learn.js

A guidebook to building
projects with javascript.

SETH VINCENT

Learn.js: a guidebook to building projects with javascript.

This book is like a collection of LEGO guides. The goal: a fun way to learn javascript.

Seth Vincent

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

This version was published on 2013-05-28

This is a Leanpub book. Leanpub empowers authors and publishers with the Lean Publishing process. Lean Publishing is the act of publishing an in-progress ebook using lightweight tools and many iterations to get reader feedback, pivot until you have the right book and build traction once you do.

Contents

Introduction

Thank you.

You believed in this project enough to get involved early, and your support is making it possible. This release went out on May 23, 2013, and it was the first release of *Learn.js*. You're a serious early adopter, and that's awesome.

Because you got the book so early, you're in a unique position to guide the direction of the book. If there are particular libraries, development tools, or programming patterns that you'd like to see covered, please submit your thoughts at the learnjs issue queue on GitHub, or email me at hi@learnjs.io.

This book is highly inspired by the lean publishing model (read more about it[1]) proposed by Peter Armstrong, founder of leanpub.com. I'm releasing the book early to get feedback from you, dear readers, so that together we can make the best book about javascript tools and libraries possible.

This book is looking pretty sparse right now at v0.1.0, but I'll be releasing updates weekly up until the book is feature-complete at v1.0.0. Help me determine what it will mean for the book to be feature-complete by filling out a survey asking what development tools, libraries, and programming patterns you'd be most interested in seeing covered.

The goal: produce a book of roughly 200 pages by the end of July. You're a big part of making that goal real. Thank you.

Target audience

This book is meant for javascript beginners, but will be useful for anyone that wants to broaden their understanding of javascript. Maybe you've written with jQuery for years and want to get an introduction to server-side code, mapping, or data visualization. Or maybe you've created client-side applications without the use of any development tools like bower, browserify, or grunt, and you want to streamline your development process.

Learn.js will introduce you to a wide range of tools and libraries, and whether you're a beginner or intermediate programmer, you're likely to find helpful material.

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

Development tools

The development tools for javascript have changed rapidly in recent years, and the popularity of server-side javascript has matured tools available for client-side developers.

Learning development tools is one of the most difficult parts of starting to work with a programming language.

Which tools should I use? Are there best practices for workflows? How should I test code?

There are many possible answers to the above questions, and this book won't give all of them, but because it will be a living document that is revised based on changing patterns and practices in the javascript world, we'll be able to create a guide that remains relevant.

Development tools the book will cover:

- npm, the package manager for node.js, which is also often used for client-side javascript modules.
- bower and component, two package management tools for client-side javascript.
- browserify, a tool for bundling client side code as node-style modules.
- grunt, a build tool that can automate repetitive development tasks.
- Chrome developer tools.
- Git and GitHub. This book won't be a full guide to git, but it will help you improve as a git user.
- Vagrant and Virtualbox. Using these tools you can set up an instance of a linux operating system on your Mac or Windows machine and interact with it from the command line.
- Phonegap/Cordova. This project allows developers to create mobile apps for multiple platforms based on one js/html/css codebase.
- Hosting using Heroku and GitHub pages.
- And other tools. Let us know what you'd like to see covered[2].

Libraries

There are many different libraries available to javascript developers. It takes time just to learn how to find and use javascript libraries, how to tell which libraries will be effective and reliable, and when not to use a library.

We'll show you the best places to look for javascript libraries, the best resources for keeping track of new releases of libraries, and where to go when you need help learning a new library.

Learn.js will cover client-side libraries as well as node.js modules, and explore opportunities for using libraries that work on both the client and the server.

[2] http://hi@learnjs.io

Some of the javascript libraries the book will cover:

- jQuery.
- leaflet.js, mapbox.js, gmaps.js, etc.
- d3.js, chart.js, raphael.js, and vega.js.
- pixi.js, voxel.js, and processing.js.
- And other libraries. Let us know what you'd like to see covered[3].

Programming patterns

Javascript is a very flexible language that can be employed using a number of styles and patterns. In this book we'll take a look at some of the most popular patterns, and develop a simple, clean coding style based on popular open-source style guides already available in the community.

Some coding styles and programming patterns the book will cover:

- CommonJS and ECMAScript 6 modules.
- Functional programming.
- Prototypal inheritance.
- Constructors.
- And other patterns. Let us know what you'd like to see covered[4].

[3] http://hi@learnjs.io
[4] http://hi@learnjs.io

Introduction to git & GitHub

Developing web sites 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 that it can track every change you make to your code. This allows you to go back in versions if you mess something up, or figure out when errors were introduced to the code.

There are many other bonuses to using git, much of which is out of scope for this book.

The best way to get started learning git (and GitHub) is to visit try.github.com[5].

Get on GitHub

If you haven't already, create an account at github.com[6].

GitHub is a great place to host your code. Many employers hiring for developer and designer positions will ask for a GitHub profile, and use your activity there 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 â€“ they'll already know your technical abilities based on your open source contributions. That's a big win.

Anyway, GitHub is super useful for collaborating on projects with people, and for most open source communities has become the de facto code hosting service.

Create a site for yourself using GitHub

GitHub has a useful service called GitHub Pages[7] that allows you to host a simple site on their servers for free.

With GitHub Pages you can:

- design it any way you want by having complete control over the html, css, and javascript.

[5] http://try.github.com
[6] http://github.com
[7] http://pages.github.com

- use simple templates that are available 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[8] to learn about it in detail.

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

Introduction to grunt.js

Grunt is a tool for managing the javascript, css, and html files of your web project, 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 like 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:

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[9].

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[10].

**Download an installer from nodejs.org. **[Here's the node.js download page][nodejs.org/download].

Installing node gives us the node package manager `npm`. We'll use it to install grunt-cli, 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
```

[9] https://github.com/creationix/nvm

[10] https://github.com/joyent/node/wiki/Installing-Node.js-via-package-manager

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/getting-started](http://gruntjs.com/getting-star\
ted)
```

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-livereload][https://github.com/gruntjs/grunt-contrib-livereload]
- [grunt-regarde][https://github.com/yeoman/grunt-regarde]

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

```
{
  "name": "wicked-ooh-game",
  "version": "0.0.1", "author": "Super Big Tree <seth@superbigtree.com>",
  "description": "A silly game.",
  "repository": {
    "type": "git",
    "url": "https://github.com/your-profile/wicked-ooh-game.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-livereload': '~0.1.2',
      'grunt-regarde': '~0.1.1'
},
 "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 wicked-ooh-game
cd wicked-ooh-game
```

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.w\
riteln()
    // 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: wicked-ooh-game
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 use a project like yeoman for its code generators. That's up to you, but when first learning it's a reasonable idea to start from scratch and see how everything works.

CHAPTER 0

Write a function that adds numbers, and make it awesome.

Our goals for chapter zero are simple:

- Get familiar with javascript syntax and data types.
- Write a function that does one task really well.
- Try out the Chrome javascript console
- Learn about debugging and refactoring code.

We'll be writing a function that adds numbers. Simple. Almost too easy. But think about it as if you were building a calculator. You'll need functions that add, subtract, multiply, and divide, and depending on the complexity of the calculator, maybe even more than that.

Our goal will be to create an add function that could be reused in various ways throughout our imaginary calculator program. We'll only make `add`, but if you want to experiment and actually make a calculator, that would be awesome! Let me know how it goes[11].

Writing javascript on paper.

You want to write javascript? OK, here's what we'll do:

- Take out a sheet of paper and a pen.
- Write the following on the paper:

```
1  var x = 1;
2  var y = 2;
3  var z = x + y;
```

- Think about how easy this is and how it's awesome that you're coding with javascript on WHATEVER YOU WANT.

You know what happens when you write `x + y`.

You already know the value of `z`.

This looks just like math, and that's because it is.

Don't worry, writing code isn't all about math, but working with numbers is a big part of the work you'll be doing.

[11] mailto:hi@learnjs.io

If the above is simple math, what's with the word `var`? What's that?

The word `var` stands for variable. We use it only when creating a variable. You might remember the concept of variables from math, too. In this case, `x` is just a name that refers to the number `1`. Any time we want to create a variable in javascript, we write something like:

```
var nameOfTheVariable = 'whatever the variable should reference';
```

Why are there single quotes around the phrase `whatever the variable should reference`?

Check out the `var x = 1;` statement above. There aren't any quotes around the `1`. This is how we can tell numbers from text. In javascript (and most programming languages), text like this is called a **string**.

Wait, why did I just write that on paper?

Asking you to write code on paper wasn't arbitrary. It's useful to think about code in different ways, especially in the abstract. Sometimes it takes brainstorming away from the computer to figure out solutions to difficult problems. Another perk to paper coding: your program doesn't have to run, so you can use *pseudocode*.

So, what's pseudocode?

Here's a simple example:

```
if x is greater than 10, subtract 1 from x.
```

Here's that pseudocode turned into javascript:

```
if (x > 10){
  x = x - 1;
}
```

We won't go into `if` statements in detail here, but based on this pseudocode you should be able to tell what's happening in that javascript.

Pseudocode is awesome.

You can use pseudocode to experiment, and to define the outline of a program without worrying about errors or syntax details. It's a great way to start thinking about the structure of the code you're about to write.

Now let's open up the web browser, Chrome, and do some experimenting.

If you haven't installed Chrome yet, do so now at google.com/chrome[12]

1. Open a Chrome window.
2. Go to learnjs.io[13].
3. Click **View**, in the top menu, hover over **Developer**, then click **Javascript Console**. (TODO: edit for windows, linux, and mac)

   You can also use a keyboard shortcut: command+option+j on mac (TODO: edit for windows, linux, and mac)

If you've used a terminal program on your computer, the javascript console is similar, except you write javascript instead of terminal commands.

Important:

Any time there are code samples just type them straight into the javascript console, hit **Enter**, and see what happens. The best way to learn a programming language is to type it a lot. More than a lot.

In the console, type:

```
1 var x = 10;
```

Hit enter, and you'll see the word `undefined` pop up below your line of javascript. That's fine, it's normal. That's what the console returns when you enter such a statement. To see the value of `x`, type `x` into the console.

The console should return the number `10`!

Let's get functional.

Consider this pseudocode:

```
1 add two arbitrary numbers.
```

We're going to create a function in javascript that adds two numbers.

We've got a function named add, and it takes two arguments and adds them together. For now, we assume the arguments to be numbers.

To use the 'add' function, write something like this:

[12] http://google.com/chrome
[13] http://learnjs.io

```
add(3, 7);
```

Let's add with a function!

A function is a block of executable code, and when we give a function a name, like we do below, it can be used throughout your program. The benefit: define a function once rather than using similar blocks of code in multiple places in your program.

```
function add(x, y){
  return x + y;
}
```

Type the above `add` function into the javascript console, and use it to do some addition!

Important note: using Chrome's javascript console, if you hit **Enter** it will execute the code. To type a function like this onto multiple lines, hit **Shift + Enter** to add a line.

Just like with the variables we created earlier, when we first define a function in the console it will return `undefined` when you hit enter. This is normal.

Also: When you are in the javascript console you can hit the up arrow to revisit previous code that you've typed in.

Try these examples:

```
// add 2 and 4:
add(2, 4);

// create a variable named num and set its value to the sum of 3 and 4:
var num = add(3, 4);
```

Hey, you made `num` equal the usage of the add function.

Heck yeah, buddy. Let's take a look at the definition of the function again:

```
function add(x, y){
  return x + y;
}
```

Check out the middle line, `return x + y;`.

With most javascript programming, one of your goals should be to write small, simple functions that take arguments as input, modify that input, and output it using the return statement. Whatever a function returns is used elsewhere in your program.

When we make the variable `num` equal `add(3, 4)`, we're really setting `num` to equal the value that's returned from the add function. In this case, `add(3, 4)` will return the number `7`, so that's what `num` references.

Wait, what if I messed up the adder?

Hey, sorry to bother you, but somebody tried to use our add function like this:

```
add('1', '5');
```

Unsurprisingly, it didn't work as expected. Can you guess what the add function returned?

It returned this: `'15'`.

Why didn't it add?

Those numbers had quotes around them. Numbers in javascript do not have quotes around them, only strings. Instead of adding numbers, our program combined two strings.

Walking like a duck

Javascript is a dynamically-typed language. Remember how we created variables earlier that referenced specific values? Each value that a variable references has a type, and in javascript, you don't have to specify a type when you create a variable. You can change a variable's type later in the program, and you will interact with a variable in different ways depending on its type. Sometimes this is called duck typing. If the variable walks like a duck and talks like a duck, it is a duck.

Here are common types in javascript:

Boolean: true or false Example:

```
var thisIsNotTrue = false;
var thisIsNotFalse = true;
```

Note that there are no quotes around the words `true` or `false`. The variable named `thisIsNotTrue` references the value `false`, and the variable named `thisIsNotFalse` references the value `true`.

Number: integer or float Example:

```
var thisIsAnInteger = 123;
var thisIsAFloat = 3.14;
```

A float is a number with a decimal. An integer is a whole number – without a decimal.

String: text in quotes. Example:

```
var thisIsAString = 'you can tell because this text is inside of quotes';
```

A string is any text inside of quotes. It can be single or double quotes, but it has to be the same on each end. This will work: `'text'`, and this will work: `"text"`, but this will not work: `'text"`.

Back to our addition issue:

This usage of our function doesn't work: `add('1', '5');`.

The reason? The + operator and the way it works with strings.

This little buddy does a little more than you might expect at first. It'll add numbers together, but it'll also add strings together. When that happens it's called concatenation.

When we use the `add` function with `'1'` and `'5'` as arguments, it executes a statement like this:

```
return '1' + '5';
```

A string + a string equals those two strings combined. That's concatenation. So `'1' + '5'` becomes `'15'`.

Some examples:

```
'abc' + 'def' // becomes 'abcdef'
'This is a ' + 'string' // becomes 'This is a string'
'1' + '5' // becomes '15'
```

Lets add some code to our `add` function to make sure this doesn't happen. By converting the arguments of the function to numbers, we can allow for the possibility of adding numbers that happen to be strings!

```
function add(x, y) {
  return parseInt(x) + parseInt(y);
}
```

All we added was the usage of the `parseInt` function. This is a part of core javascript, and is used to convert strings and floats to integers.

Now we have a float issue.

Somebody tried to add some floats together, like this:

```
add(3.14, 7.28);
```

It did not work as expected. It returned `10`. That's because `parseInt` is only going to return the integer it finds in a string. So maybe `parseInt` isn't the best solution.

There is an easy fix. Revise your add function to look like this:

```
function add(x, y) {
  return parseFloat(x) + parseFloat(y);
}
```

Yep, that works. Now that we're using `parseFloat` instead of `parseInt`, decimal numbers stay intact. Now `add(3.14, 6.28);` returns `9.42`.

What about adding more than two numbers?

Somebody found a weird workaround for adding multiple numbers using this `add` function:

```
var a = add(1, 3);
var b = add(a, 7);
var c = add(b, 21);
```

That's just sad. We can make this easier. Rewrite the add function to look like this:

```
function add() {
  var total = 0;
  for (var i = 0; i < arguments.length; i++){
    total += parseFloat( arguments[i] );
  }
  return total;
}
```

Cool. Now we can pass any number of arguments to the `add` function and get the correct result! With this change we can now pass any number of arguments to `add`, like this:

```
add(1, 2, 3, 4, 5, 6, 7);
```

This makes `add` much more flexible than the previous workaround.

We used a `for` loop to cycle through all the arguments that are passed to `add`. We also used a variable named `total` to reference the sum of all arguments.

Looper: cycling through a function's arguments

For loops are easy once you get used to them.

First we set a start value: `var i = 0`. That sets `i` to `0`, which makes our loop start at `0`.

Then compare that to an end value: `i < arguments.length`. The statement `arguments.length` gives us the number of arguments. `length` returns the number of arguments. So, with usage like this: `add(1, 2, 3);`, `arguments.length` will return `3`.

As long as `i` is less than 3, the loop will run again.

Then we give the loop an increment value: `i++`. The `++` increases `i` by one with every loop.

This might be the point where you get a little overwhelmed if you are new to programming. If so, I recommend you practice things like `for` loops over at codecademy.com[14]. It's a great site for building foundational knowledge of javascript basics.

One last problem: 'one'.

That's not a number. Can you guess what happens when someone tries to do this:

```
add('one', 'two', 3);
```

Paste in the latest version of the `add` function to the Chrome javascript console. Then try out the above usage of `add`.

It should return something like this:

```
NaN
```

Let's fix that. The problem: `NaN` means **not a number**. When we run `parseFloat` on a string like `'one'`, it returns `NaN`, and when we try to add `NaN` to a number, the whole sum becomes `NaN`.

There are a few possible solutions to this problem. Here are two:

Ignore any argument that is not a number.

[14] http://codecademy.com

```
function add() {
  var total = 0;
  for (var i = 0; i < arguments.length; i++){
    if ( isNaN( parseFloat(arguments[i]) ) === false ) {
      total += parseFloat( arguments[i] );
    }
  }
  return total;
}
```

The new code that we added looks like this:

```
...
    if ( isNaN( parseFloat(arguments[i]) ) === false )  {
      ...
    }
...
```

The function `isNaN` checks to see if `parseFloat(arguments[i])` is a number. If `isNaN` returns false, then we know that the argument is a number.

This checks to see if an argument is a number, and if not, ignores it – but there's still a problem with this. Ignoring values like `'one'` in our `add` function because it doesn't contain numbers is reasonable. The problem: this error fails silently.

Any time there's a chance our function can fail because of misuse by yourself or another programmer, it's best for the code to send errors explaining why it fails. We want our code to complain when there is a problem.

Let's switch up our code so that if an argument is not a number, `add` throws an error, and if the argument *is* a number, it gets added to the `total` variable.

```
function add() {
  var total = 0;
  for (var i = 0; i < arguments.length; i++){
    if ( isNaN( parseFloat(arguments[i]) ) ) {
      throw new Error('Arguments to `add`  must be numbers.');
    } else {
      total += parseFloat( arguments[i] );
    }
  }
  return total;
}
```

Now, when we put that version of `add` into our javascript console and use it like this:

```
add('one', 2);
```

It'll return an error with this text:

```
Error: Arguments to `add` must be numbers.
```

Good, that was our goal. Now any strings that can't be converted to numbers will result in this clear error rather than being ignored.

Here is what we covered:

Chrome javascript console

We started using Chrome's javascript console in a really basic way as preparation for upcoming chapters.

Basic data types

We covered the basics of strings, numbers, and booleans.

Writing functions.

We went over the basics of writing a simple function.

Refactoring code.

As we wrote and experimented use cases of the `add` function we identified ways it could be improved and rewrote the function to guard against mistakes.

Debugging and useful errors.

As you're getting started it probably feels like errors are just something to avoid. But, if we can embed useful errors in our code in places where we know there are likely problems, that can make debugging much easier.

Hey, adding isn't really a useful function. Let's get serious.

OK, you're ready for the first project. Continue on to the first chapter, where we'll manipulate html elements on a page. We'll learn more about javascript syntax, data structures, and programming patterns to make a simple website: **a fanpage for pizza**!

Appendix

Javascript style guide & syntax cheatsheet

Variables

Creating a variable:

```
var nameOfVariable;
```

> Variables are camelCase, meaning first letter is lowercase, and if the variable is made of multiple words, the first letter of 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 good reason to use bracket notation.

For instance, it's more clear to use bracket notation in a situation like this:

```
for (var key in object){
  thisIsAnObject[key];
}
```

This gives you an idea of how to iterate through an object using a for...in loop.

Additional resources

javascript books:

- js for cats[15]
- eloquent javascript[16]
- learning javascript design patterns[17]
- writing modular javascript[18]
- jquery fundamentals[19]
- javascript enlightenment[20]

node.js books:

- art of node[21]
- stream handbook[22]
- node beginner book[23]

[15] https://github.com/maxogden/javascript-for-cats
[16] http://eloquentjavascript.net/
[17] http://www.addyosmani.com/resources/essentialjsdesignpatterns/book/
[18] http://addyosmani.com/writing-modular-js/
[19] http://jqfundamentals.com/
[20] http://www.javascriptenlightenment.com/JavaScript_Enlightenment.pdf
[21] https://github.com/maxogden/art-of-node
[22] https://github.com/substack/stream-handbook
[23] http://www.nodebeginner.org/

html/js/dom books:

- dive into html5[24]
- dom enlightenmnet[25]

Style guides:

- idiomatic.js[26]
- idiomatic html[27]
- idiomatic css[28]
- airbnb js style guide[29]
- felixge node style guide[30]
- jQuery's javascript style guide[31]

[24] http://diveintohtml5.info/
[25] http://domenlightenment.com/
[26] https://github.com/rwldrn/idiomatic.js
[27] https://github.com/necolas/idiomatic-html
[28] https://github.com/necolas/idiomatic-css
[29] https://github.com/airbnb/javascript
[30] https://github.com/felixge/node-style-guide
[31] http://contribute.jquery.org/style-guide/js/