Lab 2

In this lab you will learn how to write tests, and practice some of the structure-building concepts we have learned so far.

Testing Basics

When we refer to tests, we typically mean automated tests. Those tests fall into various categories:

Unit Tests Unit tests test a single tiny individual component of your application. You typically want to have unit tests for every bit of your code that is part of your application's *interface*.

You should avoid testing for things that are too implementation-dependent.

Unit tests are a crucial part of the refactoring process, whose goal is to rearrange and rewrite sections of your code. A solid suite of unit tests can allow you to do this freely without worrying about breaking code. "Your tests will catch that". And revision control allows you to recover if you've messed things up too much.

Integration Tests Integration tests test bigger parts of your application, making sure that different parts come together naturally.

Time permitting, we will talk more about integration tests later in the term.

Timing Tests Timing tests are used in algorithm implementations to assess the efficiency of the algorithms.

They can also be used to try to find bottlenecks in your application, though some of the browser profiling tools might be better.

Deployment Tests Deployment tests are meant to ensure that your application performs well on various browsers / deployment environments. Hard to do.

We will focus on unit tests for now.

Test-Driven Development

In Test-Driven-Development, you typically would follow these steps:

- Decide on a small piece of functionality you want to add.
- Make a GitHub issue about it.
  - If you prefer, you can create one bigger more "logical" issue, and create a "task list" in it, following the example at this blog post[1]. Then check those items off as you implement them.

[1] https://github.com/blog/1375%0A-task-lists-in-gfm-issues-pulls-comments

- Write a test for the code you want to introduce.
- Run your tests, and watch this new test fail. This makes us more certain that the test does indeed detect the feature we want to add.
- Optionally, make a git commit of the test, using "#..." to reference the issue you created.
  - This is a bit of a style decision, whether to commit the tests separately or whether to do one commit containing both test and new code.
- Write a minimal set of code that would make the test pass.
- Check that all your tests pass.
- Make a commit, using "#..." to reference the issue you created. Say "close #..." if it was a "single-problem issue".
  - This gives you a safe backup point to revert to.
- Consider any refactoring that you might want to have take place.
- Do the refactoring, and make sure your tests still all pass.
- Commit (optionally creating an issue first to explain what the refactoring was about).

This is some of the general theory behind testing. The lab readme will have more specific instructions.

Testing in Javascript

We will be using mocha[2] in combination with chai[3] for our unit tests. Here is a basic html file:

```
<!DOCTYPE html>
<html>
<head>
   <link rel="stylesheet" href="mocha.css" />
</head>
<body>
   <div id="mocha"></div>
   <script src="chai.js" type="text/javascript"></script>
   <script src="mocha.js"></script>
   <script>
      mocha.setup('bdd');
      var expect = chai.expect;
   </script>
   <script src="yourCode.js"></script>
   <script src="tests.spec.js"></script>
   <script>
      mocha.checkLeaks();
      mocha.globals(['chai']);
      mocha.run();
  </script>
```

[2] http://mochajs.org/
[3] http://chaijs.com/

```
</body>
</html>
```

All the tests go into `tests.spec.js` (or a similarly named file). You can also use multiple test files, just put them after each other. Use the rest as a template on every project.

Basic steps

1. You should have already decided which of the two partner's GitHub account to use.
2. If you have not switched roles with your partner so far, this is a good time to do so.
3. Check out the correct branch, via `git checkout Lab2`.
4. Open the `README.md` file there, it will contain the instructions on what you need to do.
5. Commit your changes, review them, push them to your repository, and when you are ready to submit simply email me a link to your project and the SHA of the commit that contains your final submission.
6. You should use the issues page to track your progress. Create a new label called "Lab2" and use it to tag all issues related to this lab. I will review those issues to look at your work.
7. Needless to say it ,but you are NOT allowed to look at other people's forks of the project and their issues/solutions.
8. I expect you to do the coding using pair programming.