
    return console.log("do some work here");
  };

  myFunction();

}).call(this);
```

All three of the previous code examples produce the same JavaScript and are called in the same way.

Tip

Although you can write function definitions on one line, I prefer not to. Personally, I don't find it that much cleaner or easier to read. Also, by keeping the body of the function on a separate line, you make it easier to later augment your function with more code.

You should also notice that the last line of each function contains a `return` keyword. CoffeeScript adds this automatically for you. Whatever the last line of your function is, that will be the function's return value. This is similar to languages such as Ruby. Because CoffeeScript will automatically add the `return` for you in the compiled JavaScript, the use of the `return` keyword in your CoffeeScript is optional.

Tip

I find that adding the `return` keyword can sometimes help make the meaning of your code a bit clearer. Use it where you find it will help make your code easier to read and understand.

Tip

If you want your functions to not return the last line of the function, you'll have to explicitly give it a new last line to return. Something like `return null` or `return undefined` will do the trick nicely.

Arguments

Just like in JavaScript, functions in CoffeeScript can also take arguments. Arguments let us pass objects into the function so that the function can then perform calculations, data manipulation, or whatever our little hearts desire.

In CoffeeScript, defining a function that takes arguments is not much different than in JavaScript. Inside our parentheses we define a comma-separated list of the names of the arguments we want the function to accept.

Example: **(source: function_with_args.coffee)**

```
calculateTotal = (sub_total, rate) ->
  tax = sub_total * rate
  sub_total + tax

console.log calculateTotal(100, 0.0625)
```

Example: **(source: function_with_args.js)**

```
(function() {
  var calculateTotal;

  calculateTotal = function(sub_total, rate) {
    var tax;
    tax = sub_total * rate;
    return sub_total + tax;
  };

  console.log(calculateTotal(100, 0.0625));

}).call(this);
```

Output: **(source: function_with_args.coffee)**

```
106.25
```

As you can see in our example, we defined our function to take in two arguments and to do some math with them to calculate a total value. When we called the function, we passed in the two values we wanted it to use.

In Chapter 2 we discussed briefly the rules around parentheses in CoffeeScript. I want to reiterate one of those rules. Because our function takes arguments, we are allowed to omit the parentheses when calling the function. This means we could also write our example like this:

Example: **(source: function_with_args_no_parens.coffee)**

```
calculateTotal = (sub_total, rate) ->
  tax = sub_total * rate
  sub_total + tax

console.log calculateTotal 100, 0.0625
```

Example: **(source: function_with_args_no_parens.js)**

```
(function() {
  var calculateTotal;

  calculateTotal = function(sub_total, rate) {
    var tax;
    tax = sub_total * rate;
    return sub_total + tax;
  };
  console.log(calculateTotal(100, 0.0625));

}).call(this);
```

Output: **(source: function_with_args_no_parens.coffee)**

```
106.25
```

As you can see, CoffeeScript correctly compiled the JavaScript for us, putting those parentheses back where they are needed.

Tip

The use of parentheses when calling functions is hotly contested in the CoffeeScript world. Personally, I tend to use them. I think it helps make my code a bit more readable, and it cuts down on potential bugs where parentheses were misplaced by the compiler. When in doubt, use parentheses. You won't regret it.

Default Arguments

In some languages, such as Ruby, it is possible to assign default values to arguments. This means that if you do not pass in some arguments, for whatever reason, then reasonable default values can be used in their place.

Let's revisit our calculator example again. We'll write it so that the tax rate is set to a default value should one not be passed in:

Example: **(source: default_args.coffee)**

```
calculateTotal = (sub_total, rate = 0.05) ->
  tax = sub_total * rate
  sub_total + tax

console.log calculateTotal 100, 0.0625
console.log calculateTotal 100
```

Example: **(source: default_args.js)**

```
(function() {
  var calculateTotal;

  calculateTotal = function(sub_total, rate) {
    var tax;
    if (rate == null) rate = 0.05;
    tax = sub_total * rate;
    return sub_total + tax;
  };

  console.log(calculateTotal(100, 0.0625));

  console.log(calculateTotal(100));

}).call(this);
```

Output: **(source: default_args.coffee)**

```
106.25
105
```

When defining our function, we told CoffeeScript to set the default value of the `tax_rate` argument equal to `0.05`. When we first call the `calculateTotal` function, we pass in a `tax_rate` argument of `0.0625`; the second time we omit the `tax_rate` argument altogether, and the code does the appropriate thing and uses `0.05` in its place.

We can take default arguments a step further and have them refer to other arguments. Consider this example:

Example: **(source: default_args_referring.coffee)**

```
href = (text, url = text) ->
  html = "<a href='#{url}'>#{text}</a>"
  return html

console.log href("Click Here", "http://www.example.com")
console.log href("http://www.example.com")
```

Example: **(source: default_args_referring.js)**

```
(function() {
  var href;

  href = function(text, url) {
    var html;
    if (url == null) url = text;
    html = "<a href='" + url + "'>" + text + "</a>";
    return html;
  };
  console.log(href("Click Here", "http://www.example.com"));

  console.log(href("http://www.example.com"));

}).call(this);
```

Output: **(source: default_args_referring.coffee)**

```
<a href='http://www.example.com'>Click Here</a>
<a href='http://www.example.com'>http://www.example.com</a>
```

Should no one pass in the `url` argument in our example, we will set it equal to the `text` argument that was passed in.

It is also possible to use functions as default values in the argument list. Because the default value will be called only if there is no argument passed in, there is no performance concern.

Example: **(source: default_args_with_function.coffee)**

```
defaultRate = -> 0.05

calculateTotal = (sub_total, rate = defaultRate()) ->
  tax = sub_total * rate
  sub_total + tax

console.log calculateTotal 100, 0.0625
console.log calculateTotal 100
```

Example: **(source: default_args_with_function.js)**

```
(function() {
  var calculateTotal, defaultRate;

  defaultRate = function() {
    return 0.05;
  };

  calculateTotal = function(sub_total, rate) {
    var tax;
    if (rate == null) rate = defaultRate();
    tax = sub_total * rate;
    return sub_total + tax;
  };
  console.log(calculateTotal(100, 0.0625));

  console.log(calculateTotal(100));

}).call(this);
```

Output: **(source: default_args_with_function.coffee)**

```
106.25
105
```

Tip

When using default arguments it is important to note that they must be at the *end* of the argument list. It is okay to have multiple arguments with defaults, but they all must be at the end.

Splats...

Sometimes when developing a function, we are not sure just how many arguments we are going to need. Sometimes we might get one argument; other times we might get a hundred. To help us easily solve this problem, CoffeeScript gives us the option of using splats when defining the argument list for a function. Splatted arguments are denoted by placing an ellipsis (`...`) after the method definition.

Tip

A great way to remember how to use splats is to treat the `...` suffix as if you were saying etc... Not only is that easy to remember, but if you use `etc...` in your code, you'll look cool.

When would you use splats? Splats can be used whenever your function will be taking in a variable number of arguments. Before we take a look at a detailed example, let's look quickly at a simple function that takes a splatted argument:

Example: **(source: splats.coffee)**

```
splatter = (etc...) ->
  console.log "Length: #{etc.length}, Values: #{etc.join(', ')}"

splatter()
splatter("a", "b", "c")
```

Example: **(source: splats.js)**

```
(function() {
  var splatter,
    __slice = Array.prototype.slice;

  splatter = function() {
    var etc;
    etc = 1 <= arguments.length ? __slice.call(arguments, 0) : [];
    return console.log("Length: " + etc.length + ", Values: " + (etc.join(', ')));
  };

  splatter();

  splatter("a", "b", "c");

}).call(this);
```

Output: **(source: splats.coffee)**

```
Length: 0, Values:
Length: 3, Values: a, b, c
```

As you can see, whatever arguments we pass into our function automatically get put into an array, and should we not send any arguments we get an empty array.

Tip

Splats are a great example of something that can be done in JavaScript but would require a lot of boilerplate code to implement. Look at the JavaScript output of a CoffeeScript splatted argument and you'll agree boilerplate code is no fun to write.

Unlike other languages that support a similar construct, CoffeeScript does not force you to only use splats as the last argument in the argument list. In fact, splatted arguments can appear anywhere in your argument list. A small caveat is that you can have only one splatted argument in the argument list.

To help illustrate how splats can be used in any part of the argument list, let's write a method that will take some arguments and spit out a string. When building this string, we make sure that the first and last arguments are uppercased; any other arguments will be lowercased. Then we'll concatenate the string using forward slashes.

Example: **(source: splats_arg_join.coffee)**

```
joinArgs = (first, middles..., last) ->
  parts = []

  if first?
    parts.push first.toUpperCase()

  for middle in middles
    parts.push middle.toLowerCase()

  if last?
    parts.push last.toUpperCase()

  parts.join('/')

console.log joinArgs("a")
console.log joinArgs("a", "b")
console.log joinArgs("a", "B", "C", "d")
```

Example: **(source: splats_arg_join.js)**

```
(function() {
  var joinArgs,
    __slice = Array.prototype.slice;

  joinArgs = function() {
    var first, last, middle, middles, parts, _i, _j, _len;
    first = arguments[0], middles = 3 <= arguments.length ? __slice.call(arguments, 1,
➥_i = arguments.length - 1) : (_i = 1, []), last = arguments[_i++];
    parts = [];
    if (first != null) parts.push(first.toUpperCase());
    for (_j = 0, _len = middles.length; _j < _len; _j++) {
      middle = middles[_j];
      parts.push(middle.toLowerCase());
    }
    if (last != null) parts.push(last.toUpperCase());
    return parts.join('/');
  };

  console.log(joinArgs("a"));

  console.log(joinArgs("a", "b"));

  console.log(joinArgs("a", "B", "C", "d"));

}).call(this);
```

Output: **(source: splats_arg_join.coffee)**

```
A
A/B
A/b/c/D
```

I admit that is a bit of a heavy example, but it illustrates how splats work. When we call the `joinArgs` function, the first argument we pass into the function call gets assigned to the `first` variable, the last argument we pass in gets assigned to the `last` variable, and if any other arguments are passed in between the first and the last arguments, those are put into an array and assigned to the `middles` variable.

Tip

We could have written our function to just take a splatted argument and extract the first and last elements from the `middles` array, but this function definition means we don't have to write all that code. Happy days.

Finally, when dealing with splats, you might have an array that you want passed in as individual arguments. That is possible.

Let's take a quick look at an example:

Example: **(source: splats_array.coffee)**

```
splatter = (etc...) ->
  console.log "Length: #{etc.length}, Values: #{etc.join(', ')}"

a = ["a", "b", "c"]
splatter(a)
splatter(a...)
```

Example: **(source: splats_array.js)**

```
(function() {
  var a, splatter,
    __slice = Array.prototype.slice;

  splatter = function() {
    var etc;
    etc = 1 <= arguments.length ? __slice.call(arguments, 0) : [];
    return console.log("Length: " + etc.length + ", Values: " + (etc.join(', ')));
  };

  a = ["a", "b", "c"];
  splatter(a);

  splatter.apply(null, a);

}).call(this);
```

Output: **(source: splats_array.coffee)**

```
Length: 1, Values: a,b,c
Length: 3, Values: a, b, c
```

Using our earlier `splatter` example, we can try first passing in an array, but as you can see, the `splatter` function sees the array as a single argument, because that is what it is. However, if we append `...` to the array as we pass it into our function call, the CoffeeScript will split up the array into separate arguments and pass them into the function.

Wrapping Up

There you have it—everything you've ever wanted to know about functions in CoffeeScript! First we looked at how to define a simple function; in fact, we saw several ways to define a function in CoffeeScript. We then took a look at how arguments to functions are defined and how to call a function, including a recap of when and where you do and do not have to use parentheses when calling a function. We also took a look at default arguments, one of my favorite features of CoffeeScript.

Finally, we explored splats and how they help us write functions that take variable arguments.

With our nickel tour of functions and arguments over with, we can move on to the next stop, Chapter 5, "Collections and Iterations." So go grab a cold one, and we'll meet there. Ready?

Notes

1. http://en.wikipedia.org/wiki/DRY

Index