Hack Reactor Precourse

Welcome to the community, we're happy to have you in Precourse! Precourse should guide you through most everything you need on your journey to day one of class.

Here you'll find a collection of materials all Hack Reactor students complete before their first day at the intensive. We recommend you progress through the material in a linear format; learning to program is an iterative process, and many topics build upon each other. **You will be held responsible for knowing the Precourse content and completing all exercises.**

Make sure you allocate enough time. The sum of our time estimates for this material is **50-70 hours**, so you **must** take this into account when planning out your time. Remember, **you will not be allowed to begin the course** if these assignments are not completed. Identifying potential stumbling blocks early on is key to you having a smooth runway up to the course.

If you anticipate that you will not be able to complete the material before your deadline, please reach out to your Admissions contact.

Introduction and Using this Precourse Work

Requirements

Our Precourse work comes with only a few requirements:

- **A Github account.** Please make sure to **update your full name and profile image**, as it will be used to identify you in our software. Please also **don't change your username** once you've sent it to us.
- **A code editor.** We recommend Sublime Text, an excellent editor which comes with all the major facilities you'd expect from a code editor (like syntax highlighting) yet provides a user-friendly coding environment free of distracting configuration. For that reason, it's the editor you'll use when pairing at Hack Reactor.
- **A web browser.** Specifically, Google Chrome. Other browsers will work fine, but Chrome's Developer Tools are the best on the block.
- **An installation of Node.js**. Node lets you run JavaScript outside of the browser and makes it easy to install some useful development tools. Get it here.
- **This document.** You **must** read and follow all of these instructions carefully. You will be expected to be completely familiar with this material on day one, as it **will not be revisited during the course.**
- **A laptop.** You will absolutely need a laptop to work on.
- **A closed container for beverages at our work stations.** In order to protect our hardware we only allow beverages in sealed containers while you are working at our work stations. Procure and clearly label a beverage container that seals *entirely* to use at our workstations.

Recommendations

- **A computer running Mac OS X.** The standard issue for Bay Area developers is an OS X-based machine. In keeping with our focus on teaching from an industry-standard toolset, we assume you're running OS X. You can complete the coursework on a Linux or Windows machine, but some instructions will either not apply to you or be different for your platform.

- LinkedIn, Hacker News, Stack Overflow and Quora accounts. These services will be helpful in various ways going forward, and you should spend some time poking around in each, getting a feel for the interface and the community.

NOTE: Do **not** make plans for the following three months after graduation, as you will want to spend that time job searching.

Expectations

Before entry into Hack Reactor, you are required to work though all of the following sections, except the ones explicitly marked "optional." You are expected to work through it largely on your own, with limited support, to enhance your ability to learn autonomously.

We work with the best and brightest at Hack Reactor. As such, most of our incoming class already has some solid base of experience, ranging from extensive personal research to an undergraduate degree in computer science. In exceptional circumstances, we accept remarkable candidates who have not already undertaken significant coding projects. Regardless of background, though, we expect all our students to have worked through significant amounts of programming fundamentals and to be well familiarized with JavaScript before their first day of class.

Similarly, we hold ourselves to a high standard of integrity, and expect incoming students to be honest and to abide by the following honor code:

Hack Reactor Student Honor Code

Do not read other peoples' solutions to problems. Since it is so crucial that you deeply understand each character you type, we ask that you not reference completed versions of assignments. Past experience has shown us that reading and copying solutions may make you feel as though you understand the solution (and that you would have come to the same conclusion given more effort), but will actually result in a weak understanding of the problem. Research how a solution works, but don't simply look for existing solutions to a problem.

Enrolled students: We track your understanding of problems through a series of *checkpoints*, but simply knowing the answers won't be enough. You need to pour your best efforts into the precourse content. It provides you with the foundation for the rest of your work on the intensive.

Do not post solutions to problems. You might think you're doing yourself and others a service, but please respect other students' discovery process by not tempting them to look at or copy your solutions.

Prerequisite Studies

At Hack Reactor, we don't spend much time on HTML or CSS--both involve lots of memorization and edge cases, which tend to be surmountable with experience and self-study. In a similar spirit, we don't spend much time on the very basics of JavaScript; instead, we dive straight into difficult-to-master topics like inheritance, and computer science fundamentals like data structures and algorithms.

In order to make sure everyone hits the ground running, we've added lots of information to Precourse that used to be covered during the first week of the intensive. We worked extremely hard to put together this material to be available to you before class starts, and we won't be covering any of it in class or in the lectures. **You must know it in detail** in order to participate effectively in this community and understand the rest of the class materials.

JavaScript Fundamentals

- DURATION: 2 - 6 hours

Your essential guide to the basics of JavaScript. These slides are targeted at explaining the details that trip most people up. Sometimes a student will fail to read a slide deck in sufficient detail before starting the class, and that has consistently been devastating for them. In general, all questions are welcome here, but when the answer was already available in this carefully curated resource, it's unfortunate for everyone.

- [Slides] Values and Types (see left hand side of Learn2 for slides)
- [Slides] Operators, Expressions, and Statements
- [Slides] Variables
- [Slides] Objects and Properties
- [Slides] Arrays
- [Slides] Functions
- [Slides] Parameters and Arguments
- [Slides] Passing Functions to Functions

A Beginner's Guide to HTML and CSS

- LINK: http://learn.shayhowe.com/html-css
- DURATION: 8-10 Hours

An excellent introduction to HTML and CSS, and more than enough knowledge to be dangerous. If you don't already have a base level of familiarity with HTML and CSS, or would generally like to brush up, work through this.

Before Moving On

Each chapter of the Precourse curriculum will include this "Before Moving On" ("BMO") section. Take enough time to review this list of expectations each time you get to it and do any additional work necessary to make sure you have the specified material firmly under your belt. It is reasonable to expect as you review the BMO list, that you will identify topics which you will need to revisit to ensure your understanding before continuing. For Javascript Fundamentals be certain to...

- Have a basic knowledge of HTML, including:
    - Basic HTML page structure, including the `html`, `head`, `scriptbody`, `div` `a`, and `p`
- Have a basic knowledge of CSS, including:
    - CSS rules (in general)
        - CSS selectors
        - Basic styling
- Have a basic knowledge of JavaScript, including:
    - A firm grasp on JavaScript syntax and fundamental programming concepts, e.g. how to write moderately complex functions
    - Basic data types: Objects, arrays, numbers, strings, booleans, `null`, and `undefined`
    - Control flow: `for` and `for..in` loops, `while`, `if`, `else`
- Understand that this introductory material will not be revisited during the course, and you must be completely comfortable with it on day one or you will not be able to participate.

Check Your Understanding

Throughout precourse and during your time as a student in the program, we will be providing you with a tool that allows you to determine how strong your understanding of the material is -- the checkpoint. You will find one at the end each section of this precourse curriculum. You must do each section's checkpoint before moving on to the next.

To get the most value from checkpoints, attempt them only when prompted by the coursework or instructors and "test" yourself by not looking back to your work or any slides to answer the questions. Challenge yourself to *reason through* code you see rather than *remembering* what you read or were told. The latter is an anti-pattern which very often causes students to slip behind their cohortmates who strive for understanding over recollection. When you discover areas that still remain weak, use that info to direct you back to review the related content. You are always free to reattempt checkpoints to see if your understanding has improved after some study.

Finally, checkpoints are not assessments. We do look at the results -- they help us determine where we need to improve our instruction and materials -- but they are intended as a tool for you to help you identify areas where you will benefit from doing more study.

Ready?

Start the first Checkpoint

Our Development Workflow

Like most skilled professions, it's difficult to jump into software engineering without becoming familiar with the tools you'll use to get the job done. The most crucial of these tools is *Git*, an industry-standard version control system and collaboration tool.

At this point, you might be wondering: ***What is version control, anyway?***

A version control system is a tool for tracking changes to your code. Think of it as the world's longest undo history: If you've ever broken a piece of your code and been unable to figure out how to go back to a working state, version control is what you've been looking for. Git is built from the ground up to help you avoid losing code, and also has powerful collaboration facilities built right in to help you collaborate with others on the same codebase.

In keeping with our focus on practicality, Hack Reactor makes Git a central part of its curriculum. Understanding how to use Git is critical not only to your success as a student, but as a software engineer.

Illustrated Git

- [Slides] Illustrated Git

10-30 Minutes

This is your complete introduction to Git as we use it at Hack Reactor, including the concepts of repos, clones, and forks. Make sure you know these slides front to back.

How to Use Terminal: The Basics

- LINK: http://mac.appstorm.net/how-to/utilities-how-to/how-to-use-terminal-the-basics
- DURATION: 15 Minutes

Git is a text command-based tool, and so before we can use it, we need to learn a little bit about the *Terminal*, the program we'll use to input Git commands. For now, we're going to learn only enough to get us started; later on we'll come back to the Terminal and learn about it in more depth.

GitImmersion: Labs 1-30

- LINK: http://gitimmersion.com
- DURATION: 3 Hours

Part one of a comprehensive Git walkthrough. It assumes no previous experience, which makes it a friendly introduction to Git.

GitHub - Forks and Pull Requests

- LINK: http://youtu.be/75_UrC2unv4
- DURATION: 10 Minutes

A quick video covering two of GitHub's most important features: forks and pull requests. These features are central to collaborating on open- and closed-source projects and are a main component of the workflow you'll use at Hack Reactor.

Hack Reactor's Git Workflow

- LINK: Git Workflow Overview
- DURATION: 10 Minutes

A brief primer covering the Git workflow you'll use at Hack Reactor.

Note: You will be using this workflow for all Precourse coding assignments, so be sure you know this.

For a fun and quick read on commit message best practices, check out Chris Beams's How to Write a Git Commit Message.

Before Moving On

- Know how to open Terminal and be comfortable using the following commands:
  - `cd`, `ls`, `mv`, `cp`, and `rm`
- Know what a version control system is and why you'd want to use it
- Know basic `git` commands, including:
  - `clone`, `push`, `pull`, `add`, `status`, `diff`, `commit`, and `remote`
- Know what a pull request is
- Know what a fork is
- Know how to submit your Precourse work to Hack Reactor's instructors for evaluation

Take the checkpoint for this section

Git Workflow Overview

The standard git workflow for projects at Hack Reactor goes like this:

- Fork the project. You'll find the project in Github, in the `hackreactor`org, with a repo name prefixed with your cohort's start date.
    - For example, `https://github.com/hackreactor/2019-10-git-workflow-overview`.
- Clone from the forked version on your Github account to your local workstation.
    - *(Make sure you're cloning your fork, not the original repository!)*
- Add a *remote* pointing to the original (upstream) repository. `git remote add upstream https://github.com/<USER_NAME>/<REPOSITORY_NAME>.git`
    - e.g. If you fork `hackreactor/2025-11-thunderbar`, you would run `git remote add upstream https://github.com/hackreactor/2025-11-thunderbar.git`
- The revision cycle:
    - `git status` to see that you're starting from a clean state.
        - Make sure you're on the `master` branch. Unless you're using an intermediate strategy of cutting feature branches, all your changes should be made on your fork's `master` branch.
    - Write the smallest possible change to your code that adds or improves something without breaking anything that used to work.
        - Should be about a 5-10 line change, taking 10-20 minutes. 15 lines at the max!
    - `git status` to see if you're right about which files have changed.
    - `git diff` to review your code, and verify that you're happy with all the things that have changed.
    - `git add` the files you want to commit.
    - `git status` to verify that all and only the files you intended to add were added.
    - `git commit` to commit your changes to the commit history
        - Do not use the `-a` or `-m` flags.

        - The `-a` flag will add **all** unchanged files in your directory to your commit. It's best to thoughtfully and intentionally add each file to your commit, as it encourages smaller, more focused commits.
        - The `-m` flag will destroy some potentially useful metadata on your commit. Using an editor to write your commit forces you to look twice at what's being committed, reducing errors and encouraging mindfulness. When practiced consistently, this results in a deeper understanding of your own workflow.
    - Write an informative commit message, describing what these 10 lines have improved.
    - `git status` to verify that the commit went as you expected.
    - `git push origin master` to send your changes back up to the forked repo on your Github account.
    - Repeat this cycle until your code is in the shape you ultimately want it.

In case you need more information about any of these steps, a detailed breakdown of each is included below.

Prerequisites

Before getting started here, there's a little background and prep work you'll need to do.

Knowledge Requirements

Before proceeding, you should be familiar with the basic idea behind and terminology of Git. If you don't yet know what terms like `clone`, `fork`, and `remote` mean in the context of Git, please work through the Git pre-coursework until these are familiar terms.

Technical Requirements

If you haven't already, you'll need to install Xcode and the Xcode Command Line Utilities. You can install both by following the directions here.

You'll also need to install Git. You can install Git using either the one-click installer available on Git's website.

Mac users: Alternatively (but highly recommended), you can install Git through Homebrew. Homebrew is a utility that will let you install many of the web development tools you'll use during the course.

You can use Homebrew to install crucial development tools like git, node, MongoDB, and JSHint (just to name a few).

Getting and Working on Hack Reactor Projects

All Hack Reactor projects are stored on Hack Reactor's GitHub account.

The project workflow has been designed to mirror (and prepare you for) the kind of industry-standard, professional workflow most development teams use.

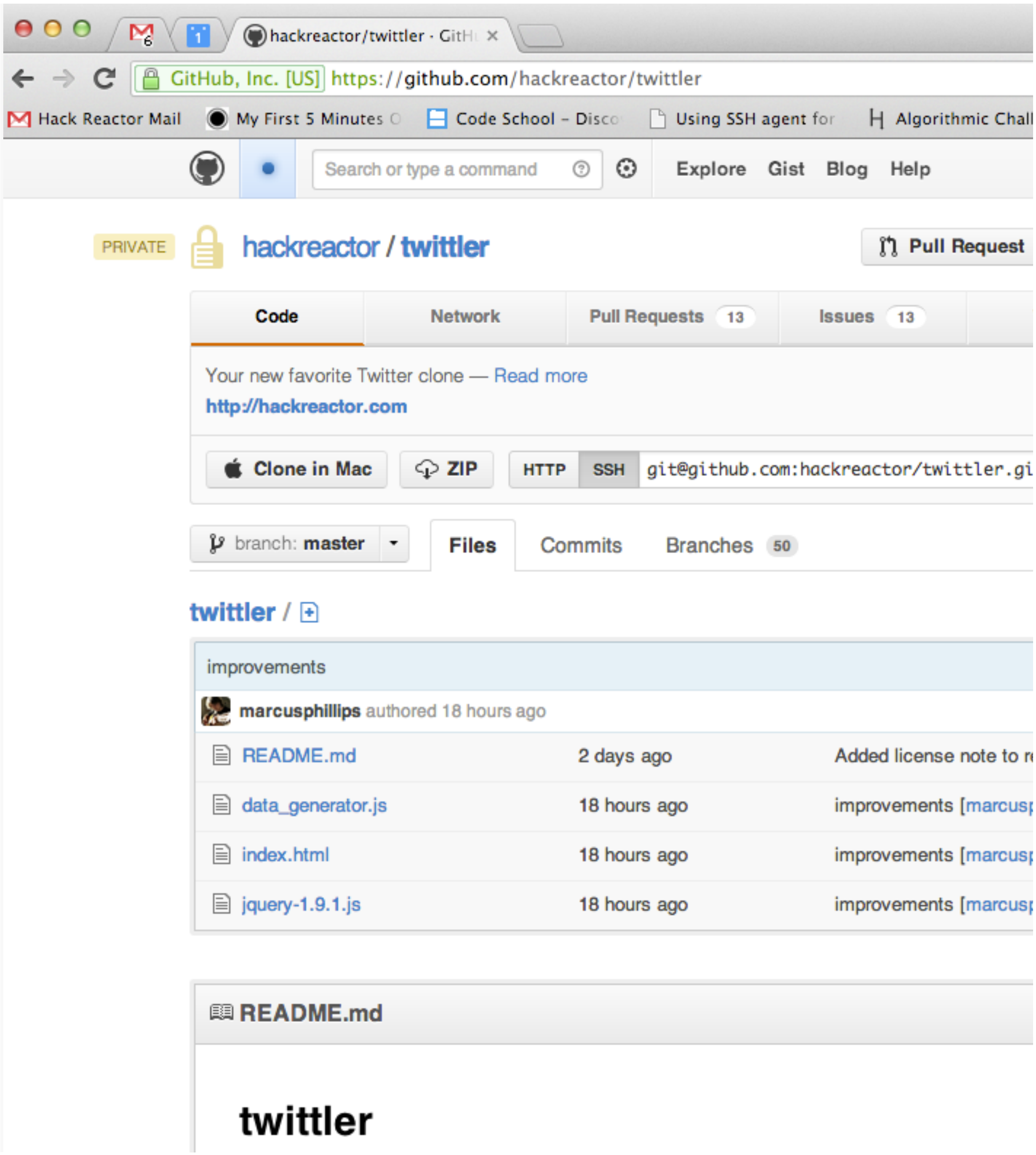
The basic idea is that you'll make a copy (a *fork*) of each repository you work on, and work through the project modifying only your copy. Among other benefits, this helps to prevent conflicts between changes made to the original repository and your version of the code. It'll also make showing your code off to potential employers dead simple, because all your code will be kept on your personal GitHub account.

The Workflow in Detail

The first thing you'll need to do is pull up the GitHub project you'd like to work on by navigating to the Hack Reactor GitHub account and selecting a project.

Forking the project

After opening a repo, you'll click the **Fork** button to make a copy of the repo. **This step is crucial**; you'll expereince pain later on if you work directly from the original repo rather than your fork.



Your new favorite Twitter clone — Read more
http://hackreactor.com

```
git@github.com:hackreactor/twittler.gi
```

improvements
marcusphillips authored 18 hours ago

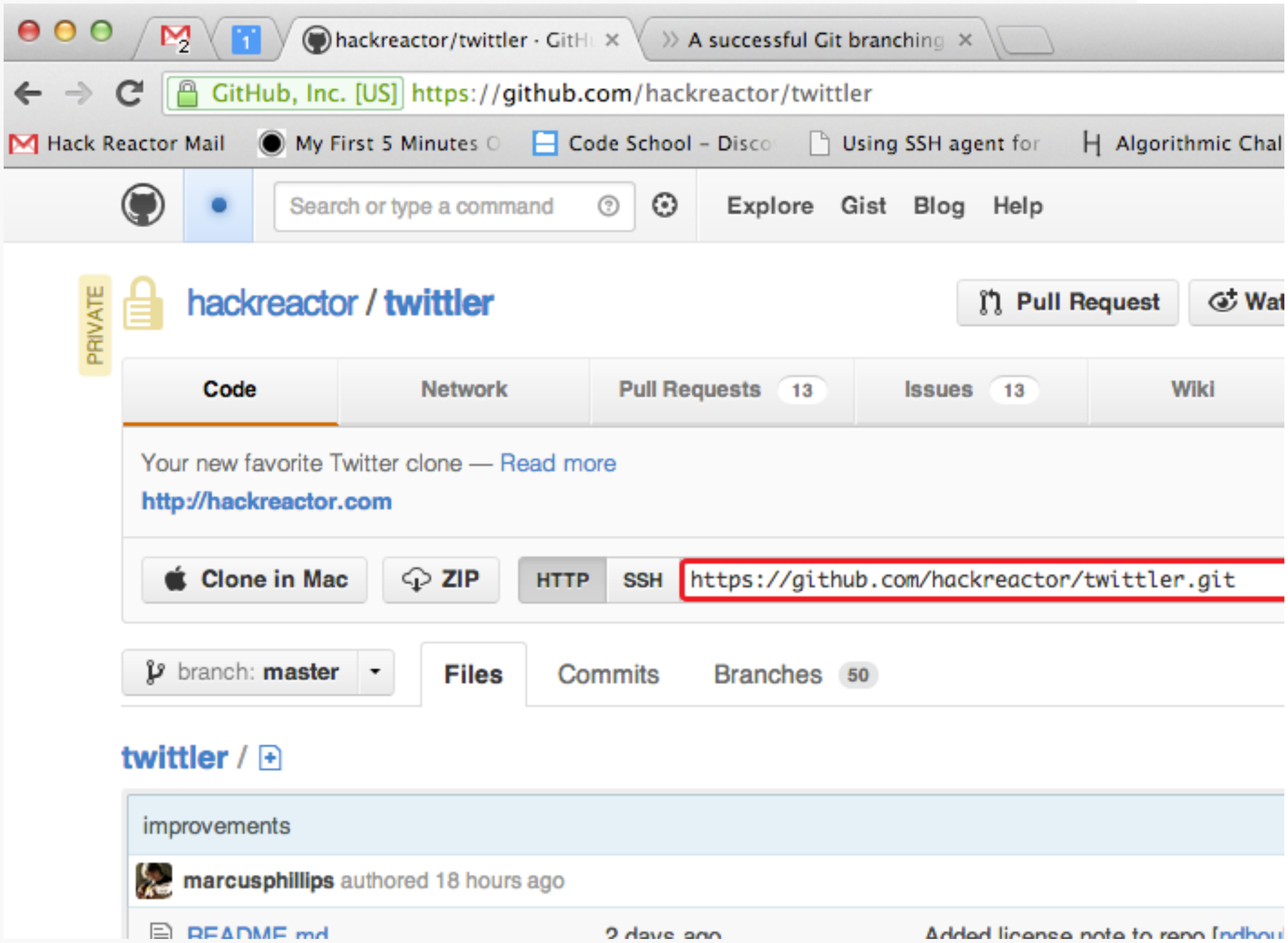
README.md	2 days ago	Added license note to r
data_generator.js	18 hours ago	improvements [marcusp
index.html	18 hours ago	improvements [marcusp
jquery-1.9.1.js	18 hours ago	improvements [marcusp

README.md

twittler

Cloning

Clone down the code from your forked repo on github to your own computer using `git clone`. You'll use the URL provided by GitHub as shown below:



For example, if your repo is located at `https://github.com/your_username/twittler.git`, you'd run `git clone https://github.com/your_username/twittler.git` at your Terminal.

Adding an upstream remote

When working from a fork, often times the original (upstream) repository's code will change. For a lot of reasons, you'll probably want a way to merge that new code into your fork. Problem is, Git doesn't know where to look to find this new code unless you give it the URL to a Git repository. This is called adding a `remote` (as in remote repository).

To add this for the twittler repo we cloned above, you would run `git remote add upstream https://github.com/hackreactor/twittler.git` in your Terminal. If necessary, you can now pull updated code from the original source repo into your local clone by running `git pull upstream master`.

Note: For now, don't worry about pulling down upstream changes. If changes have been made to the repo that you need to merge into your fork, a Hack Reactor staff member will let you know.

- The revision cycle:
    - `git status` to see that you're starting from a clean state.
        - Make sure you're on the `master` branch. Unless you're using an intermediate strategy of cuttign feature branches, all your changes should be made on your fork's `master` branch.
    - Write the smallest possible change to your code that adds or improves something without breaking anything that used to work.
        - Should be about a 5-10 line change, taking 10-20 minutes. 15 lines at the max!
    - `git status` to see if you're right about which files have changed.
    - `git diff` to review your code, and verify that you're happy with all the things that have changed.
    - `git add` the files you want to commit.
    - `git status` to verify that all and only the files you intended to add were added.
    - `git commit` to add your changes to a
        - Do not use the `-a` or `-m` flags. They almost certainly don't do what you think they do.
    - Write an informative commit message, describing what these 10 lines have improved.
    - `git status` to verify that the commit went as you expected.
    - `git push origin master` to send your changes back up to the forked repo on your Github account.

- Repeat this cycle until your code is in the shape you ultimately want it.

The revision cycle

See the list at the top of this page for blow-by-blow commands you'll want to use every time. Remember to commit changes to the `master` branch of your fork. Feel free to create additional, feature-specific branches if you'd like--just be sure to merge them back into your `master` branch at some point.

When you're all done writing code for a project, or when you'd like to share your code with someone, upload your code up to GitHub by running `git push origin master` at your Terminal.

Tips for Success

If you're new to Git, these quick tips may help you to be more productive and less intimidated by your new workflow.

Commit Often

It's good practice to do a `git commit` every time you implement a new, working part of your program, or when you fix something that doesn't work. Though it might seem like a pain at first, potential employers will see the frequency of your commit as a sort of gauge for how thoughtful of a developer you are. Developing good Git habits will also pay off in spades when you inevitably break your code and need to go back to a working state.

Don't Be Intimidated by Git

If you haven't used it before, Git will feel foreign at first. You're not alone! If you don't know how do do something related to Git, you're definitely not the first person to have had a given problem, and a quick Google search will usually turn up detailed instructions on how to resolve your problem. (Just for example, try Googling `git how do I get upstream changes`.)

While you're learning your way around Git, don't be afraid to experiment. Make a backup of your project folder (e.g. `cp -R twittler twittler.bak`) if you're about to execute a particulary unknown or scary Git command. Don't be afraid to mess up--everyone else has at some point.

Introduction to Automated Testing

The primary difficulty in software engineering is proving that code does what it is supposed to do, especially after many conflicting changes. Automated testing is a tool which is used widely in the industry to specify and improve code.

An automated test is just a special program that runs to verify that the main program is doing what you expect, and throws a noisy error otherwise. For example, if we've written a function `isDivisibleByTwo`, we could write a test function that checks that `4`, `50`, and `2^30` all return `true`, and `3`, `151`, and `2^30-1` all return `false`.

We've fully embraced automated testing as part of our core curriculum, and much of our coursework involves test suites which specify the behavior of the code you and your pair will be writing. Don't worry too much about writing your own tests, for now; you'll get your hands dirty with this during the course. This section will simply give you a first taste of test-assisted learning.

Testbuilder

- [Course] Testbuilder
- DURATION: 2-3 Hours

An interactive introduction to software testing and your first exposure to Mocha, a popular JavaScript testing suite.

JavaScript Koans

- [Course] JavaScript Koans
- DURATION: 1-2 Hours

A hands-on re-introduction to JavaScript syntax, and your first opportunity to work within a test suite. The Koans use Jasmine, a popular alternative which is largely interchangeable with Mocha.

Before Moving On

- Be comfortable with basic testing syntax such as `describe`, `it`, `expect`, `eql`, `equal`.
- Understand the purpose of an expectation or assertion, and what it means for a test to fail.
- Feel comfortable with being given a failing test, and making it pass.
- Ensure all tests are passing for the *Bare Minimum Requirements* for the Koans repo and that your work has been pushed to your fork on github

Take the checkpoint for this section.

Javascript Koans - koans to learn Javascript (the good bits)

Important: Before you begin, be sure you understand how to fork, clone, and push your work up to GitHub by reviewing the Git Workflow Overview in the previous chapter.

You can access the repo for this coding assignment on GitHub at https://github.com/hackreactor.

This repo contains a bunch of failing tests, which you can see by opening `KoansRunner.html` in a browser. The test files themselves live in `/koans`.

Bare Minimum Requirements:

- Make all tests in these files pass:
    - AboutExpects.js
    - AboutArrays.js
    - AboutFunctions.js
    - AboutObjects.js

Extra credit:

- Make the rest of the tests pass.

You shouldn't need to write any Jasmine code, but if you'd like to know more, check out their docs.

Resources for code quality:

- http://jsbeautifier.org/
- http://www.jshint.com/

This repo was mostly written by another dude. Read on for his original notes.

The Original Readme

Based on Edgecase's fantastic Ruby koans, the goal of the Javascript koans is to teach you Javascript programming through testing.

When you first run the koans, you'll be presented with a runtime error and a stack trace indicating where the error occurred. Your goal is to make the error go away. As you fix each error, you should learn something about the Javascript language and functional programming in general.

Your journey towards Javascript enlightenment starts in the `koans/AboutExpects.js` file. These koans will be very simple, so don't overthink them! As you progress through more koans, more and more Javascript syntax will be introduced which will allow you to solve more complicated problems and use more advanced techniques.

Running the Koans from a Browser

Simply navigate to the Javascript Koans folder using a file browser, and double click on `KoansRunnner.html`.

Any browser will do, but for the best results Firefox or Chrome is recommended. More stack trace information shows up for javascript on these browsers.

The first error will be in `koans/AboutExpects.js`. Fix the first test and refresh the browser. Rinse and repeat until all tests turn green.

The test runner used is Jasmine with a customized report viewer.

Changelog

- v3 - Nov 2010 - Moved out of branch of functional-koans project, into own top level project
- v2 - Sept 2010 - Second version based on jasmine (Thanks Greg Malcolm!)
- v1 - July 2010 - First version based on jsTestDriver

Inspirations & thanks

- Dick Wall (the Java posse) - for bringing the idea of koans to my attention
- Edgecase - for the great Ruby Koans
- Douglas Crockford - for Javascript; the good bits

License

This software is (c) 2010 David Laing & Greg Malcolm, and licensed under the MIT license (see LICENCE for details). Enjoy!

Advanced JavaScript, Part I

Guess what? You already know most of what there is to know about JavaScript. JavaScript is commonly seen as a fairly lightweight, minimalist language (for better and for worse). From a learner's perspective, having fewer things to learn before you can start building can be a great advantage. On the other hand, experienced developers have to rely more heavily on third-party libraries for functionality that would be built-in in other languages.

In this section, we'll ramp up the difficulty by jumping headfirst into re-implementing portions of Underscore.js, an influential and widely used library. In the process you'll write a significant amount of JavaScript, learn some of JavaScript's core functionality, and learn a little bit about *functional programming*.

Debugging (or, *Why Is Everything Broken?*)

Before we get started writing our library, we'll add a new, extremely useful skill to our toolkit: debugging.

Debugging code in the browser is quite convenient, since all modern browsers come with a powerful debugger built right in. The debugger gives you the ability to pause the execution of your program, inspect the values of variables in real time, and even run new code in a live context if necessary. Leveraging those tools to analyze and determine the cause of unexpected behavior is **the most essential programming skill you will ever learn.**

Google Chrome, our browser of choice, has a great set of development tools that we'll dive into.

Discover DevTools, Chapters 1-5

- LINK: http://discover-devtools.codeschool.com
- DURATION: 1-2 Hours

An interactive series co-produced by Google introducing the Chrome Developer Tools.

Debugging JavaScript

- LINK: https://developers.google.com/web/tools/chrome-devtools/
- DURATION: 1 Hour

Work through "Inspect and Debug Javascript > Run Snippets of Code From Any Page", "Inspect and Debug Javascript > Set Breakpoints", "Inspect and Debug Javascript > Step Through Code", and "Using the Console"

Underbar, Part I: `_.identity()` *through* `_.reduce()`

- [Course] Underbar
- DURATION: 7-9 Hours

A fun project that has you re-implementing powerful and useful helper functions from the Underscore.js library.

Before Moving On

- Start using the Chrome Developer Tools to debug your JavaScript code.
    - Be familiar with the following tools and start using them to debug your code:
        - `debugger` statements and breakpoints
        - Watch expressions
- Check your code and make sure that you're re-using functions whenever possible; for example, you should have re-used `each` inside of `map`
- Have a solid understanding of the following functions and what they do:
    - `_.each()`, `_.indexOf()`, `_.map()`, `_.reduce()`, `_.filter()`, `_.reject()`
- Be familiar with the rest of the Underbar functions you've implemented thus far

- Ensure all tests are passing for the *Bare Minimum Requirements* in part I of the Underbar repo and that your work has been pushed to your fork on github

Take the checkpoint for this section

Underbar

Important: Before you begin, be sure you understand how to fork, clone, and push your work up to GitHub by reviewing the Git Workflow Overview.

You can access the repo for this coding assignment on GitHub at https://github.com/hackreactor.

This project was written in the same spirit as JavaScript Koans, and thusly uses the Mocha Test Suite to facilitate a TDD approach to learning. It walks you through a reimplementation of underscore.js, a popular collection of useful functions authored by Jeremy Ashkenas.

Links and Resources

Some quick notes that may come in handy:

- As you work through these functions, you may sometimes have questions about what arguments they take, or about how they work (their *interface*). If the inline comments don't clarify these questions, it's a good idea to reference the official library's documentation.
- Many of the functions operate on "collections." They can take both arrays or objects as their arguments and you need to be able to handle both cases.
    - You can use `Array.isArray(obj)` to find out whether an object is an array.
    - You can use `obj.length` to test if something is either a string or an array.
- Javascript has a built-in `Math` object that provides some very useful functions. You can read up on them here.
- Within a function, you can use the `arguments` keyword to access all the parameters that were passed in--even if they aren't named in the function definition. This is useful if you don't know how many arguments are going to be passed in in advance.

    - You can count the arguments by using `arguments.length` and access each argument using `arguments[i]`.
    - The `arguments` object is very similar to an array, but note that it does not support most array functions (such as `slice` or `push`). You can read more about this here.
- If you have an array `myParameters` and would like to call a function`myFunction` using the elements in the array as parameters, you can use`myFunction.apply(context, myParameters)`. The first parameter, `context`, is the execution context for your function call. From inside `myFunction`, you can access it as `this`. For this exercise, you can just pass `null` for `context`. If you're curious, you can read more in the documentation for apply.

Goals

As is, the repository is missing code for most of the functions. It's your job to fix the library by implementing them. The functions are split in two sections, with a separate test suite for each.

The files in the `spec` directory contain the test suites. Your goal is to get all the tests to pass by implementing the missing functions. Run all the tests by opening `SpecRunner.html` in your browser.

The file `src/underbar.js` contains function definitions and explanations for the following functions (*italicized functions* are solved for you). Implement each of the functions by making all of the tests pass:

Part I:

- identity
- *first*
- last
- each
- *indexOf*
- filter
- reject
- uniq

- map
- *pluck*
- reduce

Part II:

- *contains*
- every
- some
- extend
- defaults
- *once*
- memoize
- delay
- shuffle

Extra Credit:

- invoke
- sortBy
- zip
- flatten
- intersection
- difference
- throttle

Note: Some browsers provide built-in functions--including `forEach`, `map`,`reduce` and `filter`--that replicate the functionality of some of the functions you will implement. Don't use them to implement your functions.

Throttle

There's one function that is a little more complicated, so we've included some more context to help you along.

_.throttle(func, wait): Wrap a function `func` so that it can be called at most once within a period of `wait` milliseconds. This is useful for throttling access to expensive APIs or to drawing routines in a video game. Let's see how it's used:

```
var counter = 0;
var increment = function() {
  return counter += 1;
};

// Create a function called throttledIncrement. This function can be called at
// most once every 100ms
var throttledIncrement = _.throttle(increment, 100);

throttledIncrement(); // return 1; `counter` should now be 1
throttledIncrement(); // return 1; schedule `increment()` call in 100ms
throttledIncrement(); // return 1; should do nothing

// Wait 100 ms; `increment` is called
```

- Arguments passed to the throttled function should be passed to the original function.
- The throttled function should always return the most recently returned value of the original function.
- If the wait period is 100ms and the function was last called 30ms ago, another call to the throttled function should schedule a call for 0ms after the wait period is over.

Extra Extra Credit

- Download the real underscore.js implementation and test suite, and try to understand how it works. A great way to do this is to break parts of the code and see which tests fail.
- Compare your implementations to the ones in the real library. Notice that this assignment has stripped out some complexity from

the original library; notice where these changes have been made, and try to understand what edge cases the original library is handling that your functions aren't.

- Notice that the real underscore.js uses an object named `breaker`. Look through the source and understand what this is doing, and how it optimizes some of the functions.

Advanced JavaScript, Part II

In this section, we'll move onto some more advanced functions from [Underscore.js][]. This section has a particular focus on reusing the functions you've already written, and will cover a few concept you may not have encountered before, *scopes* and *closures*.

Work through this section carefully. A concrete understanding of closures and scoping rules is one of the few things about this simple language that separate JavaScript novices from competent programmers. The rules are deceptively simple, so pay attention!

Scopes Module

- [Slides] Scopes
- [Slides] Execution Contexts
- [Slides] Contexts vs Objects

Closures Module

- [Slides] Closures

1-2 Hours

A quick introduction to how scopes and closures work. Closures are an extremely important part of JavaScript; understanding what they are will help you see how `_.memoize()` can be implemented.

Underbar, Part II: `_.contains()` *through* `_.shuffle()`

- [Course] Underbar
- DURATION: 10-12 Hours

Part two of implementing parts of the Underscore.js library.

Before Moving On

- Have a solid understanding of the following functions, what they do, and how they are implemented:

    - `_.extend()`, `_.defaults()`, `_.once()`, `_.memoize()`, `_.delay()`
- Be familiar with the rest of the Underbar functions
- Have a basic understanding of closures, lexical scope, and know what it means for a variable to be "in scope"
- Be familiar with how you used `.call` and `.apply` in your solutions. Don't worry if you don't yet understand what `this` or `null` means when passed in as the first argument to them.
- Ensure all tests are passing for the *Bare Minimum Requirements* in part II of the Underbar repo and that your work has been pushed to your fork on github

Take the checkpoint for this section

Underbar

Important: Before you begin, be sure you understand how to fork, clone, and push your work up to GitHub by reviewing the Git Workflow Overview.

You can access the repo for this coding assignment on GitHub at https://github.com/hackreactor.

This project was written in the same spirit as JavaScript Koans, and thusly uses the Mocha Test Suite to facilitate a TDD approach to learning. It walks you through a reimplementation of underscore.js, a popular collection of useful functions authored by Jeremy Ashkenas.

Links and Resources

Some quick notes that may come in handy:

- As you work through these functions, you may sometimes have questions about what arguments they take, or about how they work (their *interface*). If the inline comments don't clarify these questions, it's a good idea to reference the official library's documentation.
- Many of the functions operate on "collections." They can take both arrays or objects as their arguments and you need to be able to handle both cases.
  - You can use `Array.isArray(obj)` to find out whether an object is an array.
  - You can use `obj.length` to test if something is either a string or an array.
- Javascript has a built-in `Math` object that provides some very useful functions. You can read up on them here.
- Within a function, you can use the `arguments` keyword to access all the parameters that were passed in--even if they aren't named in the function definition. This is useful if you don't know how many arguments are going to be passed in in advance.

    - You can count the arguments by using `arguments.length` and access each argument using `arguments[i]`.
    - The `arguments` object is very similar to an array, but note that it does not support most array functions (such as `slice` or `push`). You can read more about this here.
- If you have an array `myParameters` and would like to call a function`myFunction` using the elements in the array as parameters, you can use`myFunction.apply(context, myParameters)`. The first parameter, `context`, is the execution context for your function call. From inside `myFunction`, you can access it as `this`. For this exercise, you can just pass `null` for `context`. If you're curious, you can read more in the documentation for apply.

Goals

As is, the repository is missing code for most of the functions. It's your job to fix the library by implementing them. The functions are split in two sections, with a separate test suite for each.

The files in the `spec` directory contain the test suites. Your goal is to get all the tests to pass by implementing the missing functions. Run all the tests by opening `SpecRunner.html` in your browser.

The file `src/underbar.js` contains function definitions and explanations for the following functions (*italicized functions* are solved for you). Implement each of the functions by making all of the tests pass:

Part I:

- identity
- *first*
- last
- each
- *indexOf*
- filter
- reject
- uniq

- map
- *pluck*
- reduce

Part II:

- *contains*
- every
- some
- extend
- defaults
- *once*
- memoize
- delay
- shuffle

Extra Credit:

- invoke
- sortBy
- zip
- flatten
- intersection
- difference
- throttle

Note: Some browsers provide built-in functions--including `forEach`, `map`,`reduce` and `filter`--that replicate the functionality of some of the functions you will implement. Don't use them to implement your functions.

Throttle

There's one function that is a little more complicated, so we've included some more context to help you along.

_.throttle(func, wait): Wrap a function `func` so that it can be called at most once within a period of `wait` milliseconds. This is useful for throttling access to expensive APIs or to drawing routines in a video game. Let's see how it's used:

```
var counter = 0;
var increment = function() {
  return counter += 1;
};

// Create a function called throttledIncrement. This function can be called at
// most once every 100ms
var throttledIncrement = _.throttle(increment, 100);

throttledIncrement(); // return 1; `counter` should now be 1
throttledIncrement(); // return 1; schedule `increment()` call in 100ms
throttledIncrement(); // return 1; should do nothing

// Wait 100 ms; `increment` is called
```

- Arguments passed to the throttled function should be passed to the original function.
- The throttled function should always return the most recently returned value of the original function.
- If the wait period is 100ms and the function was last called 30ms ago, another call to the throttled function should schedule a call for 0ms after the wait period is over.

Extra Extra Credit

- Download the real underscore.js implementation and test suite, and try to understand how it works. A great way to do this is to break parts of the code and see which tests fail.
- Compare your implementations to the ones in the real library. Notice that this assignment has stripped out some complexity from

the original library; notice where these changes have been made, and try to understand what edge cases the original library is handling that your functions aren't.

- Notice that the real underscore.js uses an object named `breaker`. Look through the source and understand what this is doing, and how it optimizes some of the functions.

Front-end Development

In the browser, modern web apps are composed of three major components:

- HTML, which defines the hierarchical structure and content of the page
- CSS, which defines the visual layout and style of the page
- JavaScript, which defines interactive behaviors and changes to the page over time

In this section, you'll learn about and practice using all three. By the time you're done, you'll have built an interactive site that responds to and changes depending on user input.

HTML and CSS

CSS Cross-Country, Parts 1-4

- LINK: http://www.codeschool.com/courses/css-cross-country
- DURATION: 4 Hours

A great introduction to some of the more advanced CSS techniques you'll use every day in your apps.

jQuery: Manipulating HTML Using JavaScript

You may have already discovered that web browsers come with a built-in JavaScript API for manipulating an HTML page's layout. This is huge-- the ability to control a page's content using JavaScript means that we can dynamically modify what users see based on input, requests to servers, and provide all sorts of other useful facilities.

But if you've tried to use the browser's API, you may have noticed that it's complex, sometimes unintuitive, and -- worst of all -- its behavior and feature set vary wildly across browsers.

Enter jQuery!

jQuery is a library that offers significant improvements on top of the native browser API, making manipulating your documents far easier.

Let's have a look at a simple example: We'll find all the `li` elements on the page and attach an event to them so that when clicked on, an `alert` window will pop up:

```
// A function that we want to be called whenever an `li` element is clicked
var clickHandler = function() {
  alert('You clicked on me!');
};

// The verbose way of doing this without jQuery, which may not even work across
// all browsers...
var items = document.getElementsByTagName('li');
for (var i = 0; i < items.length; i++) {
  // Check for < IE9 way of attaching click handlers
  if (items[0].attachEvent) {
    items[0].attachEvent('onclick', clickHandler);
  } else {
    items[i].addEventListener('click', clickHandler, false);
  }
}

// The jQuery way of doing things--succinct, and works in virtually any browser!
$('li').on('click', clickHandler);
```

As your web applications gets more and more complex, jQuery will help you to keep your code shorter, more readable, and more portable across browsers.

Try jQuery

- LINK: http://try.jquery.com
- DURATION: 1-2 Hours

A guided, practical introduction to jQuery, the most widely used DOM manipulation library.

Twittler

- [Course] Twittler
- DURATION: 4-6 Hours

Build a Twitter clone using the power of JavaScript and HTML/CSS! A great opportunity to practice your newfound jQuery skills by building an interactive web application.

Before Moving On

- Understand how to dynamically generate HTML and append it to a page using jQuery
- Understand what events are and how to attach them to a DOM/jQuery element
- Don't worry about CSS too much. It is famously arcane, and even expert developers need a reference often.
- Ensure all tests are passing for the *Bare Minimum Requirements* for the Twittler repo and that your work has been pushed to your fork on github

Take the checkpoint for this section

Twittler

Important: Before you begin, be sure you understand how to fork, clone, and push your work up to GitHub by reviewing the Git Workflow Overview.

You can access the repo for this coding assignment on GitHub at https://github.com/hackreactor.

What's Already Here

This is a mostly-empty repo, with a file that creates some data that represents twitter users and their tweets. It's the data you would expect to see if you had created a twitter account and followed a few people. (More tweets appear over time.)

That file is called `data_generator.js`. You don't need to understand the code that's in it, but here's what it does:

- Creates two global variables, `users` and `streams`.
  - `users` is an array of strings -- all the usernames that you're following.
  - `streams` is an object with two properties, `users` and `home`.
    - `streams.home` is an array of all tweets from all the users you're following.
    - `streams.users` is an object with properties for each user.`streams.users.shawndrost` has all of `shawndrost`'s tweets.
- Kicks off a periodic process that puts more data in `streams`.

You'll mostly be working in the javascript block of `index.html`. Note: The generated tweets will be displayed in reverse chronological order.

Bare Minimum Requirements

- Show the user new tweets somehow. (You can show them automatically as they're created, or create a button that displays new tweets.)

- Display the timestamps of when the tweets were created. This timestamp should reflect the *actual* time the tweets were created, and should not just be hardcoded.
- Design your interface so that you want to look at and use the product you're making.
- Allow the user to click on a username to see that user's timeline.

Advanced

- Show when the tweets were created in a human-friendly way (eg "10 minutes ago"). You'll want to find and use a library for this.
- Allow the user to tweet. (This is going to require you to understand a little more about `data_generator.js`, but you shouldn't need to modify anything.)

Introduction to Computer Science

Code Reuse and Object Instantiation

This section introduces you to different patterns you can use for code reuse. You'll learn about some important concepts like inheritance and classes, as well as different patterns for instantiating objects. These slide decks cover a lot of ground, so make sure to read them carefully.

The Parameter This

- [Slides] Misconceptions of This
- [Slides] The Parameter This

Code Reuse I

- [Slides] Decorators
- [Slides] Functional Classes

Prototype Chains

- [Slides] Prototype Chains

Code Reuse II

- [Slides] Prototypal Class
- [Slides] Pseudoclassical Classes

Recursion

A recursive function is a function that calls itself. One simple example of recursion is this function that calculates a number's factorial:

```
var factorial = function(number) {
  // If the number is negative, it doesn't have a factorial. Return an
  // impossible value to indicate this.
  if (number < 0) {
    return -1;
  }

  // If the number is zero, its factorial is one.
  if (number === 0) {
    return 1;
  }

  // If the number is neither illegal nor zero, call factorial again,
  // this time passing in a smaller number. Eventually we'll reach 0,
  // causing each call to return to its caller and the recursion terminates.
  return (number * factorial(number - 1));
}
```

All recursive functions have at least one *base case*, or a case when the function can produce a result without recursing. In the factorial example above, `number === 0` is a great example of this: If `number` is `0`, we know the solution must be `1` and thus we don't need to recurse to solve the problem.

They also have at least one *recursive case*, which calls the function itself. The recursive case encapsulates the self-referential aspect of the algorithm; in this example, the final `return` statement elegantly represents the definition that `n! = n * (n-1)!` for all positive values of `n`.

Recursion in JavaScript

- LINK: http://www.codecademy.com/courses/javascript-lesson-205
- DURATION: 1-2 Hours

A gentle introduction to what recursion is and why you might want to use it. (If you're already familiar with recursion, feel free to skip this.)

Recursion

- [Course] Recursion
- DURATION: 6-8 Hours

An exercise that has you re-implementing native JavaScript browser methods using recursion.

Before Moving On

- Understand what recursion is, in general
- Understand base cases, and what it means for recursion to terminate
- Spend some time thinking about what sort of problems recursion might be a good fit for
- Ensure all tests are passing for the *Bare Minimum Requirements* in the Recursion repo and that your work has been pushed to your fork on github

Take the checkpoint for this section

Recursion

Important: Before you begin, be sure you understand how to fork, clone, and push your work up to GitHub by reviewing the Git Workflow Overview.

You can access the repo for this coding assignment on GitHub at https://github.com/hackreactor.

Recursion is a technique for solving problems wherein a function makes calls to itself. By doing so, it can complete a small amount of the processing, and delegate the rest of the problem to the recursive calls.

Consider the following function:

```
var eat = function(meal){
  console.log('meal before bite:', meal);
  console.log('now eating', meal.pop());
  if(meal.length){
    eat(meal);
  } else {
    console.log('done with the meal!');
  }
}
```

Which produces this output:

```
eat(['soup', 'potatoes', 'fish']);
// => meal before bite: ["soup", "potatoes", "fish"]
// => now eating fish
// => meal before bite: ["soup", "potatoes"]
// => now eating potatoes
// => meal before bite: ["soup"]
// => now eating soup
// => done with the meal!
```

You can use recursion on problems where smaller parts of the problem look the same as the larger problem as a whole.

In this sprint, you'll be reimplementing parts of the browser that involve recursion. In so doing, don't use the things you're reimplementing, or any other built-in shortcuts that make these problems trivial. (You'll probably know if you're cheating, but feel free to ask us if you're not sure.)

- Note: The tests in this repo are written using Mocha, a JavaScript testing framework.

(Curious fact: many browsers don't have any of these functions in them, and people do need to reimplement them. When we reimplement new browser functionality in older browsers, it's called a "polyfill".)

If you are in Precourse, please skip the entire 'Install Pomander' section

Install Pomander

In Terminal, run the following command from within this repository:

```
curl -s https://raw.githubusercontent.com/hackreactor-
labs/pomander/master/bin/install | bash
```

Pomander will check your code for syntax errors and violations against the style guide before each commit.

It uses a pre-commit hook to run staged files through `eslint` before each commit. `eslint` is a linter that will block your commit should you have any syntax errors, or, should you violate the Hack Reactor style guide. There are some preferred whitespace style rules that will give warnings but not block your commit. Your work will be of a higher quality if you follow the instructions of the linter. That said, if the linter gives you any funny bugs, these bugs are not intentional, and you should feel free to skip using it during commits with the `--no-verify` option.

Bare minimum Requirements

- Replace `stringifyJSON` with your own function in `src/stringifyJSON.js`, and make the specs pass.
- Implement `getElementsByClassName` with your own function in `src/getElementsByClassName.js`, and make the specs pass.
    - You should use `document.body`, `element.childNodes`, and `element.classList`

Advanced Content

Our advanced content is intended to throw you in over your head, requiring you to solve problems with very little support or oversight, much like you would as a mid or senior level engineer. The following problem is no exception, and you may have to do a fair amount of work to get enough context to get started on the problem itself.

- Replace `parseJSON` with your own function in `src/parseJSON.js`, and make the specs pass.
    - Use a recursive descent parser.
    - Resources:
    - One of Hack Reactor's *amazing* graduates, Ron Fenolio, wrote a fabulous blog post after wrestling with this very prompt
    - Note: This is a lot of work, and you should expect (and look forward) to bumping up against several conceptual hurdles

Required Class Preparation

If you feel solid on all of the material we've covered so far, you're in great shape, but you're not quite done yet. A week or so before your start date, make sure to complete these last few **required steps**:

Etiquette - VERY IMPORTANT

Understanding the rules of the road is crucial to keeping the class running smoothly, and we don't want to waste your time going over those rules in class. Please make sure you're completely clear on every aspect of the *Etiquette* page (see left hand side of Learn2). We're all going to be working closely together in the same building for many, many hours, so you will be quizzed on this important material. **We take this quite seriously, and we ask you to as well.**

Take the Etiquette checkpoint

Double check your GitHub

Confirm that your GitHub profile has your correct email address, full name, and a picture of your face, or else you will show up oddly in our software and everyone will laugh. Make sure the GitHub username you give us is the one you want to use for the duration of the course, because we can't accommodate changing it in our system. Note you **will need a gmail address** if we don't already have one on file for you.

NO Early arrival

You **will not** be able to work out of the space before your first day of class. In particular, the week before your class starts, the Operations Team will be busy preparing the space for your class's arrival. They need students to be out of the way in order to make the space as awesome as they do! **Please don't** drop by the week before your start date.

Assessments

Be ready to take an assessment upon arrival to demonstrate your mastery on the precourse material. The pace of this course is breakneck and your understanding of the precourse content is essential to your being able to proceed successfully with the course. If you have not been able to develop expertise with the precourse material it may be a sign that that the pace and difficulty of the course is not such a good fit for you.

Week Before Checklist

One week or so before your first day of class, make sure you do all of the following:

- **Reread the Fundamentals Slides.** Make sure you've read and reread the javascript fundamental slideshows, and Illustrated Git and are ready to demonstrate your mastery of the content.
- **Reread the slides in the "Advanced JavaScript, Part II" and "Introduction to Computer Science" slides.** It's critical you arrive on the first day with a firm understanding of the content in these slides.
- **Watch the Recursion and Underbar solution videos**. The following videos are password protected and *we will email you a password to access them a week before the course begins*. Use them to support your understanding of the Underbar and Recursion assignments you have already completed, and make certain you feel like an expert on the assignments before arriving on the first day.

  Take the Underbar and Recursion checkpoint

- **Complete the Precourse curriculum.** Completion is a requirement for entering on day one; if you have not completed the precourse work, you will not be permitted to begin class on day one. Reach out to us if you have any concerns about this.
- **Reread the etiquette page.** Make sure you know information on the *Etiquette* page to a pop-quizzable level. Spending time on the etiquette content once class starts is a terrible use of time and we'd rather you spend that time learning. Proper etiquette is also essential, so be certain to understand it well *before class starts*.
- **Take the survey** that the admissions team sends to you via email.

- Make sure you know how to get here!
- **Plan to arrive early -** Don't be late! On the first day, starting at 8:30 we will be taking photos of you to use for our internal records. Doors will open at 8:25 so be there on time and ready to have a photo taken.

That's it. We can't wait to see you!

The staff and students of Hack Reactor treat this beautiful space like it's our own--after all, it is! Please become very familiar with the etiquette of the space and of the class in general, as you'll be expected to follow all of these guidelines. We used to go over these "rules of the road" in a lecture during Week 1, but in an effort to conserve our precious in-class time, we now insist that all students read it themselves and know it to a pop-quizzable level. Please be extremely well versed in this document prior to arriving on your first day.

Protect Hack Reactor's Materials!

We have worked extremely hard to build this curriculum, and need your help keeping it private. Please only use our official publication script to publish the work you add to our instructional repos. A fair amount of our materials are *not* intended to be shared, as they represent untold months of R&D toil. When in doubt, do not publish!

Communication

communication@hackreactor.com

Use communication@hackreactor.com as your one point of contact once you are a student on-site at Hack Reactor, for all emails with Hack Reactor instructors and other staff members. Precourse Students should utilize admissions@hackreactor.com as their point of contact before their on-site course begins.

Why?

We have designed this system to make your life easier (only 1 email to remember!) and ensure reliable follow-up on all of your inquiries. On the flip side, we can only commit to responding to inquiries you make through communication@hackreactor.com or admissions@hackreactor.com.

- **Please set up a filter for communication@hackreactor.com and admissions@hackreactor.com** to ensure these messages get 'Marked Important' and do not go to your spam folder. You can click this link to get started, and follow these instructions if you need any more help with filters.

This must be done before entering the first day of class.

- You are expected to check your email and your cohort's Slack channel throughout the day to receive up-to-date announcements and respond to action items.

Contributing to the Curriculum

As you learn how to sling code and use GitHub, you may find yourself wanting to contribute to our curriculum's code base. Feel free to submit pull requests to the repos you work on as a student and *send an email to communication@ that points to the pull request*. As with any open source contribution, your work may or may not be merged, but many bug fixes and improvements have come from student contributions.

Building Access

Access outside of normal hours isn't guaranteed, but if you would like access to the building on a Sunday, before class, or after the front door is locked, use our doorbell app (http://door.hackreactor.com) to send an alert to folks inside the building. If someone is inside, they'll come let you in. **See Night Owl & Blue Card section below on how to gain key for access.**

- **IMPORTANT: Close the building doors when you go through**: The main downstairs door sticks open easily. You need to be careful that it shuts behind you when you go through, *especially* when going through it in a group.
- **The "Night Owl" (Closing & locking up)**: If you are ever hoping to stay later than the last staff member, you'll need to take responsibility for the space or else leave with staff. You'll be appointed as a deputy, accountable for the safety of the space. If you are still around when the last Space Ops team member leaves at 8pm, you will be given the "Night Owl" which lists a number of closing tasks including, but not limited to, checking that all the

doors are locked before leaving. Please be advised that you may stay as late as you wish with the Night Owl, but under no circumstances are you permitted to sleep at Hack Reactor. Please only stay while you are working and when you need sleep, go home. If you are ready to go home and there are others still in the space, you should pass the Night Owl to them and make sure they fully understand the responsibility.

- **The "Blue Card" (Sundays)**: HackReactor is officially closed on Sundays; this means that there is *NO* staff on site. We know how hard you work and that you are very dedicated to your code. Us being as amazing as we all are, we will give you access to Hack Reactor's work space on Sundays, given you are responsible and clean up after yourselves and host no crazy parties. It's like when your parents go away for the long weekend and you are given your chance to prove your credibility. Please be extra mindful of safety and security on Sundays as there is no security guard working in the building, and do not allow any unauthorized people to enter the building or our Space. If this proves to be unsuccessful we will take Sunday access away from all students.
- **Student Key Procedure**: If you would like to be responsible for letting people into the space on a Sunday, a holiday, or before the building is opened for class, you can exchange an important personal item (passport, driver's license, birth certificate, etc) for the keys to the building and suite. You will be the primary point person for the rest of the student body, and should monitor bell requests using http://door.hackreactor.com and your cohort's Slack channel. Visit the Front Desk on the 8th floor with your collateral no later than 4pm on the Friday before the desired day if you are interested. The keys must be returned to the Front Desk first thing Monday morning.
- **If another student has the Student Key**: The main entrance to the building is closed on Sunday and most federally observed holidays. Staff are not guaranteed to be on site. To get into the building, you must coordinate primarily with the rest of the student body using http://door.hackreactor.com and your cohort's Slack channel.

- **Visitors**: Your family is in town and they'd like to see the black hole into which you've disappeared? While a quick tour to show off what an amazing place we share is ok, longer periods of hanging out such as sitting in on lunch or a lecture are not allowed. Please email communication@hackreactor.com for clearance if you plan on doing so. All visitors must wear a visitor badge, which are located at the front desk.
- **The Roof**: In order to be in compliance with our lease, we absolutely cannot have any students on the roof. We want you to do it so little that we have installed a camera in the stairwell and we review the footage once per week. If you are caught going on the roof, you will immediately be removed from the class. No questions. Not joking. Don't go there.
- **The Douglas (HRX Lounge, 7th floor)**: Current students are not allowed into the Douglas until after they graduate and join HRX (Hack Reactor X, aka The Alumni Program).
- **Other floors:** Any floors other than 6, 7 and 8 are not open to students at any time. We have heard from the tenants on these other floors that it is disruptive to have students using those spaces. We want to be good neighbors and ask that you stay on 6, 7 and 8.
- **Bikes**: If you ride your bike to HackReactor, you can store it in the basement while you are in the space. Make sure to lock it thoroughly to the bike rack as the basement space is accessible by anyone in the building, not just HackReactor staff and students.

Systems we use to coordinate

- **Furniture and Equipment**: Email communication@hackreactor.com if you notice chairs/any furniture is broken/malfunctioning.
- **Cubbies**: Cubbies are provided for your use. Please be mindful to keep only what you need on a daily basis in there, and empty it at the end of each week on Saturday. All cubbies are labeled with student names to ensure that everyone has a space to stow their belongings while here, but we ask that you clean them out at the

end of the week to ensure that the space stays tidy and organized. We ask that you do not place any items on top of the cubbies.
- **Lost and found**: The Lost and Found cabinet is a silver cabinet on the 8th floor near the senior cubbies. If you have lost something, please email communication@hackreactor.com and check the cabinet every so often. The Lost and Found cabinet is cleaned out every Sunday, so make sure to take your items home.
- **Label your power cord**: In your folder from your first day at Hack Reactor, a small label was provided to label your power cord. Please don't take someone else's if yours is missing. Please don't take the ones belonging to Hack Reactor, or the ones labeled with the name of the space.

Attendance

- The attendance system is designed to encourage accountability by students while allowing for minimal staff intervention for punctuality and absence concerns.
- Each morning students will log into the Attendance app **before** 9AM, at which point if they do not, are considered late for class. Students are expected to sign in even if they are late as failure to sign in will result in an absence being recorded for the day.
- **IMPORTANT NOTE: In the rare instance that you are running late or expecting to be absent, YOU MUST SEND an email message to communication@hackreactor.com with brief details regarding your tardiness/absence.**
- Hack Reactor is an intensive, rigorous program and consequently we have intensive expectations for student participation. Students can miss a **maximum of three days** for unforeseen circumstances, such as illness, family emergency, etc. We also count **three tardies as equal one absence**. Please note that tardies include both arriving late and leaving early.
- From our experience, this is the maximum amount of time that you can miss and still keep up with our material. The Hack Reactor is incredibly intense and every hour here is essential to your progress and future success and therefore, we cannot allow excused

absences for any reason. Due to the nature of our program we **are not able to offer you any material missed while you are away other than those given in class.** Furthermore, the cohort works as a team and relies on having the support of the group, so having anyone out is extremely disruptive to both your own education and those of the people around you.

- Students who have more than three absences (which may be a combination of tardies and missed days) are in **violation of their Hack Reactor contract** and will be removed from the course. If you are struggling with the attendance policy, please speak with your Coordinator or Counselor.
- Students need to be **on-site** during regular programming hours **(9AM-8PM)**, outside of lunch, dinner, and work-out breaks. If you find yourself off-site during regular programming for any reason, please notify staff immediately by emailing communication@hackreactor.com.

Punctuality

- **Morning meet-up**: Doors open at 8:30 for a morning meet up where we will provide coffee, tea, and the mixings for your brew before the first lecture starts. Please arrive in the presentation area on your home floor with time to spare before the first sound plays at 8:55am.
- Anyone repeatedly arriving late will be considered to have broken their contract, whereupon a discussion with staff is required.
- **Being on time for class**: Class starts promptly when the moment the *gong* sound plays. Be sure you're already seated and situated at that time. There are 3 audio indicators that will remind you that class will start soon. A *sound effect* will play 5 minutes before, followed by an alert that sounds 1 minute before the gong which reminds you that the time is coming up.
- **If you're late to lecture, announce a reason**: Arriving to class on time is a very big deal. Just like if a staff member was late, whenever a student is late, an explanation is in order. No matter the reason, arriving late (at the start of the day or after lunch), you are expected to interrupt briefly with a greeting of either "good

morning" or "good afternoon". You'll be acknowledged, and will have a chance to explain the circumstances. **Do not simply walk in and hope no one noticed.**

- **Respecting the end time of morning meet-up**: We provide coffee and tea in the morning as an incentive to arrive early to class and spend time with your classmates.
- **Subscribe to Google notifications on the student calendar.** This feature is available on Google calendars and will notify you via email on new events, changed events and canceled events: https://support.google.com/calendar/answer/37242

Illness

Your health is our number one priority at Hack Reactor. Hack Reactor is an intensive, accelerated program and it can be easy to neglect your health during this time. Please take adequate care of yourself throughout the course and ensure that you are getting enough rest, exercising, and eating well.

Send an email to communication@hackreactor.com to let us know if you absolutely cannot come in. The program is very rigorous and it's best not to miss any time if possible. Use your best judgment to decide whether or not you can come in. If your illness starts to affect your learning, it may be a viable option to go home. Always keep in touch with us when you are going to miss any scheduled time at HROS for any reason.

If you are feeling unwell but still able to come in to Hack Reactor, be considerate of your fellow peers and take preventative measures to keep the illness from spreading and help the community remain healthy healthy. Please read the following protocols:

- If your illness is contagious, ask your Class Coordinator for a mask. This holds especially true if you have signs of coughing or sneezing, or if your illness is airborne.
- Wash your hands often and thoroughly. Rinse, lather, and repeat. We also have hand sanitizer available for your use around the space.

- We have tissues for your use as well. Feel free to use them and dispose of them properly after use.
- If your illness is contagious, we ask that you sit in the back or on the sides of the lecture area to keep the community clean.
- We have tea and Emergen-C available for all students in the kitchen.
- Don't share food, tumblers, or dishes with your cohortmates. Please wash your dishes thoroughly after every use to prevent the spread of illness.

Behavior in Lectures

- **Do not "help" lecturers by interjecting *during lecture* to clarify points for other students**: We are glad that you are passionate about helping lectures progress smoothly, but adding comments about what you've learned or trying to contribute to a teaching moment turns out to be counter-productive an overwhelming majority of the time. Without a lot of lecture training, your comments are likely to shift the focus of the lecture significantly and can break useful trains of thought.
- **Learn to explain the topics yourself**: During the lecture and during student questions periods, your main goal should be to understand the point being explained. This is what most people believe "learning" consists of. But remember, *"Expertise is knowing the right way to solve a problem. Mastery is knowing all the wrong ways too."* If you already understand a topic, you can focus on the higher-level goal of understanding exactly why it's confusing for others. This often-overlooked use of brainpower is what turns experts into masters. Your goal here should be to "hone your own ability to communicate in a way that prevents anyone speaking with you from remaining confused."
- **No genuine questions are dumb**
- **Some questions and most comments are "out of bounds" and should be deferred until after the lecture. Lecturers should/will flag out-of-bounds questions and comments.**
- **No video recording during the lectures.** Hack Reactor material is designed to be consumed only in the Hack Reactor space. Please do

not take video of your instructor during the lecture. Slides will be made available after the lecture for you to review.

- **Bring your laptop to lecture**
- **Be present during lectures**: Giving the lecture your undivided attention is expected and shows respect for the lecturer and the material being presented. Checking emails, texting, and other distractions can be disruptive to those around you.
- **Continuing to sit through a lecture is optional.** While we do require you to attend the beginning of **every lecture on time**, if you know the material being covered, you may go back to working on other projects or reviewing other lectures with which you were not comfortable.
- **If you leave the lecture, please do so quietly and with minimal distraction.**

Instructor Availability

- **"Favor mode"**: Instructors are in "favor mode" before 9am, during lunch & dinner, and after 8pm. These are off-duty times for them. Although they frequently choose to help students anyway during these hours, this is a favor they're doing for the student, and we ask you take care to make them feel thanked and appreciated for it. It's like bugging your programmer friends. They'll often help you, but you definitely owe them one, and you buy them a six pack every now and then as a gesture of thanks.
- **Never ambush instructors with technical questions**: The help request system is the appropriate way to lodge a technical question. If you're not convinced that the instructor will be genuinely captivated by what you're about to say, it's probably an item better suited for a help request. This distribution of labor ensures that you get the help you need while reserving instructors' time for instructing.
- **(Almost) Never ambush staff with non-technical questions**: If you're trying to resolve some question about the program or what you should do in a particular situation, email is more appropriate than is instinctive. If you think it's an exception this time, ask yourself: Is this urgent? Is it also crucial? If not, email.

    - Example exceptions:
        - "My friend is the VP of Engineering for Facebook, and wants to visit the school in twenty minutes. Is a staff member free?"
        - "I just broke down crying in the bathroom. Can we talk?"
    - Example non-exceptions:
        - "Hey, my friend is [insert impressive description]. Should I invite him/her to be a guest instructor or speaker?"
        - "Hey, I'm going to be out next week Saturday for a wedding. Can I get early access to the repo?"
- **The back offices are for staff only**: Please do not go hunting for staff in the back offices. If you want to send a query to a specific staff member, email communication@hackreactor with your request and they will direct it to the proper place.

Help System

- **Don't ask for help before doing research yourself**.
- **Don't ambush Help Desk HiRs**: You **must** submit a help request through Bookstrap.
- **Hit the help button and then wait your turn**: our goal is to have all help requests attended to within ~5 minutes, and to point you in the right direction within ~5 minutes of working with you.
- **Ask for help finding the answer; do not ask for the answer**: the Help Desk staff are not supposed to give you answers. Rather, they should point you in the right direction and show you their thought processes. Much of what you will learn from working with them is the debugging process. Giving you the answer cheats you from the opportunity to practice applying the correct methodology to quickly solving your own problems.
- **Sometimes your help desk instructor will not know the answer:** This will always be true at HR or anywhere else, is not to be feared, and will ultimately provide you with access to a skill that otherwise would not be represented in the class. This kind of event allows the help-request respondent to model their thought process

for you in how they solve issues and go about figuring out what is wrong.

- **Appropriate topics with which to hit the help desk**
    - A question about what a sprint prompt means
    - A question about a toy problem
    - A question about something you didn't understand in lecture
    - A question about a blog post you're writing
    - An issue you and your pair are having
    - "Can I ask you a quick question?"
- **Don't be the Help Desk for your classmates**. Much of the work you do as a Junior will look like this: you wrestle with what to do for hours and make a ton of mistakes before writing the 3 lines of code that make it work. The wrestling and making of mistakes is **way** more valuable than writing the correct lines of code, so, while it can be tempting to let your struggling classmates know how to get out of the mess they are in, don't rob them of the learning that occurs by figuring it out for themselves. Our Help Desk staff are trained to walk this line and you should leave it to them to help get your classmates unstuck, when necessary.
- **Do be a Helpful Classmate**. While you **must** be disciplined so that you do not spoil the learning of your fellow classmates, being a helpful and supportive classmate in other ways is a fantastic idea! Use the following guidelines when supporting your classmates and you can be confident your support will be as genuinely supportive as you could wish:
    - Insist your classmates can describe in detail their error messages, failed tests, or unexpected behaviors. Over half of all stuck students have all the info they need in their error messages, or by understanding specifically what is making the test fail.
    - Insist your classmates can prove what they assume to be true. Most failed debugging efforts are based on exploring ideas that sound reasonable, but are not based on actual observations about the code. Correcting this usually looks like slowing down and describing how the code is working by referencing the code itself, and using log and/or debugger

statements to make sure the code is behaving the way they assume.
    - Don't be the one doing all the talking. And when you do talk, try to ask orienting or clarifying questions.
    - Exit the scene (or keep yourself entirely quiet) as soon as you feel confident your classmate has something new to explore. This can be uncomfortable because it usually occurs before there is a feeling of success or resolution but is essential for your support to be genuinely supportive, and not at odds with the need to encourage your classmate's autonomy.

Workstation Etiquette

- **Do not touch the underside of the computers/workstations**: If you need anything adjusted, or have any specific needs, email communication@ with your request and they'll make sure IT or Space Ops gets it all sorted for you. Ask, don't touch. It will make life much easier for everyone.
- **No open containers at the workstations**: We lose an amazing amount of equipment to careless liquid spills. Under no circumstances are open containers to be found by the workstations, at any time. This includes food and beverages in open cups.
- **No food or eating at the pairing stations**: Keyboards get sticky and jammed, workstations littered, and food attracts flies. Liquids in closed containers are acceptable, but please consume all snacks and meals in the kitchen area.
- **Judiciously change the workstation settings**: We want to maintain a consistent experience across the workstations so that student and instructor habits can be portable. Additionally, some workplaces will expect you to use a standard profile, this is a good chance to practice adapting to your environment. You should not change settings that are strictly personal preferences, however you should feel free to install or update tools that would otherwise block your progress on a sprint. For example, do not change the keyboard profile to Dvorak mode, however do use npm to install the latest version of node-inspector. If you feel a change would

improve matters for everyone's productivity, or have any other concerns, email communication@hackreactor.com.

- **Computer Monitors**: Please, **do not** flex the monitor arms out from the desk. The monitors should be left in their original configuration. **Please turn off monitors after use.** Do not use them as shades for the sun, or force them out of the way to make room for your laptop. If you want to work on your laptop, use an open table instead.
- **Files on the workstations are routinely deleted**: You should be using Git to save any work you're doing on the curriculum, and pushing that work to Github before going home. Any files found laying around the operating system will be automatically deleted from time to time. (Each student has one place on each workstation where they can keep files that will not be deleted, in the Desktop/STUDENT folder.) Workstations are wiped any time the computer restarts and a restart is scheduled for 3am each day.
- **Closing Applications**: Please shut down ALL applications before you leave at the end of the day so that workstations can restart as scheduled at 3am (This includes iTunes, App Store, Node.js)
- **HackReactor Equipment**: Please, **do not** remove, swap, or change any cords, or other equipment from HackReactor computers, or staff/student desks. If you need something, please e-mail communication@hackreactor.com and we will help you to the best of our abilities.

Kitchen Etiquette

- **Coffee Grinding** Please do not interrupt lectures and other hard-working students/staff by grinding coffee beans or producing any other distracting noises.
- **Coffee Cups**: The best bet is to bring your own travel mug with a lid on day 1 and label it as yours. Otherwise, we have purchased closed lid travel mugs for you to borrow while you are hanging out here in our space. Please treat these cups as you would your own and always be sure to wash it at the end of the day.
- **Shared Refrigerator**: Please keep in mind that the refrigerator is a shared space and all students on your floor need to be able to

access it. In respect of the refrigerator real estate and your fellow cohort-mates, only bring in 1-2 days worth of food at a time.

- **Label your food with name & date**: Anything without a name and dated label will be thrown out.
- **Throw out your food**: Don't leave food in the refrigerator for more than a few days. If you choose to use the shared refrigerator space, no one but you is responsible for keeping your leftovers in order. Every Saturday, the refrigerator will be cleaned and if you are storing any items there, they will be removed and placed on the table for you to claim before they are thrown away.
- **Food Deliveries**: If you are having food delivered to Hack Reactor, tell the delivery person to bring it to the 8th Floor Front Desk where you should be waiting for them, so that it does not interrupt the workflow of staff on other floors who have to receive and sign for your food and then figure out what to do with it. Time the delivery so that you are available to receive your food, the Front Desk doesn't have the resources to contact students to notify them of their delivery. If food deliveries become disruptive, we may ask that you not have food delivered to our space.
- **Dishes**: If you are on the 6th or 8th floor, please use the dishwasher to clean dishes. If dishwasher is full or running (as indicated by the sign), please hand-wash your dish. The 7th floor is primarily for staff use only, so please do not use the kitchen on this floor. *Do not* leave your dishes in the sink for others to wash.
- **Top of the refrigerator and cabinets**: Please do not use refrigerator tops for storage. Food rots and any other things being placed there make the space look cluttered.
- **Disposal System**
    - **Recycle** as much as possible
        - YES: Rigid plastic
        - YES: Aluminum cans
        - YES: Glass bottles
        - YES: Clean paper
        - NO: Soft plastic like saran wrap/plastic bags
        - NO: Pizza boxes, and other food-stained papers/cardboard products (compost instead)

- **Compost** as much as possible of what's left
    - YES: Food scraps
    - YES: Paper plates/bowls/cups/napkins
    - YES: Pizza boxes
    - NO: Metal (This includes Chipotle burrito covers, foils, and cans)
    - NO: Plastic of any sort (See Recycle for appropriate disposal of plastics)
    - NO: Tetra paks (Almond & Soy milk containers)
    - NO: Tea bag envelopes; these are lined with a non-recyclable plastic.
- **Landfill** as a last resort.
    - YES: Soft plastic must go in the landfill. (Plastic bags, tea bag envelopes, tetra paks)
    - YES: Plastic straws
    - YES: Stickers
    - YES: Laminated paper
    - OK: Any rigid plastic food containers that you can't stand washing out is best in the landfill, so it doesn't contaminate the paper in the recycling.

Conference Room Etiquette

The conference rooms here are a scarce resource, and it's essential that staff have access to them to get their work done. In general, conference rooms are not available for student use.

IMPORTANT NOTE: When staff members are in meetings in any of the conference rooms, please don't walk in and interrupt the meeting. It is very disruptive to the team when they are stopped during their scheduled events. Unless it is an emergency, please refrain from interrupting any of the events occurring in the conference rooms.

7th Floor

The 7th floor is reserved for staff only outside of lectures times; we ask that students stay on the 6th and 8th floor when they are not attending lecture. Staff come to the 7th floor to work without interruption and expect a quiet workspace. In some cases, staff members may choose to conduct a meeting with a student on the 7th floor if no other spaces are available. Even in the case of a legitimate exception such as this, please expect that you will be approached by a member of the Space Operations Team if you're on the 7th Floor outside of this time.

Take care of things

- **Use erasers to wipe off the small whiteboards**: The carpet is not a reasonable alternative. You wouldn't wipe a whiteboard on your carpet at your relatives' house, and it's equally inappropriate here. Yes, this actually happened. I know, right?
- **Return whiteboards after using**: In order to ensure that everyone has can access to the myriad of mobile whiteboards in the space, please remember to return small and medium whiteboards to the whiteboard supply station near the microwaves or, if you are utilizing a larger easel style whiteboard, please return it to the alcove near the senior cubbies. Bonus points if you erase it first!

Safety

Precautions recommended by the Tenderloin Police Dept:

- They recommend buddying up with someone else.
- They recommend going in a BART entrance that is not the closest one to HROS. **Cross the street to enter the MUNI/BART station**: The north-west underpass entrance to Powell Station (closest to HROS) is very dark at night and can be a place for crime to occur. The Southern entrance across Market Street (next to Nordstrom) is a better-lit, potentially safer choice.
- We recommend saving their number into your phone: 415-345-7300.
- If you feel unsafe going to the BART late at night, you can call them and they will send out an escort. However, they are not able to give a good estimate of when they will be able to swing by, so you may be waiting a while.

- Do not wear earphones while walking.
- Do not have valuables in plain sight on your way to the BART (like texting while walking).
- If you are going home late, and are not planning on working on your computer, leave it at HROS. The odds of someone breaking into HROS and stealing your laptop are lower than someone snagging it from you on the BART/on your way home (but, of course, we are not liable for your belongings, and you leave them here at your own risk).

Addionally, keep your head up when leaving the building. Wait until you get to where you're going to put on your headphones, and don't bury your head in your smartphone.

Conduct

- **Minimum Requirements**: We want to create the most cohesive learning environment possible. Therefore, Hack Reactor has a zero-tolerance policy toward hostile and disrespectful treatment of other students. If you find yourself raising your voice or being unkind to another student, take a deep breath, stretch, or go for a walk to ground yourself. It should go without saying that lewd behavior, bigotry and "isms" of any sort, and raunchy locker room humor is strictly forbidden. Behave like you would in a professional workplace, not like you would in a bar.
- **Illness**: Send an email to communication@hackreactor.com and *please* be responsible about staying home when you're sick! The program is very rigorous and it's best not to miss any time if possible. That said, coming into the office when you are sick can jeopardize others' productivity. Use your best judgment and most importantly, always keep in touch with us when you are going to miss any scheduled time at HRHQ for any reason.
- **Drugs and Alcohol on campus**: Hack Reactor is a drug and alcohol free campus. There are rare occasions when HR hosts events in which alcohol is provided, however, drinking/bringing alcohol, or using illicit drugs to campus is prohibited. Inebriated hacking can

lead to really bad code and is disruptive to the learning environment.

- **Competition**: A bit of friendly competition as motivation can be a positive thing, but striving to be better than others doesn’t help you reach your potential as a person. (Read more in The Case Against Competition and No Contest: The Case Against Competition Be eager to offer help, especially to those who look frustrated, or have had a few bad days. Be sure your help is actually wanted before you start giving it. Pay it forward and be the support you would like to receive in that instance. Note that although we organize some weekly competitions between students and staff, we do not condone or facilitate student-student or class-class competitions.

How to deal with discrimination or harassment

As the student agreement states, discrimination and harassment are completely unacceptable within our tightly-knit program. While we encourage you to assume your colleagues have the best intentions in everything that they say and do here in the community, it happens in rare occasions that people accidentally or neglectfully hurt the people around them with discriminatory statements or actions. If you are ever concerned that you or a fellow student is suffering from such an incident, please elevate the information to your cohort's class coordinator or counselor immediately. They will take the appropriate next steps to minimize potential harm for all parties involved. It's extremely important that you trust in your operators and counselor to handle such reports with sensitivity, since silence only enables such situations to get further out of hand. You can expect the coordinator and counselor to deal with the information responsibly, elevate it to the appropriate level for help when needed, and handle everyone involved with as much respect for the privacy of all parties as possible.

Subcommunities within Hack Reactor

A number of subcommunities exist within Hack Reactor, and if you're interested in participating in any of them, please see the *Communities at*

Hack-Reactor (see left hand menut) page for details on how to get involved once you are on-site!

Building Safety

During week one we ask that students wear the Hack Reactor name badges issued on day one so that our security guards can learn who you are and recognize you.

Please familiarize yourself with the following maps. Note the location of emergency exit routes, fire alarm pull stations, fire extinguishers, first aid kits and emergency plans.

- **6th Floor Map**: 6th Floor Map
- **7th Floor Map**: 7th Floor Map
- **8th Floor Map**: 8th Floor Map

Advice

- **Go to lunch with your peers**: Even if you bring your lunch. Going out to bond will serve you tremendously, both in your career and for your general health and happiness.
- **Mingle with as many different people as possible**: The other classes, the alumni, the staff, the visitors. If you find yourself out to drinks with a bunch of folks, go sit with the ones you don't already know.
- **You learn more than just coding at Hack Reactor**: When companies say that they hire for a "culture fit," they mean that they want employees who are kind and compassionate, making them easy to collaborate with and be around. H/R graduates are team-oriented individuals who will represent themselves and their school well to the community at large.
- **Make a good impression**: Remember that the people you meet here--students, alumni, instructors, and staff--are your best connections as you move forward in this industry. Use this time to make the best impression possible on us all. Everyone here wants to be a bridge for you--don't burn them. Take advantage of your

extra-curricular opportunity to fill your network with amazing people who care deeply about your success.

Thanks for learning the etiquette! :)

Two communities at Hack Reactor have self-organized and are excited for you to get involved! The virtual community exists on their respective mailing lists, where members can organize get-togethers IRL, as well as share articles and start discussions.

For current students and alumni, to join the **Women@** Hack Reactor, email communication@hackreactor.com to be added to the list.

For current students and alumni, to join the **Queers@** Hack Reactor, email erik.brown@hackreactor.com to be added to the list.

If you are a precourse student who is interested in joining one of these communities, we kindly ask that you wait until you are on-site to submit your request.

If you have ideas for other communities that should be organized via a mailing list, the folks on communication@hackreactor.com would happily hear the suggestion.

Appendix and Recommended Reading

This section is not required, but once you're done with the required assignments, you might have time and enthusiasm left for getting extra prepared. If so, here are some good things to spend your time focusing on.

Impostor Syndrome

"Impostor Syndrome" is a term for the persistent and foreboding feeling that I am undeserving of my position, that everyone around me is more talented and qualified than I am, and that any day now the other shoe is going to drop. It's all bad news, of course; feeling unqualified only sandbags future learning. Don't be alarmed! For numerous reasons it's a common experience for Hack Reactor students, and you'll hear us talk about managing it a lot. Read up on how to approach it if you're curious--note that while the article focuses on the experience of women in tech, absolutely anyone can have impostor syndrome and the points made by the article are applicable.

Style guide

Writing good-looking code is often treated as an afterthought by beginners, but it's extremely important to your development as a software engineer and it's a continual focus in our curriculum. Get into good habits early on by reading the *Hack Reactor style guide* (see left hand side of Learn2) before class; the code you write while you're a student here will be expected to conform to these guidelines.

More command line

The more comfortable you are with the shell, the easier you'll find the busywork of programming. Taking the time early on to really immerse yourself in the command line will pay huge dividends down the road.

The Command Line Crash Course

This is part of the Learn Python the Hard Way course, which you do not need to do at all, except for this awesome intro into the command line.

- LINK: https://learnpythonthehardway.org/book/appendixa.html
- DURATION: 4-5 Hours

Running Sublime from the Shell

If you're using Sublime Text, you can configure it to run with a shell command and then set it as your default editor in Git.

More Git

Git is a surprisingly simple and elegant tool, but its interface doesn't hide much of its complexity. Spending some extra time to understand how it works will help you avoid confusing surprises later on.

GitImmersion: Labs 30-50

- LINK: http://gitimmersion.com/lab_30.html
- DURATION: ?

Learn Git Branching

- LINK: http://pcottle.github.io/learnGitBranching
- DURATION: ?

An interactive guide covering Git's branching features, an essential part of a professional Git workflow.

The Hack Reactor Style Guide

This page is maintained by the Infrastructure Team, and no un-approved edits are allowed. Please do not make any edits to this guide unless they are approved by the CTO or other infrastructure team leadership.

The first thing anyone notices about your code is its style, so start prioritizing correct whitespace and other details when using the language. We've collected a list of reasonable choices into a style guide that you should adopt while you're here.

A few of these points are of critical importance to either your code's reliability or its legibility. Those points will be made first, followed by the less critical rules you should set for yourself. The style guidelines you adopt should address the following goals:

- Readability for both junior and senior engineers
    - Your readers will be both new and experienced engineers, so strive for a balance in readability for both. You can expect your reader to investigate language devices they are not familiar with, but not every minification is an improvement, even if there are some very senior engineers who could figure out what you're trying to express.
- Maintainability
    - Sometimes an especially terse or 'elegant' expression can complicate the process of reading, debugging, or refactoring. Remember that your code will be reread many months from now by people who have never seen it or have completely forgotten it.
- Brevity
    - This is the least important priority, but it's still of some importance. Try to keep your code as short and clear as possible while adhering to the goals above.

Important style issues

The following are style guidelines that we expect you to follow, and our assessment of your code's quality will depend in part on adhering to them.

Best Practices for your development workflow

We recommend that students learn as much about efficient use of their tools as possible. It is not taught or required in the course, but these techniques are particularly useful to master.

- Learn and use as many keyboard shortcuts as possible. Avoid touching the mouse whenever possible.
- Practice touch typing as much as possible (don't look at your hands, use all your fingers, and keep them resting on the home row).
- Operate programs in full screen mode. You can toggle between them more effectively by using `command`+`tab` and `command`+`~` than by using the mouse.

Don't use the dangerous stuff til we get there

- Avoid any use of the keywords "this" or "new" before they are introduced in class. They almost definitely don't work the way you think, and it will be easiest to teach them to you if you haven't acquired a lot of confusing baggage first.

Indentation

When writing any block of code that is logically subordinate to the line immediately before and after it, that block should be indented two spaces more than the surrounding lines

- Do not put any tab characters anywhere in your code. You would do best to stop pressing the tab key entirely.
- Increase the indent level for all blocks by two extra spaces
  - When a line opens a block, the next line starts 2 spaces further in than the line that opened

```
// good:
if (condition) {
  action();
}

// bad:
if (condition) {
action();
}
```

- When a line closes a block, that line starts at the same level as the line that opened the block

```
// good:
if (condition) {
  action();
}

// bad:
if (condition) {
  action();
  }
```

- No two lines should ever have more or less than 2 spaces difference in their indentation. Any number of mistakes in the above rules could lead to this, but one example would be:

```
// bad:
transmogrify({
  a: {
    b: function(){
    }
  }});
```

  - use sublime's arrow collapsing as a guide. do the collapsing lines seem like they should be 'contained' by the line with an arrow on it?

Variable names

- A single descriptive word is best.

```
// good:
var animals = ['cat', 'dog', 'fish'];

// bad:
var targetInputs = ['cat', 'dog', 'fish'];
```

- Collections such as arrays and maps should have plural noun variable names.

```
// good:
var animals = ['cat', 'dog', 'fish'];

// bad:
var animalList = ['cat', 'dog', 'fish'];

// bad:
var animal = ['cat', 'dog', 'fish'];
```

- Name your variables after their purpose, not their structure

```
// good:
var animals = ['cat', 'dog', 'fish'];

// bad:
var array = ['cat', 'dog', 'fish'];
```

Language constructs

- Do not use `for...in` statements with the intent of iterating over a list of numeric keys. Use a for-with-semicolons statement in stead.

```
// good:
var list = ['a', 'b', 'c']
for (var i = 0; i < list.length; i++) {
  alert(list[i]);
}

// bad:
var list = ['a', 'b', 'c']
for (var i in list) {
  alert(list[i]);
}
```

- Never omit braces for statement blocks (although they are technically optional).

```
  // good:
  for (key in object) {
    alert(key);
  }

  // bad:
  for (key in object)
    alert(key);
```

- Always use `===` and `!==`, since `==` and `!=` will automatically convert types in ways you're unlikely to expect.

```
// good:

  // this comparison evaluates to false, because the number zero is not the same as the empty string.
  if (0 === '') {
    alert('looks like they\'re equal');
  }

  // bad:

  // This comparison evaluates to true, because after type coercion, zero and the empty string are equal.
  if (0 == '') {
    alert('looks like they\'re equal');
  }
```

- Avoid using function statements for the entire first half of the course. They introduce a slew of subtle new rules to how the language behaves, and without a clear benefit. Once you and all your peers are expert level in the second half, you can start to use the more (needlessly) complicated option if you like.

```
  // good:
  var go = function () {...};

  // bad:
  function stop () {...};
```

Semicolons

- Don't forget semicolons at the end of lines

```
// good:
```

```
alert('hi');

// bad:
alert('hi')
```

- Semicolons are not required at the end of statements that include a block--i.e. `if`, `for`, `while`, etc.

```
// good:
if (condition) {
  response();
}

// bad:
if (condition) {
  response();
};
```

- Misleadingly, a function may be used at the end of a normal assignment statement, and would require a semicolon (even though it looks rather like the end of some statement block).

```
// good:
var greet = function () {
  alert('hi');
};

// bad:
var greet = function () {
  alert('hi');
}
```

Supplemental reading

Code density

- Conserve line quantity by minimizing the number lines you write in. The more concisely your code is written, the more context can be seen in one screen.
- Conserve line length by minimizing the amount of complexity you put on each line. Long lines are difficult to read. Rather than a character count limit, I recommend limiting the amount of complexity you put on a single line. Try to make it easily read in one glance. This goal is in conflict with the line quantity goal, so you must do your best to balance them.

Comments

- Provide comments any time you are confident it will make reading your code easier.
- Be aware that comments come at some cost. They make a file longer and can drift out of sync with the code they annotate.
- Comment on what code is attempting to do, not how it will achieve it.
- A good comment is often less effective than a good variable name.

Padding & additional whitespace

- Generally, we don't care where you put extra spaces, provided they are not distracting.
- You may use it as padding for visual clarity. If you do though, make sure it's balanced on both sides.

```
  // optional:
alert( "I chose to put visual padding around this string" );

// bad:
alert( "I only put visual padding on one side of this string");
```

- You may use it to align two similar lines, but it is not recommended. This pattern usually leads to unnecessary edits of many lines in your code every time you change a variable name.

```
// discouraged:
var firstItem  = getFirst ();
var secondItem = getSecond();
```

- Put `else` and `else if` statements on the same line as the ending curly brace for the preceding `if` block

```
// good:
if (condition) {
  response();
} else {
  otherResponse();
}

// bad:
if (condition) {
  response();
}
else {
  otherResponse();
}
```

Working with files

- Do not end a file with any character other than a newline.
- Don't use the -a or -m flags for `git commit` for the first half of the class, since they conceal what is actually happening (and do slightly different things than most people expect).

```
# good:
> git add .
> git commit
[save edits to the commit message file using the text editor that opens]

# bad:
> git commit -a
[save edits to the commit message file using the text editor that opens]

# bad:
> git add .
> git commit -m "updated algorithm"
```

Opening or closing too many blocks at once

- The more blocks you open on a single line, the more your reader needs to remember about the context of what they are reading. Try to resolve your blocks early, and refactor. A good rule is to avoid closing more than two blocks on a single line--three in a pinch.

```
// avoid:
_.ajax(url, { success: function () {
  // ...
}});

// prefer:
_.ajax(url, {
  success: function () {
    // ...
  }
});
```

Variable declaration

- Use a new var statement for each line you declare a variable on.
- Do not break variable declarations onto mutiple lines.
- Use a new line for each variable declaration.
- See http://benalman.com/news/2012/05/multiple-var-statements-javascript/ for more details

```
// good:
var ape;
var bat;

// bad:
var cat,
    dog

// use sparingly:
var eel, fly;
```

Capital letters in variable names

- Some people choose to use capitalization of the first letter in their variable names to indicate that they contain a class). This capitalized variable might contain a function, a prototype, or some other construct that acts as a representative for the whole class.
- Optionally, some people use a capital letter only on functions that are written to be run with the keyword `new`.
- Do not use all-caps for any variables. Some people use this pattern to indicate an intended "constant" variable, but the language does not offer true constants, only mutable variables.

Minutia

- Don't rely on JavaScripts implicit global variables. If you are intending to write to the global scope, export things to `window.*` explicitly instead.

```
// good:
var overwriteNumber = function () {
  window.exported = Math.random();
};

// bad:
var overwriteNumber = function () {
  exported = Math.random();
};
```

- For lists, put commas at the end of each newline, not at the beginning of each item in a list

```
// good:
var animals = [
  'ape',
  'bat',
  'cat'
];

// bad:
var animals = [
    'ape'
  , 'bat'
  , 'cat'
];
```

- Avoid use of `switch` statements altogether. They are hard to outdent using the standard whitespace rules above, and are prone to error due to missing `break` statements.
- Prefer single quotes around JavaScript strings, rather than double quotes. Having a standard of any sort is preferable to a mix-and-match approach, and single quotes allow for easy embedding of HTML, which prefers double quotes around tag attributes.

```
// good:
var dog = 'dog';
var cat = 'cat';

// acceptable:
var dog = "dog";
var cat = "cat";

// bad:
var dog = 'dog';
var cat = "cat";
```

HTML

- Do not use ids for html elements. Use a class instead.

```
<!-- good -->
<img class="lucy" />

<!-- bad -->
<img id="lucy" />
```

- Do not include a `type=text/javascript"` attribute on script tags

```html
<!-- good -->
<script src="a.js"></script>

<!-- bad -->
<script src="a.js" type="text/javascript"></script>
```