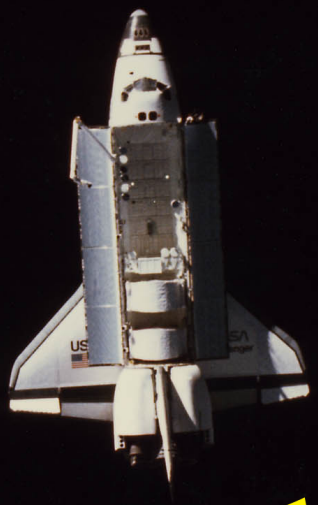

MASTER SPACE AND TIME WITH JAVASCRIPT

BOOK 1: THE BASICS

Noel Rappin

a noelrappin.com publication

Master Space and Time With JavaScript

Book 1: Basics

By Noel Rappin
http://www.noelrappin.com

© Copyright 2012 Noel Rappin. Some Rights Reserved.

Release 004 October, 2012

The original image used as the basis of the cover is described at http://commons.wikimedia.org/wiki/File:Challenger1983.jpg as a scene of the Space Shuttle Challenger taken during STS-6 in 1983 with a 70mm camera onboard the shuttle pallet satellite. According to NASA policy, the photo is in the public domain.

Master Space and Time With JavaScript by Noel Rappin is licensed under a Creative Commons Attribution-NonCommercial-ShareAlike 3.0 Unported License .

Chapter 1

Welcome to *Master Space and Time With JavaScript*

Thanks for purchasing (hopefully) or otherwise acquiring (I won't tell, but it'd be nice if you paid...) *Master Space and Time With JavaScript*.

Here are a few things I'd like for you to know:

Section 1.1

What have I purchased?

Master Space and Time With JavaScript is a book in four parts. All four parts are available at http://www.noelrappin.com/mstwjs.

You're reading *Part 1: The Basics*, which is available for free. It contains an introduction to Jasmine testing and jQuery, plus a look at JavaScript's object model.

Book 2: Objects in JavaScript, is currently available for $7. It contains more complete examples of using and testing objects in JavaScript, including communication with a remote server and JSON.

Book 3: Backbone, is currently available in beta for $7. The beta currently consists of about half of the finished book. It continues building the website using Backbone.js to create single-page interfaces for some more complex user interaction.

Book 4: Ember, is currently available in beta, also for $7. It builds out completely different parts of the website using Ember.js to create complex client-side interactions.

You may pre-purchase all four parts of the book for $15, a $6 savings over buying all three parts separately. This deal may not be available indefinitely.

This book has no digital rights management or copy protection. As far as I am concerned, you have much the same rights with this file as you would with a physical book. I would appreciate if you would support this book by keeping the files away from public file-sharing servers. If you do wish to distribute this book throughout your organization, it would be great if you would purchase additional copies or contact me at `noel@noelrappin.com` to work out a way for you to purchase a site license.

Section 1.2

Who Are You? Who? Who?

Inevitably, when writing a book like this, I need to make some assumptions about you. In addition to being smart, and obviously possessing great taste in self-published technical books, you already know some things, and you probably are hoping to learn some other things from this book.

In some ways, intermediate level books are the hardest to write. In a beginner book, you can assume the reader knows nothing, in an advanced hyper-specific book, you don't really care what the reader knows, they've probably self-selected just by needing the book. In an intermediate book, though, you are potentially dealing with a wider range of reader knowledge.

Here's a rundown of what I think you know:

JavaScript: I'm assuming that you have a basic familiarity with what JavaScript looks like. In other words, we're not going to explain what an `if` statement is or what a string is. Ideally, you're like I was a several months before I started this project – you've dealt with JavaScript when you had to, then had your mind blown by what somebody who really knew what they were doing could do. You specifically do not need any knowledge of JavaScript's object or prototype model, we'll talk about that.

Server Stuff: Since this is a JavaScript book, the overwhelming majority of the topics are on the client side and have nothing to do with any specific server-side tool. That said, there is a sample application that we'll be working on, and that application happens to use Ruby on Rails. You don't actually need to know anything about Rails to run the JavaScript examples, though if you are a Rails programmer, there will be one or two extras. It will be helpful if you are good enough at a command prompt to install the sample application per the instructions later in this preface.

CoffeeScript: I'm not assuming any knowledge of CoffeeScript. If you happen to have some, and want to follow along with the examples using CoffeeScript, have at it, I'll provide a separate CoffeeScript version of the code repository. NOTE: This isn't ready yet, don't go looking for it.

Testing Tools: I'm not assuming any knowledge of testing tools or of any test-first testing process. We'll cover all of that.

jQuery: I'm not assuming any prior jQuery knowledge.

Backbone.js: I'm not assuming any prior Backbone.js knowledge.

Ember.js: I'm not assuming any prior Ember.js knowlege.

Section 1.3

What to Expect When You Are Reading

On the flip side, it's fair of you to have certain expectations and assumptions about me and about this book. Here are a couple of things to keep in mind:

- I firmly and passionately believe in the effectiveness of Behavior-Driven Development as a way of writing great code, especially in a dynamic language like JavaScript. As a result, we're going to write tests for as much of the functionality in this book as is possible, and we're going to write the tests first, before we write the code. If you are completely unfamiliar with testing, this may be a challenge in the early going, as we're juggling Jasmine and jQuery. Don't worry, you can do it, and the rewards will be high.
- This book is focused on the current versions of the languages and libraries available. As I write this, that means ECMA Script 5 for JavaScript, jQuery 1.7.2, Jasmine 1.2.0, Backbone.js 0.9.2, Ember.js 1.0 RC3 and Rails 3.2.x. Keeping up with current versions is hard enough, without worrying about the interactions among multiple versions.

Section 1.4

But is it finished?

This book is incomplete. You will be notified from time to time that a new version of the book is available.

Here's a partial list of things that still need to be done in Books 1, 2, and 3:

- Formatting for Kindle and ePub versions is still a little wonky in spots. I'm working on it.
- The directions for setting up the sample application probably need to be improved.
- Book 4 is only half complete – there's another extended example that's coming.
- Something only you know – if you think there's something missing in the book let me know via any of the mechanisms listed below.

Section 1.5

What if I want to talk about this book?

Please do! The only way this book will be distributed widely is if people who find something useful in it tell their friends and colleagues.

You can reach me with email comments about the book at `noel@noelrappin.com`. Or you can reach me on twitter as `@noelrap`. If you want to talk about the book on Twitter, it'd be great if you use the hashtag `#mstwjs`, which gives me a good chance of seeing your comment.

This book has mistakes, I just don't know what they are yet. If you happen to find an error in the book that needs correcting, please use the email address `errata@noelrappin.com` to let me know.

There's also a discussion forum for the book at `http://www.noelrappin.com/mstwjs-forum/`. You do need to sign up in order to post, which you can do at `http://www.noelrappin.com/register-for-book-forum/`

Section 1.6

Following Along

The source code for this book is at https://github.com/noelrappin/mstwjs_code. The server side part of the code of this is a Rails application. You won't need to understand any of the Rails code to work through the examples in the book, but you will need to make the application run. Also, a basic knowledge of the Git source control application will help you view the source code.

I've tried to make this simple. The external prerequisites to run the code are Ruby 1.9 and MySQL. RailsInstaller http://www.railsinstaller.org is an easy way to get the Ruby prerequisites for this application installed if you do not already have them. MySQL can be installed via your system's package manager or from http://www.mysql.com.

Once you have the prerequisites installed and the repository copied, you can set everything up for the system by going to the new directory and entering the command `rake mstwjs:setup`. This command will install bundler, load all the Ruby Gems needed for the application, and set up databases. Then you can run the server with the command `rails server`, and the application should be visible at `http://localhost:3000`. Please contact noel@noelrappin.com if you have configuration issues with the setup, and we'll try to work through them.

The git repository for this application has separate branches for each section of the book with source code. Code samples that come from the repository are captioned with the filename and branch they were retrieved from. In order to view the branch, you need to run `git checkout -b <BRANCH> origin/<BRANCH>` from the command line.

Okay, let's get on with it.

Chapter 2

Introduction to the Fabulous Powers at Your Command

One day, you get an innocuous-looking email. You don't know it, but it will change your life.

> Dear Madam or Sir,
>
> I am not a Nigerian prince. I need urgent help with the web site for my business *Time Travel Adventures*, a travel agency for those who move through the fourth dimension.
>
> Unfortunately, I have miscalculated and the site that has been built is unsuitable for web consumers of your time. I need a developer of your skill and expertise to bring it up to the standards needed.
>
> Please see the attached code to get a sense of the problem. I need these fixes yesterday. In my business, time is money. For real.
>
> Sincerely, Doctor What

Well, that's odd, you think to yourself. Still, a client is a client, so you read through the code. And, time travel or not, it is strange. The server-side code is fully 2012 compliant: Rails 3.2, HTML5, CSS 3 and the like.

It looks like this:

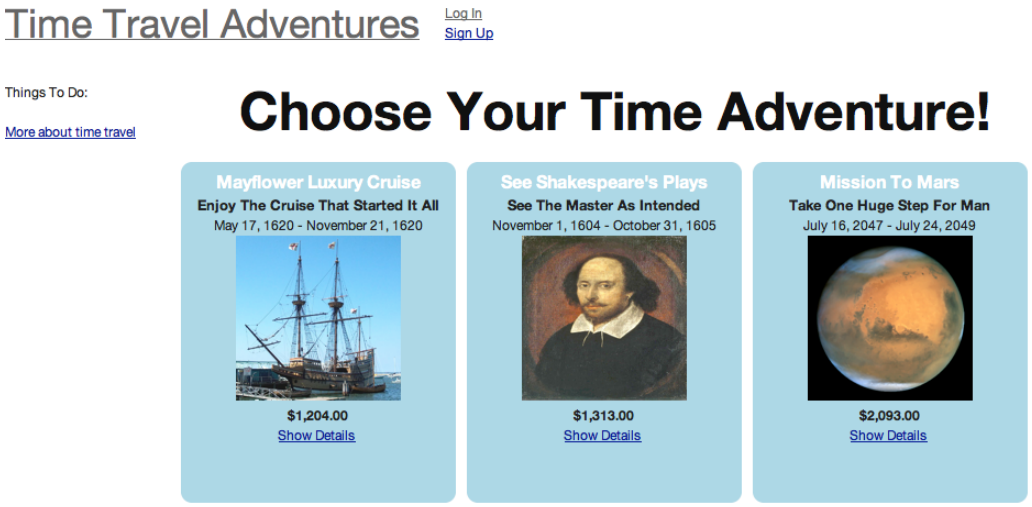

Figure 1: Time Travel Home Page

But there's no JavaScript. At all. Not even any CoffeeScript. And for 2012 and beyond, that just seems wrong. We're going to accept Dr. What's unusual commission, and turn his site into the kind of dynamic, exciting web presence that will make people want to travel through time.

Section 2.1

A Great Plan! That's What We Need

Over the course of this book, we're going to take several requests from Dr. What and use various JavaScript libraries to turn them into part of the Dr. What site. If you want to follow along, you can download the application code at https://github.com/noelrappin/mstwjs_code. The git repository is set up so that various branches in the repository match different points in the code base. As we progress through the book, you can keep up with the examples by changing to a later branch in the repository.

Each example in the book will start with a request from Dr. What, and then we will use a Behavior-Driven Development (BDD) process to write the code. We'll use the Jasmine library to write the tests – don't worry, we'll explain how Jasmine works along the way – and then we'll use common JavaScript libraries like jQuery and Backbone.js to actually implement the features. We'll start with small things, but by the end, we'll build up some complex rich client-side features.

Our first request is coming right now.

The Inevitable CoffeeScript Sidebar

CoffeeScript. It's what's for breakfast. Okay, it's really "a little language that compiles into JavaScript." It's designed to be a simpler, lower ceremony language that replaces JavaScript's surplus of curly braces and semi-colons with constructs that might be more familiar to Ruby and Python programmers. In this book, a sidebar like this one will be used if CoffeeScript behaves significantly different than JavaScript.

Chapter 3

Hide and Seek

No sooner do you agree to work on the Time Travel Adventures project then another email shows up in your inbox:

> Dear You.
>
> It's time to get things started. The page on our site that shows all our trips is boring. The first thing I'd like you to do is to add a "Show more details" note to each trip on that listing page. When the user clicks on the link, we should show them more details. When they click on the link again, the details should vanish. Oh, and the text of the link should change to reflect the new state.
>
> Thanks, Doctor What

Although this is a relatively easy thing to do in JavaScript, it touches a lot of different facets of basic jQuery.

We're going to walk through the process of writing this code in jQuery, using Jasmine as a testing framework to drive the process of creating our actual program code. This chapter is going to be an overview, touching on the most important facts about jQuery and Jasmine without getting into the details. We'll follow this chapter with more detailed examinations of both Jasmine and jQuery.

Section 3.1

Our first Jasmine Test

We use Jasmine to express the behavior we want in code, a process called *Behavior Driven Development* or BDD. Specified in words, the requirements look like this:

- When the user first loads the page, none of the detail elements for the trips are visible.

- When the user clicks on a "Show Details" link, the DOM element associated with that link becomes visible.
- Also when the user clicks on "Show Details", the caption text of the link changes to "Hide Details".
- When the user clicks on a "Hide Details" link, both things happen in reverse: the DOM element goes away, and the caption changes back.

To start the BDD process, we want to write a simple test that automates all or part of one of these requirements. We'll start with the requirement that the description shows up when the user clicks on a link – the idea that the detail elements start out as hidden is sort of implied.

We're going to build out this first piece of Jasmine line by line. If you are playing along at home, this code will eventually go in `spec/javascripts/togglerSpec.js`.

Here's a skeleton of that first spec. Eventually it will specify that the trip description is being displayed.

```
describe("Trip detail toggler", function() {
  describe("clicking a show link", function() {
    it("shows the trip description", function() {
      // Spec body will go here
    });
  });
});
```

Sample 3-1-1: This is just a skeleton of the full Jasmine spec.

Jasmine tests are just JavaScript, although they make heavy use of JavaScript's ability to create anonymous functions which can be used anywhere a regular variable can be used. Since anonymous JavaScript functions can be invoked at an arbitrary time after they are declared, they are often used for callback behavior. The key point for Jasmine specs is that the body of the functional argument will be executed not when the function is declared, but later when the tests are run.

Even this small skeleton uses two of the most important functions in the Jasmine library, `describe` and `it`. The `describe` method declares a group of related Jasmine specs called a *suite*. Typically, a suite groups specs that are related either because they are all exercising the same part of the code, or because they all require a common setup, or both. The `describe` method takes two arguments, a string description of the suite and a function in which individual specs are declared.

The Pedantic Naming Sidebar

BDD tools in general, and Jasmine in particular, try to focus your attention on the idea that you are specifying code yet to be written, not validating code that already exists. So, for example, Jasmine uses `should` rather than `assert` to make verifiable statements about the code.

In particular, the individual unit of a Jasmine suite is often referred to not as a "test", but rather as an *example*, *specification*, or *spec*. I try to stick to BDD terms, but often find "spec" to be a little awkward to use. If I slip, and refer to a Jasmine "test", promise that you'll remember that it's still supposed to be written first.

Inside a `describe`, individual specs are defined with the `it` method. Like `describe`, `it` has a string argument for a description and a function argument. In this case, the function argument is the actual body of the spec.

In this example, we've nested an inner `describe` call inside our outer `describe` call, allowing us to group all our specs related to clicking the "show" link together. Having a group of common specs allows us to have several specs share a common setup, it also makes both the spec file and the test output easier to read.

Our actual executable test will start inside the argument to the `it` method, in the commented part of our code shown above. Now we need to decide exactly what to put in the test. Conceptually, a test has three parts:

- Set up any background data the test needs. In a small unit test like the ones we are writing here, you want to have as little data as you can.
- Actually do something. That is, execute some of your application's logic.

- Validate that the behavior or state of the program is as specified.

These three phases are sometimes referred to as "Given/When/Then".[1] Even if we don't explicitly use those terms in our test, the three part structure is still a useful way to think about testing.

The first part of our test is data setup. We don't have any traditional data in this test – we're not setting up any objects. But our eventual code will depend on the existence of a particular structure in our HTML page that we need to represent as DOM elements for the test to work. Specifically, we need a DOM element that is clicked and another DOM element that contains the detail text that is shown and hidden.

If you look at the view code in the actual application, (in `app/views/trips/index.html.erb`), you see that the clickable link is an HTML anchor with a CSS class `detail_toggle`, and the detail is stored in an ordinary `p` tag with a CSS class `detail`. The `detail` `div` element also has a CSS class `hidden` which – big surprise – we use to indicate that the element is not shown. The link and the detail elements each share a common parent, a `div` with the CSS class `trip_links`.

It's common for client-side JavaScript to be dependent on DOM elements in order to work, and we'll talk about various ways to manage that dependency throughout the book. Right now, it's enough to declare the dependency. We want to show that HTML structure with only the details that are important for our test, something like this:

Filename: spec/javascripts/fixtures/one_index_trip.html (Branch: section_2_2)

```
<div class="trip_links">
  <a href="/trips/1" class="detail_toggle">Show Details</a>
  <p class="detail hidden">This text should be hidden.</p>
</div>
```

Sample 3-1-2: The basic HTML that is the input data for our first spec

The generic name for a data setup that is used as the background for a test is a *fixture*. We need to get our HTML fixture embedded into the DOM tree that is visible to the Jasmine test at runtime. This is a common need when testing jQuery, and the `jasmine-jquery` extension library provides a solution.[2]

1. You can explicitly use a Given/When/Then structure in your code with Justin Searls' `jasmine-given` library, available at https://github.com/searls/jasmine-given.

We can save our HTML fixture to a file in the `spec/javascripts/fixtures` directory of our project, then we can use jasmine-jquery's `loadFixtures`. The `loadFixtures` function takes the HTML from the specified file and embeds it into the DOM tree for the environment in which Jasmine is running. Here's what our spec looks like with the fixture being set as our given:

```
it("shows the trip description", function() {
  loadFixtures("one_index_trip.html");
  init();
  // next comes the When
  // then the then
});
```

Sample 3-1-3: Our first spec, with a given

The `init` thing is a little tricky, not in the sense of being complicated, but in the sense of being a trick we do here to make the tests work. Essentially, we'll be reinitializing our jQuery code after the fixture is loaded so that the jQuery event system knows about the DOM elements that are in the fixture. We'll explain more when we talk about how jQuery starts up.

Next is our action. The action we are testing is a user clicking on the link. We need to use a little bit of jQuery to make that happen. I realize we haven't talked about jQuery yet. We'll get there, promise. Meantime, we don't need much jQuery here, just a single line:

```
it("shows the trip description", function() {
  loadFixtures("one_index_trip.html");
  init();
  $(".detail_toggle").click();
  // then the then
});
```

Sample 3-1-4: Our first spec, with a given and a when

Our line of jQuery here is doing two things. The first half of the line, `$(".detail_toggle")`, uses jQuery to find DOM elements that have the CSS class `detail_toggle`. For the moment, we don't need to know much about what jQuery is doing, except that it returns an object that allows you to interact with the matched elements. In our case, the jQuery matches the single

2. Actually, our Rails project depends on the `jasminerice` gem, which includes a slightly modified version of `jasmine-jquery`.

anchor link in our fixture file. Then we call `click` on the object, which triggers a click event that can be handled in our code.

Okay, we've set up a Jasmine spec, added some data, done something... Now it's time for a moment of Then. We need to make a claim about the state of the world after the click. Specifically, we want the detail div to be visible. Since we are using the `hidden` class to manage visibility state, we want to specify that the `hidden` class is no longer part of the element after the click. We can do that in one line of Jasmine. Well, one line of Jasmine plus the `jasmine-jquery` library. The completed spec looks like this:

Filename: spec/javascripts/togglerSpec.js (Branch: section_2_2)

```
describe("Trip detail toggler", function() {
  describe("clicking a show link", function() {
    it("shows the trip description", function() {
      loadFixtures("one_index_trip.html");
      init();
      $(".detail_toggle").click();
      expect($('.detail')).not.toHaveClass("hidden");
    });
  });
});
```

Sample 3-1-5: Our complete first spec

The Jasmine line has two parts. In the first half, `expect($('.detail'))`, we use the Jasmine `expect` method to specify what object in the application we would like to make a testable claim about. In this case, we are using jQuery to identify an element with the CSS class `detail`, which will be our detail element. The return value of `expect` is a proxy object. The proxy's goal in life is to respond to Jasmine matcher methods and apply the matcher to the object that was the original argument to `expect`.

In our case, the matcher is `toHaveClass`, which is defined by Jasmine (well, technically, it's defined in the `jasmine-jquery` library). The `toHaveClass` matcher assumes that the original object is a jQuery data structure and checks to see if that DOM element has the associated CSS class. In our case, we also use the `not` property so that the matcher is inverted – the specification is true if the class is *not* present.

Section 3.2

Running the first test

We've written a test, so now we get to run it. There's a number of different ways to run a Jasmine test, for the moment we're only going to focus on the one that ties tightly with our Time Travel application. Since that is a Rails 3 application using the `jasminerice` gem, we can start the regular Rails server, open a web browser, and go to `http://<HOSTNAME>/jasmine`. You should see something like this:

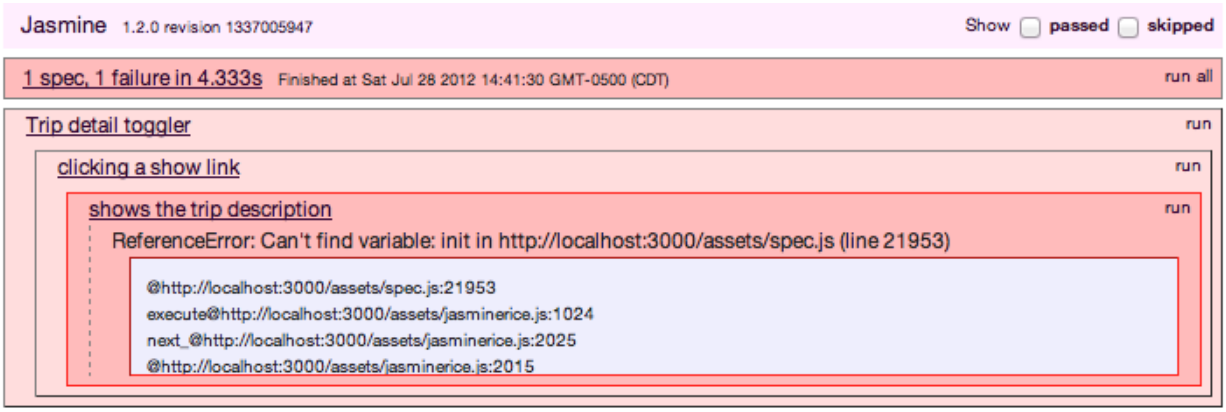

Figure 2: Initial Jasmine Failure

The bright shiny red color indicates a test failure. Specifically, we have the message:

```
ReferenceError: Can't find varaible init (etc...)
```

We get that error because there is no `init` function defined, and we try to call it. The simplest thing we can do to clear that error is to create an empty `init` function, which would go in a `app/assets/javascripts/toggler.js` file:

```
function init() {

}
```

Sample 3-2-1: A blank init function so that the tests will run

With the `init` function in place, we can run our Jasmine tests again by reloading the page. This time we get a more useful error message.

```
expected '<p class="detail hidden">This text should be hidden.</p>'
not to have class 'hidden'.
```

Sample 3-2-2:

We're failures! Which is great. Being able to deal with temporary failure is a very useful mental trick when test-driving new code.

The BDD process now tells us to write the simplest thing that could possibly work. We're going to walk through enough jQuery to build enough of our actual working toggle handler so that it actually changes the CSS class of the correct element.

Section 3.3

Getting jQuery's Attention

I'm going to describe the jQuery code line-by-line, in much the same way we just went through our Jasmine spec.

The jQuery library will take care of three things for us:

- Identifying the link and detail DOM elements
- Associating an action with a click event
- Actually changing the state of the detail DOM element.

First, though, we need for the jQuery code to actually be loaded into the page. For that to happen, all our code needs to be inside a particular jQuery event callback. The skeleton of our code looks like this:

```
function init() {
  // All our toggler functionality goes here
};

$(function() {
  init();
});
```

Sample 3-3-1: Our jQuery skeleton, showing the jQuery function as a shortcut to the document ready event. This will go in app/assets/javascripts/toggler.js.

If you've worked in jQuery, or seen jQuery code, that structure should look pretty familiar, since pretty much all jQuery code has to be inside that `$(function() {});` structure.

The `$` is special to jQuery. It indicates a function, often just called the *jQuery function*, that is the primary point of entry into jQuery.[3] The jQuery function has wildly different behavior depending on its arguments.

Right here right now, the jQuery function is being called with a single argument that is itself a function. In that circumstance, the jQuery function is a shortcut to a specific jQuery event handler. The handler being called responds to an event called `document.ready` which is fired when all the DOM elements for the page have loaded, but before any large assets have completed being downloaded.

Our snippet is interpreted as part of our browser's normal page load. When the page load is complete, the `document.ready` event happens, the anonymous argument function is invoked, and the code inside the body of that function is executed. In our case, that interior code will identify the elements we want to manage and specify what we want to do to those elements.

The problem that jQuery is solving is that any JavaScript code that depends on DOM elements of the page can only be safely loaded after all the DOM elements have themselves been loaded. If the JavaScript loads too early, then there may be errors if the JavaScript depends on the existence of DOM elements that haven't loaded yet. Since jQuery keeps our logic from being loaded until the document is ready, we won't have that particular problem.

You can have as many `document.ready` callbacks as you want – they will be executed in the order they were loaded when the event is triggered. As we'll see later on, this is just a special case of how jQuery handles all events.

The Dollar Sign, jQuery, and You

The `$` is a popular symbol in JavaScript, and jQuery is not the only JavaScript library to claim it. In jQuery, the `$` is an alias for the global object named `jQuery`. If you are in a situation where jQuery must coexist with another library that has a prior claim on the dollar sign, you can use the alias `jQuery` to access the same global object. If you are using another library, such as Prototype, that depends on controlling the `$` symbol, then you can include a call to `jQuery.noConflict()` in your JavaScript setup. The `noConflict` method eliminates jQuery's alias to the `$`, and therefore prevents name collisions with other libraries.

3. We'll see later on that the `$` also defines an object that has its own properties and methods.

We're placing all our jQuery logic in a separate `init` function then calling that `init` function from within the document ready callback. There are a couple of reasons why we might do this. First is that it gives us at least the possibility of a little bit of structure to our jQuery code, and second it lets us call the same initialize logic from our Jasmine tests as we do from the jQuery code.

Now that we have jQuery's attention, we need to give it something to do. The first thing we need is to identify the DOM elements that our code needs to act on.

Section 3.4

Show me some ID. And some class.

jQuery is a great library for changing DOM elements. But if you want to change a DOM element, first you have to find it. Luckily, jQuery is also great at finding DOM elements.

For our toggler, we need to identify two different kinds of DOM elements. First, the "Show Detail" link that the user clicks on, and second, the page element containing the description text that actually appears and disappears. We need to make a design decision as to how these elements will be identifiable in our HTML page.

There will be multiple trips on the page, so we will have multiple sets of DOM elements – one pair for each trip – that need to share a common characteristic, yet still be uniquely identifiable. Giving the elements the same CSS class works for the common part. For the unique part, in this case, it's enough to have each separate link and detail element share their own common parent – we can identify the subordinate links in relation to their parent.

Actually, we've already made this design decision, at least in part, when we specified that the HTML fixture would look like this:

Filename: spec/javascripts/fixtures/one_index_trip.html (Branch: section_2_2)

```
<div class="trip_links">
  <a href="/trips/1" class="detail_toggle">Show Details</a>
  <p class="detail hidden">This text should be hidden.</p>
</div>
```

Sample 3-4-1: Once again, our HTML test fixture

That gives us a parent element with a CSS class of `trip_links` with a subordinate element whose CSS class is `detail_toggle` and another whose CSS class is just `detail`. We will use that relationship in our jQuery calls to make sure that a toggle click changes only the detail that we want. In a very real way, we're using the testing process to make design decisions – in this case, about the structure of our output, in later cases, about the architecture of our code.

This particular design decision highlights a reason why test-driven development is never a complete replacement for other kinds of validation. We now need to remember to make our actual code have the same HTML structure that the test is using – if we don't, the code may pass all tests but still fail in the browser. Validating with tests is a topic we'll come back to as we build more complex examples.

We use jQuery *selectors* to match elements on our page. A selector is a pattern that jQuery uses to identify DOM elements, using a syntax that is very similar to CSS syntax. The simplest jQuery selector is simply an HTML tag, like `span` or `img`.

To use jQuery to match a selector, we call the same jQuery `$` function, but with a string argument instead of a functional one. When jQuery is called with a selector as an argument, the function returns an instance of an imaginatively named *jQuery object*, which contains a list of all the DOM elements that match the selector.

jQuery defines a lot of methods on the jQuery object that you use to access and manipulate the encapsulated DOM elements. The genius of the jQuery object structure is that it often frees us from having to care how many elements are in the object, since many of these jQuery object methods will happily apply themselves to all elements in the object, and many others will apply themselves to just the first element. As we'll see later on, it's easy to use a jQuery object to bind an event to multiple DOM elements at once.

In the toggler code, we'll need to use selectors to identify both the set of links that we want to click on, and the set of detail elements they will affect. We can identify our toggle links easily enough, with a selector `.detail_toggle`, matching all elements on the page with the CSS class `detail_toggle`, the same meaning the selector would have in a CSS file:

```
$(function() {
  init();
});

function init() {
```

```
  var $toggleLinks = $(".detail_toggle");
  // more stuff coming, hold your horses.
}
```

Sample 3-4-2: Using jQuery to find our detail toggle links

The use of a leading dollar sign in the variable name `$toggleLinks` is a convention to remind us that the variable in question is a jQuery object.

We still need to get our detail element itself, but in order to fully do that we need to be able to respond to the click event on the toggle link.

Section 3.5

Making Things Happen After Other Things Happen

In order to make your web application responsive to client events like clicking on targets or mouse hovers, you need to identify blocks of code to be executed when the event happens. This process is called *event binding*, and the associated code to be executed when the event happens is a *callback*. In jQuery, an event callback can be associated with the results of any selector, making it easy to make multiple elements on the page share behavior.

In our toggle code, we want to associate the mouse click event with any of the DOM elements with the CSS class `detail_toggle`. We've already shown how to find all those elements, now we can use the jQuery `click` method to define an event handler. Our toggle code now looks like this:

```
$(function() {
  init()
})

function init() {
  var $toggleLinks = $(".detail_toggle");
  $toggleLinks.click(function(event) {
    // code to be executed when the user clicks
```

```
  });
}
```

Sample 3-5-1: Now we have a click handler. This will go in app/assets/javascripts/toggler.js.

The `click` method is defined on the jQuery object, and takes one argument, which is a function.[4] When any of the DOM elements contained in that jQuery object are clicked, then the function is invoked. In addition to `click`, jQuery defines shortcut event methods for all the standard DOM events, such as `mousedown` or `blur`.

Inside the `click` method, jQuery assigns the special JavaScript variable `this` to the actual DOM element that has been clicked on. Now that we have the element that was clicked on, we can use it to uniquely identify its sibling detail element.

```
$(function() {
  init();
})

function init() {
  var $toggleLinks = $(".detail_toggle");
  $toggleLinks.click(function() {
    var $detail = $(this).siblings(".detail");
    // the rest of our handler goes here
  });
}
```

Sample 3-5-2: Inside the click handler, we find the detail link. This will go in app/assets/javascripts/toggler.js.

We've only added one line, but there's a lot going on in it. The line we've added is inside the handler function, and allows us to find the detail element related to the link that was clicked on. We are chaining together two different jQuery library functions.

The first part of our chain is `$(this)`, which takes the target of the event – remember that jQuery assigns the event target to `this` any time it invokes an event handler – and wraps the event target inside `$` to call the jQuery function. This time, the argument to the jQuery

4. Yes, it's the same `click` method we saw in the Jasmine spec – with no arguments it triggers a click, with one argument, it binds an handler to be executed when the event happens.

function is a DOM element and the jQuery function returns a jQuery object wrapping that DOM element. Because the element is now wrapped in a jQuery object, we can bring the full force of the jQuery library on it.

We need to identify the unique DOM element that is a sibling to our link and which has the CSS class `detail`. We do that with one of jQuery's traversal methods, namely, `siblings`. The `siblings` method returns a new jQuery object containing all the DOM siblings of all the elements in the original jQuery object. If you call `siblings` with a selector as an argument – as we have with `.detail` – then that selector is a filter and only sibling elements that match that selector are returned. This is exactly what we need – the sibling method returns the detail element we are interested in and only that element. Other detail elements belonging to other trips won't be DOM siblings of this click link and so won't be in the list of elements to consider.

Before we leave the event handler, there's one other thing we need it to do, or more precisely, one other thing we need it *not* to do...

```
$(function() {
  init();
})

function init() {
  var $toggleLinks = $(".detail_toggle");
  $toggleLinks.click(function(event) {
    var $detail = $(this).siblings(".detail");
    // the rest of our handler goes here
    event.preventDefault();
  });
}
```

Sample 3-5-3: Preventing default event processing. This will go in app/assets/javascripts/toggler.js.

Our final line in the handler is `event.preventDefault()`, which is a jQuery method that stops the default behavior of an event from happening. Since our event here is a click on an anchor link, we're preventing the link from actually being followed and loading another page.

We've almost got our test passing, we've found the correct element, but we haven't actually done anything to it. We're using the CSS class `hidden` to mark the visible state of the item. So

what we need to do is change it. Or, we could use a jQuery method with what is, by now, a kind of familiar name.

Filename: app/assets/javascripts/toggler.js (Branch: section_2_2)

```
function init() {
  var $toggle_links = $(".detail_toggle");
  $toggle_links.click(function(event) {
    var $detail = $(this).siblings(".detail");
    $detail.toggleClass("hidden");
    event.preventDefault();
  });
}

$(function() {
  init();
})
```

Sample 3-5-4: Our test passing toggler code

That `toggleClass` method adds its argument to all the DOM elements of the jQuery object that don't have it, and removes the class from all the ones that do. It's very handy for something like changing the selected/unselected status of a lot of tabs in one line.

At this point, if you have all this code in place, and you run the Jasmine specs at `http://localhost:3000/jasmine`, then you should see the green rectangle indicating the single spec passing.[5]

In order for this to work in the actual browser, we do need to add one small CSS definition:

```
.hidden {
  display: none;
}
```

Sample 3-5-5: This will go in the app/assets/stylesheets/application.css file

Yay! Our tests pass! We're a success! Well... we're not done yet, but first, a brief time out while we think about what we've done so far.

5. Watch out for a gotcha here – if the JavaScript spec code doesn't parse, no tests will load, but Jasmine will display the zero spec case in green as if all specs passed. Make sure that the rectangle doesn't say "0 specs, 0 failures". Also make sure that the JavaScript console doesn't mention any parse errors.

Section 3.6

A Brief Testing Retrospective

We've completed one red/green test cycle, and it's a good time to take a deep breath and reflect a little bit about what we've done.

A consistent struggle when teaching BDD is that going through the steps slowly makes it seem like the process takes forever. Here we've spent pages and pages on one test and one pass and we're not even done yet. At this point, you likely have at least two questions: Will this always take this long? and Will it be worth it?

My answers are "no", and "yes" (respectively, of course).

On the time issue, this seems slow in part because we've been absorbing a new idea on each line. Once you have the basics down, the cycle we just did takes a few minutes. We'll talk about this more next chapter, but you'll also see that writing the tests first somewhat counterintuitively means it takes less time to write them.

As for the value, I'm not blind to the fact that so far, all we've done is take an extra several lines to write really simple JavaScript code that many of you could have written perfectly fine on your own. Bear with me, I think you'll start to see the value as we finish this chapter. We'll talk more about the idea of why BDD works next chapter, and then you'll see it in practice as we tackle more complex examples.

I also wanted to make a couple of specific points about the actual tests, especially for those of you that already know a little Jasmine or jQuery and are wondering what the hell I'm doing.[6]

This one test that we've written is in some ways a top-down test, because it's starting from a user action – the click. It's also feasible to start from the logic and work up, which probably would have meant starting by testing something like being able to find the detail element given the link. We would have wound up in more or less the same place, and I thought that top down would be an easier way to see the test process.

6. Due to boring circumstance, I've been writing about this tiny example for about six months. I've had a *lot* of thoughts about how this test goes.

If you know jQuery, you may be wondering why I'm using a CSS class to manage show/hide behavior when jQuery has perfectly acceptable `show` and `hide` methods. Good question. My experience is that the effect of those methods tends to be unpredictable and a little hard to catch on the test side, whereas it's easier to detect the presence or absence of the hidden class, so I tend to work hidden behaviors directly via CSS.

Also, the need to explicitly call `init` from the Jasmine tests is a bit awkward. We need to do that largely because our Jasmine fixture is loaded after the initial jQuery load. Later, we'll talk about jQuery *delegated events* that minimize the need for reloading. And when we start more explicitly organizing the JavaScript into modules and classes, setting up that data in the Jasmine tests will become less awkward.

Finally, you can make an argument that I've cheated the BDD process here, specifically our current jQuery code goes beyond the simplest thing that could possibly work. Specifically, the jQuery code would already pass two potential next tests. I won't make you guess. The jQuery code would successfully manage the case where we click again and want the detail to hide, since we used `toggleClass` as opposed to `removeClass`. Also, since we used `siblings`, the jQuery code is robust against having more than one trip on the page.

You are undoubtedly wondering what the problem is – the code works, writing a less-working version in order to force another test seems backward. And in this case, where the code as written is not any more complex than the less-functional code, it isn't that big a deal. When code gets more complex, though, it is a good idea to move in small steps, and it's a good idea to avoid writing tests that you know should already pass. Again, this is a topic we'll return to throughout the book.

Section 3.7

The Rest of the Tests

Without much further blathering, let's get the rest of the tests in place. Our next requirement is to change the text of the link from "Show Details" to "Hide Details":

Filename: spec/javascripts/togglerSpec.js (Branch: section_2_9)

```
describe("clicking a show description link", function() {
  beforeEach(function() {
    loadFixtures("one_index_trip.html");
    init();
```

```
    $(".detail_toggle").click();
  });

  it("shows the trip description", function() {
    expect($('.detail')).not.toHaveClass("hidden");
  });

  it('changes the link action to "Hide"',function() {
    expect($('.detail_toggle')).toHaveText("Hide Details");
  });
});
```

Sample 3-7-1: First there was one test, now there are two!

There are two new things in this test – a new expectation about the text, and a refactoring of the tests. The expectation is in the last test, and does what you probably expect from reading it, namely verify that the link element changes its text.

Since both the description visibility and link text specs share a common setup, I've prevented some duplication by moving the setup out of the spec itself and into a Jasmine `beforeEach` function. The `beforeEach` function executes, yes, before each spec[7] – the `jquery-jasmine` library ensures that the DOM is automatically cleared after each test, so we don't have to worry about that.

Moving the common setup to the `beforeEach` benefits us by making the individual tests shorter and more readable, and also by ensuring that common setups are actually identical. That said, placing each individual expectation in its own test is a style choice with a tradeoff. On the plus side, each expectation runs separately, making it easier to accurately gauge the state of the code. (Although Jasmine, unlike other test tools, doesn't halt the execution of an individual spec just because an expectation fails.) On the down side, well, each expectation runs separately, which means the `beforeEach` code runs multiple times, which means that tests written in the one-expectation-per-spec style are going to run slower. Which way to go is a matter of style. Unless the speed problem gets out of hand, I tend to prefer separate specs – it makes me feel like I'm accomplishing something.

7. Parenthentically, I always feel a little silly defining what a function does when the name of the function already conveys it. You know, like `beforeEach`. I realize I have to actually define things, but sometimes... Also, it's very early in the book and I already feel like I'm running out of synonyms for "element", as in DOM element.

The test fails, of course. After making a big deal about getting ahead of the first spec, I'm actually going to just do the dumb thing that passes the test here:

Filename: app/assets/javascripts/toggler.js (Branch: section_2_9)

```
function init() {
  var $toggle_links = $(".detail_toggle");
  $toggle_links.click(function(event) {
    var $detail = $(this).siblings(".detail");
    $detail.toggleClass("hidden");
    $(this).text("Hide Details");
    event.preventDefault();
  });
}

$(function() {
  init();
})
```

Sample 3-7-2: Hey, it makes the spec pass. Which doesn't make it final.

Tests pass. Ship it! Okay, this is obviously not final. All I'm doing is using jQuery's `text` method to change the text of the link to "Hide Details". While that does pass the spec, it doesn't exactly inspire confidence.

What we need is another spec. Well, two to be exact.

Filename: spec/javascripts/togglerSpec.js (Branch: section_2_9_a)

```
describe("clicking a show description link", function() {
  beforeEach(function() {
    loadFixtures("one_index_trip.html");
    init();
    $(".detail_toggle").click();
  });

  it("shows the trip description", function() {
    expect($('.detail')).not.toHaveClass("hidden");
  });

  it('changes the link action to "Hide"',function() {
```

```
    expect($('.detail_toggle')).toHaveText("Hide Details");
  });

  describe('clicking the link again', function() {
    beforeEach(function(){
      $(".detail_toggle").click();
    });

    it('hides the description', function() {
      expect($('.detail')).toHaveClass("hidden");
    });

    it('Changes the link action back to "Show"',function(){
      expect($('.detail_toggle')).toHaveText("Show Details");
    });
  });
});
```

Sample 3-7-3: Then they told two tests, and so on...

What we've done here is added a `describe` function nested inside our existing `describe`. This inner suite has its own `beforeEach`, which calls `$(".detail_toggle").click();` again. When Jasmine goes to run these specs, both the outer and inner `beforeEach` methods are executed from the outside in, allowing us to have some specs referring to the state of the system after one click, and some specs referring to the state of the system after the second click. The specs return us back to the original state of the system – the `hidden` class is back on our detail element, and the toggle text is back to "Show Details".

Running these four specs, three of them pass. Our original two still pass, and the fact that we used `toggleClass` means that the CSS class behavior is as expected. However, we don't actually have any logic to change the text back.

Simple as this logic is, there are still a couple of ways to manage it, largely depending on whether we want to treat changing the text and changing the CSS as two different tasks or one task. Here's one quick way to get to a passing test:

Filename: app/assets/javascripts/toggler.js (Branch: section_2_9_a)

```
function init() {
  var $toggle_links = $(".detail_toggle");
```

```
  $toggle_links.click(function(event) {
    var $detail = $(this).siblings(".detail");
    if($detail.hasClass("hidden")) {
      $(this).text("Hide Details");
    } else {
      $(this).text("Show Details");
    }
    $detail.toggleClass("hidden");
    event.preventDefault();
  });
}

$(function() {
  init();
})
```

Sample 3-7-4: This code passes our toggler tests

All we've done here is use the `hasClass` method, which is a jQuery method returning boolean true or false, to check whether the detail element has the `hidden` class, then we change the link text appropriately.

Simple as the `init` function is, there probably is some room for refactoring. For one thing, I often prefer to have the boolean value guiding an `if` statement provided by a function with a semantically meaningful name rather than the raw boolean expression. In this case, naming the function something like `isDetailHidden` would make the intent of the logic clearer.

On the other hand, you could also argue that the code is checking the state of the detail element twice – once in the actual `if` statement and once as part of `toggleClass`. Technically, that's a duplication, though I think you'd have to be toggling a couple of jillion things before it made a significant difference. Still, it might be nice to have the state of the toggler be an actual piece of data, rather than something that we had to infer from the CSS class or the text of the link element.

We're going to put a pin in that right now. We'll come back to the idea of refactoring this code into its own structure in a couple of chapters when we talk about functions and JavaScript's object model.

Until we get to that point, this code passes the tests, we don't have more tests to write, and it's mostly simple. So, for the moment, we're done.

Section 3.8

Coming Attractions

We've taken a simple functionality request and applied a few rounds of Behavior-Driven Development to implement the feature. Along the way, we picked up the basic structure of how Jasmine and jQuery work.

At this point, you probably have questions, so we're going to back up and spend the next couple of chapters looking at Jasmine and jQuery in more detail. First up is Jasmine, where we'll answer questions of the "How does Jasmine work" and "Why will using Jasmine make my code awesome" variety. After that, we'll talk about jQuery, and dig into its feature set more deeply. Then we'll spin back to JavaScript and talk about objects, functions, and classes, as we refactor this toggler example a couple of times just because we can.

Chapter 4

More Jasmine Than A Disney Princess Convention

In the last chapter, we went through a basic Behavior-Driven Development process in the fast lane, without slowing down to see the details. Over the next two chapters we will show a little more of the landscape. In this chapter, we'll talk about Jasmine. Next chapter, we'll talk about jQuery.

After walking through an example of using Jasmine, you probably have at least two basic questions: "how do I use Jasmine in my projects?" and "why should I use Jasmine in my projects?". Let's talk about the why first, and explain what those of us who love test-first practices find that makes testing wildly helpful in developing code. After that, we'll talk about the how, both the details of the Jasmine library itself, and how to get Jasmine running on your project.

Section 4.1

Drive Your Behavior With Tests

In the Behavior-Driven Development (BDD) process, you write a test that describes a logical change to your program, which could be a new feature or could be a bug you are looking to squash. You then write the code that makes that test pass. The general process looks like this:

- Write a specification for a small change to your program.
- Write the simplest code that makes that test pass.
- Refine the code by cleaning it up, removing any duplication added, and improving the implementation, a process known as *refactoring*. This step is important. In a BDD system, this is where much of your design takes place.

- Repeat

Repeat until done. The process is often referred to as “Red/Green/Refactor”, since most test runners use red to indicate test failure and green to indicate test passage.

The key words in the description of the BDD process are “small”, “simple”, “refine”, and “repeat”. BDD is an iterative process, and its power comes from going back and forth between tests and code rapidly. By doing so, you tend to write your code in small methods, loosely joined, with few side effects. Code written in that style is easier to test because small methods without side effects have clear results that can be tested. The real benefit, though, is that small methods without side effects are also easier to maintain over time, both in that they are easier to debug and in that they are easier to change and grow.

In theory, if you follow this process, your code is always maximally simple and has full test coverage. Practice, of course, is never quite like theory (although in theory, they are the same...), but the BDD process is still a great way to attack a problem by splitting it into tiny, testable pieces.

The most common misconception about the BDD process is that it is primarily meant to validate code correctness. That’s not true. While code correctness is a wonderful thing to have, and having BDD tests certainly helps, the primary value of BDD is that the BDD process causes you to write better, more maintainable code. BDD is a great process for improving code quality, and it’s a very good process for determining that the code you are writing does what you think it does. BDD is not an effective process for determining that what you think the code is doing is actually the correct, real-world behavior of the code.

One of the clearest long-term benefits is that you wind up with a better class of bug. Most of your BDD bugs will be easily conceptualized as a test you forgot to write. The kind of bug that is caused by heavily stateful interactions of multiple parts of the code becomes vanishingly infrequent.

There are a lot of reasons why the BDD process works. Most simply, just the act of specifying your requirements in code forces you to think about how you will express those requirements, and also gives you a chance to design the API with the tests as an actual use case. However, I think the value of testing goes beyond merely giving you a chance to think about your code.

When you write the tests before the code, the fact that the test needs easy access to the code strongly encourages a code architecture where data is easily accessible – the code has a lot

of surface area, so to speak, and not as many values hidden deeply. The test acts as kind of a universal client, forcing all the parts of the code to have values easily accessible.

Because the tests are actually driving the structure of the code, the code and tests build together symbiotically, which is why the code benefits of BDD are much harder to come by if the tests are written after the code is already there.

Perhaps the most critical part of the BDD process is the idea that your code is actually test-driven. Being test-driven means that the tests are your actual source of truth about the behavior of your application. When there is a discrepancy between the tests and the application, your first assumption is that the test is correct and the code is wrong. When the tests become difficult to write, you do not blame the tests, but instead assume that the code architecture is flawed.

Section 4.2

Jasmine Structure

Hopefully, that rant will inspire you to write more test-driven JavaScript. Jasmine is a great BDD tool for testing JavaScript. One of Jasmine's best features is that the structure of a Jasmine test file is relatively simple.

A Jasmine test file contains one or more top-level calls to the `describe` function, which denotes a test suite. (Typically, there's one top-level `describe` per file, but there's nothing stopping you from having more but your own conscience and the anger of your teammates).

The `describe` function takes two arguments. The first argument is a string describing the suite as a whole, and the second argument is a function which contains the contents of the test suite. When Jasmine is run, the functional arguments to all the `describe` methods are invoked.

The functional argument to `describe` contains a test suite. Inside that suite, there are four Jasmine methods you can call:

- You can create a nested suite with another call to `describe`.
- You can create a spec with `it`.
- You can specify initialization behavior with `beforeEach`.

- You can specify cleanup behavior with `afterEach`.

The `beforeEach` and `afterEach` methods each take a single functional argument. The `it` method takes a string description and a functional argument.

When the Jasmine specs are run, the function arguments to each `describe` method are executed, which serves to set up the specs inside that suite. Then, each spec created as an argument to an `it` method is run according to the following algorithm:

1. Any function passed to a `beforeEach` method is executed. The order of execution is outside-in: global `beforeEach` methods are first, then outer `describe` suites, then inner ones.

2. The function passed to the actual `it` method is executed. Three outcomes are possible: the spec can pass, the spec can cause a language error, or an assertion made in the spec can fail. Unlike many other test frameworks, Jasmine does not stop execution on a mere assertion failure.

3. Any function passed to an `afterEach` method is executed. The order of execution is inside-out: innermost `describe` suites first, then outer, then global methods.

It gets a little tricky if you want to share a variable by initializing it in a `beforeEach` method and have the variable available in your specs. You need to make the variable global, or alternately declare the variable in the body of the `describe` outside the `beforeEach`. A variable that is just declared in the `beforeEach` using the normal best practice of the `var` keyword is only in scope for the `beforeEach`, not in scope for the spec itself. Globals are usually reasonable in this case because the test is short lived, so there's not much chance of polluting a global namespace. Here's an example showing both the outside declaration and the global declaration.

```
describe("how to use variables", function() {
  var outsideDeclaration;

  beforeEach(function() {
    outsideDeclaration = "outside";
    globalDeclaration = "global";
    var localDeclaration = "local";
  })

  it("can see some variables", function() {
```

```
    expect(outsideDeclaration).toEqual("outside");
    expect(globalDeclaration).toEqual("global");
    expect(localDeclaration).toBeUndefined();
  })
})
```

Sample 4-2-1: This code shows how you can share variables in Jasmine tests

Jasmine Matchers

Inside each `it` spec, Jasmine uses *matchers* to allow you to make testable claims about the state of the code. In our original specs, we had four matcher lines:

Sharing variables in CoffeeScript

If you are using CoffeeScript, the global trick won't work, because CoffeeScript automatically scopes the variable, so you can't get to the global scope. A workaround you sometimes see is to use a CoffeeScript `@`-variable, which places the variable in the test object itself, and is the equivalent of assigning the variable to `this.something`.

```
expect($('.detail')).not.toHaveClass("hidden");
expect($('.detail_toggle')).toHaveText("Hide Details");
expect($('.detail')).toHaveClass("hidden");
expect($('.detail_toggle')).toHaveText("Show Details");
```

Sample 4-3-1: Jasmine matchers from last chapter's toggler tests

In a Jasmine matcher, the function `expect` starts a chain of method calls to evaluate an actual value in the application against a particular logical assertion. The argument to `expect` is the actual object in the system, for example, the jQuery objects returned by the selectors `.detail` and `.detail_toggle`. The return value of `expect` is a new proxy object that wraps the actual value. The proxy object responds to matcher methods, which are the facts you can assert about the object. We're using the matcher methods `toHaveClass` and `toHaveText`, both of which are defined by the `jquery-jasmine` extension library.

Jasmine has its own matcher methods. The most commonly used Jasmine matcher is `toEqual`. The definition of equality that Jasmine uses for this matcher is surprisingly complex, but basically boils down to using the JavaScript type casting equality operator `==` where both the expected and actual value are basic types, a comparison of key/value pairs if both values are basic objects, and the non-typecast `===` comparator otherwise. If you are interested in a stricter test that just uses `===` equality, then the matcher you want is `toBe`. Jasmine's predefined matchers cover the most common test needs. There are four matchers for

boolean and existence values: `toBeTruthy`, `toBeFalsy`, `ToBeNull`, `toBeDefined`. "Truthy" and "falsy" may sound fancy, but they just catch `true` and `false` values. Numeric comparisons can be made with `toBeGreaterThan` or `toBeLessThan`. Strings can be compared to regular expressions with `toMatch`, while array membership can be tested with `toContain`, where the actual value is the array. Finally, you can test for errors with the slightly more involved `expect(func).toThrow(e)`, where the function argument is evaluated, and the matcher passes if the exception `e` is thrown along the way.

The Jasmine-jQuery module adds another dozen or so matchers to the Jasmine defaults. Along with `toHaveAttr`, other matchers test for the existence of particular DOM items, namely `toHaveClass` and `toHaveId`. The HTML5 data object can be examined with `toHaveData`. You can check the internals of a tag with `toHaveHtml` and `toHaveText`. A related matcher is `toContain`, which takes a jQuery selector as an argument and matches if there is an item matching that selector inside the actual object.

A few matchers check for a specific state of a DOM element, `toBeVisible`, `toBeHidden`, `toBeChecked`, and `toBeSelected`. All in all, these matchers are useful shortcuts to specify expected DOM behavior in our Jasmine specs.

A quick aside about `not`. You may be surprised that the negative matcher is written `expect(x).not.toEqual(y)` rather than `expect(x).not().toEqual(y)` – in other words, that `not` isn't a method call in its own right. I certainly was – what is `not` if it isn't a method? Turns out that the `expect` method creates two objects: the positive one that is the return value of `expect`, and a negative object that is set as the `not` property of the positive object. If you don't sweat the details, this snippet from the actual Jasmine source might make things clearer.

```
jasmine.Spec.prototype.expect = function(actual) {
    var positive = new (this.getMatchersClass_())(
            this.env, actual, this);
    positive.not = new (this.getMatchersClass_())(
            this.env, actual, this, true);
    return positive;
  };
```

Sample 4-3-2: Jasmine definition of negative matchers

Writing your own matcher is easy. As an example, let's look at `toHaveAttr`, the one that is used by the Jasmine test at the beginning of this chapter. Here's a slightly simplified version of the actual definition of that matcher:

```
var jQueryMatchers = {
  toHaveAttr: function(attributeName, expectedValue) {
    var actualValue = this.actual.attr(attributeName);
    if (expectedValue === undefined) {
      return actualValue !== undefined;
    }
    return actualValue == expectedValue;
  }
};
```

Sample 4-3-3:

The matcher is very simple: it's just a function that returns a boolean value. Within the matcher function you can assume that the variable `this.actual` contains the actual value that starts off the matcher. In other words, if the invocation of the matcher is `expect($("#external_link")).toHaveAttr("target", "_blank");`, then `this.actual` would equal `$("#external_link")`.

The matcher definition has one quirk – if the expected value is not specified (`expectedValue === undefined`), then the matcher just checks if the attribute exists (`actualValue !== undefined`). If the expected value is specified, then the test is whether the value of the actual attribute is equal to the expected value.

To get Jasmine to recognize the existence of the matcher, you register the matcher using Jasmine's `addMatcher`. You can invoke `addMatcher` inside any `beforeEach` or `it` method. The argument to `addMatcher` is a JavaScript object where the keys are the names of the matchers and the values are the matcher functions themselves. It's not a coincidence that `jQueryMatchers` on line 1 is such an object. The matcher can be added like so (assuming that `jQueryMatchers` is in scope):

```
beforeEach(function() {
  this.addMatchers(jQueryMatchers)
});
```

Sample 4-3-4:

With the call to `addMatchers` in place, you can use any of your new matcher functions in your Jasmine tests just like the existing matchers. Creating your own matchers is useful if you have complex logic in your tests that is repeated multiple times. A matcher with a meaningful name can make your tests easier to read and less error-prone.

Section 4.4

Installing and Running Jasmine in Your Project

Writing the test isn't very useful unless you can execute it. Jasmine is, at base, a plain old JavaScript program, and can be run in a plain old HTML page that loads it properly. The Jasmine library comes with just such an HTML page. That said, there are a few shortcuts to setting up Jasmine if you are working in a Rails application. We'll cover the Rails case first, then talk about other Jasmine setups.

Jasmine and the Rails Asset Pipeline

The issue with setting up Jasmine in Rails 3.1 and up is giving Jasmine full access to the assets in the Rails asset pipeline. You need access to the asset pipeline for two reasons: to get access to JavaScript libraries that Rails is managing, and also to allow you to have CoffeeScript and SASS/SCSS files automatically compiled for you. The best of solution that I've seen for Rails 3.1 is the `jasminerice` gem, available at https://github.com/bradphelan/jasminerice, which was quicker to adapt to the asset pipeline than the official gem.

In my experience, the easiest way to set up jasminerice is to first include the `jasminerice` gem into your gemfile with the statement `gem "jasminerice"`.

After the `bundle install`, install Jasmine normally with the command `rails generate jasminerice:install`. This will set up the Jasmine directory structure. The jasminerice gem sets itself up as a Rails engine. Then you need to add two files to the directory. Add a file `spec/javascripts/spec.js` as follows.

Filename: spec/javascripts/spec.js (Branch: master)

```
//= require application
//= require_tree ./
```

Sample 4-4-1: Your Rails 3.1 and up Jasmine manifest

That'll give us access to the entire application javascript and also classify anything in the `spec` directory as an asset, meaning that CoffeeScript Jasmine specs will be converted. To get

stylesheet info, add a file `spec/javascripts/spec.css`, which will similarly make application CSS available from the Jasmine specs.

Filename: spec/javascripts/spec.css (Branch: master)

```
/*
 *= require application
 */
```

Sample 4-4-2: Your Rails 3.1 and up Jasmine CSS manifest

That should get you set up. One point to note is that Jasminerice uses its own version of the `jquery-jasmine` helper, and there will be a conflict if you have the standalone version of the helper also in your project.

Jasminerice provides a URL route in developer mode that will run our Jasmine tests (the route is not activated in production). Open the URL `http://YOUR_SERVER_NAME/jasmine` and the Jasmine spec runner will be invoked. Since we've already written the code that makes this test pass, you should see something green like this:

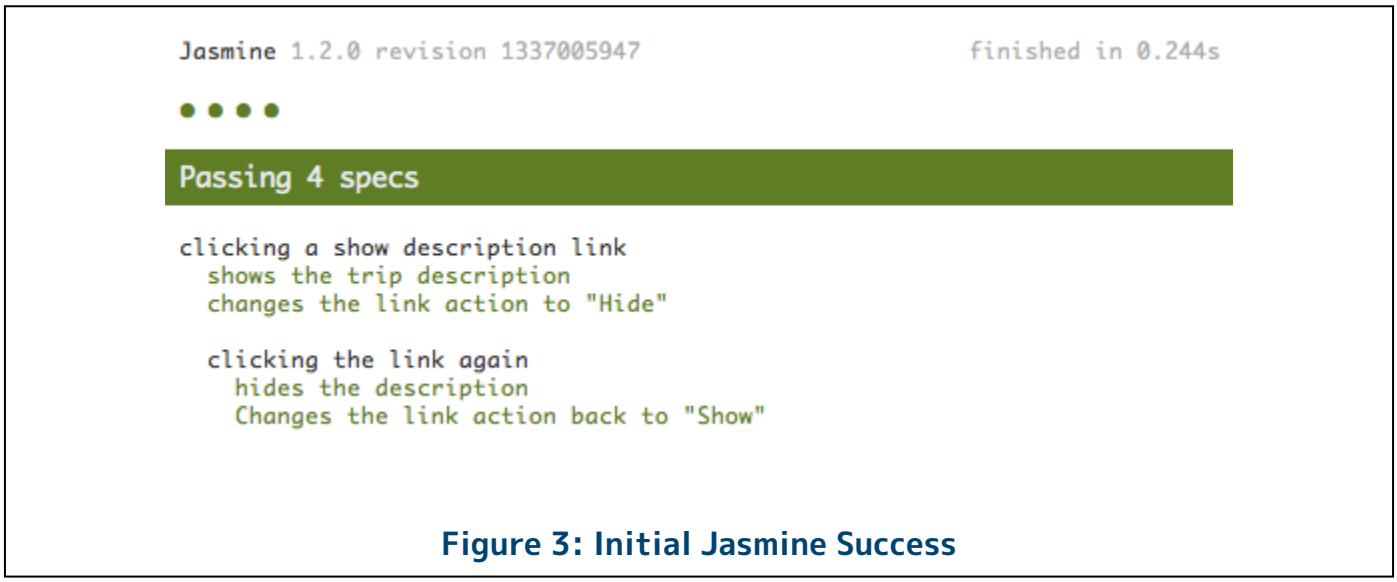

Figure 3: Initial Jasmine Success

If you are on a Mac OS X system or a Linux environment, you can get very fast command line Jasmine tests using the jasmine-headless-webkit gem. I've had some mixed success running jasmine-headless-webkit against asset pipeline versions of Rails, so be sure and check for current documentation. A gem called guard-jasmine-headless-webkit can be configured to run your Jasmine tests automatically whenever a file is changed.

Another way to run Jasmine headless is by using PhantomJS (http://phantomjs.org), which is also used by the `guard-jasmine` gem.

Jasmine and Older Rails Projects

For older Rails projects, Jasmine is best loaded via the official `jasmine` gem, which is installed using whatever gem loading tool your application uses. For Bundler, just including `gem 'jasmine'` in the test group of your `Gemfile` will work. After the gem is installed, you need to run `rails generate jasmine:install`. (For Rails 2 projects, that command is `script/generate jasmine`)

The generate command gives you a `spec/javascripts` directory with several subdirectories. The `jasmine_examples` subdirectory has your actual test files, the `helpers` directory contains JavaScript files that are automatically loaded as part of the test run, and the `support` directory has the actual Jasmine runner. Normally the only file you'd touch here would be the `jasmine.yml` file, which allows you to specify the locations of test files, helpers, and other items. For example, adding the following lines gives you some ability to use assets as stored in Rails 3.1:

```
src_files:
  - vendor/assets/javascripts/jquery.js
  - app/assets/javascripts/**.js
```

Sample 4-4-3: Jasmine gem settings for reading from asset pipeline files

In an older Rails project, you can use the original Jasmine gem, which supports a rake task `rake jasmine`, which starts a server at `localhost:8888`, hitting that server from the browser opens your tests. In addition to the `rake jasmine` task, the Jasmine gem provides a `rake jasmine:ci` task, which places output on the command line. As the `ci` in the name implies, this task is designed to be runnable by a Continuous Integration system such as Jenkins (http://jenkins-ci.org/). Behind the scenes, this task uses Selenium to open a Firefox browser and run the tests within it, so both Selenium and Firefox need to be installed on the machine for this to work.

We will also be using the Jasmine/jQuery helper (https://github.com/velesin/jasmine-jquery). Place the file in the `spec/javascripts/helper` directory.

Jasmine in a generic project

If your web framework does not have an extension that allows you to interact directly with Jasmine, the simplest thing to do is clone the Jasmine git repo at `https://github.com/pivotal/jasmine` and edit the `SpecRunner.html` file, which is in the repo in the `lib/jasmine-core/example` directory.

The header of that HTML file includes the following lines, slightly edited here for space.

```
<link rel="shortcut icon" type="image/png"
  href="lib/jasmine-1.1.0.rc1/jasmine_favicon.png">

<link rel="stylesheet" type="text/css"
  href="lib/jasmine-1.1.0/jasmine.css">
<script type="text/javascript"
  src="lib/jasmine-1.1.0/jasmine.js"></script>
<script type="text/javascript"
  src="lib/jasmine-1.1.0/jasmine-html.js"></script>

<!-- include spec files here... -->
<script type="text/javascript" src="spec/SpecHelper.js"></script>
<script type="text/javascript" src="spec/PlayerSpec.js"></script>

<!-- include source files here... -->
<script type="text/javascript" src="src/Player.js"></script>
<script type="text/javascript" src="src/Song.js"></script>
```

Sample 4-4-4: Jasmine SpecRunner.html

To run your Jasmine code, place a link to any dependent libraries, like, say, jQuery above the "include spec files" comment, place a link to your spec files below that comment – you can remove the links to the sample `Player` and `Song` files. Finally, links to your application code files should go under the "include source files here" comment.

When you load this file in a browser window as a static file, other JavaScript on the page will start Jasmine and run it on the specs defined in the linked files.

Section 4.5

What's It All About?

You frequently hear the claim that JavaScript is impossible, or at least very difficult, to test. While that certainly was true at one point, due to lack of tool support, the existence of Jasmine and the tools that have grown around makes JavaScript testing as easy as testing in any other language that has an xUnit or RSpec based testing framework.

Now, that's not the same thing as saying that JavaScript is easy to test any old way you write it. Historically, a lot of JavaScript has been written in such a way that the code has a lot of complex dependencies with the DOM, or with third-party libraries, or just between different parts of a single application that are better kept separate. By its very nature, that kind of tightly entangled code is hard to test. Unit tests work best when you can isolate individual units.

Which is where writing the tests first comes in – when you write the tests first, you make sure that every part of your code that has logic is accessible to an outside test, which encourages code that is made up of distinct units. We'll see examples of how this style might look in JavaScript starting in Chapter 5, when we talk about JavaScript objects, and later as we move into Backbone and Ember.

First, though, it's time to learn a little more about jQuery...

Chapter 5

More jQuery Than A Disney Princess Convention

Admittedly, Disney Princess Conventions aren't usually about jQuery.[8]

We're continuing our look at the libraries we used to make our simple toggler by talking about jQuery. The toggler touched on several features of jQuery including the ability to select specific DOM elements, the ability to trigger and respond to events, and the ability to change various features of a DOM element. We'll talk about each of these in more detail.

Section 5.1

Installing and using jQuery

Getting started with jQuery is easy. If you are using Rails 3.1 or higher, in fact, jQuery is included in the framework, though you do need to keep the `jquery-rails` gem in your `Gemfile`.

Older Rails apps, or, you know, web applications that don't use Rails at all, need to download jQuery from http://www.jquery.com. As I write this, the current version is 1.7.2. Like most JavaScript libraries, it comes in a development version, which is legible, and a production version, which has the code compressed to be as small a download as possible by removing whitespace, by giving variables shorter names, and the like. The development version is fine for our purposes. You'll want to place the jQuery file anywhere that you can find it and reference it with a `<script>` tag to load it into your web page.

8. Though I hear that Ariel is a big .NET fan. (Because Ariel and Arial and Microsoft and... never mind.) Also, no truth to the rumor that the Wicked Stepmother used to make Cinderella program in Perl.

In a production environment, you can choose to acquire jQuery via a Content Delivery Network (CDN). Using a CDN lets client browsers download the jQuery library faster and without adding load to your server. You can get a list of available CDN options at http://docs.jquery.com/Downloading_jQuery#CDN_Hosted_jQuery.

jQuery Function, What's Your Function?

With jQuery installed, you communicate with jQuery via the jQuery function, which is named `jQuery`, but is usually abbreviated as just `$`. As we've already seen, the `$` function has many behaviors depending on its arguments. Here is a complete list, the most common ones are bolded

Arguments	Behavior
A string with selector syntax	Searches the entire document for matching DOM elements.
A selector and a context	Searches the context for matching DOM elements.
A DOM element	Wraps the DOM element in a jQuery object.
A plain JavaScript object	Wraps the object in a jQuery object with minimal functionality.
An array of DOM elements	A jQuery object referencing the elements of the array.
A jQuery object	A clone of the jQuery object.
No arguments	An empty jQuery object.
A string with HTML tag syntax	A new DOM element with that HTML, wrapped in a jQuery object.
An HTML string and a DOM document	As with just an HTML string, plus the new element is added to the document.
An HTML string and a JavaScript object	As with the HTML string, plus the key value pairs of the object are applied as attributes.
A callback function	The callback function is executed when the `document.ready` event is fired.

That's a lot, obviously. The most important ones to keep in mind are the string with selector arguments, the HTML string, and the callback object.

We talked about the callback function already, and we'll talk about it again later in this chapter when we cover events. First, though, let's talk about jQuery selectors.

Section 5.2

Selecting DOM Elements

We use jQuery *selectors* to match elements on our page. A selector is a pattern that jQuery uses to identify DOM elements, using a syntax that is very similar to Cascading Style Sheet (CSS) syntax. The simplest jQuery selector is simply an HTML tag, like `span` or `img`. In our previous chapter, we saw jQuery selectors like `.detail` used to find elements with specific characteristics.

To use jQuery to match a selector, all we do is pass the selector as a string argument to the jQuery function. When the jQuery function is called with a selector argument, the return value is an instance of the imaginatively named *jQuery object*, which contains a list of all DOM elements that match the selector. So the following method call would return our headline elements.

```
$("h1")
```

Sample 5-2-1:

While this call would return every anchor tag on the page.

```
$("a")
```

Sample 5-2-2:

If you've done web development, there's a good chance you already know the basics of jQuery selector syntax. CSS syntax uses selectors to identify sets of DOM elements to which styles should be applied. This selector syntax has been adapted by many different web tools, including jQuery. Here are the most basic rules of jQuery selectors:

Selector	Descriptor
A bare element like `div`	All elements matching that tag.
An identifier preceded with a dot, as in `.post_title`	All elements with a matching DOM class.
An identifier preceded with a hash, as in `#item_id`	All elements with a matching DOM id (normally, there would only be one matching element).

Selector	Descriptor
An HTML tag composed with a CSS class or DOM id element, as in `div.post_title` or `h1#headline`	DOM elements that match both the HTML tag and the DOM class or ID.
Two CSS selectors composed one after the other, and separated by a comma, as in `div, td`	Any DOM element that matches either CSS selector.
Two CSS elements composed one after the other, and separated by a space, as in `div.post_title a`	All elements matching the second selector that are contained in an element matching the first selector.
A CSS element with a square bracket attachment, as in `a[attr]`	Any DOM element matching the selector and containing an HTML attribute matching the attribute in brackets, no matter what the attribute's value is.
A CSS element with a square bracket and an equality relationship, as in `[attribute="value"]`.	Matches all tags that contain the exact attribute with this exact value. When used without a tag in front, matches any tag.
A CSS element with a square bracket and a caret plus equality relationship, as in `[attribute^="start_value"]`	Matches all tags that contain the attribute where the value of the attribute starts with the requested string.
A CSS element with a square bracket and a dollar sign plus equality relationship, as in `[attribute$="end"]`	Matches all tags that contain the attribute where the value of the attribute ends with the requested string.
A CSS element with a square bracket and a star plus equality relationship, as in `[attribute*="substring"]`	Matches all tags with the given attribute where the value of that attribute contains the requested substring.

jQuery also supports a lot of pseudo-classes, some of which are part of CSS and some of which are jQuery extensions. Any jQuery selector that contains only elements defined by CSS will be optimized to use the core DOM method `querySelectorAll` in those browsers that support it. Which is all "modern" browsers, where the typical meaning of "modern" is "browsers that support this feature I care about." In this case, it's browsers that are equal to or newer than Internet Explorer 8, Firefox 3.1, Safari 3.1, and all versions of Chrome. Internet Explorer 8 support, naturally, is flaky, but that's jQuery's problem, not yours.[9]

The reason why you care which implementation is used is that the core DOM method will be faster where it exists. So if you use a non-CSS selector, you may get better performance by performing the DOM lookup without that selector, and then using native jQuery methods to filter the jQuery object after it has been obtained.

Section 5.3

Responding to Events

In order to make your web application responsive to client events like clicking on targets or mouse hovers, you need to associate blocks of code with the events. This process is sometimes called *event binding*, and the associated code to be executed when the event happens is a *callback*. In jQuery, callbacks can be associated with the results of any selector, making it easy to make multiple elements on the page share behavior.

When we first looked at events, we used the `click` method to define an event callback. In jQuery, the `click` handler is a special case of the general jQuery method for event callbacks, which is `on`[10]. We could write the click handler from our toggler using `on` directly.

```
$toggle_links.on('click', function(event) {
  // the handler goes here
};
```

Sample 5-3-1:

In the most basic form of `on`, the first argument is the name of the event, and the second argument is the functional callback. Inside the functional callback, the JavaScript variable `this` will be set to refer to the event target.

You may recall that back in Chapter 2 when we were initially writing our Jasmine specs, we needed to explicitly initialize the event handler in our tests after the fixture was loaded. The `click` method and the two-argument form of `on` bind only to DOM elements that exist when the event method itself is first loaded. Elements that are created later are not bound, even if they exist when the event itself occurs. In our example, that means that only `detail-toggle` elements that are on the page initially will be bound to the click handler. If an infinite-

9. To be clear, IE 8 will be your problem for other things, but not when dealing with jQuery selectors.

10. At least in jQuery 1.7. Earlier versions of jQuery use methods called `live` and `delegate` to cover the same functionality.

scrolling page loader or some other client-side mechanism added new trips to the page after initial load, the newcomers would not be bound. Clicking on the newcomer's "Show Detail" links would leave the details sadly unshown. In jQuery, these events which are bound statically to the elements that are in existence when the handler is loaded, are called `direct` events.

There is a way to bind event handlers to elements that do not yet exist, using the `on` method. A method so bound is called a *delegated* event handler. A delegated event handler defers calculating whether a DOM element matches the event until run time when the event happens, which means that even elements that are added after the handler is defined can still participate in the event binding.

To define a delegated event handler, you need two jQuery selectors, a parent selector and a subordinate filter. The parent selector matches a set of elements when the handler is loaded, just like a direct event would. However, when the event actually takes place, all the children of all the elements in that set are filtered using the second selector. Children that match that subordinate selector will respond to the event. Since this filtering happens at event time, even subordinate elements that didn't exist at load time will respond to the event.

If we wanted to make the toggle click event delegated, we'd need to choose a parent element that does exist at load time, which means that the element would have to be outside the entire trip grid. Luckily, if you look at the actual Rails view, we have a `.trips_container` element surrounding the whole grid for just such an eventuality. (Though if we were not so lucky we could use `document` as the parent element, which would encompass the entire page.) We'd then use our `.detail_toggle` selector as the subordinate pattern, with the final call looking like this:

```
$(".trips_container").on('click', ".detail_toggle", function(event) {
  // the handler goes here
};
```

Sample 5-3-2:

By using delegation in this way, we make our page robust in the face of future additions of trips to the page. It's good practice to make the parent element as close to the actual event target as possible, to limit the amount of steps the event has to spend inside the DOM tree before it matches the correct element and also to reduce the possibility that the event might bind to elements that you don't intend to be targets.

Section 5.4

Modifying DOM elements

In our toggler, we call the `toggleClass` and `text` methods of our jQuery object, which are only two of many methods that jQuery provides to change the contents of DOM elements. These are methods of the jQuery object, and typically they act on all the DOM elements in the object, though some methods only act on the first element of the jQuery object.

One of the most common changes to make to a DOM element is to change its CSS class. jQuery treats the list of an element's CSS classes as a set-like object. If you want to change the CSS class of an element in a structured way, you can use the methods `addClass` and `removeClass`. The argument to each method is the same: a space-delimited list of class names, or much more rarely, a function that returns such a list. The `class` attributes of all the elements in the jQuery object are changed appropriately.

We can actually make this a little simpler. Both `removeClass` and `addClass` are backed by the more generic `toggleClass`, which takes a space-delimited list of class names and an optional boolean. If the boolean is not specified, then each class in the list has its presence or absence from the element flipped – classes already in the element are removed, classes not in the element are added. If the boolean is specified, then the state of each class for each element is set to match the boolean value, with `true` meaning the element has the class.

Modifying HTML Attributes

The most generic method for attribute modification is `attr`, which has a getter variant and a setter variant. The getter version takes one argument: a string, which is the name of an attribute. The return value is the value of that attribute for the first (and only the first) element of the jQuery object. So, given this HTML:

```
<a href="http://www.externallink.com" id="external_link">
<a href="/trips/1" id="internal_link">
<a href="" id="full_internal_link">
```

Sample 5-4-1:

The jQuery expression

```
$("a").attr("href");
```

Sample 5-4-2:

would return the value `http://www.externallink.com`, since the external link would be the href value of first anchor element in the fixture.

The setter comes in three forms. The first takes the same string as the first argument, and a second string as the second argument. This second value becomes the value of the attribute for all members of the set, not just the first one. The following line changes the `href` value for both anchor tags in the fixture.

```
$("a").attr("href", "http://www.oops.com");
```

Sample 5-4-3:

This pattern of changing a getter to a setter by adding one more argument is very common in jQuery, as is the pattern of having a getter return a single value while the setter applies to the entire set.

The `attr` method has two more complicated forms. In one form, `attr` takes one argument, which is itself a JavaScript object made up of key/value pairs. Each key represents an attribute to change, each value, the new value of that attribute.

```
$(this).attr({target: "_blank", title: "Opens in a new window" });
```

Sample 5-4-4:

If you want more dynamic behavior, you can make the second argument to `attr` a function, with arguments being an index and the current value of the attribute. This admittedly contrived snippet sets the target of the first five anchors to blank, while leaving the rest alone.

```
$("a").attr("target", function(i, val) {
  if (i < 5) {
    return "_blank";
  } else {
    return val;
  }
});
```

Sample 5-4-5:

jQuery has methods that perform more specific data access, for example accessing the value of a form element. You can do this with the `val` method, which returns the form value of the first element in the jQuery object. One nice shortcut is that the value of a `select` element is calculated as the value of the selected option or options. The one-argument setter version of `val` changes the form value of all elements in the jQuery object. Like the setter for `attr`, the argument can also itself be a function that returns the new value.

In HTML 5, arbitrary data can be added to any HTML element by giving the element an attribute with a name that starts with `data-`. jQuery allows you to access those attributes using the `data` method with a string argument corresponding to the stem of the attribute name. So, an attribute of `data-initialValue` is retrievable using `data('initialValue')`. Although the data attribute can be a useful way to pass data between HTML and jQuery, using the data attribute adds a dependency between your HTML view and your application code that may make your code harder to change over time.

If you'd rather get at the CSS-style attributes for a DOM element, you can use the `css` function. The one-argument version is the getter, taking a CSS property as the argument. The `css` method allows you to use either the CSS stylesheet version of the property name with all lower case and dash-separated, or the DOM version of the property name, which is camelCased. It also papers over some differences in how different browsers name CSS styles. Like other getters, only the first element in the jQuery object is used for the result value.

The getter retrieves a value only if it was explicitly specified in the element's `style` attribute or if it was set as a style by jQuery. Styles implicitly declared via CSS class are not found. The setter version of `css` is consistent with other methods we've seen, taking either an attribute name and value, an attribute name and function, or a JavaScript object with key/value pairs representing attribute names and values.

Modifying HTML Text

We used the jQuery `text` method to change the link text of our toggle link. Specifically, we used the single-argument version of `text` to set the content of the DOM element receiving the method. The `text` method is similar to many of the attribute methods we've already discussed in that the setter and getter have the same name but a different number of arguments. In this case, the single argument version of `text` is the setter. When you use the `text` method, you are assumed to be dealing with plain, non-HTML text. Any HTML elements in the text you are passing as the new value are escaped.

The no-argument getter of the `text` method is a little different from other getters we've seen. Most unusually, compared to other getters the `text` is not limited to just the first element of the selection set, but will combine the text of all elements in the selector set. Also, any HTML syntax in the text will be removed.

If we wanted the text of the link to contain live HTML, we would use the `html` method. The no-argument version of `html` is the getter, returning the contents of the first element in the jQuery object's element set – what would often be called the `innerHtml` of the element. This getter, unlike `text`, works only on the first element of the set.

The one-argument version of `html` is the setter. If the argument is a string, then the HTML contents of each element in the jQuery object is set to that string. If the string contains HTML, then the HTML is inserted right into the DOM. The `html` method, like other setters, can also take a functional argument that returns a string. You can also append or prepend HTML to the existing text using the `append` or `prepend` methods.

Section 5.5

jQuery Retrospective

jQuery is a fantastic library for dealing with DOM elements, finding them, changing them responding to their events. It's concise and powerful. In the next chapter we'll take a look at integrating jQuery into more complicated JavaScript structures so that you can build more modular JavaScript code.

Chapter 6

JavaScript Is Like A Teacher on Sunday... No Class

You are about to submit your code change to the mysterious time-traveling Dr. What. However, mere seconds before you are about to commit your change, an email arrives:

> Dear Programmer Friend.
>
> I was looking over your code change. Nicely done. Functionally, it's exactly what I wanted. I'm thrilled that you've made this change so easily.
>
> However...
>
> I think you'll agree that the structure of your code could be improved. What if I want this behavior for multiple different areas of my page? How can I avoid namespace pollution? Can I reuse this code easily? Please take a look and report back with ideas for improvement.
>
> Yours Until The Mayan Calendar Ends,
>
> Dr. What

This email raises some troubling questions. First off, how did Dr. What see your code before you committed it? Only slightly less importantly, where does Dr. What get the gall to suggest code architectures to us, the programmers? Oh, and how can you improve the structure of our code in JavaScript?

On further reflection, you realize that the Doctor is right. This code could be made more reusable, more maintainable, and easier to work with. It's time to explore JavaScript objects and functions.

This chapter is going to be a little bit different than most chapters in the book. Rather than focusing on adding new functionality to the client side of the Time Travel Adventures application, we'll be working on patterns and practices for how to express that functionality in our code. Our goal is to allow the code to be modular, maintainable, and reusable. Doing that requires some exploration of JavaScript's object and functional models.

JavaScript has many similarities with other languages that are derived from the syntax of C. It has `if` statements, `for` loops, variables, arrays, strings, and the like. However, JavaScript also contains a few ideas that despite resembling Java, Ruby, or Python, actually behave quite differently. In particular, JavaScript's object model is fundamentally different from most object-oriented languages, and attempting to create complicated applications without understanding the difference is doomed to be frustrating.

Every complex JavaScript library critically depends on JavaScript's incredible flexibility when it comes to defining, assigning, and invoking functions. Understanding JavaScript functions correctly is very important both in understanding the libraries we will be using, and in creating your own JavaScript applications.

Over the course of this chapter, we'll show several ways to structure our simple toggle code so as to make it more modular and object-like. Although some of these patterns may seem overly complex for such a simple example, a simple example is best for showing the essential parts of each pattern. At the end of this chapter, you'll be able to make JavaScript code behave like objects and classes in other languages, and you'll see why anonymous functions are such a big part of modern JavaScript. We'll continue to use these patterns throughout the rest of the book to support more complicated functionality.

Section 6.1

Objects With No Class

Let's start by understanding JavaScript objects. In this section, we'll talk about how objects are specified, how they are used, and what object methods are. We'll also perform the first of our transformations on the toggler code.

JavaScript has what is called a *prototype/instance* object model. While the JavaScript model has superficial similarities to class/instance Object Oriented languages such as Java, Python, and Ruby, JavaScript's internal management of object relationships is fundamentally different. To make things more complicated, a number of structures have evolved via library or

convention to augment JavaScript objects to make them look or behave in a more familiar way. Depending on your perspective, these structures either destroy the purity of JavaScript's model to protect programmers who don't want to learn something new, or add functionality to drag JavaScript kicking and screaming into the world of languages you can actually do something with. And hey, both those ideas have some truth. Let's go through what JavaScript provides and find a way to get the kind of encapsulation that you'll need for larger projects.

(Almost) Everything is an Object

Everything in JavaScript is an object, except for the things that aren't. Three of the five non-object types in JavaScript – boolean literals, numbers, and strings – can be trivially wrapped in object types provided by JavaScript. The other two are `null` and `undefined`. The difference between the two is subtle. A `null` value indicates that the variable in question has been declared in the current scope, but does not currently have a value, while `undefined` means that the value has not even been declared in the current scope.

Objects are most easily created in JavaScript using the object literal syntax:

```
var obj = {};
var objWithProperties = {color: "red", name: "Zach"};
```

Sample 6-1-1:

If you are familiar with Ruby or Python, then you are likely thinking to yourself that JavaScript's object literal syntax is suspiciously similar to Hash or Dictionary literals in those languages. You'd basically be right. Objects in JavaScript are mostly just a namespace with key/value pairs. (In many respects, quite similar to Python objects).

Accessing the properties of an object is done with one of two different syntaxes:

```
var objWithProperties = {color: "red", name: "Zach"};
var objColor = objWithProperties.color;
var objColor = objWithProperties["color"];
var x = "color";
var objColor = objWithProperties[x];
```

Sample 6-1-2:

The dot form (`objWithProperties.color`) requires the object property to be a valid JavaScript identifier, meaning no spaces or special characters, and is evaluated at load time. The square

bracket form (`objWithProperties["color"]`) looks for a property whose name matches the string. In that form, the property is determined at run time, can include spaces or weird characters, and can be a dynamic variable, as in the last two lines.

A couple of things about that JavaScript snippet are worth mentioning in passing. Semicolons are technically optional when used at the end of a line. However, in some cases the JavaScript parser will get confused and behave unexpectedly without the explicit statement separator, so including the trailing semicolon is a good habit to get into. Secondly, the `var` keyword, which you really want to be in the habit of using, keeps the variable being defined in the local scope, as opposed to being in the global scope. Keeping all your variables in global scope is a good way to have subtle and hard to find bugs.

The value of an object property can be a function, which is placed on the right side of a key/value pair just like any other value.

```
var obj = {
  color: "red",
  announce: function() { console.log(this.color); }
}
```

Sample 6-1-3:

The function can then be invoked using regular method syntax, in this case `obj.announce()`. When a function is called via an object, then inside that function body, the keyword `this` can be used to access other properties of that object, as in `this.color` in the above snippet. We'll talk more about `this` in functions and scope, but one important issue to note here is the value of the `this` keyword can actually be affected by the context of the running program. For example, within jQuery event handlers, jQuery modifies `this` to refer to the event target, even if the event handler is defined as part of another object.

We can take our toggler code from last chapter and convert it into a simple object. Doing so gives us a couple of related advantages relatively cheaply. First, we can make all the magic literal string values in our original code into object properties, separating data from logic and making it easier to change behavior later on. Similarly, we can refactor the event handler and break it into smaller, easier to manage methods. The resulting object might look like this:

Filename: app/assets/javascripts/toggler.js (Branch: section_5_1)

```
var toggler = {
  linkSelector: ".detail_toggle",
```

```
  detailSelector: ".detail",
  hiddenClass: "hidden",
  hideText: "Hide Details",
  showText: "Show Details",

  init: function() {
    var self = this;
    $(this.linkSelector).on("click", function(event) {
      self.toggleOnClick(event)
    });
  },

  toggleOnClick: function(event) {
    this.$link = $(event.target);
    this.$link.text(this.isDetailHidden() ? this.hideText : this.showText);
    this.detailElement().toggleClass(this.hiddenClass);
    event.preventDefault();
  },

  detailElement: function() {
    return this.$link.parent().find(this.detailSelector);
  },

  isDetailHidden: function() {
    return this.detailElement().hasClass(this.hiddenClass);
  }
};

$(function() {
  toggler.init();
});
```

Sample 6-1-4: Our toggler as a JavaScript literal object

Although the overall length of the code has increased, the object is now made up of smaller pieces, and is now arguably easier to understand. We've moved the entire thing out of the jQuery function, and now the only thing we do inside the jQuery `document.ready` event is call the `init` method of the toggler object. We make the same change to the `beforeEach` method of the Jasmine spec:

Filename: spec/javascripts/togglerSpec.js (Branch: section_5_1)

```
beforeEach(function() {
  loadFixtures("one_index_trip.html");
  toggler.init();
  $(".detail_toggle").click();
});
```

Sample 6-1-5: Change to beforeEach method in Jasmine Spec

Let's look at what we've actually done to modularize the code. First, we've broken off the literal text pieces as properties at the start of the class. Now those properties act as defaults, if we want to use the object with different values, we can set them when used:

```
toggler.detailSelector = ".internal_detail"
```

Sample 6-1-6:

We still have a click handler function and the actual handler itself. However, we've refactored the logic quite a bit. We've pushed several parts of the click handler logic into their own methods, including the logic to find the detail DOM element, detect if the element is hidden, and then adjust the link text and visibility status.

In the actual click handler, we add one property to the object, `this.$link`, which is both the DOM element of the link itself and the event target of the click handler. Because we can add the link as an object property, we don't need to continually pass the link element to all the other methods, simplifying the code. With that element defined, we can then call methods to adjust the display visibility and link text, with same result as in the original code.

Outside the event handler, where `this` still refers to the object, we give `this` an alias, namely `self`. Because of the way JavaScript handles scoping, the `self` name is visible inside the event handler, so we can call methods on `self` inside the event handler.

A Few Words About this

In JavaScript, the value of `this` is determined by the nature of the function call. In particular, jQuery manipulates `this` inside event handlers such that `this` is the target of the event. However, method calls made from the event handler are processed normally. In the previous snippet inside the click handler function, `this` is the event target, but when the click handler makes the method call `self.toggleOnClick(event)`, then inside the `toggleOnClick` method,

`this` is back to being the receiver of the method, in this case the `self` variable referring to the toggle object itself. Weird, right?

The value of `this` is a dynamic construct in JavaScript which depends on how the function was called. Under normal circumstances, `this` has one of three values:

- If the function is called as a message to an object, as in `foo.bar()`, then `this` is the object.
- If the function is called without a message to an object, then `this` is the global object, which in a browser context is `window`.
- If the function is called via `new`, then `this` is the newly created instance.

Complicating matters are the JavaScript core functions `call` and `apply`, both of which are methods of functions, as in `bar.call(foo, arg1, arg2)`. When `call` is used, then inside the function, `this` refers to the argument to call, sometimes called the *context*. (The `apply` method works the same way, except it takes a single array for the arguments, as in `bar.apply(foo, [arg1, arg2])`.

Under normal circumstances, you can't do anything to prevent your function from being invoked in a weird context using `call` or `apply`. Many libraries, however, have extensions that allow you to permanently bind a function to a particular context object – we'll see jQuery's version, `proxy` in Book 2, and Underscore's version, `bind` in Book 3. In CoffeeScript, that functionality is always available with the fat arrow `=>` operator. All of these work by wrapping the original function inside a outer function that reassigns `this` to the original context no matter how the function is invoked.

The Language Feature That Wasn't There

Those minor quirks aside, the truly weird thing about object creation in JavaScript is what isn't in the code snippet. Specifically, the object was not assigned any kind of type or class. You might assume that's because there's some kind of default class, but that's not actually the case. In fact, even though JavaScript has objects, it does not have classes.

I'll repeat that, because it's important: JavaScript doesn't have classes.

How can that be, you might ask if you are familiar with Ruby, or Python, or Java. (Or Smalltalk, C++, C#, and so on and so on.) JavaScript has a different flavor of Object-orientation than

those other languages, providing the same basic set of OO concepts, but with a different underlying architecture. Object-orientation, whatever the language, is supposed to provide three features:

- *Encapsulation* means that data is grouped together with the code that acts on that data, and that some effort can be made to protect knowledge of that data from other parts of the system (even if that protection is basically a matter of convention).
- *Polymorphism* suggests that the behavior of a method is dependent on the kind of object responding to the method. In a strict language, like Java, this largely means that a subclass can define different behavior than the parent class, but in a more flexible language like Ruby or JavaScript, polymorphism implies that the behavior of a method is determined at run time.
- *Inheritance* is the ability of one object to use data or methods defined in another object as a default if the object does not itself define that method. Inheritance was originally conceived as the primary mechanism of reuse in an Object-oriented system, but these days, most OO developers are as likely to use modules to compose objects as they are to use strict inheritance.

In JavaScript, the class is not the basic unit that objects are grouped in. Instead, JavaScript relates objects and behavior using *prototypes*.

Section 6.2

A Prototype Instance Language

Protoytypes are important to us as JavaScript programmers, because we'd like to be able to use JavaScript to create multiple objects that share common behavior. As our toggler example currently stands, there's only one toggler object, and no obvious mechanism to create another one without duplicating code. If, for example, we want toggle-style behavior for multiple different elements on the same page, we haven't seen a clear way to do that yet.

In other Object-Oriented languages, we'd define a class, and place all the toggle behavior in the class, which would allow us to create multiple instances of togglers, each with its own set of properties. JavaScript doesn't quite work that way. Most of the other OO languages in existence are called *class-instance* languages. As you might expect, this is because these languages have things called "classes" and "instances."

JavaScript is a *prototype-instance* language. In JavaScript, any two objects can have the same relationship as a class and an instance might have in other languages. In other words, any object, regardless of its type, can act as the *prototype* of any other object. A prototype is sort of like a class in that if a property is requested of the child object, and the child object does not define the property, then the prototype object is the next place JavaScript looks for that property. However, prototype relationships are more flexible because any object can act as a prototype and the prototype can change over the life of the object. This can mean either that changes in the prototype are reflected in the child object or that the actual relationship can change because object's prototype has been changed to a completely different object.

Supporters of the prototype mechanism point out that class-based inheritance is effectively a subset of prototype inheritance, and that prototype inheritance is more flexible and, you know, baked into the language – most modern implementations of JavaScript optimize prototype lookup. Supporters of class-based inheritance say that prototypes are too flexible, and too hard to understand, and that classes are a useful abstraction when describing a system. In my usual middle-of-the-road kind of way, I personally don't feel like prototypes are too complicated in general, but I am willing to grant that classes are a useful abstraction.

Section 6.3

Using Prototypes in JavaScript

"Okay," you say, "I'm convinced. I'd like some of that sweet prototype action in my application. How do I get it?" Well, that's where things get a little complicated. One of the reasons why JavaScript's prototype-instance model takes a lot of flak is that the implementation of it in the language is far more confusing than it needs to be. Here's a partial application of the most widely available JavaScript mechanism for assigning a prototype to a new object as it might be applied to our toggler object. Look at this example, and repeat after me, "JavaScript doesn't have classes."

Filename: app/assets/javascripts/toggler.js (Branch: section_5_3)

```
function Toggle() {
  this.linkSelector = ".detail_toggle";
  this.detailSelector = ".detail";
  this.hiddenClass = "hidden";
  this.hideText = "Hide Details";
  this.showText = "Show Details";
```

```
}

Toggle.prototype = {
  init: function() {
    var toggler = this;
    $(this.linkSelector).on("click", function(event) {
      toggler.toggleOnClick(event)
    });
  },

  toggleOnClick: function(event) {
    this.$link = $(event.target);
    this.$link.text(this.isDetailHidden() ? this.hideText : this.showText);
    this.detailElement().toggleClass(this.hiddenClass);
    event.preventDefault();
  },
  // and so on...
```

Sample 6-3-1: Our toggler as a JavaScript Prototype/Instance

That's part of the code to create a prototype, here's the code that creates a new object with the above object as its prototype.

```
$(function() {
  var toggle = new Toggle();
  toggle.init();
});
```

Sample 6-3-2: The code to create a toggler

There's a lot going on here, and we'll unpack it over the course of the next few pages, but the upshot is that the use of `new` inside the function of the second snippet causes the `Toggle` function defined in the first snippet to be used as a *constructor*, which in JavaScript means that the prototype attached to `Toggle` and defined in by `Toggle.prototype` and onward becomes the prototype of `toggle`, the object returned as a result of the `new` call. Which, you'll note, is different from the object returned by the `Toggle` function itself, since the `Toggle` function does not return any value.

There is one other quirk in the way this code works that I need to call attention to, which is the weird `self = this` assignment. Well talk more about `this` later, but the short version is that inside the jQuery event handler `this` refers not to the enclosing object but to the event target. However, we still need to reference elements of the enclosing object for the object nature of the toggler to be useful. Hence, we assign `this` to a local variable that will be visible inside the event handler.

If you are a little bit confused by what's going on in that snippet, well, it's a little bit confusing. It's not consistent with how normal JavaScript functions behave, and it seems to deliberately obfuscate the prototype relationship. In order to fully explain how this works, and how JavaScript can create more complex structures, we need to talk about JavaScript functions and scope.

Use Functions To Create Scope In JavaScript

Classes in CoffeeScript

CoffeeScript, despite the fact that it compiles directly to JavaScript, actually does contain classes, in a manner of speaking. In CoffeeScript, you can build something using the keyword `class` that behaves like a class would in other Object-Oriented languages, however that class is defined in terms of ordinary JavaScript prototype behavior – specifically, it's defined using the module pattern that we'll talk about at the end of this chapter.

Our next problem, then, is to build a toggler object that we can easily create multiple instances of. We tried using a bare JavaScript object, but that only allows for one instance. We saw JavaScript's `new` keyword in conjunction with a special property called `prototype`, but haven't really explained yet why that might work. Even with the `new` keyword, the default JavaScript functionality doesn't have all the features we might want. Besides the somewhat awkward syntax, objects created with `new` don't have support for potentially useful features like privacy or shared data.

The answer to many of our problems lies in understanding the unique way in which JavaScript handles functions. If you're coming back to JavaScript programming after some time away, one of the first things you might notice about JavaScript these days is the liberal use of anonymous functions in all kinds of places. Because of the way that JavaScript functions both create new scopes and retain knowledge of the scope in which they were created, creative use of functions can simulate features common to other Object-Oriented

languages, such as private data. In this section, we'll talk about JavaScript functions, and we'll start to show patterns for simulating class structures using functions.

JavaScript functions are fully first-class objects in ways that go beyond what you might expect even if you are familiar with dynamic languages. Functions can be assigned to variables, returned as the result of a function, passed as the argument to a function, and set as the value of an object property.

JavaScript supports what you might call the traditional way to define a function:

```
function foo(arg1, arg2) {
  return arg1 + arg2;
}
```

Sample 6-4-1:

After that snippet is loaded, the `foo` function exists in the global scope (assuming that the definition is at global scope). You don't see this form much in JavaScript anymore because the general goal of JavaScript these days is to avoid defining things in global scope. Instead, functions are generally declared anonymously and assigned to values like so:

```
var foo = function(arg1, arg2) {
  return arg1 + arg2;
}
```

Sample 6-4-2:

The reason to prefer the second form over the first is that the mechanism of assigning an anonymous function to a variable enables the function name to exist in something other than the global scope. Specifically, naming variables this way makes it easy to name a function in a local scope or have a function be assigned as a property of an object.

```
var obj = {
  name: "fred",
  log: function() { console.log(this.name); }
}
```

Sample 6-4-3:

The `log` function is then accessible as `obj.log()`. Technically, the two forms are not mutually exclusive, so it's legal to give a function a name and assign it to a variable with a different name: `var left = function right() {}`. Don't do that. It's confusing. You can give an

otherwise anonymous function a name to make the code more readable and improve the usefulness of stacktraces, as in `$(this).click(function clickHandler() {});`.

What makes functions special?

Functions have some special abilities relative to regular JavaScript objects. Most obviously, functions can be called. I assume you're basically familiar with this – a call is a function object and an argument list. The arguments are evaluated, passed to the function, and body of the function is executed. If the function has a `return` statement, then the value of that statement is the return value of the function. If not, or if the `return` statement has no value attached then the return value of the function is `undefined`.

```
var foo = function(arg1, arg2) {
  return arg1 + arg2;
}
x = foo(1, 2); // returns 3
```

Sample 6-4-4:

One quirk of the JavaScript function semantics is that JavaScript does not enforce that the length of the argument list in the function definition is the same as the length of the argument list in the call. If the call has fewer arguments, then any extra arguments in the definition are silently set to `undefined`.

JavaScript does not have default argument values baked into the language the way that Ruby and Python do, however the idea can be simulated by taking advantage of an undefined argument:

```
var multiplyBySomething = function(value, multiplier) {
  multiplier = multiplier || 3;
  return value * multiplier;
}
x = multiplyBySomething(10) // x is now 30
```

Sample 6-4-5:

In this example, we take advantage of the fact that the undefined object has a false value and use the `||` operator to give the argument a value if it has not already been defined.

On the other hand, if the call has more arguments than the definition, the extra arguments are not assigned, however you can still access those values using a special local value called `arguments`. The `arguments` object is an array of the values used in the call, and is always available even if the argument list is perfectly normal. (Technically, `arguments` is not a true array, but it's definitely array-ish.) At its most extreme, this gives you a function that declares no arguments explicitly, but takes an arbitrary number:

```
var avgHeight = function() {
  var sum = 0;
  for(var i = 0; i < arguments.length; i++) {
    sum = sum + arguments[i].height;
  }
  return sum / arguments.length
}
x = avgHeight({height: 10}, {height: 20});
```

Sample 6-4-6:

Please note that if you are using ECMAScript version 5 or the library `underscore.js`, the array loop can be vastly simplified. The last thing I want to do here is get into a huge discussion over JavaScript versions, so suffice to say that most of ECMAScript version 5 is supported by Internet Explorer 8 and up, Firefox 4 and up, Safari 5 and up, and Chrome 6 and up.[11]

If a function is called via an object property, as in `user.age`, then the function has access to other properties of the receiving object via the keyword `this`. In other words, the function call behaves like a method call in other object-oriented languages. There is an important difference, though. In JavaScript, the value of `this` is a result of the way in which the function is invoked, not an inherent property of the object the function is enclosed in. As we've already seen, and will see in future chapters, some kinds of activity, most notably jQuery event handlers, redefine `this`, making `this` refer to something other than the enclosing object.

```
var person = {
  firstName: "Zach";
  lastName: "Paleozogt";
  fullName: function() {
    return this.firstName + " " + this.lastName;
  }
```

11. See http://kangax.github.com/es5-compat-table for a current compatibility table.

```
};
x = person.fullName();      // "Zach Paleozogt"
x = person["fullName"]();   // Looks weird, still works
```

Sample 6-4-7:

JavaScript function arguments can also be used as *constructors* to create new instances based on a particular prototype. We closed the last section showing this example without explaining it fully:

Filename: app/assets/javascripts/toggler.js (Branch: section_5_3)

```
function Toggle() {
  this.linkSelector = ".detail_toggle";
  this.detailSelector = ".detail";
  this.hiddenClass = "hidden";
  this.hideText = "Hide Details";
  this.showText = "Show Details";
}

Toggle.prototype = {
  init: function() {
    var toggler = this;
    $(this.linkSelector).on("click", function(event) {
      toggler.toggleOnClick(event)
    });
  },

  toggleOnClick: function(event) {
    this.$link = $(event.target);
    this.$link.text(this.isDetailHidden() ? this.hideText : this.showText);
    this.detailElement().toggleClass(this.hiddenClass);
    event.preventDefault();
  },
  // and so on...
```

Sample 6-4-8: Our toggler as a JavaScript Prototype/Instance

Let's go back through this in a little more detail. This code has three parts. In the first part, the constructor function `Toggle` is declared. The `Toggle` function is an ordinary JavaScript

function – the capitalized name is by convention, both to differentiate functions meant to be used as constructors and to make them look more like class definitions in other languages. Inside the constructor function you can use the `this` keyword to access the newly created object. Typically, you would set properties of the new object in the constructor. Functions that are going to be used as constructors do not need to return a value, their return value is managed by the JavaScript `new` construction process.

Next up we define the prototype of the function. In JavaScript functions are objects and objects can have properties, so functions can have arbitrary properties assigned to them. So, it's perfectly reasonable to take the function `Toggle`, and assign a value to the property `Toggle.prototype` (Among other things, function properties can be used as a convenient place to store memoized return values.) You can create arbitrarily named properties. However, the `prototype` property, when it exists, is used by JavaScript constructors to define what will become the prototype of any object created by the constructor.

Finally, we call the constructor function using the `new` keyword, which binds all these pieces together. Specifically, when you call a function with `new`, JavaScript creates a new object, sets that object's prototype to the prototype property of the function being called, and executes the function body in the context of the new object, so that `this` references in the constructor allow you to set properties on the new object. The new object is then returned, but by the statement as a whole, not by the constructor function – remember that the `Toggle` function itself does not return a value.

Section 6.5

Creating Prototypes Directly and Indirectly

This process serves as a convoluted prototype-instance constructor method. If the program requests a property of the object that the object does not itself define, then the prototype is the next place that JavaScript looks, in much the same way that a classical object-oriented language looks to the class for methods that are not defined as part of an instance. In the example, the method call `toggler.onClick` uses the `onClick` property. That property is not defined as part of the `toggler` object, but it is defined in the prototype that was attached to the object, so that function will be found and called.

However, the `new` statement is unsatisfying and overly complex, and somewhat error-prone. You are defining a special method that is supposed to be called only as part of a `new` statement, and that will not work correctly if called normally. The convention of having constructors start with a capital letter is designed to prevent the improper use of constructors, with the side effect of making a constructor call look like instance creation in other programming languages, which is a dubious advantage at best. The main problem with the `new` statement, of course, is that it's counter-intuitive and baroque to have to go through the extra step of creating a function and assigning the prototype to that function to do what you really want to do, which is just assign a prototype to the new object directly.

The awkwardness of the default mechanism is one reason why there are approximately sixty-five kajillion patterns, frameworks, and hacks designed to rationalize object creation and inheritance in JavaScript. In ECMAScript 5, one of these patterns, the `Object.create` method, was actually built into the language. Here's our code, rewritten to use `Object.create`.

Filename: app/assets/javascripts/toggler.js (Branch: section_5_5)

```
var togglePrototype = {
  linkSelector: ".detail_toggle",
  detailSelector: ".detail",
  hiddenClass: "hidden",
  hideText: "Hide Details",
  showText: "Show Details",

  init: function() {
    var toggler = this;
    $(this.linkSelector).on("click", function(event) {
      toggler.toggleOnClick(event)
    });
  },

// Skipped content: All the rest is the same, until...

$(function() {
  var toggler = Object.create(togglePrototype);
  toggler.init();
});
```

Sample 6-5-1: Our toggler using Object.create

The `Object.create` method takes two arguments: the first is the prototype of the new object, and the second is optional. If defined, the second object is expected to contain property descriptors that are added to the new object.[12] In this snippet, the predefined `togglerPrototype` object is directly added as the prototype of `toggler`.

The `Object.create` method has the advantage of being much more direct in terms of what's actually happening to create the new object, and the disadvantages of not being available in older browsers, looking somewhat confusing if you aren't used to thinking in prototypes, and generally not being used much (at least, as far as I can tell).

If you want something like `Object.create` in your own non-ECMAScript 5 code, many frameworks provide something similar, like this function, which you could use in place of `Object.create`.

```
createObject = function(proto, properties) {
  function PrototypeContainer() {};
  PrototypeContainer.prototype = proto;
  newObj = new PrototypeContainer();
  for each in properties {
    if properties.hasOwnProperty(each) {
      newObj[each] = properties[each]
    }
  }
  return newObj;
}
```

Sample 6-5-2:

The `Object.create` and `new` with prototype mechanisms allow us to do something with our toggler object that we could not do before – create more than one instance in the same application, each with its own set of data. Even on something with just the small amount of logic as this toggle switch, that's valuable. On code that contains much more complex logic, having individual instance data will be vital. That said, we're not going to focus on these specific mechanisms for creating instance-like behavior. They are a little convoluted, and hard to manage. Instead, we'll take advantage of the way JavaScript uses focus and scope to show another way of creating class-like behavior in JavaScript.

12. We are decidedly not getting into what a JavaScript property descriptor is. Suffice to say that it is an ECMAScript 5 way to specify extended information about an object property. For example, you can define custom getter and setter functions.

Section 6.6

Functions Create Scopes and Class-Like Behavior

Our problem right now is that we'd like to be able to create unique instances of objects with common behavior. Or, to be more direct, we want the functionality that other Object-Oriented languages provide using classes. We've seen that JavaScript's native prototype and `Object.create` behavior sort of allows this sharing of behavior, but with an awkward syntax and semantics that don't match the common expectation of classes.

It is possible to create class-like behavior in JavaScript, using one more piece of the JavaScript puzzle to create reusable modules and inheritance. To build this behavior, we need to understand another special feature of function objects, which is that they can create a local scope.

JavaScript has two scopes: the global scope and a nested local scope bounded by the current function being executed and any functions the current code is defined within. If an assignment statement is prefaced with the `var` keyword, then the variable is defined in the local scope. Any assignment that is not prefaced with a `var` statement goes in global scope, making it way too easy to accidentally put names in the global scope. Keeping names in global scope makes them prone to be clobbered by unrelated code, which is why using `var` consistently to keep variables local is important.

One unusual quirk of the JavaScript scope rules is that a local variable will magically shadow a global variable even before the local variable is actually declared. It's hard to describe without an example, so here is one:

```
var x = "global";
console.log(x);

var scopinator = function() {
  console.log(x);
  var x = "local";
  console.log(x);
```

```
}
scopinator();
```

Sample 6-6-1:

Which produces the console output:

```
global
undefined
local
```

Sample 6-6-2:

The middle line of the outputs is `undefined` because, and only because, the variable `x` is declared on line 6, *after* the log statement on line 5. Since the variable is defined somewhere in the current scope, it shadows the global value, but since the variable has yet to actually be defined, it still has the `undefined` value. You can verify this by removing line 6, the console output will change to `global global global`.

This behavior is called *hoisting*, which is way more charitable and polite than what I would have called it if I was in charge. It's the reason why JavaScript style suggests that you place all `var` declarations at the top of a function, and some style guides suggest defining all variables in a single `var` statement separated by commas.

At this point it's standard to give an example of the edge cases of scope so as to show you how bad code behaves. I'm going to avoid that temptation though, because it makes my head hurt in a bad way to come up with them, and I'm also not sure that it's helpful. Frankly, in most cases, any code that comes within shouting distance of the scope rule edge cases is guilty until proven innocent. Always declare variables with `var`, and you'll basically be fine.

Critically, not only do JavaScript functions create local scope, they retain the scope from where they were defined no matter where they are called, a feature that is technically known as a *closure*. This means that a function defined inside another function always have access to variables defined in the outer function, even if the inner function is assigned to a variable, passed around like a counterfeit bill, and eventually called in a completely different context.

This is probably easier to visualize with an example.

```
var outer = function() {
  var a = "look at me";
  var b = "cheeseburger";
```

```
  var inner = function() {
    console.log(a);
    console.log(b);
  }
  return inner;
}

var thing = outer();
thing();
```

Sample 6-6-3:

The `outer` function declares two local strings and a function called `inner`, then returns `inner`. When we call the `outer` function on the next to last line, the `inner` function is returned. When we call that function on the last line, the variables `a` and `b` are both printed to the console, even though neither one of them is visible from the global scope. When `inner` is called, it retains visibility of the local variables `a` and `b` because they were visible when the function was declared.

Having those local variables may not seem like much, but in fact, it's the key to doing all kinds of fancy class-like information hiding in JavaScript. Note the key point that the local variables are accessible from the function but not the global scope. To borrow a somewhat loaded Object-Oriented term, those variables are *private*.

And we can use that privacy to rewrite our Toggler object to allow for multiple instances with different data.

Filename: app/assets/javascripts/toggler.js (Branch: section_5_6)

```
var Toggler = function() {
  var self = {
    linkSelector: ".detail_toggle",
    detailSelector: ".detail",
    hiddenClass: "hidden",
    hideText: "Hide Details",
    showText: "Show Details"
  };

  self.init = function() {
```

```
    $(self.linkSelector).on("click", function(event) {
      toggleOnClick(event)
    });
  };

  var toggleOnClick = function(event) {
    self.$link = $(event.target);
    self.$link.text(isDetailHidden() ? self.hideText : self.showText);
    detailElement().toggleClass(self.hiddenClass);
    event.preventDefault();
  };

  var detailElement = function() {
    return self.$link.parent().find(self.detailSelector);
  };

  var isDetailHidden = function() {
    return detailElement().hasClass(self.hiddenClass);
  };

  return self;
};

$(function() {
  var toggler = Toggler();
  toggler.init();
});
```

Sample 6-6-4: Our toggler as a JavaScript function class-like thing

Although this structure is similar to the original object form of the toggle feature, there are some significant differences. The main difference is that instead of the Toggler "constructor" being just an object, instead, it is a function. Specifically, a function that returns an object. More specifically, each time the function is called, a new object is created, with its own copy of data.

When we want a new toggler, we call the function, as in `var toggler = Toggler()`. We can call the `Toggler` function multiple times, each time getting a new object with its own separate

properties. In other words, we now have instances with separate data but shared behavior, because each instance is getting the same methods as part of its object. Interesting.

Here's how it works. First off, right inside the function, we define the basic features of the literal object we're planning on returning. We're calling the object `self`, which is arbitrary but expressive. In this object, we're defining the default values of all our properties.

We can also give the `self` object methods, as we do with the `init` method. We do this just by the regular process of assigning a function to a property of the object, and the method is accessible by a normal method call, such as `toggler.init()`.

The other functions are not defined as properties on the `self` object, but rather as local variables in the scope of the outer `Toggle` function. Since all these functions retain access to their scopes, the local variable functions are visible from the functions defined as properties of `self`, so `self.toggleOnClick` can call `detailElement`, and `detailElement` can conversely refer back to `self` and any of its properties. This effectively makes the functions defined as local variables private methods of the toggler object. Notice that since all the private methods are not actually defined as part of the `self` object, we don't need to access them via `this`, and therefore we don't need the little extra line dance step to redefine `this` outside of the actual `click` event handler.

This setup is really nice, it's not that complex, and it gives us many of the features of Object-Oriented classes, though there is a slight cost to recreating the functions for each instance. However, we have abandoned the JavaScript prototype and `new` structure, and while it's tempting to just say "good riddance," the prototype structure is still valuable for inheritance and may be expected by other JavaScript libraries. In the next section, we'll see the *module pattern*, which combines the functions and scope ideas we've just seen with the prototype and `new` structure we saw earlier in the chapter.

Section 6.7

The Module Pattern Makes Fake Classes

We've successively refined our ability to create Object-Oriented structures over the course of this chapter. First, we created a literal object, but without the ability to separate class-like structures from instances. Then we saw prototypes, but the native structure makes it hard to truly encapsulate data. So we learned how to use functions to create scope and create

behavior that was only visible when accessed via the object. Now we're going to combine the best features of all these approaches into the *module* pattern, yet another way to simulate class-like behavior in JavaScript. One of the selling points of the module pattern is that it's a way to simulate private data and functions in JavaScript while exposing a public API to that data.

As a general rule, I'm not sold on the need for private methods. Still, the module pattern is worth discussing because it's somewhat baffling if you've never seen it before, plus if you can walk your way through it, that's a strong sign that you really understand JavaScript objects and functions. As an added bonus, this pattern is essentially what CoffeeScript uses to simulate class behavior.

The module pattern has a number of variants, but all of them use function scope in a different way than we've already seen: by defining a function and immediately calling it anonymously, which is sometimes called an *immediate* function. (In particular, the immediate function designation is used in the book *JavaScript Patterns* by Stoyan Stefanov.)

```
(function() {
  console.log("Immediately, Immediately, Immediate-L-Y");
}());
```

Sample 6-7-1:

The code snippet defines an anonymous function and immediately calls it, leading to the text being logged to the console when the entire snippet is executed. The various nested parenthesis are the key here. The outer parentheses surrounding the entire expression are required to keep the parser from getting confused – however, if the entire expression is the right-hand side of an assignment statement, then the outer parentheses are not required. The empty set of parentheses on the last line is what clues the JavaScript parser that the function is actually being called – basically, it's a function call like any other function call, except the function definition itself is the target of the call, rather than a name that points to the function.

A natural reaction upon first seeing this construct is to wonder why you would ever want to use it. The main reason is that an immediate function allows you to create what are effectively block-level variables that are visible anywhere inside the immediate function, but hidden from the rest of the application.

At its simplest, this mechanism blocks you from accessing the global scope, even when you are writing code at the top level of your file. It's not uncommon to see entire JavaScript files wrapped in an anonymous function:

```
(function() {
  var x = 3;
}());
```

Sample 6-7-2:

Without the surrounding function, just using a bare `var x = 3`, `x` would be at the top level, and would therefore be global. Using an immediate function gives us our own little local scope that contains its own top level, and is not visible outside the function. Wrapping what would otherwise be top-level behavior in an immediate function is further protection against global namespace pollution.

We can also use immediate functions to give us a scope we can use to create objects. Here's an example:

```
var users = function() {
  var knownUsers = [];

  return {
    addUser: function(name) {
      knownUsers.push(name);
    },
    mostRecentUser: function() {
      return knownUsers[knownUsers.length - 1];
    }
  }
}();

users.addUser("Barney Rubble");
users.addUser("Wilma Flintstone");
console.log(users.mostRecentUser());
```

Sample 6-7-3:

This snippet defines a function, starting on at the beginning of the snippet, and ending with the immediate call, that set of empty parenthesis that seems to just be hanging out. Inside

that function, we create a local variable – using `var` – called `knownUsers`, and then we return a literalwhich itself defines two function properties. Then, we call the function that has just been defined (that's the apparently unattached parenthesis), returning the object literal and assigning it to the variable `users`.

The critical point here is that JavaScript functions defined inside other functions retain the scope of where they were defined no matter where they are called. In this case, it means that the `knownUsers` variable is always available to the functions defined in the literal object, because `knownUsers` is part of the same local scope. But since `knownUsers` is local, the variable is otherwise inaccessible. So we can use `users` just like a typical object, but the `users` object is able to have hidden data. As we've already seen, the variable `knownUsers` is private.

A downside of the immediate function presented above is that there's only one `users` object – it's just a namespace, without the replication behavior you'd expect from a parent class or prototype. But if we set up our immediate function to return not a literal object, but a constructor function with its own prototype, then we're combining JavaScript's prototype behavior with the idea of immediate function scope. This is called the *module pattern*. Here's our toggler, organized using the module pattern.

Filename: app/assets/javascripts/toggler.js (Branch: section_5_7)

```
var Toggler = (function() {
  var totalTogglers = 0;
  var Toggler = function() {
    this.linkSelector = ".detail_toggle";
    this.detailSelector = ".detail";
    this.hiddenClass = "hidden";
    this.hideText = "Hide Details";
    this.showText = "Show Details";
  };

  Toggler.prototype = {
    init: function() {
      var toggler = this;
      $(this.linkSelector).on("click", function(event) {
        toggler.toggleOnClick(event)
      });
    },
```

```
    toggleOnClick: function(event) {
      this.$link = $(event.target);
      this.$link.text(this.isDetailHidden() ? this.hideText : this.showText);
      this.detailElement().toggleClass(this.hiddenClass);
      event.preventDefault();
    },

    detailElement: function() {
      return this.$link.parent().find(this.detailSelector);
    },

    isDetailHidden: function() {
      return this.detailElement().hasClass(this.hiddenClass);
    }
  };
  return Toggler;
})();

$(function() {
  var toggler = new Toggler();
  toggler.init();
})();
```

Sample 6-7-4: Our toggler using the module pattern

The basic elements here are the same: a function expression that is immediately evaluated. Inside the function, a local variable, `totalTogglers`, that will be private to the module, and a value to return (we're not actually using the local variable, I'm just showing where one would go). In this case, the value to return is the constructor function, `var Toggler = function()`. Like the other constructors we've seen, this one defines properties for the object that will be created when the constructor is invoked. And like the other constructors we've seen, it needs to define a prototype, starting with `Toggler.prototype`, which becomes the prototype of the object created when the constructor is invoked.

When the whole function is immediately invoked, the constructor `Toggler` is returned and assigned to the variable `Toggler`, which in this example has the same name, either clarifyingly or confusingly, I'm not sure. (Honestly, I kept the names the same only because CoffeeScript does, and I tend to trust the structure of the JavaScript code that CoffeeScript generates.) When the constructor is invoked, the same object creation happens as before, except that the

`totalTogglers` variable is now in scope for both the constructor and the methods defined in the constructor's prototype. Again, that makes the variable effectively private, although shared between all instances. If we wanted instance- specific private behavior or methods, we could replace the literal declaration of the prototype on line 11 with another immediate function.

Okay, we've now created the world's most complicated toggler function. There is a method to the madness, though. With objects that have more complicated logic, the module pattern allows us to use the standard JavaScript prototype tool for creating objects with common behavior and still allow for the kind of data encapsulation that makes complex projects easier to reason about. We'll be using this pattern to manage our objects for the next few chapters, though when we get to Backbone.js, we'll see that Backbone uses its own mechanism for assigning behavior to objects.

Section 6.8

Retrospective

This chapter was about taking the basic JavaScript and jQuery behavior we explored in the last few chapters and molding the code into a shape that is more suitable to complex projects. We presented three different versions of the Toggler object.

The first version used JavaScript's literal object syntax to make the Toggler a singleton object. This allowed us to refactor the methods to make them smaller and more focused, but it limited us to only one Toggler object in the code at a time. As a partial fix, we showed how JavaScript used the `new` keyword and a special property of a function called a `prototype` to allow any object to have its behavior defined by another object. This allows for instance behavior.

However, we also saw that prototypes can be combined with the way JavaScript uses functions to create more useful structures. First, we built a version of the Toggler that made the object into a function that returned an object. Inside that function, we could create variables and functions that were hidden from outside the Toggler. This mechanism allowed us to create reproducible Togglers, but it's not consistent with JavaScript's prototype behavior.

We pulled all these concepts together into the module pattern, which combined function scope, an interesting approach to calling anonymous functions immediately, and prototypes

to mimic some of the encapsulation features associated with classes in other Object-Oriented languages.

All of these constructs will make future extensions to the Toggler easier because they all minimize the interaction between the Toggler and the outside world, and make it easier to split the Toggler into small, easy to manage pieces. Our original Toggler tests only test the behavior of the Toggle as a whole, but these object versions also suggest that building the object via more focused testing is feasible. Whether the full module pattern is necessary in your application depends on whether you need or want the standard `new` and `prototype` behavior.

With all that under our belts, it's time to make more changes to Dr. What's Time Travel Adventures site.

Those changes are in Book 2, available at http://www.noelrappin.com. Check it out!

Chapter 7

Acknowledgements

A number of people helped in the creation of this book, whether they knew it or not.

As you may know, this book began life with a non-me publisher. Technical reviewers of the manuscript at that point included Pete Campbell, Kevin Gisi, Kaan Karaca, Evan Light, Chris Powers, Wes Reisz, Martijn Reuvers, Barry Rowe, Justin Searls, Stephen Wolff. Kay Keppler and Susannah Pfalzer acted as editors. Thanks to all of them.

Avdi Grimm provided a lot of useful tools for self-publishing.

I've been lucky enough to work with people who were willing to share JavaScript experience with me, including, but not limited to: Dave Giunta, Sean Massa, Fred Polgardy, and Chris Powers.

Justin Searls and his Jasmine advocacy and toolkit have been a huge help.

One of the great things about self-publishing is that people take the time to help improve the book by pointing out mistakes and typos. Thanks to Pierre Nouaille-Degorce, Sean Massa, Vlad Ivanovic and Nicolas Dermine for particular efforts.

Emma Rosenberg-Rappin helped me design the noelrappin.com web site, pick cover fonts, and also did some copyediting.

Elliot Rosenberg-Rappin inspired the idea of a spaceship cover and laughed at some of my jokes.

This book, and many other wonderful things, would not exist without my wife Erin, who has been nothing but supportive through the ups and downs of this project. She's amazing.

Chapter 8

Colophon

Master Space and Time with JavaScript was written using the PubRx workflow, available at https://github.com/noelrappin/pub_rx. The initial text is written in MultiMarkdown, home page http://fletcherpenney.net/multimarkdown/. PubRx augments MultiMarkdown with extra descriptors allowing for page layout effects, such as sidebars, that Markdown does not manage on its own.

PDF conversion is managed by PrinceXML http://www.princexml.com, with a little bit of additional processing by PubRx. EPub and Mobi generation is managed by using the Calibre http://calibre-ebook.com/ command line interface, using a method described by Avdi Grimm's Orgpress https://github.com/avdi/orgpress.

For the PDF version, the header font is Museo and the body font is Museo Sans. Both are free fonts from the exljbris font foundry http://www.exljbris.com/. The code font is "M+ 1c" from M+ Fonts http://mplus-fonts.sourceforge.jp/. On the cover, the title font is Bangers by Vernon Adams http://code.google.com/webfonts/specimen/Bangers, and the signature font is Digital Delivery, from Comicraft http://www.comicbookfonts.com/, and designed by Richard Starkings & John 'JG' Roshell. Cover image credit is on the title page up front.

Section 8.1

Image Credits

Cover

The original image used as the basis of the cover is described at http://commons.wikimedia.org/wiki/File:Challenger1983.jpg as a scene of the Space Shuttle Challenger taken during STS–6 in 1983 with a 70mm camera onboard the shuttle pallet satellite. According to NASA policy, the photo is in the public domain.

Time Travel Adventure Images

Mayflower http://commons.wikimedia.org/wiki/File:Plymouth_Mayflower_II.jpg. Replica ship Mayflower II at the State Pier in Plymouth, Massachusetts, US. This work has been released into the public domain by its author, wikitravel:user:OldPine.

Shakespeare http://commons.wikimedia.org/wiki/File:Shakespeare.jpg. It may be by a painter called John Taylor who was an important member of the Painter-Stainers' Company. National Portrait Gallery, London, Public domain.

Mars http://commons.wikimedia.org/wiki/File:Mars_Hubble.jpg. NASA's Hubble Space Telescope took the picture of Mars on June 26, 2001, when Mars was approximately 68 million kilometers (43 million miles) from Earth. Public Domain.

Cubs http://commons.wikimedia.org/wiki/File:Patrick_Joseph_Moran_1908.jpg. Patrick Joseph Moran, Chicago baseball player, standing at home plate with bat in hand during baseball game. From the Library of Congress George Grantham Bain Collection - http://hdl.loc.gov/loc.pnp/cph.3c33635. Author unkown. Public Domain.

Lewis and Clark http://commons.wikimedia.org/wiki/File:Lewis_and_Clark.jpg. Lewis portrait by Charles Wilson Peale. Clarke portrait by Charles Wilson Peale. Public domain.

Xerox PARC http://commons.wikimedia.org/wiki/File:Xerox_Alto_mit_Rechner.JPG. Xerox Alto computer. Placed into public domain by user Joho345.

Washington Detail of Washington at the Delaware by Edward Hicks, http://commons.wikimedia.org/wiki/File:Washington_at_the_Delaware_c1849_Edward_Hicks.jpg. Public Domain.

Napoleon Portrait by Jacques-Louis David http://commons.wikimedia.org/wiki/File:Jacques-Louis_David_017.jpg Public Domain.

Everest Himalaya from the International Space Station. http://commons.wikimedia.org/wiki/File:Himalayas.jpg. Public Domain per NASA policy.

Beatles http://commons.wikimedia.org/wiki/File:The_Fabs.JPG. UPI photo, Public Domain in the United States.

Enigma http://commons.wikimedia.org/wiki/File:Enigma.jpg. Enigma machine on display at the NSA. Public Domain.

Marco Polo http://commons.wikimedia.org/wiki/File:Marco_Polo_-_costume_tartare.jpg. Public Domain.

Chapter 9

Changelog

July 23, 2012 Initial release to those who signed up for early information.

Release 002: July 30, 2012

- Removed line numbers from code samples. I'm generally not referencing them in the body of the text, and they were really screwing up code layouts in Kindle and ePub.
- Book URL added to preface.
- Slight change to description of Jasminerice.
- Setup rake task alluded to in text actually added to code.
- Added images to Time Travel Adventure site and credits to colophon
- Added screen shot of Time Travel Adventure site to chapter 1
- New section in Object chapter about dealing with `this`.

Release 003: August 18, 2012

- Fixed typos and errors reported by Pierre Nouaille-Degorce, Sean Massa, Avdi Grimm, Vlad Ivanovic, and Ashish Dixit

Release 004: October 2, 2012

- Fixed typos and errors reported by Seamus Roche, Deirdre Lonergan, Jakob Lehner, Enric Ribas, Jonathan Davis, Nicolas Dermine, Marc Ryan Riginding, Dutch Rapley, and Bharat Ruparel.

Release 005:

- Fixed a very bad paragraph about `this`, pointed out by Dan Steinicke.

- Found my own typo, the installer task for Jasminerice is now correct
- Other typos from Rob Warner

Release 006: May, 2013

- Fixed errata from Dan Steinicke, Rob Warner, Rizwan Reza, Phillip Duba, Jakob Lehner, Patrick Berkeley, Mike Gleason, Erik Trom, Michael Kaiser-Nyman, Geoff Lanotte, Manoj Kumar, Matthias Multhaup, and Ruy Díaz Jara
- Really great close read from Kevin McGee to fix a bunch of errors and confusions
- Removed more line numbers from code descriptions, thanks to Roberto Guerra