Testing & Debugging JavaScript

By: Rob Levin
Version: 0.0.1
Dated: Mon, October 19, 2009

Disclaimer

The information in this document is based on publicly available documentations and author's personal & professional experience. In no event shall author be liable for any direct, indirect, consequential, punitive, special or incidental damages (including, without limitation, damages for loss of profits, business interruption or loss of information) arising out of the use or inability to use this document, even if Author has been advised of the possibility of such damages. Author makes no representations or warranties with respect to the accuracy or completeness of the contents of this document and reserves the right to make changes to this document at any time without notice. Author does not make any commitment to update the information contained in this document.

Title	TDD JavaScript
Author	Rob Levin
Email	roblevintennis@gmail.com (author) johri.maya@gmail.com (template author)
Version	0.0.1
Document Date	10/19/09Notes

Table of Contents

Introduction

Contents:

1.1. Why this book?

While there are a plethora of good JavaScript books, blogs, etc., information on testing JavaScript is scattered across the net and a bit cumbersome to wade through. The author's hope is to remedy this with a nice “ramp up” document for developers that need to get up and running quickly.

1.2. Rationale

Most programming books put off getting in to things like debugging and testing until very late in the book (if at all). This is a bit of a shame because debugging and testing are not only indispensable skills, they're actually quite helpful in aiding the learning of a new language.

An example of using testing to *learn and confirm* features of a new language can be found in the Part II: The xUnit Example in Kent Beck's TDD By Example book. Mr. Beck actually uses unit testing to both “play” with Python and build a unit testing framework from out of thin air! [BKTDD2002]

TDD JavaScript will attempt to allow the user to “kill two birds with one stone” - get up to speed on both JavaScript and testing JavaScript programs.

1.3. Goals of this book

This book is not meant to be an exhaustive reference or even complete tutorial on JavaScript – there are already a number of good choices for that:

- JavaScript: The Good Parts by Douglas Crockford
- JavaScript: The Definitive Guide by David Flanagan
- Professional JavaScript for Web Developers by Nicholas C. Zakas
- Books and blogs by John Resig

There are of course others as well – these are some the author can vouch for. Please note that these may be considered advanced so a newbie will likely need another resource before tackling any of these. Both w3schools and Mozilla, have some very nice free online alternatives for those just trying to get started.

https://developer.mozilla.org/en/Core_JavaScript_1.5_Guide
http://www.w3schools.com/js/default.asp

This book's primary goals are:

1. Get you up and running using JavaScript AND testing quickly.
2. Act as supplemental material to more authoritative tomes (see books above).

1.4. Who should read?

This book is aimed at the following cross-section of readers:

- Those coming to JavaScript from another language (eg Java, PHP, C#, etc.)
- JavaScript programmers that would like to test, but only if they can get set up and going quickly.
- Those that have avoided JavaScript like the plague in favor of server side programming, but now find themselves in a project that demands client-side programming skills.
- A fast reader that wants to do an hour's research and get a sense of some of the tools presented before wading through a bunch of README's and environment setup. If this is you, just read the first part of this book to get a sense of some nice JavaScript tools

available.
- Although this book isn't targeted at newbies, the exploration of testing frameworks, and setting up a solid JavaScript development environment should prove as valuable supplemental material. If things get too complicated, just move on to another section and come back to these sections later in career.

If you are already a JavaScript guru, have already set up a complete TDD JavaScript environment, etc., this book may not be for you. You may want to cherry pick the chapters that interest you specifically. Perhaps there's a tool or two covered that you just haven't got around to using yet? Moreover, you may have the job of getting a new hire, junior developer, etc., to use some discipline in their JavaScript development but don't have time to pair program them. You may consider using this as a resource to "throw at them".

1.5. What this book is not?

A lot! We are not going to get into every testing framework or debugger and we will be doing most of our work in Firefox. That being said, the reader is definitely encouraged to also test their scripts in IE, Chrome, Safari, Opera, etc. However, there's just not enough room for this book to cover all of these.

As brevity is one of the chief goals of this document, it will be up to the reader to find other background material to learn the rationale for things like TDD (test driven development), BDD, xUnit. One suggestion, is that if this book makes a foreword reference to something you are not familiar with, take 10 minutes to read the Wikipedia, etc. Moreover, we won't be getting into every feature available from each framework. Unfortunately, that means Ajax, mocks and stubs, and certain other features have been left out. The goal is to give you a point of departure from which you can move into these more interesting areas on your own.

Also, this book does not try to get deep into the JavaScript language itself. It is meant to be used as supplemental material to a more "proper" JavaScript book. The author does plan to use parts of the Zakas and Crockford books previously mentioned to inspire some of the examples later in the book (specifically, we will use some tests to iron out confusion of Object Creation and Inheritance). If the reader is so inclined, they will probably get a lot out of using this as a study aid on top of those more exhaustive resources. But not, neither is for the faint or heart!

Lastly, this document is not meant to act as a comparison chart or matrix on the tools examined!

1.6. Examples

The code examples were ran on either Mac OS X (10.5.8) or Ubuntu 8+ system (or both) running Firefox 3.5.3. You system's results may be different.

The author maintains a repository on github with the book itself, as well as any supplemental code, etc., at:
http://github.com/roblevintennis/Testing-and-Debugging-JavaScript
As it is always challenging to take source code from a word processed document, it is strongly suggested that you grab the code from there (as opposed to cut and pasting from this document). Moreover, it is likely that any updates in the source will be in the github code before this document!

1.7. Disclaimer

This is a free book, and the author is doing this on the side (on top of full time development and family concerns). As it is more important to get the information "out there" and provide that it is

accurate, the formatting may leave much to be desired. If this really bothers the reader, and they have the knowledge to do so, they may contribute to this project by reformatting the book (if they are so inclined). If so, please create a clone and add it. Moreover, the author does not claim to be a superstar of JavaScript wizardry – if you are only interested in getting information from the *"heads of state"*, please see the list presented earlier (or search for JavaScript + any of the following: Crockford, Zakas, Resig, Flannigan, Edwards, PPK, etc.)

Part I – Debugging Tools

This section of the book is meant to provide short chapters on some of the various tools available for debugging and testing JavaScript.

Firebug

Contents:

2.1 Introduction

Firebug is a Firefox Add-on that gives you a grab bag of web development tools to aid in debugging your JavaScript code. You can use it: as a JavaScript interpreter; to inspect DOM elements on a web page; edit HTML, CSS, and JavaScript in-place and view the results live in your browser; debug code using typical breakpoint/step functionality found in IDE debuggers; view network stats to determine page responsiveness. The current url for Firebug is: http://getfirebug.com/

There's also a great presentation by the original author himself, Joe Hewitt: http://yuiblog.com/blog/2007/01/26/video-hewitt-firebug/

2.2 Installation

The following screen-shots show what the author did to install the Firebug Add-on for Firefox:

Figure 1-1. Finding Firebug is easy from the Firefox Add-ons search page.

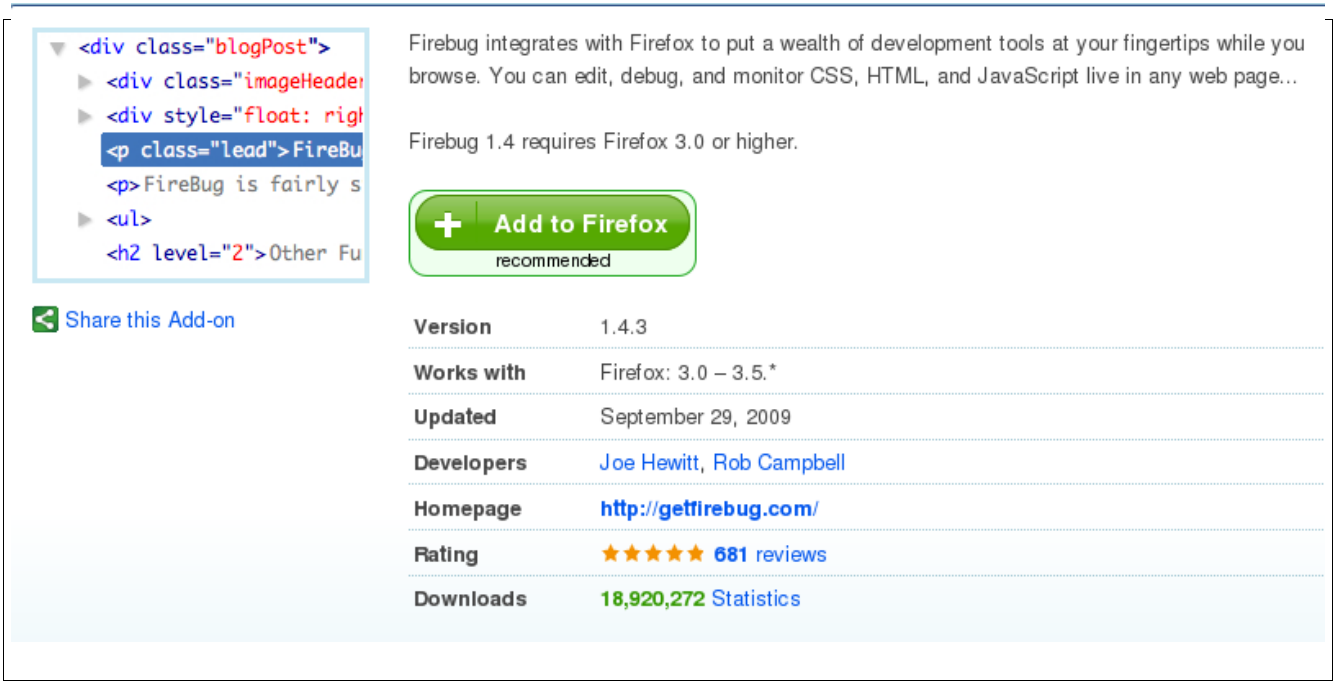


Figure 3-2. Click the Add to Firefox (green button)

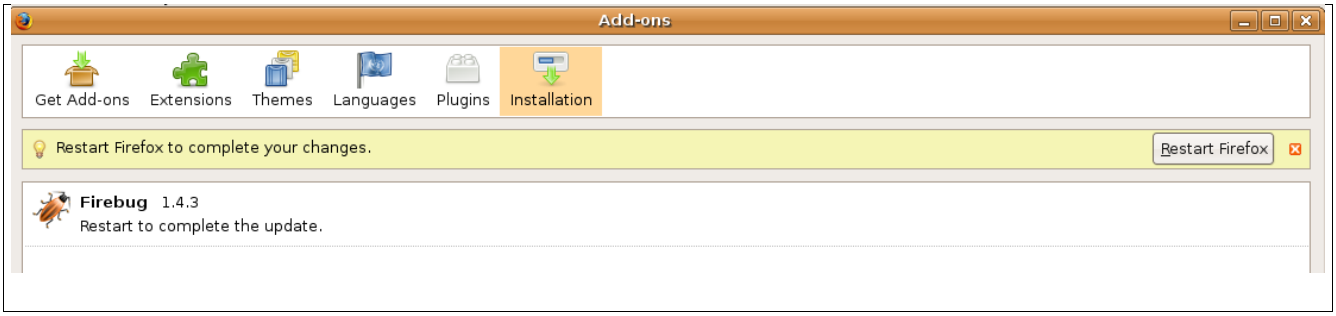


Figure 3-3. Restart Firefox

Upon restarting Firefox, you will then have the Firebug Add-on installed. To actually use it, you have to open it up by either three methods: 1) Tools->Firebug->Open Firebug 2) Hitting the F12 function key 3) Clicking the little bug icon on your status bar (at the bottom of your browser). All of these should present you with something similar to the following where you can verify the console works by typing in a simple log message and clicking the 'Run' at the bottom of the console input pane (it's highlighted in the screen-shot below):

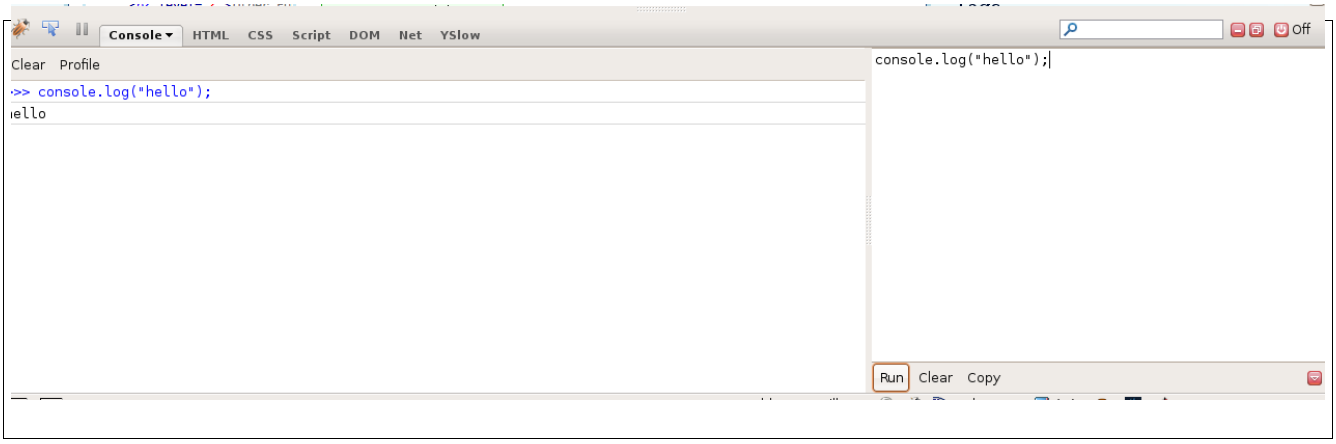


Figure 3-4. Verifying Firebug console with a simple console.log(“hello”) message and clicking the 'Run' button

In the above, you've used the Firebug console interpreter to enter JavaScript directly into the input pane (shown right) and have it output the result of executing that JavaScript (show left). The console.log(“hello”) is simply a call to the log method on the console object with the string message 'hello'. We'll be using this logging functionality throughout the rest of this book so make sure you can duplicate the result before moving on.

2.3 Usage

As this book is aiming at brevity we'll show you just a couple more of the features of Firebug. There's a ton of hidden features that are very useful and I suggest watching the Joe Hewitt video previously cited.

If you go to any web site with Firebug open and click the 'Net' tab, you should see page load information similar to that in Figure 3-5. Note that if you don't initially see anything, you may have to Reload the web page. One thing not shown in the figure is that if you scroll to the button of the Net section, you can determine the total page load time which will show up on the bottom right side. This is very useful when trying to improve the page responsiveness of your site. Perhaps you

have many different includes that are each causing a separate HTTP request and should consider creating one amalgamation file instead? Have you minified your JavaScript? Is my image too big? These are advanced topics but exemplify how you may use the Net tab to improve response time.

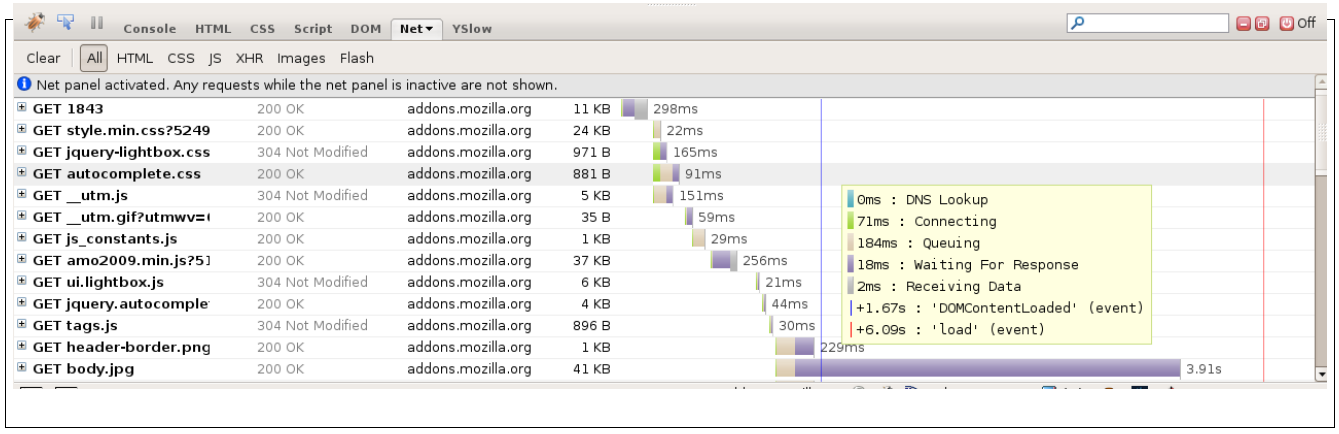


Figure 3-5. Clicking on the Net tab and then reloading a web page. Notice the yellow stat box? That shows up from simply hovering over one of the timeline bars. This allows you to see how long certain parts of the page are taking to load.

Below is a screen-shot showing how to use the inspector to find the exact DOM for a particular item on a web page. Note that the inspector icon (just to the right of the bug) may look differently on your system.

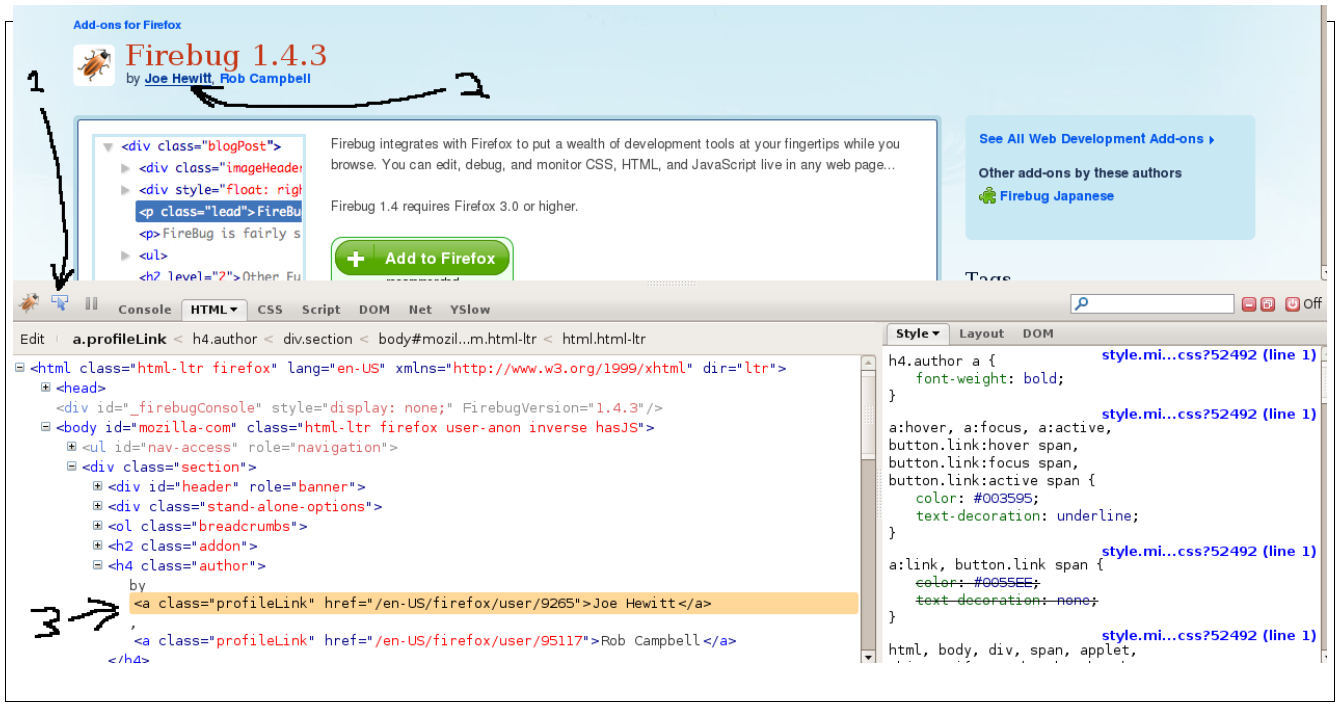


Figure 3-6. Firebug Inspector. 1) We click the inspector icon 2) We click on a page item ("Joe Hewitt") 3) Firebug shows the exact HTML element within the markup!

Note that you can also right-click (or context-click on a Mac) the highlighted HTML element and a context menu will appear in which you can select useful things like Copy XPath. This gives you the valid XPath string you can paste later, which may look something like:

```
//html/body/div/h4/a
```

This will be useful later when we use another tool Selenium IDE. As mentioned, this indispensable tool has much more to offer like breakpoints and step execution. Some of this will be discussed in later chapters as appropriate.

Note that the following chapters (to save space) will not show as much installation detail, and will assume that you can figure out how to download and install the libraries yourself.

Selenium IDE

Contents:

3.1 Introduction

Selenium is a comprehensive suite of software for testing applications from within the context of the browser. It is really a blasphemy to call it a debugging tool as it's so much more! However, it seems to fit nicely here (apologies to those offended!), and we're only going to get into simple usage of the IDE portion of this suite of tools.

Selenium itself, consists of Selenium Core, Selenium IDE, Selenium RC, and Selenium Grid. This book will focus on Selenium IDE which is a Firefox Add-on. The IDE allows you to “record in” your browser interface interactions (eg Open the http://localhost:3000/events page, click the Create New Record button, etc.). You can then “playback” these interactions and you will see the browser responding to these prerecorded interactions (much the way the old player pianos would play a song and the piano keys would move as if an invisible person was playing it).

You can also manually type in such commands (as opposed to recording them). The best way to understand the IDE is to begin playing with it.

3.2 Installation

You can essentially search the Firefox Add-on site for Selenium IDE and install exactly as you did with Firebug. If you've installed it successfully, it should show up in the Tools menu of Firefox.

3.3 Usage

When you first open up Selenium IDE, the record key (the red key on the upper right hand side) will be engaged. This means that you are already in record mode. You can of course toggle this off if you wish, but for the purposes of our discussion, we'll leave it engaged. Simply interact with you web browser (eg open one of your pages, try to carry out a user story, etc.) and then click the record button to turn recording off.

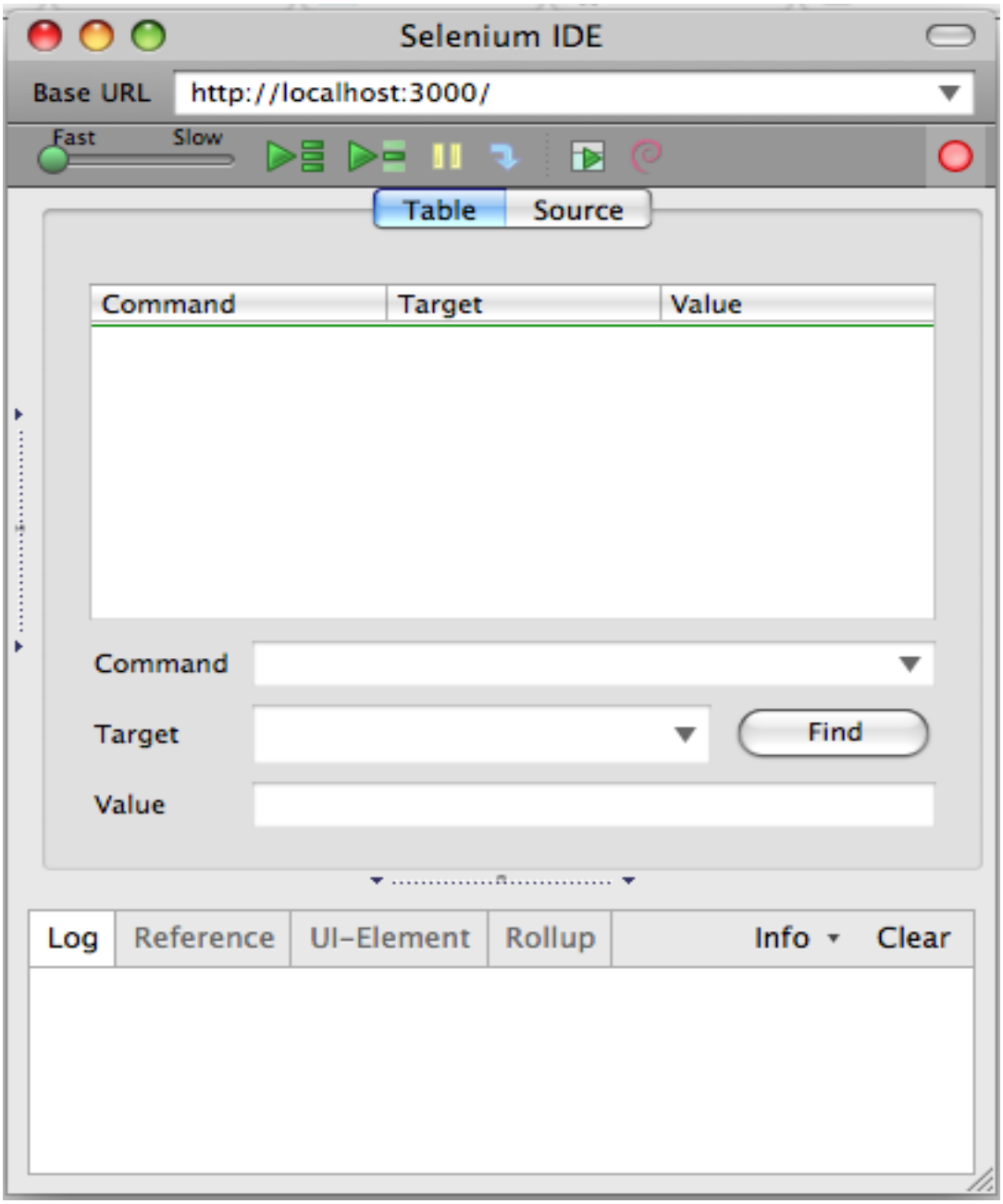


Figure 4-1. Selenium IDE opens up in record mode so you can start recording your browser interactions

TODO:

1. Put in a section which shows creating a test suite for a simple application and put source in /code in repository.

2. Mention something about the problem of having to open the browser, manually running test, alternatives including Selenium RC, possibly Cucumber and Webrat, etc.

Example of a Debugging Session

– Put in an example of using Firebug and Selenium to debug a script –

Part II – Testing Tools

Testing Methodologies

This section (and the rest of this book) will assume that the reader has been exposed to some type of unit testing framework, be it an xUnit variant like Junit, or a BDD variant like RSpec.

xUnit Frameworks

The first testing framework we'll discuss is the YUI Test framework – this is an xUnit testing framework. With xUnit testing frameworks there are *test methods*, *test cases*, and *test suites*. There's also a *test runner* for executing the tests.

- Test methods allow you to test one isolated piece of functionality in your software.
- Assertions are usually one executable line of code that asserts that some condition has some characteristic (e.g. something is true, something is of a particular object type, etc.) Test methods generally have one or more assertions.
- Test cases are one or more test methods cohesively grouped to test one object or module.
- Test suites are just a way of grouping test cases so that they will all run together (just as a musical suite would be a group of songs that are to be performed together).

Usually, you will have one object under test that will map to one particular test suite, with methods of that object being tested individually by test cases. [ZNTDD2008]

In order to run, either an individual test case or a suite of such test cases, a test runner acts essentially as a driver bootstrapping the suite of tests.

BDD Frameworks

BDD tends to emphasize the inclusion of stakeholders by using an *outside-in* process which focuses on the goals of the stakeholder, and then works its way in to the system that supports these goals. It starts by *describing* what a unit of software should do and then makes some assertions about that functionality with the idiom of starting with 'it should' (do such and such).

According to Dan North, the pioneer of this methodology, '*if the methods do not comprehensively describe the behaviour of your system, then they are lulling you into a false sense of security.*' Essentially, his solution is to do away with the testMethod convention, and instead *describe* the desired behavior of the thing being tested. [NDBDDINTRO]

BDD may have some advantages over traditional xUnit in that it's more closely tied to the specific features required by the stakeholders, thus insuring that you only write software you need, but there's nothing preventing achieving similar results from xUnit testing. It really is a matter of

preference and taste.

For a more complete discussion of both xUnit and BDD please see:

http://en.wikipedia.org/wiki/XUnit
http://en.wikipedia.org/wiki/Behavior_driven_development

The jspec framework is a BDD testing framework – a whole chapter is devoted to this framework later in the book. There, we will get deeper into what's involved with using BDD.

YUI Test

Contents:

4.1 Introduction

According to Yahoo's yuitest page located at the time of this writing at:
http://developer.yahoo.com/yui/yuitest
"YUI Test is a testing framework for browser-based JavaScript solutions. Using YUI Test, you can easily add unit testing to your JavaScript solutions. While not a direct port from any specific xUnit framework, YUI Test does derive some characteristics from nUnit and Junit."

It allows for the creation of test cases, test suites, asynchronous testing and more.

4.2 Installation

For our purposes, we'll go ahead and download the includes straight from Yahoo's servers from within our script so we don't have to actually install anything! However, you may wish to download and host the yuitest includes on your server (which I will leave to the reader to do for themselves if so).

If you'd like to optimize your includes, you can use the YUI Dependency Configurator which will allow you to choose the specific YUI components you want to use on your web page. You can also choose to combine all JS files into a single files which will reduce the number of HTTP requests improving your browser page load time dramatically. However, we will not use this tool to configure our setup as it is out of the scope of this document.

Please note that the bulk of this chapter uses YUI Test from YUI 2. The latest version is YUI 3 and there is a move in this direction. Therefore, you can find a bare minimum working example in the github repository for this book. See the last section of this chapter for more details.

4.3 Usage

Getting yuitest to work is fairly straight forward. You basically need to require all of the boiler plate

CSS, dependencies, and yuitest source code. Once you've done that, you will create one or more *test cases*, each of which will in turn contain individual *test methods*, and then add those *test cases* to a *test suite*. Lastly, you will instantiate a *test runner* object and call *run* on that object (essentially bootstrapping the running of the test suite). If you're impatient, you should be able to just “load and go” the source code in the repository under /code prefixed with yui_test

In general, it is advised that for each object you want to test you have ([ZakasTDD2008]):

1. One test case for each method in the object. This test case will contain one or more test methods to test the various aspects of the object method. For example, if you are testing an addTwoNums method, you may one test method that tests if 1 + 2 == 3, another that tests that -1 + 3 == 2, yet another that insures that an exception is thrown if an invalid value is passed in (eg NotANumber), etc. So all of these test methods combined are meant to exercise the one method in the object.
2. One test suite per object. If you have two methods in your object under test, you may have two test cases with one or more test methods, and then add those test cases to a test suite which will effectively test that one object.

The following code creates an HTML file that includes another JavaScript file (as we typically do). Upon loading this into a browser window you should see the YUI test results widget showing your success or failure (please use code from github as opposed to cut and pasting this code!):

yui_test.html

```
<html>
<head>

<!-- Add Logger -->
<link rel="stylesheet" type="text/css"
href="http://yui.yahooapis.com/2.8.0r4/build/logger/assets/logger.css">
<link rel="stylesheet" type="text/css"
href="http://yui.yahooapis.com/2.8.0r4/build/yuitest/assets/testlogger.css">

<!-- Dependencies -->
<script type="text/javascript" src="http://yui.yahooapis.com/2.8.0r4/build/yahoo-
dom-event/yahoo-dom-event.js"></script>
<script type="text/javascript"
src="http://yui.yahooapis.com/2.8.0r4/build/logger/logger-min.js"></script>

<!-- YUI Test Source -->
<script type="text/javascript"
src="http://yui.yahooapis.com/2.8.0r4/build/yuitest/yuitest-min.js"></script>

<!-- Our JS file with the test cases, test suite, and test runner instances -->
<script src="yui_test.js"></script>

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

yui_test.js

```
// Simple test case 1
var nameAgeTest = new YAHOO.tool.TestCase({
```

```
    name: "Test Name and Age",

    setUp : function () {
        this.data = { name : "Rob", age : 39 };
    },

    tearDown : function () {
        delete this.data;
    },

    testName: function () {
        YAHOO.util.Assert.areEqual("Rob", this.data.name, "Name should be 'Rob'");
    },

    testAge: function () {
        YAHOO.util.Assert.areEqual(39, this.data.age, "Age should be 39");
    }
});
// Simple test case 2
var fooNumTest = new YAHOO.tool.TestCase({

    name: "Test Foo Num",

    setUp : function () {
        this.data = { foo : "FOO", num : 123 };
    },

    tearDown : function () {
        delete this.data;
    },

    testFoo: function () {
        YAHOO.util.Assert.areEqual("FOO", this.data.foo, "foo should be 'FOO'");
    },

    testBar: function () {
        YAHOO.util.Assert.areEqual(123, this.data.num, "num should be 123");
    }
});

// Create a test suite and add the two test cases from above
yuiSuite = new YAHOO.tool.TestSuite("YUI Test Suite");
yuiSuite.add(nameAgeTest);
yuiSuite.add(fooNumTest);

YAHOO.util.Event.onDOMReady(function (){

    var logger = new YAHOO.tool.TestLogger();// Create logger
    YAHOO.tool.TestRunner.add(yuiSuite);     // Add Suite to test runner
    YAHOO.tool.TestRunner.run(); // Call run on the runner to run tests
});
```

Upon grabbing the above source code from your github downloaded code, and putting it on your local hard drive, you should be able to go to your browser and enter something similar to the following URL (depending of course on the exact path to your file):

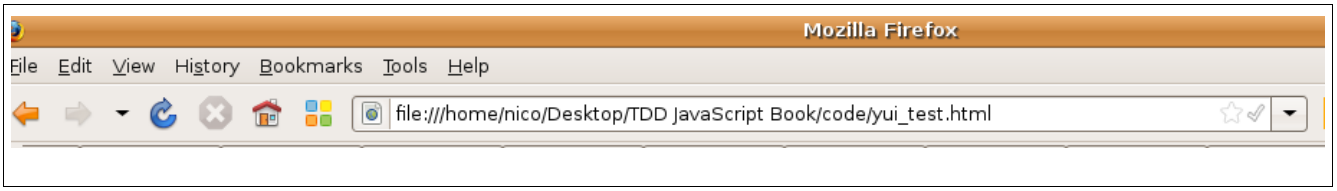


Figure 3-1. Entering a file URL in the browser to test a file on your computer

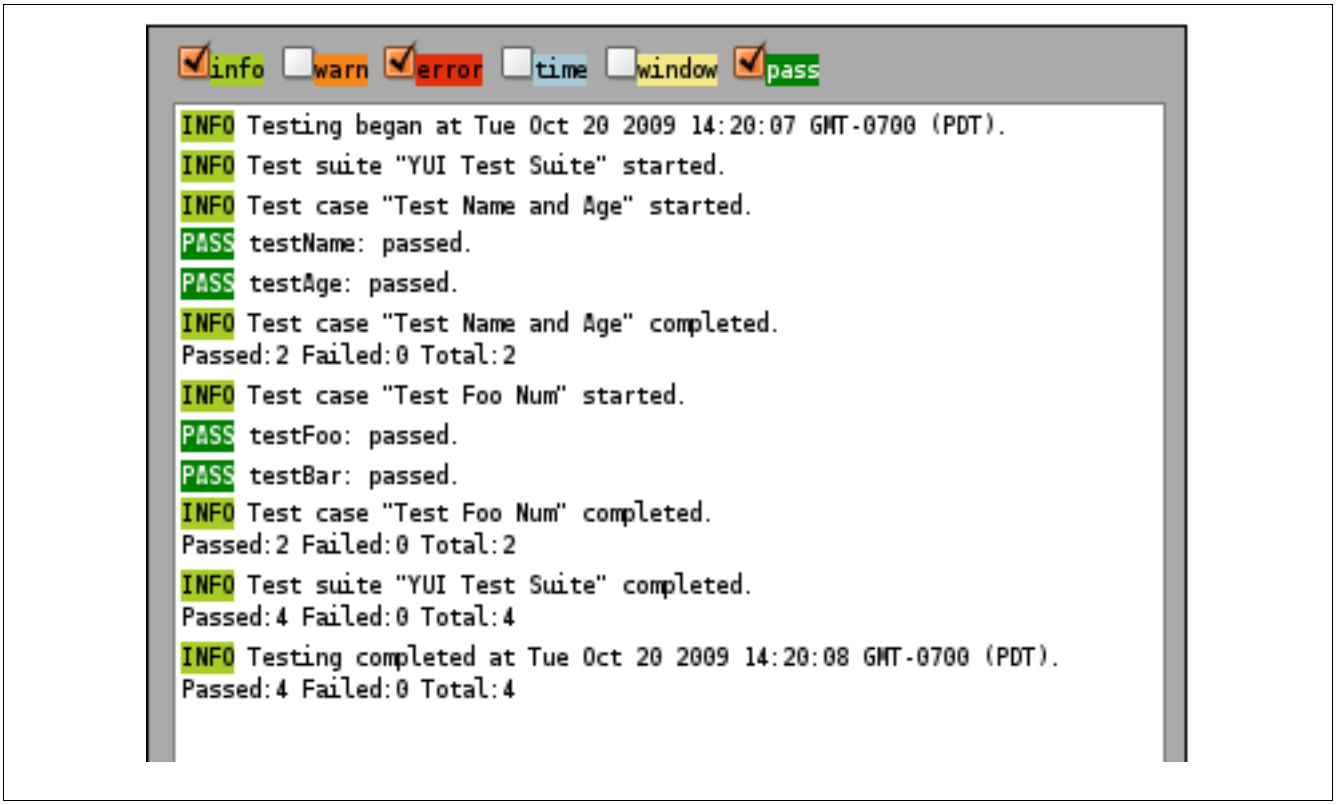


Figure 3-1. YUI test results.

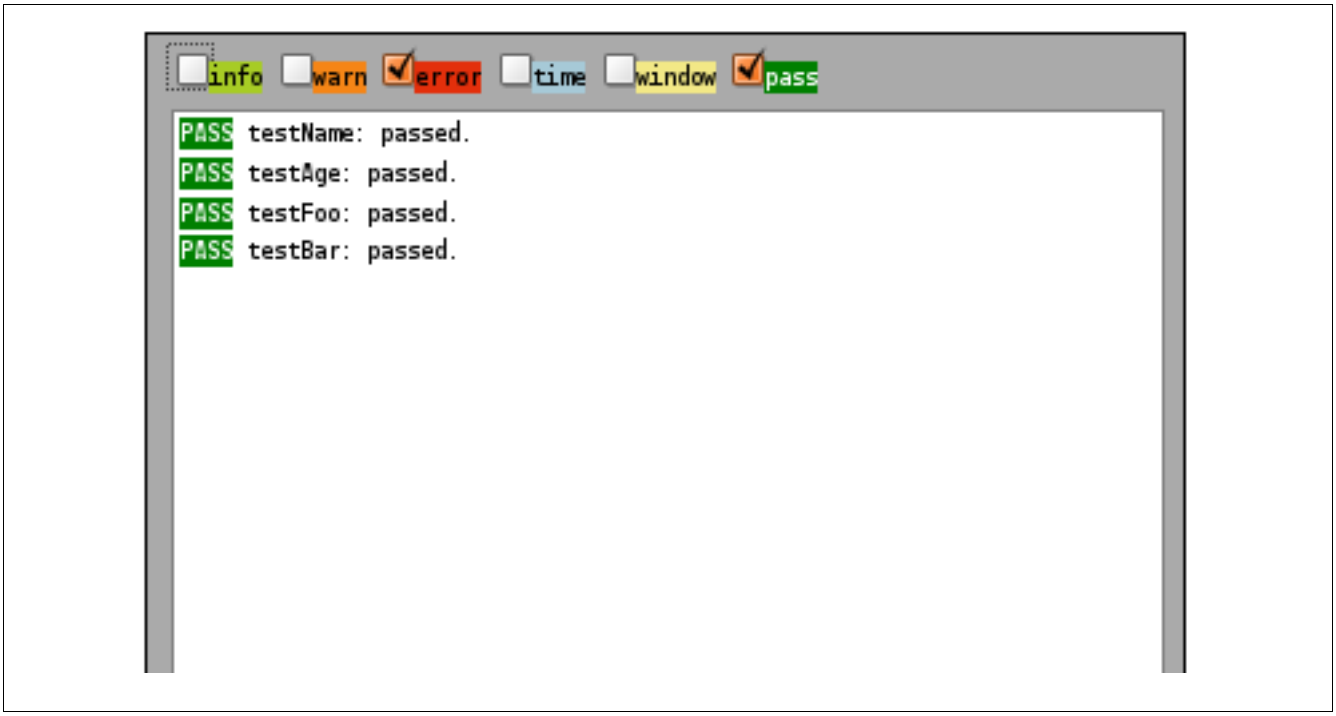


Figure 3-2.Same results with 'Info' unchecked.

If you get it to work you should do something creative to make it “break” so you can also see what it looks like to get a failure.

There are many other options available which you can learn more about yourself from the yuitest web site such as: *AdvancedOptionsTestCase* (which allows you to define tests that should fail, be ignored, or throw an error); *Asynchronous Tests* for testing Ajax; simulation of user interaction *(note that this is a “moving target” between YUI 2 & 3 and you will need to get this from another package – see notes YUI 3 section);* and many more features that are well beyond the scope of this book.

4.4 Setup, Teardown

As you may have noticed that in addition to our tests, there are two additional methods, *setUp* and *tearDown*. Intuitively, one defines and initializes a “data” object on this, while the other deletes it essentially cleaning up. The “data” object is what is called a *fixture* – basically a common resource that will be used for one or more tests. This allows us to avoid repetitiously creating the same object over and over again at the beginning of each method. The setUp and tearDown methods are hooks that will get called automatically before your tests run.

One important thing to note about the setUp and tearDown methods, is that they are called for each of your test methods and not just once for each test case. This insures that any results from one test case do not affect the integrity of following test methods. To prove this fact, I added a simple console.log to the setUp method for a test case that had exactly four test methods within it:

```
console.log("setUp called...");
/* Outputs:
setUp called...tearDown called...
setUp called...tearDown called...
setUp called...tearDown called...
```

```
setUp called...tearDown called...
*/
```
This exemplifies that fact that for each test method within a particular test case, both setUp and tearDown gets called as expected for an xUnit family testing framework.

4.5 Additional Features

Now that you have a working template to add to, I suggest that you create a copy of the file (so you don't clobber your working test), and then experiment with the available YUI Test assertions: http://developer.yahoo.com/yui/yuitest/#assertions

Additionally, if you'd like to have a report of the test run sent to your site, you can enter a URL
```
var oReporter = new
YAHOO.tool.TestReporter("http://you.com/path",YAHOO.tool.TestFormat.JSON);
oReporter.report(results);
```

In the above, you have specified a URL for the first parameter, and the JSON format for the second parameter.

There is other functionality such as asynchronous testing which basically allows you to simulate "waiting" for a specified amount of time before a callback gets invoked. This is again outside the scope of this document.

4.6 YUI 3

YUI 3 has changed the look and feel of the console. As of the time of this writing, YUI 3 is pretty "bleeding edge" but perhaps worth the effort to use. According to the support page at: http://yuilibrary.com/projects/yui3/wiki
'YUI 3.0.0 was made available 9/29/09.'
This is 3 weeks before the author's writing of this chapter and it seemed that there were a lot of various CSS styles rules, and div id's, etc., that were required to get the result widget to ook correct. Of particular note, it appears that the User Actions functionality of *YAHOO.util.UserAction* has moved to the *event-simulate* module. So just be forewarned of this change if you're migrating older tests.

In order to get this working, Yahoo's example code was tweaked a bit and is in this book's github repository under: yui_test3.html and yui_test3.js respectively. Of course, all rights and authorship of this code remains with Yahoo! as it's essentially the same as their example code (please don't sue us Yahoo!). Jump on board and run the code if you wish!

QUnit

Contents:

6.1 Introduction

QUnit is the test framework used by the JQuery project to test their own code-base. However, as with YUI Test, it's just a couple of includes and you can use it with any of your own generic JavaScript code.

The syntax is a bit different in that *tests* are analogous to *test methods*, and *modules* are (sort of) analogous to *test cases*. Within a *test* you specify how many assertions will be ran via a call to *expect* (eg expect(3) would indicate 3 assertions are in the current test). Moreover, the syntax (namely the assertions) is much less verbose than YUI Test. If you get overwhelmed by huge lists of assertion options you may like this. Another stylistic twist is that the assertions use inline functions ostensibly so that the framework can use them as hooks. Modules have *setup* and *teardown* functions which act as expected.

6.2 Installation

To use Qunit you only need to include two files: qunit.js and qunit.css (and use the HTML structure it expects to output the results. As with YUI Test, you can do this by either downloading the respective JavaScript and CSS files mentioned above, and including them from the relative path of your script, or alternatively point your includes to their repository location (currently github). For our purposes, we'll just point to the repository.

6.3 Usage

QUnit takes a “meat and potatoes” approach to assertions with three of them: *ok*, *same*, and *equals*. The following is the basic API for these:

```
ok( bool_expression, message )
equals( actual, expected, message )
same( actual, expected, message )
```

Basically, ok is the same as an xUnit assertTrue, equals compares to assertEquals. The last one, same, is interesting in that it can do a 'deep recursive comparison assertion' for arrays, and objects, passing if actual and expected are “meaningfully equal”.

One unfortunate thing I noticed was that when I commented out a failing test but forgot to decrement the expect call, it just silently continued to failed without indicating the discrepancy between expect and the actual number of assertions:

```
test("some other test", function() {
  expect(2); // silently fails without indicating that there's only 1 assertion!
  //equals( true, false, "failing test" );
  equals( true, true, "passing test" );
});
```

If I switch the above expect to 1 it worked - one would think that the failing test would indicate the discrepancy.

The following is an example of a complete if simple test (don't paste this code – use the repo version instead!):

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
                    "http://www.w3.org/TR/html4/loose.dtd">
<html>
<head>
```

```
  <script src="http://code.jquery.com/jquery-latest.js"></script>
  <link rel="stylesheet"
href="http://github.com/jquery/qunit/raw/master/qunit/qunit.css" type="text/css"
media="screen" />
  <script type="text/javascript"
src="http://github.com/jquery/qunit/raw/master/qunit/qunit.js"></script>

  <script>
  $(document).ready(function(){

    test("In the spirit of Rails scaffolding, true is true ;-) ", function() {
      expect(2)
      ok( true, "true is true alright!" );
      var artist = "David Bowie";
      equals( "David Bowie", artist, "We expect artist to be 'David Bowie'." );
    });

    test("Meaningfully equal objects are the 'same'", function() {
      var actual = {name : "Rob", age : "I'd have to kill you!", happy : true};
      same(   actual,
            {name : "Rob", age : "I'd have to kill you!", happy : true},
            "Separate but meaningfully equal object are considered the 'same'.");
    });

    test("Meaningfully equal arrays are the 'same'", function() {
      var actual = [1,2,3,4,"five"];
      same( actual, [1,2,3,4,"five"], "Separate but meaningfully equal arrays are
considered the 'same'.");
    });

    module("My Module");

    test("Module Test 1", function() {
      ok( true, "This bettor pass!" );
    });

    test("Module Test 2", function() {
      ok( true, "So better this!" );
    });

  });
  </script>

</head>
<body>
 <h1 id="qunit-header">QUnit example</h1>
 <h2 id="qunit-banner"></h2>
 <h2 id="qunit-userAgent"></h2>
 <ol id="qunit-tests"></ol>
</body>
</html>
```

As you can see, it's pretty darn straight forward to get some QUnit testing going! You can grab the above template from the github repository under: code/qunit_test.html (JavaScript was in-lined in this case but you'll want to go ahead and separate it from the HTML in practice).

The results of running a failing test will show something like the following:
1. my test (2, 2, 4)

The first number indicates the number of failed assertions, the second the number of passed assertions, and the third the total number of assertions. If you're wondering where the messages are, you have to click the particular failing test and you'll see a nice verbose message.

QUnit also has functionality for testing Ajax similar to *wait* and *resume* in YUI Test but calls them respectively *start* and *stop*. Again, this is beyond our scope, so refer to the online docs for information on how to implement this if you need it.

6.4 FireUnit

FireUnit is an extension to Firebug which provides support for logging your tests into a new tab of Firebug called, appropriately enough 'Test'. As of the time of this writing it's still in the experimental phases. If you're like the author and love Firefox Addons and Firebug, you may want to give this a try. To install:

- http://github.com/jeresig/fireunit and click the Download button at the top (or do a git clone if you prefer)
- Navigate to the directory you've placed this in and simply type `make`
- This will build the .xpi file that Firefox uses as a plugin installer. Therefore, within Firefox do File → Open File and load the previously created .xpi file (fireunit-1.0a3.xpi at the time of this writing but likely to change as they keep updating the code).
- Open Firebug (this requires Firebug of course!) and look for a 'Test' tab. Note that one of the plugin authors suggested a version of Firebug earlier than what we tested on (Firebug 1.4.3) but it seems to work for us so just fine.
- Run the following code in your browser (example in the github repository under: /code/fireunit_test.html)

```
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
<script>
fireunit.ok(true, "Passing test result");
fireunit.ok(false, "Failing test result.");
fireunit.compare("expected data", "expected data",
    "Passing verification of expected and actual input.");
fireunit.compare("<div>expected</div>", "<div>actual</div>",
    "Failing verification of expected and actual input.");

// Wait for asynchronous operation.
setTimeout(function(){
    // Finish test
    fireunit.testDone();
}, 1000);
</script>
</head>
<body/>
</html>
```

Now in Firebug you should initially see something similar to the following:

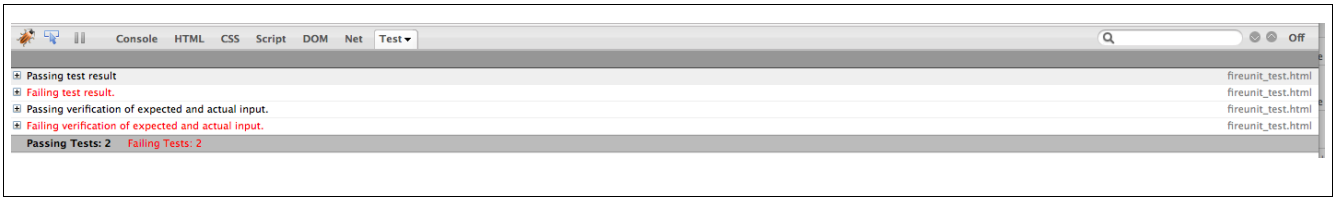


Figure 6-1. FireUnit starts out like above.

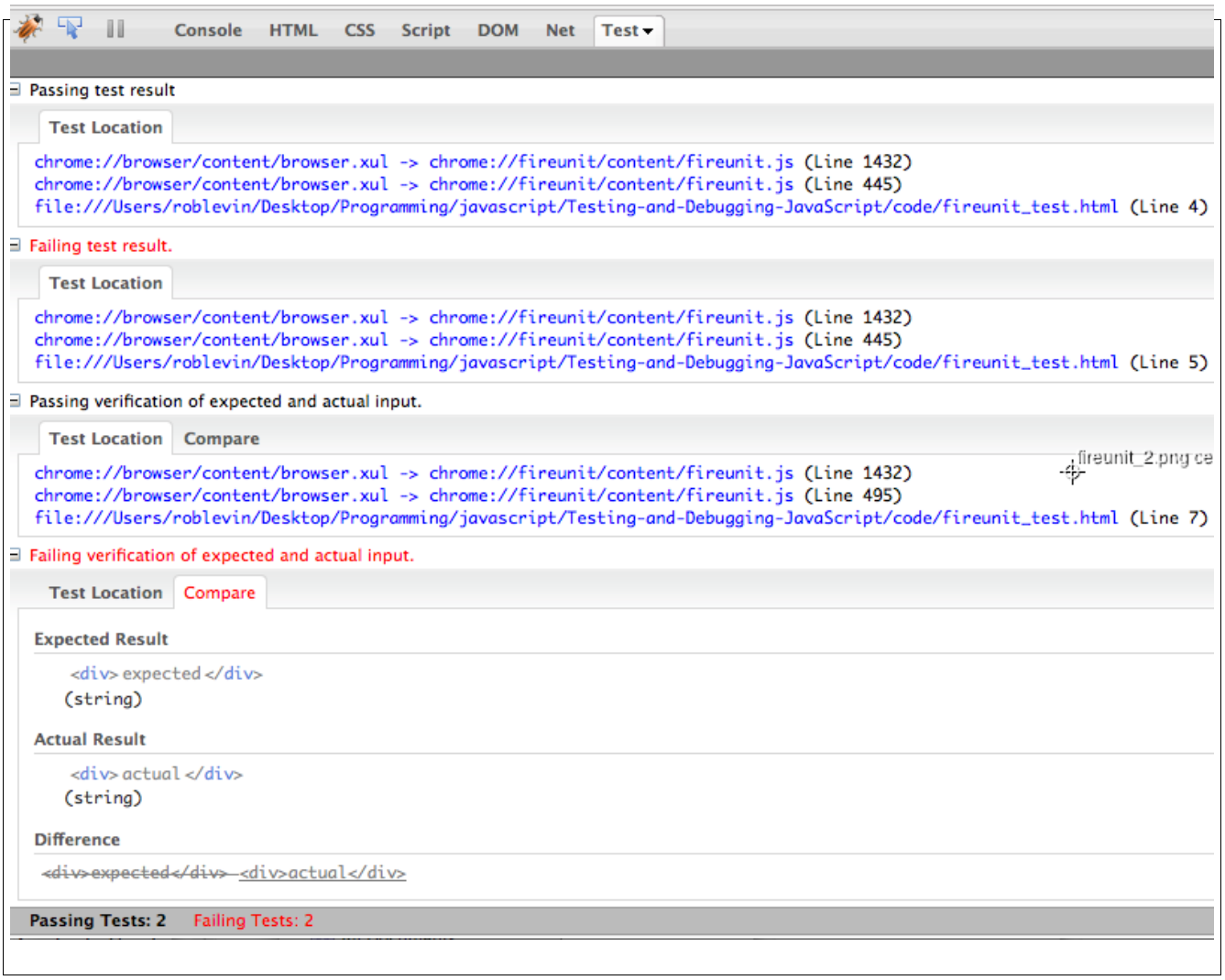


Figure 6-2. FireUnit with more detailed information after being expanded. Note the bottom test – the Compare tab has been clicked and very useful 'Difference' data is supplied.

One of the authors, John Resig, blogged about FireUnit here:
http://ejohn.org/blog/fireunit/

Jan Odvarko, one of the other authors blogs here:
http://www.softwareishard.com/blog/firebug/fireunit-testing-in-the-firebug-world/

You can actually integrate FireUnit with QUnit if you already have a test suite in the later. However,

you don't get the “sexy” comparison diff described above. In any event, here's the hook you need to put in your QUnit code:

```
if( typeof fireunit === "object" ) {
    QUnit.log = fireunit.ok;
    QUnit.done = fireunit.testDone;
}
```

The FireUnit extension is still very much in it's beginning stages, but proves to be an exciting additional tool for JavaScript developers!

jspec

Contents:

7.1 Introduction

Jspec is a BDD testing framework that has much in common with the Ruby BDD framework RSpec. If you are a Ruby developer coming from using RSpec (or any other BDD framework), you'll feel at home with this framework. If not, and you're in a hurry to get going, you may want to skip this chapter and come back when you're ready to experiment with BDD.

– Does this work on Linux too or just Macs??? NEED TO TEST!!!

At the time of this writing, the http://visionmedia.github.com/jspec/ site listed several screen-casts of the author, TJ Holowaychuk, which were quite helpful in getting an overview of the tool. In the screen-cast about jspec's grammar, the he states that “it is not JavaScript, but JavaScript runs it”. In fact, it looks a bit like a combination of JavaScript and Ruby.

If this bothers you, you can edit the spec.dom.html file and force it use just JavaScript. If you want to do this, please see the documentation as this is out of the scope of this document.

7.2 Installation

In order to install jspec you will first need to install the gem command line utility. If you're a Ruby developer you already have this. If not, you can get directions for installing it here: http://docs.rubygems.org/read/chapter/3

Once you have the gem utility installed, you can issue the following command to get the jspec gem:

```
$sudo gem install visionmedia-jspec --source http://gems.github.com
```

Then you can verify it's installed with:

```
$ jspec --version
// On the authors system outputs: JSpec 2.11.2
```

Alternatively, you can manually include it from your JavaScript program, but we will use the gem for this document. You will also want to turn on the developer debugging in Safari if you plan to use it on a Mac (it defaults to opening up the results in whatever browser your environment defaults to using – but this can be overiden with the --browsers option discussed a few pages later):

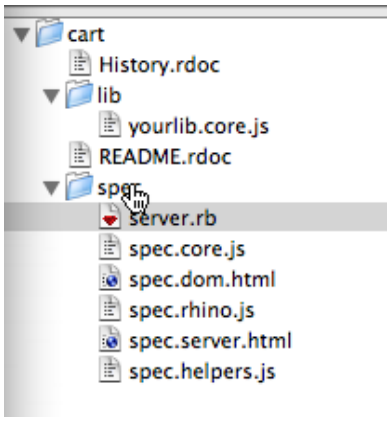
```
$ defaults write com.apple.Safari IncludeDebugMenu 1
```

7.3 Usage

Jspec has a whole different way of developing in comparison to the other frameworks we've looked at so far. You actually use it to generate a minimal set of project directories a bit like you would if generating a Rails project.

```
$jspec init PROJECT_NAME
$cd PROJECT_NAME
```

If you open the created project directory you will see something similar to the image on the right:



To start the jspec environment type:

```
$jspec run
```

On the author's MacBook, this automatically opened up the Safari browser as seen in Figure 7.1 below:

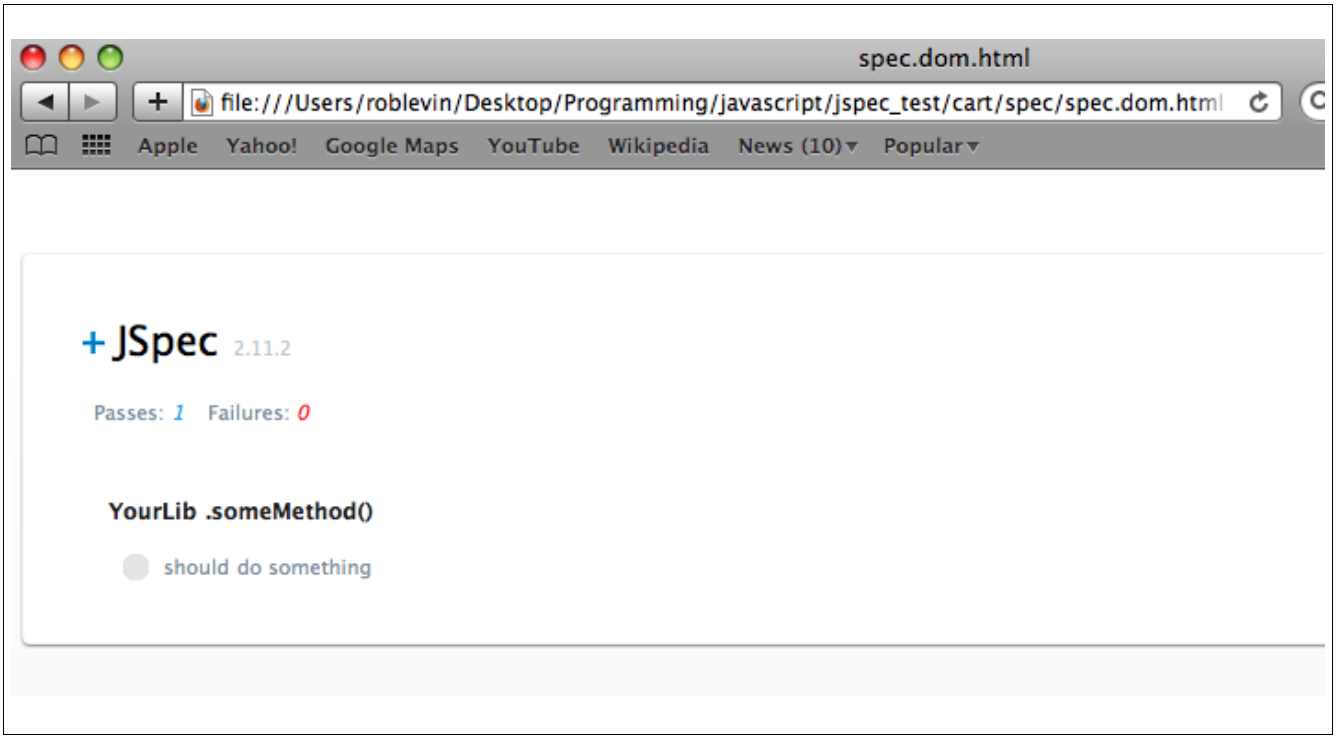
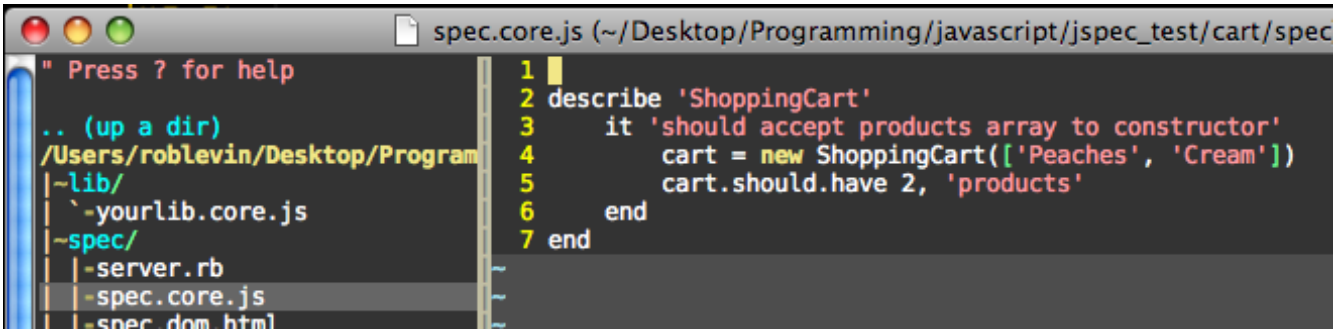


Figure 7-1. jspec run will automatically open Safari to point to the spec.dom.html as above.

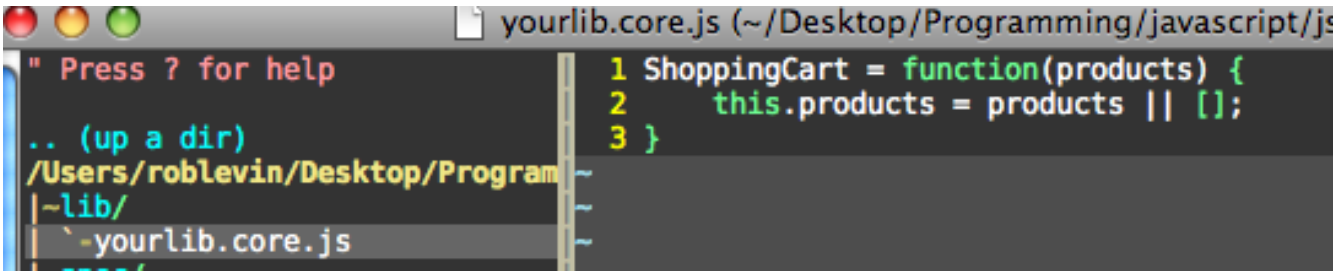
This is showing a passing test because we just left the boiler plate default test. Let's do one iteration of a BDD cycle (from jspec's author's screen-casts):

1. First we generate a test in the spec_core.js file (note that you can create other spec files, but we'll go ahead and use the default by deleting the boiler plate code and adding our own as follows):

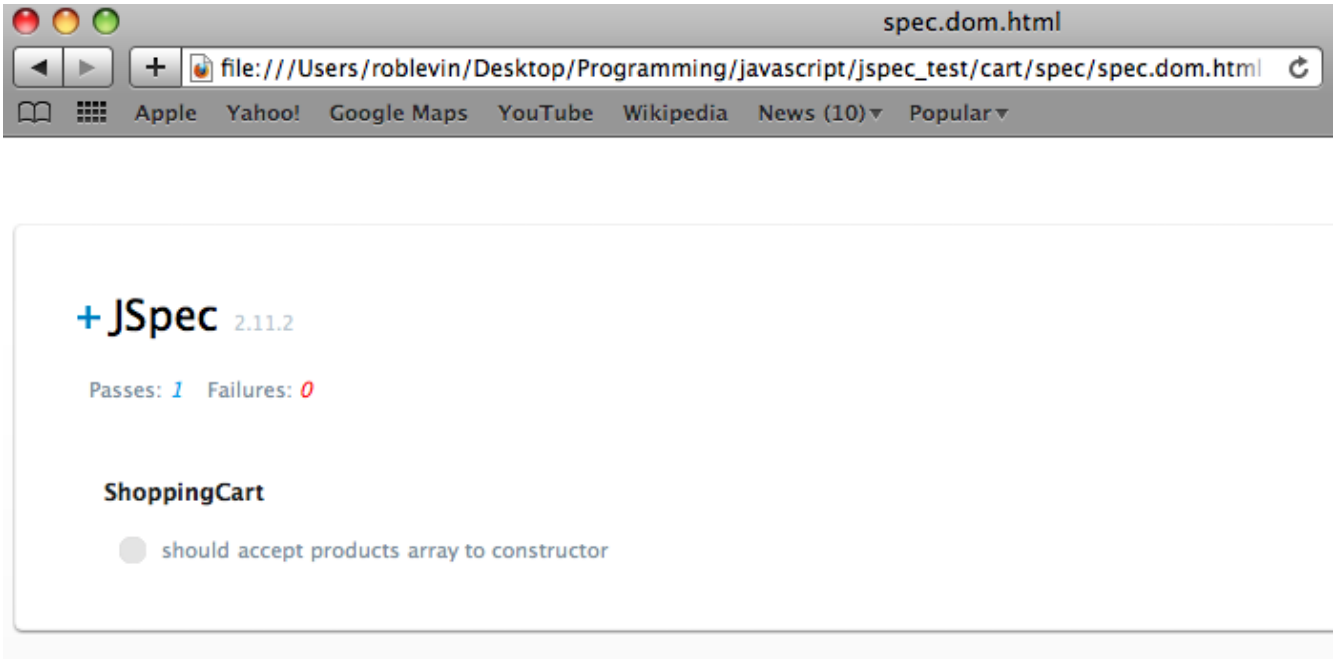


2. Reload the Safari browser and see the failing test (we haven't implemented anything yet!):

3. Implement the minimal code to make this pass:



4. Reload the browser once more and see it pass:



This is essentially the same introduction that the jspec author gives in the screen-casts which you might previously mentioned.

These steps show the basic flow of BDD development. To quote the official RSpec web site (an infamous Ruby BDD framework) the following steps outline an iteration of BDD:

1. *Start with a very simple example that expresses some basic desired behavior.*
2. *Run the example and watch it fail.*
3. *Now write just enough code to make it pass.*
4. *Run the example and bask in the joy that is green.*

[RSPECBDD]

The idea, is that you continue to do the above steps as you add on features, fix bugs, etc.

7.2 Additional Features

The following are some of the additional features available in jspec. For more details see the official site and/or the very detailed README file.

Callbacks

Just as the xUnit derivatives used setup and teardown, jspec has before and after hooks. There are also similar callbacks such as before_each and after_each which give you finer grained control as you'd expect. See documentation for more details on the hooks available (or better yet, the jspec.js source code itself which is pretty concise and easy to browse).

Options

It also allows you to add the '--browsers' switch to test in multiple browser:

$jspec run --browsers ff,safari

This will open up the respective browsers on your system if they're installed. You can also run the '--server' switch to see the passes and failures in your console (if you don't have a particular browser installed it will tell you it couldn't find it):

```
$ jspec run --browsers ff,chrome,safari --server
Started JSpec server at http://0.0.0.0:4444
Unable to find application named 'Chromium'

 Firefox Passes: 0 Failures: 0
 Safari Passes: 0 Failures: 0
```

If you leave out the browsers switch it will attempt to open all and again inform you for each one that you don't have:

```
$ jspec run --server
Started JSpec server at http://0.0.0.0:4444
Unable to find application named 'Chromium'
Unable to find application named 'Opera'

 Firefox Passes: 0 Failures: 0
 Safari Passes: 0 Failures: 0
```

Note that you'll need to do a Ctrl-C to end the process.

Server Formatting

You can fine tune the server settings in the spec.server.html and server.rb files. So for example, if you'd like less verbosity in your test runs you can edit the following line in spec.server.html:

```
.run({ formatter: JSpec.formatters.Server, verbose: true, failuresOnly: true })
```

to:

```
.run({ formatter: JSpec.formatters.Server })
```

and your output will be more concise. Additionally, if you have rhino installed on your system you can use the --rhino switch to run your tests through rhino.

Other Tidbits

- *You can also test for a throw from within a test case as in:*
  `function(){ throw 'some_error' }.should.throw_error 'some_error'`
- Fixtures: If you need a DOM document to test on, you can create a fixture and then load it in your before or before_each hook.
- The above fixtures can be in HTML, json, XML, etc. See documentation for details.
- Not shown on the screen-shots is the rightmost side of our browser window when running the jspec tests. On our system, it shows vertical lines or “pipes”, each representing an assertion within that test case; so if you made two assertions it would show something like '||', three '|||', etc., on the right side of the jspec results page.

Summary

In this section we've taken a look at 4 testing frameworks. YUI Test, QUnit, and FireUnit are of the xUnit derivative, whereas jspec is a BDD framework much in the spirit of RSpec. There are many more JavaScript testing frameworks available. Some of the frameworks which we would have loved to included had this document been a “JavaScript testing tome”:

- JSUnit (been around for a long time and used by many projects)
- JSSpec (used by the MooTools framework)
- unittest.js (script.aculo.us and Prototype)
- JsUnitTest (based off Prototype's unittest.js)
- D.O.H. (Dojo Objective Harness)
- screw.unit (another BDD style framework)
- Jasmine (a Pivotol Lab's BDD framework)

As far as determining which one of these frameworks is right for your projects, the first thing to figure out is if you're going to go with xUnit or BDD style testing. If you're coming from a Java shop that's been using JUnit in Eclipse for ages, and have no intentions on ramping up on 'this BDD thing', than of course you've already made your decision. However, if you're a Ruby developer, it's pretty hard to escape the fact that all the “cool kids” have embraced BDD - you may want to keep things somewhat familiar by using BDD in your JavaScript testing framework. If you're of the “non-religious” derivative of programmers out there, play with both and see which feels more comfortable to you. Whichever you use you'll be better off than you were just cutting code and releasing!

Unit Tests vs. Regression Tests

One thing that you should be cognizant of, is the difference in what you gain from unit tests versus integration tests (also called acceptance tests or regression tests) - we are purposely blurring the lines between acceptance, regression, and integration testing as they are quite similar by nature but in fact different. For our purposes we'll use the terms interchangibly.

Deferring to Wikipedia:
Unit testing: 'a software verification and validation method in which a programmer tests if individual units of source code are fit for use. A unit is the smallest testable part of an application...'

Regression testing: 'any type of software testing that seeks to uncover software regressions...Typically, regressions occur as an unintended consequence of program changes. *Common methods* of regression testing include rerunning previously run tests and checking whether previously fixed faults have re-emerged.'

It's important to note that, although some regression-benefit may be derived from old unit tests, this is usually not the case. For example, you have a public API that must never change, but for what ever reason, you decide to “re-gut” the entire internal implementation. However, you maintain that you must not change your public API. Since you are completely changing your internal system, your unit tests may not help you test the public API at all – they probably just tested you're implementation. HIf you have a regression suite, you can benefit by '*rerunning previously run tests*' to insure that your new system maintains it's public interface.

In fact, Merb is a notable piece of software (now merged into Ruby on Rails) for which this was exactly the case. [KYCTDS2008] Yehuda Katz, the lead developer and presenter of the presentation

in the following link gives a very pragmatic view on the benefits of unit and integration testing: http://rubyconf2008.confreaks.com/writing-code-that-doesnt-suck.html

If this diversion has scared you, be comforted by the fact that this book will primarily use “vanilla unit tests”. Note that we will be primarily using jspec for the upcoming chapters.

Part III – Test Driving JavaScript

It turns out that unit testing is a great way to quickly grok a programming language. If an interpreter is available for language as well you may want to combine these tools by:

- using the interpreter for immediate feedback when “staying in the groove” is of the utmost importance (I.e. while you're working on a real project).
- using unit tests to explore the edges of a particular language so that you have an “automated recording” of your findings.

For JavaScript, we've already discussed a very nice interpreter – the Firebug console!

Before we get started

Note that the examples will use jspec for this section by the author's choice. If you have chosen another framework like QUnit or YUI Test, feel free to convert the examples to use that framework instead. Rest assured that, the following tests will be simple pedantic examples so transposition should be trivial.

Much of the object creation and inheritance sections can be used as supplemental study to the topics provided in Chapter 6 of Professional JavaScript for Web Developers, 2nd Edition by Nicholas C. Zakas, and Chapter 3 & 5 of JavaScript: The Good Parts by Douglas Crockford. Both of these chapters go into some interesting but quirky details of JavaScript's object creation and inheritance models. But first we'll start with some very basic JavaScript...

5 Minute Drill

Let's dive into a couple of JavaScript basics to get our bearings. If you're a guru you could skip this section but you may want to skim just to see the general concept of *grokking* a language via unit tests.

Getting things set up is always a challenge so here's the first steps to getting tests going using jspec (again, if you'd like to use one of the other frameworks that's fine too):

1. *jspec init*
2. *Then open the created project in your IDE or editor of choice*
3. *jspec needs to be able to find your 'system under test'. We changed that to be /lib/test_js_itself.js (instead of /lib/yourlib.core.js which we deleted)*
4. *Run 'jspec run --browser ff' – from the console for each test run*

spec.core.js

```
describe 'TestJsItself'
   before_each
        jsTest = new JSTest
   end
   describe '.switchTest()'
```

```
      it 'should return corresponding case primitive int as a string'
        jsTest.switchTest(1).should.eql "one"
      end
  end
end
```

test_js_itself.js

```
JSTest = function() {}
JSTest.prototype = {
    constructor: JSTest,
    switchTest : function(c) {
        switch(c) {
          case 0:
            return "zero";
            break;
          case 1:
            return "one";
            break;
        }
    }
}
```

Running the above with jspec passes. If it's not obvious from context, we are testing the basic functionality of the switch statement in JavaScript. In the above, the *system under test* (idiomatic for the object you're testing) is the test_js_itself.js program, and the describe '.switchTest()' line is a test case that tests the switchTest() method. The 'it should' is an assertion. The .eql is a BDD “matcher”.

In the QUnit framework, a transposition of the above test case might look something like:

```
    test("Returns corresponding case primitive int as a string", function() {
      jsTest = new JSTest();         // likely moved to setup
      actual = jsTest.switchTest(1); // calls the 'sut' system under test
      same(actual, "one", "Returns corresponding...as a string"); // assertion
    });
```

Note that we did not test the above code so consider it pseudocode! Other xUnit family frameworks might use something like assertSame or assertEquals.

By writing both the test case for switchTest and the method implementation of switchTest itself, we've jumped ahead of the TDD process – normally we would: 1) write the test 2)watch it fail 3) implement the code 4) rerun the test and watch it pass.

So we've confirmed that we understand the basics of how a switch statement works in JavaScript. But how about the venerable *fall-through*. We hypothesize that omitting a break means that it will fall-through to the next case. Moreover, we've heard that JavaScript allows us to combine primitives and reference types in the case statements. We add the following in bold to our test:

```
describe '.switchTest()'
  it 'should return corresponding case primitive int as a string'
      jsTest.switchTest(1).should.eql "one"
  end
  it 'should fall through to next case when omitting a break'
      jsTest.switchTest("no break case") "next case"
```

```
   end
end
```

Running this I get an error:

TypeError: actual is undefined

Although it's a bit hard to tell from that message, we know that we haven't even defined the corresponding case in our switch statement, so jspec is choking. We add the bold section to our switch:

```
   switch(c) {
        case 0:
           return "zero";
           break;
        case 1:
           return "one";
           break;
        case "no break case":
           // do nothing...testing fall through
        case "next case":
           return "next case";
           break;
   }
}
```

It passes. We've proved that fall through works as expected (and that we can mix and mash Number types with String types, etc.) 'Big deal' you say. Hold on, things will get more interesting soon!

Are we confident enough in creating simple vars? For example:
var x =1; var y = 2; result = x + y; // result == 3 Of course we are. No need to test that!

Let's have a look at creating Arrays. We'll check for creation and that indexing is zero based. So we add a test:

```
describe '.arrayTest()'
    it 'should create an array and index by zero based subscripts'
      arr = jsTest.arrayTest()
      (arr.class == 'Array').should.eql "true"
    end
end
```

When we run jspec we get the following error:
TypeError: jsTest.arrayTest is not a function

Great! We've created a test, watched it fail (because there's no implementation yet) – we're doing TDD development! So we add the following remembering to place a comma after our switchCase method:

```
arrayTest : function() {
    return [1,2,3,4,5];
}
```

it fails: *expected false to eql 'true'*

Ah right, we accidentally tried using .class because we were thinking in Ruby not JavaScript – ouch! So we remember that there's a typeof operator in JavaScript for this sort of thing. Before going back to the test, we open up Firebug because we're a bit unsure of ourselves and need some instantaneous feedback:

```
typeof [1,2,3] == "array";  // false, huh???
arr = [1,2,3];
typeof arr == "array";      // false, what the heck?
arr = new Array(1,2,3);
typeof arr == "array";      // false
typeof [1,2,3] == "object"; // true, oh yeah Crockford said something about this!
```

Upon doing this of course we realize that JavaScript returns “object” for Arrays and recollect Douglas Crockford mentioned something about this. Sure enough we Google: 'typeof javascript array' and his page comes up first:
http://javascript.crockford.com/remedial.html

Ok, so we'd already heard about this problem but still forgot – oh the shame! How can we make sure we don't forget next time? Well, now that we know the rule lets change our test. In fact, let's break this up into two cohesive tests – one to confirm that typeof array returns “object” and another for our original claim that array's in JavaScript use zero based indexing:

```
...
    describe '.arrayTest()'
      it 'should return object for typeof array because JS is weird in that way'
          typeof arr == 'object' // uh oh! No matcher should.eql
      end
      it 'should use zero based indexing'
          arr[1].should.eql 2
      end
    end
...
```

Interestingly, jspec now shows that we have three of the tests passing out of four - jspec shows that something's went wrong with the third test. Right, we forgot to use a *matcher* (eg should.eql). It doesn't explode but the test doesn't end up doing anything. So we change the bold line from above to read:

```
          (typeof arr == 'object').should.eql true
```

We run jspec and we get: 'Passes: 4' – cool!

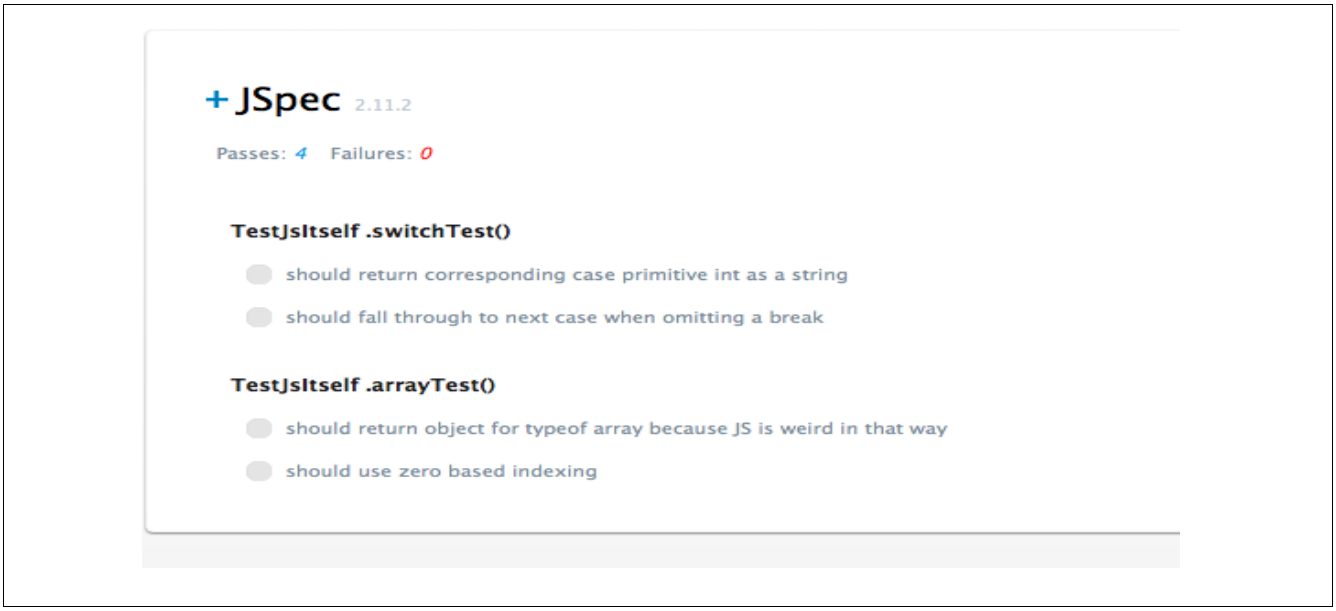


Figure 8-1. We've used TDD development to test JavaScript's switch statement and Array data structure. After some initial misconceptions, we've managed to finally make all the tests pass!

We've only tested a couple of areas of JavaScript but we've already cataloged an interesting JavaScript oddity – the way typeof returns 'object' on Arrays (it does this with nulls too!). Next we could exercise some of the Array methods like sort, push, etc., just to confirm our understanding. This would take just minutes but our tests would live on for later reference. Here's a helpful page for doing just that:
https://developer.mozilla.org/en/Core_JavaScript_1.5_Reference/Global_Objects/Array

What just happened?

The above exemplifies the process of using test driven development to confirm a language's rules and syntax. The mistakes and fumbles along the way were left in because they realistically show the mindset used to deal with road blocks.

Two other popular ways of experimenting with a language are: using an interpreter for instant feedback (useful in it's own right), and writing small test programs. Small test programs are not very effective to aid in recalling concepts later – you end up with either a rats nest of small programs which become hard to later find, or one jumbled file virtually impossible to navigate.

As unit tests are, by design, meant to be self documenting, they will provide a means for future reference that won't “hurt your eyes”. If you want to be able to grep the files later, there's nothing stopping you from adding easy-to-grep comments like /* ARRAY_TESTS */ allowing you to do:
`grep -A 10 'ARRAY_TESTS' my_tests.js`
to get the 10 lines after the matching ARRAY_TESTS section.

Next we look at some interesting areas of JavaScript that the author encountered reading both the Zakas and Crockford books – namely Object Creation and Inheritance.

Objects

Contents:

This chapter will deal with JavaScript objects and will use unit tests confirm our understanding of how they work. The author will examine interesting topics from Chapter 3 & 5 of JavaScript: The Good Parts – Douglas Crockford, and Chapter 6 of Professional JavaScript for Web Developers, 2nd edition – Nicholas Zakas.

Object Creation

So let's get right into creating objects with object literals. Our test:

```
describe 'JSObjects'
    before_each
      jso = new JSObjects();
    end
    describe '.objLiteralCreate()'
      it 'should create an object literal'
          (typeof jso.objLiteralCreate() == 'object').should.eql true
      end
    end
end
```

We run our test in jspec and of course get the fail as there's no implementation:

Figure 8-2. Step two in TDD is to run your unimplemented test which will inevitably fail!

So we implement the following in our system under test:

```
function JSObjects(){};
JSObjects.prototype = {
    constructor : JSObjects,
    objLiteralCreate : function() {
      return {
          name : 'Rob',
          age : 39
      }
    }
}
```

And we pass:

Figure 8-3. Steps 3 & 4 our respectively: implement the feature, watch it pass

There's a bit of extra stuff going on in that implementation but it essentially does the following:

```
function JSObjects(){};
```

Creates a constructor for our system under test JSObjects.

```
JSObjects.prototype = …
```

JavaScript is a prototypal language meaning that it provides a “prototype linkage” allowing one object to inherit properties and methods from another. [CDJGS20] Suffice it to say that this is preferable to adding methods directly into a constructor which would cause method objects to be created for each instance.

Our return statement creates and returns an *object literal* which is essentially a grouping of *name : value* pairs within brackets {} where the names can represent either properties or methods. This notation is very convenient and visually helpful. However, we could have also done this using the new Object() constructor and dot notation like so:

```
o = new Object();
o.name = 'Rob';
o.age  = 39;
return o;
```

In fact, we tested this by commenting out the object literal return statement and adding the above.

Accessing Object Properties

JavaScript can use either dot or subscript notation to access properties:

```
describe 'Accessing Object Properties'
  it 'should be able to use either dot or subscript notation for access'
      o = {name:'Rob', age:39}
      o.name.should.eql 'Rob'
      o['name'].should.eql 'Rob'
  end
end
```

Note that we did something a bit different here – instead of calling a method on our *system under test*, we just tested the feature from right within our test case. Since we're simply testing language features this is perfectly acceptable.

Object Assigned by Reference

Objects are assigned by reference and not copied [CDJGP22]:

```
describe 'Object Assignment'
  it 'should reference original object and reflect any changes made to
original'
      o = jso.objLiteralCreate();
      var copy = o;
      copy.name.should.eql 'Rob'
```

```
          o.name = 'Charlie'
          copy.name.should.eql 'Charlie'
      end
    end
```

So once we change the original we see the changes in the variable that was assigned to ('copy' is probably a bad variable name here because we've just proved it's not a copy but a reference! 'ref' would be a much better variable name). In any event, we get a pass.

Note that while objects are passed by reference, number and boolean are passed by value, and as strings are immutable this essentially does not affect you. [FDDGJS48]

Prototypes

Prototypes are useful in that you can add properties to an object, and those properties will be shared by all instances of that object. Alternatively, you can define properties in an object's constructor, but doing so means that you have to create these properties for each instance created. That's fine for instance variables but for methods (which are essentially JavaScript functions) remember that functions are objects. So if I have 1 method defined in my constructor, and then instantiate that constructor 5 times, I will have created 5 method objects (one for each instance). This can get quite memory intensive if you need a non-trivial collection of such objects. Therefore, the idiom is to define such methods in a prototype which will be shared by all instances.

<--- PROTOTYPE DIAGRAM NEEDED HERE --->

Prototypes are also a means for achieving inheritance in JavaScript, and we will discuss that aspect of prototypes more in the chapter on Inheritance. For now, let's take a look at how prototype properties work with objects and their instances. We hypothesize that prototype properties are shared by instances:

```
describe 'ObjectCreation'
    before_each
      function MyObj(){};
      MyObj.prototype.foo = 'foo property';
      MyObj.prototype.bar = function() { return "me is bar method"; }
      obj1 = new MyObj();
      obj2 = new MyObj();
    end

    describe 'Prototypes and Object Creation'
      it 'should add prototype property which should be available to all
instances'
          obj1.foo.should.eql 'foo property'
          obj2.foo.should.eql 'foo property'
          obj1.bar().should.eql 'me is bar method'
          obj2.bar().should.eql 'me is bar method'
      end
    end
end
```

Above, we created an object constructor on the first line of the before_each hook, added two prototypal properties, one an *instance variable* of MyObj and the other a method of MyObj. Lastly, we create two instances. In our test case, we assert that for each instance both properties are available. This test passes. This begs the question “What if we manually overwrite a prototype property? Does it affect all instances?” Let's see...

Prototype Shadowing

If we create a case like:

```
          obj1.foo = "new version of foo"
          obj2.foo.should.eql "new version of foo"
```

It fails because obj2 still has the old prototype value (“foo property”). Therefore, this does work:

```
      it 'should update a prototype property but only affect that same instance'
          obj1.foo = "new version of foo"
          obj1.foo.should.eql "new version of foo"
          obj2.foo.should.eql "foo property"
      end
```

“Ok, well are we really updating the property or just *shadowing* it?”. Let's test this by updating a property, deleting the updated property, then checking again to see if original prototype value shows itself again [ZNPJWD158]:

```
    describe 'Updated Prototype Props are really shadowed'
      it 'should return to showing original prototype value after deleting shadow'
          obj1.foo = "new version of foo"
          obj1.foo.should.eql "new version of foo"
          delete obj1.foo;
          obj1.foo.should.eql "foo property"
      end
     end
```

That's interesting! When we delete the updated property, the prototype version of the property shows up again! Let's check if prototype methods exhibit the same shadow behavior:

```
          obj1.bar = function() { return "me is NEW bar method!"; }
          obj1.bar().should.eql 'me is NEW bar method!'
          delete obj1.bar;
          obj1.bar().should.eql 'me is bar method'
```

It's as if we put a layer in front of the original prototype method when we assigned the “me is NEW bar...” version. It's very important to note that the original did not go away as shown above! Hence, we say it's shadowed.

Reinventing Thyself

You can completely overwrite the default prototype and use a prototype literal. In fact, since you can add all of the additional properties at once, it makes visual sense to do so:

js_obj_creation.js

```
function FancyProto(){}
FancyProto.prototype = {
  constructor : FancyProto, // instance.contructor will == Thing and not Object!
  x : 100,
  y : 200,
  foo : function() { return "I am foo"; }
};
```

```
function SloppyProto(){}
SloppyProto.prototype = {
  //constructor : SloppyProto // instance.contructor will now be Object
  name : 'Sloppy Joe'
};
```

Above, we've commented out the constructor field in SloppyProto so calls to sloppyInstance.constructor will equal Object. In our test we can see this is in fact the case:

```
      it 'should have constructor field if needs it to be available to instances'
          t = new FancyProto();
          t.foo().should.eql 'I am foo'
          (t instanceof FancyProto).should.eql true
          (t.constructor == FancyProto).should.eql true
          s = new SloppyProto();
          (s.constructor == SloppyProto).should.eql false
          (s.constructor == Object).should.eql true
      end
```

Note that when we overwrite the prototype we must be sure to do so before creating instances. If not, the instances will just point to default prototype and we'll get an error as soon as we try to access the properties of the new prototype [ZNPJWD163]:

```
      it 'should be that instances created before overwriting prototype point to
old default prototype so new prototype properties WILL NOT be available'
          function Thing(){}
          var beforeProto = new Thing();
          Thing.prototype = {
            foo : 'foo'
          };
          ('foo' in beforeProto).should.eql false
          var afterProto = new Thing();
          ('foo' in afterProto).should.eql true
      end
```

Above, we create instances before and after overwriting the prototype. We can see in the highlighted lines that the *beforeProto* instance does not have access to the foo method but the *afterProto* instance does as this test passes. Note that if we had actually called the foo method on beforeProto we would recieve an error.

Prototype Issue

Putting stuff in prototypes is great because it saves memory. Agreed. But this shared state is not so great for reference properties. The reason this is not a great thing is that the state of these reference properties gets shared between all instances. Java and many other languages call this behavior *static*.

There's an odd caveat that's frankly hard to memorize – primitive types seem to work just fine – it's reference types like Arrays and Objects that exhibit static behavior! The author uses an imperative heuristic: “Never put non-methods in prototypes!” Let's examine this through code. Given the following object:

```
function Issues(){}
Issues.prototype = {
```

```
  x : 1,
  arr : ['uno','dos','tres'],
  hsh : {phone : 'iPhone', carrier : 'ATT' }
}
```

We then test the prototype-static-issue on the primitive, Array, and Hash defined:

```
    describe 'Prototype Issue - Static-Like Behavior ONLY for Reference Types'
      it 'should share state on prototype non-method properties'
          var instance1 = new Issues()
          var instance2 = new Issues()

          // Reference types are essentially static
          instance1.arr.push('cuatro')
          (instance1.arr === instance2.arr).should.eql true

          instance1.hsh['phone'] = 'Blackberry'
          (instance1.hsh === instance2.hsh).should.eql true
          i2Phone = instance2.hsh['phone'].should.eql 'Blackberry'

          // OMG, primitives hide their state just fine!
          instance1.x = 9
          instance1.x.should.eql 9
          instance2.x.should.eql 1
      end
    end
```

The salient point to remember here is: prototype primitives do hide state and act like “non-statics” whereas reference types act like statics and therefore do not hide their state.

The convention is to put methods in the prototype and put the object's properties in the constructor thus combining the constructor and prototype patterns. According to Nicholas Zakas [ZNPJWD166]:

'*The hybrid constructor/prototype pattern is the most widely used and accepted practice for defining reference types in ECMAScript.*'

By doing this, you will retain the integrity of your object properties between instances, while saving memory creating method objects just once to be shared by all instances.

If you are more comfortable with having all of your object creation code in the constructor, you can place the prototype definition directly in the constructor with a conditional to insure that it only gets created one time:

```
if (typeof this.someProtoMethod != “function”) {
  // Define your prototype methods here
}
```

In summary, we have taken a look at object creation details such as:

- object creations through the literal and new Object() idioms
- object references (namely that they are passed by reference not copied)
- many aspects of prototypes relating to object creation

There's a somewhat hazy distinction between object creation and inheritance – you can't inherit anything unless you've creating something to inherit from in the first place. Because of that coupling, we have purposely deferred the discussion of concepts like durable object to the next

chapter.

Inheritance

Contents:

Let's start by discussing what Douglas Crockford calls the Pseudoclassical style of inheritance.

Pseudoclassical

The main characteristics of the pseudoclassical style of inheritance are that we combine constructor and prototype definitions on our parent object (this is more or less the same as what Zakas calls a *hybrid constructor/prototype pattern* as discussed in the last chapter. His version just explicitly specifies that instance variables go in the constructor and methods in the prototype. Crockford's 'mammal' example on page 47 of JavaScript: The Good Parts basically follow this pattern as well [CDJGP47]). Then we create a child object and manually connect it to the parent by overwriting it's prototype and pointing that to the parent object. Let's have a look.

First we make sure we can create the parent object and instantiate it successfully. First the failing test gets written and ran (in /code/spec/spec.js_inheritance.js):

```
    describe 'Pseudoclassical Inheritance Creation'
      it 'should create parent and child objects using pseudoclassical style'
          // First part: create a parent object and successfully instantiate it
          var onyx = new Animal("Onyx")
          onyx.whoAmI().should.match /I am Onyx.*/

          // More to come...
      end
    end
```

ReferenceError: Animal is not defined

Then we implement the code to make it pass:

```
function Animal(name){
    this.name = name;
}
Animal.prototype = {
    whoAmI: function() { return "I am " + this.name + "!\n"; }
};
```

Notice that we have used jspec's '*match*' regex *matcher* which gives us regular expression

matching capabilities. There's no reason to try to remember that we had a '!\n' so regex works nicely here. Next is to create the child object and connect its prototype to the parent object (Animal).

```
function Dog(name, breed) {
    this.name = name;
    this.breed = breed;
}
Dog.prototype = new Animal();
Dog.prototype.getBreed = function() {
    return this.breed;
}
Dog.prototype.bark = function() {
    return 'ruff ruff';
}
```

We start out be creating a Dog constructor that takes an additional property 'breed'. Then we augment the Dog with the *getBreed* and *bark* methods (jumping ahead of ourselves again!). In any event, the initial test passes. Then we decide to test the heck out of the instance and constructor relationships:

```
      it 'should create parent and child object using pseudoclassical inheritance'
          animal.constructor.should.eql Animal
          // dog.constructor.should.eql Dog // Nope: expected Animal to eql Dog
          dog.constructor.should.eql Animal
          animal.should.be_a Animal
          dog.should.be_a Animal
          // dog.should.be_a Dog // Nope!
          // We severed the original prototype pointer and now point to Animal!
          dog.should.be_an_instance_of Animal
          dog.should.be_an_instance_of Dog
          (animal instanceof Dog).should.be_false
      end
```

This is a rather verbose test! It doesn't matter since we're merely testing our understanding of the language, but in a real application we'd probably do away with half the assertions that essentially add code duplication.

The bold lines show that prototypal inheritance has a quirk in that the derivative object's constructor points to the parent (so Dog's constructor points to Animal). The constructor essentially becomes useless to us at runtime in the pseudoclassical pattern.

Both the *whoAmI()* method and the *name* instance variable are defined in the parent class but become available to the child as expected:

```
      it 'should behave so child inherits methods & instance vars from parent'
          animal.whoAmI().should.match /I am Onyx.*/
          dog.whoAmI().should.match /Sebastian.*/
          animal.should.respond_to 'whoAmI'
          dog.should.respond_to 'whoAmI'
          dog.should.have_prop 'name'
      end
```

Prototype Shared State Issue

The issue of shared state was already brought up in the last chapter on object creation – when we created reference types in the prototype, instances shared the state. Well this time we put a reference type in the constructor, but because the child's prototype now points to the parent's constructor, we end up with the same type of issue – the following shows this:

First we add the following to our Animal constructor:

```
this.arr = [1,2,3];
```

Again, recall that at a certain point we essentially do: Child.prototype = new Parent() so the child's prototype will inherit the above 'arr' from Parent. Now all instances of the child will share the state! Here's an example of the potential mess we've created:

```
it 'should behave so reference variables on the parent are "staticy" to all child instances'
    dog.arr.should.eql([1,2,3])
    dog.arr.push(4)
    dog.arr.should.eql([1,2,3,4])
    spike = new Dog("Spike", "Pitbull")
    spike.arr.should.eql([1,2,3,4])
    spike.arr.push(5)
    rover = new Dog("Rover", "German Sheppard")
    spike.arr.should.eql([1,2,3,4,5])
    rover.arr.should.eql([1,2,3,4,5])
    dog.arr.should.eql([1,2,3,4,5])
end
```

We have our original 'dog' instance, and add 'spike' and 'rover' to the mix. Being the spunky canines that they are, each one pushes another element in to the 'arr'. But we see in the last three bolded lines they can all see the changes made. What if these types of changes were made by different developers in a big program? It would be very likely that side effects would get introduced!

Constructor Stealing & Combination Inheritance

Constructor stealing allows a child object to call its parent's constructor. A single line of code is added that calls the parent-constructor with the appropriate properties. So if we had Parent and Child signatures as follows:

```
function Parent(name)
function Child(name, age)
```

The Child constructor would add a line like:

Parent.call(this, name)

By using the '*call*' method on the Parent object, we pass *this* (the newly created Child instance) to the Parent's constructor. By doing so, the Parent constructor now executes with the Child instance as the *current object*. Since the *current object* (or *this*) is what gets acted on in any public function, the Child instance essentially gets constructed to “be an” Animal. Note that we don't pass in age because that's a property that's left unique to Child objects.

The net result is that we get the benefits of prototype chaining while maintaining encapsulation by

adding *constructor stealing* to the mix. The pattern of combining prototype chaining and constructor stealing is called *combination inheritance* [ZNPJWD176]. When we create instances of the Child they will each have their own state.

The following is both the implementation and test for combination inheritance:

```
// Combination Inheritance
function Parent(name) {
    this.name = name;
    this.arr = [1,2,3];
}
Parent.prototype = {
    constructor: Parent,
    toString: function() { return "My name is " + this.name; }
}
function Child(name, age) {
    this.age = age;
    Parent.call(this, name);
}
Child.prototype = new Parent();
Child.prototype.getAge = function() {
    return this.age;
}
```

And the test:

```
    describe 'Combination Inheritance Solves Static Prototype Properties Issue'
      it 'should maintain separate state for each child object'
          child_1 = new Child("David", 21)
          child_2 = new Child("Peter", 32)
          child_1.getAge().should.eql 21
          child_1.arr.push(999)
          child_2.arr.push(333)
          child_1.arr.should.eql([1,2,3,999])
          child_2.arr.should.eql([1,2,3,333])
          child_1.should.be_a Parent
      end
    end
```

The bold lines above show that even though each instance altered the 'arr' reference type, the changes were encapsulated within both instances. Unfortunately, our Child is a Parent which is sort of correct, but not very useful to us later if we need to determine type. Another detail is that this pattern requires calling the Parent constructor twice: once in the constructor stealing Parent.call line, and next for the prototype chaining `Child.prototype = new Parent()`. We'll later see a trick that uses an inherit prototype helper method to chain the prototypes directly.

Prototypal Inheritance

Prototypal inheritance is a simple inheritance option if we don't need to impose strict access. It depends on the availability of a helper function to attach a new object to another parent object:

```
Object.prototype.inherit = function(p) {
   NewObj = function(){};
   NewObj.prototype = p;
   return new NewObj();
```

```
}
```

There's of course nothing requiring you to *monkey patch* Object – you can create a wrapper that returns such functionality and keep Object's namespace "clean". In fact, we've altered the code that follows to use such a wrapper object (see /code/objects/lib/js_inheritance.js and search for *var helper*).

We tap into the simple *stubbing* functionality in jspec to create a person object at the beginning of our tests, by putting the following in a *before* hook:

```
    person = { password : 'secret', toString : function(){ return '<Person>' } }
    stub(person, 'toString').and_return('Original toString method!')
```

and then use it in our test:

```
  describe 'Prototypal Inheritance'
    it 'should inherit properties from parent'
        person.toString().should.match /Original toString.*/i
        person.password.should.eql 'secret'
        joe = helper.inherit(person)
        joe.password.should.eql 'secret'
        joe.password = 'letmein'
        joe.password.should.eql 'letmein'
        person.password.should.eql 'secret'
    end
  end
```

Notice that our child object 'joe' inherits the person password? This may or may not be what you want, but you should be cognizant of the behavior. The nice thing is you didn't have to create two distinct constructors for super and sub types, but you still managed to make the instances "behave like" a Person.

Inheriting Prototypes Not Constructors

On page 181 of Professional JavaScript for Web Developers, 2nd edition, Nicholas Zakas provides the *Parasitic Combination Inheritance* pattern which utilizes a helper function for inheriting prototypes [ZNPJWD181]. Ours is quite similar:

```
// Prototypal Inheritance
var helper = {
    inherit: function(p) {
      NewObj = function(){};
      NewObj.prototype = p;
      return new NewObj();
    },
    inheritPrototype: function(subType, superType) {
      var prototype = helper.inherit(superType.prototype);
      prototype.constructor = subType;
      subType.prototype = prototype;
    }
};

function SubType(name, age) {
```

```
    Parent.call(this, name);
    this.age = age;
};
//Child.prototype = new Parent();   // Gets replaced by:
helper.inheritPrototype(SubType, Parent);
SubType.prototype.getAge = function() {
    return this.age;
};
```

And here's our test:

```
   describe 'Parisitic Combination Inheritance'
      it 'should use inheritPrototype (to call parent constructor once) and still
work as expected'
          sub = new SubType("Nicholas Zakas", 29)
          sub.toString().should.match /.*Nicholas Zakas/
          sub.getAge().should.eql 29
          charlie = new SubType("Charlie Brown", 69)
          charlie.arr.should.eql([1,2,3])
          charlie.arr.push(999)
          charlie.arr.should.eql([1,2,3,999])
          sub.arr.should.eql([1,2,3])
          sub.should.be_an_instance_of SubType
          charlie.should.be_an_instance_of SubType
          (sub instanceof SubType).should.eql true
          (sub instanceof Parent).should.eql true
      end
    end
```

We create two SubType instances, 'sub' and 'charlie'. We push an element on the 'arr' charlie inherited, and test to see if both instances can see the change. The '999' element only is available on to charlie's version of 'arr', so our references are not exhibiting static nature! Moreover, our instanceof appears to be working correctly as well. Notice the code duplication at the end? I just wanted to be sure that jspec's **be_an_instance_of** matcher was working as expected, and in fact testing instanceof behind the scenes. In production code we would want remove the essentially duplicate assertions.

According to Nicholas Zakas, this pattern is used for the YUI's YAHOO.lang.extend() method. [ZNPJWD181]

Functional Inheritance

Douglas Crockford introduced a pattern of object creation and inheritance that does not use the *this* or *new* and has the ability to keep data private. He calls such objects *durable* because they cannot be compromised. A durable object does not provide access to its internal state (even if one has access to the durable object itself). [ZNPJWD169][CDJGP52]

First we will create a durable super object and test that it works as expected while keeping it's variables hidden. First the test:

```
    describe 'Functional Durable Inheritance'
      it 'should hide private variables'
          sub = new SuperFunk(
                {name: "Superfly", age: 39, foo: "foo", bar: "bar"} )
```

```
          sub.getName().should.eql 'Superfly'
          sub.name.should.be_undefined
          sub.getAge().should.eql 39
          sub.age.should.be_undefined
          sub.getFoo().should.eql 'foo'
          sub.foo.should.be_undefined
      end
    end
```

Here is the implementation of the super object:

```
function SuperFunk(blueprint) {
    var obj = {};
    obj.getName = function() { return blueprint.name; };
    obj.getAge  = function() { return blueprint.age; };
    obj.getFoo  = function() { return blueprint.foo; };
    obj.getBar  = function() { return blueprint.bar; };
    return obj;
}
```

Attempts to directly access such private variables (i.e. *name*, *age*, *foo*, etc.) from descendent objects will return undefined. Let's do the inheritance part so we can test this. We create a test and implementation to prove:

```
      it 'should create descendent obj inherits props while maintaining privacy'
          sub = new sub_func(
                {name: "Submarine", age: 1, foo: "food", bar: "barfly"} )
          sub.getName().should.eql 'Submarine'
          sub.name.should.be_undefined
          sub.getAge().should.eql 1
          sub.age.should.be_undefined
          sub.getFoo().should.eql 'food'
          sub.foo.should.be_undefined
          sub.getBar().should.eql 'barfly'
          sub.bar.should.be_undefined
          sub.coolAugment().should.match /.*fresh new perspective.*/
          //sub.should.be_an_instance_of super_func NOPE!
          //sub.should.be_an_instance_of sub_func   NOPE!
          sub.should.be_an_instance_of Object
      end
```

So we need the sub object implementation:

```
function sub_func(blueprint) {
    blueprint.name = blueprint.name || "Crockford's Place";
    supr = super_func(blueprint);
    supr.coolAugment = function() { return "I give a fresh new perspective on
things!" };
    return supr;
};
```

So we got our privacy but notice that we don't get the instanceof functionality. At this point, you may be asking yourself - “Will I actually need reflection capabilities?” If the anwer is “Yes”, then go back and implement the *Parasitic Combination Inheritance* pattern we discussed previously. Note one trade off is that this pattern is simpler than the parasitic combination pattern. In any event,

the *Durable* (otherwise called *Functional)* pattern gives us a big win in the security department!

Summary of Inheritance Patterns

JavaScript gives us a lot of choices when it comes to how we want to implement inheritance! Which one you choose to use should take in to account the strengths and weaknesses of that pattern. Sometimes you may know in advanced that you're only going to need shallow inheritance trees and don't care about *instanceof* capabilities. In this case, Prototypal Inheritance may work just fine. Or, perhaps you're getting a project started, and you know that the developers are you going to be much more productive if they have some familiar capabilities (e.g. *instanceof* reflection, classical looking inheritance models, etc.). You may then want to go with the Parasitic Combination Inheritance pattern. Is security imperative for the object? Functional Inheritance may be a good choice. Feel free to scour the web for even more patterns if none of these suite your particular needs.

The following is a table that summarizes the strengths and weaknesses of these patterns:

Table 1-1. JavaScript Inheritance Patterns

Pattern	Strengths/Weaknesses
Pseudoclassical	Pros: Conventional pattern. Cons: Static references.
Combination Inheritance	Pros: Non-static references. Cons: Calls parent constructor twice.
Prototypal Inheritance	Pros: Very Simple. Good when you don't care about instanceof. Cons: Static references. instanceof and constructors not useful.
Functional Inheritance (Durable)	Pros: Privacy (security). Good when you need durable tamper-proof instance variables. Cons: instanceof and constructors not useful.
Parasitic Combination Inheritance	Pros: Very clean and efficient. Non-static references. Constructor stealing. Only one constructor call. Cons: Complicated (two helper methods required: create_object and inherit_prototype).

JavaScript Gotchas

Contents:

Namespace

One issue inherent in JavaScript is that top-level variables belong to the global namespace. Thus, this can cause problems such as name collisions in content heavy sites like mash-ups. Start combining libraries, advertisers scripts, etc., and you've got a real problem on your hands. One solution to the namespace issue is to create an application-wide object literal on the global namespace and then put all your stuff in there. For example, your name is John Jacob Smith and you're designing an application for ABC Inc. You could create the following namespace:

```
var JJS = {};
JJS.ABC = {};
```

With this in place, you can then attach your objects, functions, etc., to the JJS.ABC object like:
JJS.ABC.Obj = { ... }
JJS.ABC.Func = function(){ ... }
etc.

[DDNSPACING][ZNPJWD645]

parseInt

The parseInt function parses a string and returns an integer. It allows you to provide an optional radix as a second argument for: binary (base 2), octal (base 8), decimal (base 10), hex (base 16), by specifying the base like:

```
parseInt("1000",2); // returns 8
```

The ECMA-262 3 Standard defines the following caveats:

'***parseInt*** *may interpret only a leading portion of the string as an integer value; it ignores any characters that cannot be interpreted as part of the notation of an integer, and no indication is given that any such characters were ignored.*

When radix is 0 or undefined *and the string's number begins with a* 0 *digit not followed by an* x *or* X*, then the implementation may, at its discretion, interpret the number either as being octal or as being decimal. Implementations are encouraged to interpret numbers in this case as being decimal.' [ECMA3-ParseInt]*

Obvious implications are:

1. particular implementation may discard anything after first non-integer
2. 08 and 09 will not be interpreted as decimals and are not valid octal – returns 0!

Let's write some tests:

```
describe 'parseInt Weirdness'
  it 'should ignore any non-integer chars & discard after first non-integer'
      gotchas.callParseInt('82I_AM_IGNORED_1234', 10).should.eql 82
      gotchas.callParseInt('82#$I_AM_IGNORED_1234', 10).should.eql 82
      gotchas.callParseInt('1010_binary_too!_1234', 2).should.eql 10
  end
  it 'should return zero for 08 and 09'
      gotchas.callParseInt('08', undefined).should.eql 0
      gotchas.callParseInt('09', undefined).should.eql 0
      gotchas.callParseInt('09').should.eql 0
      gotchas.callParseInt('09000', undefined).should.eql 0
      gotchas.callParseInt('090jibberish00', undefined).should.eql 0
      gotchas.callParseInt('08', 10).should.eql 8
  end
end
```

We'll save space by omitting the callParseInt implementation as it just calls parseInt behind the scenes. The fact that non-integers are discarded is not so bad, but inadvertent octal issue (exemplified in the second assertion; especially the bold line) shows that we should always supply the base 10 radix to insure decimal behavior if that's what we want.

Equality and == vs ===

To test for equality, JavaScript has both double equals operators: == and !=, and triple equals operators: === and !==. You should pretty much always use the === family of operands because the double equals == tries to do type coercion. The type coercion rules are complex, and as Douglas Crockford asserts, “unmemorable”. Here are some examples extracted from JavaScript: The Good Parts, page 109, and put in jspec test format:

```
describe '== type coercion example from Crockford Good Parts'
  it 'should show that double equals type coercion is complicated and
"unmemorable"'
      ('' == '0').should.eql false
      (0 == '').should.eql true
      ('0' == 0).should.eql true
      (false == 'false').should.eql false
      (false == '0').should.eql true
      (false == undefined).should.eql false
      (false == null).should.eql false
```

```
        (null == undefined).should.eql true
        (' \t\r\n ' == 0).should.eql true
    end
  end
```

[CDJGP109]

Just type in the extra character and don't worry about the above rules! *If you fear that you may inadvertently forget this rule, note that JSLint can help you out. We discuss JSLint later in the book.*

Reserved Words in Object Literals

JavaScript has many reserved words which at the time of this writing are here:

http://javascript.crockford.com/survey.html

There are some oddities in the way JavaScript restricts the use of these reserved words. Namely, object literals cannot use a reserved word as a name (in a name:value pair). But the browsers seem to enforce this differently. For example, when we ran the following test, Firefox 3.5.3 on a Mac did not complain and we got a pass, whereas Safari 4.0.3 choked and gave a 'parse error' in the Web Inspector's JavaScript error console. When we commented out this test, all other tests succeeded proving that this test caused the issue (in Safari).

```
  describe 'reserved words in object literals'
    it 'should not allow reserved words to be used as names in object literals'
        var obj = {
          case: 'case',
          return: 'return'
        };
        obj.case.should.eql 'case'
        obj.return.should.eql 'return'
    end
  end
```

However, enclosing the reserved word names in quotes, and using subscript notation to access the object properites worked for both Firefox and Safari:

```
    it 'should not allow reserved words to be used as names in object literals'
        var obj = {
          'case': 'case',
          'return': 'return'
        };
        obj['case'].should.eql 'case'
        obj['return'].should.eql 'return'
    end
```

If for some reason you need to use a JavaScript reserved word, you should use the above strategy to prevent inconsistent behavior between browsers.

Scope

JavaScript has function scope but not block scope. This means that vars defined in loops,

conditionals, etc., are visible after the block has finished executing:

```
function foo() {
    for(i=0; i<5; i++) {
        console.log(i);
    }
    console.log("i after running for loop is: ", i); // returns 5
}
foo();
```

The above code prints 5 as the *i* variable is still visible outside the for block. This is very odd to most programmers coming from other formal languages. There's a trick to get around the “no block scope” rule if you need it – you can wrap your block inside an anonymous function that calls itself. [ZNPJWD192] Since functions do provide their own scope, you will achieve the desired result:

```
    describe 'block scope workaround'
      it 'should use anonymous function to mimic block scope'
          function foo() {
            try {
                (function() {
                  for(var i=0; i<5; i++) {
                      console.log(i);
                  }
                })();
                return i;
            } catch(e) {
                return "ReferenceError caught";
            }
          }
          foo().should.eql "ReferenceError caught"
      end
    end
```

Unfortunately this example is complicated by the try/catch, but this is the only way we could get the test to run (jspec's should.throw_error didn't seem to help us). Essentially, the bold code shows that our line attempting to **`return i`** – no longer visible – will cause a error.

The reason that the function is preceded by an open parenthesize, is that the interpreter will assume a *function declaration* without it (a *function expression* is what we're after). We can actually use this same self-calling anonymous function to create a singleton with private properties. The next section covers just that.

Privileged Singleton

In order to get privacy in JavaScript, you can use the Privileged Singleton (aka Module) Pattern:

```
var SomeObject = {
    foo: 'foo',
    bar: 'bar'
};

var privileged_singleton = function() {

    // Define some private members
    var privateVar = 'this is private';
    function privateFunc() {
      return 'top secret';
    }

    // Create an object and augment it
    var obj = SomeObject;
    obj.publicVar = 'this is public';

    // Privileged method has access to private members
    obj.publicFunc = function() {
      var message = 'publicFunc is privileged, and hereby allows you to see
private var: ' + privateVar + ', as well as see the return value of privateFunc: '
+ privateFunc();

      return message;
    };

    return obj;
}();
```

Here's the test:

```
    describe 'Priveleged Singleton Pattern (also called Module Pattern)'
      it 'should create a singleton with private members'
          privileged_singleton.privateVar.should.be_undefined
          privileged_singleton.privateFun.should.be_undefined
          privileged_singleton.publicFunc().should.match /publicFunc is.*this is
private.*top secret/i
          privileged_singleton.publicVar.should.eql 'this is public'
          privileged_singleton.foo.should.eql 'foo'
      end
    end
```

Notice that the private members return undefined, while the public function can return private data through it's privileged relationship within the function. Moreover, we have access to both the original SomeObject, as well the augmented properties (e.g. *publicVar* and *publicFunc*).

There are other gotchas in JavaScript such as: bitwise manipulation having to do multiple conversions from 64-bit double precision to 32-bit and back [FDDGJS22]; issues with NaN; the with statement; DOM related issues. The reader is encouraged to do their own research on these.

Part IV – Using What You've Learned

If you've made it this far, congratulations, you should have a pretty good grasp of how to use a TDD approach with JavaScript. You've also learned about some of the peculiarities of JavaScript itself. In this last section, you will put the concepts to use by “rolling up your sleeves” and writing some code.

As opposed to having one continuous project that we build upon, we have chosen to use a more focused example in keeping with the documents goal of brevity.

TDD Example: Linked List

Contents:

Singly Linked Lists

One of the most ubiquitous data structures of programming is the *linked list*. It consists of a number of *links,* or *link nodes,* that each have a reference to the very *next* link. Because of this fact, link nodes are said to be *self-referential* (because they contain a reference to a data member of the same type). Generally, the very last link node points to null indicating that it's the last node.

More accurately, the above paragraph defines a singly linked list - a list that it goes in one direction, front to back. There's also the doubly linked list – a list that can traverse in both directions. It does so by keeping an additional self-reference to the previous node. Here we'll be looking at the singly linked list.

Linked lists offer offer a nice and comprehensive alternative to arrays given the improvement on insertion and deletion performance. Recall that when you insert or delete on an *Array*, you must either rearrange all subsequent elements one at a time, or leave a “gap” in the array. With linked lists, you simply sever the connection by pointing the *previous* link to the (about to be removed) node's *next* link.

Another benefit, is that you can use linked lists as the basis for two other popular data structures, stacks and queues. [LRDSA145][PIESLL]

You may want to see Wikipedia's entry as it has some wonderful graphics to illustrate the previously mentioned removal process:

http://en.wikipedia.org/wiki/File:Singly_linked_list_delete_after.png

http://en.wikipedia.org/wiki/Linked_list

Decomposing Requirements

Let's define the basic requirements for our singly linked list:

- Link nodes **should** contain a simple *id* member and point to the very next node.
- The linked list itself **should** always have an accurate reference to the first link node: the *head*.
- The last node **should** point to null to signify the end of the list.
- We **should** be able to traverse the list from front to back using the last link's null to identify we've reached the end.
- We **should** be able to find an element given its *id* member.
- We **should** be able to delete a link by it's id without messing up our traversal functionality.
- We **should** be able to insert a link at the beginning of the list making it the new *head*.

Wow, look at all the '*shoulds*' – as our specs are pretty self-evident we should be able to convert our requirements one by one as we add to our functionality. It's quite easy to direct a client to use the style: 'something *should* do some behavior' and will pay off for both parties! By using this style, you're allowing the client to use a *ubiquitous language* (in that it's understandable to all), so that '*the business vocabulary permeates right into the codebase.*'

[NDBDDINTRO] *(also see http://en.wikipedia.org/wiki/Domain-driven_design , and http://www.amazon.com/Domain-Driven-Design-Tackling-Complexity-Software/dp/0321125215)*

Getting Started

Using an iterative approach, we will work on one small chunk at a time and try not to get ahead of ourselves. Let's try to simply create a single link node:

```
describe 'Create a node containing simple id member and point to the next node.'
  it 'should create first node with next pointing to null'
    node = new LinkNode(1)
    node.toString().should.eql "LinkNode id: 1"
    node.next.should.be_null
    node.id.should.eql 1
  end
end
```

While writing this first test we decided that it would be useful for LinkNode to have a toString() method so we can do some basic debugging. Given our requirements, we also need to have a *next* member that defaults to null. We do the usual drill of writing the test, watching it fail, and

then implementing just enough to make it pass. At this point, that just means having an appropriate LinkNode constructor and prototype:

```
function LinkNode(id) {
    this.id = id;
    this.next = null;
};
LinkNode.prototype = {
    constructor: LinkNode,
    toString: function() { return "LinkNode id: " + this.id; }
};
```

This allows our first spec to pass. Now we create the *linked list* object – it will need a *head* element which will always point to the first node in the list and be initialized to null. We also want our list to start out with zero elements. So we write our spec: *it 'should create a linked list with zero elements and a head member pointing to null.'* This spec is the first step in tackling our requirement that: *the linked list should always have an accurate reference to the the head.*

Regressions

Of course, we are creating an empty linked list so pointing to null is appropriate, but as we add other features, we will have to insure that "accurate reference to the head" assertion continues to be true...which brings us nicely to the concept of *regressions*. *A*ccording to Wikipedia at *http://en.wikipedia.org/wiki/Software_regression*:

'*A software regression is a software bug which makes a feature stop functioning as intended after a certain event.*'

Let's just take the above 'certain event' to mean new code was added to the system. For example, say we implement our *insertion* requirement, and as a result, our list's *head* member stops pointing to the first node. We'd then have a regression. Hopefully, our automated tests will help us catch this.

List Creation

Moving on, we look back at our spec. The *"linked list with zero elements"* tells us that we'll also need a *length* element to determine how big the list is. So that's a head and a length member for Linked List. Let's get to coding (*code in /code/singly_linked/*):

```
    it 'should create list with zero elements and head member pointing to null.'
      list = new SinglyLinkedList()
      list.length.should.eql 0
      list.head.should.be_null
    end
```

We run the test and get our all too familiar *ReferenceError* (no implementation yet!), so we implement the following:

```
function SinglyLinkedList() {
```

```
    this.head = null;
    this.length = 0;
};
```

So we have our initial specs setup and passing. We immediately realize that our members are publicly accessible which breaks encapsulation. We could use a *durable object,* hide the vars and return a closure, etc., but for this pedantic example we won't worry about every such detail.

Insertion

Given that we basically have an empty list, we decide it makes sense to tackle *insertion*. There are many variations of insertion that we could go for: insert after a given node or node key, insert at end, at an index, etc. For now we'll go with implementing an insert beginning which will essentially replace our list's head with the newly inserted node:

```
    describe 'List insertion'
      it 'should insert link at the beginning of the list making it the new head.'
          node = new LinkNode(1)
          list.insertBeginning(node)
          list.head.id.should.eql node.id
      end
    end
```

And then to get it to pass quickly we add the following:

```
SinglyLinkedList.prototype = {
    constructor: SinglyLinkedList,
    insertBeginning: function(node) {
      this.head = node;
    }
};
```

The savvy reader will say, 'Hey, your new node will always point to null...when you put another node in, the list's chain will be broken!'. Correct. We need to improve our spec from:

`'should insert link at the beginning of the list making it the new head.'`

to:

`'should insert link at the beginning of the list making it the new head, and also have head->next point to the old head.'` But that's too wordy, how about:

`'should insert node as new head of list, and have its next point to old head.'`

Notice how much easier that reads? We've taken out extraneous words like 'at the', etc., and have

stated only the essentials. Here's our new test:

```
  it 'should insert node as head of list & have its next point to old head.'
      firstNode = new LinkNode(1)
      list.insertBeginning(firstNode)
      list.head.id.should.eql firstNode.id
      secondNode = new LinkNode(2)
      list.insertBeginning(secondNode)
      list.head.id.should.eql secondNode.id
      list.head.next.id.should.eql firstNode.id
      list.length.should.eql 2
  end
```

The last two lines at the end prove that when we insert two or more nodes, the head will be the last node inserted and that node's next will point to the previous head (which has now become the second link). We essentially alter the implementation of `insertBeginning()` to read:

```
insertBeginning: function(node) {
  node.next = this.head;
  this.head = node;
  this.length = this.length + 1;
}
```

Voilà! We have insertion. Let's work on traversal.

Traversing

Our implementation will traverse the list and return an array of node ids:

```
  it 'should traverse list from front to back and return array of node ids'
      list.insertBeginning(firstNode)
      list.insertBeginning(secondNode)
      arr = list.traverse()
      arr.should.eql([2,1])
  end
```

The implementation is a simple matter of storing our node ids in the array and then moving our current node to node.next (until we reach the end signified by null):

```
traverse: function() {
  var arrayOfIds = [];
  var current = this.head;
  while(current != null) {
      arrayOfIds.push(current.id);
      current = current.next;
  }
  return arrayOfIds;
}
```

This passes and now we're ready for removal.

Removal

At this point, we'll just show you the whole set of tests and implementation as opposed to taking you through one test spec at a time:

```
describe 'Link node removal'
  it 'should return false when list is empty and we try to remove'
      removed = list.remove(2)
      removed.should.eql false
  end
  it 'should return true when list has one matching link & length should==0'
      list.insertBeginning(firstNode)
      list.remove(1).should.eql true
      list.length.should.eql 0
  end
  it 'should remove a link node by id'
      list.insertBeginning(firstNode)
      list.insertBeginning(secondNode)
      list.insertBeginning(new LinkNode(3))
      list.remove(2).should.eql true
      list.length.should.eql 2
      list.remove(999).should.eql false
      /* Insure we in fact removed link */
      list.traverse().should.eql([3,1])
  end
  it 'should return false if non-integer passed in'
```

```
        list.insertBeginning(firstNode)
        list.remove('abc').should.eql false
    end
end
```

And the implementation:

```
remove: function(id) {
  var current  = this.head;
  var previous = this.head;
  while(current != null) {
      if(current.id === id) {         // If found match
        if(current === this.head) {   // If first link in list
            this.head = this.head.next;
        } else {
            previous.next = current.next;
        }
        this.length--;
        return true;
      } else {                        // Didn't match. Move to next link
        previous = current;
        current  = current.next;
      }
  }
  return false;
}
```

In our implementation of removal, we essentially loop through the list until one of two conditions occurs – we either:

1. find a match, in which case we remove the node passed in, update the pointers, and return true.
2. don't find a match at all, in which case we return false.

There's a corner case to 1. above – if it's the very first node that matches, we simply move the head to the current head's next.

Finding

Next we need to be able to find a particular node by its id:

```
    describe 'Find an element given its id member.'
      it 'should find and return a node by id if match in list otherwise returns
null'
          list.insertBeginning(firstNode)
          list.insertBeginning(secondNode)
          node = list.find(firstNode.id)
          node.should.equal firstNode
          list.find(999).should.be null
          list.find('junk').should.be null
      end
      it 'should return null if list is empty'
          list.find(firstNode.id).should.be null
          list.find(secondNode.id).should.be null
          list.find("garbage").should.be null
      end
    end
```

Implementation:

```
    find: function(id) {
      var current = this.head;
      while(current != null) {
          if(current.id === id) {
            return current;
          }
          current = current.next;
      }
      return null;
    }
```

We've now fulfilled our initial requirements for our singly linked list!

Disclaimer: This is not a “production-ready” linked list, nor may it even be necessary given JavaScript's Array.prototype capabilities. Also, we should have things like isEmpty(), insertBefore(node), guards against bad inputs, etc. These might make for good exercises for the reader. While you're at it, why not make a doubly linked list, a stack, a queue, etc.?

Extra Credit

We already mentioned that it would be easy enough to add useful additional functionality to our

list. Methods like insertBefore(n) or insertAfter(n) seem pretty straight forward, so we won't go in to those. However, an interesting problem is finding the Nth-to-last element in the list. An intriguing book, *Programming Interviews Exposed*, suggests the following basic algorithm (*they show their implementation in C – but we don't have to worry about pointers per se*):

1. Fast forward a *pointer* (they call it 'current') 'N' elements.
2. Until current->next is null, forward 'current' and an 'nBehind' pointer to the next element.
3. Once current->next points to null, the nBehind will be the Nth-to-last element so return it.

They also put a check at the beginning to insure that the list is in fact at least N elements long or return null. Looking at this algorithm, it's essentially O(n) as the *current* pointer traverses the whole list, while the n*Behind* will never completely traverse the list. [PIESLL43]

Upon reading this, it occurred to us that another solution would be to simply traverse the list to get the *size*; let's assume the list does not provide a length property (which ours actually does). Then we could get the index by computing *index* = (*size* - n) and then traverse to that *index*. It doesn't improve on efficiency, but it does seem like another way to skin the cat, and we'd like to check that we're correct.

Just to be clear, if we have 10 elements and we want the 2nd to last we subtract can 2 from 10 and get 8. If we consider the first element to be zero, than this index will work (remember the 8th element starting at 0 is actually 9!)

Let's try implementing our algorithm in JavaScript using our typical jspec approach (*you can find this code in /code/singly_linked/*):

```
describe 'Find Nth-to-last element.'
    it 'should find Nth-to-last if list at least N elements, otherwise return null'
      list.insertBeginning(firstNode)
      list.insertBeginning(secondNode)
      list.insertBeginning(new LinkNode(3))
      list.findNthToLast(2).should.eql secondNode
    end
end
```

If you're anything like the author, you like to insure that you don't have an *off by one* error. We cheat and and use Firebug to do a console.log of the size (our implementation still incomplete):

```
findNthToLast: function (n) {
  if(typeof n === 'number' && isFinite(n)) {
      var size = 1;
      while(current.next !== null) {
        size++;
        current = current.next;
      }
      console.log("size after while is: ", size);
```

....

We try creating 2 nodes, 5 nodes, etc., and insure that the size printed out is correct. Of course our tests will fail while we're doing this, but it's just a sanity check so that's fine. Once we're confident our size is correct, we go ahead and finish off our first stab and get a pass on our first test case (code omitted for brevity sake).

Ok, so what are some conditions we should defend against? Typically, *invalid inputs*, *empty data structures*, etc., are the types of things we need consider – so we add these tests:

```
    it 'should return null if list is empty or less than N'
      // List is empty but we try to find N
      list.findNthToLast(2).should.be null
      list.insertBeginning(firstNode)
      list.insertBeginning(secondNode)
      // List is 2 elements. Lets make N == 3
      list.findNthToLast(3).should.be null
    end
    it 'should return null if n argument is not a number'
      list.insertBeginning(firstNode)
      list.findNthToLast('abc').should.be null
      list.findNthToLast({one:1,two:2}).should.be null
      list.findNthToLast([1]).should.be null
    end
```

It turns out we have to do a bit to insure that we defend against all of these – our code is now working but ugly:

```
  findNthToLast: function (n) {

    if(typeof n === 'number' && isFinite(n)) {  // Is it a number?

        if(this.head !== null) {    // Guard: List empty?
          var current = this.head;
        } else {
          return null;
        }
        var size = 1;
        while(current.next !== null) {
          size++;
```

```
          current = current.next;
        }

        if(size < n) return null;   // List needs to be bigger then n

        var index = size - n;
        current = this.head;          // reset current to beginning
        for(i=0; i<index; i++)
          current = current.next
        return current;
    } else {
        return null; // not a valid number input
    }
}
```

We check for the input being a number and our list being empty or less then N. Notice that our guard code is almost half of the function size! Up to now, we've been pretty lax on our guard code, but this is more typical of real world functions. In fact, if we were being really thorough, we may want to use try/catch/throw code as well. One thing we can do is to refactor some of the code into a helper:

```
findNthToLast: function (n) {

  if(!isANumber(n) || this.isEmpty() ) { return null; } // Guard code

  // Set current to head - loop list to get size
  var current = this.head;
  var size = 1;
  while(current.next !== null) {
      size++;
      current = current.next;
  }
  if(size < n) return null;   // List needs to be bigger then n

  var index = size - n;       // Compute the index

  current = this.head;        // Reset current to beginning
```

```
    for(i=0; i<index; i++)      // Traverse to index and return
        current = current.next


    return current;
}
```

We've put all the ceremonious guard code into two helpers: `isANumber` and `isEmpty` and can now put the guard code in one line! The essence of what this function does is now self-evident and easier on the eyes. Notice that we've even managed to add space to make it read better? Having moved out the helper functionality (input checks), we can know think more stylishly about our code – before we were self conscious of the fact that our overall function length might be excessive. Moreover, we can now call our helpers from other functions as well. If we discover an issue with this, we only have to fix it in one place – always a nice gain.

So what did we just do? We essentially refactored working, albeit ugly, code. Having our unit tests in place, we were able to move confidently, knowing that if we could reproduce our passes, our code essentially worked as before. The refactored function is clearer and more maintainable by other programmers. We've also shown that refactoring and testing go hand in hand.

JavaScript Builds

Contents:

So far, we've been using a process that essentially involves: writing a test, watching it fail, implementing some code, making the test pass. This is essentially the TDD process, and is great for adding new functionality to an application one micro-feature at a time. However, there are some other things we should do when it comes time to deploy our code:

- Validation: Use a tool to help validate whether we're adhering to best practices, style conventions, etc. JSLint is one such tool for doing this which we'll look at soon.
- Compression: Rather than serving up a page with several JavaScript files, it is better to condense them into one file and strip out comments, etc.
- Automation: Ideally, we should be able to automate the validation and compression steps. Any build tool should help with this such as Ant, rake, make, etc.
- HTTP Compression: This essentially has to do with applying gzip compression to our JavaScript files before sending back from server. *This is out of the scope of this document, but if you are using Apache you may look into modules such as mod_gzip for doing this.*

Before we try to completely automate our build, we should become familiar with the individual

components themselves. Let's start with validation.

Validation

JSLint, the tool we will use for validation, was written by Douglas Crockford. You can optionally paste your individual scripts into an online version of JSLint to manually validate them, but we will use another tool, the Rhino shell, to run JSLint from the command line. Rhino is a Java based JavaScript engine created by the Mozilla Foundation which essentially converts JavaScript into Java. It can be used in many ways, but we just want to use Rhino's shell to create a JavaScript environment to run JSLint. Rhino requires Java and so you will need to make sure you have Java installed on your system before proceeding.

Installing Rhino

First we need to download the Rhino bundle:

http://www.mozilla.org/rhino/download.html

Once you've downloaded the bundle and unzipped it, there will be a lot of stuff in the directory. As a sanity check, let's make sure the Rhino shell is working as expected: open a terminal and cd to your unzipped Rhino directory and execute the Rhino shell as follows:

```
$ java -jar js.jar
Rhino 1.7 release 2 2009 03 22
js> print("hi");
hi
```

Note that you will need to enter Control-D to exit the Rhino shell.

Run JSLint via the Rhino Shell

Now let's get JSLint and connect it to Rhino:

http://www.jslint.com/rhino/index.html or alternatively:

```
curl http://jslint.com/rhino/jslint.js > jslint.js
```

To get up and running quickly, let's copy the js.jar, jslint.js, and our test program mytest.js into the same directory – in the repository, we already have these set up in /code/`jslint_rhino_test/`.

Now we should be able to run our program into JSLint with something like:

```
java -jar js.jar jslint.js mytest.js.
```

JSLint Options

Upon our first run of JSLint, we got some errors like:

```
Expected 'alert' to have an indentation...
```

and,

```
'alert' is not defined.
```

This is because JSLint expects you to supply options for certain assumptions like:

- browser globals are predefined (i.e. 'alert' above)
- indentation should not be checked

The following is how such options are supplied from within your script:

```
/*jslint browser: true, nomen: false, glovar: false, rhino: true, eqeqeq: true,
white: false */
```

Essentially, *browser: true* allows you to use the browser's globals, *white: false* fixes the indentation check error from above, *nomen: false* turns off var name checking, *eqeqeq: true* makes sure you use '===' not '=='. The full set of JSLint options available are listed here: http://www.jslint.com/lint.html#options

Given the following JavaScript file:

```
/*jslint browser: true, nomen: false, glovar: false, rhino: true, eqeqeq: true, white: false */
var person = {
   name: "Rob",
   foo: function() {
        alert("OMG...BOOOORRRRING!!!!");
   },   // See if JSLint sees this?
};
var pickyBrackets =
{
   picky: "Doug likes brackets in K&R style"
}
```

We assumed that JSLint would complain about: the extra comma; the use of the ANSI C++ style bracket placement. However, JSLint doesn't seem to mind the bracket placement:

```
$ java -jar js.jar jslint.js mytest.js
Lint at line 6 character 6: Extra comma.
},    // See if JSLint sees this?

Lint at line 11 character 2: Missing semicolon.
}
```

It does catch the extra comma as we expected, but also tells us about a missing semicolon at the end (which we didn't expect because it was an actual human mistake!). So we fix these simple issues and then get:

```
jslint: No problems found in mytest.js
```

Success!

Note that running JSLint as we develop is advisable, and will be easy once we define it as it's own "build task". See build section presented later for details on how to do so.

Compression

There are several options available for *minification* of your JavaScript files. Some of the popular ones are:

- YUI Compressor(Julient Lecomte and the YUI team)
- jsmin (Douglas Crockford)
- packer (Dean Edwards)
- Dojo's ShrinkSafe

We will look at YUI Compressor as we feel it is the best option, but you are encouraged to review

the other options as well.

YUI Compressor

YUI Compressor requires that your system has Java and Rhino installed. If you've been following along you already have this set up. You can download the YUI Compressor library from:
http://yuilibrary.com/downloads

We have included the YUI Compressor jar file, and our test code for the following example in our github repository under:
/code/yui_compressor/

We will use the mytest.js script that we previously put through the JSLint.

```
java -jar yuicompressor-2.4.2.jar myfile.js -o myfile-min.js
```

This produced the following:

```
var person={name:"Rob",foo:function(){alert("OMG...BOOOORRRRING!!!!")}};var
pickyBrackets={picky:"Doug likes brackets in K&R style"};
```

As can be seen, by default YUI Compressor will remove spaces when possible. However, it didn't replace our variable names because didn't have any local vars for it to rename. Let's put the following code through the compressor (nested.js):

```
var person = {
    name: "Rob",
    foo: function() {
      var aVeryLongVariableName = 'long var';
      alert("OMG...BOOOORRRRING!!!!");
      (function() {
          var nestedFunc = 'nest function value';
      })();
    }
};
```

Compression applied produces (*note that this document wraps but the following is essentially one long line*):

```
var person={name:"Rob",foo:function(){var a="long
var";alert("OMG...BOOOORRRRING!!!!");(function(){var b="nest function value"})
()}};
```

We can see that in addition to white space being stripped out, the local variables:
`'aVeryLongVariableName' got renamed to 'a' and 'nestedFunc' got renamed to 'b'.`

YUI Compressor works on CSS files as well.

If you are working in a Ruby environment, you may want to try a wrapper for YUI Compressor written by Sam Stephenson On github here:
http://github.com/sstephenson/ruby-yui-compressor

If you're looking for .NET:
http://yuicompressor.codeplex.com/

An alternative compressor for PHP: http://code.google.com/p/minify/

None of the above have been tested by the author so you will have to check these out for your self.

Automated Builds

We would like a way to automate validation, concatenation, and compression of our JavaScript files using the individual tools we've already looked at. Although this can be done by many tools such as: Ant, make, etc., we will use rake as the author is in “Ruby-mode”. The syntax is simple enough that it should be easy to transpose in to one of the other tools should you have a preference that leads you to do so. If you don't already have rake you can get it from here:

http://rake.rubyforge.org/

The source code is in the /code/js_build directory of the github repository. The important file for your to edit is the Rakefile. Please find the line with the following comment:

```
    # ---------- EDIT THESE PROPERTIES! --------- #
```

You will need to edit the paths appropriately for your system. That said, the repository already has the paths correct for this script. From within this directory you should be able to run:

```
rake js:min
```

As does make and Ant, rake allows you to create *tasks* and *dependencies*. A *task* is a specific action to carry out (i.e. delete all files in current directory, compile some code, push *.js to http://server.com/js_files/, etc.) *Dependencies* are a list of prerequisite tasks the current task depends on. For example, we choose to run JSLint validation before doing any sort of compression, and so our :min (compression task) depends on :lint to execute successfully first. :lint, in turn depends on :clean, etc.

Essentially, the js:min rake task will recursively take all of the JavaScript files (excluding the jslint.js file) and do the following:

- delete any previously created amalgamation.js file - if exists (the :clean task)
- run JSLint validation (the :lint task)
- concatenate all of the files (:min task)
- run the YUI Compressor on the concatenated file producing: *amalgamation.js (:min task)*

If you happen to be a Ruby/Rails developer, there are some tools you may try which can leverage the automated deployment: *juicer* or *sprockets*:
http://cjohansen.no/en/ruby/juicer_a_css_and_javascript_packaging_tool
http://github.com/sstephenson/sprockets

The following is the Rakefile found at /code/js_build/Rakefile in the github repository [RAKERDOC] [RAKEGUIDE]:

```
require 'tempfile'
namespace :js do
    # ---------- EDIT THESE PROPERTIES! --------- #
    RHINO_JAR_PATH        = "rhino/js.jar"
    JSLINT_PATH           = "jslint/jslint.js"
    JAVA_JAR_CMD          = "java -jar"
    JSLINT_SUCCESS        = "^jslint: No problems found in"
    JS_EXCLUDE            = JSLINT_PATH
```

```
    YUI_DEV_PROPS         = "--nomunge --line-break"
    YUI_COMP_JAR_PATH     = "yui_comp/yuicompressor-2.4.2.jar"
    AMALGAMATION_FILE     = "amalgamation.js"

    desc "Deletes the amalgamation file (if exists)."
    task :clean do
       if File.exists?("#{AMALGAMATION_FILE}") then
           system "rm #{AMALGAMATION_FILE}"
           puts "Deleted: #{AMALGAMATION_FILE}."
       else
           puts "Nothing to clean: #{AMALGAMATION_FILE} file does not exist."
       end
    end

    # Modified the :jslint task from: http://jonathanjulian.com/
    desc "Puts each JS script through JSLint. Exits if there's a problem."
    task :lint => [:clean] do
       failed_js_scripts = []
       jsfiles = File.join("**", "*.js")      # ** makes it recursive

       # A bit of ugliness to reject the jslint.js file itself (if it's in a subdir of the
cwd)
       Dir.glob(jsfiles).reject{|path| path =~ /#{Regexp.quote(JS_EXCLUDE)}/ }.each do |js|
           command = "#{JAVA_JAR_CMD} #{RHINO_JAR_PATH} #{JSLINT_PATH} #{js}"
           jslnt_result = %x{#{command}}
           unless jslnt_result =~ /#{Regexp.compile(JSLINT_SUCCESS)}/
              puts "#{jslnt_result}:"
              puts jslnt_result
              failed_js_scripts << js
           end
       end
       if failed_js_scripts.size > 0
           exit 1
       else
           puts "-------------------------------------------"
           puts "Passed JSLint tests for all JavaScript files"
           puts "-------------------------------------------"
           puts
       end
    end

    desc "Minify JavaScript by concatenating files, then apply YUI Compressor to
amalgamation file"
    task :min => [:lint] do
       concat_file = Tempfile.new("temp_file.js", Dir.getwd )
       jsfiles = File.join("**", "*.js")      # ** makes it recursive
       Dir.glob(jsfiles).reject{|path| path =~ /#{Regexp.quote(JS_EXCLUDE)}/ }.each do |js|
           open(js) do |f|
              concat_file.write(f.read)
           end
       end
       concat_file.rewind
       command = "#{JAVA_JAR_CMD} #{YUI_COMP_JAR_PATH} --type js  #{concat_file.path} -o
#{AMALGAMATION_FILE}"
       %x{#{command}}
       puts "-------------------------------------------"
       puts "Minified JavaScript - resulting file: #{AMALGAMATION_FILE}"
       puts "-------------------------------------------"
       puts
    end
end
```

In order to do the whole “shebang” you could do:

```
rake js:min
```

in the /code/js_build/ directory, and rake will run JSLint on each of your JavaScript files, concatenate them, and compress them using the YUI Compressor. You may also want to add a task to fire off your unit tests, a :dev task to run JSLint then your unit tests, etc.

Ant Alternative

If you prefer using Ant, there's an excellent tutorial by the creator of YUI Compressor himself, Julient Lecomte, which is very useful:
http://www.julienlecomte.net/blog/2007/09/16/

References

The following bibliography lists references to sources cited throughout this book:

Bibliography

BKTDD2002: Beck, Kent, Test Driven Development: By Example, November 18, 2002
ZNTDD2008: Zakas, Nicholas, Nicholas C. Zakas: Test-Driven Development with YUI Test, 2008, http://yuiblog.com/blog/2008/10/20/video-zakas-yuitest/
NDBDDINTRO: North, Dan, Introducing BDD, 2006, http://dannorth.net/introducing-bdd
ZakasTDD2008: Zakas, Nicholas, Nicholas C. Zakas: Test-Driven Development with YUI Test, 2008, http://yuiblog.com/blog/2008/10/20/video-zakas-yuitest/
RSPECBDD: Unknown, RSpec: Get Started Now, , http://rspec.info/
KYCTDS2008: Yehuda Katz , Writing Code That Doesn't Suck, 2008
CDJGS20: Crockford, Douglas, JavaScript: The Good Parts, 2008
CDJGP22: Crockford, Douglas, JavaScript: The Good Parts, 2008
FDDGJS48: Flanagan, David, JavaScript: The Definitive Guide, Fifth Edition, 2006
ZNPJWD158: Zakas, Nicholas, Professional JavaScript for Web Developers, 2nd Edition, 2009
ZNPJWD163: Zakas, Nicholas, Professional JavaScript for Web Developers, 2nd Edition, 2008
ZNPJWD166: Zakas, Nicholas, Professional JavaScript for Web Developers, 2nd Edition, 2008
CDJGP47: Crockford, Douglas, JavaScript: The Good Parts, 2008
ZNPJWD176: Zakas, Nicholas, Professional JavaScript for Web Developers, 2nd Edition, 2009
ZNPJWD181: Zakas, Nicholas, Professional JavaScript for Web Developers, 2nd Edition, 2009
ZNPJWD169: Zakas, Nicholas, Professional JavaScript for Web Developers, 2nd Edition, 2009
CDJGP52: Crockford, Douglas, JavaScript: The Good Parts, 2008
DDNSPACING: Dustin Diaz, Namespacing your JavaScript , 2006, http://www.dustindiaz.com/namespace-your-javascript/
ZNPJWD645: Zakas, Nicholas, Professional JavaScript for Web Developers, 2nd Edition, 2009
ECMA3-ParseInt: ECMA General Assembly, Standard ECMA-262 3rd Edition - December 1999, 1999
CDJGP109: Crockford, Douglas, JavaScript: The Good Parts, 2008
ZNPJWD192: Zakas, Nicholas, Professional JavaScript for Web Developers, 2nd Edition, 2009
FDDGJS22: Flanagan, David, JavaScript: The Definitive Guide, Fifth Edition, 2006
LRDSA145: Lafore, Robert, Data Structuresand Algorithms in 24 Hours, 1999
PIESLL: Noah Suojanen, John Mongan, Programming Interviews Exposed, 2000

PIESLL43: Noah Suojanen, John Mongan, Programming Interviews Exposed, 2000
RAKERDOC: Unknown, Rakefile Format (as of version 0.8.3), 2008, http://rake.rubyforge.org/
RAKEGUIDE: Jim Weirich, Rake User Guide, 2005

Notes to self

Contents:

- fib, next prime, towers/hanoi, data structues/algor., etc.

Oreilly Template

The following are from the Oreilly Open Office template:

1

ChapterTitle

All of the proper ORA styles are under “Custom Styles” (choose this at the bottom of the Stylist). I highly recommend having the Stylist open at all times when using the template. Please use only these styles and do not modify them in any way.

Use the ORATools→Insert... menu to insert an arrow character (→) or an em dash (—). Don't add a space before or after either character.

Heading examples:

HeadA

HeadB

HeadC

HeadD—usually discouraged

Do not use any character styles in headings ever.

Inline font examples:

Body with *Emphasis,fi* and Strong and `Literal` and Hyperlink and **`User Input`** and ***`Replaceable`*** and ***`User Input Replaceable`*** and *Technical Italic* and *Filename* and some Superscript and then some Subscript.

Here's XRefColor and XRefColorCW, which are only used in *some* books.

Note: Ignore the WW... styles in Character Styles section of the Stylist--they are not part of the template

---

Here's a properly formatted table:

Table 1-1. Here's the TableTitle

CellHeading	**CellHeading**
CellSubhead	*<----If* **necessary** bsksdflkjsdfsdlfkjsdflkj
CellBody with **Strong** and **Emphasis,fi**	CellCode

Some code examples:

Example 1-1. ExampleTitle (use only if you want a numbered example)

```
this is Code
this is CodeEmphasis
1  CodeNum
This is code with User Input and User Input Replaceable
Code may only be 85 characters wide for the widest books. Do not use carriage
returns—separate all lines onto separate paragraphs
0        10        20        30        40        50        60        70        80
12345678901234567890123456789012345678901234567890123456789012345678901234567890123456789012345
```

Here's a Figure:

FigureHolder

Figure 1-1. Figure Title (notice this para is below the FigureHolder, unlike TableTitle and ExampleTitle)

Note: FigureTitle, ExampleTitle, and TableTitle paragraphs should always start with the caption in the form:

Figure X-YY. [Caption Text]

X = ChapterLabel, YY = elements sequential number (1 digit for <10), notice trailing period and space.

Examples:

- Figure 2-5. Stars in the Milky Way
- Table 12-15. C++ Functions

In-text cross references should match the element they want to point to (with the capital Figure, etc) but should not include the period.

Examples:

- See Figure 2-5 for more on the Milky Way.
- Table 12-15 examines some common C++ functions...

And notes:

Note

`Note>Code`

- Note>ListBullet

1. Note>ListNumber this is literal

NoteWarning

`NoteWarning>Code`

- NoteWarning>ListBullet

1. NoteWarning>ListNumber

Lists:

ListSimple

- ListBullet

  ```
  >Code, use instead of code (only) inside lists
  ```

  ListBullet...

  - >ListBullet

    >ListBullet...

Note: the numbering on ListNumber won't restart correctly.. sorry. It'll look fine in FrameMaker; just ignore it for now.

1. ListNumber

   ListNumber...

   a. >ListNumber

      >ListNumber...

ListVariableTerm

ListVariable

>ListVariableTerm

>ListVariable

Sidebars:

SidebarTitle

SIDEBARTYPE (ONLY FOR SOME BOOKS

SidebarBody

`SidebarCode`

- SidebarListBullet

1. SidebarListNumber

Misc:

Quote

Comment (Production Note)

Epigraph

EpigraphAuthor

EpigraphCitation

Hacks books use EpigraphAuthor at the end of Hacks written by other contributors.

RefName (again, only use these if you've been instructed to)

- RefPurpose

RefSynopsis

RefSectA

RefSectB

RefSectC

RefSectD

```
0        10        20        30        40        50        60        70        80
12345678901234567890123456789012345678901234567890123456789012345678901234567890123456789012345
12345678901234567890123456789012345678901234567890
123456789012345678901234567890123456789012345678901234567890123456789012345678901234567890123456789012345678
1234567890
```

1234567890