CoffeeScript

Accelerated JavaScript Development

Trevor Burnham

Foreword by Jeremy Ashkenas

edited by Michael Swaine

What readers are saying about
CoffeeScript: Accelerated JavaScript Development

It's hard to imagine a new web application today that doesn't make heavy use of JavaScript, but if you're used to something like Ruby, it feels like a significant step down to deal with JavaScript, more of a chore than a joy. Enter CoffeeScript: a pre-compiler that removes all the unnecessary verbosity of JavaScript and simply makes it a pleasure to write and read. Go, go, Coffee! This book is a great introduction to the world of CoffeeScript.

➤ **David Heinemeier Hansson**
   Creator, Rails

Just like CoffeeScript itself, Trevor gets straight to the point and shows you the benefits of CoffeeScript and how to write concise, clear CoffeeScript code.

➤ **Scott Leberknight**
   Chief Architect, Near Infinity

Though CoffeeScript is a new language, you can already find it almost everywhere. This book will show you just how powerful and fun CoffeeScript can be.

➤ **Stan Angeloff**
   Managing Director, PSP WebTech Bulgaria

This book helps readers become better JavaScripters in the process of learning CoffeeScript. What's more, it's a blast to read, especially if you are new to Coffee-Script and ready to learn.

➤ **Brendan Eich**
   Creator, JavaScript

CoffeeScript may turn out to be one of the great innovations in web application development; since I first discovered it, I've never had to write a line of pure JavaScript. I hope the readers of this wonderful book will be able to say the same.

➤ **Dr. Nic Williams**
   CEO/Founder, Mocra

CoffeeScript: Accelerated JavaScript Development is an excellent guide to Coffee-Script from one of the community's most esteemed members. It'll help you get up to speed with the language in no time, whether you write code that runs in the browser or on the server. Trevor's book belongs on every CoffeeScript developer's shelf.

➤ **Sam Stephenson**
   Creator, Prototype JavaScript framework

CoffeeScript is one of the most interesting developments in the world of programming languages in the last few years. Taking the lessons learned over the last decade from languages like Ruby and Python, it is a language with immense expressive power. *CoffeeScript: Accelerated JavaScript Development* is your guide to this new language and a must-read for those interested in being productive in JavaScript.

➤ **Travis Swicegood**
   Author, *Pragmatic Version Control Using Git*

Trevor serves up a rich blend of language overview and real-world examples, showcasing why I consider CoffeeScript my secret weapon for iOS, Android, and WebOS mobile development.

➤ **Wynn Netherland**
   Co-host, The Changelog

Fasten your seat belt and enjoy the ride with Trevor Burnham from JavaScript to CoffeeScript and have fun with web development again.

➤ **Javier Collado**
   QA Automation Engineer, Canonical Ltd.

CoffeeScript

Accelerated JavaScript Development

Trevor Burnham

Pragmatic Bookshelf

Many of the designations used by manufacturers and sellers to distinguish their products are claimed as trademarks. Where those designations appear in this book, and The Pragmatic Programmers, LLC was aware of a trademark claim, the designations have been printed in initial capital letters or in all capitals. The Pragmatic Starter Kit, The Pragmatic Programmer, Pragmatic Programming, Pragmatic Bookshelf, PragProg and the linking *g* device are trademarks of The Pragmatic Programmers, LLC.

Every precaution was taken in the preparation of this book. However, the publisher assumes no responsibility for errors or omissions, or for damages that may result from the use of information (including program listings) contained herein.

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)

Contents

Foreword

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

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

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

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

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

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

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

Jeremy Ashkenas, creator of CoffeeScript
April 2011

Acknowledgments

CoffeeScript is a young language. But from the start, it's drawn an exceptionally diverse and spirited crowd. That wonderful energy—on IRC, GitHub, Hacker News, blogs, Twitter, and elsewhere—is what inspired me to write this book. To everyone who greeted CoffeeScript with enthusiasm in its infancy, I thank you.

Thanks, of course, to Jeremy Ashkenas for creating the language and contributing a generous foreword to this book; CoffeeScript could not have asked for a better BDFL. Thanks also to CoffeeScript's other contributors, who are too numerous to name here.[1]

Thanks to the technical reviewers—any remaining errors are completely and utterly "my bad." I received helpful feedback from Javier Collado, Kevin Gisi, Darcy Laycock, Scott Leberknight, Sam Stephenson, Travis Swicegood, Federico Tomassetti, Stefan Turalski, and Dr. Nic Williams. Special shout-outs to Jeremy Ashkenas (again) and Michael Ficarra, core contributors to the CoffeeScript project who took time from their busy schedules to set me straight on many of the language's finer points. Thanks also to Brendan Eich, the creator of JavaScript, who graciously clarified several points.

Thanks to the Pragmatic Bookshelf crowd. First and foremost to Michael Swaine, whom I'm proud to call my editor. Thanks also to managing editor Susannah Pfalzer and to bigwigs Dave Thomas and Andy Hunt for taking a chance on a book on a lesser-known language from an even less-known author.

Thanks, finally, to Scott and Teresa Burnham, more commonly referred to by me and at least two other people as "Dad" and "Mom." Their support, and their example, has been valuable beyond measure.

---

1. http://github.com/jashkenas/coffee-script/contributors

Preface

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

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

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

That is, until now.

The New Kid in Town

On Christmas Day 2009, Jeremy Ashkenas first released CoffeeScript, a little language he touted as "JavaScript's less ostentatious kid brother." The project quickly attracted hundreds of followers on GitHub as Ashkenas and

---

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

other contributors added a bevy of new features each month. The language's compiler, originally written in Ruby, was replaced in March 2010 by one written in CoffeeScript.

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

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

The Good Parts Are Still There

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

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

The Compiler Is Here to Help

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

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

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

CoffeeScript can't cure all of these ills, but the compiler tries its best to generate JavaScript Lint-compliant output[3], which is a great filter for common human errors and nonstandard idioms. And if you type something that

---

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

just doesn't make any sense, such as 2 = 3, the CoffeeScript compiler will tell you. Better to find out sooner than later.

It's All So Clear Now

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

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

Now here's an equivalent snippet of CoffeeScript:

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

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

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

It's worth adding that switching to CoffeeScript isn't an all-or-nothing proposition—CoffeeScript code and JavaScript code can interact freely. CoffeeScript's strings are just JavaScript strings, and its numbers are just JavaScript numbers; even its classes work in JavaScript frameworks like Backbone.js.[5] So don't be afraid of calling JavaScript code from CoffeeScript code or vice versa. As an example, we'll talk about using CoffeeScript with one of JavaScript's most popular libraries in Chapter 5, *Web Interactivity with jQuery,* on page 75.

---

4. http://www.paulgraham.com/power.html
5. http://documentcloud.github.com/backbone/

> ### Embedding JavaScript in CoffeeScript
>
> This is as good a place as any to mention that you can stick JavaScript inside of CoffeeScript code by surrounding it with backticks, like so:
>
> ```
> console.log `impatient ? useBackticks() : learnCoffeeScript()`
> ```
>
> The CoffeeScript compiler simply ignores everything between the backticks. That means that if, for instance, you declare a variable between the backticks, that variable won't obey conventional CoffeeScript scope rules.
>
> In all my time writing CoffeeScript, I've never once needed to use backtick escapes. They're an eyesore at best and dangerous at worst. So in the immortal words of Troy McClure: "Now that you know how it's done—don't do it."

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

Who This Book Is For

If you're interested in learning CoffeeScript, you've come to the right place! However, because CoffeeScript is so closely linked to JavaScript, there are really two languages running through this book—and not enough pages to teach you both. Therefore, I'm going to assume that you know *some* JavaScript.

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

You may notice that I talk about Ruby a lot in this book. Ruby inspired many of CoffeeScript's great features, like implicit returns, splats, and postfix if/unless. And thanks to Rails 3.1, CoffeeScript has a huge following

---

6. http://www.codinghorror.com/blog/2009/07/meta-is-murder.html
7. http://eloquentjavascript.net/
8. http://javascriptgarden.info/
9. https://developer.mozilla.org/en/JavaScript/Guide

in the Ruby world. So if you're a Rubyist, *great!* You've got a head start. If not, don't sweat it; everything will fall into place once you have a few examples under your belt.

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

How This Book Is Organized

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

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

No matter what level you're at, be sure to do the exercises at the end of each chapter. They're designed to be quick yet challenging, illustrating some of the most common pitfalls CoffeeScripters fall into. Try to solve them on your own before you check the answers in Appendix 1, *Answers to Exercises*, on page 107.

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

About the Example Project: 5x5

The last section of each chapter applies the new concepts to an original word game called 5x5. As its name suggests, 5x5 is played on a grid five tiles wide and five tiles high. Each tile has a random letter placed on it at the start. Then the players take turns swapping letters on the grid, scoring points for all words formed as a result of the swap (potentially, this can be

---

10. http://forums.pragprog.com/forums/169
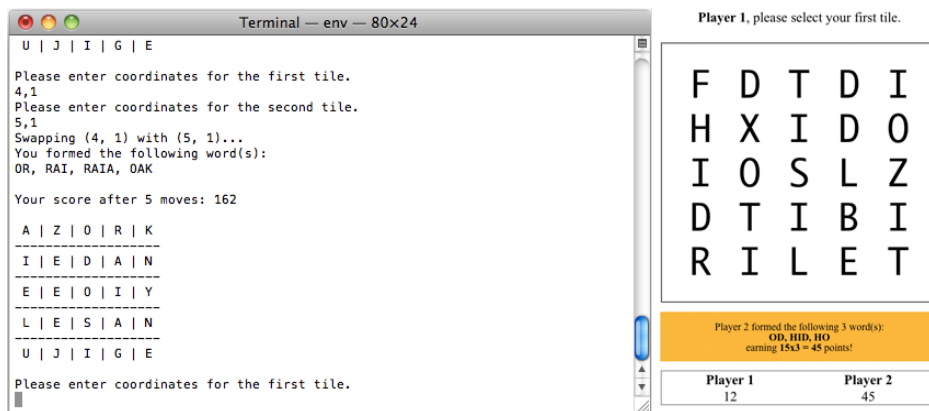
Figure 1—In the console and web versions of our project, the game logic code will be the same.

four words at each of the two swapped tiles: one running horizontally, one vertically, and two diagonally—only left-to-right diagonals count).

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

We'll build a command-line version of the game in Chapters 2–4, then move it to the browser in Chapter 5, *Web Interactivity with jQuery, on page 75*, and finally add multiplayer capability in Chapter 6, *Server-Side Apps with Node.js, on page 91*. Moving the code from the command line to the browser to the server will be super-easy—they all speak the same language!

The CoffeeScript Community

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

Posting a question to StackOverflow (being sure to tag your question coffee-script) is a terrific way to get help, especially if you post a snippet of the code that's hassling you.[11] If you need a more immediate answer, you can usually find friendly folks in the #coffeescript channel on Freenode IRC. For relaxed

_____

11. http://stackoverflow.com

discussion of CoffeeScript miscellany, try the Google Group.[12] For more serious problems, such as possible bugs, you should create an issue on GitHub.[13] You can also request new language features there. CoffeeScript is still evolving, and the whole team welcomes feedback.

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

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

These are exciting times for web development. Welcome aboard!

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

Getting Started

If you read the preface, then you now know what CoffeeScript is, where it came from, and why it's the best thing to happen to programmers since Herman Miller. But you haven't actually written a line of code yet. The wait is unbearable, isn't it?

Well, take a deep breath; the time has come. In this chapter, we're going to install CoffeeScript on your system, get your editor up to speed, and finally run some code!

1.1 Installing CoffeeScript

The CoffeeScript compiler is written in CoffeeScript. That presents a chicken-and-egg problem: How do we run the compiler on a system that doesn't already have the CoffeeScript compiler? If only there were some way to run JavaScript on your machine without a web browser and give that code access to the local file system…

Ah, but there is: Node.js! People think of Node as a JavaScript web server (more on that in Chapter 6, *Server-Side Apps with Node.js*, on page 91), but it's so much more. Fundamentally, it's a bridge between JavaScript code and your operating system. Node also has a wonderful tool called npm, the Node Package Manager.[1] If your background is in Ruby, think of it as the Node analog of RubyGems. It's become *the* de facto standard for installing and managing Node apps and libraries.

The rest of this section will be about installing Node and npm, which we need in order to use CoffeeScript's canonical coffee compiler. (We'll also need Node and npm for the last chapter of this book.) But if you're in a rush to get your feet wet, you might want to head over to http://coffeescript.org/,

---

1.   http://npmjs.org/

hit the "Try CoffeeScript" button, and skip ahead to the next chapter. (You'll need something in your browser to display console output, such as Firebug Lite.[2])

Ready? Let's get to it.

CoffeeScript with Node.js and npm

Although there are many ways to run CoffeeScript without Node (several of which are covered in Appendix 2, *Ways of Running CoffeeScript,* on page 115), I'll assume throughout this book that you're using the standard coffee command, which was designed to run under Node. The final chapter, Chapter 6, *Server-Side Apps with Node.js,* on page 91, is the only one that explicitly requires Node and npm.

Heads up—if you're on Windows, you'll need to get Cygwin before we continue.[3] Cygwin basically acts as a Linux emulator. While first-class Windows support is on the Node.js roadmap for version 0.6, using Cygwin is the most reliable approach available as of this writing.

If you're on a Mac, you'll need to install Xcode,[4] not for the app itself but for the command-line developer tools that come with it. You can check whether these tools are already on your system by trying to run gcc, the GNU Compiler Collection:

```
$ gcc
i686-apple-darwin10-gcc-4.2.1: no input files
```

If your output looked like that, you're set. If not, get Xcode (if you're on a Mac) or install the standard build tools directly (if you're on Linux or Cygwin).

Everyone's on a Linux/Unix/Mac-type system with standard build tools now? Great! Now head to http://gist.github.com/579814. There you'll find a bewildering array of installation options curated by npm creator Isaac Schlueter. For all the Mac users out there, I recommend the Homebrew approach (install Homebrew first).[5] For everyone else, the direct approach—first on the list—is probably best. Node is a big package, so it may take a few minutes to install.

Once Node is on your system, run the latest remote install script for npm:

```
$ curl http://npmjs.org/install.sh | sh
```

---

2. http://getfirebug.com/firebuglite
3. http://www.cygwin.com/
4. http://developer.apple.com/xcode/
5. http://github.com/mxcl/homebrew

If you get a permissions error, either chown the directory Node is installed in (this will save you from headaches down the road) or use sudo sh instead of plain sh.

No matter how you installed them, check that node and npm are on your path: PATH

```
$ node -v
v0.4.8
$ npm -v
1.0.13
```

(A word on versions: Node's API is stable in even-numbered point releases. So, the examples in this book should run fine under the latest 0.4.x. Node 0.5.x, on the other hand, will feature API changes, which will be incorporated into the stable 0.6.x. As to npm, I'll assume throughout this book that you're using npm 1.x. So if you're still on npm 0.x, now would be a good time to upgrade.)

Now grab the latest CoffeeScript release:

```
$ npm install -g coffee-script
/usr/local/bin/cake -> /usr/local/lib/node_modules/coffee-script/bin/cake
/usr/local/bin/coffee -> /usr/local/lib/node_modules/coffee-script/bin/coffee
```

The -g flag, short for --global, makes the installed library available system-wide. (By default, npm install [package] puts the given package in the local node_modules subdirectory, which is handy when installing a package that is only for a specific project.) I recommend using -g whenever you install packages that include binaries.

The output from npm install tells us that two binaries were installed as part of the package: cake and coffee. Let's check that coffee is on our system's PATH:

```
$ coffee -v
CoffeeScript version 1.1.1
```

If that didn't work, look at the directory before -> in your npm install output (for example, /usr/local/bin) and add that directory to your PATH. On a Mac with the default bash shell, do that by adding the following line to your ~/.profile:

```
export PATH=/usr/local/bin:$PATH
```

Make sure to include the :$PATH part—otherwise, /usr/local/bin would replace your PATH rather than being added to it! For the line to take effect, you'll have to save the file and start a new shell session (for example, by opening a new Terminal window and closing the old one).

If you're using a different OS or shell, these steps might be slightly different; enter echo $SHELL to find out which shell you're using. Don't forget that you have to restart your shell session after modifying the file in order for it to take effect.

There's one last step: just as our binaries have to be on PATH for us to be able to use them from anywhere, the Node libraries npm installs have to be on NODE_PATH. To see where Node is installing its libraries, type the following:

```
$ npm ls -g
/usr/local/lib
```

(This command also lists all of the packages that npm has installed globally. If you omit -g, you'll see all the packages installed in the current directory.) We need to add the node_modules subdirectory of that path to NODE_PATH. On my system, that means adding the following to ~/.profile:

```
export NODE_PATH=/usr/local/lib/node_modules
```

Once again, the steps you'll need to take may be different on your system. To test that NODE_PATH is working its magic, start a new shell session and enter the node command. That'll take you to the Node.js REPL, an environment where you can interactively run commands. Now enter this:

```
> require('coffee-script')
```

I promise, that's the only JavaScript you'll have to type in this book!

If NODE_PATH isn't set correctly, you'll get Error: Cannot find module 'coffee-script'. If you see a long object description instead, you're golden. When you're done with Node's REPL, enter process.exit(), or just hit Ctrl - c .

By the way, the coffee-script library is beyond the scope of this book; suffice it to say that it lets you compile CoffeeScript to JavaScript from within your CoffeeScript or JavaScript program. You can do some pretty cool stuff with this, like writing your own compiler with custom postprocessing,[6] or writing your own build script as a Cakefile.[7]

Whew! I know that installation may have felt like a lot of work, but believe me, those efforts will pay off now that we have the full power of Node and npm at our disposal. Now let's set up your editing environment.

---

6. http://github.com/jashkenas/coffee-script/wiki/%5BExtensibility%5D-Hooking-into-the-Command-Line-Compiler
7. https://github.com/jashkenas/coffee-script/wiki/%5BHowTo%5D-Compiling-and-Setting-Up-Build-Tools

> **Staying on the Bleeding Edge**
>
> If you absolutely must have the latest CoffeeScript, it's actually pretty easy. Just use git to clone the CoffeeScript repo, then have npm install it from the local directory:
>
> ```
> $ git clone http://github.com/jashkenas/coffee-script.git
> $ cd coffee-script
> $ npm install -g
> ```
>
> This will install the current master branch, which may or may not be stable. You can revert to a specific version of CoffeeScript (say, 1.1.1) by running the following:
>
> ```
> $ npm install -g coffee-script@1.1.1
> ```

1.2 Text Editors for CoffeeScript

An up-to-date list of text editors with CoffeeScript support can be found at http://github.com/jashkenas/coffee-script/wiki/Text-editor-plugins. If you're on a Mac, I recommend the TextMate plugin maintained by Jeremy Ashkenas himself.[8] As of this writing, there are also plugins for Vim, Emacs, gedit, jEdit, and IntelliJ IDEA.

Recently, it's become viable to code with a web-based text editor, enabling real-time collaboration and freeing you from any particular device. Currently, the web-based editor with the best support for CoffeeScript is Cloud9, with the Cloud9 Live CoffeeScript Extension.[9]

Of course, you can use any editor you like, but using an editor with Coffee-Script support gives you three big advantages: syntax highlighting, smart indentation, and built-in compilation shortcuts. The first two are easy to appreciate, but the last is something many coders fail to take advantage of.

In TextMate, I can use ⌘R ("Run") to run a CoffeeScript file, or ⌘B ("Build") just to look at the compiled JavaScript. Compilation takes mere milliseconds, so if I'm not sure how a CoffeeScript expression translates into JavaScript, a quick build is the fastest way to find out. If text is selected, these commands run on the selection instead of on the whole file, which makes it a lot easier to test pieces of code and nail down syntax errors.

One quick caution—some editors (including TextMate) don't pick up PATH by default, which means you get an error like command not found when it tries to run coffee. If you run into this problem, go into your editor's preferences

---

8. http://github.com/jashkenas/coffee-script-tmbundle
9. http://cloud9ide.com/ and https://github.com/tanepiper/cloud9-livecoffee-ext, respectively.
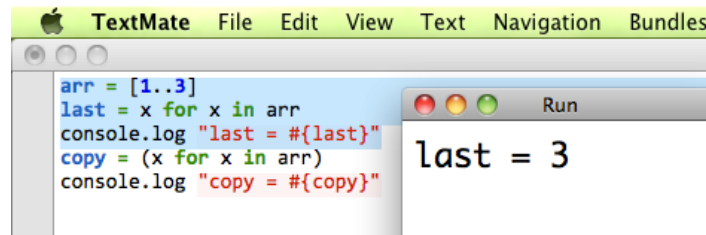
Figure 2—Running selected code directly from TextMate

(perhaps under Shell Variables) and set PATH to match the output you get when you run echo $PATH in your shell. You might want to set NODE_PATH while you're at it.

1.3 Meet 'coffee'

Now that you've got your editor set up, it's time to introduce coffee, the standard command line compiler. Let's start with the obligatory "Hello, world!" program. Open up your editor and create a new file called hello.coffee with the following contents:

```
console.log 'Hello, world!'
```

Now you just need to run it:

```
$ coffee hello.coffee
Hello, world!
```

You might be wondering several things: First, where did that console.log function come from? (Answer: It's a Node.js global.) Second, where's the JavaScript? Isn't the point of CoffeeScript that it compiles to JavaScript?

What's happening here is that coffee is compiling hello.coffee to JavaScript internally, then piping that output straight to Node for immediate execution. If that's not what you want to do, you'll have to use one or more of coffee's many options. To see them, use coffee -h:

```
$ coffee -h
Usage: coffee [options] path/to/script.coffee

  -c, --compile      compile to JavaScript and save as .js files
  -i, --interactive  run an interactive CoffeeScript REPL
  -o, --output       set the directory for compiled JavaScript
  -j, --join         concatenate the scripts before compiling
  -w, --watch        watch scripts for changes, and recompile
  -p, --print        print the compiled JavaScript to stdout
  -l, --lint         pipe the compiled JavaScript through JSLint
```

```
  -s, --stdio       listen for and compile scripts over stdio
  -e, --eval        compile a string from the command line
  -r, --require     require a library before executing your script
  -b, --bare        compile without the top-level function wrapper
  -t, --tokens      print the tokens that the lexer produces
  -n, --nodes       print the parse tree that Jison produces
      --nodejs      pass options through to the "node" binary
  -v, --version     display CoffeeScript version
  -h, --help        display this help message
```

So if you wanted to see the JavaScript that the compiler hid from you just now, you'd run this:

```
$ coffee -p hello.coffee
(function() {
  console.log('Hello, world!');
}).call(this);
```

See *JavaScript, Under Wraps*, on page 8 for an explanation of those extra two lines.

Compiling to JavaScript

Probably the most common flag is -c ("compile"), which saves the JavaScript output to a file. The file is named the same as the original, except with a .js extension instead of .coffee. Let's stick with the caffeinated beverage theme:

```
$ coffee -c mochaccino.coffee
```

This compiles to a file named mochaccino.js in the same directory. You can put the output somewhere else with the -o ("output") flag, followed by the name of the target directory:

```
$ coffee -co output source
```

This example reads every .coffee file in source (and its subdirectories) and writes the corresponding .js files in output. Note that -co is simply shorthand for -c -o. The order matters: -o needs to *immediately* precede the output directory name.

Another popular flag is -w ("watch"), which tells coffee to keep running in the background; in conjunction with -c, it'll recompile your code every time you make changes. It even works on directories and preserves nested file structures. So if I run the following, everything in the coffee directory will be continuously recompiled to the js directory:

```
$ coffee -cwo js coffee
```

This will continue until I kill the compiler with Ctrl - c .

JavaScript, Under Wraps

You're probably wondering why CoffeeScript output comes wrapped in a function. The reason is, in a word, *namespacing.* If you load a bunch of JavaScript files into a browser application, they're treated like one big block of code. That can easily lead to unintended consequences:

```javascript
// First file
function declareNuclearWar() {
  alert('Relax. This is only a test');
}
window.onload = function() {
  declareNuclearWar();
}

// Second file
function declareNuclearWar() {
  alert('The bombing begins in 5 minutes.');
}
```

Whoever wrote the first file had no idea the havoc that code was going to unleash! Calamity could have been averted by wrapping each file in an anonymous function, thus isolating the two declareNuclearWar declarations. (See Section 2.2, *Scope: Where You See 'Em*, on page 18.) This is called the *module pattern*.

To get modules to talk to each other, you've got to "export" some variables. (There's more on that in Section 4.1, *Modules: Splitting Up Apps*, on page 60.)

Oh—and if you must get rid of the wrapping, run coffee with the -b ("bare") flag.

The REPL

If you just run coffee with no arguments, you'll enter what overly sophisticated programmers call the REPL, or the Read-Eval-Print Loop. In layman's terms, this means you type something, it runs, you see the output, repeat.

This is great for playing around with the language. The REPL runs in a Node.js environment and prints the result of each expression. For instance, if we want to remind ourselves of some of the quirks of JavaScript's parseInt function, we can try this:

```
$ coffee
coffee> parseInt '221'
221
coffee> parseInt '221b'
221
coffee> parseInt 'b221'
NaN
```

That's it for our coverage of coffee. By the way, if you want to see how coffee works, check out the annotated source.[10] If you'd like, you can even reverse-engineer it and write your own interface for the CoffeeScript compiler (like my own Jitter[11]).

Remember that coffee is a lightweight tool; it doesn't offer features like minification or automatically running tests after compilation. If you want to add those to your project, you should write your own build script, typically as a Cakefile. You can find some documentation on Cakefiles over at the CoffeeScript wiki.[12]

You're almost ready to start writing CoffeeScript code—but first, what should you do if something goes awry?

1.4 Debugging CoffeeScript

One issue many folks have with writing code in a language like CoffeeScript is that runtime errors reference compiled code, not source code. That's a legit-imate concern, and several solutions have been discussed.[13] Unfortunately, for now you're left with stack traces whose line numbers have little to do with your source code.

Here's the good news: CoffeeScript's compiled JavaScript is very readable. If you understand how the two languages are related (and I hope you will after reading this book), then matching a point of failure in your program to the original CoffeeScript source is pretty easy.

It's not ideal, but it's the price of being on the cutting edge. As the Coffee-Script ecosystem grows and the tools get better, it'll get easier and easier to track down bugs. The folks at the Mozilla Foundation are hard at work adding CoffeeScript debugging support to Firefox, and Node can't be far behind. Until then, test your code thoroughly, use debug-mode logging, and know your JavaScript.

What was that middle thing? Oh, right. Under Node.js and browsers equipped with a developer console (or a bookmarklet like the previously mentioned Firebug Lite), you can display messages using console.log. Two possible problems: you don't always want to log every detail, and you don't want to call console.log if it doesn't exist. A common solution is to use a wrapper function, but then you don't get those precious JavaScript line numbers

---

when you log something (since all the logging is being done from the same place: the wrapper function). So here's one approach I recommend:

```
window.debugMode = document.location.hash.match(/debug/) and console?
console.log 'This is the first of many debug-mode outputs' if debugMode
```

In this example, debugMode will be true if and only if the "hash" in the address bar contains the string debug (e.g. page.html#debug) and the browser has a console object. This gives you an easy way to enable/disable all those extra messages when you load the page. Declaring debugMode as a property of window makes it a global variable.

A simpler but less versatile approach is to use a soak (see *Soaks: 'a?.b', on page 40*) to ensure that console.log is only called when console exists:

```
console?.log 'Thanks to ?, this line is perfectly safe!'
```

Under Node, there are plenty of libraries for displaying output at several levels of verbosity (just do a quick Google search for "nodejs logging library"), including my own styout, which comes with support for colored-console output.[14]

Logging can replace comments, providing more information during development on how the code is working. For example, here's a typical piece of documented code:

```
area = height * (base1 + base2) / 2
# now we have the area of the trapezoid
```

The comment could be replaced with a call to console.log as follows:

```
area = height * (base1 + base2) / 2
console.log "The area of the trapezoid is #{area}" if debugMode
```

Another common idiom is to make assertions throughout code. The standard console object has an assert function that serves that purpose nicely, taking a value and an error message (to be displayed if the value is non-truthy):

```
fundamentalLaws = ['death', 'taxes', 'gravity']
if debugMode
  console.assert 'gravity' in fundamentalLaws, 'gravity oughta be a law!'
```

Finally, the most important guard against bugs is to write well-structured code. While the tools don't exist yet to give you an exact line number for your runtime error, at least you should always be able to track down the part of your app that's causing you grief.

---

14. http://github.com/TrevorBurnham/styout

1.5 Ready to Roll!

In this chapter, you've learned how to install CoffeeScript on your machine using Node.js and npm. You've also gotten your favorite text editor on speaking terms with the language, explored some of the ways you can use CoffeeScript as part of your development workflow, and considered the challenge of debugging.

So now that you know how to run CoffeeScript code, it's about time we went into the nuts and bolts of the language itself. The rest of this book will be jam-packed with snippets of code. The best way to follow along is to run these from your favorite text editor; if you don't understand how they work, try tweaking a line or two to see what happens. You might also want to look at the compiled JavaScript from time to time.

Code snippets that refer to a file, like so, may require additional pieces in order to run:

Download GettingStarted/outOfContext.coffee
```
foo bar, baz
```

Those that don't are self-contained:

```
OK = 'computer'
console.log 'No alarms and no surprises.' if OK
```

And trust me—you're going to have a *lot* more fun if your editor is equipped with a Run command so that you can see the code's results just by tapping a keyboard shortcut. It's a CoffeeScript learner's best friend.

Functions, Scope, and Context

The heart and soul of CoffeeScript consists of two characters: ->. That's all it takes to define a new function, but don't let the terseness fool you; as we'll soon see, functions are powerful, versatile objects. Mastering them is the first step to mastering CoffeeScript.

While functions are the major players of this chapter, we'll meet a cheerful supporting cast along the way: variables, strings, conditionals, exceptions, and everything else you need to write useful functions. We'll also have a refresher on two crucial concepts, scope and context, and show how they carry over to CoffeeScript. Then we'll conclude our tour by looking at some very cool features: property arguments, default arguments, and splats.

At this point we'll be ready to tackle our first project, in which we'll put together an input prompt for our little word game. And last but not least, this chapter's exercises will push your newfound CoffeeScript expertise to its limits.

2.1 Functions 101

It's finally time to declare our first function! Check it out:

```
-> 'Hello, functions!'
```

I didn't say it would be a *useful* function, did I? But it does do something: it returns a string.

Don't take my word for it. Punch this into your favorite text editor and hit the Run command:

```
console.log do -> 'Hello, functions!'
```

You'll be greeted with something cheerful:

```
Hello, functions!
```

What does `do` do? (It has nothing to do with JavaScript's `do…while` loop.) It just means "run the following function." We could have done the same thing using a mess of parentheses:

```coffeescript
console.log (-> 'Hello functions!')()
```

```
Hello, functions!
```

"Where's the `return` keyword," you ask? CoffeeScript takes a cue from Ruby here, implicitly returning the last expression from each function. You can still use `return` explicitly, but it's optional, and the preferred style is to omit it unless you're breaking the flow of execution. If you *don't* want to return anything, use `return` by itself.

So far, our function has been anonymous. Anonymous functions have their uses, but this one is really aching for a name:

```coffeescript
hi = -> 'Hello, functions!'
console.log hi()
```

Once again, we get this response:

```
Hello, functions!
```

Naming a function in CoffeeScript just means assigning it to a variable. Note that we could have written `do hi` instead of `hi()`. But in practice, `do` is usually only used to create scope, especially during iteration. More on that in *Scope in Loops*, on page 96.

But what's the use of just returning a constant from a function? Not much. So let's make this function a bit more versatile:

```coffeescript
greeting = (subject) -> "Hello, #{subject}!"
console.log greeting 'arguments'
```

```
'Hello, arguments!'
```

We've added an argument list, `(subject)`, in front of the `->`. (Note that parentheses are optional in function calls but not in argument lists, except when the argument list is empty. For details, see *Implicit Parentheses*, on page 16.) And we've used string interpolation to insert an expression into a string.

CoffeeScript's interpolation syntax is similar to Ruby's: `"A#{expression}Z"` is equivalent to `'A' + (expression) + 'Z'`. Interpolations only work in *double-quoted* strings. (As a matter of style, I prefer to use single-quoted strings whenever I'm not doing an interpolation so as to clearly convey that there's no funny business going on.)

> ### A Tale of Two Function Declaration Syntaxes
>
> In JavaScript, there are two ways of defining a function. Here's one:
>
> ```
> var cube1 = function(x) { return Math.pow(x, 3); };
> ```
>
> Here's another:
>
> ```
> function cube2(x) { return Math.pow(x, 3); }
> ```
>
> The main difference between these two is that if you were to call `cube1` before it's defined, you would get an error, but if you called `cube2` from earlier within its scope, the interpreter would automatically look ahead to the function definition.
>
> Due to a thorny issue in IE, CoffeeScript always generates variable-style function declarations, like `cube1`. (The one exception is that "named" functions are generated by the `class` keyword, as we'll see in Section 4.3, *Classes: Functions with Prototypes*, on page 63.) So don't forget to define your functions before you call them!

Word to the wise: CoffeeScript's + operator is sensitive to whitespace. So string concatenation like this works fine:

```
squadron = 'Red'
xWing = squadron + 5     # 'Red5'
```

However, this doesn't:

```
squadron = 'Red'
xWing = squadron +5      # TypeError
```

The problem is that `squadron +5` compiles to `squadron(+5)`. (The + prefix is a handy way of converting strings to numbers.) Since `squadron` is a string, not a function, this gives us an error. String interpolation prevents this gotcha:

```
squadron = 'Red'
xWing = "#{squadron}5"  # 'Red5'
```

Accessing 'arguments'

This is as good a time as any to mention that you can access all arguments to a function using JavaScript's array-like `arguments` object, whether they're declared in the argument list or not. For example, we could've written our `greeting` function like this:

```
greeting = -> "Hello, #{arguments[0]}!"
```

The `arguments` object is typically used when a function needs to accept a varying number of arguments. Of course, this versatility comes at the price of readability. It's also a common source of JavaScript headaches, as `arguments` acts like an array without supporting many of a normal array's methods.

> ## Implicit Parentheses
>
> The ability to omit parentheses from function calls is a double-edged sword. To use this power wisely, you need to understand one simple rule: *implicit parentheses don't close until the end of the expression.*
>
> Expecting CoffeeScript to understand what you're trying to do is a common rookie mistake. For instance, you might be surprised if you were to write the following:
>
> ```
> console.log(Math.round 3.1, Math.round 5.2)
> ```
>
> The output for this is 3. What happened to `Math.round 5.2`? The answer is clear when we make the parentheses explicit:
>
> ```
> console.log(Math.round(3.1, Math.round(5.2)))
> ```
>
> `Math.round(5.2)` was evaluated, but then it was passed as an argument to the other `Math.round` (which ignored it), rather than to `console.log` as intended.
>
> To avoid confusion, I like to use parentheses for everything but the outermost function call:
>
> ```
> console.log Math.round(3.1), Math.round(5.2) # 3, 5
> ```

Fortunately, you rarely need to work with `arguments` directly in CoffeeScript, thanks to a feature we'll learn about in .

Conditionals and Exceptions

Now, let's write a numeric function for a change of pace:

```
cube = (num) -> Math.pow num, 3
```

Notice that the `Math` object, as part of the JavaScript standard, is identical in CoffeeScript (and, for that matter, across all major browser and server-side environments).

How about something a little more complex: a boolean test?

```
odd = (num) -> num % 2 is 1
```

`%` is the modulus operator; it gives us the remainder after division. The `is` keyword compiles to JavaScript's `===`, the strict equality operator. (There is no analog to JavaScript's `==`; see .) Hence, `odd` will return `true` if the given number is a positive integer that is not divisible by 2 and `false` otherwise. (Because `%` coerces its values to numbers, `odd` will also return `true` for, say, the string `'3'`.)

Now in most settings this would be considered a perfectly good oddness check. But let's suppose that you're writing a math library with very strict

> ## Strict Equality or Nothing
>
> CoffeeScript's is and == both compile to JavaScript's ===; there's no way to get the loose, type-coercing equality check of JavaScript's ==, which is frowned upon by JSLint and others as the source of many "WTF?" moments. Let's borrow an example from http://wtfjs.com/2011/02/11/all-your-commas-are-belong-to-Array:
>
> ```
> ",,," == new Array(4)  // true
> ```
>
> There are also cases where == isn't transitive:
>
> ```
> '' == '0'          // false
> 0 == ''            // true
> 0 == '0'           // true
> ```
>
> To avoid these head-scratchers, you should perform type conversions explicitly.

specifications that state that if the function is given a value that isn't strictly a positive integer, then it should throw an exception. We can do that by using conditionals, like so:

Download Functions/odd.coffee

```coffee
odd = (num) ->
  if typeof num is 'number'
    if num is Math.round num
      if num > 0
        num % 2 is 1
      else
        throw "#{num} is not positive"
    else
      throw "#{num} is not an integer"
  else
    throw "#{num} is not a number"
```

Note the use of significant indentation to delimit both the function and each conditional branch, rather than the curly braces of JavaScript. In Coffee-Script, curly braces are used for one thing only: declaring JSON-style objects. (More on that in the next chapter.)

Now if you try calling odd with anything but a positive integer, you'll get undefined (since throw statements have no return value). In order to actually see the error message, you'll need to use a try...catch block:

Download Functions/odd.coffee

```coffee
try
  odd 5.1
catch e
  console.log e

5.1 is not an integer
```

We could improve the style of the odd function by simply checking each of our three conditions in turn:

```
Download Functions/odd.coffee
odd = (num) ->
  unless typeof num is 'number'
    throw "#{num} is not a number"
  unless num is Math.round num
    throw "#{num} is not an integer"
  unless num > 0
    throw "#{num} is not positive"
  num % 2 is 1
```

(If those lines weren't so long, we'd use the postfix style, throw a unless b, rather than the indented style.) In general, whenever conditions lead to throw or return, we can simplify branching logic to a simple series check.

Of course, functions aren't limited to just returning values and throwing exceptions; they can also do stuff by modifying variables and running other functions. (In functional programming parlance, these are known as *side effects*.) These work just as you would expect from JavaScript:

```
count = 0
incrementCount = -> count++
incrementCount()   # count is now 1
```

Now you know the basics of defining and calling functions in CoffeeScript. But the devil's in the details, so let's look at one of the most important details: Where are variables visible?

2.2   Scope: Where You See 'Em

So far, we haven't worried about where variables live. Alas, we can't always be so cavalier. Consider this example:

```
age = 99
reincarnate = -> age = 0
reincarnate()
console.log "I am #{age} years old"
```

As you'd probably expect, the output is this:

```
I am 0 years old
```

However, we can try flipping the first and second lines around, like this:

```
reincarnate = -> age = 0
age = 99
reincarnate()
console.log "I am #{age} years old"
```

Now hit Run and the code tells a very different story:

```
I am 99 years old
```

Strange—the reincarnate() call doesn't have any effect! And what if we cut out age = 99 altogether?

```
reincarnate = -> age = 0
reincarnate()
console.log "age = #{age}"
```

```
ReferenceError: age is not defined
```

We've got ourselves a "scope" issue! But what is *scope*? A variable's scope is its home, as defined by three rules:

1. Every function creates a scope, and the only way to create a scope is to define a function.

2. A variable lives in the outermost scope in which an assignment has been made to that variable.

3. Outside of its scope, a variable is invisible.

For instance, the scope of age in the first example was the global scope; in the middle example, there was a variable named age in the global scope and another in the reincarnate function's scope; and in the last example, there was just the reincarnate-scoped age. That's why we got a ReferenceError from trying to display age outside of reincarnate: no variable with that name exists outside of that function.

CoffeeScript's approach to scope is known as *lexical scope*, and it's the same as in JavaScript, except that a variable's scope is defined explicitly in JavaScript using the var keyword while CoffeeScript infers scope from assignments. This not only saves your fingers precious effort, it also discourages you from shadowing one variable by giving a different variable in a nested scope the same name. See *Shadowing: The Name's the Same,* on page 20.

A function's scope is nested within the scope that the function itself lives in. (Remember, functions are just variables.) This is another important thing to understand about scope: it doesn't depend on where or how the function is being called. You can always tell which scope something lives in just by eyeballing the code or by compiling to JavaScript and looking for var declarations, which are always placed at the top of their scope:

Shadowing: The Name's the Same

There are only two ways to shadow a variable in CoffeeScript: One is, as we saw in the second reincarnate example, to create a variable in a surrounding scope *after* a variable with inner scope. The other way is with a function argument:

```
x = 5
triple = (x) -> x *= 3
triple x     # 15
x            # 5
```

Shadowing is generally considered bad style and should be avoided. Give your variables distinct names, lest you sew the seeds of scope confusion.

```coffeescript
singCountdown = (count) ->
  singBottleCount = (specifyLocation) ->
    locationStr = if specifyLocation then 'on the wall' else ''
    bottleStr = if count is 1 then 'bottle' else 'bottles'
    console.log "#{count} #{bottleStr} of beer #{locationStr}"
  singDecrement = ->
    console.log "Take one down, pass it around"
    count--
  singBottleCount true; singBottleCount false
  singDecrement(); singBottleCount true
  if count isnt 0 then singCountdown count
```

This example yields the following (omitting everything but the functions that create scope and the vars that inhabit them):

```javascript
var singCountdown;
singCountdown = function(count) {
  var singBottleCount, singDecrement;
  singBottleCount = function(specifyLocation) {
    var bottleStr, locationStr;
    // ...
  }
  // ...
}
```

You might be wondering, "How do I give a variable a scope without making an assignment?" The answer is, you don't. Instead, you make an assignment, traditionally with null or some more sensible initial value. Here's an example:

```coffeescript
obj = null
initializeObj = ->
  obj = ...   # create object with superpowers
window.onload = initializeObj
```

That does it for our discussion of scope in CoffeeScript. And now for something completely different: this.

2.3 Context (or, "What Is 'this'?")

Scope and context are kissing cousins, but don't get 'em mixed up. While scope is about which variable an identifier is referring to, context (also known as the receiver) is about the this keyword—and its handy CoffeeScript shorthand, @.

JavaScript and CoffeeScript newcomers often find this baffling. Used rightly, it feels like magic. Used wrongly, it may be unrivaled as a source of errors. No doubt some of the confusion stems from the word itself; people expect this to refer to "*this* object." Instead, you should think of it as "*this* context." And as we'll soon see, the context (aka this/@) can be something different every time a function is called.

(Before we go on, I should note that using the term *context* to describe this, while popular, is not standard. Some frown on it because the JavaScript specification defines something called *execution context*, which is related but different. Unfortunately, there is no other universally agreed-upon term for the value of this, so throughout this book, I'll continue to refer to it as the context.)

Let's go through some examples using this simple function:

```coffee
setName = (name) -> @name = name
```

Here, name and @name are totally different variables: name (which we could call anything, really) is a local variable, one that will never be visible outside of the function, while @name (shorthand for this.name) is a property of the context.

The main purpose of context is to give an object's *methods* (functions attached as properties) a way of referring to the object they're being called on:

```coffee
cat = {}
cat.setName = setName
cat.setName 'Mittens'
console.log cat.name   # 'Mittens'
```

When we call cat.setName, we're calling setName with the cat object as its context; thus this (or @) refers to the cat, and @name refers to cat.name. The function itself hasn't changed. We could call the following:

```coffee
setName 'Mr. Mistoffelees'
```

And it would have no effect on cat.

We can invoke a function in a particular context without attaching the function to that object by using the call or apply methods, which all functions have. (If you're fuzzy on how properties work in JavaScript or surprised to learn that functions are objects, then you may want to skip ahead to Section 3.1, *Objects as Hashes*, on page 37 and then come back.) apply takes a context and an array of arguments to pass to the function:

```coffee
pig = {}
setName.apply pig, ['Babe']
console.log pig.name   # 'Babe'
```

call works the same way, except that it takes individual arguments rather than a single array. So the equivalent of the previous would be this:

```coffee
setName.call pig, 'Babe'
```

In practice, apply is more commonly used than call because it's more versatile: call only lets you change the context of a normal function call, while apply lets you change the context *and* pass in an arbitrary number of arguments.

You can use call and apply to "borrow" one object's methods and use them on another:

```coffee
horse = {}
cat.setName.apply horse, ['Mr. Ed']
console.log horse.name   # 'Mr. Ed'
```

Here, it doesn't matter that we're using cat.setName instead of setName: they're the same function.

Finally, one last way of giving a function a context is with new, which creates a new object using the function as a constructor:

```coffee
Dog = setName    # By convention, constructors are capitalized
dog1 = new Dog('Jimmy')
dog2 = new Dog('Jake')
console.log dog1.name   # 'Jimmy'
console.log dog2.name   # 'Jake'
```

The new keyword says, "Don't return the result of the function; instead, create a new object, run the function in that object's context, and then return the object." (We can also give the object created by new additional properties using a prototype, as we'll see in Section 4.2, *The Power of Prototypes*, on page 61.) Because the new keyword sets the new Dog as the context, @name points to the new dog's name property.

If a function isn't called as a method and you don't use the new keyword or call/apply, then the context is the global object. We'll learn more about the global context in Section 4.1, *Modules: Splitting Up Apps*, on page 60; for now, just remember that it's generally a bad idea to use this when it's pointing at the global object:

```
Download Functions/setName.coffee
setName 'Lulu'
console.log name      # 'Lulu'
console.log @name     # undefined
```

So, you see that context is determined purely by how a function is called; unlike scope, it has nothing to do with where the function is defined. (Of course, we'd often like for the context to be determined by where the function is defined. Fortunately, there's a clever technique that lets us effectively do this, one we'll meet in *Bound Functions: 'this' Means 'this'*, on page 23.)

To review, here are the CoffeeScript rules of context in a nutshell, with earlier rules taking precedence over later ones:

1. When the new keyword is put in front of a function call, its context is the new object.

2. When a function is called with call or apply, the context is the first argument given.

3. Otherwise, if a function is called as an object property (obj.func) or obj['func']), it runs in that object's context.

4. If none of the above apply, then the function runs in the global context.

We'll learn more about the global context in Section 4.1, *Modules: Splitting Up Apps*, on page 60.

Bound Functions: 'this' Means 'this'

Sometimes you want a function to run in the current context no matter how it's called. This is especially common with event callbacks. Let's say that you want someone to leave a message in your voicemail array (bound to the current context). Then you might write something like this:

```
callback = (message) -> @voicemail.push message
```

Ah, but of course you realize that when callback is called without the relevant object, this will simply refer to the global object—or to whichever context the other library sets via call or apply. Isn't there any way to make this always point to the same object, no matter how the function is called?

JavaScript lets you do that, but there's a lot of boilerplate code involved (see *How Does => Work?, on page 25*, though it's worth mentioning that ECMAScript 5, a standard supported by the latest generation of browsers, provides a much simpler bind method on the Function prototype). Thankfully, CoffeeScript makes binding a function to the current context as easy as a single character change: => instead of ->. We call => the "bound function operator" or, less formally, the "fat arrow."

So our callback function becomes this:

```
callback = (message) => @voicemail.push message
```

Now we can relax, assured that the meaning of this is the same within the function as it is where the function is defined, no matter where it's called from!

You might wonder why you shouldn't just always use => instead of ->. There are two reasons. First, the binding code leads to some small overhead in both file size and execution time and is usually unnecessary. But more importantly, while the chameleon-like nature of context is often confusing, it can also allow for very elegant code. For instance, many libraries pass critical information to callback functions through context. Here's a simple example (one that will make more sense in Chapter 5, *Web Interactivity with jQuery, on page 75*):

```
$('#clickToHide').click -> $(this).hide()
```

Rather than strictly using either normal functions or bound functions, you need to think carefully each time you define a function that uses this/@ about what the context should mean.

That's it for scope and context—consider yourself a master of function semantics! The next couple of sections will cover some helpful syntactic sugars before we finally tackle the first iteration of our game project.

2.4 Property Arguments (@arg)

Remember that function we defined just to set a property on its context object to the value of an argument?

```
setName = (name) -> @name = name
```

Well, it turns out CoffeeScript offers a handy shorthand for this:

```
setName = (@name) -> # no code required!
```

> ### How Does => Work?
>
> We could write callback = (message) => @voicemail.push message in JavaScript like so:
>
> ```
>   var callback;
> ① callback = (function(__this) {
> ②   var __func = function(message) {
>       return this.voicemail.push(message);
>     };
> ③   return function() {
>       return __func.apply(__this, arguments);
>     };
> ④ })(this);
> ```
>
> ① ④
>
> > The outermost function takes the current context, this, as an argument called __this.
>
> ②
>
> > __func contains the code of the function we want to bind to the current context.
>
> ③
>
> > The anonymous function defined here is what becomes callback. So whenever callback is called, its arguments are passed to __func, which is run in the __this scope, and __func's result is returned.
>
> Notice that the context given to callback itself is never used, and __func is never exposed to the outside world, ensuring that it's always called in the context in which callback was defined.
>
> In practice, CoffeeScript uses a helper function named __bind, but the underlying technique is the same. => may be CoffeeScript's most powerful shorthand.

Quite simply, when @ precedes the name of an argument to a function, that function automatically makes the assignment from the argument to the property with that name on the context object, this.

This is especially great for constructors, which we'll be up to our knees in when we get to Chapter 4, *Modules and Classes,* on page 59. It's common to pass four or five arguments to a constructor just to set initial properties on the instance object. By using the @argument syntax instead, you save that many lines of code. Nice.

2.5 Default Arguments (arg =)

Let's say that you have a function where one of its arguments is going to have one particular value most of the time, like this:

```
ringFireAlarm = (isDrill) ->
  # it's pretty much always a drill
  ...
```

Wouldn't it be nice if ringFireAlarm() were shorthand for the much more common ringFireAlarm true? Well, we could do that by writing this:

```
ringFireAlarm = (isDrill) ->
  isDrill = true unless isDrill?
  ...
```

Here unless (expression) is a shorthand for if not (expression). The ? in isDrill? is the existential operator, and it's shorthand for checking that the given value (1) exists in this scope and (2) isn't undefined or null. In your head, x? should read as "x exists."

The existential operator can also be placed between two variables: a ? b returns a if a?, and b otherwise. And it can be combined with = to form a compound assignment operator: c ?= d is shorthand for c = d unless c?. You can read that as, "Let d be the default value for c."

```
ringFireAlarm = (isDrill) ->
  isDrill ?= true
  ...
```

Of course, as you probably inferred from the section heading, there's an even more succinct way of doing this:

```
ringFireAlarm = (isDrill = true) ->
  ...
```

"So why," you might ask, "did you spend all that time talking about the existential operator?" There are two reasons. First, the existential operator is *sweet*. Second, the behavior of default arguments in CoffeeScript is somewhat different from that of other programming languages like Ruby, Python, and PHP. In those languages, the number of arguments passed to the function is what matters—the isDrill = true assignment would only be carried out if ringFireAlarm was called with no arguments. By contrast, CoffeeScript uses the existence operator behind the scenes. This means that explicitly passing null or undefined is the same as omitting the argument altogether.

This can lead to unpleasant surprises if you're not careful, but it's also more flexible. You can have an argument with several default values, and callers can choose to use the defaults for any subset of them, like so:

```
chooseMeals = (breakfast = 'waffles', lunch = 'gyros', dinner = 'pizza') ->
  ...
chooseMeals null, 'burrito', null   # not a gyro fan
```

You can, of course, implement the more conventional default argument behavior by making the assignments conditional on arguments.length. But if you

Getting Truthy with 'or='

You're likely to see someone write the following:

```
a or= b
```

This (or it's equivalent, a ||= b) is a way of making b the default value for a. How, exactly, does this differ from a ?= b?

The answer has to do with "truthiness." In CoffeeScript (as in JavaScript), all values are implicitly coerced to booleans by the boolean logic operators, && and || (known as and and or under CoffeeScript style), as well as if. Most values become true, while a handful—notably null, undefined, 0, and the empty string—become false.

This loose, flexible approach to boolean logic carries over to the results of boolean operators. While some languages would just return true or false from the statement a or b, CoffeeScript (like JavaScript) returns a if a is truthy and returns b otherwise. (x and y returns y if both are truthy, x if both are falsy, and the falsy one if only one is falsy.)

This gives us lots of useful shortcuts—including a = a or b (typically shortened to a or= b) as a way of saying "set a to b if the value of a is falsy." While not quite as handy as ?=, or= is a good friend to have.

find yourself doing so, ask yourself whether you really need to accept null as a value. How about false or NaN or even a custom type instead?

There is one more detail worth mentioning—you can use arbitrary expressions as default arguments, though this generally isn't recommended. If you do so, the expression will be executed from whatever context the function is being called in, before any expression in the function body and only if the assignment is made. In other words, the following two expressions are *exactly* equivalent. Here's the first way:

```
dontTryThisAtHome = (noArgNoProblem = @iHopeThisWorks()) ->
  ...
```

Here's the other way:

```
dontTryThisAtHome = (noArgNoProblem) ->
  noArgNoProblem ?= @iHopeThisWorks()
  ...
```

There's just one last feature to talk about before we get to our first project, and it's a humdinger. Never before has a single ellipse done so much…

2.6   Splats (...)

Taking a varying number of arguments is both easy and hard in JavaScript: it's easy because every argument passed to a function (regardless of the number in the function's declaration) is available from the arguments object, and it's hard because arguments doesn't support standard Array methods like slice and shift.

Fortunately, CoffeeScript lets you convert any range of arguments to an array automatically. Just add an ellipsis, ... (also known as "the splat"), to the end of any argument name:

```
refine = (wheat, chaff...) ->
  console.log "The best: #{wheat}"
  console.log "The rest: #{chaff.join(', ')}"
```

The splat here means "take every argument after the first one, wheat, and combine them into an array, chaff." Calling refine with a list of four arguments results in the following:

```
refine 'great', 'not bad', 'so-so', 'meh'

The best: great
The rest: not bad, so-so, meh
```

If just one argument is given, or if none are, then chaff will just be an empty array.

A splatted argument doesn't have to go at the end of an argument list. The CoffeeScript compiler is smart about determining the appropriate arguments to put in the array:

```
sandwich = (beginning, middle..., end) ->
  ...
```

Nonsplatted arguments always get filled in first. So if sandwich is called with only two arguments, those will become beginning and end. Only when there are three or more arguments is there a middle. *Splats soak up any and all extra arguments.*

Even if the splat comes first, the plain arguments take priority:

```
spoiler = (filler..., theEnding) -> console.log theEnding
spoiler 'Darth Vader is Luke\'s father!'

Darth Vader is Luke's father!
```

Of course, it only makes sense to have one splatted argument in a given function. Otherwise, the splats would have to duke it out over how to split the arguments amongst themselves.

It's worth mentioning that splats can also be used to divvy up arrays on the fly without the use of a function. Go ahead and launch the REPL (just run coffee with no arguments, remember?) and play with this feature a little:

```
coffee> birds = ['duck', 'duck', 'duck', 'duck', 'goose!']
coffee> [ducks..., goose] = birds
coffee> ducks
duck,duck,duck,duck
```

We'll learn more about this syntax in Section 3.6, *Pattern Matching (or, Destructuring Assignment)*, on page 48.

In a function call, splats mean precisely the inverse of what they mean in argument lists and pattern-matching assignments: they expand an array into a series of arguments, rather than collapsing a series of arguments into an array. Let's go to the REPL again:

```
coffee> console.log 1, 2, 3, 4
1 2 3 4
coffee> arr = [1, 2, 3]
coffee> console.log arr, 4
[ 1, 2, 3 ] 4
coffee> console.log arr..., 4
1 2 3 4
```

As you might have guessed, this syntax uses apply (which we met in Section 2.3, *Context (or, "What Is 'this'?")*, on page 21) behind the scenes.

Hopefully this chapter has given you a lot to take in. Now it's time for us to put it all together with a small project, followed by a healthy helping of brainteasers.

2.7   Project: 5x5 Input Parser

Remember our little word game idea (see Section 3.4, *About the Example Project: 5x5*, on page xix)? Well, it's time to turn it into a reality! Now, we don't know enough about hashes, arrays, and loops yet to implement a fully working version (omissions we'll rectify in Chapter 3, *Collections and Iteration*, on page 37). Still, we can at least get our feet wet by writing a command-line prompt using Node.js.

But before we get started, we need to understand nonblocking IO. In most languages, we could write something like this:

```
input = getInput()
# now process input...
```

The getInput function would wait for the user to type something and then returns it. This is called *blocking* IO, because getInput would "block" the execution of our program until the input is available.

However, we can't do this with Node because its IO is *nonblocking* (with the exception of a handful of convenience functions whose names end with Sync). Instead, we need to give Node a callback, which will get run whenever there's an input event. (We'll go into more depth on Node's event model in Section 6.3, *Thinking Asynchronously*, on page 93.) The closest analog to the above in Node is this:

```
stdin.on 'data', (input) ->
  # now process input...
```

(The stdin object requires some initialization, which we'll get to in a moment.) If you're used to blocking IO, the transition to nonblocking IO can be jarring. But the benefits can be tremendous, because waiting for input (and, to a lesser extent, output) has long been a major source of performance degradation and limited scalability. Neither of these is really a big issue in this particular app, but designing applications to keep running while waiting for input is a good habit to develop, and Node essentially forces you to. The only way to write a blocking function is to create a native extension in C++.

So let's think a little bit about the structure of our app. Here's what we need it to do:

1. Prompt for coordinates (x, y) for the first tile.

2. If the input is valid (two integers, each between 1 and the size of the grid), then prompt for the coordinates of the second tile.

3. Validate the input again. If it passes, say we're swapping the two tiles. If it fails, explain why and offer a chance to try again.

Let's start by "opening" the standard input:

Download Functions/5x5/prompt.coffee
```
stdin = process.openStdin()
stdin.setEncoding 'utf8'
```

Where did process come from? It's part of the Node environment, one of the few parts that doesn't require a require statement to access. process provides methods for getting command-line arguments, managing memory, and, of course, dealing with standard IO.

If we run the app at this point (with coffee prompt.coffee), we get a never-ending series of prompts. (Ctrl - c is your friend.) Each time you hit Return , Node looks for a callback to pass that input to. Since we haven't given it a callback, nothing happens. Let's fix that, shall we?

Download Functions/5x5/prompt.coffee
```coffee
inputCallback = null
stdin.on 'data', (input) -> inputCallback input
```

The stdin.on 'data' call tells Node, "Each time a new line of input comes in, pass it to this function." That function simply forwards the input to another function, inputCallback. Or rather, inputCallback will be a function later—right now it's just null. Why are we doing this? Because this proxy function makes it easy to change the callback behavior. The inputCallback = null line (where null is arbitrary) tells the compiler to give the variable module-level scope, allowing it to be modified outside of the anonymous function.

Note that if we tried to set multiple callbacks on stdin.on 'data', they would simply "stack," so each one would be called every time new input came in. We could unbind the existing callback using stdin.removeListener if we stored a reference to the listener, but that would entail a two-step process (unbind, bind). Instead, we just change the value of inputCallback.

Our simple app will have two "states": prompting for the coordinates of the first tile and prompting for those of the second. Going into a state is a simple matter of displaying a message and then setting inputCallback:

Download Functions/5x5/prompt.coffee
```coffee
promptForTile1 = ->
  console.log "Please enter coordinates for the first tile."
  inputCallback = (input) ->
    promptForTile2() if strToCoordinates input
```

When new input comes in, the callback checks with the not-yet-defined strToCoordinates function; if it gives the go-ahead, we pass control to the mirror image prompt for the second tile of the move:

Download Functions/5x5/prompt.coffee
```coffee
promptForTile2 = ->
  console.log "Please enter coordinates for the second tile."
  inputCallback = (input) ->
    if strToCoordinates input
      console.log "Swapping tiles...done!"
      promptForTile1()
```

Now, what's this about validation? Well, first let's write a simple test for whether a (zero-indexed) x, y pair is on the grid or not:

```coffee
GRID_SIZE = 5
inRange = (x, y) ->
  0 <= x < GRID_SIZE and 0 <= y < GRID_SIZE
```

GRID_SIZE is in all-caps to indicate that it's a constant (a stylistic convention for us humans, not the compiler). The function takes advantage of Coffee-Script's chained comparisons feature: 0 <= x < GRID_SIZE is shorthand for (0 <= x) and (x < GRID_SIZE).

Here's another simple test for whether a number is an integer:

```coffee
isInteger = (num) ->
  num is Math.round(num)
```

Now let's use these to make a magical string-to-coordinate converter, complete with handy error messages:

```coffee
strToCoordinates = (input) ->
  halves = input.split(',')
  if halves.length is 2
    x = parseFloat halves[0]
    y = parseFloat halves[1]
    if !isInteger(x) or !isInteger(y)
      console.log "Each coordinate must be an integer."
    else if not inRange x - 1, y - 1
      console.log "Each coordinate must be between 1 and #{GRID_SIZE}."
    else
      {x, y}
  else
    console.log 'Input must be of the form `x, y`.'
```
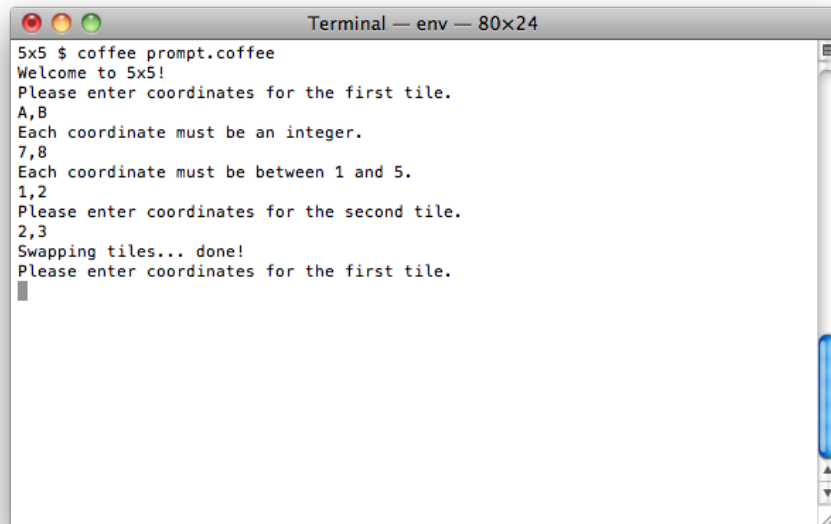
You might wonder how this works for the if conditions in the two inputCallback implementations. Well, console.log returns undefined, which gets coerced to false by condition checks, and any nonempty object gets coerced to true. So, we either return {x, y} or we give an error message, and all our boolean needs are taken care of for us.

Now there's just one piece missing: we need to start in one of our states so that inputCallback gets defined before it's called:

```coffee
console.log "Welcome to 5x5!"
promptForTile1()
```

And that's it! Run prompt.coffee to try it for yourself. It should look like Figure 3, *Playing with our command-line prompt,* on page 33.

Figure 3—Playing with our command-line prompt

Hopefully you've developed an appreciation for the power and syntactic ease of CoffeeScript functions. In the next chapter, we'll learn about working with objects and arrays, and we'll turn our little experiment in standard IO into a full-fledged game.

2.8 You've Done Well, Young Padawan

It's safe to say that you now know more about CoffeeScript than 99.999 percent of the Earth's population. You've learned how to define, call, and return values from functions. You've also learned that functions create scope, and that the context variable this is a fickle creature that depends on how the function is called—unless the function is bound to the context it's defined in by being defined with =>.

You've also sampled a smorgasbord of CoffeeScript's other features: if/unless, try...catch, default argument values, and property arguments among them. Then we used those features to write a simple command-line app that runs on Node.js.

There are only two areas of CoffeeScript that we haven't touched yet: collections (objects and arrays) and iteration (loops). By a happy coincidence, those are the subject of the next chapter.

But before we go on, test your knowledge with the following exercises. You weren't thinking of skipping them, were you? Trust me: they'll save you from pain in the future.

2.9   Exercises

1. The following function will remove all elements from the given array and return the result of the splice, which in this case will be a copy of the original array. (We'll learn more about that particular function in *Slicing and Splicing,* on page 42.)

   ```
   clearArray = (arr) ->
     arr.splice 0, arr.length
   ```

   How would I make clearArray return the cleared array instead? How would I make it return nothing at all?

   (While this is a trivial example, marking functions as returning nothing often allows the CoffeeScript compiler to generate more efficient output, especially when loops are involved.)

2. Write a function called run that takes a function as its first argument and passes all additional arguments to the called function. That is, run func, a, b should be equivalent to func(a, b). Hint: This shouldn't take more than one line.

3. Implicit parentheses always go to the end of the expression but not necessarily to the end of the line. Find a case where implicit parentheses fall short of the end of the line.

4. When you use explicit parentheses in a function call, CoffeeScript doesn't allow any whitespace after the function name. For instance, f (a, b) is a syntax error. Can you think of a reason why this rule is in place? (Hint: What does f (g) h mean?)

5. What is the context of the function call foo.bar.baz()? What about @hoo()? @hoo.rah()?

6. x refers to a variable that obeys scoping rules, while @x refers to a variable that obeys context rules. The two will never be equivalent—they may reference the same object, but writing x = y would not affect @x and vice versa.) But what.x and @x *can* be equivalent. If they are, then what is what?

To find the answer, add a very short line to this code that will generate the output "quantum entanglement":

```
xInContext = ->
  console.log @x
what = {x: 'quantum entanglement'}
```

7. Will this code work?

```
x = true
showAnswer = (x = x) ->
  console.log if x then 'It works!' else 'Nope.'
showAnswer()
```

Explain why or why not.

Collections and Iteration

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 .

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 25*. 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()
```

---

1. http://developer.mozilla.org/en/JavaScript/Reference/Global_Objects/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 Prototype.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
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
coffee> ['a', 'b', 'c', 'd'][0...3]
['a', 'b', 'c']
```

And you can use an inclusive range, too:

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

> ## 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.

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
['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,

'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.

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.

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

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 .

also the only way to prevent any value from being added to the list returned by the loop, as we'll see in .

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 66.)

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.

3.6 Pattern Matching (or, Destructuring Assignment)

In JavaScript, assignment is a strict one-at-a-time affair. If you had a list of values that you wanted to transfer to a list of variables, you'd probably write a custom function. But in CoffeeScript, you can just write one line of code:

```
[firstName, middleInitial, lastName] = ['Joe', 'T', 'Plumber']
```

This syntax, called *array pattern matching*, may look odd at first. The square brackets on the left side of the assignment aren't really creating an array, after all. Instead, it's just describing a "pattern" of variables that the array on the right side fills in. So the line above is equivalent to this:

```
firstName = 'Joe'; middleInitial = 'T'; lastName = 'Plumber'
```

But array pattern matching isn't just a pretty syntax for making multiple assignments. For one thing, you can reference the same variable(s) on both sides of the assignment, making swaps a one-line affair instead of a three-line chore:

```
[newBoss, oldBoss] = [oldBoss, newBoss]
```

We can also use splats the same way that we used them in function definitions in Section 2.6, *Splats (...)*, on page 28:

```
[theBest, theRest...] = topStudents
```

If you think array pattern matching is great, you'll love object pattern matching. Say we want to pick out a few values from an object:

```
myRect =
  x: 100
  y: 200
{x: myX, y: myY} = myRect
```

Again, the "object" on the left side of the assignment isn't really an object, it's a pattern, providing keys and variable names to match these keys. So this is equivalent to the following:

```
myX = myRect.x; myY = myRect.y
```

This might not look like much of a gain, but remember (as we discussed in *Same-Name Key-Value Pairs*, on page 39) how {x} is short for {x: x}? Well, that works in patterns, too, which means that if you just want to copy rect.x and rect.y to local variables named x and y, then all you have to write is this:

```
{x, y} = rect
```

Believe me, this syntax will change the way you read properties from objects *forever*. To give another example, let's say that we're writing some tests using Node's assert module. Specifically, we want to use the assert.ok and assert.strictEqual methods. We can load them as variables called ok and strictEqual by writing this:

```
{ok, strictEqual} = require 'assert'
```

One last tip—did I mention that array patterns and object patterns can be nested inside of each other?

```
{languages: [favoriteLanguage, otherLanguages...]} = resume
```

This code translates as "Take resume.languages, assign its first entry to a variable called favoriteLanguage, then assign the remaining entries to a new array called otherLanguages." Not bad for one line, right?

Pattern matching is also known as *destructuring assignment* and is a part of the JavaScript 1.7 standard (currently supported by the latest version of Firefox). But you don't have to wait for that standard to catch on—Coffee-Script compiles each pattern to a series of simple assignments.

I know this chapter has thrown a lot at you, but now it's time to pick up the pieces and put them together for the first working version of our word game. Don't forget the exercises afterward—they're a challenge, I promise.

3.7   Project: 5x5 Solitaire

Since we handled input in the last chapter, there are just a few things our app has to do in order to be a full-fledged game. Let's start with the simplest pieces of the puzzle, then move on to the harder ones.

First things first: We'll need some words. Fortunately, the word list used at Scrabble tournaments, the Second Official Tournament and Club Word List (known to die-hard Scrabblers as the OWL2) is publicly available. One version of it is included with the book's code.[3]

We'll store this word list as an array. (Another data structure, such as a binary tree, would yield more efficient lookups, but I'll leave such optimizations in the reader's capable hands.) But first, we need to access it through Node's file system (fs) module:

Download Collections/5x5/game.coffee
```coffee
fs = require 'fs'
owl2 = fs.readFileSync 'OWL2.txt', 'utf8'
```

readFileSync simply reads the contents of a file into a string. The Sync suffix means that, unlike most Node.js I/O functions, this one blocks our program's execution until it completes. If we had some work to do in the background while the file is being read, then we'd use readFile with a callback instead, allowing us to continue processing while the operating system loads the file in the background.

Each line of our dictionary contains a word along with its definition and sentence part, such as this:

```
OD a hypothetical force of natural power [n -S]
```

As fascinating as this additional information is, we won't be using it. So let's use a simple regex to extract just the first word from each line:

---

3.   From http://www.zyzzyva.net/wordlists.shtml

```coffee
wordList = owl2.match /^(\w+)/mg
```

Then we cut the word list down to just those words that could plausibly fit on the grid:

```coffee
wordList = (word for word in wordList when word.length <= GRID_SIZE)
```

where `GRID_SIZE` is a constant that we've set to 5. (Of course, we could have done this more efficiently by tweaking our regex to only capture short words.)

Now we'll define a function to check whether a given word is valid:

```coffee
isWord = (str) ->
  str in wordList
```

Simple, right? Recall that `x in arr` just tells us whether the value `x` is in the array `arr`. (This differs from JavaScript's `in`, whose CoffeeScript analog is `of`.)

Generating a random grid will be a little bit trickier. First we'll need a way of generating random letters with the right probability distribution:

```coffee
# Probabilities are taken from Scrabble, except that there are no blanks.
# See http://www.hasbro.com/scrabble/en_US/faqGeneral.cfm
tileCounts =
  A: 9, B: 2, C: 2, D: 4, E: 12, F: 2, G: 3, H: 2, I: 9, J: 1, K: 1, L: 4
  M: 2, N: 6, O: 8, P: 2, Q: 1, R: 6, S: 4, T: 6, U: 4, V: 2, W: 2, X: 1
  Y: 2, Z: 1
totalTiles = 0
totalTiles += count for letter, count of tileCounts

# JavaScript hashes are unordered, so we need to make our own key array:
alphabet = (letter for letter of tileCounts).sort()

randomLetter = ->
  randomNumber = Math.ceil Math.random() * totalTiles
  x = 1
  for letter in alphabet
    x += tileCounts[letter]
    return letter if x > randomNumber
```

And now we generate the grid using a nested array comprehension:

```coffee
# grid is a 2D array: grid[col][row], where 0, 0 is the upper-left corner
grid = for x in [0...GRID_SIZE]
  for y in [0...GRID_SIZE]
    randomLetter()
```

We'll also want to pretty-print the grid (this is a word game, not Zork):

```coffee
printGrid = ->
  # Transpose the grid so we can draw rows
  rows = for x in [0...GRID_SIZE]
    for y in [0...GRID_SIZE]
      grid[y][x]
  rowStrings = (' ' + row.join(' | ') for row in rows)
  rowSeparator = ('-' for i in [1...GRID_SIZE * 4]).join('')
  console.log '\n' + rowStrings.join("\n#{rowSeparator}\n") + '\n'
```

Finally, we get to scoring. We'll start with a hash of Scrabble letter values and an array for keeping track of words that have already been used:

```coffee
# Each letter has the same point value as in Scrabble.
tileValues =
  A: 1, B: 3, C: 3, D: 2, E: 1, F: 4, G: 2, H: 4, I: 1, J: 8, K: 5, L: 1
  M: 3, N: 1, O: 1, P: 3, Q: 10, R: 1, S: 1, T: 1, U: 1, V: 4, W: 4, X: 8,
  Y: 4, Z: 10

moveCount = 0
score = 0
usedWords = []
```

Now for our actual scoring function. It takes a grid and the zero-indexed coordinates of the two tiles that were swapped and relies on a wordsThroughTile function that returns every word going through a particular tile of the grid:

```coffee
scoreMove = (grid, swapCoordinates) ->
  {x1, x2, y1, y2} = swapCoordinates
  words = wordsThroughTile(grid, x1, y1).concat wordsThroughTile(grid, x2, y2)
  moveScore = multiplier = 0
  newWords = []
  for word in words when word not in usedWords and word not in newWords
    multiplier++
    moveScore += tileValues[letter] for letter in word
    newWords.push word
  usedWords = usedWords.concat newWords
  moveScore *= multiplier
  {moveScore, newWords}
```

The wordsThroughTile function itself is trickier, since we need to go through a point in a 2-D array in four different directions while ensuring we don't go out of range. Let's look at the whole thing, and then I'll break it down a bit:

```coffee
wordsThroughTile = (grid, x, y) ->
  strings = []
```

```
  for length in [MIN_WORD_LENGTH..GRID_SIZE]
    range = length - 1
    addTiles = (func) ->
      strings.push (func(i) for i in [0..range]).join ''
    for offset in [0...length]
      # Vertical
      if inRange(x - offset, y) and inRange(x - offset + range, y)
        addTiles (i) -> grid[x - offset + i][y]
      # Horizontal
      if inRange(x, y - offset) and inRange(x, y - offset + range)
        addTiles (i) -> grid[x][y - offset + i]
      # Diagonal (upper-left to lower-right)
      if inRange(x - offset, y - offset) and
         inRange(x - offset + range, y - offset + range)
        addTiles (i) -> grid[x - offset + i][y - offset + i]
      # Diagonal (lower-left to upper-right)
      if inRange(x - offset, y + offset) and
         inRange(x - offset + range, y + offset - range)
        addTiles (i) -> grid[x - offset + i][y + offset - i]
  str for str in strings when isWord str
```

The outer loop for length in [MIN_WORD_LENGTH..GRID_SIZE] iterates through all game-level legal word sizes (in our case, from 2 to 5). We define an addTiles function that calls a given function with each integer value i from 0 to length -1 inclusive and then combines the results (which are characters) as a single string. The argument to addTiles is a function that, given i, returns the value of the tile i steps from the starting point in a certain direction.

The inner loop, for offset in [0...length], is there because we want all words *through* the given tile, not just those starting at it. For instance, in the vertical direction and when offset is 0, we get all the words going down from the given tile; when offset is 1, we get all the words going down from the tile above the given tile; and so on up to the edge of the grid.

Speaking of edges, those inRange checks ensure that both ends of a potential word are within the grid. If those tests pass, we send a callback to addTiles providing the tile value at each step i, and addTiles pushes the potential word to our strings list. Those items in strings that pass the isWord test get returned from the function.

(The use of function passing in addTiles may seem unnecessary, but it actually greatly simplified the wordsThroughTile function; in an earlier version, the same loop and join code was repeated four times!)

Whew! Now to initialize the game. We'll want to count words that are already in the grid at the start as used, which we can do by performing scoreMove on a "swap" of each tile with itself:

```coffee
console.log "Welcome to 5x5!"
for x in [0...GRID_SIZE]
  for y in [0...GRID_SIZE]
    scoreMove grid, {x1: x, x2: x, y1: y, y2: y}
unless usedWords.length is 0
  console.log """
    Initially used words:
    #{usedWords.join(', ')}
  """
console.log "Please choose a tile in the form (x, y)."
```

Finally, we plug in the input code from the last chapter, modifying our prompts to actually work:

```coffee
promptForTile1 = ->
  printGrid()
  console.log "Please enter coordinates for the first tile."
  inputCallback = (input) ->
    try
      {x, y} = strToCoordinates input
    catch e
      console.log e
      return
    promptForTile2 x, y
```

```coffee
promptForTile2 = (x1, y1) ->
  console.log "Please enter coordinates for the second tile."
  inputCallback = (input) ->
    try
      {x: x2, y: y2} = strToCoordinates input
    catch e
      console.log e
      return
    if x1 is x2 and y1 is y2
      console.log "The second tile must be different from the first."
    else
      console.log "Swapping (#{x1}, #{y1}) with (#{x2}, #{y2})..."
      x1--; x2--; y1--; y2--; # convert 1-based indices to 0-based
      [grid[x1][y1], grid[x2][y2]] = [grid[x2][y2], grid[x1][y1]]
      {moveScore, newWords} = scoreMove grid, {x1, y1, x2, y2}
      unless moveScore is 0
        console.log """
          You formed the following word(s):
          #{newWords.join(', ')}

        """
        score += moveScore
      moveCount++
```
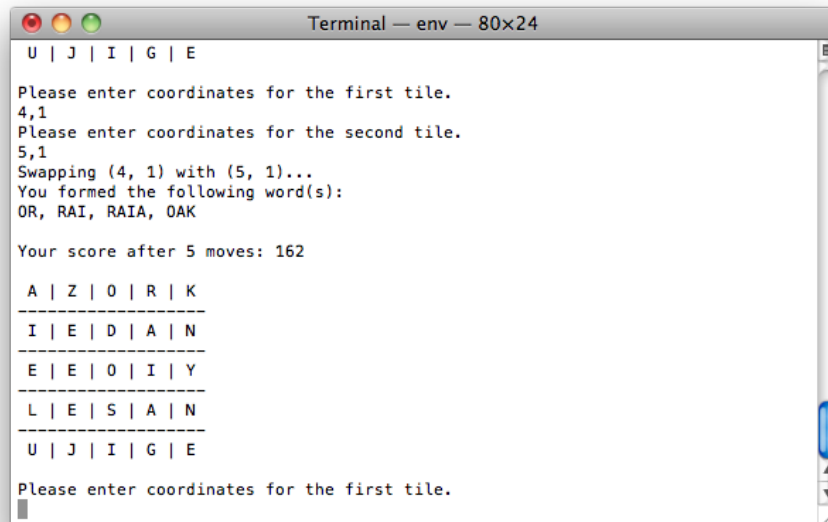
Figure 4—Playing with our command-line prompt

```
console.log "Your score after #{moveCount} moves: #{score}"
promptForTile1()
```

That's it! Run the game (coffee game.coffee), and you should see something like Figure 4, *Playing with our command-line prompt*, on page 55.

Of course, this program is far from a masterpiece. On the front end, the interface is a little...classical? And on the programmer side of things, the code is awfully disorganized. We'll be addressing the latter problem in the next chapter, and then we'll move on to use jQuery to turn our humble command-line game into a much more sophisticated browser-based experience. Finally, we'll use a Node server-side app to add multiplayer support.

The great thing about CoffeeScript is that we can reuse its code in all of these settings. Whether we're running the game in a browser or on a server, the code for scoring a move will remain the same. That convergence of client and server has all kinds of practical uses—think of form validation, for one. And with a thorough understanding of objects and loops, you're well on your way to mastering the language.

3.8 Beyond the Basics

After you've completed the exercises below, you can call yourself a yellow belt CoffeeScripter. How to get to black? Modules, grasshopper. While CoffeeScript gives us the power to express much with few words, some programs need many words. Better to use many small files, each with a well-defined purpose, than a few large files that make readers feel like rats in a maze.

3.9 Exercises

1. It's common to use slice to copy an entire array:

   ```coffee
   coffee> original = ['Mary', 'Poppins']
   coffee> copy = original[0..]
   coffee> copy[0] = 'Sh' + copy[0][1..]
   coffee> copy[1] = 'B' + copy[1][1..]
   coffee> original.join ' '
   Mary Poppins
   coffee> copy.join ' '
   Shary Boppins
   ```

   Explain how copy = original[0..] differs from copy = original.

2. One subtle difference between CoffeeScript's for...in loops and the C-style for loops in JavaScript is illustrated by this code. Explain why this code produces the following result:

   ```coffee
   once = ->
     if once.hasRun
       null
     else
       once.hasRun = true
       [1, 2, 3]
   console.log x for x in once()

   1
   2
   3
   ```

3. What is the output of this code?

   ```coffee
   for x in [1, 2]
     setTimeout (-> console.log x), 50
   ```

   Bonus question: Does it matter if the timeout is 0?

   For illumination, see *Scope in Loops,* on page 96.

4. Recall that 'foo' in arr will tell you whether the array arr contains the string 'foo' and whether 'bar' of obj will tell you if obj.bar exists. But how would you

check whether an arbitrary object contains a given value? Start with this:

```
objContains = (obj, val) ->
```

5. Let's say that we need to run a function at least once and then run it again repeatedly until a condition is met. In C/Java/JavaScript, we can write this:

```
do {
  user.harangue()
} while (!user.paidInFull)
```

The direct CoffeeScript equivalent would be the following:

```
user.harangue()
user.harangue() until user.paidInFull
```

But this violates the sacred principle of DRY (Don't Repeat Yourself). Define a doAndRepeatUntil function that takes two functions (equivalent to the loop body and the condition), so that we can instead write it this way:

```
doAndRepeatUntil user.harangue, -> user.paidInFull
```

6. For the project in this chapter, we set MIN_WORD_LENGTH as a constant. However, it makes more sense from a modularity standpoint to derive this from the dictionary we load into the game. How would you do that on one line using Math.min.apply and a list comprehension? (Math.min returns the argument given with the lowest value, such as Math.min 15, 16, 23, 42, 5, 8 is 5.)

Modules and Classes

In the preceding chapters, we learned how to make sentences out of the verbs and nouns of CoffeeScript. But strings of sentences do not always an elegant program make. What we need are higher-order abstractions. In particular, we need a clean way of describing *types* of objects.

In classical object-oriented languages like C++, there's a sharp distinction between objects and classes: an object is an instance of a class, inheriting its methods but storing its own data. Many dynamic languages, like Ruby, blur that distinction by allowing an object's methods to be modified at run-time. JavaScript obliterates the distinction: there are no classes. There are only prototypes, which make it easy for objects to share methods. For that reason, JavaScript is sometimes described as a prototype-based language.

This dynamic approach to sharing methods is powerful but has a cost in clarity. If you're reading code in a strictly class-based language and you want to find out which methods a particular object supports, you just have to look at the code that defines that object's class. But if you want to know which methods a JavaScript object has (without running the code), you have to track down every possible reference to that object or its prototype anywhere in the application.

Several approaches have popped up over the years for organizing JavaScript code into something resembling classes. Over time, a standard pattern evolved. That pattern provides the basis for CoffeeScript's class keyword. In this chapter, we'll learn how CoffeeScript classes work and then use them to modularize our code for 5x5.

But before we get into the workings of CoffeeScript classes, we need to talk a bit about how modules interact in CoffeeScript.

4.1   Modules: Splitting Up Apps

In a browser environment, JavaScript doesn't care about files. No matter how many files you may have, it's just a bunch of lines. So, if two files in an application happen to declare global-scope variables with the same name, well, they're just going to have to fight it out. May the best code win.

This is a huge problem for complex JavaScript apps. If a team is splitting up a project, how can they be sure that they aren't overwriting each other's variables? How can you be sure that open-source code you drop in doesn't conflict with your own?

The solution is *namespacing*. In JavaScript (and therefore CoffeeScript), every function has its own namespace. A variable declared in a function is never visible outside of that function. So, a common JavaScript convention is to make each file a *module* by wrapping it in a function, which is immediately executed. Server-side environments like Node.js, which implement the CommonJS specification, always treat each file as a separate module.

As mentioned in *[JavaScript, Under Wraps,](#)* on page 8, the CoffeeScript compiler wraps each .coffee file in an anonymous function wrapper, unless it's invoked with the --bare flag. CoffeeScript also prevents you from declaring global variables by mistake, which in JavaScript is as easy as forgetting the var keyword. So the question is, how do you share data between modules?

The answer is you attach them to an existing global variable. One option is to use the *root object*, which is the only object whose properties can be referenced without qualification. In a browser environment, the root object is window. In Node, it's global.

In fact, all of the globals you're used to dealing with are actually properties of the root object. For instance, parseInt is actually window.parseInt/global.parseInt, and Math is window.Math/global.Math. Even the objects that define the built-in types, like String, are actually window.String/global.String.

In JavaScript, attaching variables to the root object is easy—in fact, it's commonly done by mistake when the var keyword is omitted. In CoffeeScript, on the other hand, you need to be explicit:

```
root = global ? window

# file1.coffee
root.emergencyNumber = 911

# file2.coffee
console.log emergencyNumber                # '911'
emergencyNumber is root.emergencyNumber  # true
```

The first line defines root to be global if it exists, and window otherwise, ensuring compatibility with both Node and browser environments.

Node has another special object called exports. Usually, you'll want to use that instead of global (more on that in Section 6.2, *Modularizing Code with 'exports' and 'require',* on page 92). I used global ? window in the example above rather than exports ? window so that the code looks the same in every environment; an alternative would be to use a library, like RequireJS, that allows you to modularize code the same way in every environment without using objects like global and window that are available everywhere by default.[1]

Of course, it's considered poor practice to put every little thing in the global namespace. Instead, it's much more natural to package variables into cleanly referenced objects. Here's an example:

```
root = global ? window
root.httpCodes =
  movedPermanently: 301
  pageNotFound:     404
  serverError:      500
```

Once this module has run, other modules can reference httpCodes.pageNotFound, for instance.

4.2 The Power of Prototypes

Before we move on to classes, it's important to understand how prototypes work. If you're fluent in JavaScript, this section will just be a refresher.

A prototype is an object whose properties are shared by all objects that have that prototype. An object's prototype can usually be accessed using the aptly-named prototype property, though there are exceptions.[2]

However, you can't just go and write A.prototype = B. Instead, you need to use the new keyword, which takes a function called a constructor and creates an object that "inherits" the constructor's prototype. Here's a quick example:

```
Boy = ->      # by convention, constructor names are capitalized
Boy::sing = -> console.log "It ain't easy being a boy named Sue"
sue = new Boy()
sue.sing()
```

Here, Boy::sing is shorthand for Boy.prototype.sing. The :: symbol is to prototype as @ is to this.

---

1.   http://requirejs.org/
2.   http://javascriptweblog.wordpress.com/2010/06/07/understanding-javascript-prototypes/

> **Caution: Globals and Implicit Declarations**
>
> While you can get globals without referencing the root object, you can't set them that way. This is an easy mistake to make:
>
> ```
> root = global ? window
> # file1.coffee
> root.dogName = 'Fido'
> dogName is root.dogName  # true
>
> # file2.coffee
> console.log dogName       # undefined
> dogName = 'Bingo'
> dogName is root.dogName  # false
> ```
>
> Why is `dogName` undefined at the start of the second module? Because a variable with the same name is assigned later on. CoffeeScript interprets that as a new variable declaration, and all variable declarations in CoffeeScript are automatically moved to the top of the scope.
>
> So remember: always, always, *always* use the root object when setting a global variable! Statements of the form `x = y` will never change the value of `x` in other modules.

The output looks like this:

```
It ain't easy being a boy named Sue
```

This is the result because `sue` inherits the properties of `Boy.prototype`. Pretty cool! But how does it work?

When we use `new`, several things happen: a new object is created, that object is given the prototype from the constructor, and the constructor is executed (in the new object's context). So let's say we want every new object to store its name and announce the existing number of gifts:

```
Gift = (@name) ->
  Gift.count++
  @day = Gift.count
  @announce()

Gift.count = 0
Gift::announce = ->
  console.log "On day #{@day} of Christmas I received #{@name}"

gift1 = new Gift('a partridge in a pear tree')
gift2 = new Gift('two turtle doves')
```

Here's the output:

```
On day 1 of Christmas I received a partridge in a pear tree
On day 2 of Christmas I received two turtle doves
```

> ### Prototypes, Precedence, and 'hasOwnProperty'
>
> When an object inherits properties from a prototype, changes to the prototype will
> change the inherited properties as well:
>
> Download Classes/Raven.coffee
> ```coffee
> Raven = ->
> Raven::quoth = -> console.log 'Nevermore'
> raven1 = new Raven()
> raven1.quoth()     # Nevermore
>
> Raven::quoth = -> console.log "I'm hungry"
> raven1.quoth()     # I'm hungry
> ```
>
> Properties attached directly to objects take precedence over prototype properties.
> So we can remove that ambiguity by writing this:
>
> Download Classes/Raven.coffee
> ```coffee
> raven2 = new Raven()
> raven2.quoth = -> console.log "I'm my own kind of raven"
> raven1.quoth()     # I'm hungry
> raven2.quoth()     # I'm my own kind of raven
> ```
>
> To check whether a property is attached to an object directly or inherited from a
> prototype, use the hasOwnProperty function:
>
> Download Classes/Raven.coffee
> ```coffee
> console.log raven1.hasOwnProperty('quoth')  # false
> console.log raven2.hasOwnProperty('quoth')  # true
> ```

Each time the Gift constructor runs, it does four things: assigns the given
name to @name (using the argument shorthand), increments the count prop-
erty on the Gift constructor, copies that value to @day, and runs the @announce
function inherited from the prototype. The important thing to notice here
is that all of the functions on the new object run in the context of the object.

This is all well and good, but it's a bit messy, isn't it? Shouldn't there be a
clearer way of distinguishing constructor properties (like Gift.count) from
prototype properties (like Gift::announce)? And of distinguishing constructors
from ordinary functions? Well, as a matter of fact, there is.

4.3 Classes: Functions with Prototypes

CoffeeScript's class declaration syntax looks just like its object declaration
syntax. That's no coincidence; when you're defining a class, you're defining
an object. Specifically, you're defining a prototype. The only class property
that isn't part of the prototype is the constructor function, if you define one.

Let's look at an example, illustrating the well-known fact that the trouble a
tribble makes is directly proportional to the number of existing tribbles:

```coffee
class Tribble
  constructor: ->
    @isAlive = true
    Tribble.count++

  # Prototype properties
  breed: -> new Tribble if @isAlive
  die: ->
    Tribble.count-- if @isAlive
    @isAlive = false

  # Class-level properties
  @count: 0
  @makeTrouble: -> console.log ('Trouble!' for i in [1..@count]).join(' ')
```

There's a lot of new syntax here. Let's go through this one piece at a time.

Each time a new tribble is created, Tribble.count is increased by one. (We can call it @count here because the value of this in the class body is the class itself.) When Tribble.makeTrouble() is called, it'll display "Trouble!" Tribble.count times.

Let's test this:

```coffee
tribble1 = new Tribble
tribble2 = new Tribble
Tribble.makeTrouble()   # "Trouble! Trouble!"
```

Notice that Tribble.count is referred to as @count in the Tribble class context but not within Tribble methods. This is a little baffling at first, but remember that there are three objects we're dealing with here: the Tribble object itself (which is actually the constructor function), Tribble.prototype, and the Tribble instance. By default, Tribble properties (other than constructor) are attached to the prototype. When we use the @ prefix, we're insisting that we want to attach the property to the class object itself.

Because the functions attached to the prototype are invoked in the context of the individual object (as is the constructor), variables prefixed with @ within those functions are references to instance properties. This is why we define @isAlive in the constructor: we need to attach a separate @isAlive property to each instance. That lets us do this:

```coffee
tribble1.die()
Tribble.makeTrouble()   # "Trouble!"
```

Killing tribble1 off again would have no effect, thanks to the if @isAlive check. And as we know, tribbles are born pregnant, so it won't be long before the species repopulates our program:

```
Download Classes/Tribble.coffee
tribble2.breed().breed().breed()
Tribble.makeTrouble()   # "Trouble! Trouble! Trouble! Trouble!"
```

4.4 Inheritance with 'extends'

So far, we've talked about how prototypes make it easy to share functionality across a set of objects and how CoffeeScript's classes provide a useful syntax for bundling prototype properties together. And if that were all classes did, they'd be mildly useful. But where classes really shine is when we want to use inheritance.

JavaScript supports inheritance through something called "prototype chains." Let's say that A's prototype, B, has its own prototype, C. Then we write this:

```
a = new A
console.log a.flurb()
```

First, the runtime checks to see if the particular A instance, a, has a property flurb; if not, it checks A's prototype (B); and if that's still no dice, it checks B's prototype (C). In short, it's traversing the prototype chain.

What happens if C has no flurb, either? Then the runtime checks the default object prototype (that is, the prototype of {}). So, every prototype inherits from {}'s prototype, but there may be other prototypes in between.

All of this assigning prototypes to prototypes to prototypes gets a little messy. That's where CoffeeScript's extends keyword comes in.

Let's make a declaration:

```
class B extends A
```

Then, B's prototype inherits from A's prototype. In addition, A's class-level properties are copied over to B. So if we left the definition of B alone, B instances would have exactly the same behavior as A instances. (There is one exception: B.name would be 'B' while A.name would be 'A'—name is a special property).

Now let's look at a slightly deeper example:

```
class Pet
  constructor: -> @isHungry = true
  eat: -> @isHungry = false

class Dog extends Pet
```

```
eat: ->
  console.log '*crunch, crunch*'
  super()
fetch: ->
  console.log 'Yip yip!'
  @isHungry = true
```

The constructor from `Pet` is inherited by `Dog`, which means that dogs start out hungry. When a dog eats, it makes some noises and then invokes super(), which means "call the method of the same name on the parent class." (More precisely, it means Pet::eat.call this.) Then the dog is no longer hungry.

If a constructor is defined on the child class, then it overrides the constructor from the parent class. It can invoke the parent constructor at any time using super(). It's usually wise to call super() (or, more likely, super—see *'super' Isn't 'super()', on page 67*) at the start of a subclass constructor.

Believe it or not, you now know everything there is to know about classes. As with everything in CoffeeScript, the syntax may be distant from JavaScript, but the translation is straightforward. If you're a fan of classical OOP (object-oriented programming) methodology, this next section's for you.

Polymorphism and Switching

One great use of classes is *polymorphism*, which is a fancy object-oriented programming term for "a thing can be a lot of different things, but not just *any* thing." Here's a classic example:

```
class Shape
  constructor: (@width) ->
  computeArea: -> throw new Error('I am an abstract class!')

class Square extends Shape
  computeArea: -> Math.pow @width, 2

class Circle extends Shape
  radius: -> @width / 2
  computeArea: -> Math.PI * Math.pow @radius(), 2

showArea = (shape) ->
  unless shape instanceof Shape
    throw new Error('showArea requires a Shape instance!')
  console.log shape.computeArea()

showArea new Square(2)  # 4
showArea new Circle(2)  # pi
```

Notice that the `showArea` function checks that the object passed to it is a `Shape` instance (using the `instanceof` keyword). But it doesn't care what kind of shape it's been given; both `Square` and `Circle` instances will work. While this is a

'super' Isn't 'super()'

What's wrong with this code?

```coffeescript
class Appliance
  constructor: (warranty) ->
    warrantyDb.save(this) if warranty

class Toaster extends Appliance
  constructor: (warranty) ->
    super()
```

When we create a new `Toaster`, `super()` will invoke the parent constructor without passing along the `warranty` argument, which means that the toaster won't be saved in the warranty database.

We could fix this by writing `super(warranty)`, but there's a shorthand we can use instead: `super`. With no parentheses or arguments, `super` will pass on every one of the current function's arguments. If you're a Rubyist, this will seem familiar. If not, just think of `super` as a greedy, greedy keyword—if you don't tell it which arguments you want it to pass along, it'll take 'em all.

trivial example, it's not hard to imagine a rich geometry library that takes this approach.

If we didn't use the `instanceof` check, that would be known as "duck typing" (as in, "If it looks like a duck…"). If the target object doesn't have a `computeArea` function, then we'll get a meaningful error message anyway. Duck typing is great, but there are times when you want to be sure that a particular object is what you think it is.

A common idiom in more classical object-oriented languages is to use polymorphism with `switch`. We haven't talked about CoffeeScript's `switch` yet, and there are a number of differences between it and JavaScript's: first, there's an implicit break between clauses to prevent unintended "fall-through"; second, the result of the switch is returned. (When the return value is used, break and return are not allowed. If you try, you'll get `Syntax-Error: cannot include a pure statement in an expression`. This is jargon for saying that it doesn't make sense to say `a = return x`, so the compiler won't allow it as a possibility.)

CoffeeScript also makes several syntactic changes, in part to remind JavaScripters of these hidden differences: `when` is used instead of `case` and `else` instead of `default`. (The keywords are borrowed from Ruby, where the case structure has similar semantics.) A single `when` can be followed by several potential matches, delimited by commas. Also, instead of `:`, those match clauses are separated from their outcomes by indentation (or `then`).

Here's how it all comes together in a simple factory function:

```coffee
requisitionStarship = (captain) ->
  switch captain
    when 'Kirk', 'Picard', 'Archer'
      new Enterprise()
    when 'Janeway'
      new Voyager()
    else
      throw new Error('Invalid starship captain')
```

That's it for our coverage of modules and classes. Just remember: Coffee-Script certainly doesn't require you to use classes or classical object-oriented design patterns—most JavaScript developers do perfectly fine without either, after all—but for some applications, classes feel like a natural fit.

Speaking of which, remember that mess of code from the project in the last chapter? Let's see what we can do about it.

4.5 Project: Refactoring 5x5

Classes give a nice clean way to reorganize the code we've written so far, to encourage modularity and extensibility. Let's create three classes:

1. Dictionary, which can find valid words on the grid
2. Grid, which manages the letter tiles
3. Player, which keeps track of a player's score

We'll save these classes in three different .coffee files. Also, we'll make these classes compatible with all major browser environments as well as with Node.js, paving the way for the jQuery version of the game in the next chapter. To make the game playable under Node, we'll have a console.coffee file that provides a prompt and requires the three classes.

The Dictionary Class

Because we want to support browsers, it makes sense to convert our word list from a text file to JavaScript code that can be loaded directly. I wrote a short script to do just that:

```coffee
Download Classes/5x5/convert.coffee
fs = require 'fs'
owl2 = fs.readFileSync 'OWL2.txt', 'utf8'
wordList = owl2.match /^(\w+)/mg
fileContents = """
  root = typeof exports === "undefined" ? window : exports;
  root.OWL2 = ['#{wordList.join "',\n'"}']
"""
fs.writeFile 'OWL2.js', fileContents
```

The triple-quote """ syntax, which should be familiar to Python coders, is called a *heredoc string.* It allows you to write multiline strings in a human-readable format. You can also use three single quotes, ''', to delimit a heredoc string. The difference between """ and ''' is the same as the difference between " and ': The former allows interpolation, while the latter does not.

After running the script, the resulting OWL2.js looks like this:

```
root = typeof global === "undefined" ? window : global;
root.OWL2 = ['AA',
...
'ZOOGEOGRAPHICAL'];
```

(I've omitted the middle 178,687 lines for the sake of brevity.)

Now, in order to decouple Dictionary from Node, let's have the word list and the game's grid get passed in to Dictionary's constructor:

Download Classes/5x5/Dictionary.coffee
```
class Dictionary
  constructor: (@originalWordList, grid) ->
    @setGrid grid if grid?
```

Note the implicit assignment of the first argument to @originalWordList (a feature we covered back in Section 2.4, *Property Arguments (@arg),* on page 24).

When a new game starts, we'll call setGrid ourselves with the new grid. This grid could be of a different size, so we'll copy and filter our original word list each time (slice(0) is a well-known trick for copying JavaScript arrays):

Download Classes/5x5/Dictionary.coffee
```
setGrid: (@grid) ->
  @wordList = @originalWordList.slice(0)
  @wordList = (word for word in @wordList when word.length <= @grid.size)
  @minWordLength = Math.min.apply Math, (w.length for w in @wordList)
  @usedWords = []
  for x in [0...@grid.size]
    for y in [0...@grid.size]
      @markUsed word for word in @wordsThroughTile x, y
```

Notice that we also reset the usedWords list here. We'll want a way of indicating that a word has been used:

Download Classes/5x5/Dictionary.coffee
```
markUsed: (str) ->
  if str in @usedWords
    false
  else
    @usedWords.push str
    true
```

Let's provide one method that says whether a word is valid and another that says whether a word is usable at this point in the game:

```
isWord: (str) -> str in @wordList
isNewWord: (str) -> str in @wordList and str not in @usedWords
```

Here's the hard part: given a pair of coordinates, we'd like to be able to find all words that go through that tile of the Grid instance. I'll spare you the excerpt, since it's basically the same as our wordsThroughTile function from the last chapter.

Now in order to make the class accessible, let's make it a global:

```
root = exports ? window
root.Dictionary = Dictionary
```

The Grid Class

Before we define the Grid class, let's put some other variables in grid.coffee, giving them module scope:

```
tileCounts =
  A: 9, B: 2, C: 2, D: 4, E: 12, F: 2, G: 3, H: 2, I: 9, J: 1, K: 1, L: 4
  M: 2, N: 6, O: 8, P: 2, Q: 1, R: 6, S: 4, T: 6, U: 4, V: 2, W: 2, X: 1
  Y: 2, Z: 1

totalTiles = 0
totalTiles += count for letter, count of tileCounts
alphabet = (letter for letter of tileCounts).sort()

randomLetter = ->
  randomNumber = Math.ceil Math.random() * totalTiles
  x = 1
  for letter in alphabet
    x += tileCounts[letter]
    return letter if x > randomNumber
```

When we instantiate a grid, let's generate its initial tiles matrix automatically:

```
class Grid
  constructor: ->
    @size = size = 5
    @tiles = for x in [0...size]
      for y in [0...size]
        randomLetter()
```

Now we'll define some simple functions that allow us to check whether an (*x*, *y*) pair is out of range, to swap two tiles (given an object with two coordinate pairs), and to get a transposed version of the grid (an array of rows rather than columns):

```
inRange: (x, y) ->
  0 <= x < @size and 0 <= y < @size

swap: ({x1, y1, x2, y2}) ->
  [@tiles[x1][y1], @tiles[x2][y2]] = [@tiles[x2][y2], @tiles[x1][y1]]

rows: ->
  for x in [0...@size]
    for y in [0...@size]
      @tiles[y][x]
```

I'll spare you the two lines where we make the class a global.

The Player Class

Each Player instance should be given its own name and, optionally, an initial grid. (As with Dictionary, we'll use setGrid when a new game starts.) The player starts with a score of 0, naturally:

```
class Player
  constructor: (@name, dictionary) ->
    @setDictionary dictionary if dictionary?

  setDictionary: (@dictionary) ->
    @score = 0
    @moveCount = 0
```

Let's provide a way for a player to make a move:

```
makeMove: (swapCoordinates) ->
  @dictionary.grid.swap swapCoordinates
  @moveCount++
  result = scoreMove @dictionary, swapCoordinates
  @score += result.moveScore
  result
```

The Console.Coffee Interface

All of the code from the last chapter that we didn't refactor into other classes—that is, everything related to command-line IO—is in console.coffee. I won't repeat the recycled code here, but the important part is these first four lines:

```
{Dictionary} = require './Dictionary'
{Grid} = require './Grid'
{Player} = require './Player'
{OWL2} = require './OWL2'
```

The ./ prefix in front of each filename tells Node to load the file from the current path. An alternative approach, added in Node 0.4, would be to create a node_modules directory in console.coffee's working path and then put the files in there.[3] Then no prefix would be needed.

And we're done! Try it for yourself:

```
$ coffee console.coffee
Welcome to 5x5!
```

We've refactored our old, haphazard code into four nice, clean modules, three of which we'll be reusing in the next two chapters, first in a browser, then on a server. We saved ourselves a ton of work by separating the reusable game logic code from the command-line stuff, which (sigh of relief) we shall never speak of again. Onward!

4.6 Just a Spoonful of Sugar

In this chapter, we've learned about two ways that CoffeeScript supports modular programs: First, individual files ("modules") are isolated from each other, except when variables are explicitly exported. And second, functions and data can be combined into classes.

We applied both techniques to clean up our quick-and-dirty version of 5x5 from the last chapter. The result? Not only is the code more readable, but it's also much easier to refactor, as we'll see in the next chapter when we transform our old-school text-based program into a newfangled web app. And we'll do it with a little help from JavaScript's good pal, jQuery.

4.7 Exercises

1. Explain this output, and fix the code so that the old aphorism is displayed twice:

   ```
   root = global ? window

   root.aphorism = 'Fool me 8 or more times, shame on me'

   do restoreOldAphorism = ->
     aphorism = 'Fool me once, shame on you'
     console.log aphorism
   ```

---

3. http://nodejs.org/docs/v0.4.8/api/modules.html

```
console.log aphorism

Fool me once, shame on you
Fool me 8 or more times, shame on me
```

2. As everyone knows, the Genie Workers International Union mandates a limit of three wishes across all genies. The following code is designed to enforce that rule, but it has a flaw:

```
Genie = ->
Genie::wishesLeft = 3

Genie::grantWish = ->
  if @wishesLeft > 0
    console.log 'Your wish is granted!'
    @wishesLeft--
```

What's wrong with this code, and how would you fix it?

3. The prototype property is not, unfortunately, the "true" prototype of an object. The good news is that you can get the true prototype with __proto__. The bad news is that __proto__ isn't supported in all JS environments; in particular, it's not available under Internet Explorer.

Still, it's useful to use __proto__ to illustrate the rules we talked about regarding prototype inheritance:

```
class Season
class Spring extends Season

(new Season).__proto__.__proto__
(new Spring).__proto__.__proto__.__proto__
```

In an environment that supports it (such as Node), what is the value of the two prototypes shown here?

4. We haven't discussed bound functions on classes, so perhaps a demonstration is in order. What output do you expect the following code to generate?

```
(window ? global).property = 'global context'
@property = 'surrounding context'
class Foo
  constructor: -> @property = 'instance context'
  bar: => console.log @property

foo = new Foo
bar = foo.bar
foo.bar()
bar()
```

Why might you prefer to define bar using -> instead?

Web Interactivity with jQuery

Once upon a time, programmers wrote web applications in pure JavaScript, unencumbered by heavy frameworks. Adventurous programmers roamed these fertile lands, devising exciting new functionality for their apps. But all was not well, for every bold feat performed by these dynamic knights had to be done twice (at least): once in Netscape (and its open-source descendants) and once in Internet Explorer.

To alleviate this burden, programmers began to write functions that allowed them to avoid repetition in their code. These functions became ever more numerous and complex, fusing into libraries made up of thousands of lines of code. Soon, warring factions began to form around them. Among these were MooTools, Prototype, Dojo, and YUI.

But as the years passed, one library became so popular as to become the de facto standard: jQuery.[1] Initially released by the then twenty-two-year-old John Resig in 2006, jQuery is now used by nearly one-third of the ten thousand most-visited websites,[2] and Resig has become one of the world's most famous programmers. Recent spin-offs, including jQuery Mobile, have brought jQuery's familiar syntax far beyond the desktop-oriented web.[3] Although other libraries are still widely used, jQuery truly stands apart.

In this chapter, we'll go through the basics of using jQuery to manipulate the elements of a web page and respond to events. (If you're familiar with jQuery on JavaScript, this will just be a review with minor syntactic adjustments.) At the end of the chapter, we'll use these newfound superpowers to turn our quaint little word game into a full-fledged browser-based app.

---

1. http://jquery.com/
2. http://trends.builtwith.com/javascript/JQuery
3. http://jquerymobile.com/

5.1   The Tao of jQuery

Each JavaScript library has its own particular philosophy, and jQuery's is a natural fit for JavaScript (and CoffeeScript, too!).

jQuery doesn't modify JavaScript's built-in object prototypes, such as `String` and `Array`. That's the fundamental difference between jQuery and Prototype, the second-most popular JavaScript library. Prototype can do some amazing things, but using it means potentially breaking any code that wasn't written with Prototype in mind. For instance, a library might use `for x of arr` to iterate over an array's indices, not realizing that this will also include the properties of `Array.prototype` added by Prototype.js. (It's partly for this reason that `for...in` is preferred to `for...of` for array iteration. `for own...of` would also work in this case, because it skips properties that fail the `hasOwnProperty` test.)

Instead, jQuery's power is safely tucked away in one object, `jQuery`, which is normally aliased as `$`. We'll use the `$` alias throughout this book, but be aware that it's possible to disable it, in case another library wants to use that variable name.[4]

The jQuery object lets you do everything from animating transitions to adding event callbacks to pulling data from the server. And that's without even drawing on the thousands of free plugins available.[5] Of course, this chapter and the next are too brief to show you everything jQuery is capable of, but it should give you enough of an understanding to avoid common pitfalls while exploring jQuery's vast capabilities, which are exquisitely documented at http://api.jquery.com.

5.2   Manipulating the DOM

If you're familiar with raw JavaScript in a web context, you know that HTML tags like <p> define DOM elements, which JavaScript code can read, modify, and create from whole cloth.

jQuery wraps these elements with its own objects, which provide more convenient functionality (and much more cross-browser consistency) than accessing the elements directly. To obtain a jQuery object, you usually use a *selector*, a string passed to the jQuery object. We'll learn more about selectors in the next section, Section 5.3, *Getting Selective,* on page 77; for now, all you need to know is that they're a superset of CSS. So to select a DOM element with the ID `pikachu` and store the result in a jQuery object `$pokemon`

---

4.   http://api.jquery.com/jQuery.noConflict/
5.   http://plugins.jquery.com/

(it's a common stylistic convention to prefix all jQuery object names with $), you would write this:

```
$pokemon = $('#pikachu')
```

Once you've got a jQuery object, you have a massive arsenal of functions at your disposal. As a rule, these functions act on all of the matched elements. So if you have two paragraphs and you write the following, both paragraphs will be faded out:

```
$('p').fadeOut()
```

An exception to this rule is "getter" functions. These only act on the first matched element (with the notable exception of text, which combines the textual context of all matched elements into a single string). Consider this:

```
sonnet = $('p').html()
```

This code will only return the HTML content of the first paragraph in your document. By contrast, "setters" act on all matched elements. Often, these have the same name, like so:

```
sonnet = $('p').html()
$('p').html sonnet
```

This will first read the HTML of the first paragraph, then set the HTML of *all* paragraphs to match. (The text getter would strip out HTML tags, as well as concatenating the content of all matched elements. Think carefully about whether text or html is more appropriate when extracting content.[6]) Of course, this can be condensed to a (somewhat baffling) one-liner:

```
$('p').html $('p').html()
```

The moral of this story is this: jQuery and CoffeeScript can be used to write elegant, readable code—or you can just make runnable ASCII art. With great power comes great responsibility.

5.3 Getting Selective

When you pass a string to jQuery, it's interpreted as a *selector*, and an object containing the matching elements is returned. The selector syntax is designed to mirror and extend CSS's selector syntax. By itself, an HTML tag type (such as 'p') will match all elements of that type. An identifier preceded by # is a unique ID, and one preceded by . is a class name.

---

6.   http://stackoverflow.com/questions/1910794/jquery-text-vs-html

Where'd Those DOM Elements Go?

Every jQuery object you create is just an ordered list of DOM elements wrapped up with a neat feature-rich bow. To get those elements out, the official docs suggest that you use the get:

```
pikachu = $('#pikachu').get(0)
```

But a dirty little secret of jQuery objects is that they store their DOM elements by numerical index, allowing array-style access:

```
pikachu = $('#pikachu')[0]
```

You can even use array functions like length and slice (though not push or concat—instead, try add).

Of course, unless you know what you're doing, you should resist the urge to use DOM elements directly. If your Firefox-tested DOM code crashes in IE7, you have only yourself to blame.

You can list multiple selectors, separated by commas, to match all of them. For instance, $('a, button, .link') would match all a elements, all button elements, and all elements with the link class. The same set of elements could be obtained with $('a').add($('button')).add($('.link')).

When multiple identifiers are joined by spaces, they match descendants. For instance, $('#header img') matches all img tags that are within the element with the unique ID header. The same selection could be done through chaining with the find method: $('#header').find('img'). If you only wanted images that are the immediate children of header, you could either use the CSS2 syntax $('#header > img') or the chain $('#header').children('img').

In addition to these CSS selectors, there are several special modifiers added by jQuery. For instance, to match only odd table rows, we could write $('tr:odd'). To match only list items that contain links, we could write $('li:has(a)'). Matching all checked checkboxes is as easy as writing $(':checked').

There are two important points to remember about selected elements in jQuery. First, the selection is performed only once—selectors are not "live" (except in the appropriately named live method and the functionally similar delegate, which run the given selector as needed every time an event is fired). Second, there's no distinction in jQuery between a single element and a collection of elements; a single element is just a collection of size 1. So for instance, if the first div on your page has the id header, then $('#header'), $('div:first'), and $('div').first() are all equivalent.

Feeling overwhelmed yet? Relax! The important thing is to remember the general concept: you pass a string to the $ function; you get a set of elements out. We'll review some of these selection techniques in the exercises. You can find a thorough list of selector strings jQuery understands at http://api.jquery.com/category/selectors/.

5.4 Reacting to Events

We've seen how jQuery makes it easy to grab, change, and even add elements to the page. But jQuery, in its heart of hearts, has seen fit to bestow still one more miracle upon us mortals: simple event binding.

How simple? Well, let's say that we want the big headline on our page to gain an exclamation point every time it gets clicked:

```
$('h1').click -> $(this).html $(this).html() + '!'
```

This code selects all h1 elements on the page, then binds a click event handler to each of them. When that click handler is triggered, it appends '!' to the clicked element's contents.

Event callbacks are called in the context of the DOM element that triggered the event. Often, we want to jQuerify the context element, as we do in the example above, by writing $(this).

What happens if we bind multiple event handlers of the same type to the same elements?

```
$('h1')
  .click(-> $(this).html $(this).html() + '!')
  .click(-> alert $(this).text())
```

Each of those click calls is made on the same object; see *jQuery Chain Gang, on page 80*. Note that whitespace here is not significant; it's purely a stylistic convention. We could easily collapse this code into a single line: $('h1').click(-> …).click(-> …). That means that while the parentheses around the callback on the last line are optional, those around 'h1' and the callback on the second line are not.

All of the event handlers will run in the order in which they were attached. So when we click a heading in this example, '!' will be added to its contents and the resulting text will be (irritatingly) displayed in an alert box.

You can unbind events using—wait for it—unbind. Calling $elem.unbind() will unbind *everything* attached to $elem, while $elem.unbind 'click' will just unbind the click events. (What if you want to unbind a specific event? See the exercises at the end of this chapter.)

> **jQuery Chain Gang**
>
> Check this out:
>
> ```
> $('#logo')
>   .css(fontSize: 64)
>   .hover(-> $(this).css(fontWeight: 'bold'))
>   .click(-> alert 'How dare you click the mighty logo!')
> ```
>
> Here we're using chaining—nearly all jQuery methods return the object they were called on, so the above is equivalent to this:
>
> ```
> $logo = $('#logo')
> $logo.css fontSize: 64
> $logo.hover -> $logo.css fontWeight: 'bold'
> $logo.click -> alert 'How dare you click the mighty logo!'
> ```
>
> Generally, chaining is a good thing—it tends to improve readability by reducing repetition, and it lets code minify a tad better. But be warned: some jQueryers go "chain crazy," turning their entire app into one long, meandering chain. Use it in moderation.

There's one last thing you'll need to know before we get to our project. Remember how I said that selectors aren't "live"? Well, there's one exception to that rule: when you use the aptly-named live method to bind an event handler. Let's do an example:

```
$('#oldSpiceGuy').live 'click', -> alert "I'm on a horse."
$('body').html '<p id="oldSpiceGuy">The man your man could smell like</p>'
```

Even though $('#oldSpiceGuy') doesn't match anything in the first line, the event handler will work. You see, unlike all other jQuery methods, live doesn't care about the elements that were matched when you made the selection: it only cares about the selector string itself. (jQuery makes this string available via the .selector property.)

A full explanation of how live works is beyond the scope of this book, but it has to do with "event bubbling."[7]

5.5 Project: Browser-Based 5x5

We're going to create a browser-based version of the game from the last chapter using the same three class files: Grid.coffee, Dictionary.coffee, and Player.coffee. These encapsulate our game state and logic. We'll add one new

---

7. http://www.alfajango.com/blog/the-difference-between-jquerys-bind-live-and-delegate/

CoffeeScript file, jq5x5.coffee, which will define our interface in conjunction with index.html and style.css.

index.html

For a production deployment, we'd want to compile our CoffeeScript files to JavaScript, then minify and concatenate that code into a single tidy package. However, for our purposes, in-browser compilation will do fine. We'll use the coffee-script.js from http://jashkenas.github.com/coffee-script/extras/coffee-script.js to do just that. Let's include it, and let's throw in the old OWL2 word list and jQuery while we're at it:

```
Download jQuery/5x5/index.html
<script type="text/javascript" src="coffee-script.js"></script>
<script type="text/javascript" src="OWL2.js"></script>
<script type="text/javascript" src="jquery-1.5.2.min.js"></script>
```

And now let's include the source files using the special type="text/coffeescript" attribute:

```
Download jQuery/5x5/index.html
<script type="text/coffeescript" src="Grid.coffee"></script>
<script type="text/coffeescript" src="Dictionary.coffee"></script>
<script type="text/coffeescript" src="Player.coffee"></script>
<script type="text/coffeescript" src="jq5x5.coffee"></script>
```

Oh, and we'll want to include our style.css:

```
Download jQuery/5x5/index.html
<link rel="stylesheet" type="text/css" href="./style.css" />
```

That's it for the head of the page. Now we just need a body with three elements—a p named message (to show messages to the players), a div named grid (to contain the tiles), and a table named scores (guess):

```
Download jQuery/5x5/index.html
<body>
  <p id="message"></p>
  <div id="grid"></div>
  <table id="scores">
    <tr>
      <th id="p1name"></th>
      <th id="p2name"></th>
    </tr>
    <tr>
      <td id="p1score"></td>
      <td id="p2score"></td>
    </tr>
  </table>
</body>
```

Now open up index.html in your favorite browser. (If that happens to be Chrome, you'll have to start the browser using the -allow-file-access-from-files flag; otherwise, Chrome's strict security policies will prevent the CoffeeScript files from being loaded. This issue only occurs when loading local files directly in the browser, such as URLs starting with file://. In the next chapter, we'll use Node.js to serve our site from localhost:// instead.)

style.css

In principle, everything that can be done in a CSS file can be done from code by using jQuery's css method, for example. However, it's often much more efficient to use static styles. For instance, if we want all of the text on the page to be dark gray, we can simply write this:

```
body { color: #333 }
```

In order to do this without a stylesheet, we'd have to call $elem.css 'color', '#333' every single time we created a new element with text. Ugh.

To save us that trouble, let's decide now how we're going to lay out our page. We'll represent our 5x5 grid of tiles as 5 ul rows, each with 5 lis, within the #grid div. We want #grid to be centered and have a large, monospaced font:

Download jQuery/5x5/style.css

```
#grid {
  position: relative;
  text-align: center;
  width: 480px;
  margin: 16px auto;
  padding: 32px 0;
  border: 2px solid #555;
  font-size: 64px;
  font-family: Monaco, "DejaVu Sans Mono", "Lucida Console", monospace;
}
```

To display each row horizontally, we set the ul elements to have list-style: none and the li elements to have display: inline. We also set cursor: pointer, so that the mouse cursor reacts to the tiles as if they were links, and provide a hover style, so that the tile being hovered over changes color. Oh, and tiles also change color if they have the selected class.

Other CSS elements that we'll need include a #message to prompt the next move, a .notice div to display the result of each move, and a #scores table:

Download jQuery/5x5/style.css

```
#message {
  position: relative;
  text-align: center;
```

```
    margin: 32px;
    font-size: 24px;
}
```

```
.notice {
  position: relative;
  text-align: center;
  width: 486px;
  margin: 0 auto;
  padding: 16px 0;
  background: #eb4;
  font-size: 18px;
}
```

```
#scores {
  position: relative;
  text-align: center;
  width: 484px;
  margin: 16px auto;
  border: 1px solid #555;
  font-size: 24px;
}
```

jq5x5.coffee

Now for the meat of the project! We'll start by defining some variables that we want to have module-level scope:

```
grid = dictionary = currPlayer = player1 = player2 = selectedCoordinates = null
```

Most of those variables will be given real values in our new newGame function:

```
newGame = ->
  grid = new Grid
  dictionary = new Dictionary(OWL2, grid)
  currPlayer = player1 = new Player('Player 1', dictionary)
  player2 = new Player('Player 2', dictionary)
  drawTiles()

  player1.num = 1
  player2.num = 2
  for player in [player1, player2]
    $("#p#{player.num}name").html player.name
    $("#p#{player.num}score").html 0
  showMessage 'firstTile'
```

Now, because this function refers to HTML on the page, we want to ensure that it isn't called until the document is ready. Hence, it's called from a $(document).ready callback:

```
$(document).ready ->
  newGame()
  $('#grid li').live 'click', tileClick
```

You might be wondering about the drawTiles function in newGame. Here it is:

```
drawTiles = ->
    gridHtml = ''
    for x in [0...grid.tiles.length]
      gridHtml += '<ul>'
      for y in [0...grid.tiles.length]
        gridHtml += "<li id='tile#{x}_#{y}'>#{grid.tiles[x][y]}</li>"
      gridHtml += '</ul>'
    $('#grid').html gridHtml
```

We could've used jQuery to generate and insert each tile individually, but there's nothing wrong with creating a big ol' HTML string when you need to add a whole lot of stuff at once to a document. In fact, it's often the most efficient approach, since it's less expensive to manipulate a string than to manipulate the DOM.

Now, it's not much of a game until we take some input—we need to define tileClick, which we bound to the li elements within #grid using jQuery's live function. Why live rather than bind? Well, live events are triggered no matter what happens to the elements they're bound to—the elements can be burned down to the ground then built back up from nothing, and those trusty live events will still be there. There's also an efficiency advantage—live only creates a single event handler, while bind would create twenty-five of them.

Anyway, here's what happens when you click a tile:

```
tileClick = ->
  $tile = $(this)
  if $tile.hasClass 'selected'
    # undo
    selectedCoordinates = null
    $tile.removeClass 'selected'
    showMessage 'firstTile'
  else
    $tile.addClass 'selected'
    [x, y] = @id.match(/(\d+)_(\d+)/)[1..]
    selectTile x, y
```

The context of a jQuery event callback is the DOM element that triggered the event—in this case, the li that was clicked. We wrap that element to get a jQuery object, which we call $tile. If the tile is already selected, we unselect

it and go back to asking players to select their first tile. If it's a bona fide selection, we proceed to selectTile:

```coffee
selectTile = (x, y) ->
  if selectedCoordinates is null
    selectedCoordinates = {x1: x, y1: y}
    showMessage 'secondTile'
  else
    selectedCoordinates.x2 = x
    selectedCoordinates.y2 = y
    $('#grid li').removeClass 'selected'
    doMove()
```

If selectedCoordinates is null, then this is the player's first tile selection, so we just store the coordinates. Otherwise, the player has chosen both tiles for this turn, so we go on to doMove, which has the Player instance move the tiles around and tabulate score, then displays the results in a notice box:

```coffee
doMove = ->
  {moveScore, newWords} = currPlayer.makeMove selectedCoordinates
  if moveScore is 0
    $notice = $("#{currPlayer.name} formed no words this turn.")
  else
    $notice = $("""
    <p class="notice">
      #{currPlayer} formed the following #{newWords.length} word(s):<br />
      <b>#{newWords.join(', ')}</b><br />
      earning <b>#{moveScore / newWords.length}x#{newWords.length} =
      #{moveScore}</b> points!
    </p>
    """)
  showThenFade $notice
  endTurn()
```

showThenFade is a bit of eye candy that adds a yellow box below the grid, then fades and squashes it before removing it from the DOM entirely:

```coffee
showThenFade = ($elem) ->
  $elem.insertAfter $('#grid')
  animationTarget = opacity: 0, height: 0, padding: 0
  $elem.delay(5000).animate animationTarget, 500, -> $elem.remove()
```

Finally, endTurn updates the tile grid and the score table and then tells the next player that they're up to bat.

To see how the game looks when it all comes together, take a look at .

Player 1, please select your first tile.

```
F D T D I
H X I D O
I O S L Z
D T I B I
R I L E T
```

Player 2 formed the following 3 word(s):
OD, HID, HO
earning **15x3 = 45** points!

Player 1	Player 2
12	45

Figure 5—5x5, powered by jQuery

There are, of course, plenty more features we could add—for instance, a list of words that have already been used, a time limit for each turn, and a "game over" screen after each player has taken a certain number of turns (or after a certain time limit), not to mention fancier animations and interface elements (like drag-and-drop, a feature that's a snap to implement with jQuery UI.)[8] But I'll leave those avenues for you to explore on your own.

5.6 The Future Is jQueryfied

In this chapter, we've touched on all the basics of jQuery, the closest thing JavaScript has to a standard library for working with web pages. You've

---

8.   http://jqueryui.com/

learned to grab elements with selectors, manipulate their CSS styles and HTML attributes, and attach events. The prospect of building an interactive web page from scratch should no longer cause you to break out in a cold sweat.

One major topic we haven't addressed is Ajax. jQuery makes communicating with the server a breeze. You can find the relevant docs at http://api.jquery.com/category/ajax/.

There's something else worth mentioning. Thanks to Node.js, jQuery can now be run easily on the server. How? Through a library called jsdom, which provides a simulated browser environment within Node.[9] One fantastic use case for jsdom and jQuery is templating. Why bother writing server-side templates (such as ERB files in Ruby on Rails) when you can just take raw HTML and manipulate it on the server with jQuery before serving it? That way, you can serve an initial version of the page that makes sense (great for search engines and screen readers) and then reuse the same code on the client side to add dynamic content.

I encourage you to play with Ajax and jsdom on your own. In the next chapter, we'll take a different approach: we'll use WebSocket (through a library called Socket.io) to enable two-way asynchronous communication between the client and the server. And we'll get to know Node.js a little better while we're at it, using it to host the multiplayer version of 5x5.

5.7 Exercises

1. One common (in fact, nearly universal) mistake made by jQuery newbies is that they think of jQuery selectors as "live." For example, they think that they can add new items to the menu (that is, inserting new lis as children of #menu) that will also be hidden by using the following:

   ```
   $('#menu li').hide()
   ```

   False! How would you go about achieving the desired behavior, without calling any methods on the individual li elements? (You're allowed to use a stylesheet.)

2. What does this code do? Is the world safe?

   ```
   $('a').click(destroyWorld).unbind('click')
   ```

3. Three of the following selectors are functionally equivalent. Which would behave differently, and how?

---

9. http://jsdom.org/

```
$('#awayTeam .redShirt').die()
$('#awayTeam').find('.redShirt').die()
$('.redShirt, #awayTeam').die()
$('.redShirt', $('#awayTeam')).die()
```

(By the way, die really is a jQuery method—it's used to unbind events attached with live!)

4. Find, explain, and fix the bug in this code:

```
$('#drJekyll').click ->
  alert 'Now I shall transform!'
  $('#drJekyll').attr 'id', 'mrHyde'
  $('#drJekyll').unbind 'click'
```

Server-Side Apps with Node.js

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

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

6.1 What Is Node.js?

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

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

A number of awesome projects have already been written with Node and CoffeeScript. The following is a small, select sampling. You might want to

---

1.   http://nodeknockout.com/

come back to this list after you've completed the book; reading real-world code is a great way to take your mastery to the next level:

- *Docco* [Ash11]: Uber-computer scientist Donald Knuth advocated "literate programming," in which code and comments are written so that someone encountering the program for the first time can understand it just by reading it once. Docco, written by Jeremy Ashkenas, supports this methodology by generating beautiful web pages in which comments and code are displayed side-by-side.

- *Eco* [Ste11]: Say you're writing a Node-based web application. You have all of these HTML skeletons and a mess of application code, but you're not sure how to combine the two. Eco lets you embed CoffeeScript *within* your markup, turning it into a server-side templating language.

- *Zappa* [NM11]: Creating web applications from scratch has never been simpler. Zappa is a layer on top of Node's popular Express framework that lets you succinctly define how your web server should respond to arbitrary HTTP requests.[2] Works great with Eco, too!

- *Zombie.js* [Ark11]: There's a new kid on the full-stack web app testing block: Zombie.js. Zombie lets you validate your application's behavior with the power of Sizzle, the same selection engine that powers jQuery. Not only is it easy to use, it's also insanely fast.

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

6.2 Modularizing Code with 'exports' and 'require'

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

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

Download Nodejs/app.coffee
```coffee
util = require './util'

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

---

2. http://expressjs.com

Download Nodejs/util.coffee

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

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

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

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

Download Nodejs/app.js

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

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

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

6.3 Thinking Asynchronously

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

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

(Of course, without threads there's no way to take advantage of multiple processors. The good news is that there are already projects out there, such

as multi-node and cluster, that effectively bind multiple instances of your app to the same server port, giving you the performance advantages of parallel processing without the headaches of sharing data across threads.[3])

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

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

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

Here's what happens:

1. We ask Node.js to read the current directory with fs.readdir, passing a callback.

2. Node.js passes the request along to the operating system, then immediately returns.

3. We print 'This will happen first.' to the console.

4. Once our code has run, Node.js checks to see if the operating system has answered our request yet. It has, so it runs our callback, and a list of files in the current directory is printed to the console.

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

All of that is as true in the browser as it is in Node, but it's doubly important to understand in Node because your application logic *will* take the form of

---

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

tangled chains of callbacks. Have no doubt about it. The challenge is to manage them in a way that humans can understand.

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

1. We get the user's information from the database to check that they have permission to make the request.

2. If so, we update the database accordingly.

3. We read a template from the file system, customize it appropriately, and send it to the user.

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

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

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

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

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

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

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

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

Scope in Loops

Remember what we learned in Section 2.2, *Scope: Where You See 'Em, on page 18*: *only functions create scope.* Expecting loops to create scope leads otherwise mild-mannered programmers to summon forth horrific bugs when dealing with asynchronous callbacks. In fact, this is probably the most common source of confusion in asynchronous code.

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

A first attempt might look like this:

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

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

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

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

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

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

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

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

6.4   Project: Multiplayer 5x5

We're going to build a web server that can let folks take each other on at 5x5. On the client side, we'll be using essentially the same HTML and CSS as in the last chapter; our server will serve all of those static files (and some CoffeeScript, of course) and handle the game state.

There are a number of ways that we could handle the problem of coordinating the clients with the server. How much logic do we want to put in the client and how much on the server? Many frameworks, such as Backbone.js, exist to make this problem conceptually simpler.[4] Often it's desirable to have some logic on the client side (for performance) and some logic on the server side (for security), and the two may overlap. But for our purposes, we're going to lean toward a "dumb client" approach, putting all of the logic on the server side. That means that when player A makes a move, the following happens:

1. Player A's client sends the move (that is, the coordinates of the swapped tiles) to the server.

2. If the move is valid, the server sends both clients the results of the move.

3. The two clients display the results.

Simple, right? This approach makes the client library very light, since the Dictionary, Grid, and Player classes only need to exist on the server side (not to mention the 2.1 MB list of valid words!). The downside is that there's going to be a bit of lag before players see the results of their actions. In a real application, we'd want to put more thought into optimizing responsiveness. (For instance, Google Docs syncs every word you type to the server, but imagine how frustrating it would be to not see those letters on your screen until the server acknowledged them!)

---

4.   http://documentcloud.github.com/backbone/

There are two files we'll be working with: 5x5client.coffee and 5x5server.coffee. Let's start with the server.

5x5server.coffee

We're going to build our server on the popular Connect framework.[5] Connect extends Node's own net module and is in turn extended by more robust frameworks such as Express and Zappa. For more sophisticated apps, you should definitely take a look at those other frameworks, which add niceties such as DSLs for defining URL routes. However, for our single-page app, Connect will do just fine. Let's install it:

```
$ npm install connect
```

(Be sure to either run that command from the project directory or do a global install by adding the -g flag.)

Now let's start our server code by creating a Connect server instance:

Download Nodejs/5x5/5x5server.coffee
```coffee
connect = require 'connect'

app = connect.createServer(
  connect.compiler(src: __dirname + '/client', enable: ['coffeescript']),
  connect.static(__dirname + '/client'),
  connect.errorHandler dumpExceptions: true, showStack: true
)

port = 3000
app.listen port
console.log "Browse to http://localhost:#{port} to play"

io = require 'socket.io'
socket = io.listen app
socket.on 'connection', (client) ->
  if assignToGame client
    client.on 'message', (message) -> handleMessage client, message
    client.on 'disconnect', -> removeFromGame client
  else
    client.send 'full'

assignToGame = (client) ->
  idClientMap[client.sessionId] = client
  return false if game.isFull()
  game.addPlayer client.sessionId
  if game.isFull() then welcomePlayers()
  true
```

---

5. http://senchalabs.github.com/connect/

```coffee
removeFromGame = (client) ->
  delete idClientMap[client.sessionId]
  game.removePlayer client.sessionId

welcomePlayers = ->
  players = [game.player1, game.player2]
  info = {players, tiles: game.grid.tiles, currPlayerNum: game.currPlayer.num}
  for player in players
    playerInfo = extend {}, info, {yourNum: player.num}
    idClientMap[player.id].send "welcome:#{JSON.stringify playerInfo}"

handleMessage = (client, message) ->
  {type, content} = typeAndContent message
  if type is 'move'
    return unless client.sessionId is game.currPlayer.id # no cheating!
    swapCoordinates = JSON.parse content
    {moveScore, newWords} = game.currPlayer.makeMove swapCoordinates
    result = {swapCoordinates, moveScore, newWords, player: game.currPlayer}
    socket.broadcast "moveResult:#{JSON.stringify result}"
    game.endTurn()

typeAndContent = (message) ->
  [ignore, type, content] = message.match /(.*?):(.*)/
  {type, content}

extend = (a, others...) ->
  for o in others
    a[key] = val for key, val of o
  a
```

While we could use Connect in conjunction with Apache or nginx, it's per-
fectly capable of serving our static files (in the client subdirectory) on its own.
We just have to ask it to do so in its configure block. We'll also throw in an
error handler while we're at it; otherwise, exceptions would just be silently
swallowed:

```coffee
app = connect.createServer(
  connect.compiler(src: __dirname + '/client', enable: ['coffeescript']),
  connect.static(__dirname + '/client'),
  connect.errorHandler dumpExceptions: true, showStack: true
)
```

We only need to do one more thing in order to start our server: tell it which
port to run on, via the listen function. The choice of port is largely arbitrary;
in production, we'd most likely want to use port 80 (the standard for HTTP),
but to avoid potential conflicts, let's use 3000:

```coffee
port = 3000
app.listen port
console.log "Browse to http://localhost:#{port} to play"
```

If we just run what we've got and head to http://localhost:3000/ in our browser, we'll be automatically served index.html. Now we just need to add the means to talk to our client code. But how? When one player performs an action, both players need to be informed of the results. Ajax is ill-suited to that sort of thing; what we need is a way to broadcast data from the server to several clients at once, at any time.

Fortunately, a new technology called WebSocket provides exactly that. A Node library called Socket.IO makes this a breeze; plus, it automatically falls back on other protocols in browsers that don't yet have WebSocket support.

We'll install Socket.io in trusty npm fashion:

```
$ npm install socket.io
```

And now we'll use it on the server:

```coffee
io = require 'socket.io'
socket = io.listen app
socket.on 'connection', (client) ->
  if assignToGame client
    client.on 'message', (message) -> handleMessage client, message
    client.on 'disconnect', -> removeFromGame client
  else
    client.send 'full'
```

Now each time a browser connects to the server, we get a new Socket.io client instance. We implement two callbacks: one for when the client sends a message (that is, a move) and another for when they disconnect. We also assign the client to a game immediately.

For simplicity, this server implementation only hosts one game at a time, but the game state has been encapsulated by a Game class, so it should be simple to extend the project to a multigame server. That gives us just two variables with module scope (other than the functions, of course): the game itself and a map from IDs to clients.

```coffee
game = new Game
idClientMap = {}
```

Each `Player` instance now has an `id` attribute, so when we need to send a message to a particular player, we use `idClientMap` to get their `client` object. Here's how we add a new client to the game:

```coffee
assignToGame = (client) ->
  idClientMap[client.sessionId] = client
  return false if game.isFull()
  game.addPlayer client.sessionId
  if game.isFull() then welcomePlayers()
  true
```

Once the game has two players, we send each of them a welcome message with the list of tiles and, in case someone arrives midgame, the scores:

```coffee
welcomePlayers = ->
  players = [game.player1, game.player2]
  info = {players, tiles: game.grid.tiles, currPlayerNum: game.currPlayer.num}
  for player in players
    playerInfo = extend {}, info, {yourNum: player.num}
    idClientMap[player.id].send "welcome:#{JSON.stringify playerInfo}"
```

Notice that `extend` in there? That's a small utility for adding the properties of one object to another. It's equivalent to _.extend from Underscore.js:

```coffee
extend = (a, others...) ->
  for o in others
    a[key] = val for key, val of o
  a
```

The only thing that's left is to handle player moves:

```coffee
handleMessage = (client, message) ->
  {type, content} = typeAndContent message
  if type is 'move'
    return unless client.sessionId is game.currPlayer.id # no cheating!
    swapCoordinates = JSON.parse content
    {moveScore, newWords} = game.currPlayer.makeMove swapCoordinates
    result = {swapCoordinates, moveScore, newWords, player: game.currPlayer}
    socket.broadcast "moveResult:#{JSON.stringify result}"
    game.endTurn()
```

Notice that we use the client's unique session ID (provided by Socket.io) as a simple security check to prevent one player from making the other player's move. After we've calculated the results of the move, we use `socket.broadcast`, a handy shorthand when we want to send the same message to every connected client.

5x5client.coffee

So how do we interact with Socket.io from the client? Well, handily, Socket.io provides us with a client library. We just have to include it:

```
Download Nodejs/5x5/client/index.html
<script type="text/javascript" src="/socket.io/socket.io.js"></script>
```

If you're exceptionally nosy, you might have noticed that there is no file called socket.io.js in our static directory. And yet, if you request it from the server, it's there!

It turns out that the server-side Socket.io library *automagically* serves its own client library. Of course, in production you'd want to minify and concatenate that script along with all your other scripts, but for development, the feature is very handy. If you update to a new version of Socket.io on the server side, then you won't have to worry about serving the new client library—it's taken care of for you.

So, how do we actually use it? Well, it's actually quite similar to what we wrote for the server:

```
Download Nodejs/5x5/client/5x5client.coffee
$(document).ready ->
  $('#grid li').live 'click', tileClick
  socket = new io.Socket()
  socket.connect()
  socket.on 'connect', -> showMessage 'waitForConnection'
  socket.on 'message', handleMessage
```

We need to handle two types of messages: the initial welcome and the result of a move:

```
Download Nodejs/5x5/client/5x5client.coffee
handleMessage = (message) ->
  {type, content} = typeAndContent message
  switch type
    when 'welcome'
      {players, currPlayerNum, tiles, yourNum: myNum} = JSON.parse content
      startGame players, currPlayerNum
    when 'moveResult'
      {player, swapCoordinates, moveScore, newWords} = JSON.parse content
      showMoveResult player, swapCoordinates, moveScore, newWords

startGame = (players, currPlayerNum) ->
  for player in players
    $("#p#{player.num}name").html player.name
    $("#p#{player.num}score").html player.score
  drawTiles()
  if myNum is currPlayerNum
```

```
    startTurn()
  else
    endTurn()

showMoveResult = (player, swapCoordinates, moveScore, newWords) ->
  $("#p#{player.num}score").html player.score
  $notice = $('<p class="notice"></p>')
  if moveScore is 0
    $notice.html "#{player.name} formed no words this turn."
  else
    $notice.html """
      #{player.name} formed the following #{newWords.length} word(s):<br />
      <b>#{newWords.join(', ')}</b><br />
      earning <b>#{moveScore / newWords.length}x#{newWords.length}
      = #{moveScore}</b> points!
    """
  showThenFade $notice
  swapTiles swapCoordinates
  if player.num isnt myNum then startTurn()
```

And that's it! The rest of the code is pretty similar to the version from the last chapter—simpler, in fact, since the game logic is all on the server.

Running `coffee 5x5server.coffee` gives you this invitation:

```
Browse to http://localhost:3000 to play
```

Open up two browsers, point them both to that address, and you'll get something like Figure 6, .

Is That All?

Gee whiz—you've built a fully buzzword-compliant multiplayer gaming app with Node.js and WebSocket! It may not be the next viral sensation on YouFace, but it demonstrates that using cutting-edge web technologies can actually be pretty easy.

Of course, for larger-scale apps, you'd want to use a more robust web framework (like brunch or the aforementioned Zappa);[6] hook in a database (Node.js already has first-rate bindings for MySQL, MongoDB, and Redis); and add in logging, tracking, and performance-monitoring (perhaps with the sleek, Node-powered Hummingbird).[7] It's just amazing how vibrant the Node ecosystem is—and it hasn't even hit 1.0 yet.

---

6. http://brunchwithcoffee.com/
7. http://projects.nuttnet.net/hummingbird/
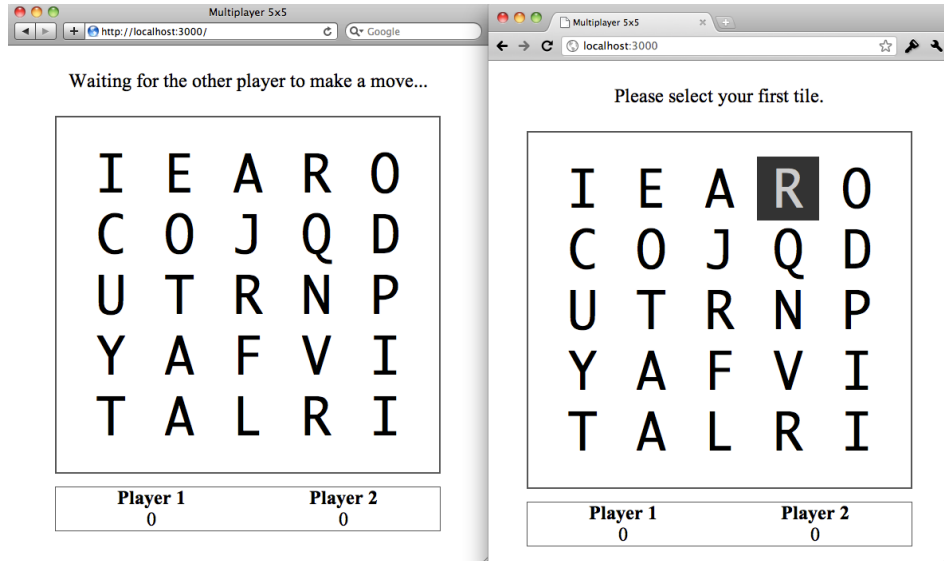
Figure 6—Playing multiplayer 5x5

6.5 Client, Server—What's the Difference?

We've only scratched the surface of Node.js in this chapter. Despite its youth, Node is already a powerful framework with a thriving ecosystem, making CoffeeScript a viable language well beyond the browser.

Node has attracted a lot of hype in part because of its evented architecture, built around nonblocking IO, which makes it an efficient alternative to web platforms that rely on multithreading. But perhaps more exciting for the code-monkey masses is that we can now write for our client and server in the same language and even migrate code from one to the other. The techniques this allows are only beginning to be explored, ranging from testing to templating to validation. Who knows what's next?

That, alas, is beyond the realm of this humble book, which I hereby congratulate you on completing! You've now learned to apply CoffeeScript to both front- and back-end environments, using its power to provide shorter, clearer code than even the greatest JavaScripter could produce (nothing personal, Resig).

Consider yourself a graduate of CoffeeScript University, and remember: *virtus brevitas!*

6.6 Exercises

1. What will this code do?

```
countdown = 10
h = setInterval (-> countdown--), 100
do (->) until countdown is 0
clearInterval h
console.log 'Surprise!'
```

(Bonus exercise: Rewrite this to do what the coder likely intended.)

Answers to Exercises

A1.1 Functions, Scope, and Context

Answers to :

1. Functions in CoffeeScript return the value of their last expression—in this case, the return value of the splice method. To change the return value, just add a new line, like so:

    ```
    clearArray = (arr) ->
      arr.splice 0, arr.length
      arr
    ```

    This returns the now-empty arr. To have the function return nothing, use this:

    ```
    clearArray = (arr) ->
      arr.splice 0, arr.length
      return
    ```

2. You can write this one of two ways, either of which will do just fine. Here's the first:

    ```
    run = (func, args...) -> func.apply this, args
    ```

    Here's the second:

    ```
    run = (func, args...) -> func.call this, args...
    ```

    Note that the use of this allows a context to be passed in by calling run in the desired context.

3. The postfix operators (if/unless and for/while/until) are the only major exceptions to the rule that implicit parentheses go to the end of the line. For example, all of the following lines are equivalent:

```
return abortMission warning if warning?
return abortMission(warning) if warning?
if warning? then return abortMission warning
if warning? then return abortMission(warning)
```

Adding explicit parentheses that go to the end of the line would change the meaning considerably:

```
return abortMission(warning if warning?)
```

Now we return and call abortMission no matter what! The expression warning if warning? is evaluated, but to little effect (it converts null to undefined but leaves all other values of warning untouched).

4. CoffeeScript doesn't allow space between a function and its explicit parentheses because this would allow parentheses around an expression to radically change its meaning. Here are some examples:

```
f g h
```

This expression really means the following:

```
f(g(h))
```

Compare that meaning with this expression:

```
f (g) h
```

Here the parentheses really mean this:

```
f(g)(h)
```

CoffeeScript's rule is that if there's whitespace after any identifier (and something other than a postfix operator after the whitespace), then that identifier is a function with implicit parentheses.

5. foo.bar.baz() runs in the foo.bar context. @hoo runs in this (also known as @). @hoo.rah() runs in @hoo.

6. what.x and @x are, of course, equivalent if and only if what is this. Again, it's perfectly possible for what.x and @x to refer to the same object, but what.x = y will not overwrite @x unless what is this.

   The code example can be solved by writing xInContext.call what.

7. The code fails because of the x = x, which is a no-op. Here's the problem part again:

```
x = true
showAnswer = (x = x) ->
  console.log if x then 'It works!' else 'Nope.'
showAnswer()
```

Recall that the default argument syntax a = b is equivalent to placing a ?= b at the top of the function body; there's no way to bring the x from the outer scope into the function. The solution is to either use showAnswer x or to ditch the shadowing.

A1.2 Collections and Iteration

Answers to :

1. When you use slice, the result is a new array containing some or all of the items from the original array; adding, removing, or replacing items in the new array will not affect the original. That's why you'll see arr.slice[0..] in a lot in functions—when someone passes you an array and you want to modify it for your own purposes, working with a copy is just common courtesy.

2. In the following code it's important to realize that once is only called, well, once():

```coffeescript
once = ->
  if once.hasRun
    null
  else
    once.hasRun = true
    [1, 2, 3]
console.log x for x in once()
```

That last line is equivalent to this:

```coffeescript
onceResult = once()
console.log x for x in onceResult
```

In short, CoffeeScript takes care of caching the function result automatically in a for loop. If you want to call a function on each loop iteration, you should use while or until.

3. Look at this section:

```coffeescript
for x in [1, 2]
  setTimeout (-> console.log x), 50
```

This gives the following output:

```
2
2
```

What's going on? The key here is that there's only one x variable. The timeout is invoked after the loop has finished and x has been set to 2; it doesn't matter what the value of x was when the function was declared.

Changing the timeout to 0 has no effect because setTimeout always adds its target to the "event queue," which isn't invoked until after all other code has run.

The easiest solution is to use do to capture the value of x in each loop iteration:

```
for x in [1, 2]
  do (x) ->
    setTimeout (-> console.log x), 50
```

4. Here's a function that checks whether a particular value is attached to the given object:

```
objContains = (obj, match) ->
  for k, v of obj
    if v is match
      return true
  false
```

Note that k is unused but necessary; the of syntax always goes in the order key, value, and we want that value.

In practice, you should be writing your code so that this sort of loop is unnecessary. The whole point of the hash structure is that fetching values is fast when you know the corresponding key. If you're frequently checking whether a value is in a hash or not, you should be using a different data structure.

5. To run a function once and then repeat it until a condition is called, we can write this:

```
doAndRepeatUntil = (func, condition) ->
  func.call this
  func.call this until condition()
```

6. To get the length of the shortest string in our wordList array, we can write the following:

```
Math.min.apply Math, (w.length for w in wordList)
```

The comprehension (w.length for w in wordList) generates a list of the length of each word in wordList. Using apply passes it to Math.min as an enormous list of arguments. (The first argument to apply ensures that Math.min runs in the Math context, just as it would if we called Math.min directly.) This isn't the most efficient approach, but it's very succinct.

A1.3 Modules and Classes

Answers to Section 4.7, *Exercises*, on page 72:

1. Why doesn't this give us the same old aphorism twice?

```
root = global ? window

root.aphorism = 'Fool me 8 or more times, shame on me'

do restoreOldAphorism = ->
  aphorism = 'Fool me once, shame on you'
  console.log aphorism

console.log aphorism
```

The issue here is that restoreOldAphorism declares a variable called aphorism in its own scope. The compiler doesn't realize that setting root.aphorism has created a variable with the same name in the global scope; its scoping rules only apply to simple assignments of the form aphorism = ….

It's fine to read root.aphorism as just aphorism, but assignments have to be made to root.aphorism.

2. The following code is confusing because it uses @wishesLeft to refer to both a property of the prototype and a property of the instance:

```
Genie = ->
Genie::wishesLeft = 3

Genie::grantWish = ->
  if @wishesLeft > 0
    console.log 'Your wish is granted!'
    @wishesLeft--
```

The result is that *each* genie will grant three wishes, rather than enforcing a total limit of three wishes.

To understand why, consider this:

```
@wishesLeft--
```

This is equivalent to @wishesLeft = @wishesLeft - 1. The first time this line is run on a Genie instance (let's call it genie1), it reads Genie::wishesLeft, subtracts 1 from it, and then assigns that value to a new instance property, genie1.wishesLeft!

You can read prototype properties from an object as if the property were attached to the object, but when you write obj.x = y, you're always setting the value of a property on the object itself (potentially shadowing a prototype property).

As a rule, you should never use the same name for a prototype property and an instance property. The best solution in this case is to replace both Genie::wishesLeft and @wishesLeft with Genie.wishesLeft.

3. The answer is based on the definitions given:

```
class Season
class Spring extends Season
```

Both (new Season).__proto__.__proto__ and (new Spring).__proto__.__proto__.__proto__ are equal to {}.__proto__, the default object prototype.

4. Bound functions on classes work as you were probably hoping. When you call foo = new Foo(), instance methods defined on Foo with => are automatically bound to the instance context.

This is the behavior you want 95 percent of the time. However, there is a overhead from bound functions (in both code size and instantiation time), so they may not be suitable for performance-critical code.

A1.4 Web Interactivity with jQuery

Answers to Section 5.7, *Exercises*, on page 88:

1. This line of code will hide all current list items contained by #menu, but it will not affect those created in the future:

```
$('#menu li').hide()
```

To hide all current and future list items in #menu, there are two things we need to do. First, we modify our stylesheet to provide a special class for #menu, hideItems:

```
#menu.hideItems li {
  visibility: hidden;
}
```

With the aid of that bit of CSS, hiding all menu items is as easy as using jQuery's addClass method:

```
$('#menu').addClass 'hideItems'
```

Now all current and future menu items shall be unseen until we call removeClass.

2. There are two things worth noticing about this code:

```
$('a').click(destroyWorld).unbind('click')
```

First, destroyWorld will never get called, because the click event handler is unbound immediately after it's bound. Second, the unbind here will remove all click event handlers from all <a> elements.

3. Of the four selectors, the odd one out is #3, $('.redShirt, #awayTeam').die():

```
$('#awayTeam .redShirt').die()
$('#awayTeam').find('.redShirt').die()
$('.redShirt, #awayTeam').die()
$('.redShirt', $('#awayTeam')).die()
```

This would kill all live events on all elements with the redShirt class and on the awayTeam element as well, rather than just on the redShirt-class members of the awayTeam.

4. The following code will, in fact, change drJekyll's ID to mrHyde.

```
$('#drJekyll').click ->
  alert 'Now I shall transform!'
  $('#drJekyll').attr 'id', 'mrHyde'
  $('#drJekyll').unbind 'click'
```

But annoyingly, it'll shout "Now I shall transform!" every time it's clicked. The problem is that the '#drJekyll' selector doesn't match anything after the ID change. Moving the unbind to the top of the click function would be one solution. Using $(this) instead of $('#drJekyll') would be even better.

A1.5 Server-Side Apps with Node.js

Answers to Section 6.6, *Exercises*, on page 105:

1. The code given is, alas, an infinite loop:

```
countdown = 10
h = setInterval (-> countdown--), 100
do (->) until countdown is 0
clearInterval h
console.log 'Surprise!'
```

No matter how many hundreds of milliseconds pass, countdown will never decrease, because the loop code never finishes running. Here's a working version:

```
countdown = 10
decreaseCountdown = ->
  countdown--
  if countdown is 0
    clearInterval h
    console.log 'Surprise!'
h = setInterval decreaseCountdown, 100
```

Note that it's not a problem that `h` doesn't exist yet when we write the line `clearInterval h`. When the line is run, the program will see that there's no `h` in the current scope, check the outer scope, and thus find the handle returned by `setInterval`. Such nonlinear thinking is, alas, the price we pay to be rid of the scourge of threads.

Ways of Running CoffeeScript

Although CoffeeScript's ecosystem is young, there are a vast number of tools out there for compiling and running CoffeeScript code. We already covered the official command-line compiler in Section 1.3, *Meet 'coffee'*, on page 6; this appendix covers a few of the other options I recommend. For a more comprehensive list, visit http://github.com/jashkenas/coffee-script/wiki.

A2.1 Web Consoles

If you head over to http://coffeescript.org, you'll find not only a ton of examples but also a button labeled "Try CoffeeScript." Click it, and a live console pops out. You'll want to use a browser with a developer console so that you don't have to put up with alert for output.[1] Try typing this:

```
console.log ['a', 'b', 'c'][0...-1]
```

You'll instantly see the compiled JavaScript appear on the right:

```
console.log(['a', 'b', 'c'].slice(0, -1));
```

Click Run and you'll see the result ["a", "b"] in your browser's console.

"Try CoffeeScript" is great when you want to see CoffeeScript and compiled JavaScript side-by-side, but what if you want more of a REPL experience, maybe one that lets you try CoffeeScript along with libraries like jQuery and Underscore.js? Then check out JS Console.[2] Although limited (there's currently no way to write multiline expressions), it's very slick, and there's even an iPhone version!

You might be surprised at how fast these consoles are. With a web-based console for Ruby or Python, say, all the commands have to run remotely on

---

1.   http://getfirebug.com/firebuglite
2.   http://jsconsole.com/

the server. But these sites run the CoffeeScript compiler *in your browser*. And you can do this, too—see the next section.

A2.2 Running CoffeeScript in Your Web App

Wouldn't it be nice if you could include CoffeeScript directly in your HTML without having to compile it to JavaScript first? Ah, but you can!

```
<script type="text/coffeescript">
  alert 'Wow, this CoffeeScript is right in your HTML!'
</script>
```

The only catch is that you have to include a special version of the Coffee-Script compiler, which weighs in at a whopping 170 KB.[3]

Since it simplifies development, we used the browser-based compiler for our example project in Chapter 5, *Web Interactivity with jQuery*, on page 75. But unless you're building your own CoffeeScript console, this approach isn't recommended for production use. In addition to the sheer size of the compiler, all CoffeeScript files are loaded via Ajax after the compiler has been loaded.

The next few tools we'll be looking at are aimed at bridging that gap between easy web development and efficient deployment.

A2.3 CoffeeScript on Rails

CoffeeScript owes a lot to Ruby. Its first compiler was written in Ruby; many of its earliest users were also Rubyists; and now the language enjoys the support of the preeminent Rubyists at 37signals. In April 2011, it was announced that CoffeeScript support will be included in Rails 3.1 via Sprockets 2. As of this writing, neither has been released, but you can check out the latest Sprockets at https://github.com/sstephenson/sprockets.

Earlier versions of Rails can still enjoy first-class CoffeeScript integration, thanks to a plugin called Barista:[4] you put your .coffee files in one directory, and the corresponding .js files are automatically generated as needed on each page request. You can also embed CoffeeScript in ERB and Haml templates.

Barista goes even further by allowing you to package and pull in CoffeeScript from gems. This is a terrific way to split large projects into reusable, cleanly versioned components. Plus, it's 100 percent Heroku-compatible; just plug

---

3. http://jashkenas.github.com/coffee-script/extras/coffee-script.js
4. http://github.com/Sutto/barista

in therubyracer-heroku, a JavaScript interpreter designed to run in any Ruby environment.[5] (The same applies to Rails 3.1 apps; both Barista and Rails 3.1 employ the same gem to wrap around the CoffeeScript compiler.)[6]

One bummer with Barista (and most other web framework integrations for CoffeeScript) is that if there's a syntax error in your code, you won't find out about it until you refresh the page and get broken JavaScript. To help with this, I wrote a plugin that extends Barista to provide a Growl notification whenever your CoffeeScript fails to compile.[7]

A2.4 CoffeeScript via Middleware

Wouldn't it be nice if your app could think it's using plain old JavaScript, while the server handles the CoffeeScript compilation transparently? Well, with middleware—software that fits snugly between your web framework and the server—all these dreams, and more, can come true!

In the Ruby world, the middleware of choice is Rack. The most mature Rack integration is the aptly named rack-coffee.[8] It's compatible with Rails, Sinatra, and all other major Ruby web frameworks (though Sinatra has actually had built-in CoffeeScript support since October 2010). More recently, the aforementioned Barista has been modified to run in any Rack-based framework, not just Rails.

Meanwhile, over in the land of Python, CoffeeCup offers some first-class CoffeeScript support for Django, Pylons, CherryPy, and countless other WSGI-based frameworks.[9] If having significant whitespace in both your front and back end is important, that's a great option. Of course, another option would be to write your front and back end in the same language entirely.

A2.5 CoffeeScript on Node.js

Recall that .coffee files can be run directly on Node.js using the coffee command. So on the back end, no compilation is needed. The real trick is serving compiled JavaScript for the front end.

Fortunately, Connect (the standard core for Node.js web frameworks) comes with middleware that does this automatically. It just takes a little bit of configuration. You need to hook in Connect's compiler and static middlewares,

---

5. https://github.com/aler/therubyracer-heroku
6. https://github.com/josh/ruby-coffee-script
7. http://github.com/TrevorBurnham/barista_growl
8. http://github.com/mattly/rack-coffee
9. http://github.com/dsc/coffeecup

in that order, like so. (We used this technique in Chapter 6, *Server-Side Apps with Node.js*, on page 91.)

```
compiler = connect.compiler src: coffeeDir, enable: ['coffeescript']
static = connect.static coffeeDir
connect.createServer compiler, static
```

In this example, .coffee files in coffeeDir will automatically be compiled and served as .js files.

Some Node frameworks include preconfigured CoffeeScript-serving power. See, for instance, brunch and Zappa.

While it seems like everyone and their dog is developing web apps with Rails or Django these days, there are still plenty of sites that just don't need a back end at all. Wouldn't it be nice to be able to leverage CoffeeScript—along with Haml and Sass, perhaps—to develop ordinary websites, all while generating standards-compliant HTML/CSS and minified JavaScript for deployment?

A2.6 Rapid Websites with Middleman

Luckily, there's Thomas Reynolds's Middleman.[10] If you've already got Ruby and RubyGems installed, then getting up and running with Middleman is as easy as doing this:

```
$ gem install middleman
$ mm-init myProject
$ cd myProject
$ mm-server
```

Now head to http://localhost:4567, where your site is up and running. Then modify the template files in the view directory and refresh your browser; repeat as needed until you're ready to deploy.

Middleman expects CoffeeScript files to be saved anywhere in the view folder with the extension .js.coffee. For instance, if you put awesome.js.coffee in view/scripts, then the corresponding <script> tag would refer to scripts/awesome.js. No such file exists, mind you—it's generated automatically by the server on every request. This means that when you change your CoffeeScript code and refresh the browser, the page will always be served with the latest and greatest JavaScript.

When you're ready to deploy, just run this:

```
$ mm-build
```

---

10. http://middlemanapp.com/

All the HTML/CSS/JavaScript you need will show up in the build folder. Upload that folder to a server, and your site is live!

A2.7 CoffeeScript for System Scripts

Before we finish this appendix, did you know that you can use CoffeeScript for the kinds of system tasks that you'd normally associate with Perl or Python? Just start your file with a "shebang" line that gives the path to coffee (sorry, Windows folks, this only works on Unix-y systems):

```
#!/usr/bin/env coffee
console.log 'Hello, world!'
```

Ensure that the script is executable (chmod +x helloscript.coffee); then you can run it with either sh helloscript.coffee or ./helloscript.coffee. (The familiar coffee hello-script.coffee will also still work.)

Remember, this script will only run on systems where the CoffeeScript compiler is installed and on the PATH.

That concludes our tour of the CoffeeScript build tool ecosystem, but no doubt many other awesome projects have sprung up since the time of this writing! Be sure to check out http://github.com/jashkenas/coffee-script/wiki for an up-to-date list, and follow @CoffeeScript on Twitter to keep tabs on new developments.

Cheat Sheet for JavaScripters

This cheat sheet summarizes most of the keywords and operators that CoffeeScript provides in terms of their JavaScript equivalents. In the tables below, *Symbol* refers to the preferred style in CoffeeScript, while *Alias* refers to a less-preferred alternative. Often, the alias is the same in JavaScript. For instance, or is recommended over || in CoffeeScript, but both are allowed.

Occasionally, the JavaScript code given is slightly simplified and differs from the CoffeeScript in some edge cases; these are clarified in notes at the foot of each table.

Symbols that are unchanged from JavaScript, such as the math and bitwise operators, are not mentioned here.

A3.1 Boolean Operators

Symbol	Alias	JavaScript
not x	!x	!x
x or y	x || y	x || y
x or= y	x ||= y	x = x || y
x and y	x && y	x && y
x and= y	x &&= y	x = x && y

A3.2 The Existential Operator

Symbol	Alias	JavaScript
x?		typeof x != 'undefined' && x !== null
func?()		if (typeof func === 'function') {func()}
x?.y	x.y if x?	typeof x !== 'undefined' && x !== null ? x.y : undefined
x ?= y	x = y unless x?	if (typeof x === 'undefined' || x === null) {x=y}

A3.3 Context and Prototype Accessors

Symbol	Alias	JavaScript
@	this	this
@x	this.x	this.x
@['x']	this['x']	this['x']
obj::y	obj.prototype.y	obj.prototype.y
obj::['y']	obj.prototype['y']	obj.prototype['y']

A3.4 Function Definitions

Symbol	JavaScript
func = (a) -> ...	func = function(a) {...}
func = (a) => ...	func = bind(function(a) {...}, this)*

*Where bind returns a wrapper that runs the function in the given context. See *How Does => Work?*, on page 25.

A3.5 Conditionals

Symbol	Alias	JavaScript
y if x	if x then y*	if (x) {y}
y unless x	y if not x	if (!x) {y}
a = if x then y	if x then a = y else a = un-defined	a = x ? y : undefined
a = if x then y else z	if x then a = y else a = z	a = x ? y : z
switch x		See *Polymorphism and Switching*, on page 66.

*Indentation may be used instead of then.

A3.6 Property Existence

Symbol	JavaScript
x of obj	x in obj
x in arr	y.indexOf(x) >= 0*

*CoffeeScript actually uses the indexOf method directly from the Array prototype; and if Array.prototype.indexOf is undefined (as in Internet Explorer 8 and below), an equivalent function is used instead.

A3.7 Iteration

Symbol	Alias	JavaScript
f() for x of obj	for x of obj then f()	for (x in obj) {f()}
f() for x in arr	for x in arr then f()	for (var i = 0; i < arr.length; i++) {f()}*
f() for x in [a..b]	for x in [a..b] then f()	for (var i = a; i <= b; i++) {f()}**
f() for x in [a...b]	for x in [a...b] then f()	for (var i = a; i < b; i++) {f()}**

*arr.length is cached in the CoffeeScript version, so that changes made to the array during the loop will not affect the range of iteration.

**Assuming a < b. If a > b, then the loop counts downward instead of upward.

APPENDIX 4

Bibliography

[Ark11]   Assaf Arkin. *Zombie.js*. Labnotes.org, http://zombie.labnotes.org/, 2011.

[Ash11]   Jeremy Ashkenas. *Docco*. Github, https://github.com/, 2011.

[Hav11]   Marijn Haverbeke. *Eloquent JavaScript: A Modern Introduction to Programmings*. No Starch Press, San Francisco, CA, 2011.

[NM11]   Nishiyama Nishiyama and Maurice Machado. *Zappa*. Github, https://github.com/, 2011.

[Sei09]   Peter Seibel. *Coders at Work: Reflections on the Craft of Programming*. Apress, New York City, NY, 2009.

[Ste11]   Sam Stephenson. *Eco*. Github, https://github.com/, 2011.

Index

Welcome to the New Web

The world isn't quite ready for the new web standards, but you can be. Get started with HTML5, CSS3, and brush up on your JavaScript today.

HTML5 and CSS3 are the future of web development, but you don't have to wait to start using them. Even though the specification is still in development, many modern browsers and mobile devices already support HTML5 and CSS3. This book gets you up to speed on the new HTML5 elements and CSS3 features you can use right now, and backwards compatible solutions ensure that you don't leave users of older browsers behind.

Brian P. Hogan
(280 pages) ISBN: 9781934356685. $33
http://pragmaticprogrammer.com/titles/bhh5
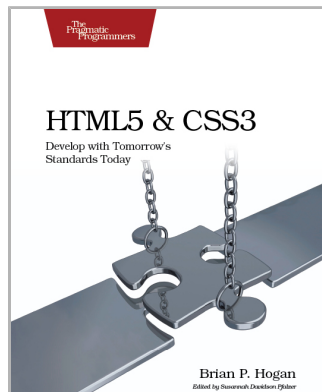
JavaScript is everywhere. It's a key component of to-day's Web—a powerful, dynamic language with a rich ecosystem of professional-grade development tools, infrastructures, frameworks, and toolkits. This book will get you up to speed quickly and painlessly with the 35 key JavaScript tasks you need to know.

NEW: Part of the new *Pragmatic Guide* series

Christophe Porteneuve
(150 pages) ISBN: 9781934356678. $25
*http://pragmaticprogrammer.com/titles/pg_js*
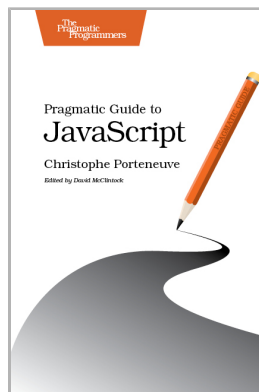
Learn a New Language This Year

Want to be a better programmer? Each new programming language you learn teaches you something new about computing. Come see what you're missing.

You should learn a programming language every year, as recommended by *The Pragmatic Programmer.* But if one per year is good, how about *Seven Languages in Seven Weeks*? In this book you'll get a hands-on tour of Clojure, Haskell, Io, Prolog, Scala, Erlang, and Ruby. Whether or not your favorite language is on that list, you'll broaden your perspective of programming by examining these languages side-by-side. You'll learn something new from each, and best of all, you'll learn how to learn a language quickly.

Bruce A. Tate
(300 pages) ISBN: 9781934356593. $34.95
http://pragmaticprogrammer.com/titles/btlang
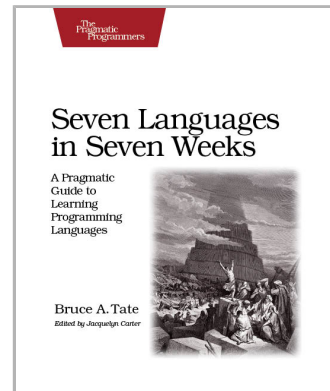
Bill Karwin has helped thousands of people write better SQL and build stronger relational databases. Now he's sharing his collection of antipatterns—the most common errors he's identified out of those thousands of requests for help.

Most developers aren't SQL experts, and most of the SQL that gets used is inefficient, hard to maintain, and sometimes just plain wrong. This book shows you all the common mistakes, and then leads you through the best fixes. What's more, it shows you what's *behind* these fixes, so you'll learn a lot about relational databases along the way.

Bill Karwin
(352 pages) ISBN: 9781934356555. $34.95
http://pragmaticprogrammer.com/titles/bksqla
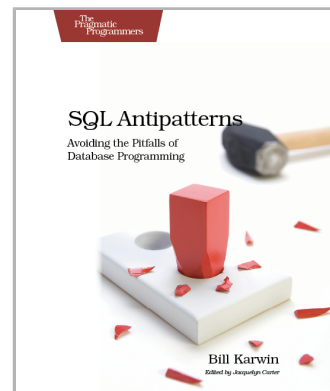
The Pragmatic Bookshelf

The Pragmatic Bookshelf features books written by developers for developers. The titles continue the well-known Pragmatic Programmer style and continue to garner awards and rave reviews. As development gets more and more difficult, the Pragmatic Programmers will be there with more titles and products to help you stay on top of your game.

Visit Us Online

This Book's Home Page
http://pragprog.com/titles/tbcoffee
Source code from this book, errata, and other resources. Come give us feedback, too!

Register for Updates
http://pragprog.com/updates
Be notified when updates and new books become available.

Join the Community
http://pragprog.com/community
Read our weblogs, join our online discussions, participate in our mailing list, interact with our wiki, and benefit from the experience of other Pragmatic Programmers.

New and Noteworthy
http://pragprog.com/news
Check out the latest pragmatic developments, new titles and other offerings.

Buy the Book

If you liked this eBook, perhaps you'd like to have a paper copy of the book. It's available for purchase at our store: *http://pragprog.com/titles/tbcoffee*

Contact Us

Online Orders:	*http://pragprog.com/catalog*
Customer Service:	*support@pragprog.com*
International Rights:	*translations@pragprog.com*
Academic Use:	*academic@pragprog.com*
Write for Us:	*http://pragprog.com/write-for-use*
Or Call:	+1 800-699-7764