learn.js
BOOK #1

#1

console.log('hi buds.');

introduction to javascript

SETH VINCENT

Learn.js #1

An introduction to javascript

Seth Vincent

This book is for sale at http://leanpub.com/learnjs-01

This version was published on 2014-02-03

Also By Seth Vincent

Ruby, Javascript, & Python: Development Environments for Beginners

Learn.js #2

Learn.js #3

Contents

Enter the wild and wondrous land of javascripting.

To control a computer with code can feel like wielding a weird and mighty magic. It can seem intangible and unfamiliar, but it's important to know that code is real and learnable.

Magic in popular culture typically belongs to those who are born with the power.

The magic of programming belongs to those who practice.

See what some old wizards (Gerald Jay Sussman and Hal Abelson) had to say about the similarities between programming and sorcery in a book called *Structure and Interpretation of Computer Programs*[1]:

> A computational process is indeed much like a sorcerer's idea of a spirit. It cannot be seen or touched. It is not composed of matter at all. However, it is very real. It can perform intellectual work. It can answer questions. It can affect the world by disbursing money at a bank or by controlling a robot arm in a factory. The programs we use to conjure processes are like a sorcerer's spells. They are carefully composed from symbolic expressions in arcane and esoteric *programming languages* that prescribe the tasks we want our processes to perform.

Let's be sorcerers. *Open Sourcerers*. Let's write some weird little programs, and, as they say in SICP, "conjure the spirits of the computer with our spells."

Keep coding

You're at a computer and your hands are sweaty. You have a text editor open and you're reading dense, arcane instructions.

You're about to write javascript for the first time.

It's difficult, and it doesn't make sense at first. This is normal.

You're going to make mistakes. There's a trick for dealing with that problem – a trick that works really well.

The trick is to be OK with making mistakes.

Accept that you're going to fail really hard at first.

[1] http://mitpress.mit.edu/sicp/full-text/book/book.html

I couldn't ride a bike until I was embarrassingly old – in middle school. All my friends were riding bikes, and I couldn't keep up with them unless I was on a bike. That motivated me to learn. To be a competent bike rider I had to ignore the moments when I ran into parked cars and fell over.

I had to really want it.

You will forget commas, or type semi-colons instead of colons, or type something with a capital letter that's supposed to be all lowercase.

You will run a program and spend agonizing minutes wondering why it spits errors, then realize you haven't downloaded the needed dependencies.

You're going to fuck up.

But that's OK, because you're OK with fucking up.

This book is meant to help guide you past common fuck-ups, but it won't solve all your problems for you.

Your mastery of programming relies on how motivated you are to learn, and how diligent you are in solving frustrating errors.

Keep coding.

Overview of the future

This book is the first in a series about building projects with javascript. If you haven't already, you should sign up for updates by subscribing to the Super Big Tree newsletter[2].

Thank you.

I really appreciate the support you've given by purchasing this book. I welcome you to help guide the direction of this book and the Learn.js series of javascript books. If there are particular libraries, development tools, or programming patterns that you'd like to see covered, please email me at hi@learnjs.io[3].

The Learn.js series is highly inspired by the lean publishing model (read more about it here[4]) proposed by Peter Armstrong, founder of leanpub.com. It has proven successful as a way to receive feedback from you, the readers, so that together, we can make the best book about javascript tools and libraries possible.

You'll get updates about upcoming books in the series, and I'd love to hear your thoughts on what would be most useful to you, as a reader.

[2] http://eepurl.com/rN5Nv

[3] mailto:hi@learnjs.io

[4] https://leanpub.com/manifesto

Who are you? Who am I? What is this?

The book

This book is an introductory text. You likely got that from the title. I aim for this book to be a conversational and low-barrier approach to learning javascript. Everything we work on in this book can be done with just a browser, a terminal, and a text editor.

The book covers introductory node.js, and writing client-side code using node modules and browserify.

It's meant as an introductory text that will get people up to speed for following books in the Learn.js series.

The reader

I expect that the ideal reader for this book is someone who likes exploring, imagining, and inventing for themselves. You might even have some experience with javascript already. And that's OK, because practice, and even repetition, is an important part of learning.

The author

I'm Seth Vincent. I write code, stories, and music.

I'm an independent programmer, designer and writer that is passionate about news, publishing and civic technology – particularly as it applies to local issues.

I'm a co-organizer of seattle.io[5], Code for Seattle[6], and SeattleWiki[7].

In case you couldn't tell, I currently live in Seattle, Washington.

I write books like the one you're reading, and build things like crtrdg.js, a toolkit for 2d games[8] at Super Big Tree[9].

Setting up a development environment

There's a lot of wind-up to getting started with programming. You should understand things like git, github, the terminal, and more.

[5] http://seattle.io
[6] http://codeforseattle.org/
[7] http://seattlewiki.net/
[8] http://crtrdg.github.io/
[9] http://superbigtree.com/

Instead of baking that information into each book in the series, I created a book called *Development Environments for Beginners* that helps you set up a javascript development environment (as well as ruby and python, but you can skip those sections if needed).

From that book you'll learn how to install node.js, work with version control and testing tools, best practices for automating tasks and other programming tips and tricks.

If you're feeling like you could use more information about what a development environment is and how best to set one up, you can purchase the Development Environments book at superbigtree.com/books/dev-envs[10].

If needed you can check out the Development Environments book on GitHub for free here: github.com/sethvincent/dev-envs-book[11].

Though, if you're feeling generous and able to purchase the book, that'll get you pdf, epub, and mobi versions, as well as support my work.

Node style

We will write in the style of node.js.

Even our code written for the browser will utilize the node.js style of modules thanks to browserify, a tool for bundling node modules for the browser.

This means that we won't cover the RequireJS/AMD toolset for javascript development, but will focus on node/CommonJS modules.

You'll learn more about this later in the book as we go into depth with browserify and node modules.

But for now, know that this book will be applicable to pretty much any javascript you write, and will provide additional resources for writing in the style of node.js.

This book is open source

Contribute errata or content requests at the GitHub repository for this book: github.com/learn-js/learnjs-01-introduction[12]

[10] http://superbigtree.com/books/dev-envs

[11] http://github.com/sethvincent/dev-envs-book

[12] https://github.com/learn-js/learnjs-01-introduction

Part 1: the basics

In this section, we'll get started learning:

Chrome's Developer Tools

All browsers include tools for evaluating, debugging, and auditing your code and your site's performance. This section will introduce you to the tools offered in the browser Google Chrome, and later in the book we'll go into these tools in more detail.

Basic html and css

For many of our projects in this book, html and css will be kept as minimal as possible. This refresher will get you up to speed if you haven't worked with css or html much before.

Javascript syntax, variables, data types, and functions

Here we'll go over the basic parts of javascript. We'll cover the equivalents of a programming language's grammar and punctuation, as well as the basic building blocks of javascript: strings, numbers, booleans, arrays, objects, and functions.

Node.js, npm, & browserify

Server side javascript is a seriously awesome thing, and while this book will only give an introductory look at what's possible, we'll be using many command line tools based on node.js that are installable using `npm`, node's package manager. We'll also introduce browserify, a tool for bundling Node-style modules for the browser that also allows the use of many npm and core Node modules in the browser.

Testing javascript

Writing tests for your code does two things: ensure your code works as expected when changes are made, and provides examples of usage of your project. When applicable, we'll write the tests for a project first before writing the code that does the real work, and we'll describe later why this is a useful workflow.

Chrome Developer Tools

Hello, javascript. It's nice to meet you.

```
1 console.log('hello, javascript. it's nice to meet you');
```

Open up the browser Google Chrome.

> If you don't already have Chrome installed, download and install it now at google.com/chrome[13]

Now, use this keyboard shortcut on a Mac: command + option + j Or this, for Windows/Linux: control + shift + j

You just opened the javascript console.

Type in this code:

```
1 console.log('what is this?');
```

You just told the javascript console to print some text!

Any time you put `console.log()` in javascript code that executes in the browser, whatever you put between the parentheses will show up in your browser's javascript console.

As you learn javascript, the browser console and `console.log()` will be good buddies as you prototype new functionality and debug your code.

Your first challenge:

Type this into the console:

```
1 console.error('this is an error');
```

That's an error! Note how it shows up in red. Look in the bottom right corner of the browser. You'll see a little red circle with an x in the middle, and a number on its right side. That's a helpful little indicator of errors in your javascript, and any time something is wonky with your code, that red circle will show up.

With most errors you'll also be able to see a line number from your javascript file, which will help you pinpoint the offending code. We'll get into errors and debugging in more detail later in the book.

[13] http://google.com/chrome

More developer tools

The javascript console is just one of the tools available for web development inside of Chrome. For this book, we will focus on using Chrome and its developer tools for two reasons: Chrome has a a set of versatile and powerful tools, and focusing on the tools of one browser helps keep the instructions simple.

Check out the Chrome Developer Tools documentation[14] to learn more about all that Chrome has to offer for developers working with javascript, html, and css.

It's also good to familiarize yourself with the developer tools in Firefox. Check out the Mozilla Developer Network documentation for the Firefox developer tools[15].

Recap! We learned:

- the javascript console and learned that we can type in javascript!
- that we can use code like `console.log()` and `console.error()` to print information to the console
- that Chrome has a lot of useful tools (later in the book, we'll learn how they can help with experimenting with code, auditing the performance of our site, investigating the information sent between the browser and the server, and more!)

[14] https://developers.google.com/chrome-developer-tools/
[15] https://developer.mozilla.org/en-US/docs/Tools

HTML & CSS: an introduction

If you're new to building projects for the web, knowing javascript alone won't be enough.

This book focuses on javascript, but you'll certainly pick up some html and css along the way.

You might find it useful to learn about html and css before reading Learn.js, or to have a resource handy that you can refer to when you're introduced to new html elements or css properties.

But here's a quick refresher to get you started:

An example of an html element:

```
<h1>This is a headline</h1>
```

That's an `h1` element. It is used for the most prominent headline of a document – often the site title or article title. Note that this element has an opening tag, `<h1>`, and a closing tag, `</h1>`, which looks the same except it has a forward slash, `/`.

This is a common pattern for html elements. There are a few html elements that don't require closing tags. Like these:

```
<br>
```

The `br` element creates a line break in text.

```
<hr>
```

The `hr` element creates a horizontal rule, a straight line, across your web site. It's usually used as a break between sections.

```
<img src="http://source.com/imagefilename.jpg" alt="this is the alt text of the i\
mage" />
```

The `img` element is a little unique. Note that it has a `src` attribute that specifies the image we want to have show up, and an `alt` attribute that provides text that will be displayed for screen readers, or that might be used as a caption. The `img` element is also a self-closing tag, meaning it has a forward-slash before the closing angle bracket.

HTML attributes and CSS selectors

We just saw the `src` and `alt` attributes on the `img` element. There are many attributes that can be used on any given html element. Some attributes control behavior of the element; some are used as selectors for css rules.

Here are the two attributes most used as selectors in css rules:

`id`

Using the `id` attribute of an html tag looks like this:

```
<p id="introduction">This is the first paragraph of a story.</p>
```

Here we're marking a paragraph (a `p` element) as being the introduction. The `id` of `introduction` should only be used on one html element, making this one `p` element special.

We do this so that later, in the css, we can give the `introduction` different styling than the rest of the `p` elements on the page.

Here's some css that makes the `introduction` italic:

```
p#introduction {
  font-style: italic;
}
```

We don't have to include the `p` in `p#introduction`, but it's useful for clarity. Note the hash mark: `#`. `id` attributes are identified in css by using a `#`.

`class`

Now, let's say that we want some of the paragraphs in our story to have a different background color to highlight them. One way of accomplishing this would be to add a `class` to these paragraphs.

The use of `class` attributes looks like this:

```
<p class="highlight">This paragraph will be bold and slightly bigger.</p>
```

We've given this `p` element a class of `highlight`, so let's add some css that gives this paragraph a yellow background.

```
p.highlight {
  background-color: yellow;
}
```

One of the differences between `id` and `class` attributes is that a `class` can be given to multiple elements on a page, whereas an `id` should be unique to one element. So we can give this class to multiple paragraphs to make them highlighted in yellow.

Where does the css go?

There are two options: include separate css files, or place css directly in your html file. In almost all situations, including a separate css file for your styles will be a faster and more organized option.

Here's what it looks like to embed css in your html file.

```
<!DOCTYPE html>
<html>
<head>

  <title>Your website</title>

  <style>

    p.highlight {
      background-color: yellow;
    }

  </style>

</head>
<body>

</body>
</html>
```

You'll add your css between the opening and closing `style` tags, which should in turn go between the opening and closing tags of the `head` element.

But the cleaner option is to create a separate css file that you reference from your html file:

```
<!DOCTYPE html>
<html>
<head>

  <title>Your website</title>

  <link rel="stylesheet" href="style.css" />

</head>
<body>

</body>
</html>
```

This tells the browser to load the css from the `style.css` file. Having a separate file for your styles helps with keeping your site easy to maintain. Having everything in one big html file can be a pain to work on.

So where will the javascript be in an html file?

Just as with css, there are a couple options for where you place your javascript code.

You can embed it in your html file between opening and closing `script` tags:

```
<!DOCTYPE html>
<html>
<head>

  <title>Your website</title>

</head>
<body>

<script>
console.log('this is javascript');
</script>

</body>
</html>
```

Or you can use a `script` tag to reference an external javascript file:

```
<!DOCTYPE html>
<html>
<head>

  <title>Your website</title>

</head>
<body>

<script src="site.js"></script>

</body>
</html>
```

Note that the `script` element needs a closing tag even though there's nothing between the opening tag and closing tag.

Just like with css, it's better to keep your javascript in a separate files. By separating the types of code in your project you make it easier for yourself and for others to figure out how to make changes. The more organized your project is, the quicker you can revise its design and functionality.

Layout with html and css

There are a few common tags used for laying out an html document. Check out this example:

```
<!DOCTYPE html>
<html>
<head>

<title>Your website</title>

</head>
<body>

<header>
    <h1>Your website</h1>
</header>

<main role="main">
  <p>This is the content of your website</p>
</main>

```

```
<footer>
  <p>Contact: your info.</p>
</footer>

</body>
</html>
```

We're using the `header` element for the header of the page, the `main` element for the content of the page, and the `footer` element for supplementary information that goes in the footer. All three of these tags are relatively new as part of html5. Another new tag is `section`. You might use it to break up your main content into parts:

```
<main role="main">
  <section id="part-one">
    <h1>Section one</h1>
    <p>This is the content of the first section</p>
  </section>

  <section id="part-two">
    <h1>Section two</h1>
    <p>This is the content of the second section</p>
  </section>
</main>
```

The `section` element is for breaking up your web page into distinct sections. Note that I've got `h1` tags in each of the sections as headers. In html5 anytime you create a new `section` you're allowed to start a new heirarchy of header tags.

Where `section` elements are for specifying blocks of content on your page in a semantic way, `div` elements are used primarily for the positioning and alignment of your content. For example, you might want to use a `div` to act as a container that restricts the width of your content:

```
<section id="part-one">
  <div class="container">
    <h1>Section one</h1>
    <p>This is the content of the first section</p>
  </div>
</section>
```

With css like below you can have the `section` tag the full width of the page while the container stays centered at 800 pixels.

```
.container {
  width: 800px;
  margin: 0px auto 0px auto;
}
```

The `div` is centered by using `auto` for the left and right margins (the order goes top, right, bottom, left).

More resources

These will give you an introduction to some of the html and css concepts we'll use most often in this book.

To learn html and css in more depth, check out these resources:

Don't Fear The Internet[16]
This is a wonderful introduction to html and css. Watch, enjoy, and follow along.

Web fundamentals track at codecademy.com[17]
After you've had your fears eased by Don't Fear The Internet, check out the codecademy.com web fundamentals course. It'll get you some nitty gritty experience with the basics.

Mozilla Documentation[18]
Have a question about some css property or html element? The Mozilla Developer Network has awesome documentation. If you're searching on google.com for anything css/html/js related, add the abbreviation "mdn" to your search query to see results from Mozilla Documentation. This site also has a bunch of introductory tutorials that are really useful.

WebPlatform.org[19]
This is a newer resource, but a good one. It's got a great design and well-organized resources.

[16] http://www.dontfeartheinternet.com/
[17] http://www.codecademy.com/tracks/web
[18] https://developer.mozilla.org/en-US/
[19] http://www.webplatform.org/

JavaScript variables

Creating a variable:

```
var nameOfVariable;
```

Variables are written in camelCase, meaning that the first letter of the first word is in lowercase, and if the variable is made up of multiple words, then the first letter of the following words are capitalized.

Creating a variable that references a string:

```
var thisIsAString = 'this is a string';
```

Surround strings with single quotes.

Creating a variable that references a number:

```
var thisIsANumber = 3.14;
```

Numbers do not have quotes around them.

Creating a variable that references an array:

```
var thisIsAnArray = [1, "two", [3, 4]];
```

Note that one of the values in the array is a number, one is a string, and another is an array. Arrays can hold any value, in any order.

Accessing the values in an array:

```
thisIsAnArray[0];
```

The above will return the number `1`. Arrays use numbers as the index of their values, and with javascript an array's index always start at `0`, making `0` reference the first value of the array.

```
thisIsAnArray[1];
```

This returns the string ‘two’;

How would you return the number `4` from the nested array?

Like this:

```
thisIsAnArray[2][1];
```

Creating a variable that references an object:

```
var thisIsAnObject = {
  someString: 'some string value',
  someNumber: 1234,
  someFunction: function(){
    return 'a function that belongs to an object';
  }
};
```

Here we’re setting `someString` to `'some string value'`, `someNumber` to `1234`, and `we're creating a function named` someFunction `that returns the string` ‘a function that belongs to an object’‘. So how do we access these values?

To get the value of `someString` using dot notation:

```
thisIsAnObject.someString;
```

Or using bracket notation:

```
thisIsAnObject['someString'];
```

To get the value of `someNumber` using dot notation:

```
thisIsAnObject.someNumber;
```

Or using bracket notation:

```
thisIsAnObject['someNumber'];
```

To use the function `someFunction` using dot notation:

```
thisIsAnObject.someFunction();
```

Or using bracket notation:

```
thisIsAnObject['someFunction']();
```

Using square bracket notations with functions looks a little wacky. It will be useful if you are storing function names in variables as strings, and need to use the variable to call the function being stored. Otherwise, stick with dot notation. That goes for other attributes on an object, too – stick with dot notation unless there's a really good reason to use bracket notation.

Operators & arithmetic

How do I check if something is equal, not equal, less than or greater than?

Equals:

```
===
```

Examples:

```
true === false
// returns false

'pizza' === 'pizza'
// returns true

123 === '123'
// returns false
```

You may have noticed there are three equals signs. If you've tried out other programming languages, you'll probably remember that two equals signs were used to check equality, like this: `something == otherThing`.

That works in javascript, too, but there's a difference in the behavior of == and ===.

When using ==, the types of your values will be coerced into being the same.

Open your javascript console and type this in:

```
'123' == 123
```

This will return true, because the == operator is coercing the values to be the same type.

Now try this one out in the javascript console:

```
'123' === 123
```

This returns false, because when using the === operator it checks to make sure the two values are of the same type, and if not, returns false. If the two values are the same type, then it'll do the usual equality check.

For this reason it is a good idea to use === rather than == in most situations.

Does not equal:

```
!==
```

Examples:

```
'pizza' !== 'gross'
// returns true
```

Like the === operator, `!==` has a counterpart that coerces the type of values, `!=`.

```
123 !== '123'
// returns true
```

Greater than and less than:

```
>
<
```

Examples:

```
1 < 3
// returns true

'pizza'.length > 'poop'.length
// returns true
```

With this example: `'pizza'.length > 'poop'.length`, the `.length` method returns how many characters are in the string, so it's really evaluating those numbers: `5 > 4`.

Greater than or equal to, less than or equal to:

```
>=
<=
```

Examples:

```
i = 0;
i <= 10;
// returns true

[1, 2, 3].length >= [4, 3, 2, 1].length
// returns false
```

Like the example above, using the .length property on an array returns the number of items in the array, so a statement like this:

```
[1, 2, 3].length >= [4, 3, 2, 1].length
```

Is really being evaluated like this:

```
3 >= 4
```

How do I group equality statements?

With logical operators `&&` and `||`.

By using `&&`, you're saying that all of the comparison checks must return true.

By using `||`, you're saying that at least one of the comparison checks must return true.

Examples:

```
true && false
// returns false

3 < 10 && 'pizza'.length > 3
// returns true

false || true
// returns true
```

You can also alter the boolean value that's returned from a variable by using `!`. Any time you put `!` it'll return the opposite of its actual boolean value.

Examples:

```
!true
// returns false

!false
// returns true

var pizza = null;
!pizza
// returns true
```

It's like saying "NOT false", or "NOT true".

How do I do math?

Add:

```
+
```

Examples:

```
3 + 5
// returns 8

'abc' + 'def'
// returns 'abcdef'
```

Wait, those strings just glomped together!

The + operator will add two numbers together, and it will also concatenate two strings, meaning that the two strings will be combined into one.

For fun, you should open the javascript console and experiment with using the + operator with arrays. The results of operations like this one are messy and unexpected:

```
['a', ['b', 3]] + [1, 2, 3, 'a', 'b', 'c']
```

The + operator is the only arithmetic operator that has this multipurpose behavior, so you won't be able to subtract, multiply, or divide strings with the relative operators.

Subtract:

```
-
```

Examples:

```
5 - 3
// returns 2

1 - .1
// returns .9
```

Note that any time an operator is used with a mix of integer and float numbers, the result will typically be a float.

Multiply:

```
*
```

Examples:

```
var x = 5;
x * 10;
// returns 50

var pizzasIWantToEat = 23;
var percentageIWillActuallyEat = .033;
pizzasIWantToEat * percentageIWillActuallyEat;
// returns 0.759
```

Divide:

```
/
```

Examples:

```
6 / 2
// returns 3

var i = 21
i / 3
// returns 7
```

Remainder:

```
%
```

You can check to see if there's a remainder from division really easily by using the remainder operator.

Examples:

```
12 % 4
// returns 0

3 % 2
// returns 1
```

This is really useful for doing things like checking to see if a number is even or odd.

Parentheses

Just like in the math you learned in school, you can use parentheses to group operations.

Examples:

```
(3 + 3) * (21 / 7)
// returns 18

3 + 3 * 21 / 7
// returns 12
```

The Math object

A great resource for learning more about javascript's Math object is the Mozilla Developer Network's javascript documentation[20].

[20] https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Math

Control flow: making decisions in code

> How do I control the flow of a program?

There are a few basic approaches for controlling the flow of a javascript program.

"if" statements

An if statement looks like this:

```
if (something === true){
  /* do something */
}
```

An extended example might look like this:

```
if (hungerLevel > 10) {
  eat(food).fast();
} else if (hungerLevel > 5) {
  prepare(food).leisurely():
} else {
  do(somethingElse);
}
```

Note that you can use `else if` to check follow-up values, and that you can use `else` for any other case, as a kind of fallback.

Loops

"while" loops

```
var i = 0;

while (i < 10){
  console.log(i);
  i = i + 1;
}
```

“for” loops

Any time you want a program to do something a specific number of times, or you want to perform an action on every item in an array or object, you’ll likely end up using a loop.

“for” loops

A for loop

A simple for loop looks like this:

```
for (var i = 0; var i < 10; i++){
  console.log(i);
}
```

You can use this to loop through items in an array like this:

```
var alpha = ['a', 'b', 'c'];
var alphaLength = alpha.length;
for (var i = 0; i < alphaLength; i++){
  console.log(alpha[i]);
}
```

Run this in your javascript console and you’ll see the items in the array being logged to the console one at a time.

There’s an alternate way to iterate through arrays that is somewhat supported in browsers, and is fully supported in Node, the `.forEach` method.

Here’s an example:

```
var alpha = [‘a’, ‘b’, ‘c’];
alpha.forEach(function(item, i, array){
  console.log(item);
});
```

The forEach method takes a callback function that executes once for every item in the array.

The “for … in” loop

The forEach method works great for arrays, but for objects I typically use the “for … in” loop.

```
var foods = {
  pizza: ['cheese', 'dough', 'sauce'],
  taco: ['tortilla', 'cheese', 'hot sauce']
}

for (var type in foods) {
  console.log (type + ': ' + foods[type]);
}
```

Functions

Eating, digesting, and pooping.

A function is a block of code that takes input, processes that input, and then produces output.

You can think of it like eating, digesting, and pooping.

And when we use a number of functions in succession, it's almost like that movie The Human Centipede[21], only less gross.

Let's make a function named `eat`.

```
1  // take input / eat food
2  function eat(food){
3
4    // process the input / digest the food
5    var poop = digest(food);
6
7    // send output / poop
8    return poop;
9  }
```

The above example should make sense just from reading it.

Note that lines that start with `//` are comments, and they get ignored when the code is executed.

To create a function, we first write `function`. Next, we can name the function, and in this case it is named `eat`.

Inside of the parentheses we specify the input, which are also called arguments. We only have one argument in this case, named `food`.

Next, we use an opening curly bracket to indicate the beginning of the block of code connected with this function.

We create a variable named `poop`, which contains a "digested" form of the `food` argument. Here we're using another function named `digest` that is using the `food` argument as its own input.

Finally, we `return poop;` so that the output of this function can be used in other parts of our code.

[21] http://www.imdb.com/title/tt1467304/

Using the `eat` function:

We can use the `eat` function like this:

```
eat('pizza');
```

When we run this, it'll return something like `zpzia`, `apizz`, or `pzazi`. You know, something random like that.

So what is the `digest` function doing?

You've probably already guessed that it is a function that randomly shuffles letters in a string. In actual production projects you would want to name it something a little more clear, like`shuffleLetters()`.

Here's an example of the `shuffleLetters()` function using our food/poop language:

```
function digest(food){
  var food = food.split('')
  var digesting = food.length, digested, randomFoodPart;

  while (digesting) {

    randomFoodPart = Math.floor(Math.random() * digesting--);

    digested = food[digesting];

    food[digesting] = food[randomFoodPart];

    food[randomFoodPart] = digested;
  }

  var poop = food.join('');

  return poop;
}
```

Adapted from Mike Bostocks's Fischer-Yates Shuffle[22].

You're probably aware that the `digest` function is doing the heavy lifting, while the `eat` function is just a wrapper around `digest`.

If you were really modeling eating, digesting, and pooping using javascript functions, how would you do it?

[22] http://bost.ocks.org/mike/shuffle

Introduction to callbacks.

A callback is a function that you pass as an argument to another function. Typically, you'll use a callback as a way to work with data after it's been processed by a function.

A simple callback looks like this:

```
function holla(callback){
  callback();
}
```

Note how we're setting up an argument named `callback`, then calling that argument as a function: `callback()`.

Usage if the `holla` function:

```
holla(function(){
  console.log('this is part of the callback function);
});
```

Our function named `holla()` takes one argument, and we expect it to be a function. In this example we're using an anonymous function, but we could use a named function like this:

```
function holla(callback){
  callback();
}

function back(){
  console.log('this is part of the callback function);
}

holla(back);
```

But usually we're providing a function some kind of parameter, that function performs an action on the parameter, then we use the callback to work with the output of the function we called. A simple example looks like this:

```
// create an eat function
function eat(food, callback){
  var food = food + " was eaten";
  callback(food);
}

// create a poop function to use as the callback
function poop(output){
  console.log(output);
}

// call the eat function, passing a food and the poop function as arguments
eat("pizza", poop);
```

Note that when we pass the callback function `poop` as an argument we don't write it like `poop()`. This would *call* or execute the function, and we don't want that to happen when we pass the `poop` function as an argumet. The `poop` function gets called later inside the `eat` function.

Getting started with Node.js

Node.js is server-side javascript. It is well-suited to real-time applications and systems that are heavy on input and output. You can use it to create web servers, to manage information in databases, to build real-time communication tools, and more.

To learn node in detail, read these resources in this order:

- art of node[23]
- streams handbook[24]

Install node:

There are a few options for this, and I've put them in my order of preference:

Use nvm to manage node versions.

This option gives you the most control, allows you to switch between versions of node similar to using rvm or rbenv for Ruby. Get nvm here[25]. This is the method I use, and the one I recommend.

If you're on a Mac you'll need to first install Xcode, Apple's developer tools. You can now do this through the Mac App Store[26].

You should then install homebrew. Homebrew is a package manager for Macs. You can install it with this command:

```
1  ruby -e "$(curl -fsSL https://raw.github.com/mxcl/homebrew/go)"
```

Install using a package manager.

This is a good option, but sometimes package managers can be out of date. If the node version you'll be using matters for your project, you should make sure that the version in the package manager works for you. Check out a list of package manager instructions here[27].

[23] https://github.com/maxogden/art-of-node
[24] https://github.com/substack/stream-handbook
[25] https://github.com/creationix/nvm
[26] https://itunes.apple.com/us/app/xcode/id497799835?mt=12
[27] https://github.com/joyent/node/wiki/Installing-Node.js-via-package-manager

Download an installer from nodejs.org.

Here's the node.js download page[28].

Installing node gives us the node package manager `npm`. We'll use it to install a wide range of packages, including web frameworks, game dev libraries, and client-side javascript modules.

> This section of the book is still a work in progress. Make suggestions at github.com/learn-js/learnjs/issues[29].

[28] http://nodejs.org/download
[29] http://github.com/learn-js/learnjs/issues

Introduction to npm

You use `npm` to install javascript modules. It is most often used for installing node.js modules, but it is not limited to server side code.

You might expect `npm` to stand for Node Package Manager. But that's misleading – it reinforces the idea that `npm` is just for node.js projects.

Instead, we can follow the guidance of javascript developer Max Ogden and let is stand for Node Packaged Modules.

Read more about this idea on his blog post: maxogden.com/node-packaged-modules.html[30], where he presents three projects based on the tool `browserify`. We'll make heavy use of browserify in the book.

Getting started with `npm`

If you've alreadt installed node.js, you've got npm.

Check the version of `npm` like this:

```
npm -v
```

This should return something like:

```
1.2.32
```

The exact version numbers might differ, and that's ok.

Installing a module will automatically place the module in a folder named `node_modules` in your current working directory.

Make a new folder:

```
mkdir npm-experiments
cd npm-experiments
```

Install browserify:

[30] http://maxogden.com/node-packaged-modules.html

```
npm install browserify
```

Now, if you look in the node_modules directory, you'll see browserify!

You can also install modules globally, typically so that you can run their commands at any time in any directory. This is useful with a module like browserify, so let's try it out:

```
npm install -g browserify
```

It's the `-g` option that install the module globally.

You can delete the node_modules directory:

```
rm -rf node_modules
```

And run the browserify command:

```
browserify
```

Without any options it'll only return help text. For more about browserify, check out the Introduction to browserify section.

To learn more about `npm` run the command without any options:

```
npm
```

This gives you a full list of the commands and options available through `npm`. To learn about any particular command you can run:

```
npm help name-of-command
```

Finding modules

You can check out npmjs.org[31] as well as npmsearch.com[32] to find modules that you can use in your projects.

You can also run the `search` command in the terminal:

[31] http://npmjs.org
[32] http://npmsearch.com

```
npm search template
```

This will return a bunch of modules related to templates!

Creating a module

Any time you're using javascript modules from `npm` in a project you should create a package.json file. In this file you can save the dependencies for your project, along with license, author, repo inforamtion, and other details.

You can use the `npm init` command to generate a package.json file:

```
npm init
```

Answer the questions.

When you're done, you'll have a package.json file. You can install modules and save them as dependencies of your project like this:

```
npm install request --save
```

And if you are installing a module (like a test framework) as a development dependency, you can do so like this:

```
npm install tap --save-dev
```

Here's an example of an extremely simple module:

Take a string as input, and output that string in all uppercase letters:

```
module.exports = function(someString){
  return someString.toUpperCase();
}
```

Put that module definition in a file called index.js.

Now, create a file named test.js, and enter this text:

```
var upperize = require('./index');

var pizza = upperize('pizza is really awesome!');

console.log(pizza);
```

Run the test by typing this into your terminal:

```
node test.js
```

You should see this output:

```
PIZZA IS REALLY AWESOME!
```

In upcoming sections you'll learn about how to use node modules in the browser with browserify, and learn how to do more useful testing using a node module named tape.

Publishing modules

Once you've written a module, you can publish it to `npm` super easy:

```
npm publish .
```

Before running the `npm publish` command you'll want to edit your package.json file to make sure that properties like version, author, homepage, and repository are all filled in. You should also first create a useful readme.md file, as that is displayed on a modules project page on npmjs.org[33].

[33] http://npmjs.org

Introduction to browserify.

There's all this wonderful code on `npm`, the node.js package manager.

What if we could use that code in the browser?

Hey, we can. Use browserify.

With browserify[34], we can use some core node modules and many of the thousands of modules on `npm` in our browser-side code.

We can also write our browser-side javascript in the node.js style by using `require`.

Install browserify:

```
npm install -g browserify
```

We use the `-g` option to install browserify globally on your machine, allowing you to use it on the command line.

Brief example:

```
// require the core node events module
var EventEmitter = require('events').EventEmitter;

//create a new event emitter
var emitter = new EventEmitter;

// set up a listener for the event
emitter.on('pizza', function(message){
  console.log(message);
});

// emit an event
emitter.emit('pizza', 'pizza is extremely yummy');
```

Put the above code in a file named index.js.

Now, to be able to run this code in the browser, enter this command in the terminal:

[34] https://github.com/substack/node-browserify

```
browserify index.js > bundle.js
```

The bundle.js file now has your event emitter code along with any dependencies on core node modules and shims to make them work in the browser.

You can include bundle.js in your html now like any other javascript file.

Example:

```
<!DOCTYPE html>
<html>
<head>
  <title>node / browserify example</title>
</head>
<body>

<script src="./bundle.js"></script>
</body>
</html>
```

That's it! Now you can use node modules and `require` in the browser!

Live reload development environment

If you're in the middle of writing code, you'll find running `browserify` in the terminal to regenerate bundle.js, then refreshing the browser to be time-consuming and annoying.

Enter beefy!

`beefy` is a command-line tool for automatically generating and serving your browserify bundles as you develop. Each time you save your javascript file `beefy` will regenerate bundle.js and refresh the browser automatically.

Install beefy:

```
npm install -g beefy
```

Now, run this:

```
beefy index.js:bundle.js --live
```

The `--live` option enables the live reload functionality of beefy.

This will by default serve your index.html file at http://localhost:9966. Open Chrome, enter that url, then open the javascript console by using the keyboard shortcut `command+option+j`.

You'll see `pizza is extremely yummy` in the javascript console!

Introduction to testing with tape

We'll be testing code using tape[35].

tape

tape is a simple, easy to learn testing library for javascript.

Install tape as a development dependency:

```
npm install tape --save-dev
```

Example:

Here's a simple example of using tape to check if two strings are the same:

```
var test = require('tape');

var food = 'pizza';

test('string test', function(t){
  t.plan(1);

  t.equal('pizza', food);
});
```

Let's step through this example line by line.

In this line we require the tape module, effectively importing its functionality, and assign tape to a variable named test:

```
var test = require('tape');
```

Here we assign the string `'pizza'` to a variable named `food`:

[35] https://github.com/substack/tape

```
var food = 'pizza';
```

This shows usage of the `test` function that we created by requiring the `tape` module:

```
test('string test', function(t){
```

The first argument is an arbitrary string, whatever description you want to explain what you're testing. The second argument is a callback, which gets the `t` callback that's used to actually test our code.

In the following line we specify how many things you will test:

```
t.plan(1);
```

Here's an actual test, where we make sure that the variable `food` is equal to the string `'pizza'`:

```
t.equal('pizza', food);
```

Here we close the callback function with a curly brace, close the call to the test function with a parentheses, and end the statement with a semi-colon:

```
});
```

Introduction to git & GitHub

Developing websites and applications without using git is equivalent to writing in Microsoft Word without ever saving your work.

Use git.

Git is a version control system, which means it can track every change you make to your code. This allows you to review and edit past versions if you mess something up. And it allows you to figure out when errors were introduced to the code.

There are many other bonuses to using git, which are mostly out of this book's scope.

The best way to start learning git (and GitHub) is to visit try.github.com[36]. You should also try githug, a game for learning git[37].

Here are some basics of using git:

Create a git repository:

```
cd name-of-folder
git init
```

Add files:

```
git add name-of-file

// or add all files in directory:

git add .
```

When you add files to a git repository they are "staged" and ready to be committed.

Remove files:

[36] http://try.github.com

[37] https://github.com/Gazler/githug

```
git rm name-of-file

// force removal of files:

git rm -rf name-of-file-or-directory
```

Commit files and add a message using the `-m` option:

```
git commit -m 'a message describing the commit'
```

Create a branch:

```
git branch name-of-branch
```

Checkout a branch:

```
git checkout name-of-branch
```

Shortcut for creating a new branch and checking it out:

```
git checkout -b name-of-branch
```

Merge a branch into the master branch:

```
git checkout master
git merge name-of-branch
```

Add a remote repository:

```
git remote add origin git@github.com:yourname/projectname.git
```

List associated repositories:

```
git remote -v
```

Pull changes from a remote repository:

```
git pull origin master
```

Push changes to a remote repository

```
git push origin master
```

Checkout a remote branch:

```
git checkout -t origin/haml
```

Get on GitHub

If you haven't already, create an account at github.com[38].

GitHub is a great place to host your code. Many employers hiring for developer and designer positions will ask for a GitHub profile, and they'll use your GitHub activity as part of the criteria in their decision-making process.

In fact, if you're looking to get a job with a particular company, try to find *their* GitHub profile and start contributing to their open source projects. This will help you stand out, and they'll already know your technical abilities based on your open source contributions. That's a big win.

GitHub has become the de facto code hosting service for most open source communities.

With GitHub Pages you can:

- design a website any way you want by having complete control over the html, css, and javascript.
- use simple templates for getting started using GitHub Pages.
- create sites for yourself and all of your projects hosted on GitHub.
- use a custom domain name if you want!

Visit the help section for GitHub Pages[39] to learn more details about hosting sites on GitHub.

[38] http://github.com

[39] https://help.github.com/categories/20/articles

Create a site for yourself using GitHub

GitHub has a useful service called GitHub Pages[40] that allows you to host a simple site on their servers for free.

To get started, fork this simple template: github.com/maxogden/gh-pages-template[41].

Visit that github project, make sure you're logged in, and click Fork in the upper right side of the screen.

Fork gh-pages-template to your personal account.

Rename the repository from gh-pages-template to whatever you want by clicking on Settings on the right side of your fork of the repository, and changing the name there. GitHub will warn

That's it! You now have a website hosted through GitHub Pages.

You'll be able to visit your site at **YOUR-USERNAME**.github.com/**YOUR-PROJECT-NAME**.

You'll want to edit the content though, right? Add your cat pictures or resume or pizza recipes? You can do that.

You can create, edit, move, rename, and delete files all through the GitHub website. Check out these blog posts on GitHub for details on how to do those things: - Create files[42] - Edit files[43] - Move and rename files[44] - Delete files[45]

You can also clone the project repository onto your computer:

```
1 git clone git@github.com:__YOUR-USERNAME__/__YOUR-PROJECT-NAME__.git
```

You can copy the git url to clone from the right-hand sidebar of your project repository.

After cloning the repository, `cd` into it and make some changes:

```
1 cd __YOUR-PROJECT-NAME__
2 nano index.html
```

Add a bunch of content to index.html, and change the styles in style.css.

After you've made some changes, add them to the repo and commit the changes:

[40] http://pages.github.com
[41] https://github.com/maxogden/gh-pages-template
[42] https://github.com/blog/1327-creating-files-on-github
[43] https://github.com/blog/143-inline-file-editing
[44] https://github.com/blog/1436-moving-and-renaming-files-on-github
[45] https://github.com/blog/1545-deleting-files-on-github

```
git add .
git commit -m 'include a brief, clear message about the changes'
```

Now, push your changes back to GitHub:

```
git push origin gh-pages
```

Introduction to grunt.js

Grunt is a tool for managing the javascript, css, and html files of your web project. Grunt is a task manager similar to Ruby's `rake`. You can run any arbitrary tasks you want, and there are a number of grunt plugins that make it easy to set up common tasks. Grunt is useful for running tests or for build steps, including turning sass, stylus, or less files into css, concatenating files, or creating .zip or .tar.gz packages of your project.

Outline of the steps in this tutorial:

- Install node.
- Install grunt-cli.
- Setup project.
- Set up package.json.
- Create Gruntfile.js.
- Run grunt tasks.
- Make an awesome project.

Install node:

You should already have Node.js installed from chapter 4, "In-depth with Node.js". If not, backtrack to that chapter for a guide to installing Node.

Install grunt-cli

Installing node gives us the node package manager `npm`. We'll use it to install grunt-cli, which is the command-line tool that is used to run grunt tasks.

Run this in your terminal after installing node.js:

```
1  npm intall -g grunt-cli
```

This installs the grunt command-line tool globally on your machine. Now you can run the `grunt` command!

And, it won't do anything.

Bummer. **But it will give you a message like this:**

```
grunt-cli: The grunt command line interface. (v0.1.6)
  Fatal error: Unable to find local grunt.
  If you're seeing this message, either a Gruntfile wasn't found or grunt hasn't \
been installed locally to your project. For more information about installing and\
 configuring grunt, please see the Getting Started guide: [http://gruntjs.com/get\
ting-started](http://gruntjs.com/getting-started)
```

The grunt command looks for a locally installed version of grunt, which you can include in your project as a development dependency in a package.json file.

Hey, package.json files are cool.

You can use a package.json file for a lot of useful purposes. Primarily, it's used to list your project's dependencies on npm packages, as well as list the name, description, version, and source repository of the project. You can specify the type of license, version of node the project requires, the project's contributors, and more. Check out [this interactive package.json cheat-sheet][http://package.json.nodejitsu.com/] for a nice rundown on the basics.

So, our package.json will specify some development dependencies.

Some basic requirements:

- We'll test the javascript with qunit.
- We'll write scss and compile it to css, then minify the css.
- We'll concatenate and uglify our javascript files.
- We'll use the `grunt watch` command to automatically run grunt tasks when files are edited.
- We'll want a little http server to check out our game as we're developing it.

Some of the above requirements could be perceived as excessive, but they help make this a meaty and useful tutorial, so deal with it.

So, we'll need to use some grunt plugins. We'll use these ones:

- [grunt-contrib-qunit][https://github.com/gruntjs/grunt-contrib-qunit]
- [grunt-contrib-jshint][https://github.com/gruntjs/grunt-contrib-jshint]
- [grunt-contrib-connect][https://github.com/gruntjs/grunt-contrib-connect]
- [grunt-contrib-watch][https://github.com/gruntjs/grunt-contrib-watch]

That means our package.json file will look like this:

```
{
  "name": "your-project-name",
  "version": "0.0.1",
  "author": "Super Big Tree <seth@superbigtree.com>",
  "description": "A silly game.",
  "repository": {
    "type": "git",
    "url": "https://github.com/your-profile/your-project-name.git"
  },
  "devDependencies": {
      "grunt": "~0.4.0",
      "grunt-contrib-qunit": "~0.2.0",
      "grunt-contrib-jshint": "~0.1.1",
      "grunt-contrib-connect": "~0.1.2",
      "grunt-contrib-watch": "~0.4.4"
},
 "license": "MIT",
 "engines": {
 "node": ">=0.8"
 }
}
```

Go to your terminal. Create a folder that you want to serve as the project's folder:

```
cd wherever/you/want/the/project/to/live
mkdir your-project-name
cd your-project-name
```

Now, create your package.json file:

```
touch package.json
```

Copy and paste the above package.json example into your package.json file using your favorite text editor. Save the file. **Now, you can run this:**

```
npm install
```

to install all the dependencies.

If you run the command and get an error like this at the end, then something is not ok:

```
npm ERR! not ok code 0
```

There's an error of some kind that will need to be worked out. For me, typically the problem is that I messed up the syntax or put the wrong version number for a dependency, so check things like that first.

Project setup:

Let's make all our files and folders now!

This will make all the folders we want:

```
> mkdir -p test js css/scss img
```

This will make the files we want:

```
touch js/player.js js/game.js js/enemies.js js/ui.js \
touch css/scss/main.scss css/scss/reset.scss css/scss/ui.scss \
touch test/player.js test/enemies.js test/game.js test/ui.js
```

Cool. Did that. **Now we make the Gruntfile:**

```
touch Gruntfile.js
```

Open Gruntfile.js in your favorite editor and paste in this:

```
module.exports = function(grunt) {
  grunt.initConfig({
    // and here we do some cool stuff
  });
};
```

The above code is the required wrapper code to make a Gruntfile work. Now, remember our package.json file. Buds, we can use the values from that file in our Gruntfile.

**Check it out: **Let's say we're making a javascript library and want to put stuff like the name, version, author, source repository, and license of the project in a multi-line comment at the top of the file. It would be a bummer to have to edit that by hand every time the file is compiled for a new release. Instead, we can use values from package.json in our Gruntfile!

First step is to read the contents of package.json by **putting this line in Gruntfile.js:**

```
pkg: grunt.file.readJSON('package.json');
```

A package.json file is just JSON, right? Yeah, so it's easy to get at the values to do cool stuff.

For fun, let's see what it takes to run a custom task inside a Gruntfile, and have it log some attributes from the package.json file. Alright? OK.

This is a really simple task that logs the package name and version to the console, shown here as the complete Gruntfile.js:

```
module.exports = function(grunt) {
  grunt.initConfig({
    // read the json file
    pkg: grunt.file.readJSON('package.json'),

    log: {
      // this is the name of the project in package.json
      name: '<%= pkg.name %>',

      // the version of the project in package.json
      version: '<%= pkg.version %>'
    }
  });

  grunt.registerMultiTask('log', 'Log project details.', function() {
    // because this uses the registerMultiTask function it runs grunt.log.writeln\
()
    // for each attribute in the above log: {} object
    grunt.log.writeln(this.target + ': ' + this.data);
  });
};
```

You can now run your task on the command line!:

```
grunt log
```

You should get output like this:

```
Running 'log:name' (log) task
name: your-project-name
Running 'log:version' (log) task
version: 0.0.1
Done, without errors.
```

If you didn't get output like that, check your Gruntfile for typos. If you did get output like that: Awesome! So we've made it pretty far. We've set up a project with a bunch of files and folders, created a package.json file with a list of devDependencies, installed the dependencies, and tried out a simple Gruntfile for running arbitrary tasks.

If this seems like a lot, like it's beating up your brain, don't worry. After a few times of starting a project like this, these initial steps will get faster and easier. Heck, you might even create some kind of base project that you can build on with each new project so that you don't have to write the boilerplate every time. Or you could use a project like yeoman for its code generators. That's up to you, but when first learning this it's a reasonable idea to start from scratch and see how everything works.

Package managers for browser code: what should I use?

When should I use npm, bower, or component for managing client-side code?

To start: always use one of them. The old days of downloading js/css libraries one by one and updating them manually are totally over, and that's exciting. Using a package manager for front-end code means we can easily download libraries, keep track of versions, and ease dependency management.

But we still have too many options for how to manage our client-side code. Beyond npm, bower, and component there are still many other options, but those three seem to be settling in as the most widely adopted.

And all three are quite different:

npm

npm nominally started out as a package manager for node, but now is used for any javascript, and along with a tool like browserify it's easy to use npm packages and node-style modules in the browser. It's not super useful for css libraries – but it's easy to imagine a tool (built on something like brfs) that could make bundling css npm packages super pleasant.

bower

bower is a clear hero of client-side code: it's great for both css and javascript. It's easy to manage dependencies – even if those dependencies don't have a config file for bower. You can install a file or git repository as a dependency alongside packages in the bower registry. Bower doesn't make any assumptions about how you build or deploy code, so it is compatible with amd modules.

component

component is a tool with a more focused and defined goal than npm and bower. It uses common js style modules. Each component may contain javascript, css, fonts, and images. Some javascript-only components can also be used in node.js. See the Component FAQ[46] for more details.

[46] https://github.com/component/component/wiki/F.A.Q

When I use npm for browser code:

Games. There's been a lot of interesting activity in javascript game development in the node.js community using browserify. voxel.js is one of the biggest examples, with a goal of creating minecraft-like games in the browser. See the work of these people for examples:

npm/browserify will also be useful for creating applications that might share javascript code on the server and the browser. This approach also works well when the javascript requirements for the client-side are minimal, or if you prefer to write client-side code in a node style.

When I use bower for browser code:

When the project requires a front-end framework like backbone, angular, or ember, I use bower as the package manager. It's currently the best way to package arbitrary dependencies of a javascript application. Typically using bower means that I am also using the build tool grunt.

I don't really use component yet.

I really want to. It seems like there's some great work happening around component. I expect that there will be instances when I'll want to use some of the available components. But when that happens, I'll probably need to figure out a way to integrate component with bower or npm/browserify.

Use all three at once!

It's possible to use packages from both bower and component while using browserify!

Check out this great guide for using components, bower packages, amd modules, and even global variables with browserify: http://dontkry.com/posts/code/browserify-and-the-universal-module-definition.html

Part 2: In depth with javascript data types

In this section of the book we'll review strings, numbers, arrays, and objects, and focus on a problem that often comes up with each of them.

Each chapter will follow this pattern:

- Present a problem that needs to be solved
- Describe a possible solution that can be written as a Node.js-style module
- Write tests for the module
- Write the actual module
- Show usage of the module

Strings

Problem

We have a large amount of text that has weird strings in it that need to be replaced.

How we might solve it

We can use the String.replace() method.

It would be cool if there were a function that accepted a buffer or a string, replaced the characters we want to remove with whatever we want instead, and returned it as either a buffer or a string, depending on what the input was.

That approach assumes that when we are working with buffers, we want the data stay a buffer.

We haven't talked about buffers before in this book. You can learn more about them by reading the node.js docs about buffers[47].

Buffers are used for dealing with binary data. A likely reason for using buffers: you're reading or writing files using node.js.

Create a module

Our module will be a single function that takes three arguments: - the string or buffer that has a substring we want to replace - a string or regular expression that represents the substring we want to remove - an optional string that will replace whatever we want to remove

Tests

Let's write a test that shows how our module might be implemented.

[47] http://nodejs.org/api/buffer.html

```
var test = require('tape');
var replace = require('./');

var food = 'pizza $$$ awesome';

test('string test', function(t){
  t.plan(1);

  food = replace(food, '$$$', 'is');

  t.equal('pizza is awesome', food);
});
```

Implement the module

```
module.exports = function(str, remove, replacer){
  var replacer = replacer || '';
  var buffer = false;
  var str = str;

  if (Buffer.isBuffer(str)){
    str = str.toString();
    buffer = true;
  }

  str = str.replace(remove, replacer);

  if (buffer){
    str = new Buffer(str);
  }

  return str;
}
```

Usage

The test we wrote shows pretty clearly how to use this module, but here's an example of using it with a string:

```
var replace = require('./index');

var messyString = 'this is a WHAT string';

cleanString = replace(messyString, 'WHAT ');
```

Because we made the 3rd argument optional, this will just remove the `'WHAT '` substring to return this:

```
this is a string
```

Here's another example that works with a buffer and uses a regular expression to find the substring that should be removed:

Create a file named example.txt and save it with this text:

```
This is some #we#ird text that has #a bunch# of pound sig#ns mixed in. #wtf.
```

```
var fs = require('fs');
var replace = require('./index');

var file = 'example.txt';

fs.readFile(file, callback(err, content){
  if (err) throw err;

  fs.writeFile(file, replace(content, /#/g));
});
```

We're using this regular expression: `/#/g`.

A regular expression has forward slashes (`/`) at the beginning and end, followed by optional parameters that change the behavior of what the regexp matches. By placing `g` at the end of the regexp our replace statement will find all instances of the pound symbol. Without the `g` only the first instance of the `#` would be matched.

After running `node test.js` in the terminal the file should now look like this:

```
This is some weird text that has a bunch of pound signs mixed in. wtf.
```

Note that this usage with the `fs` module from node.js won't work in the browser, because you won't be able to read or write files from the browser in this way. In the browser you'll typically be working with strings instead of buffers anyway.

Numbers

Problem

We need a bunch of random numbers to use in a game. They all need to have a different range, so we need to set the minimum and maximum values. We also need to be able to specify if the numbers should be integers or floats.

How we might solve it

Let's make a module that takes three arguments, the mimimum value, the maximum value, and an optional boolean that if set to `true`, ensures that the function only returns integers.

Create a module

Tests

```
var test = require('tape');
var randomizer = require('./');

test('string test', function(t){
  t.plan(2);

  t.ok(randomizer(1, 5) % 1 !== 0);
  t.ok(randomizer(1, 5, true) % 1 === 0);
});
```

Implement the module

```
module.exports = function(min, max, int){
  var num = Math.random() * (max - min) + min;

  if (int){
    return Math.floor(num)
  }

  return num;
}
```

Usage

Here's an example that returns a random float between 1 and 5:

```
var randomizer = require('./math');

console.log(randomizer(1, 5));
```

And here's an example that returns a random integer between 1 and 5:

```
var randomizer = require('./math');

console.log(randomizer(1, 5, true));
```

Arrays

Problem

We have a bunch of arrays that need to be put together into one array, then ordered alphabetically.

How we might solve it

We can use the `.concat()` method to concatenize arrays.

We can use the `.sort()` method to order the arrays alphabetically.

Here's a rough example:

```
var first = ['a', 'd', 'b'];
var second = ['c', 'f', 'e'];

var joined = first.concat(second);

var ordered = joined.sort();

console.log(ordered);
```

Run this, and it should return the following:

```
['a', 'b', 'c', 'd', 'e', 'f']
```

Cool, so that works.

Let's write some tests to make sure everything works as expected, and to explore some possible use cases.

Create a module

Tests

```
var test = require('tape');
var concatSort = require('./concat-sort');

/* a helper function to check if two arrays have the same contents */
function arraysEqual(first, second) {
  if(first.length !== second.length) {
    return false;
  }

  var firstLength = first.length;
  for (var i = 0; i < firstLength; i++) {
    if(first[i] !== second[i]) {
      return false;
    }
  }
  return true;
}


test('array test', function(t){
  t.plan(2);

  var firstCheck = [1, 2, 'a', 'b', 'c', 'd'];
  var sorted = concatSort([1, 2, 'c'], ['b', 'a', 'd']);

  t.ok(arraysEqual(firstCheck, sorted));

  var secondCheck = [6, 5, 4, 3, 2, 1];

  function reverser(a, b){
    return b - a;
  }

  var reversedSort = concatSort([2, 6, 3], [1, 4, 5], reverser);

  t.ok(arraysEqual(secondCheck, reversedSort));
});
```

When we run the tests using `node test.js`, we'll get an error saying `TypeError: object is not a function` because we haven't defined our concat-sort.js module file yet.

Implement the module

Create the skeleton of our module in the concat-sort.js file:

```
module.exports = function(){

}
```

Run the tests again and you'll see that the two tests are failing. Excellent. Now we can start filling it in based on previous experiments.

We want to be able to concat two arrays, so we need those two arguments.

In the second test we're also passing a callback function that is meant to provide an alternate sort behavior than the default alphanumeric sort.

This can work because the `.sort()` method accepts such a callback function.

Here's the working module:

```
module.exports = function(first, second, sorter){
  return first.concat(second).sort(sorter);
}
```

Usage

As seen in the tests, the module can be used like this for simple concatenation and alphanumeric sorting:

```
var concatSort = require('./concat-sort');

var sorted = concatSort([1, 2, 'c'], ['b', 'a', 'd']);
console.log(sorted)
```

In this example we want only the strings in the arrays, sorted in reverse order, and everything else should be excluded:

```
var concatSort = require('./concat-sort');

var firstArray = ['food', 'yum', function pizza(){}];
var secondArray = ['pizza', 'yeah', 3333333333];

function onlyStrings(value){
  return (typeof value === 'string') ? value : false;
}

function sorter(a, b){
  return b - a;
}

var sorted = concatSort(firstArray, secondArray, sorter);
var filtered = sorted.filter(onlyStrings)

console.log(filtered);
```

Objects

Problem

We need to iterate through all the properties of an object, excluding any methods on that object.

How we might solve it

```
var anObject = {
  aString: 'some string value',
  anInteger: 1234,
  aMethod: function(){
    return 'a function that belongs to an object';
  }
}

for (var key in anObject){
  console.log(anObject[key]);
}
```

This gives you an idea of how to iterate through an object using a for...in loop. Note that this will also log the method `aMethod` to the console in a useless and potentially problematic way. This approach works best with objects that do not have methods.

Running the above code will return:

```
some string value
1234
function (){
    return 'a function that belongs to an object';
  }
```

Ouch, returning that function definition in that way is going to mess stuff up.

You can, however, check if the any key on an object references a function using an if statement like this:

```
for (var key in anObject){
  if (typeof anObject[key] !== 'function'){
    console.log(anObject[key]);
  }
}
```

This will return:

```
some string value
1234
```

Nice, now that method on `anObject` isn't an issue.

Create a module

Let's create a simple module that we can reuse that will iterate through the properties of an object, but not any methods on an object.

Tests

```
var test = require('tape');
var eachKey = require('./each-object-key');

var anObject = {
  aString: 'some string value',
  anInteger: 1234,
  aMethod: function(){
    return 'a function that belongs to an object';
  }
}

test('test object keys', function(t){

  // test to make sure aMethod isn't included as one of the keys
  eachKey(anObject, function(key, value){
    t.notEqual(anObject.aMethod, value)
  });

  t.end();
});
```

Implement the module

Create a file named each-object-key.js, and add the following module code:

```
module.exports = function(obj, callback){
  for (var key in obj){
    if (typeof obj[key] !== 'function'){
      return callback(key, obj[key]);
    }
  }
}
```

Usage

To use the each-object-key module, do the following:

```
var eachKey = require('each-object-key');

var anObject = {
  aString: 'some string value',
  anInteger: 1234,
  aMethod: function(){
    return 'a function that belongs to an object';
  }
}

eachKey(anObject, function(key, value){
  console.log(key, value)
});
```

Part 3: Example applications

In this section of the book we'll explore extended examples that show usage patterns for the concepts, modules, and development tools described earlier in the book.

List of examples:

Using Backbone and jQuery with Browserify

Using browserify with modules from npm can be a little overwhelming at first, so what if we were able to use a couple of common front-end development libraries using along with browserify to ease the process of getting started?

This section is a work in progress

I'm currently working on new examples to include in this part of the book, and it would be wonderful to hear what types of examples you'd like to see! Please email suggestions to hi@learnjs.io.

Using Backbone and jQuery with Browserify

Using browserify with modules from npm can be a little overwhelming at first, so what if we were able to use a couple of common front-end development libraries using along with browserify to ease the process of getting started?

It's possible to build applications using backbone[48] and jquery[49] that are bundled by browserify[50], and in this post we'll take a look at the basics of how that works. We'll use a tool called beefy[51] as the development server.

To get started, create a directory for your project, change directory into it, and run `npm init` to create a package.json file for your project:

```
1 mkdir my-project
2 cd my-project
3 npm init
```

Answer the questions from the `npm init` prompt.

Install jquery and backbone:

```
npm install --save jquery backbone
```

Create a view module in a file named app-view.js:

[48] https://github.com/jashkenas/backbone
[49] https://github.com/jquery/jquery
[50] https://github.com/substack/node-browserify
[51] https://github.com/chrisdickinson/beefy

```
var Backbone = require('backbone');
var $ = require('jquery/dist/jquery')(window);
Backbone.$ = $;

module.exports = Backbone.View.extend({
  initialize: function(){
    console.log('wuuut')
    this.render();
  },

  render: function(){
    $('body').prepend('<p>wooooooooooooooo</p>');
  }
});
```

This should look familiar if you've used Backbone before, with the slight variation of exporting the view using module.exports.

There's also the weird require statement for jQuery:

```
var $ = require('jquery/dist/jquery')(window);
```

We have to use the path to the actual jQuery build rather than just pass the module name, and specify `window` so that jQuery actually uses the window object. Otherwise, we'd get an annoying error like this:

```
Uncaught Error: jQuery requires a window with a document
```

This require statement will likely get simpler in upcoming versions of jQuery.

Create an index.js file with this code to use the module:

```
    var AppView = require('./app-view')

    var appView = new AppView();
```

Install browserify and beefy to use to bundle the code and create a dev server:

```
npm install --save-dev browserify beefy
```

Add a `start` command and a 'bundle' command to your package.json scripts object, so that it looks like this:

```
"scripts": {
    "start": "beefy index.js:bundle.js --live",
    "bundle": "browserify index.js -o bundle.js"
  },
```

Now, you can run `npm start` and view the simple Backbone/jQuery app bundled by browserify at `http://localhost:9966`. Beefy will watch your files for changes, then regenerate and serve a bundle.js file and reload the browser window each time you save a file. Convenient!

And when you're ready to deploy your project, you can run `npm run bundle` to get an actual bundle.js file. Host this thing on GitHub Pages or wherever you like.

Example source code

See the full, operational code on github: github.com/learn-js/jquery-backbone-browserify-example[52].

[52] https://github.com/learn-js/jquery-backbone-browserify-example

Appendix

Additional resources

interactive terminal tutorials:

You should definitely visit nodeschool.io[53]. It features tutorials that are almost like adventure games, except for learning programming:

- Learn You The Node.js For Much Win![54] An intro to Node.js via a set of self-guided workshops.
- Stream Adventure[55]. Learn about streams in node.js
- Level Me Up Scotty![56] Learn about using leveldb with node.js

Online interactive tutorials:

- codecademy.com[57]
- Kahn Academy Computer Science[58]

javascript books:

- js for cats[59]
- eloquent javascript[60]
- learning javascript design patterns[61]
- writing modular javascript[62]
- jquery fundamentals[63]
- javascript enlightenment[64]

[53] http://nodeschool.io
[54] https://github.com/rvagg/learnyounode
[55] https://github.com/substack/stream-adventure
[56] https://github.com/rvagg/levelmeup
[57] http://codecademy.com
[58] https://www.khanacademy.org/cs
[59] https://github.com/maxogden/javascript-for-cats
[60] http://eloquentjavascript.net/
[61] http://www.addyosmani.com/resources/essentialjsdesignpatterns/book/
[62] http://addyosmani.com/writing-modular-js/
[63] http://jqfundamentals.com/
[64] http://www.javascriptenlightenment.com/JavaScript_Enlightenment.pdf

node.js books:

- art of node[65]
- stream handbook[66]
- node beginner book[67]

html/js/dom books:

- dive into html5[68]
- dom enlightenmnet[69]

Style guides:

- idiomatic js[70]
- idiomatic html[71]
- idiomatic css[72]
- airbnb js style guide[73]
- felixge node style guide[74]
- [jQuery's javascript style guide](http://contribute.jquery.org/style-guide/js/

Mozilla Documentation[75]

Have a question about some css property or html element? The Mozilla Developer Network has awesome documentation. If you're searching on google.com for anything css/html/js related, add the abbreviation "mdn" to your search query to see results from Mozilla Documentation. This site also has a bunch of introductory tutorials that are really useful.

WebPlatform.org[76]

This is a newer resource, but a good one. It's got a great design and well-organized resources.

[65] https://github.com/maxogden/art-of-node
[66] https://github.com/substack/stream-handbook
[67] http://www.nodebeginner.org/
[68] http://diveintohtml5.info/
[69] http://domenlightenment.com/
[70] https://github.com/rwldrn/idiomatic.js
[71] https://github.com/necolas/idiomatic-html
[72] https://github.com/necolas/idiomatic-css
[73] https://github.com/airbnb/javascript
[74] https://github.com/felixge/node-style-guide
[75] https://developer.mozilla.org/en-US/
[76] http://www.webplatform.org/

Javascript strings cheatsheet

String methods

Here are a collection of common and useful string methods that exist in Javascript.

.charAt()

Returns the character of a string at a specific index.

Usage example

```
var someString = 'pizza';

someString.charAt(1)
// returns: 'i'
```

.concat()

Join two or more strings together, and it returns a copy of the combined strings.

Usage example

```
var someString = 'pizza';
var anotherString = ' is awesome!';

someString.concat(anotherString);
// returns: 'pizza is awesome!'
```

.indexOf()

Find a string in another string, and it returns the index position of its first occurrence.

Usage example

```
var someString = 'pizza';

someString.indexOf('a');
// returns: 4
```

.lastIndexOf()

Find a string in another string, and it returns the index position of its last occurrence.

Usage example

```
var someString = 'pizza';

someString.lastIndexOf('z');
// returns: 3
```

.match()

Returns an array of all matches of a regular expression in a string.

Usage example

```
var someString = 'pizza is awesome and soda is awesome.';

someString.match(/is/g);
// returns: ['is', 'is']
```

.replace()

Find and replace a substring in a string with a new substring. You can use a regular expression in place of a substring for the first argument.

Usage example

```
var someString = 'pizza is awesome';

someString.replace(/awesome/, 'delicious');
// returns: 'pizza is delicious'
```

.search()

Very similar to .indexOf(), only it takes a regular expression as the argument, and returns the index of the substring's position in the string.

Usage example

```
var someString = 'pizza is awesome';

someString.search(/is/);
// returns: 6
```

.slice()

Specify start and end index positions, and .slice() will return the part of the string that exists within the start and end points.

Usage example

```
var someString = 'pizza is awesome and soda is awesome';

someString.slice(0, 5)
// returns: 'pizza'
```

.split()

Create an array from a string, split by using a separator you define.

Usage example

```
var someString = 'pizza is awesome';

someString.split(' ');
// returns: ['pizza', 'is', 'awesome']
```

.substr()

Extracts characters from a string, beginning at a specified index and through the specified number of characters.

Usage example

```
var someString = 'pizza is awesome';

someString.subst(0,5)
// returns: 'pizza'
```

.substring()

Specify two index positions, and .substring() will return the characters between those two indexes.

Usage example

```
var someString = 'pizza is awesome';;

someString.substring(9,16)
// returns: 'awesome'
```

.toLowerCase()

Convert a string of any case to all lowercase.

Usage example

```
var someString = 'PIZZA IS AWESOME';

someString.toLowerCase();
// returns: 'pizza is awesome'
```

.toUpperCase()

Converts a string to uppercase letters

Usage example

```
var someString = 'pizza is awesome';

someString.toUpperCase();
// returns: 'PIZZA IS AWESOME'
```

.trim()

Use .trim() to remove extra white space from both ends of a string.

Usage example

```
var someString = '  pizza is awesome ';

someString.trim();
// returns: 'pizza is awesome'
```

Changelog

v0.7.0 - February 3, 2014

- Start Part 3: examples
  - Add Using Backbone & jQuery with Browserify chapter
- Refactor chapters to make more sense.
- Many small typo fixes and improved explanations.
- fixes from suisea[77]

v0.6.0 - December 18, 2013

- Typo fixes
- Add cheatsheet and examples for string methods
- Expand and improve objects section
- Add back intro to git, github, grunt & package managers chapters
- Add link to GitHub repo

v0.5.0

- Major refactoring to turn the project into a series rather than one book.
- Removed sections to turn this book into an introduction to javascript.

v0.4.1

- started player and keyboard sections of node-rogue chapter
- fixed html/css stuff in node-rogue chapter
- add package managers comparison for npm, bower, component
- expand introduction to git & github
- expand intro to canvas
- start an intro to testing chapter
- start intro to terminal chapter
- start chapter about voxel.js
- add placeholders for main sections
- rearrange some existing chapters
- start websites section with pizza-fanpage project

[77] https://github.com/suisea

v0.4.0

- Add Introduction to npm
- Add Introduction to callbacks
- Add Introduction to canvas (still working on this)
- Add Introduction to browserify
- Substantial revisions to Chapter 01 - making a game
- Many small typo / formatting fixes

v0.3.2

- Substantial copy editing
- Start chapter 1 about making an rpg game
- Rearrange and edit Basics intro section

v0.3.1

- start intro to node section
- add contributors list
- add simple keyboard interaction example

v0.3.0

- start Basics section
- add intro to Chrome Developer Tools
- add intro to functions

v0.2.0:

- added introduction to grunt.js
- added introduction to git & GitHub
- small typo fixes

v0.1.0:

- first release
- intro to functions - create an add function
- appendix with initial style guide and additional resources