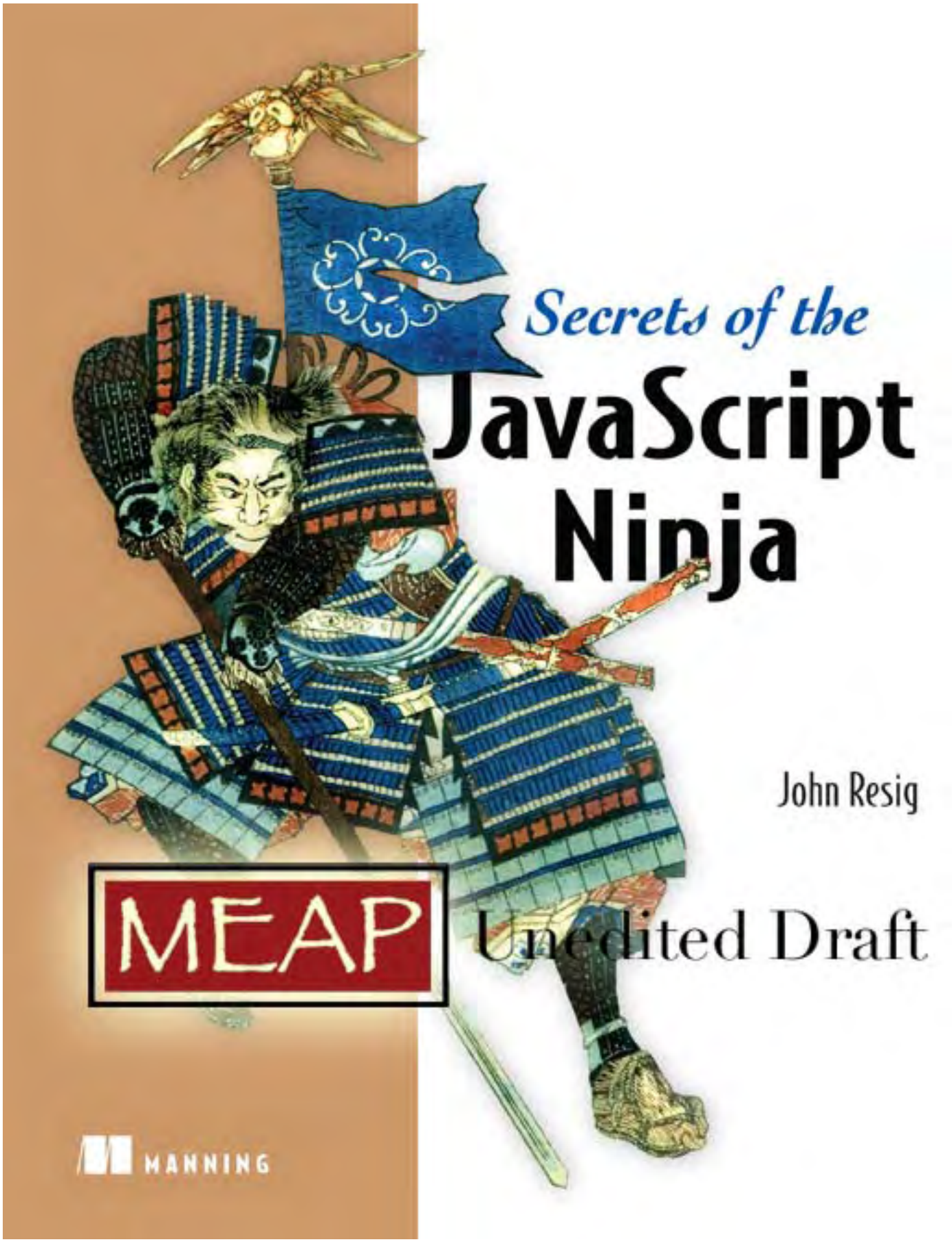


Secrets of the JavaScript Ninja

John Resig

MEAP Unedited Draft

MANNING

MEAP Edition
Manning Early Access Program

For more information on this and other Manning titles go to
www.manning.com

Table of Contents

List of Tables

Chapter 1. Introduction

In this chapter:

- Overview of the purpose and structure of the book
- Overview of the libraries of focus
- Explanation of advanced JavaScript programming
- Theory behind cross-browser code authoring
- Examples of test suite usage

There is nothing simple about creating effective, cross-browser, JavaScript code. In addition to the normal challenge of writing clean code you have the added complexity of dealing with obtuse browser complexities. To counter-act this JavaScript developers frequently construct some set of common, reusable, functionality in the form of JavaScript library. These libraries frequently vary in content and complexity but one constant remains: They need to be easy to use, be constructed with the least amount of overhead, and be able to work in all browsers that you target.

It stands to reason, then, that understanding how the very best JavaScript libraries are constructed and maintained can provide great insight into how your own code can be constructed. This book sets out to uncover the techniques and secrets encapsulated by these code bases into a single resource.

In this book we'll be examining the techniques of two libraries in particular:

- Prototype (http://prototypejs.org/): The godfather of the modern JavaScript libraries created by Sam Stephenson and released in 2005. Encapsulates DOM, Ajax, and event functionality in addition to object-oriented, aspect-oriented, and functional programming techniques.
- jQuery (http://jquery.com/): Created by John Resig and released January 2006, popularized the use of CSS selectors to match DOM content. Includes DOM, Ajax, event, and animation functionality.

These two libraries currently dominate the JavaScript library market being used on hundreds of thousands of web sites and interacted with by millions of users. Through considerable use they've become refined over the years into the optimal code bases that they are today. In addition to Prototype and jQuery we'll also look at a few of the techniques utilized by the following libraries:

- Yahoo UI (http://developer.yahoo.com/yui/): The result of internal JavaScript framework development at Yahoo released to the public in February of 2006. Includes DOM, Ajax, event, and animation capabilities in addition to a number of pre-constructed widgets (calendar, grid, accordion, etc.).
- base2 (http://code.google.com/p/base2/): Created by Dean Edwards and released March 2007 supporting DOM and event functionality. Its claim-to-fame is that it attempts to implement the various W3C specifications in a universal, cross-browser, manner.

All of these libraries are well-constructed and tackle their desired problem areas comprehensively. For these reasons they'll serve as a good basis for further analysis. Understanding the fundamental construction of these code bases will be able to give you greater insight into the process of large JavaScript library construction.

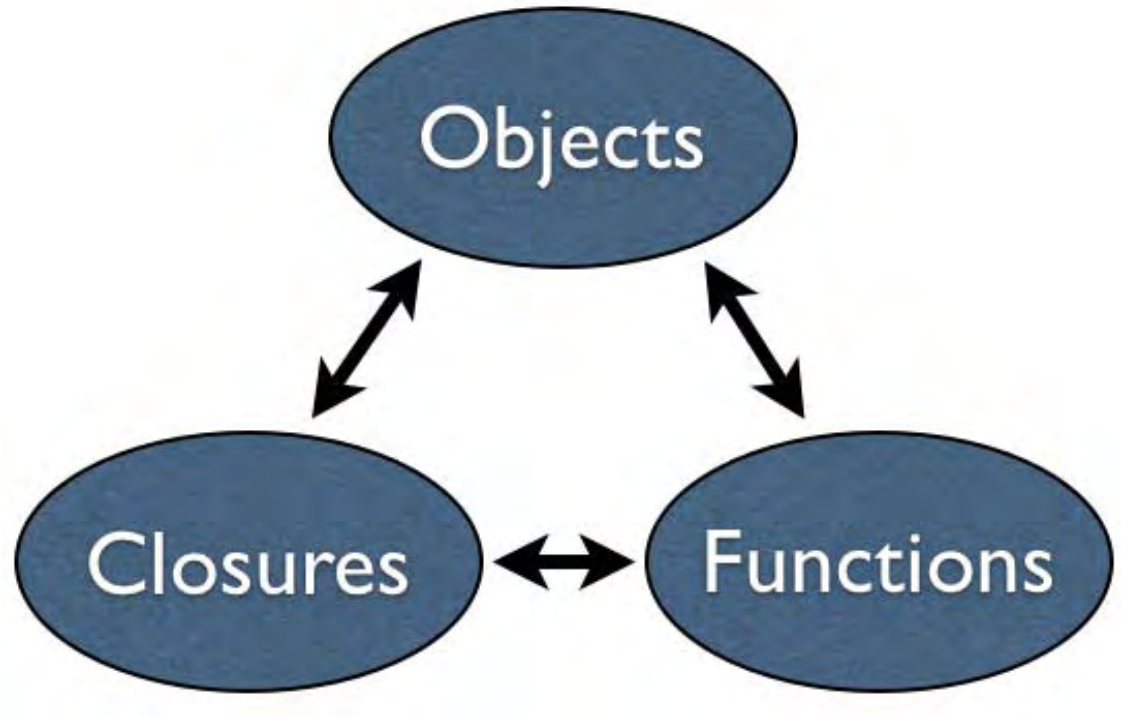
The make up of a JavaScript library can be broken down into three portions: Advanced use of the JavaScript language, comprehensive construction of cross-browser code, and a series of best practices that

tie everything together. We'll be analyzing all three of these together to give us a complete set of knowledge to create our own, effective, JavaScript code bases.

The JavaScript Language

Most JavaScript users get to a point at which they're actively using objects, functions, and even anonymous functions throughout their code. Rarely, however, are those skills taken beyond the most fundamental level. Additionally there is generally a very poor understanding of the purpose and implementation of closures in JavaScript, which helps to irrevocably bind the importance of functions to the language.

Figure 0-1: A diagram showing the strong relation between the three, important, programmatic concepts in JavaScript.



Understanding this strong relationship between objects, functions, and closures will improve your JavaScript programming ability giving you a strong foundation for any type of application development.

There are, also, two features that are frequently used in JavaScript development that are woefully underused: timers and regular expressions. These two features have applications in virtually any JavaScript code base but aren't always used to their full potential due to their misunderstood nature. With timers the express knowledge of how they operate within the browser is often a mystery but understanding how they work can give you the ability to write complex pieces of code like: Long-running computations and smooth animations. Additionally, having an advanced understanding of how regular expressions work allows you to make some, normally quite complicated, pieces of code quite simple and effective.

Finally, in our advanced tour of the JavaScript language, we'll finish with a look at the `with` and `eval` statements. Overwhelmingly these features are trivialized, misused, and outright condemned by most JavaScript programmers but by looking at the work of some of the best coders you can see that, when used appropriately, they allow for the creation of some fantastic pieces of code that wouldn't be possible otherwise. To a large degree they can, also, be used for some interesting meta-programming exercises

molding JavaScript into whatever you want it to be. Learning how to use these features responsibly will certainly affect your code.

All of these skills tie together nicely to give you a complete package of ability with which any type of JavaScript application authoring should be possible. This gives us a good base for moving forward starting to write solid cross-browser code.

Writing Cross-Browser Code

While JavaScript programming skills will get us far, when developing a browser-based JavaScript application, they will only get us to the point at which we begin dealing with browser issues. The quality of browsers vary but it's pretty much a given that they all have some bugs or missing APIs. Therefore it becomes necessary to develop both a comprehensive strategy for tackling these browser issues in addition to the knowledge of the bugs themselves.

The overwhelming strategy that we'll be employing in this book is one based upon the technique used by Yahoo! to quantify their browser support. Named "Grade Browser Support" they assign values to the level of support which they are willing to provide for a class of browser. The grades work as follows:

- **A Grade**: The most current and widely-used browsers, receives full support. They are actively tested against and are guaranteed to work with a full set of features.
- **C Grade**: Old, hardly-used, browsers, receives minimal support. Effectively these browsers are given a bare-bones version of the page, usually just plain HTML and CSS (no JavaScript).
- **X Grade**: Unknown browsers, receives no special support. These browsers are treated equivalent to A-grade browsers, giving them the benefit of a doubt.

To give you a feel for what an A-grade browser looks like here is the current chart of browsers that Yahoo uses:

Figure 0-2: A listing of A-grade browsers as deemed by Yahoo!, early 2009.

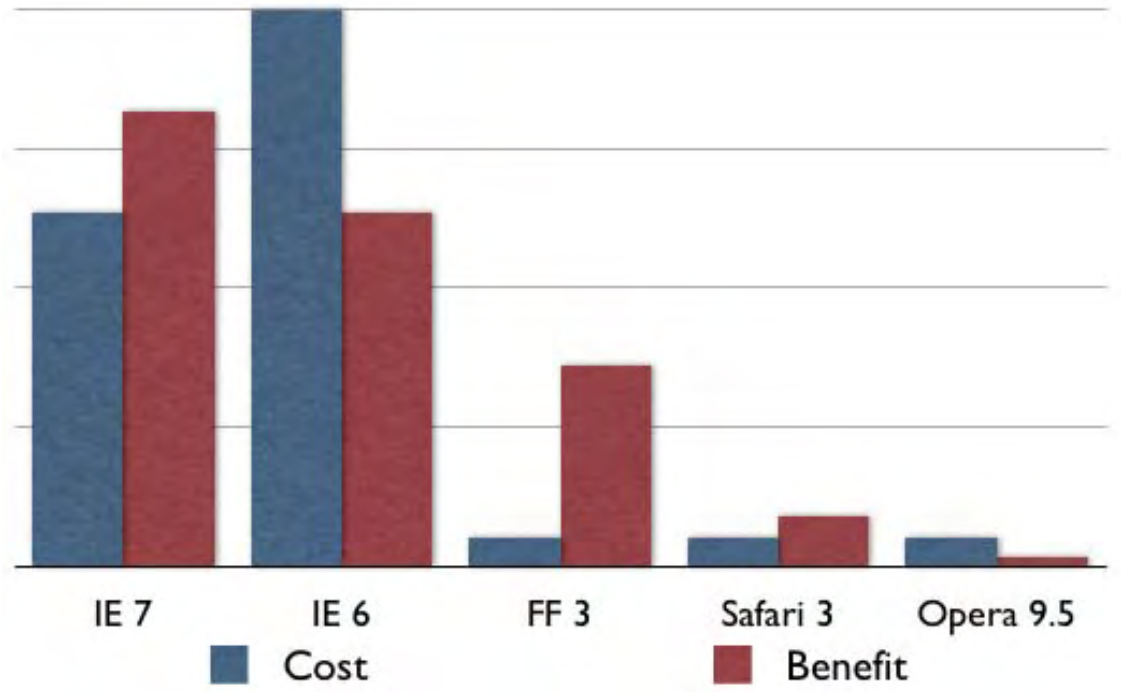
	Win 98	Win 2000	Win XP	Win Vista	Mac 10.4	Mac 10.5
IE 7.0			A-grade	A-grade		
IE 6.0	A-grade	A-grade	A-grade			
Firefox 2.†	A-grade	A-grade	A-grade	A-grade	A-grade	A-grade
Opera 9.†	A-grade	A-grade	A-grade		A-grade	A-grade
Safari 3.0†					A-grade	A-grade

- Graded Browser Support: http://developer.yahoo.com/yui/articles/gbs/

What's good about this strategy is that it serves as a good mix of optimal browser coverage and pragmatism. Since it's impractical in your daily development to actually be able to develop against a large number of platforms simultaneously it's best to chose an optimal number. You end up having to balance the cost and the benefit that is required in supporting a browser.

What's interesting about analyzing the cost-benefit of a browser is that it's done completely differently from straight-up analysis of browser market share. It's really a combination of market share and time that'll be spent customizing your application to work in that browser. Here's a quick chart to represent my personal choices when developing for browsers:

Figure 0-3: A cost-benefit rough comparison for popular web browsers. These numbers will vary based upon your actual challenges relating to developing for a browser.



The "Cost" is represented by the % of time that will be spent, beyond normal application development, spent exclusively with that browser. The "Benefit" is the % of market share that the browser has. Note that any browser that has a higher cost, than benefit, needs to be seriously considered for development.

What's interesting about this is that it use to be much-more clear-cut when choosing to develop for IE 6 - it had 80-90% market share so it's benefit was always considerably higher (or, at least, equal to) the time spent making it work in that browser. However, in 2009, that percentage will be considerably less making it far less attractive as a platform. Note that Firefox and Safari, due to their less-buggy nature (and standards compliance) always have a higher benefit than cost, making them an easy target to work towards. Opera is problematic, however. It's, continued, low market share makes it a challenging platform to justify. It's for this reason that major libraries, like Prototype, didn't treat Opera as an A-grade browser for quite some time - and understandably so.

Now it's not always a one-to-one trade-off in-between cost and benefit. I think it would even be safe to say that benefit is, at least, twice as important as cost. Ultimately, however, this depends upon the choices of those involved in the decision making and the skill of the developers working on the compliance. Is an extra 4-5% market share from Safari 3 worth 4-5 developer days? What about the added overhead to Quality Assurance and testing? It's never an easy problem - but it's one that we can, certainly, all get better at over time - but really only through hard work and experience.

These are the questions that you'll need to ask yourself when developer cross-browser applications. Thankfully cross-browser development is one area where experience significantly helps in reducing the cost overhead for a browser and this is something that will be supplemented further in this book.

Best Practices

Good JavaScript programming abilities and strong cross-browser code authoring skills are both excellent traits to have but it's not a complete package. In order to be a good JavaScript developer you need to maintain the traits that most good programmers have including testing, performance analysis, and debugging.

It's important to utilize these skills frequently and especially within the context of JavaScript development (where the platform differences will surely justify it).

In this book we'll be actively using a number of testing techniques in our examples to, both, show the use of a basic test suite and to ensure that our examples operate as we would expect them to.

The basic unit of our testing library is the following `assert` function:

Listing 0-1: Simple example of assert statements from our test suite.

```
assert( true, "This statement is true." );
assert( false, "This will never succeed." );
```

The function has one purpose: To determine if the value being passed in as the first argument is true or false and to assign a passing or failing mark to it based upon that. These results are then logged for further examination.

> **Note:** You should realize that if you were to try the `assert()` function (or any of the other functions in this section) that you'll need the associated code to run them. You can find the code for them in their respective chapters or in the code included with this book.

Additionally we'll be occasionally testing pieces of code that behave asynchronously (the begin instantly but end at some indeterminate time later). To counter-act this we wrap our code in a function and call `resume()` once all of our `assert()`s have been executed.

Listing 0-2: Testing an asynchronous operation.

```
test(function(){
  setTimeout(function(){
    assert( true, "Timer was successful." );
    resume();
  }, 100);
});
```

These two pieces give us a good test suite for further development, giving us the ability to write good coverage of code easily tackling any browser issues we may encounter.

The second piece of the testing puzzle is in doing performance analysis of code. Frequently it becomes necessary to quickly, and easily, determine how one function performs in relation to another. Throughout this book we'll be using a simple solution that looks like the following:

Listing 0-3: Performing performance analysis on a function.

```
perf("String Concatenation", function(){
  var name = "John";
  for ( var i = 0; i < 20; i++ )
    name += name;
});
```

We provide a single function which will be execute a few times to determine its exact performance characteristics. To which the output looks like:

Table 0-1: All time in ms, for 5 iterations, in Firefox 3.

Table 1.1.

	Average	Min	Max	Deviation
String Concatenation	21.6	21	22	0.50

This gives us a quick-and-dirty method for immediately knowing how a piece of JavaScript code might perform, providing us with insight into how we might structure our code.

Together these techniques, along with the others that we'll learn, will apply well to our JavaScript development. When developing applications with the restricted resources that a browser provides coupled with the increasingly complex world of browser compatibility having a couple set of skills becomes a necessity.

Summary

Browser-based JavaScript development is much more complicated than it seems. It's more than just a knowledge of the JavaScript language it's the ability to understand and work around browser incompatibilities coupled with the development skills needed to have that code survive for a long time.

While JavaScript development can certainly be challenging there are those who've already gone down this route route: JavaScript libraries. Distilling the knowledge stored in the construction of these code bases will effectively fuel our development for many years to come. This exploration will certainly be informative, particular, and educational - enjoy.

Chapter 2. Testing and Debugging

In this chapter:

- Tools for Debugging JavaScript Code
- Techniques for Generating Tests
- How to Build a Test Suite
- How to Test Asynchronous Operations

Constructing an effective test suite for your code can be incredibly important in situations where externalities have the potential to affect your code base - which is especially the case in cross-browser JavaScript development. Not only do you have the typical problem of dealing with multiple developers working on a single code base and the result of breaking portions of an API (generic problems that all programmers deal with) but you also have the problem of determining if your code works in all the browsers that you support.

We'll discuss the problem of cross-browser development in-depth when we look at Strategies for Cross-Browser Code in a later chapter. For now, though, it's important that the importance of testing be outlined and defined, since we'll be using it throughout the rest of the book.

In this chapter we're going to look at some tools and techniques for debugging JavaScript code, generating tests based upon those results, and constructing a test suite to reliably run the tests.

Debugging Code

The ability to debug JavaScript code has dramatically improved in the last few years - in large part thanks to the popularity of the Firebug developer extension for Firefox. There are now similar developer tools for all major browsers:

- Firebug: The popular developer extension for Firefox http://getfirebug.org/
- IE Developer Tools: Included in Internet Explorer 8.
- Opera Dragonfly: Included in Opera 9.5 and newer - also works with Mobile versions of Opera.
- Safari Developer Tools: Introduced in Safari 3 and dramatically improved in Safari 4.

There are two important parts to debugging JavaScript: Logging statements and breakpoints. They are both useful for tackling a very similar situation, but from a different angle: Figuring out what is occurring at a specific line in your code.

Logging statements (such as using the `console.log` method in Firebug, Safari, and IE) are useful in a cross-browser sense. You can write a single logging call in your code and you can benefit from seeing the message in the console of all browsers. The browser consoles have dramatically improved this process over the old 'add an alert into your page' technique, since all your statements can be collected into the console and be browsed at a later time (not impeding the normal flow of the program). An example of a simple, cross-browser, method for console logging can be seen in Listing 1-1.

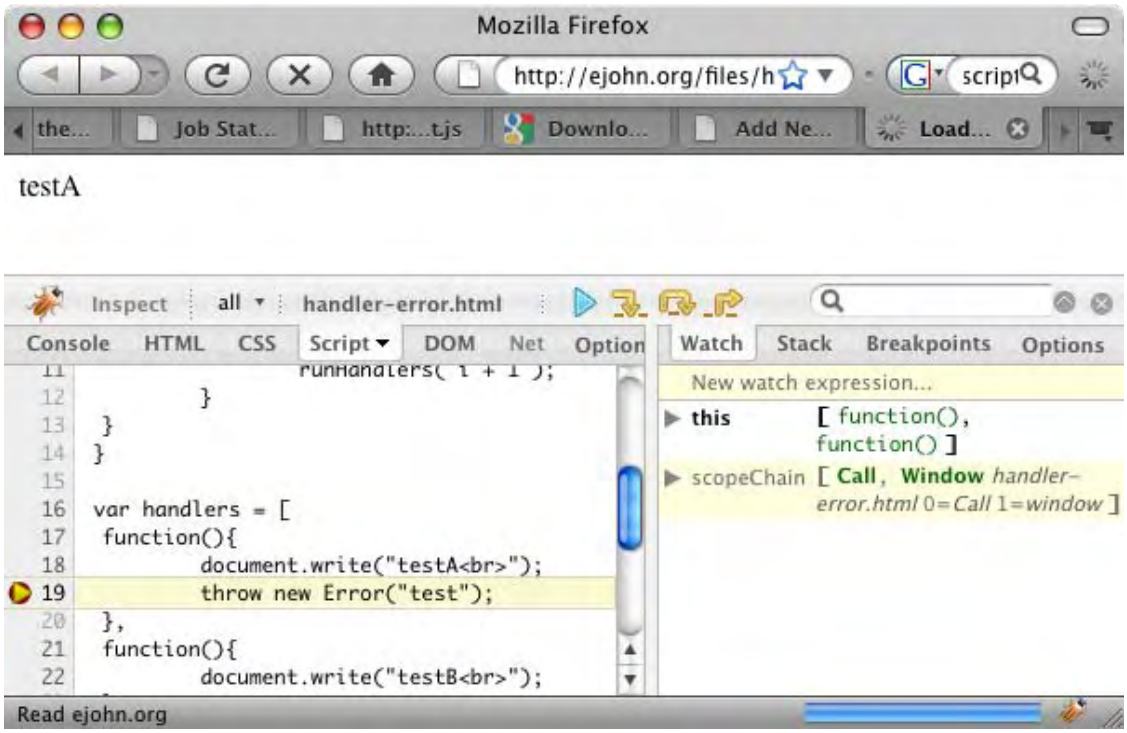
Listing 1-1: A simple logging statement that works in all browsers.

```
function log() {
  try {
```

```
    console.log.apply( console, arguments );
  } catch(e) {
    try {
      opera.postError.apply( opera, arguments );
    } catch(e){
      alert( Array.prototype.join.call( arguments, " " ) );
    }
  }
}
```

Breakpoints aren't as simple as logging statements but they do provide a notable advantage: They can halt the execution of a script at a specific line of code (pausing the browser) and investigate all sorts of state that you don't normally have access to (including all accessible variables, the context, and the scope chain, like in the example showing in Figure 1-1:

Figure 1-1: A screenshot of Firebug paused at a breakpoint.



A full debugger, with breakpoint capabilities, is a feature that is highly dependent upon the browser environment in which it is run - it's for this reason that the aforementioned developer tools were created, since the functionality provided by them could not be duplicated in any other way. It's such a relief then that all the browser developers have come on board to create effective utilities for performing this action.

Debugging code does serve a useful purpose (beyond the obvious fixing of bugs) it leads towards the good practice of generating effective test cases.

Test Generation

There are a number of techniques that can be used for constructing a test with the two major ones being: deconstructive tests and constructive tests. Regardless of the test type there are a few points that are important to remember:

- **Your tests should be highly reproducible.** It should be possible to run your test over and over and never receive a different result. Additionally, this includes making sure that your tests are dependent upon outside issues like network or CPU load.

- **Your tests should be as simple as possible.** You should strive to remove as much HTML markup, CSS, or JavaScript as you can without disrupting the original test case. The more that you can remove the greater the likelihood that the test case will only be influenced by the exact bug that you're trying to detect.

- **Break tests apart.** Try not to make the results from one test be dependent upon another. Break tests down into their smallest possible unit (which helps you to determine the exact source of a bug when an error occurs).

With those points in mind, let's look at the two techniques for developing test cases:

Deconstructive Test Cases

Deconstructive test cases center around the use of an existing site or page that exhibits an issue and tearing it down to only show the representative problem. This is the ultimate case of applying the above points to test generation. You might start with a complete site but after removing extra markup, CSS, and JavaScript you'll arrive at a smaller site that reproduces the problem. You continue the process until no more code can be removed.

Constructive Test Cases

With a constructive test case you start from a known, reduced, case and build up until you're able to reproduce the bug in question. In order to do this style of testing you'll need a couple simple test files from which to build and a way to generate these new tests with a clean copy of your code.

For example, when I'm attempting to create reduced test cases I'll end up having a few HTML files with minimum functionality already included in them (one for DOM manipulation, one for Ajax tests, one for animations, etc.). For example, in Listing 1-2, is the simple DOM test case that I use.

Listing 1-2: A reduced DOM test case for jQuery.

```
<script src="dist/jquery.js"></script>
<script>
$(document).ready(function(){
  $("#test").append("test");
});
</script>
<style>
#test { width: 100px; height: 100px; background: red; }
</style>
<div id="test"></div>
```

To generate a test, with a clean copy of the code base, I use a little shell script to check the library, copy over the test case, and build the test suite, as shown in Listing 1-3.

Listing 1-3: A simple shell script used to generate a new test case.

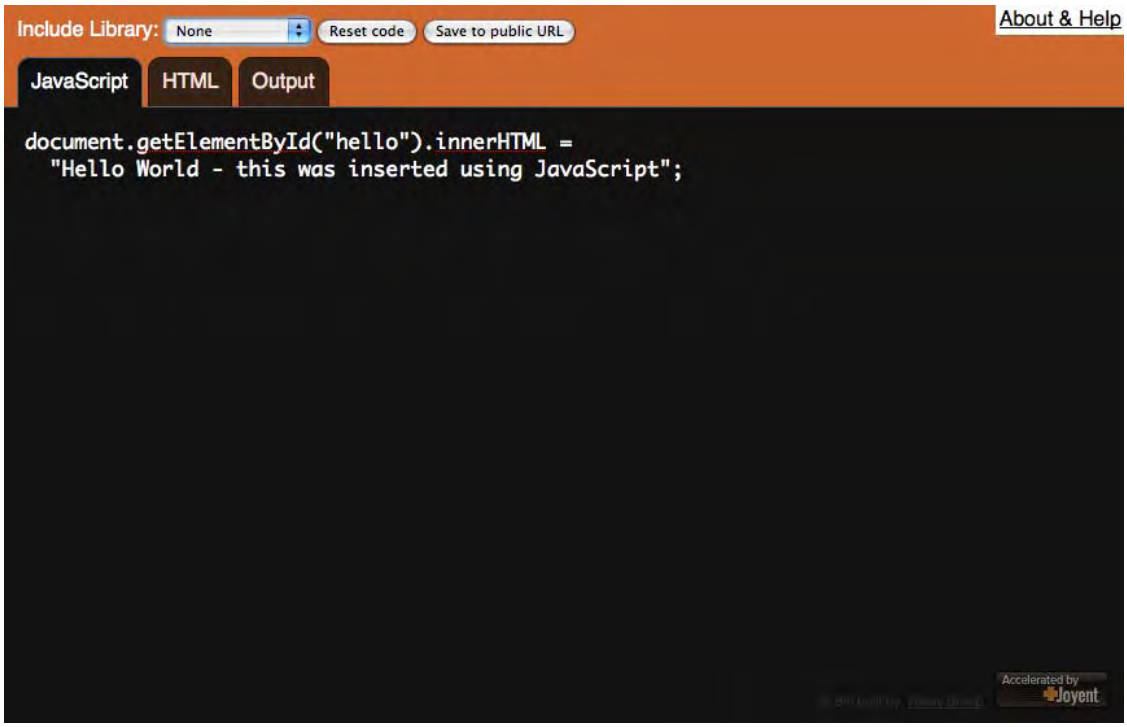
```
#!/bin/sh
# Check out a fresh copy of jQuery
svn co https://jqueryjs.googlecode.com/svn/trunk/jquery $1

# Copy the dummy test case file in
```

```
cp $2.html $1/index.html

# Build a copy of the jQuery test suite
cd $1 && make test
```

The above script would be run like so: `./gen.sh mytest dom` (which would pull in the DOM test case from dom.html).

Another alternative, entirely, is to use a pre-built service designed for creating simple test cases. One of these services is JSBin (http://jsbin.com/), a simple tool for building a test case that then becomes available at a unique URL - you can even include copies of some of the most popular JavaScript libraries, an example of which is shown in Figure 1-2.

Figure 1-2: A screenshot of the JSBin web site in action.



With the tools and knowledge in place for figuring out how to reduce a test case we should start to build a test suite around it so that it becomes easier to run these tests over-and-over again.

Building a Test Suite

The primary purpose of a test suite is to aggregate all the individual tests that your code base might have into a single location, so that they can be run in bulk - providing a single resource that can be run easily and repeatedly.

One would have to ask, though: Why would I want to build a new test suite? It's a good question. For most cases it probably isn't necessary to write your own JavaScript test suite, there already exist a number of good-quality suites to choose from:

- QUnit: http://docs.jquery.com/QUnit
- JSUnit: http://www.jsunit.net/

- YUI Test: http://developer.yahoo.com/yui/yuitest/

For the vast majority of use cases a JavaScript test suite should serve a single need: Display the result of the tests (making it easy to determine which tests have passed or failed).

There are a number of features for which you might want to use a proper JavaScript unite testing framework, those include:

- The ability to simulate browser behavior (clicks, keypresses, etc.)
- Interactive control of tests (pausing and resuming tests)
- Handling when asynchronous tests time out
- Filtering of tests to run

Although, if none of those features are ones that you need then I recommend writing your own unit testing framework: It's surprisingly easy to do.

The core of a unit testing framework is the 'assert' method. This method takes a value and a description - if the value is 'truth-y' then it passes, otherwise it is marked as a failure. The associated message is logged to a list with the proper pass/fail indicator. A simple implementation of this concept can be seen in Listing 1-4.

Listing 1-4: A simple implementation of a JavaScript test suite.

```
<html>
<head>
  <title>Test Suite</title>
  <script>
  function assert( value, desc ) {
    var li = document.createElement("li");
    li.className = value ? "pass" : "fail";
    li.appendChild( document.createTextNode( desc ) );
    document.getElementById("results").appendChild( li );
  }

  window.onload = function(){
    assert( true, "The test suite is running." );
  };
  </script>
  <style>
    #results li.pass { color: green; }
    #results li.fail { color: red; }
  </style>
</head>
<body>
  <ul id="results"></ul>
</body>
</html>
```

The above test suite is terribly simple - but it serves as a good building block for future development. We'll be using the assert() method that we defined above throughout this book to test the individual code listings, to verify their integrity.

Beyond simple testing of code the next important part of a testing framework is handling of asynchronous operations.

Asynchronous Testing

The first complicated step that most developers encounter, when using or developer a JavaScript test suite, is handling asynchronous tests. These are tests whose results will come back after an indeterminate amount of time (such as an Ajax test or an animation).

Often, handling this issue is made much more complicated than it need be. To handle asynchronous tests you'll need to follow a couple steps:

1. Assertions that are relying upon the same asynchronous operation will need to be grouped into a unifying 'test'.

2. Each test group will need to be placed on a queue to be run after all the previous test groups have finished running.

3. Thus, each test group must be capable of run asynchronously.

An example implementation of this technique is shown in Listing 1-5.

Listing 1-5: A simple asynchronous test suite.

```
(function(){
  var queue = [], paused = false;

  this.test = function(fn){
    queue.push( fn );
    runTest();
  };

  this.pause = function(){
    paused = true;
  };

  this.resume = function(){
    paused = false;
    setTimeout(runTest, 1);
  };

  function runTest(){
    if ( !paused && queue.length ) {
     queue.shift()();
     if ( !paused ) {
        resume();
      }
    }
  }
})();

test(function(){
  pause();
  setTimeout(function(){
    assert( true, "First test completed" );
    resume();
  }, 100);
});
```

```
test(function(){
  pause();
  setTimeout(function(){
    assert( true, "Second test completed" );
    resume();
  }, 200);
});
```

To break down the functionality exposed in in Listing 1-5, there are three publicly accessible functions: test, pause, and resume. These three functions have the following capabilities:

1. `test(fn)` takes a function (which contains a number of assertions which will be run either synchronously or asynchronously), places it on the queue to await execution.

2. `pause()` should be called from within a test function and tells the test suite to pause executing tests until the test group is done.

3. `resume()` un-pauses the tests and starts the next test running, after a short delay (this in place to avoid long-running code blocks).

The one internal function, `runTest`, is called whenever a test is queued and dequeued. It checks to see if the suite is currently un-paused and if there's something in the queue (if that's the case, then it'll dequeue a test and try to execute it). Additionally, after the test group is finished executing it will check to see if the suite is currently 'paused' and if not (meaning that only asynchronous tests were run in the test group) it will begin executing the next group of tests.

Some of the techniques outlined above are explained better in the chapter on Timers, where much of the nitty-gritty relating to delayed execution is gone into depth.

Summary

In this chapter we've looked at some of the basic technique surrounding debugging JavaScript code and constructing simple test cases based upon those results. Building upon that we looked at how to construct a simple test suite capable of handling asynchronous test cases - which we will be using throughout the rest of the book to test the validity of the rest of the listings. Altogether this will serve as an important cornerstone to the rest of your development with JavaScript.

Chapter 3. Functions

In this chapter:

- Overview of the importance of functions
- Using functions as objects
- Context within a function
- Handling a variable number of arguments
- Determining the type of a function

The quality of all code that you'll ever write, in JavaScript, relies upon the realization that JavaScript is a functional language. All functions, in JavaScript, are first-class: They can coexist with, and can be treated like, any other JavaScript object.

One of the most important features of the JavaScript language is that you can create anonymous functions at any time. These functions can be passed as values to other functions and be used as the fundamental building blocks for reusable code libraries. Understanding how functions, and by extension anonymous functions, work at their most fundamental level will drastically affect your ability to write clear, reusable, code.

In this chapter we'll explore the various ways in which functions work, all the way from the absolute basics (defining functions) to understanding the complex nature of function context and arguments.

Function Definition

While it's most common to define a function as a standalone object, to be accessed elsewhere within the same scope, the definition of functions as variables, or object properties, is the most powerful means of constructing reusable JavaScript code.

Let's take a look, in Listing 2-1, at some different ways of defining functions (one, the traditional way, and two using anonymous functions):

Listing 2-1: Three different ways to define a function.

```
function isNimble(){ return true; }
var canFly = function(){ return true; };
window.isDeadly = function(){ return true; };
assert( isNimble() && canFly() && isDeadly(),
  "All are functions, all return true" );
```

All those different means of definition are, at a quick glance, entirely equivalent (when evaluated within the global scope - but we'll cover that in more detail, later). All of the functions are able to be called and all behave as you would expect them to. However, things begin to change when you shift the order of the definitions, as in Listing 2-2.

Listing 2-2: A look at how the location of function definition doesn't matter.

```
var canFly = function(){ return true; };
window.isDeadly = function(){ return true; };
```

```
assert( isNimble() && canFly() && isDeadly(),
  "Still works, even though isNimble is moved." );
function isNimble(){ return true; }
```

We're able to move the isNimble function because of a simple property of function definitions: No matter where, within a scope, you define a function it will be accessible throughout. This is made incredibly apparent in Listing 2-3:

Listing 2-3: Defining a function below a return statement.

```
function stealthCheck(){
  var ret = stealth() == stealth();
  assert( ret, "We'll never get below the return, but that's OK!" )
  return true;
  function stealth(){ return true; }
}
stealthCheck();
```

In the above, `stealth()` will never be reached in the normal flow of code execution (since it's behind a return statement). However, because it's specially privileged as a function it'll still be defined as we would expect it to. Note that the other ways of defining the function (using anonymous functions) do not get this benefit - they only exist after the point in the code at which they've been defined, as in Listing 2-4.

Listing 2-4: Types of function definition that can't be placed anywhere.

```
assert( typeof canFly == "undefined",
  "canFly doesn't get that benefit." );
assert( typeof isDeadly == "undefined", "Nor does isDeadly." );
var canFly = function(){ return true; };
window.isDeadly = function(){ return true; };
```

This is all very important as it begins to lay down the fundamentals for the naming, flow, and structure that functional code can exist in - it also helps to establish the framework through which we can understand anonymous functions.

Anonymous Functions and Recursion

Recursion is, generally speaking, a solved problem. When a function calls itself (from inside itself, naturally) there is some level of recursion occurring. However, the solution becomes much less clear when you begin dealing with anonymous functions, as in Listing 2-5.

Listing 2-5: Simple function recursion.

```
function yell(n){
  return n > 0 ? yell(n-1) + "a" : "hiy";
}
assert( yell(4) == "hiyaaaa",
  "Calling the function by itself comes naturally." );
```

A single named function works just fine, but what if we were to use anonymous functions, placed within an object structure, like in Listing 2-6?

Listing 2-6: Function recursion within an object.

```
var ninja = {
```

```
  yell: function(n){
    return n > 0 ? ninja.yell(n-1) + "a" : "hiy";
  }
};
assert( ninja.yell(4) == "hiyaaaa",
  "A single object isn't too bad, either." );
```

This is all fine until we decide to create a new samurai object; duplicating the yell method from the ninja. We can see how things start to break down, since the anonymous yell function, within the ninja object, is still referencing the yell function from the ninja object. Thus, if the ninja variable is redefined we find ourselves in a hard place, as seen in Listing 2-7.

Listing 2-7: Recursion using a function reference that no longer exists (using the code from Listing 2-6).

```
var samurai = { yell: ninja.yell };
var ninja = {};

try {
  samurai.yell(4);
} catch(e){
  assert( "Uh, this isn't good! Where'd ninja.yell go?" );
}
```

How can we work around this, making the yell method more robust? The most obvious solution is to change all the instances of ninja, inside of the ninja.yell function, to 'this' (the generalized form of the object itself). That would be one way, another way could be to give the anonymous function a name. This may seem completely contradictory, but it works quite nicely, observe in Listing 2-8:

Listing 2-8: A named, anonymous, function.

```
var ninja = {
  yell: function yell(n){
    return n > 0 ? yell(n-1) + "a" : "hiy";
  }
};
assert( ninja.yell(4) == "hiyaaaa",
  "Works as we would expect it to!" );

var samurai = { yell: ninja.yell };
var ninja = {};
assert( samurai.yell(4) == "hiyaaaa",
  "The method correctly calls itself." );
```

This ability to name an anonymous function extends even further. It can even be done in normal variable assignments with, seemingly bizarre, results, like in Listing 2-9:

Listing 2-9: Verifying the identity of a named, anonymous, function.

```
var ninja = function myNinja(){
  assert( ninja == myNinja,
  "This function is named two things - at once!" );
};
ninja();
assert( typeof myNinja == "undefined",
  "But myNinja isn't defined outside of the function." );
```

This brings up the most important point: **Anonymous functions can be named but those names are only visible within the functions themselves.**

So while giving named anonymous functions may provide an extra level of clarity and simplicity, the naming process is an extra step that we need to take in order to achieve that. Thankfully, we can circumvent that entirely by using the callee property of the arguments object, as shown in Listing 2-10.

Listing 2-10: Using `arguments.callee` to reference a function itself.

```
var ninja = {
  yell: function(n){
    return n > 0 ? arguments.callee(n-1) + "a" : "hiy";
  }
};
assert( ninja.yell(4) == "hiyaaaa",
  "arguments.callee is the function itself." );
```

`arguments.callee` is available within every function (named or not) and can serve as a reliable way to always access the function itself. Later on in this chapter, and in the chapter on closures, we'll take some additional looks at what can be done with this particular property.

All together, these different techniques for handling anonymous functions will greatly benefit us as we start to scale in complexity, providing us with an extra level of clarity and conciseness. Understanding the object-oriented nature of functions in JavaScript helps to take this knowledge to the next level.

Functions as Objects

In JavaScript functions behave just like objects; they can have properties, they have an object prototype, and generally have all the abilities of plain vanilla objects - the exception being that they are callable.

Let's look at some of the similarities and how they can be made especially useful. To start with, let's look at the assignment of functions to a variable, in Listing 2-11.

Listing 2-11: Assigning a function to a variable.

```
var obj = {};
var fn = function(){};
assert( obj && fn, "Both the object and function exist." );
```

> **Note** One thing that's important to remember is the semicolon after `function(){}` definition. It's a good practice to have semicolons at the end of lines of code - especially so after variable assignments. Doing so with anonymous functions is no exception. When compressing your code (which will be discussed in the distribution chapter) having thorough use of your semicolons will allow for greater flexibility in compression techniques.

Now, just like with an object, we can attach properties to a function, like in Listing 2-12.

Listing 2-12: Attaching properties to a function.

```
var obj = {};
var fn = function(){};
obj.prop = "some value";
fn.prop = "some value";
assert( obj.prop == fn.prop,
  "Both are objects, both have the property." );
```

This aspect of functions can be used in a number of different ways throughout a library (especially so when it comes to topics like event callback management). For now, let's look at some of more interesting things that can be done.

Storing Functions

A challenge when storing a collection of unique functions is determining which functions are actually new and need to be added (A solution to this is very useful for event callback management). An obvious technique would be to store all the functions in an array and then loop through the array looking for duplicate functions. However, this performs poorly and is impractical for most applications. Instead, we can make good use of function properties to achieve an acceptable result, as in Listing 2-13.

Listing 2-13: Storing unique functions within a structure.

```
var store = {
  id: 1,
  cache: {},
  add: function( fn ) {
    if ( !fn.id ) {
      fn.id = store.id++;
      return !!(store.cache[fn.id] = fn);
    }
  }
};

function ninja(){}
assert( store.add( ninja ), "Function was safely added." );
assert( !store.add( ninja ), "But it was only added once." );
```

In Listing 2-13, especially within the add function itself, we check to see if the incoming function already has a unique id assigned to it (which is just an extra property that we could have added to the function, previously). If no such property exists we then generate a unique id and attach it. Finally, we store the function in the cache (we also return a boolean version of the function itself, if the addition was successful, so that we have some way of externally determining its success).

> **Tip** The `!!` construct is a simple way of turning any JavaScript expression into its boolean equivalent. For example: `!!"hello" === true` and `!!0 === false`. In the above example we end up converting a function (which will always be true) into its boolean equivalent: `true`.

Another useful trick, that can be provided by function properties, is the ability of a function to modify itself. This technique could be used to memorize previously-computed values; saving time with future computations.

Self-Memoizing Functions

Memoization is the process of building a function which is capable of remembering its previously computed answers. As a basic example, let's look at a simplistic algorithm for computing prime numbers. Note how the function appears just like a normal function but has the addition of an answer cache to which it saves, and retrieves, solved numbers, like in Listing 2-14.

Listing 2-14: A prime computation function which memorizes its previously-computed values.

```
function isPrime( num ) {
```

```
  if ( isPrime.answers[ num ] != null )
    return isPrime.answers[ num ];

  var prime = num != 1; // Everything but 1 can be prime
  for ( var i = 2; i < num; i++ ) {
    if ( num % i == 0 ) {
      prime = false;
      break;
    }
  }
  return isPrime.answers[ num ] = prime;
}
isPrime.answers = {};

assert( isPrime(5), "Make sure the function works, 5 is prime." );
assert( isPrime.answers[5], "Make sure the answer is cached." );
```

There are two advantages to writing a function in this manner: First, that the user gets the improved speed benefits from subsequent function calls and secondly, that it happens completely seamlessly (the user doesn't have to perform a special request or do any extra initialization in order to make it work). It should be noted that any sort of caching will certainly sacrifice memory in favor of performance so should only be used when absolutely needed

Let's take a look at a modern example, like in Listing 2-15. Querying for a set of DOM elements, by name, is an incredibly common operation. We can take advantage of our new-found function property power by building a cache that we can store the matched element sets in.

Listing 2-15: A simple example of using a self-caching function.

```
function getElements( name ) {
  return getElements.cache[ name ] = getElements.cache[ name ] ||
    document.getElementsByTagName( name );
}
getElements.cache = {};
```

The code ends up being quite simple and doesn't add that much extra complexity to the overall querying process. However, if were to do some performance analysis (the results shown in Table 2-1) upon it we would find that this simple layer of caching yields us a 7x performance increase.

Table 2-1: All time in ms, for 1000 iterations, in a copy of Firefox 3.

Table 3.1.

	Average	Min	Max	Deviation
Original getElements	12.58	12	13	0.50
Cached getElements	1.73	1	2	0.45

Even without digging too deep the usefulness of function properties should be quite evident: We have all of the state and caching information stored in a single location, gaining all the benefits of performance without any extra storage or caching objects polluting the scope. We'll be revisiting this concept throughout the upcoming chapters, as its applicability extends throughout the JavaScript language. Much of a function's power is related to its context, which we'll explore next.

Context

A function's context is, simultaneously, one of its most powerful and confusing features. By having an implicit 'this' variable be included within every function it gives you great flexibility for how the function can be called and executed. There's a lot to understand about context - but understanding it well can make a drastic improvement in the quality of your functional code.

To start, it's important to realize what that the function context represents: The object within which the function is being executed. For example, defining a function within an object structure will ensure that its context always refers to that object - unless otherwise overwritten, like in Listing 2-16:

Listing 2-16: Examining context within a function.

```
var katana = {
  isSharp: true,
  use: function(){
    this.isSharp = !this.isSharp;
  }
};
katana.use()
assert( !katana.isSharp,
  "Verify the value of isSharp has been changed." );
```

However, what about a function that isn't explicitly declared as a property of an object? Then the function's context refers to the global object, as in Listing 2-17.

Listing 2-17: What context refers to within a function.

```
function katana(){
  this.isSharp = true;
}
katana();
assert( isSharp === true,
  "A global object now exists with that name and value." );

var shuriken = {
  toss: function(){
    this.isSharp = true;
  }
};
shuriken.toss();
assert( shuriken.isSharp === true,
  "It's an object property, the value is set within the object." );
```

All of this becomes quite important when we begin to deal with functions in a variety of contexts - knowing what the context will represent affects the end result of your functional code.

> **Note** In ECMAScript 3.1 strict mode a slight change has taken place: Just because a function isn't defined as a property of an object doesn't meant that its context will be the global object. The change is that a function defined within an other function will inherit the context of the outer function. What's interesting about this is that it ends up affecting very little code - having the context of inner functions default to the global object turned out to be quite useless in practice (hence the change).

Now that we understand the basics of function context, it's time to dig deep and explore its most complex usage: The fact that a function context can be redefined anytime that it's called.

JavaScript provides two methods (call and apply), on every function, that can be used to call the function, define its context, and specify its incoming arguments, an example of which can be seen in Listing 2-18.

Listing 2-18: Modifying the context of a function, when call.

```
var object = {};
function fn(){
  return this;
}
assert( fn() == this, "The context is the global object." );
assert( fn.call(object) == object,
  "The context is changed to a specific object." );
```

The two methods are quite similar to each other, as to how they're used, but with a distinction: How they set incoming argument values. Simply, `.call()` passes in arguments individually whereas `.apply()` passes in arguments as an array, as shown in Listing 2-19.

Listing 2-19: Two methods of modifying a function's context.

```
function add(a, b){
  return a + b;
}
assert( add.call(this, 1, 2) == 3,
  ".call() takes individual arguments" );
assert( add.apply(this, [1, 2]) == 3,
  ".apply() takes an array of arguments" );
```

Let's look at some simple examples of this is used, practically, in JavaScript code.

Looping

The first example is that of using call to make an array looping function with a function callback. This is a frequent addition to most JavaScript libraries. The result is a function that can help to simplify the process of looping through an array, as in Listing 2-20.

Listing 2-20: Looping with a function callback.

```
function loop(array, fn){
  for ( var i = 0; i < array.length; i++ )
    if ( fn.call( array, array[i], i ) === false )
      break;
}
var num = 0;
loop([0, 1, 2], function(value, i){
  assert(value == num++,
    "Make sure the contents are as we expect it.");
});
```

The end result to this looping is, arguably, syntactically cleaner than a traditional for loop and can be of great service for when you want to simplify the rote looping process. Additionally, the callback is always able to access the original array via the 'this' context, regardless of its actual instantiation (so it even works on anonymous arrays, like in the example above).

One thing that's especially important about the previous example is the use of a (typically anonymous) function argument being called in response to some action - generally referred to as a 'callback.'

Callbacks are a staple of JavaScript libraries. Overwhelmingly, callbacks are related to asynchronous, or indeterministic, behavior (such as a user clicking a button, an Ajax request completing, or some number of values being found in an array). We'll see this pattern repeat again-and-again in user code.

Fake Array Methods

Let's pretend that we wanted to create an object that behaved similarly to an array (say, a collection of aggregated DOM elements) but take on additional functionality not related to most arrays. One option might be to create a new array every time you wish to create a new version of your objects and imbue it with your desired properties and methods. Generally, however, this can be quite slow. Instead, let's examine the possibility of using a normal object and just giving it the functionality that we desire. For example in Listing 2-21:

Listing 2-21: Simulating array-like methods.

```
<input id="first"/><input id="second"/>
<script>
var elems = {
  find: function(id){
    this.add( document.getElementById(id) );
  },
  length: 0,
  add: function(elem){
    Array.prototype.push.call( this, elem );
  }
};

elems.add("first");
assert( elems.length == 1 && elems[0].nodeType,
  "Verify that we have an element in our stash" );

elems.add("second");
assert( elems.length == 2 && elems[1].nodeType,
  "Verify the other insertion" );
</script>
```

A couple important aspect, to take note of, in this example. We're accessing a native object method (`Array.prototype.push`) and are treating it just like any other function or method - by using `.call()`. The most interesting part is the use of 'this' as the context. Normally, since a push method is part of an array object, setting the context to any other object is treated as if it was an array. This means that when we push this new element on to the current object, its length will be modified and a new numbered property will exist containing the added item.

This behavior is almost subversive, in a way, but it's one that best exemplifies what we're capable of doing with mutable object contexts and offers an excellent segue into discussing the complexities of dealing with function arguments.

Variable Arguments

JavaScript, as a whole, is very flexible in what it can do and much of that flexibility defines the language, as we know it, today. Incidentally, a powerful feature of a JavaScript function is its ability to accept any number of arguments, this flexibility offers the developer great control over how their applications can be written.

Let's take a look at a prime example of how we can use flexible arguments to our advantage.

Min/Max Number in an Array

Finding the smallest, or the largest, values contained within an array can be a tricky problem - there is no, included, method for performing this action in the JavaScript language. This means that we'll have to write our own, from scratch. At first glance it seems as if it might be necessary to loop through the contents of the array in order to determine the correct numbers, however we have a trick up our sleeve. See, the call and apply methods also exist as methods of built-in JavaScript functions - and some built-in methods are able to take any number of arguments. In our case, the methods `Math.min()` and `Math.max()` are able to take any number of arguments and find their appropriate values. Thus, if we were to use `.apply()` on these methods then we'd be able to find the smallest, or largest, value within an array, in one fell swoop, like in Listing 2-22:

Listing 2-22: A generic min and max function for arrays.

```
function smallest(array){
  return Math.min.apply( Math, array );
}
function largest(array){
  return Math.max.apply( Math, array );
}
assert(smallest([0, 1, 2, 3]) == 0, "Locate the smallest value.");
assert(largest([0, 1, 2, 3]) == 3, "Locate the largest value.");
```

Also note that we specify the context as being the `Math` object. This isn't necessary (the min and max methods will continue to work regardless of what's passed in as the context) but there's no reason not to be tidy in this situation.

Function Overloading

All functions are, also, provided access to an important local variable, `arguments`, which gives them the power necessary to handle any number of provided arguments. Even if you only ask for - or expect - a certain number of arguments you'll always be able to access all specified arguments with the `arguments` variable.

Let's take a quick look at an example of effective overloading. In the following we're going to merge the contents of multiple objects into a single, root, object. This can be an effective utility for performing multiple inheritance (which we'll discuss more when we talk about Object Prototypes), an example of which is shown in Listing 2-23.

Listing 2-23: Changing function actions based upon the arguments.

```
function merge(root){
  for ( var i = 1; i < arguments.length; i++ )
    for ( var key in arguments[i] )
      root[key] = arguments[i][key];
  return root;
}

var merged = merge({name: "John"}, {city: "Boston"});
assert( merged.name == "John", "The original name is intact." );
assert( merged.city == "Boston",
  "And the city has been copied over." );
```

Obviously this can be an effective mechanism for, potentially, complex methods. Let's take a look at another example where the use of the arguments variable isn't so clean-cut. In the following example we're building a function which multiplies the largest remaining argument by the first argument - a simple math operation. We can gain an advantage by using the `Math.max()` technique that we used earlier, but there's one small hitch: The `arguments` variable isn't a true array. Even thought it looks and feels like one it lacks basic methods necessary to make this operation possible (like `.slice()`). Let's examine Listing 2-24 and see what we can do to make our desired result.

Listing 2-24: Handling a variable number of function arguments.

```
function multiMax(multi){
  return multi * Math.max.apply( Math,
    Array.prototype.slice.call( arguments, 1 ));
}
assert( multiMax(3, 1, 2, 3) == 9, "3*3=9 (First arg, by largest.)" );
```

Here we use a technique, similar to the one we used with the `Math.max()` method, but allows us to use native Array methods on the `arguments` variable - even though it isn't a true array. The important aspect is that we want to find the largest value in everything but the first argument. The slice method allows us to chop off that first argument, so by applying it via `.call()` to the `arguments` variable we can get our desired result.

Function Length

There's an interesting property on all functions that can be quite powerful, when working with function arguments. This isn't very well known, but all functions have a length property on them. This property equates to the number of arguments that the function is expecting. Thus, if you define a function that accepts a single argument, it'll have a length of 1, like in Listing 2-25:

Listing 2-25: Inspecting the number of arguments that a function is expecting.

```
function makeNinja(name){}
function makeSamurai(name, rank){}
assert( makeNinja.length == 1, "Only expecting a single argument" );
assert( makeSamurai.length == 2, "Multiple arguments expected" );
```

What's important about this technique, however, is that the length property will only represent the arguments that are actually specified. Therefore if you specify the first argument and accept and additional, variable, number of arguments then your length will still only be one (like in the `multiMax` method shown above).

Incidentally, the function length property can be used to great effect in building a quick-and-dirty function for doing simple method overloading. For those of you who aren't familiar with overloading, it's just a way of mapping a single function call to multiple functions based upon the arguments they accept.

Let's look at Listing 2-26 which shows an example of the function that we would like to construct and how we might use it:

Listing 2-26: An example of method overloading using the `addMethod` function from Listing 2-27.

```
function Ninjas(){
  var ninjas = [ "Dean Edwards", "Sam Stephenson", "Alex Russell" ];
  // addMethod is defined in Listing 2-27
  addMethod(this, "find", function(){
    return ninjas;
  });
```

```
  addMethod(this, "find", function(name){
    var ret = [];
    for ( var i = 0; i < ninjas.length; i++ )
      if ( ninjas[i].indexOf(name) == 0 )
        ret.push( ninjas[i] );
    return ret;
  });
  addMethod(this, "find", function(first, last){
    var ret = [];
    for ( var i = 0; i < ninjas.length; i++ )
      if ( ninjas[i] == (first + " " + last) )
        ret.push( ninjas[i] );
    return ret;
  });
}

var ninjas = new Ninjas();
assert( ninjas.find().length == 3, "Finds all ninjas" );
assert( ninjas.find("Sam").length == 1,
  "Finds ninjas by first name" );
assert( ninjas.find("Dean", "Edwards").length == 1,
  "Finds ninjas by first and last name" );
assert( ninjas.find("Alex", "X", "Russell") == null, "Does nothing" );
```

There's a couple things that we can determine about the functionality of `addMethod`. First, we can use it to attach multiple functions to a single property - having it look, and behave, just like a normal method. Second, depending on the number of arguments that are passed in to the method a different bound function will be executed. Let's take a look at a possible solution, as shown in Listing 2-27:

Listing 2-27: A simple means of overloading methods on an object, based upon the specified arguments.

```
function addMethod(object, name, fn){
  var old = object[ name ];
  object[ name ] = function(){
    if ( fn.length == arguments.length )
      return fn.apply( this, arguments )
    else if ( typeof old == 'function' )
      return old.apply( this, arguments );
  };
}
```

Let's dig in and examine how this function works. To start with, a reference to the old method (if there is one) is saved and a new method is put in its place. We'll get into the particulars of how closures work, in the next chapter, but suffice it to say that we can always access the old function within this newly bound one.

Next, the method does a quick check to see if the number of arguments being passed in matches the number of arguments that were specified by the passed-in function. This is the magic. In our above example, when we bound `function(name){...}` it was only expecting a single argument - thus we only execute it when the number of incoming arguments matches what we expect. If the case arises that number of arguments doesn't match, then we attempt to execute the old function. This process will continue until a signature-matching function is found and executed.

This technique is especially nifty because all of these bound functions aren't actually stored in any typical data structure - instead they're all saved as references within closures. Again, we'll talk more about this in the next chapter.

It should be noted that there are some pretty caveats when using this particular technique:

- The overloading only works for different numbers of arguments - it doesn't differentiate based on type, argument names, or anything else.
- All methods will have some function call overhead. Thus, you'll want to take that into consideration in high performance situations.

Nonetheless, this function provides a good example of the potential usefulness of the function length property, to great effect as well.

Function Type

To close out this look at functions I wanted to take a quick peek at a common cross-browser issue relating to them. Specifically relating to how you determine if an object, or property, is actually a function that you can call.

Typically, and overwhelmingly for most cases, the `typeof` statement is more than sufficient to get the job done, for example in Listing 2-28:

Listing 2-28: A simple way of determining the type of a function.

```
function ninja(){}
assert( typeof ninja == "function",
  "Functions have a type of function" );
```

This should be the de-facto way of checking if a value is a function, however there exist a few cases where this is not so.

- Firefox 2 and 3: Doing a typeof on the HTML <object/> element yields an inaccurate "function" result (instead of "object"), like so: `typeof objectElem == "function"`.
- Firefox 2: A little known feature: You can call regular expressions as if they were functions, like so: `/test/("a test")`. This can be useful, however it also means that `typeof /test/ == "function"` in Firefox 2 (was changed to "object" in 3).
- Internet Explorer: When attempting to find the type of a function that was part of another window (such as an iframe) and no longer exists, its type will be reported as 'unknown'.
- Safari 3: Safari considers a DOM NodeList to be a function, like so: `typeof document.body.childNodes == "function"`

Now, for these specific cases, we need a solution which will work in all of our target browsers, allowing us to detect if those particular functions (and non-functions) report themselves correctly.

There's a lot of possible avenues for exploration here, unfortunately almost all of the techniques end up in a dead-end. For example, we know that functions have an `apply()` and `call()` method - however those methods don't exist on Internet Explorers problematic functions. One technique, however, that does work fairly well is to convert the function to a string and determine its type based upon its serialized value, for example in Listing 2-29:

Listing 2-29: A more complex, but more resistant, way of determining if a value is a function.

```
function isFunction( fn ) {
  return Object.prototype.toString.call(fn) ===
    "[object Function]";
```

```
}
```

Now this function isn't perfect, however in situations (like the above), it'll pass all the cases that we need, giving us a correct value to work with. There is one notable exception, however: Internet Explorer reports methods of DOM elements with a type of "object", like so: `typeof domNode.getAttribute == "object"` and `typeof inputElem.focus == "object"` - this particular technique does not cover this case.

The implementation of the function requires a little bit of magic, in order to make it work correctly. We start by accessing the internal toString method of the `Object.prototype` this particular method, by default, is designed to return a string that represents the internal representation of the object (such as a Function or String). Using this method we can then call it against any object to access its true type (this technique expands beyond just determining if something is a function and also works for Strings, RegExp, Date, and other objects).

The reason why we don't just call `fn.toString()` to try and get this result is two-fold: 1) Individual objects are likely to have their own toString implementation and 2) Most types in JavaScript already have a pre-defined toString that overrides the method provided by `Object.prototype`. By accessing this internal method directly we can end up with the exact information that we need.

This is just a quick taste of the strange world of cross-browser scripting. While it can be quite challenging the result is always rewarding: Allowing you to painlessly write cross browser applications with little concern for the painful minutia.

Summary

In this chapter we took a look at various, fascinating, aspects of how functions work in JavaScript. While their use is completely ubiquitous, understanding of their inner-workings is essential to the writing of high-quality JavaScript code.

Specifically, within this chapter, we took a look at different techniques for defining functions (and how different techniques can benefit clarity and organization). We also examined how recursion can be a tricky problem, within anonymous functions, and looked at how to solve it with advanced properties like `arguments.callee`. Then we explored the importance of treating functions like objects; attaching properties to them to store additional data and the benefits of doing such. We also worked up an understanding of how function context works, and can be manipulated using `apply` and `call`. We then looked at a number of examples of using a variable number of arguments, within functions, in order to make them more powerful and useful. Finally, we closed with an examination of cross-browser issues that relate to functions.

In all, it was a thorough examination of the fundamentals of advanced function usage that gives us a great lead-up into understanding closures, which we'll do in the next chapter.

Chapter 4. Closures

In this chapter:

- The basics of how closures work
- Using closures to simplify development
- Improving the speed of code using closures
- Fixing common scoping issues with closures

Closures are one of the defining features of JavaScript. Without them, JavaScript would likely be another hum-drum scripting experience, but since that's not the case, the landscape of the language is forever shaped by their inclusion.

Traditionally, closures have been a feature of purely functional programming languages and having them cross over into mainstream development has been particularly encouraging, and enlightening. It's not uncommon to find closures permeating JavaScript libraries, and other advanced code bases, due to their ability to drastically simplify complex operations.

How closures work

Simply: A closure is a way to access and manipulate external variables from within a function. Another way of imagining it is the fact that a function is able to access all the variables, and functions, declared in the same scope as itself. The result is rather intuitive and is best explained through code, like in Listing 3-1.

Listing 3-1: A few examples of closures.

```
var outerValue = true;

function outerFn(arg1){
  var innerValue = true;

  assert( outerFn && outerValue, "These come from the closure." );
  assert( typeof otherValue === "undefined",
    "Variables defined late are not in the closure." );

  function innerFn(arg2){
    assert( outerFn && outerValue,
  "These still come from the closure." );
    assert( innerFn && innerValue && arg1,
  "All from a closure, as well." );
  }

  innerFn(true);
}

outerFn(true);

var otherValue = true;

assert( outerFn && outerValue,
  "Globally-accessible variables and functions." );
```

Note that Listing 3-1 contains two closures: function `outerFn()` includes a closured reference to itself and the variable `outerValue`. function `innerFn()` includes a closured reference to the variables `outerValue`, `innerValue`, and `arg1` and references to the functions `outerFn()` and `innerFn()`.

It's important to note that while all of this reference information isn't immediately visible (there's no "closure" object holding all of this information, for example) there is a direct cost to storing and referencing your information in this manner. It's important to remember that each function that accesses information via a closure immediately has at "ball and chain," if you will, attached to them carrying this information. So while closures are incredibly useful, they certainly aren't free of overhead.

Private Variables

One of the most common uses of closures is to encapsulate some information as a "private variable," of sorts. Object-oriented code, written in JavaScript, is unable to have traditional private variables (properties of the object that are hidden from outside uses). However, using the concept of a closure, we can arrive at an acceptable result, as in Listing 3-2.

Listing 3-2: An example of keeping a variable private but accessible via a closure.

```
function Ninja(){
  var slices = 0;

  this.getSlices = function(){
    return slices;
  };
  this.slice = function(){
    slices++;
  };
}

var ninja = new Ninja();
ninja.slice();
assert( ninja.getSlices() == 1,
  "We're able to access the internal slice data." );
assert( ninja.slices === undefined,
  "And the private data is inaccessible to us." );
```

In Listing 3-2 we create a variable to hold our state, `slices`. This variable is only accessible from within the `Ninja()` function (including the methods `getSlices()` and `slice()`). This means that we can maintain the state, within the function, without letting it be directly accessed by the user.

Callbacks and Timers

Another one of the most beneficial places for using closures is when you're dealing with callbacks or timers. In both cases a function is being called at a later time and within the function you have to deal with some, specific, outside data. Closures act as an intuitive way of accessing data, especially when you wish to avoid creating extra variables just to store that information. Let's look at a simple example of an Ajax request, using the jQuery JavaScript Library, in Listing 3-3.

Listing 3-3: Using a closure from a callback in an Ajax request.

```
<div></div>
<script src="jquery.js"></script>
<script>
var elem = jQuery("div");
```

```
elem.html("Loading...");

jQuery.ajax({
  url: "test.html",
  success: function(html){
    assert( elem, "The element to append to, via a closure." );
    elem.html( html );
  }
});
</script>
```

There's a couple things occurring in Listing 3-3. To start, we're placing a loading message into the div to indicate that we're about to start an Ajax request. We have to do the query, for the div, once to edit its contents - and would typically have to do it again to inject the contents when the Ajax request completed. However, we can make good use of a closure to save a reference to the original jQuery object (containing a reference to the div element), saving us from that effort.

Listing 3-4 has a slightly more complicated example, creating a simple animation.

Listing 3-4: Using a closure from a timer interval.

```
<div id="box" style="position:absolute;">Box!</div>
<script>
var elem = document.getElementById("box");
var count = 0;

var timer = setInterval(function(){
  if ( count < 100 ) {
    elem.style.left = count + "px";
    count++;
  } else {
    assert( count == 100,
  "Count came via a closure, accessed each step." );
    assert( timer, "The timer reference is also via a closure." );
    clearInterval( timer );
  }
}, 10);
</script>
```

What's especially interesting about the Listing 3-4 is that it only uses a single (anonymous) function to accomplish the animation, and three variables, accessed via a closure. This structure is particularly important as the three variables (the DOM element, the pixel counter, and the timer reference) all must be maintained across steps of the animation. This example is a particularly good one in demonstrating how the concept of closures is capable of producing some surprisingly intuitive, and concise, code.

Now that we've had a good introduction, and inspection, into closures, let's take a look at some of the other ways in which they can be applied.

Enforcing Function Context

Last chapter, when we discussed function context, we looked at how the `.call()` and `.apply()` methods could be used to manipulate the context of a function. While this manipulation can be incredibly useful, it can also be, potentially, harmful to object-oriented code. Observe Listing 3-5 in which an object method is bound to an element as an event listener.

Listing 3-5: Binding a function, with a specified context, to an element.

```
<button id="test">Click Me!</button>
<script>
var Button = {
  click: function(){
    this.clicked = true;
  }
};

var elem = document.getElementById("test");
elem.addEventListener("click", Button.click, false);
trigger( elem, "click" );
assert( elem.clicked,
  "The clicked property was accidentally set on the element" );
</script>
```

Now, Listing 3-5 fails because the '`this`' inside of the click function is referring to whatever context the function currently has. While we intend it to refer to the Button object, when it's re-bound to another object context the `clicked` property is transferred along with it. In this particular example `addEventListener` redefines the context to be the target element, which causes us to bind the `clicked` property to the wrong object.

Now, there's a way around this. We can enforce a particular function to always have our desired context, using a mix of anonymous functions, `.apply()`, and closures. Observe Listing 3-6 - same as Listing 3-5 - but actually working with our desired result, now.

Listing 3-6: An alternative means of enforcing function context when binding a function.

```
<button id="test">Click Me!</button>
<script>
function bind(context, name){
  return function(){
    return context[name].apply(context, arguments);
  };
}

var Button = {
  click: function(){
    this.clicked = true;
  }
};

var elem = document.getElementById("test");
elem.addEventListener("click", bind(Button, "click"), false);
trigger( elem, "click" );
assert( Button.clicked,
  "The clicked property was set on our object" );
</script>
```

The secret here is the new `bind()` method that we're using. This method is designed to create - and return a new function which will continually enforce the context that's been specified. This particular function makes the assumption that we're going to be modifying an existing method (a function attached as a property to an object). With that assumption, we only need to request two pieces of information: The object which contains the method (whose context will be enforced) and the name of the method.

With this information we create a new, anonymous, function and return it as the result. This particular function uses a closure to encapsulate the context and method name from the original method call. This means that we can now use this new (returned) function in whatever context we wish, as its original context will always be enforced (via the `.apply()` method).

The `bind()` function, above, is a simple version of the function popularized by the Prototype JavaScript Library. Prototype promotes writing your code in a clean, classicial-style, object-oriented way. Thus, most object methods have this particular "incorrect context" problem when re-bound to another object. The original version of the method looks something like the code in Listing 3-7.

Listing 3-7: An example of the function binding code used in the Prototype library.

```
Function.prototype.bind = function(){
  var fn = this, args = Array.prototype.slice.call(arguments),
    object = args.shift();

  return function(){
    return fn.apply(object,
      args.concat(Array.prototype.slice.call(arguments)));
  };
};

var myObject = {};
function myFunction(){
  return this == myObject;
}

assert( !myFunction(), "Context is not set yet" );

var aFunction = myFunction.bind(myObject)
assert( aFunction(), "Context is set properly" );
```

Note that this method is quite similar to the function implemented in Listing 3-7, but with a couple, notable, additions. To start, it attaches itself to all functions, rather than presenting itself as a globally-accessible function. You would, in turn, use the function like so: `myFunction.bind(myObject)`. Additionally, with this method, you are able to bind arguments to the anonymous function. This allows you to pre-specify some of the arguments, in a form of partial function application (which we'll discuss in the next section).

It's important to realize that `.bind()` isn't meant to be a replacement for methods like `.apply()` or `.call()` - this is mostly due to the fact that its execution is delayed, via the anonymous function and closure. However, this important distinction makes them especially useful in the aforementioned situations: event handlers and timers.

Partially Applying Functions

Partially applying a function is a, particularly, interesting technique in which you can pre-fill-in arguments, to a function, before it is ever executed. In effect, partially applying a function returns a new function which you can call. This is best understood through an example, as in Listing 3-8.

Listing 3-8: Partially applying arguments to a native function (using the code from Listing 3-10).

```
String.prototype.csv = String.prototype.split.partial(/,\s*/);

var results = ("John, Resig, Boston").csv();
assert( results[1] == "Resig",
```

```
  "The text values were split properly" );
```

In Listing 3-8 we've taken a common function - a String's `.split()` method - and have pre-filled-in the regular expression upon which to split. The result is a new function, `.csv()` that we can call at any point to convert a list of comma-separated values into an array. Filling in the first couple arguments of a function (and returning a new function) is typically called currying. With that in mind, let's look at how the curry method is, roughly, implemented in the Prototype library, in Listing 3-9.

Listing 3-9: An example of a curry function (filling in the first specified arguments).

```
Function.prototype.curry = function() {
  var fn = this, args = Array.prototype.slice.call(arguments);
  return function() {
    return fn.apply(this, args.concat(
      Array.prototype.slice.call(arguments)));
  };
};
```

This is a good example of using a closure to remember state. In this case we want to remember the arguments that were pre-filled-in (`args`) and transfer them to the newly-constructed function. This new function will have the filled-in arguments and the new `arguments` concat'd together and passed in. The result is a method that allows us to fill in arguments, giving us a new function that we can use.

Now, this style of partial function application is perfectly useful, but we can do better. What if we wanted to fill in any missing argument from a given function - not just the first ones. Implementations of this style of partial function application have existed in other languages but Oliver Steele was one of the first to demonstrate it with his Functional.js (http://osteele.com/sources/javascript/functional/) library. Listing 3-10 has a possible implementation.

Listing 3-10: An example of a more-complex partial application function (filling in the specified arguments, leaving undefined arguments blank).

```
Function.prototype.partial = function(){
  var fn = this, args = Array.prototype.slice.call(arguments);
  return function(){
    var arg = 0;
    for ( var i = 0; i < args.length && arg < arguments.length; i++ )
      if ( args[i] === undefined )
        args[i] = arguments[arg++];
    return fn.apply(this, args);
  };
};
```

This implementation is fundamentally similar to the `.curry()` method, but has a couple important differences. Notably, when called, the user can specify arguments that will be filled in later by specifying `undefined`, for it. To accommodate this we have to increase the ability of our arguments-merging technique. Effectively, we have to loop through the arguments that are passed in and look for the appropriate gaps, filling in the missing pieces that were specified.

We already had the example of constructing a string splitting function, above, but let's look at some other ways in which this new functionality could be used. To start we could construct a function that's able to be easily delayed, like in Listing 3-11.

Listing 3-11: Using partial application on a timer function.

```
var delay = setTimeout.partial(undefined, 10);
```

```
delay(function(){
  assert( true,
  "A call to this function will be temporarily delayed." );
});
```

This means that we now have a new function, named `delay`, which we can pass another function in to, at any time, to have it be called asynchronously (after 10 milliseconds).

We could, also create a simple function for binding events, as in Listing 3-12.

Listing 3-12: Using partial application on event binding.

```
var bindClick = document.body.addEventListener
  .partial("click", undefined, false);

bindClick(function(){
  assert( true, "Click event bound via curried function." );
});
```

This technique could be used to construct simple helper methods for event binding in a library. The result would be a simpler API where the end-user wouldn't be inconvenienced by unnecessary function arguments, reducing them to a single function call with the partial application.

In the end we've used closures to easily, and simply, reduce the complexity of some code, easily demonstrating some of the power that functional JavaScript programming has.

Overriding Function Behavior

A fun side effect of having so much control over how functions work, in JavaScript, is that you can completely manipulate their internal behavior, unbeknownst to the user. Specifically there are two techniques: The modification of how existing functions work (no closures needed) and the production of new self-modifying functions based upon existing static functions.

Memoization

Memoization is the process of building a function which is capable of remembering its previously computed answers. As we demonstrated in the chapter on functions, it's pretty straight-forwards implementing these into an existing function. However, we don't always have access to the functions that we need to optimize.

Let's examine a method, `.memoized()`, in Listing 3-13 that we can use to remember return values from an existing function. There are no closures involved here, only functions.

Listing 3-13: An example of function value memoization.

```
Function.prototype.memoized = function(key){
  this._values = this._values || {};
  return this._values[key] !== undefined ?
    this._values[key] :
    this._values[key] = this.apply(this, arguments);
};

function isPrime( num ) {
  var prime = num != 1;
```

```
  for ( var i = 2; i < num; i++ ) {
    if ( num % i == 0 ) {
      prime = false;
      break;
    }
  }
  return prime;
}

assert( isPrime.memoized(5),
  "Make sure the function works, 5 is prime." );
assert( isPrime._values[5], "Make sure the answer is cached." );
```

We're using the same `isPrime()` function from when we talked about functions - it's still painfully slow and awkward, making it a prime candidate for memoization.

Our ability to introspect into an existing function is limited. However, by adding a new `.memoized()` method we do have the ability to modify and attach properties that are associated with the function itself. This allows us to create a data store (`._values`) in which all of our pre-computed values can be saved. This means that the first time the `.memoized()` method is called, looking for a particular value, a result will be computed and stored from the originating function. However upon subsequent calls that value will be saved and be returned immediately.

Let's walk through the `.memoized()` method and examine how it works.

To start, before doing any computation or retrieval of values, we must make sure that a data store exists and that it is attached to the parent function itself. We do this via a simple short-circuiting:

```
this._values = this._values || {};
```

If the computed values already exist, then just re-save that reference to the property, otherwise create the new data store (an object) and save that to the function property.

To retrieve the value we have to look into the data store and see if anything exists and, if not, compute and save the value. What's interesting about the above code is that we do the computation and the save in a single step. The value is computed with the `.apply()` call to the function and is saved directly into the data store. However, this statement is within the return statement meaning that the resulting value is also returned from the parent function. So the whole chain of events: computing the value, saving the value, returning the value is done within a single logical unit of code.

Now that we have a method for memoizing the values coming in-and-out of an existing function let's explore how we can use closures to produce a new function, capable of having all of its function calls be memoized, in Listing 3-14.

Listing 3-14: A different technique for building a memoized function (using the code from Listing 3-13).

```
Function.prototype.memoize = function(){
  var fn = this;
  return function(){
    return fn.memoized.apply( fn, arguments );
  };
};

var isPrime = (function( num ) {
  var prime = num != 1;
  for ( var i = 2; i < num; i++ ) {
```

```
    if ( num % i == 0 ) {
      prime = false;
      break;
    }
  }
  return prime;
}).memoize();

assert( isPrime(5), "Make sure the function works, 5 is prime." );
assert( isPrime._values[5], "Make sure the answer is cached." );
```

Listing 3-14 builds upon our previous `.memoized()` method constructing a new method: `.memoize()`. This method returns a function which, when called, will always be the memoized results of the original function.

Note that within the `.memoize()` method we construct a closure remembering the original function that we want to memoize. By remembering this function we can return a new function which will always call our, previously constructed, `.memoized()` method; giving us the appearance of a normal, memoized, function.

In Listing 3-14 we show a, comparatively, strange case of defining a new function, when we define `isPrime()`. Since we want `isPrime` to always be memoized we need to construct a temporary function whose results won't be memoized (much like what we did in the first example - but in a temporary fashion). We take this anonymous, prime-figuring, function and memoize it immediately, giving us a new function which is assigned to the `isPrime` name. We'll discuss this construct, in depth, in the `(function() {})()` section. Note that, in this case, it is impossible to compute if a number is prime in a non-memoized fashion. Only a single `isPrime` function exists and it completely encapsulates the original function, hidden within a closure.

Listing 3-14 is a good demonstration of obscuring original functionality via a closure. This can be particularly useful (from a development perspective) but can also be crippling: If you obscure too much of your code then it becomes unextendable, which can be undesirable. However, hooks for later modification often counter-act this. We'll discuss this matter in depth throughout the book.

Function Wrapping

Function wrapping is a means of encapsulating the functionality of a function, and overwriting it, in a single step. It is best used when you wish to override some previous behavior of a function, while still allowing certain use-cases to still execute. The use is best demonstrated when implementing pieces of cross-browser code, like in Listing 3-15 from Prototype:

Listing 3-15: Wrapping an old function with a new piece of functionality.

```
function wrap(object, method, wrapper){
  var fn = object[method];
  return object[method] = function(){
    return wrapper.apply(this, [ fn.bind(this) ].concat(
      Array.prototype.slice.call(arguments)));
  };
}

// Example adapted from Prototype
if (Prototype.Browser.Opera) {
  wrap(Element.Methods, "readAttribute", function(orig, elem, attr){
    return attr == "title" ?
```

```
      elem.title :
      orig(elem, attr);
  });
}
```

The `wrap()` function overrides an existing method (in this case `readAttribute`) replacing it with a new function. However, this new function still has access to the original functionality (in the form of the `original` argument) provided by the method. This means that a function can be safely overriden without any loss of functionality.

In Listing 3-15 the Prototype library is using the wrap function to implement a piece of browser-specific functionality. Specifically they're attempting to work around some bug with Opera's implementation of accessing title attributes. Their technique is interesting, as opposed to having a large if/else block within their `readAttribute` function (debatably messy and not a good separation of concerns) they, instead, opt to completely override the old method, simply implementing this fix, and deferring the rest of the functionality back to the original function. As with the previous example, in which we overwrote, and encapsulated, the `isPrime` function, so we are doing here, using a closure.

Let's dig in to how the `wrap()` function works. To start we save a reference to the original method in `fn`, which we'll access later via the anonymous function's closure. We then proceed to overwrite the method with our new function. This new function will execute our `wrapper` function (brought to us via a closure) and pass in our desired arguments. Notably, however, the first argument is the original function that we're overriding. Additionally, the function's context has been overridden using our previously-implemented `.bind()` method, enforcing its context to be the same as the wrapper's.

The full result is a reusable function that we can use to override existing functionality of object methods in an unobtrusive manner, all making efficient use of closures.

(function(){})()

So much of advanced JavaScript, and the use of closures, centers around one simple construct: `(function(){})()`. This single piece of code is incredibly versatile and ends up giving the JavaScript language a ton of unforeseen power. Since the syntax is a little strange, let's deconstruct what's going on.

First, let's examine the use of `(...)()`. We know that we can call a function using the `functionName()` syntax. In this case the extra set of parentheses sort of hides us from whatever is inside of it. In the end we just assume that whatever is in there will be a reference to a function and executed. For example, the following is also valid: `(functionName)()`. Thus, instead of providing a reference to an existing function, we provide an anonymous function, creating: `(function(){})()`.

The result of this code is a code block which is instantly created, executed, and discarded. Additionally, since we're dealing with a function that can have a closure, we also have access to all outside variables. As it turns out, this simple construct ends up becoming immensely useful, as we'll see in the following sections.

Temporary Scope and Private Variables

Using the executed anonymous function we can start to build up interesting enclosures for our work. Since the function is executed immediately, and all the variables inside of it are kept inside of it, we can use this to create a temporary scope within which our state can be maintained, like in Listing 3-16.

> **Remember** Variables in JavaScript are scoped to the function within which they were defined. By creating a temporary function we can use this to our advantage and create a temporary scope for our variables to live in.

Listing 3-16: Creating a temporary enclosure for persisting a variable.

```
(function(){
  var numClicks = 0;

  document.addEventListener("click", function(){
    alert( ++numClicks );
  }, false);
})();
```

Since the above anonymous function is executed immediately the click handler is also bound right away. Additionally, a closure is created allowing the `numClicks` variable to persist with the handler. This is the most common way in which executed anonymous functions are used, just as a simple wrapper. However it's important to remember that since they are functions they can be used in interesting ways, like in Listing 3-17.

Listing 3-17: An alternative to the example in Listing 3-16, returning a value from the enclosure.

```
document.addEventListener("click", (function(){
  var numClicks = 0;

  return function(){
    alert( ++numClicks );
  };
})(), false);
```

This is a, debatably, more complicated version of our first example in Listing 3-16. In this case we're, again, creating an executed anonymous function but this time we return a value from it. Since this is just like any other function that value is returned and passed along to the `addEventListener` method. However, this function that we've created also gets the necessary `numClicks` variable via its closure.

This technique is a very different way of looking at scope. In most languages you can scope things based upon the block which they're in. In JavaScript variables are scope based upon the function they're in. However, with this simple construct, we can now scope variables to block, and sub-block, levels. The ability to scope some code to a unit as small as an argument within a function call is incredibly powerful and truly shows the flexibility of the language.

Listing 3-18 has a quick example from the Prototype JavaScript library:

Listing 3-18: Using the anonymous function wrapper as a variable shortcut.

```
(function(v) {
  Object.extend(v, {
    href:        v._getAttr,
    src:         v._getAttr,
    type:        v._getAttr,
    action:      v._getAttrNode,
    disabled:    v._flag,
    checked:     v._flag,
    readonly:    v._flag,
    multiple:    v._flag,
    onload:      v._getEv,
    onunload:    v._getEv,
    onclick:     v._getEv,
    ...
  });
})(Element._attributeTranslations.read.values);
```

In this case they're extending an object with a number of new properties and methods. In that code they could've created a temporary variable for `Element._attributeTranslations.read.values` but, instead, they chose to pass it in as the first argument to the executed anonymous function. This means that the first argument is now a reference to this data structure and is contained within this scope. This ability to create temporary variables within a scope is especially useful once we start to examine looping.

Loops

One of the most useful applications of executed anonymous functions is the ability to solve a nasty issue with loops and closures. Listing 3-19 has a common piece of problematic code:

Listing 3-19: A problematic piece of code in which the iterator is not maintained in the closure.

```
<div></div>
<div></div>
<script>
var div = document.getElementsByTagName("div");
for ( var i = 0; i < div.length; i++ ) {
  div[i].addEventListener("click", function(){
    alert( "div #" + i + " was clicked." );
  }, false);
}
</script>
```

In Listing 3-19 we encounter a common issue with closures and looping, namely that the variable that's being enclosed (`i`) is being updated after the function is bound. This means that every bound function handler will always alert the last value stored in `i` (in this case, '2'). This is due to the fact that closures only remember references to variables - not their actual values at the time at which they were called. This is an important distinction and one that trips up a lot of people.

Not to fear, though, as we can combat this closure craziness with another closure, like in Listing 3-20.

Listing 3-20: Using an anonymous function wrapper to persist the iterator properly.

```
<div></div>
<div></div>
<script>
var div = document.getElementsByTagName("div");
for ( var i = 0; i < div.length; i++ ) (function(i){
  div[i].addEventListener("click", function(){
    alert( "div #" + i + " was clicked." );
  }, false);
})(i);
</script>
```

Using this executed anonymous function as a wrapper for the for loop (replacing the existing `{...}` braces) we can now enforce that the correct value will be enclosed by these handlers. Note that, in order to achieve this, we pass in the iterator value to the anonymous function and then re-declare it in the arguments. This means that within the scope of each step of the for loop the `i` variable is defined anew, giving our click handler closure the value that it expects.

Library Wrapping

The final concept that closures, and executed anonymous functions, encapsulate is an important one to JavaScript library development. When developing a library it's incredibly important that you don't pollute

the global namespace with unnecessary variables, especially ones that are only temporarily used. This is a point at which the executed anonymous functions can be especially useful: Keep as much of the library private as possible and only selectively introduce variables into the global namespace. The jQuery library actually does a good job of this, completely enclosing all of its functionality and only introducing the variables it needs, like `jQuery`, in Listing 3-21.

Listing 3-21: Placing a variable outside of a function wrapper.

```
(function(){
  var jQuery = window.jQuery = function(){
    // Initialize
  };

  // ...
})();
```

Note that there are two assignments done here, this is intentional. First, the jQuery constructor is assigned to `window.jQuery` in an attempt to introduce it as a global variable. However, that does not guarantee that that variable will be the only named `jQuery` within our scope, thus we assign it to a local variable, `jQuery`, to enforce it as such. That means that we can use the `jQuery` function name continuously within our library while, externally, someone could've re-named the global `jQuery` object to something else. Since all of the functions and variables that are required by the library are nicely encapsulated it ends up giving the end user a lot of flexibility in how they wish to use it.

However, that isn't the only way in which that type of definition could be done, another is shown in Listing 3-22.

Listing 3-22: An alternative means of putting a variable in the outer scope.

```
var jQuery = (function(){
  function jQuery(){
    // Initialize
  }

  // ...

  return jQuery;
})();
```

The code in Listing 3-22 would have the same effect as what was shown in the Listing 3-21, but structured in a different manner. Here we define a `jQuery` function within our anonymous scope, use it and define it, then return it back out the global scope where it is assigned to a variable, also named `jQuery`. Oftentimes this particular technique is preferred, if you're only exporting a single variable, as it's clearer as to what the result of the function is.

In the end a lot of this is left to developer preference, which is good, considering that the JavaScript language gives you all the power you'll need to make any particular application structure happen.

Summary

In this chapter we dove in to how closures work in JavaScript. We started with the basics, looking at how they're implemented and then how to use them within an application. We looked at a number of cases where closures were particularly useful, including the definition of private variables and in the use of callbacks and timers. We then explored a number of advanced concepts in which closures helped to sculpt the JavaScript language including forcing function context, partially applying functions, and overriding

function behavior. We then did an in-depth exploration of the `(function(){})()` construct which, as we learned, has the power to re-define how the language is used. In total, understanding closures will be an invaluable asset when developing complex JavaScript applications and will aid in solving a number of common problems that we encounter.

Chapter 5. Function Prototypes

In this chapter:

- Examining function instantiation
- Exploring function prototypes
- A number of common gotchas
- A full system for building classes and doing inheritance

Function prototypes are a convenient way to quickly attaching properties to an instance of a function. It can be used for multiple purposes, the primary of which is as an enhancement to Object-Oriented programming. Prototypes are used throughout JavaScript as a convenient means of carrying functionality, applying it to instances of objects. The predominant use of prototypes is in producing a classical-style form of Object-Oriented code and inheritance.

Instantiation and Prototypes

All functions have a `prototype` property which, by default, contains an empty object. This property doesn't serve a purpose until the function is instantiated. In order to understand what it does it's important to remember the simple truth about JavaScript: Functions serve a dual purpose. They can behave both as a regular function and as a "class" (where they can be instantiated).

Instantiation

Let's examine the simplest case of using the prototype property to add functionality to an instance of a function, in Listing 4-1.

Listing 4-1: Creating an instance of an function that has a prototyped method.

```
function Ninja(){}

Ninja.prototype.swingSword = function(){
  return true;
};

var ninja1 = Ninja();
assert( ninja1 === undefined,
  "Is undefined, not an instance of Ninja." );

var ninja2 = new Ninja();
assert( ninja2.swingSword(), "Method exists and is callable." );
```

There's two points to learn from Listing 4-1: First that in order to produce an instance of a function (something that's an object, has instance properties, and has prototyped properties) you have to call the function with the `new` operator. Second we can see that the `swingSword` function has now become a property of the `ninja2` `Ninja` instance.

In some respects the base function (in this case, `Ninja`) could be considered a "constructor" (since it's called whenever the `new` operator is used upon it). This also means that when the function is called with

the `new` operator it's `this` context is equal to the instance of the object itself. Let's examine this a little bit more in Listing 4-2.

Listing 4-2: Extending an object with both a prototyped method and a method within the constructor function.

```
function Ninja(){
  this.swung = false;

  // Should return true
  this.swingSword = function(){
    return !this.swung;
  };
}

// Should return false, but will be overridden
Ninja.prototype.swingSword = function(){
  return this.swung;
};

var ninja = new Ninja();
assert( ninja.swingSword(),
  "Calling the instance method, not the prototype method." );
```

In Listing 4-2 we're doing an operation very similar to the previous example: We add a new method by adding it to the prototype property. However, we also add a new method within the constructor function itself. The order of operations is as such:

1. Properties are bound to the object instance from the prototype.

2. Properties are bound to the object instance within the constructor function.

This is why the `swingSword` function ends up returning true (since the binding operations within the constructor always take precedence). Since the `this` context, within the constructor, refers to the instance itself we can feel free to modify it until our heart's content.

An important thing to realize about a function's prototype is that, unlike instance properties which are static, it will continue to update, live, as its changed - even against all existing instance of the object. For example if we were to take some of the code in Listing 4-2 and re-order it, it would still work as we would expect it to, as shown in Listing 4-3.

Listing 4-3: The prototype continues to update instance properties live even after they've already been created.

```
function Ninja(){
  this.swung = true;
}

var ninja = new Ninja();

Ninja.prototype.swingSword = function(){
  return this.swung;
};

assert( ninja.swingSword(), "Method exists, even out of order." );
```

These, seamless, live updates gives us an incredible amount of power and extensibility, to a degree at which isn't typically found in other languages. Allowing for these live updates it makes it possible for you to create a functional framework which users can extend with further functionality - even well after any objects have been instantiated.

Object Type

Since we now have this new instance of our function it becomes important to know which function constructed the object instance, so we can refer back to later (possibly even performing a form of type checking, if need be), as in Listing 4-4.

Listing 4-4: Examining the type of an instance and its constructor.

```
function Ninja(){}

var ninja = new Ninja();

assert( typeof ninja == "object",
  "However the type of the instance is still an object." );
assert( ninja instanceof Ninja,
  "The object was instantiated properly." );
assert( ninja.constructor == Ninja,
  "The ninja object was created by the Ninja function." );
```

In Listing 4-4 we start by examining the type of a function instance. Note that the `typeof` operator isn't of much use to us here; all instance will be objects, thus always returning `"object"` as its result. However, the `instanceof` operator really helps in this case, giving us a clear way to determine if an instance was created by a particular function.

On top of this we also make use of an object property: `.constructor`. This is a property that exists on all objects and offers a reference back to the original function that created it. We can use this to verify the origin of the instance (much like how we do with the `instanceof` operator). Additionally, since this is just a reference back to the original function, we can, once-again, instantiate a new `Ninja` object, like in Listing 4-5.

Listing 4-5: Instantiating a new object using only a reference to its constructor.

```
var ninja = new Ninja();
var ninja2 = new ninja.constructor();

assert( ninja2 instanceof Ninja, "Still a ninja object." );
```

What's especially interesting about Listing 4-5 is that we can do this without ever having access to the original function; taking place completely behind the scenes.

> **Note** The `.constructor` property of an object can be changed - but it doesn't have any immediate, or obvious, purpose (since it's primary purpose is to inform as to where it was constructed from).

Inheritance and the Prototype Chain

There's an additional feature of the `instanceof` operator that we can use to our advantage when performing object inheritance. In order to make use of it, however, we need to understand how the prototype chain works in JavaScript, an example of which is shown in Listing 4-6.

Listing 4-6: Different ways of copying functionality onto a function's prototype.

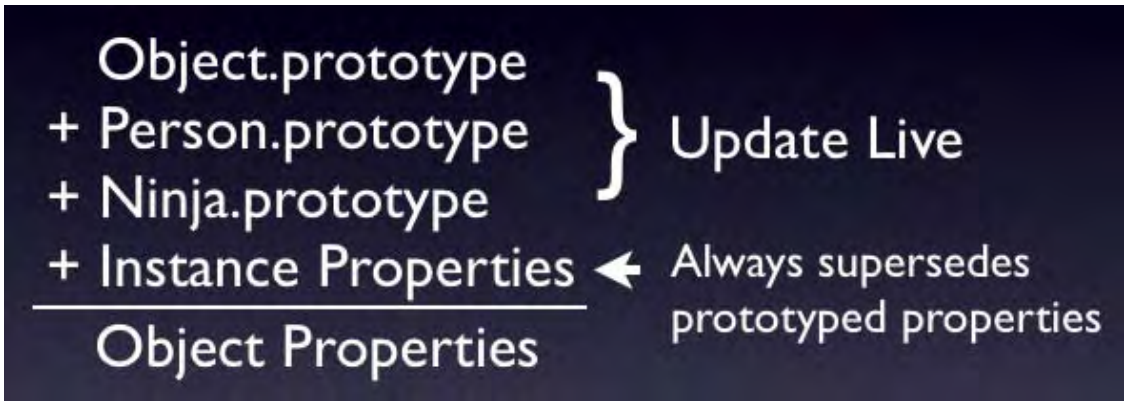
```
function Person(){}
Person.prototype.dance = function(){};

function Ninja(){}

// Achieve similar, but non-inheritable, results
Ninja.prototype = Person.prototype;
Ninja.prototype = { dance: Person.prototype.dance };

// Only this maintains the prototype chain
Ninja.prototype = new Person();

var ninja = new Ninja();
assert( ninja instanceof Ninja,
  "ninja receives functionality from the Ninja prototype" );
assert( ninja instanceof Person, "... and the Person prototype" );
assert( ninja instanceof Object, "... and the Object prototype" );
```

Since the prototype of a function is just an object there are multiple ways of copying functionality (such as properties or methods). However only one technique is capable of creating a prototype 'chain': `SubFunction.prototype = new SuperFunction();`. This means that when you perform an `instanceof` operation you can determine if the function inherits the functionality of any object in its prototype chain.

> **Note** Make sure that you don't use the `Ninja.prototype = Person.prototype;` technique. When doing this any changes to the Ninja prototype will also change the Person prototype (since they're the same object) - which is bound to have undesired side effects.

An additional side-effect of doing prototype inheritance in this manner is that all inherited function prototypes will continue to update live. The result is something akin to what's shown in Figure 4-1.

Figure 4-1: The order in which properties are bound to an instantiated object.



It's important to note from Figure 4-1 is that our object has properties that are inherited from the Object prototype. This is important to note: All native objects constructors (such as Object, Array, String, Number, RegExp, and Function) have prototype properties which can be manipulated and extended (which makes sense, since each of those object constructors are functions themselves). This proves to be an incredibly powerful feature of the language. Using this you can extend the functionality of the language itself, introducing new, or missing, pieces of the language.

For example, one such case where this becomes quite useful is with some of the features of JavaScript 1.6. This version of the JavaScript language introduces a couple helper methods, including some for Arrays. One such method is `forEach` which allows to iterate over the properties of an array, calling a function on every property. This can be especially handy for situations where we want to plug in different pieces of functionality without changing the overall looping structure. We can duplicate this functionality, right now, completely circumventing the need to wait until the next version of the language is ready, like in Listing 4-7.

Listing 4-7: Implementing the JavaScript 1.6 array forEach method in a future-compatibile manner.

```
if (!Array.prototype.forEach) {
  Array.prototype.forEach = function(fn, thisp){
    for ( var i = 0; i < this.length; i++ ) {
      fn.call( thisp || null, this[i], i, this );
    }
  };
}

["a", "b", "c"].forEach(function(value, index, array){
  assert( value, "Is in position " + index + " out of " +
    (array.length - 1) );
});
```

Since all the built-in objects include these prototypes it ends up giving you all the power necessary to extend the language to your desire.

An important point to remember when implementing properties or methods on native objects is that introducing them is every bit as dangerous as introducing new variables into the global scope. Since there's only ever one instance of a native object there still significant possibility for naming collisions to occur.

When implementing features on native prototypes that are forward-looking (such as the previously-mentioned implementation of Array.prototype.forEach) there's a strong possibility that your implementation won't match the final implementation (causing issues to occur when a browser finally does implement the functionality correctly). You should always take great care when treading in those waters.

HTML Prototypes

There's a fun feature in Internet Explorer 8, Firefox, Safari, and Opera which provides base functions representing objects in the DOM. By making these accessible the browser is providing you with the ability to extend any HTML node of your choosing, like this in the Listing 4-8.

Listing 4-8: Adding a new method to all HTML elements via the HTMLElement prototype.

```
<div id="a">I'm going to be removed.</div>
<div id="b">Me too!</div>
<script>
HTMLElement.prototype.remove = function(){
  if ( this.parentNode )
    this.parentNode.removeChild( this );
};

// Old way
var a = document.getElementById("a");
a.parentNode.removeChild( a );
```

```
// New way
document.getElementById("b").remove();
</script>
```

More information about this particular feature can be found in the HTML 5 specification:

- http://www.whatwg.org/specs/web-apps/current-work/multipage/section-elements.html

One library that makes heavy use of this feature is the Prototype library, adding all forms of functionality onto existing DOM elements (including the ability to inject HTML, manipulate CSS, amongst other features).

The most important thing to realize, when working with these prototypes, is that they don't exist in older versions of Internet Explorer. If that isn't a target platform for you, then these features should serve you well.

Another point that often comes in contention is the question of if HTML elements can be instantiated directly from their constructor function, perhaps something like this:

```
var elem = new HTMLElement();
```

However, that does not work, at all. Even though browsers expose the root function and prototype they selectively disable the ability to actually call the root function (presumably to limit element creation to internal functionality, only).

Save for the overwhelming difficulty that this feature presents, in platform compatibility, the benefits of clean code can be quite dramatic and should be strongly investigated in applicable situations.

Gotchas

As with most things, in JavaScript, there are a series of gotchas associated with prototypes, instantiation, and inheritance. Some of them can be worked around, however, most of them will simply require a dampening of your excitement.

Extending Object

Perhaps the most egregious mistake that someone can make with prototypes is to extend the native `Object.prototype`. This difficulty is that when you extend this prototype ALL objects receive those additional properties. This is especially problematic since when you iterate over the properties of the object these new properties appear, causing all sorts of unexpected behavior, as shown in Listing 4-9.

Listing 4-9: The unexpected behavior of adding extra properties to the Object prototype.

```
Object.prototype.keys = function(){
  var keys = [];
  for ( var i in this )
    keys.push( i );
  return keys;
};

var obj = { a: 1, b: 2, c: 3 };
```

```
assert( obj.keys().length == 4,
  "The 3 existing properties plus the new keys method." );
```

There is one workaround, however. Browsers provide a method called `hasOwnProperty` which can be used to detect properties which are actually on the object instance and not be imported from a prototype. The result can be seen in Listing 4-10.

Listing 4-10: Using the hasOwnProperty method to tame Object prototype extensions.

```
Object.prototype.keys = function(){
  var keys = [];
  for ( var i in this )
    if ( this.hasOwnProperty( i ) )
      keys.push( i );
  return keys;
};

var obj = { a: 1, b: 2, c: 3 };

assert( obj.keys().length == 3,
  "Only the 3 existing properties are included." );
```

Now just because it's possible to work around this issue doesn't mean that it should be abused. Looping over the properties of an object is an incredibly common behavior and it's uncommon for people to use `hasOwnProperty` within their code. Generally you should avoid using it if only but in the most controlled situations (such as a single developer working on a single site with code that is completely controlled).

Extending Number

It's safe to extend most native prototypes (save for Object, as previously mentioned). One other problematic natives is that of Number. Due to how numbers, and properties of numbers, are parsed by the JavaScript engine the result can be rather confusing, like in Listing 4-11.

Listing 4-11: Adding a method to the Number prototype.

```
Number.prototype.add = function(num){
  return this + num;
};

var n = 5;
assert( n.add(3) == 8,
  "It works fine if the number is in a variable." );

assert( (5).add(3) == 8,
  "Also works if a number is wrapping in parentheses." );

// Won't work, causes a syntax error
// assert( 5.add(3) == 8, "Doesn't work, causes error." );
```

This can be a frustrating issue to deal with since the logic behind it can be rather obtuse. There have been libraries that have continue to include Number prototype functionality, regardless, and have simply stipulated how they should be used (Prototype being one of them). That's certainly an option, albeit one that'll have to be rectified with good documentation and clear tutorials.

Sub-classing Native Objects

Another tricky point is in the sub-classing of native objects. The one object that's quite simple to sub-class is Object (since it's the root of all prototype chains to begin with). However once you start wanting to sub-class other native objects the situation becomes less clear-cut. For example, with Array, everything works as you might expect it to, as shown in Listing 4-12.

Listing 4-12: Inheriting functionality from the Array object.

```
function MyArray(){}
MyArray.prototype = new Array();

var mine = new MyArray();
mine.push(1, 2, 3);
assert( mine.length == 3,
  "All the items are on our sub-classed array." );
assert( mine instanceof Array,
  "Verify that we implement Array functionality." );
```

Of course, until you attempt to perform this functionality in Internet Explorer. For whatever reason the native Array object does not allow for it to be inherited from (the length property is immutable - causing all other pieces of functionality to become stricken).

In general it's a better strategy to implement individual pieces of functionality from native objects, rather than attempt to sub-class them completely, such as implementing the push method directly, like in Listing 4-13.

Listing 4-13: Simulating Array functionality but without the true sub-classing.

```
function MyArray(){}
MyArray.prototype.length = 0;

(function(){
  var methods = ['push', 'pop', 'shift', 'unshift',
    'slice', 'splice', 'join'];

  for ( var i = 0; i < methods.length; i++ ) (function(name){
    MyArray.prototype[ name ] = function(){
      return Array.prototype[ name ].apply( this, arguments );
    };
  })(methods[i]);
})();

var mine = new MyArray();
mine.push(1, 2, 3);
assert( mine.length == 3,
  "All the items are on our sub-classed array." );
assert( !(mine instanceof Array),
  "We aren't sub-classing Array, though." );
```

In Listing 4-13 we end up calling the native Array methods and using `.apply()` to trick them into thinking that they're working on an array object. The only property that we have to implement ourselves is the `length` (since that's the one property that must remain mutable - the feature that Internet Explorer does not provide).

Instantiation

Let's start by looking at a simple case of a function which will be instantiated and populated with some values, in Listing 4-14.

Listing 4-14: The result of leaving off the new operator from a function call.

```
function User(first, last){
  this.name = first + " " + last;
}

var user = User("John", "Resig");
assert( typeof user == "undefined",
  "Since new wasn't used, the instance is undefined." );
```

On a quick observation of the above code it isn't immediately obvious that the function is actually something that is meant to be instantiated with the 'new' operator. Thus, a new user might try to call the function without the operator, causing severely unexpected results (e.g. 'user' would be undefined).

If a function is meant to be instantiated, and isn't, it can in turn end up polluting the current scope (frequently the global namespace), causing even more unexpected results. For example, observe the following code in Listing 4-15.

Listing 4-15: An example of accidentally introducing a variable into the global namespace.

```
function User(first, last){
  this.name = first + " " + last;
}

var name = "Resig";
var user = User("John", name);

assert( name == "John Resig",
  "The name variable is accidentally overridden." );
```

This can result in a debugging nightmare. The developer may try to interact with the name variable again (being unaware of the error that occurred from mis-using the User function) and be forced to dance down the horrible non-deterministic wormhole that's presented to them (Why is the value of their variables changing underneath their feet?).

Listing 4-16 shows a solution to all of the above problems, with a simple technique.

Listing 4-16: Enforcing the correct context with arguments.callee and instanceof.

```
function User(first, last){
  if ( !(this instanceof arguments.callee) )
    return new User(first, last);

  this.name = first + " " + last;
}

var name = "Resig";
var user = User("John", name);

assert( user,
```

```
  "This was defined correctly, even if it was by mistake." );
assert( name == "Resig", "The right name was maintained." );
```

This new function seems straightforward but takes a little bit of consideration. What we're doing is checking to see if we're currently inside of an instantiated version of the function, rather than just the function itself. Let's look at a simpler example, in Listing 4-17.

Listing 4-17: Determining if we're inside of an instantiated function, or not.

```
function test(){
    return this instanceof arguments.callee;
}

assert( !test(), "We didn't instantiate, so it returns false." );
assert( new test(), "We did instantiate, returning true." );
```

Using this bit of logic we can now construct our structure within the constructor function. This means that if the 'this instanceof arguments.callee' expression is true then we need to behave like we normally would, within a constructor (initializing property values, etc.). But if the expression is false, we need to instantiate the function, like so:

```
return new User(first, last);
```

It's important to return the result, since we're just inside a normal function at the point. This way, no matter which way we now call the User() function, we'll always get an instance back.

```
// New Style:
assert( new User(), "Normal instantiation works." );
assert( User(), "As does a normal function call." );

// Old Style:
assert( new User(), "Normal instantiation works." );
assert( !User(), "But a normal function call causes problems." );
```

This can be a trivial addition to most code bases but the end result is a case where there's no longer a delineation between functions that are meant to be instantiated, and not, which can be quite useful.

Class-like Code

A common desire, for most developers, is a simplification - or abstraction - of JavaScript's inheritance system into one that they are more familiar with. This tends to head towards the realm of, what a typical developer would consider to be, Classes. While JavaScript doesn't support classical inheritance natively.

Generally there are a couple features that developers crave:

- A system which trivializes the syntax building new constructor functions and prototypes
- An easy way of performing prototype inheritance
- A way of accessing methods overridden by the function's prototype

For example, if you had a system which made this process easier, Listing 4-18 shows an example of what you can do with it:

Listing 4-18: An example of classical-style inheritance, using the code from Listing 4-19.

```
var Person = Object.subClass({
  init: function(isDancing){
    this.dancing = isDancing;
  },
  dance: function(){
    return this.dancing;
  }
});

var Ninja = Person.subClass({
  init: function(){
    this._super( false );
  },
  dance: function(){
    // Call the inherited version of dance()
    return this._super();
  },
  swingSword: function(){
    return true;
  }
});

var p = new Person(true);
assert( p.dance(), "Method returns the internal true value." );

var n = new Ninja();
assert( n.swingSword(), "Get true, as we expect." );
assert( !n.dance(),
  "The inverse of the super method's value - false." );

// Should all be true
assert( p instanceof Person && n instanceof Ninja &&
  n instanceof Person,
  "The objects inherit functionality from the correct sources." );
```

A couple things to note about this implementation:

- Creating a constructor had to be simple (in this case simply providing an init method does the trick).
- In order to create a new 'class' you must extend (sub-class) an existing constructor function.
- All of the 'classes' inherit from a single ancestor: Object. Therefore if you want to create a brand new class it must be a sub-class of Object (completely mimicking the current prototype system).
- And the most challenging one: Access to overridden methods had to be provided (with their context properly set). You can see this with the use of `this._super()`, above, calling the original `init()` and `dance()` methods of the `Person` super-class.

There are a number of different JavaScript classical-inheritance-simulating libraries that already exist. Out of all them there are two that stand up above the others: The implementations within base2 and Prototype. While they contain a number of advanced features the absolute core is the important part of the solution and is what's distilled in the implementation. The code in Listing 4-19 helps to enforce the notion of 'classes' as a structure, maintains simple inheritance, and allows for the super method calling.

Listing 4-19: A simple class creation library.

```
// Inspired by base2 and Prototype
(function(){
  var initializing = false,
    // Determine if functions can be serialized
    fnTest = /xyz/.test(function(){xyz;}) ? /\b_super\b/ : /.*/;

  // Create a new Class that inherits from this class
  Object.subClass = function(prop) {
    var _super = this.prototype;

    // Instantiate a base class (but only create the instance,
    // don't run the init constructor)
    initializing = true;
    var proto = new this();
    initializing = false;

    // Copy the properties over onto the new prototype
    for (var name in prop) {
      // Check if we're overwriting an existing function
      proto[name] = typeof prop[name] == "function" &&
        typeof _super[name] == "function" && fnTest.test(prop[name]) ?
        (function(name, fn){
          return function() {
            var tmp = this._super;

            // Add a new ._super() method that is the same method
            // but on the super-class
            this._super = _super[name];

            // The method only need to be bound temporarily, so we
            // remove it when we're done executing
            var ret = fn.apply(this, arguments);
            this._super = tmp;

            return ret;
          };
        })(name, prop[name]) :
        prop[name];
    }

    // The dummy class constructor
    function Class() {
      // All construction is actually done in the init method
      if ( !initializing && this.init )
        this.init.apply(this, arguments);
    }

    // Populate our constructed prototype object
    Class.prototype = proto;

    // Enforce the constructor to be what we expect
    Class.constructor = Class;

    // And make this class extendable
```

```
    Class.subClass = arguments.callee;

    return Class;
  };
})();
```

The two trickiest parts of Listing 4-19 are the "initializing/don't call init" and "create _super method" portions. Having a good understanding of what's being achieved in these areas will help with your understanding of the full implementation.

Initialization

In order to simulate inheritance with a function prototype we use the traditional technique of creating an instance of the super-class function and assigning it to the prototype. Without using the above it would look something like the code in 4-20.

Listing 4-20: Another example of maintaining the prototype chain.

```
function Person(){}
function Ninja(){}
Ninja.prototype = new Person();
// Allows for instanceof to work:
(new Ninja()) instanceof Person
```

What's challenging about Listing 4-20, though, is that all we really want is the benefits of 'instanceof', not the whole cost of instantiating a Person object and running its constructor. To counteract this we have a variable in our code, `initializing`, that is set to true whenever we want to instantiate a class with the sole purpose of using it for a prototype.

Thus when it comes time to actually construct the function we make sure that we're not in an initialization mode and run the init method accordingly:

```
if ( !initializing )
  this.init.apply(this, arguments);
```

What's especially important about this is that the init method could be running all sorts of costly startup code (connecting to a server, creating DOM elements, who knows) so circumventing this ends up working quite well.

Super Method

When you're doing inheritance, creating a class that inherits functionality from a super-class, a frequent desire is the ability to access a method that you've overridden. The final result, in this particular implementation, is a new temporary method (`._super`) which is only accessible from within a sub-classes' method, referencing the super-classes' associated method.

For example, if you wanted to call a super-classes' constructor you could do that with the technique shown in Listing 4-21.

Listing 4-21: An example of calling the inherited super method.

```
var Person = Class.extend({
  init: function(isDancing){
    this.dancing = isDancing;
  }
});
```

```
var Ninja = Person.extend({
  init: function(){
    this._super( false );
  }
});

var p = new Person(true);
assert( p.dancing, "The person is dancing." );

var n = new Ninja();
assert( !n.dancing, "The ninja is never dancing." );
```

Implementing this functionality is a multi-step process. To start, note the object literal that we're using to extend an existing class (such as the one being passed in to `Person.extend`) needs to be merged on to the base `new Person` instance (the construction of which was described previously). During this merge we do a rather complex check: Is the property that we're attempting to merge a function and is what we're replacing also a function - and does our function contain any use of the `_super` method? If that's the case then we need to go about creating a way for our super method to work.

In order to determine if our function contains the use of a `_super` method we must use a trick provided by most browsers: function decompilation. This occurs when you convert a function to a string, you end up with the contents of the function in a serialized form. We're simply using this as a shortcut so that we won't have to add the extra super-method overhead if we don't have to. In order to determine this we must first see if we can serialize methods properly (currently, only Opera Mobile doesn't do the serialization properly - but it's better to be safe here). That gives us this line:

```
fnTest = /xyz/.test(function(){xyz;}) ? /\b_super\b/ : /.*/
```

All this is doing is determining if the string version of the function contains the variable that we're expecting and, if so, producing a working regular expression and if not, producing a regular expression that'll match any function.

Note that we create an anonymous closure (which returns a function) that will encapsulate the new super-enhanced method. To start we need to be a good citizen and save a reference to the old `this._super` (disregarding if it actually exists) and restore it after we're done. This will help for the case where a variable with the same name already exists (don't want to accidentally blow it away).

Next we create the new `_super` method, which is just a reference to the method that exists on the super-class' prototype. Thankfully we don't have to make any additional changes, or re-scoping, here as the context of the function will be set automatically when it's a property of our object (`this` will refer to our instance as opposed to the super-class').

Finally we call our original method, it does its work (possibly making use of `_super` as well) after which we restore `_super` to its original state and return from the function.

Now there's a number of ways in which a similar result, to the above, could be achieved (there are implementations that have bound the super method to the method itself, accessible from `arguments.callee`) but this particular technique provides the best mix of usability and simplicity.

Summary

Function prototypes and prototypal inheritance are two features that, when used properly, provide an incredible amount of wealth to developers. By allowing for greater amounts of control and structure to the code your JavaScript applications will easily improve in clarity and quality. Additionally, due to the inherit extensibility that prototypes provide you'll have an easy platform to build off of for future development.

Chapter 6. Timers

In this chapter:

- An overview of how timers work
- In depth examination of timer speed
- Processing large tasks using timers
- Managing animations with timers
- Better testing with timers

Timers are one of the poorly understood, and often misused, feature of JavaScript that actually can provide great benefit to the developer in complex applications. Timers provide the ability to asynchronously delay the execution of a piece of code by a number of milliseconds. Since JavaScript is, by nature, single-threaded (only one piece of JavaScript code can be executing at a time) timers provide a way to dance around this restriction resulting in a rather oblique way of executing code. (It should be mentioned that timers will never interrupt another currently-running timer.)

One point that's interesting is that timers are not, contrary to popular belief, part of the actual JavaScript language but, rather, part of the objects and methods that a web browser introduces. This means that if you choose to use JavaScript in another, non-browser, environment it's very likely that timers will not exist and you'll have to implement your own version of them using implementation-specific features (such as Threads in Rhino).

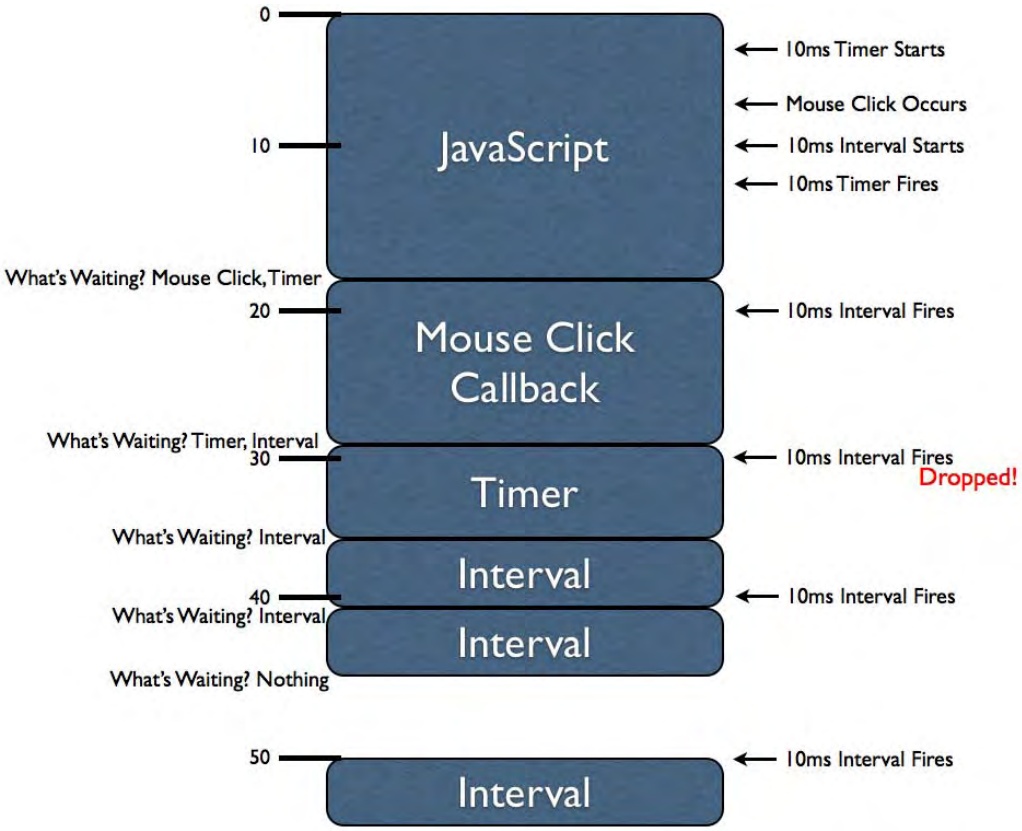
How Timers Work

At a fundamental level it's important to understand how timers work. Often times they behave unintuitively because of the single thread which they are in. Let's start by examining the three functions to which we have access that can construct and manipulate timers.

- `var id = setTimeout(fn, delay);` - Initiates a single timer which will call the specified function after the delay. The function returns a unique ID with which the timer can be cancelled at a later time.
- `var id = setInterval(fn, delay);` - Similar to `setTimeout` but continually calls the function (with a delay every time) until it is cancelled.
- `clearInterval(id);`, `clearTimeout(id);` - Accepts a timer ID (returned by either of the aforementioned functions) and stops the timer callback from occurring.

In order to understand how the timers work internally there's one important concept that needs to be explored: timer delay is not guaranteed. Since all JavaScript in a browser executes on a single thread asynchronous events (such as mouse clicks and timers) are only run when there's been an opening in the execution. This is best demonstrated with a diagram, like in Figurer 5-1.

Figure 5-1: An explanation of timers.



There's a lot of information in Figure 5-1 to digest but understanding it completely will give you a better realization of how asynchronous JavaScript execution works. This diagram is one dimensional: vertically we have the (wall clock) time, in milliseconds. The blue boxes represent portions of JavaScript being executed. For example the first block of JavaScript executes for approximately 18ms, the mouse click block for approximately 11ms, and so on.

Since JavaScript can only ever execute one piece of code at a time (due to its single-threaded nature) each of these blocks of code are "blocking" the progress of other asynchronous events. This means that when an asynchronous event occurs (like a mouse click, a timer firing, or an XMLHttpRequest completing) it gets queued up to be executed later (how this queueing actually occurs surely varies from browser-to-browser, so consider this to be a simplification).

To start with, within the first block of JavaScript, two timers are initiated: a 10ms `setTimeout` and a 10ms `setInterval`. Due to where and when the timer was started it actually fires before we actually complete the first block of code. Note, however, that it does not execute immediately (it is incapable of doing that, because of the threading). Instead that delayed function is queued in order to be executed at the next available moment.

Additionally, within this first JavaScript block we see a mouse click occur. The JavaScript callbacks associated with this asynchronous event (we never know when a user may perform an action, thus it's consider to be asynchronous) are unable to be executed immediately thus, like the initial timer, it is queued to be executed later.

After the initial block of JavaScript finishes executing the browser immediately asks the question: What is waiting to be executed? In this case both a mouse click handler and a timer callback are waiting. The browser then picks one (the mouse click callback) and executes it immediately. The timer will wait until the next possible time, in order to execute.

Note that while mouse click handler is executing the first interval callback executes. As with the timer its handler is queued for later execution. However, note that when the interval is fired again (when the timer handler is executing) this time that handler execution is dropped. If you were to queue up all interval callbacks when a large block of code is executing the result would be a bunch of intervals executing with no delay between them, upon completion. Instead browsers tend to simply wait until no more interval handlers are queued (for the interval in question) before queueing more.

We can, in fact, see that this is the case when a third interval callback fires while the interval, itself, is executing. This shows us an important fact: Intervals don't care about what is currently executing, they will queue indiscriminately, even if it means that the time between callbacks will be sacrificed.

Finally, after the second interval callback is finished executing, we can see that there's nothing left for the JavaScript engine to execute. This means that the browser now waits for a new asynchronous event to occur. We get this at the 50ms mark when the interval fires again. This time, however, there is nothing blocking its execution, so it fires immediately.

Let's take a look at an example to better illustrate the differences between `setTimeout` and `setInterval`, in Listing 5-1.

Listing 5-1: Two examples of repeating timers.

```
setTimeout(function(){
  /* Some long block of code... */
  setTimeout(arguments.callee, 10);
}, 10);

setInterval(function(){
  /* Some long block of code... */
}, 10);
```

The two pieces of code in Listing 5-1 may appear to be functionally equivalent, at first glance, but they are not. Notably the `setTimeout` code will always have at least a 10ms delay after the previous callback execution (it may end up being more, but never less) whereas the `setInterval` will attempt to execute a callback every 10ms regardless of when the last callback was executed.

There's a lot that we've learned here, let's recap:

- JavaScript engines only have a single thread, forcing asynchronous events to queue waiting for execution.
- `setTimeout` and `setInterval` are fundamentally different in how they execute asynchronous code.
- If a timer is blocked from immediately executed it will be delayed until the next possible time of execution (which will be longer than the desired delay).
- Intervals may execute back-to-back with no delay if they take long enough to execute (longer than the specified delay).

All of this is incredibly important knowledge to build off of. Knowing how a JavaScript engine works, especially with the large number of asynchronous events that typically occur, makes for a great foundation when building an advanced piece of application code.

Minimum Timer Delay and Reliability

While it's pretty obvious that you can have timer delays of seconds, minutes, hours - or whatever large interval you desire - what isn't obvious is the smallest timer delay that you can choose.

At a certain point a browser is simply incapable of providing a fine-enough resolution on the timers in order to handle them accurately (since they, themselves, are restricted by the timings of the operating system). Across most browsers, however, it's safe to say that the minimum delay interval is around 10-15ms.

We can come to that conclusion by performing some simple analysis upon the presumed timer intervals across platforms. For example if we analyze a `setInterval` delay of 0ms we can find what the minimum delay is for most browsers, as in Figure 5-2 and Figure 5-3.

Figure 5-2: OSX Browsers, From Top Left: Firefox 2, Safari 3, Firefox 3, Opera 9.

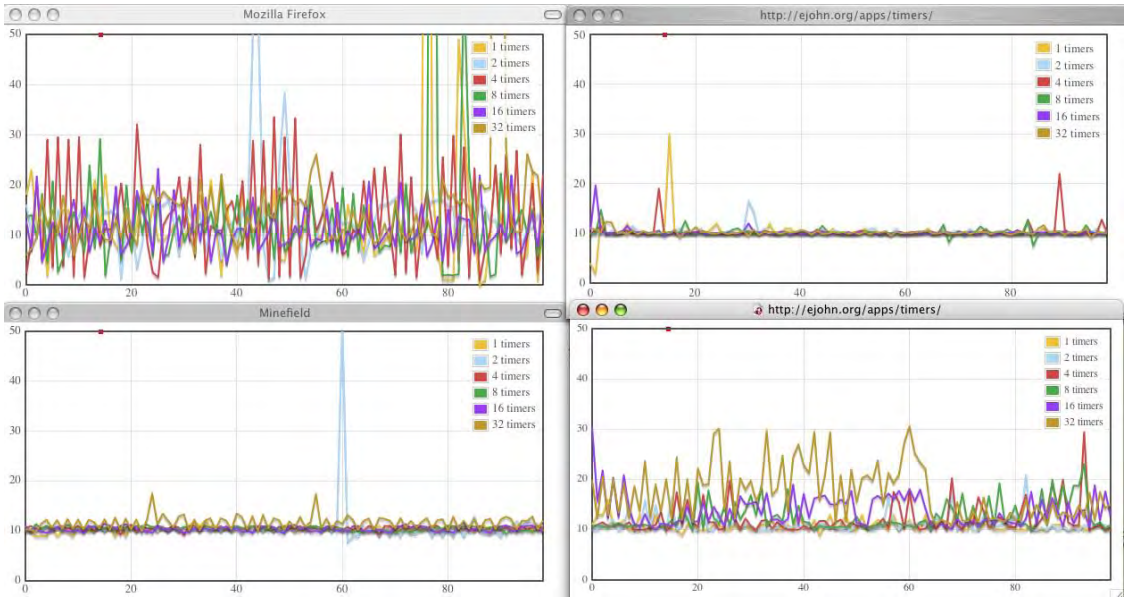
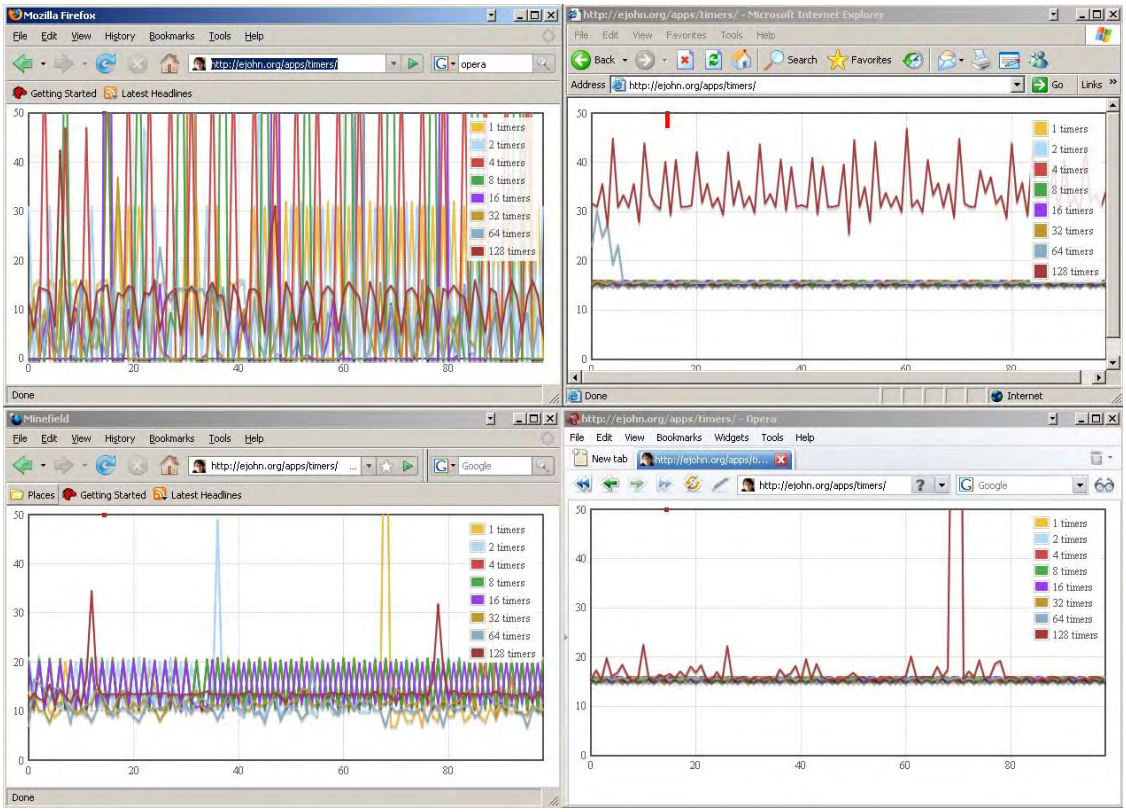


Figure 5-3: Windows Browsers, From Top Left: Firefox 2, Internet Explorer 6, Firefox 3, Opera 9



In Figure 5-2 and Figure 5-3 the lines and numbers indicate the number of simultaneous intervals being executed by the browser.

We can come to some conclusions: Browsers all have a 10ms minimum delay on OSX and a (approximately) 15ms delay on Windows. We can achieve either of those values by providing '0' (or any other number below 10ms) as the value to the timer delay.

There is one catch, though: Internet Explorer is incapable of providing a 0ms delay to a `setInterval` (even though it'll happily work with a `setTimeout`). Whenever a 0ms delay is provided the interval turns into a single `setTimeout` (only executing the callback once). We can get around this by providing a 1ms delay instead. Since all browsers automatically round up any value below their minimum delay using a 1ms delay is just as safe as effective using 0ms - only moreso (since Internet Explorer will now work).

There's some other things that we can learn from these charts. The most important aspect is simply a reinforcement from what we learned previously: browsers do not guarantee the exact interval that you specify. Browsers like Firefox 2 and Opera 9 (on OSX) have difficulties providing a reliable timer execution rate. A lot of this has to do with how browsers perform garbage collection on JavaScript (the significant improvement that was made in Firefox 3 in JavaScript execution and garbage collection is immediately obvious in these results).

So while very small delay values can be provided by a browser the exact accuracy is not always guaranteed. This needs to be taken into account in your applications when using timers (if the difference between 10ms and 15ms is problematic then you might have to re-think how your application code is structured).

Computationally-Expensive Processing

Probably the largest gotcha in complex JavaScript application development is the single-threaded nature of the user interface and the script that powers it. While JavaScript is executing user interaction becomes,

at best, sluggish and, at worst, unresponsive - causing the browser to hang (all updates to the rendering of a page are suspended while JavaScript is executing). Because of this fact reducing all complex operations (any more than a few hundred milliseconds) into manageable portions becomes a necessity. Additionally some browsers will produce a dialog warning the user that a script has become 'unresponsive' if it has run for, at least, 5 seconds non-stop (browsers such as Firefox and Opera do this). The iPhone actually kills any script running for more than 5 seconds (without even opening a dialog).

These issues are, obviously, less than desirable. Producing an unresponsive user interface is never good. However there will, almost certainly, arise situations in which you'll need to process a significant amount of data (situations such as manipulating a couple thousands DOM elements can cause this to occur).

This is where timers becomes especially useful. Since timers are capable of, effectively, suspending execution of a piece of JavaScript until a later time it also prevents the browser from hanging (as long as the individual pieces of code aren't enough to cause the browser to hang). Taking this into account we can now convert normal, intensive, loops and operations into non-blocking operations. Let's look at Listing 5-2 where this type of operation would become necessary.

Listing 5-2: A long-running task.

```
<table><tbody></tbody></table>
<script>
// Normal, intensive, operation
var table = document.getElementsByTagName("tbody")[0];
for ( var i = 0; i < 2000; i++ ) {
  var tr = document.createElement("tr");
  for ( var t = 0; t < 6; t++ )
    var td = document.createElement("td");
    td.appendChild( document.createTextNode("" + t) );
    tr.appendChild( td );
  }
  table.appendChild( tr );
}
</script>
```

In this example we're creating a total of 26,000 DOM nodes, populating a table with numbers. This is incredibly expensive and will likely hang the browser preventing the user from performing normal interactions. We can introduce timers into this situation to achieve a different, and perhaps better, result, as shown in Listing 5-3

Listing 5-3: Using a timer to break up a long-running task.

```
<table><tbody></tbody></table>
<script>
var table = document.getElementsByTagName("tbody")[0];
var i = 0, max = 1999;

setTimeout(function(){
  for ( var step = i + 500; i < step; i++ ) {
    var tr = document.createElement("tr");
    for ( var t = 0; t < 6; t++ )
      var td = document.createElement("td");
      td.appendChild( document.createTextNode("" + t) );
      tr.appendChild( td );
    }
```

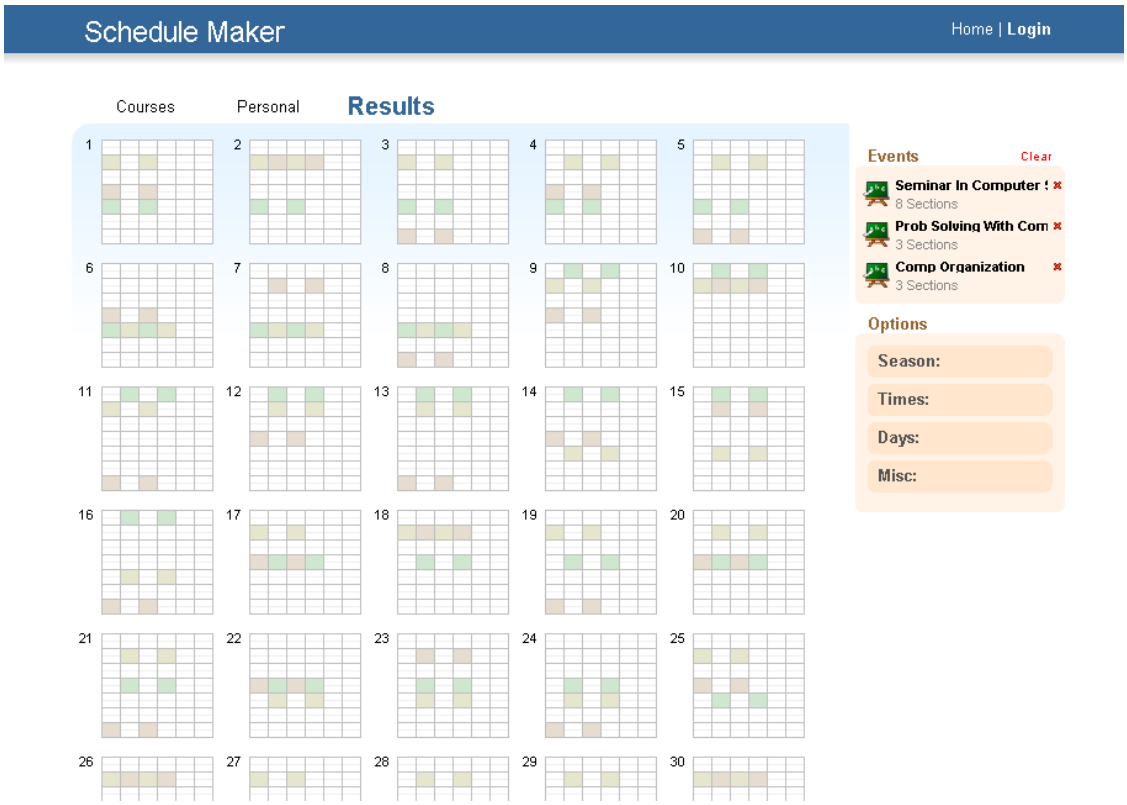
```
    table.appendChild( tr );
  }

  if ( i < max ) {
    setTimeout( arguments.callee, 0 );
  }
}, 0);
</script>
```

In our modified example we've broken up our intense operation into 4 smaller operations, each creating 6,500 DOM nodes. These operations are much less likely to interrupt the flow of the browser. Worst case these numbers can always be tweaked (changing 500 to 250, for example, would give us 8 steps of 3,500 DOM nodes each). What's most impressive, however, is just how little our code has to change in order to accommodate this new, asynchronous, system. We have to do a little more work in order to make sure that the correct number of elements are being constructed (and that the looping doesn't continue on forever) but beyond that the code looks very similar to what we started with. Note that we make use of a closure to maintain the iterator state from code segment to code segment undoubtedly without the use of closures this code would be much more complicated.

There's one perceptible change when using this technique, in relation to the original technique, namely that a long browser hang is now replaced with 4 visual updates of the page. While the browser will attempt to do these segments as quickly as possible it will also render the DOM changes after each step of the timer (just as it would after one large bulk update). Most of the time it's imperceptible to the user to see these types of updates occur but it's important to remember that they occur.

One situation in which this technique has served me particularly well was in an application that I constructed to compute schedule permutations for college students. Originally the application was a typical CGI (communicating from the client to the server, where the schedules were computed and sent back) but I converted it to move all schedule computation to the client-side. A view of the schedule computation screen can be seen in Figure 5-4.

Figure 5-4: A screenshot of a web-based schedule computation application.



These operations were quite expensive (having to run through thousands of permutations in order to find correct results). This problem was solved by breaking up clumps of schedule computation into tangible bites (updating the user interface with a percentage of completion as it went). In the end the user was presented with a usable interface that was fast, responsive, and highly usable.

It's often surprising just how useful this technique can be. You'll frequently find it being used in long-running processes, like test suites, which we'll be discussing at the end of this chapter. Most importantly though this technique shows us just how easy it is to work around the restrictions of the browser environment with the features of JavaScript language, while still providing a useful experience to the user.

Central Timer Control

A problem that arises, in the use of timers, is in just how to manage them when dealing with a large number of them. This is especially critical when dealing with animations as you'll be attempting to manipulate a large number of properties simultaneously and you'll need a way to manage that.

Having a central control for your timers gives you a lot of power and flexibility, namely:

- You only have to have one timer running per page at a time.
- You can pause and resume the timers at your will.
- The process for removing callback functions is trivialized.

It's also important to realize that increasing the number of simultaneous timers is likely to increase the likelihood of a garbage collection occurring in the browser. Roughly speaking this is when the browser goes through and tries to tie up any loose ends (removing unused variables, objects, etc.). Timers are particularly problematic since they are generally managed outside of the flow of the normal JavaScript

engine (through other threads). While some browsers are more capable of handling this situation others cause long garbage collection cycles to occur. You might have noticed this when you see a nice, smooth, animation in one browser - view it in another and see it stop-and-start its way to completion. Reducing the number of simultaneous timers being used will drastically help with this (and is the reason why all modern animation engines include a construct like the central timer control).

Let's take a look at an example where we have multiple functions animating separate properties but being managed by a single timer function in Listing 5-4.

Listing 5-4: Using a central timer control to manage multiple animations.

```
<div id="box" style="position:absolute;">Hello!</div>
<script>
var timers = {
  timerID: 0,
  timers: [],
  start: function(){
    if ( this.timerID )
      return;

    (function(){
      for ( var i = 0; i < timers.timers.length; i++ )
        if ( timers.timers[i]() === false ) {
          timers.timers.splice(i, 1);
          i--;
        }
      timers.timerID = setTimeout( arguments.callee, 0 );
    })();
  },
  stop: function(){
    clearTimeout( this.timerID );
    this.timerID = 0;
  },
  add: function(fn){
    this.timers.push( fn );
    this.start();
  }
};

var box = document.getElementById("box"), left = 0, top = 20;

timers.add(function(){
  box.style.left = left + "px";
  if ( ++left > 50 )
    return false;
});

timers.add(function(){
  box.style.top = top + "px";
  top += 2;
  if ( top > 120 )
    return false;
});
</script>
```

We've created a central control structure in Listing 5-4 that we can add new timer callback functions to and stop/start the execution of them. Additionally, the callback functions have the ability to remove themselves at any time by simply returning 'false' (which is much easier to do than the typical `clearTimeout` pattern). Let's step through the code to see how it works.

To start, all of the callback functions are stored in a central array (`timers.timers`) alongside the ID of the current timer (`timers.timerID`). The real meat comes in with in the `start()` method. Here we need to verify that there isn't already a timer running, and if that's the case, start our central timer.

This timer consists of a loop which moves through all of our callback functions, executing them in order. It also checks to see what the return value of the callback is - if it's false then the function is removed from being executed. This proves to be a much simpler manner of managing timers than

One thing that's important to note: Organizing timers in this manner ensures that the callback functions will always execute in the order in which they are added. That is not always guaranteed with normal timers (the browser could choose to execute one before another).

This manner of timer organization is critical for large applications or really any form of JavaScript animations. Having a solution in place will certainly help in any form of future application development and especially when we discuss how to create [[Animations|Cross-Browser Animations]].

Asynchronous Testing

Another situation in which a centralized timer control comes in useful is in the case where you wish to do asynchronous testing. The issue here is that when we need to perform testing on actions that may not complete immediately (such as something within a timer or an XMLHttpRequest) we need to break our test suite out such that it works completely asynchronously.

For example in a normal test suite we could easily just run the tests as we come to them however once we introduce the desire to do asynchronous testing we need to break all of those tests out and handle them separately. Listing 5-5 has some code that we can use to achieve our desired result.

Listing 5-5: A simple asynchronous test suite.

```
(function(){
  var queue = [], paused = false;

  this.test = function(fn){
    queue.push( fn );
    runTest();
  };

  this.pause = function(){
    paused = true;
  };

  this.resume = function(){
    paused = false;
    setTimeout(runTest, 1);
  };

  function runTest(){
    if ( !paused && queue.length ) {
     queue.shift()();
```

```
     if ( !paused ) {
        resume();
      }
    }
  }
})();

test(function(){
  pause();
  setTimeout(function(){
    assert( true, "First test completed" );
    resume();
  }, 100);
});

test(function(){
  pause();
  setTimeout(function(){
    assert( true, "Second test completed" );
    resume();
  }, 200);
});
```

The important aspect in Listing 5-5 is that each function passed to `test()` will contain, at most, one asynchronous test. It's asynchronicity is defined by the use of the `pause()` and `resume()` functions, to be called before and after the asynchronous event. Really, the above code is nothing more than a means of keeping asynchronous behavior-containing functions executing in a specific order (it doesn't, exclusively, have to be used for test cases, but that's where it's especially useful).

Let's look at the code necessary to make this behavior possible. The bulk of the functionality is contained within the `resume()` and `runTest()` functions. It behaves very similarly to our `start()` method in the previous example but handles a queue of data instead. Its sole purpose is to dequeue a function and execute it, if there is one waiting, otherwise completely stop the interval from running. The important aspect here is that since the queue-handling code is completely asynchronous (being contained within an interval) it's guaranteed to attempt execution after we've already called our `pause()` function.

This brief piece of code enforces our suite to behave in a purely asynchronous manner while still maintaining the order of test execution (which can be very critical in some test suites, if their results are destructive, affecting other tests). Thankfully we can see that it doesn't require very much overhead at all to add a reliable asynchronous testing to an existing test suite, with the use of the most-effective timers.

This problem id discussed more in the chapter on Testing and Debugging.

Summary

Learning about how JavaScript timers function has been illuminating: These seemingly simple features are actually quite complex in their implementations. Taking all their intricacies into account, however, gives us great insight into how we can best exploit them for our gain. It's become apparent that where timers end up being especially useful is in complex applications (computationally-expensive code, animations, asynchronous test suites). But due to their ease of use (especially with the addition of closures) they tend to make even the most complex situations easy to grasp.

Chapter 7. Regular Expressions

Covered in this chapter:

- Compiling regular expressions
- Capturing Within regular expressions
- Frequent problems and how to resolve them

Regular expressions are a necessity of modern development. They trivialize the process of tearing apart strings, looking for information. Everywhere you look, in JavaScript development, the use of regular expressions is prevalent:

- Manipulating strings of HTML nodes
- Locating partial selectors within a CSS selector expression
- Determining if an element has a specific class name
- Extracting the opacity from Internet Explorer's filter property

While this chapter won't be covering the actual syntax of regular expressions in this chapter - there are a number of excellent books on the subject, including O'Reilly's "Mastering Regular Expressions" - instead this chapter will be examining a number of the peculiar complexities that come along from using regular expressions in JavaScript.

Compiling

Regular expressions go through multiple phases of execution - understanding what happens during each of these phases can help you to write optimized JavaScript code. The two prominent phases are compilation and execution. Compilation occurs whenever the regular expression is first defined. The expression is parsed by the JavaScript engine and converted into its internal representation (whatever that may be). This phase of parsing and conversion must occur every time a regular expression is encountered (unless optimizations are done by the browser).

Frequently, browsers are smart enough to determine that if an identically-defined regular expression is compiled again-and-again to cache the compilation results for that particular expression. However, this is not necessarily the case in all browsers. For complex expressions, in particular, we can begin to get some noticeable speed improvements by pre-defining (and, thus, pre-compiling) our regular expressions for later use.

There are two ways of defining a compiled regular expression in JavaScript, as shown in Listing 6-1.

Listing 6-1: Two examples of creating a compiled regular expression.

```
var re = /test/i;
var re2 = new RegExp("test", "i");

assert( re.toString() == "/test/i",
  "Verify the contents of the expression." );
assert( re.test( "TesT" ), "Make sure the expression work." );
assert( re2.test( "TesT" ), "Make sure the expression work." );
assert( re.toString() == re2.toString(),
  "The contents of the expressions are equal." );
assert( re != re2, "But they are different objects." );
```

In the above example, both regular expressions are now in their compiled state. Note that they each have unique object representations: Every time that a regular expression is defined, and thus compiled, a new regular expression object is also created. This is unlike other primitive types (like string, number, etc.) since the result will always be unique.

Of particular importance, though, is the new use `new RegExp(...)` to define a new regular expression. This technique allows you to build and compile an expression from a string. This can be immensely useful for constructing complex expressions that will be heavily re-used.

> **Note** The second parameter to `new RegExp(...)` is the list of flags that you wish to construct the regular expression with ('i' for case insensitive, 'g' for a global match, 'ig' for both).

For example, let's say that you wanted to determine which elements, within a document, had a particular class name. Since elements are capable of having multiple class names associated with them (separated via a space) this makes for a good time to try regular expression compilation, shown in Listing 6-2.

Listing 6-2: Compiling a regular expression for future use.

```
<div class="foo ninja"></div>
<div class="ninja foo"></div>
<div></div>
<div class="foo ninja baz"></div>
<script>
function find(className, type) {
  var elems = document.getElementsByTagName(type || "*");
  var re = new RegExp("(^|\\s)" + className + "(\\s|$)");
  var results = [];

  for ( var i = 0, length = elems.length; i < length; i++ )
    if ( re.test( elems[i].className ) )
      results.push( elems[i] );

  return results;
}

assert( find("ninja", "div").length == 3,
  "Verify the right amount was found." );
assert( find("ninja").length == 3,
  "Verify the right amount was found." );
</script>
```

There are a number of things that we can learn from Listing 6-2. To start, note the use of `new RegExp(...)` to compile a regular expression based upon the input of the user. We construct this expression once, at the top of the function, in order to avoid frequent calls for re-compilation. Since the contents of the expression are dynamic (based upon the incoming `className` argument) we can get a major performance savings by handling the expression in this manner.

Another thing to notice is the use of a double-escape within `new RegExp`: `\\s`. Normally, when creating regular expressions with the `/\s/` syntax we only have to provide the backslash once. However, since we're writing these backslashes within a string, we must doubly-escape them. This is a nuisance, to be sure, but one that you must be aware of when constructing custom expressions.

The ultimate example of using regular expression compilation, within a library, can be seen in the RegGrp package of the base2 library. This particular piece of functionality allows you to merge regular expressions

together - allowing them to operate as a single expression (for speed) while maintaining all capturing and back-references (for simplicity). While there are too many details within the RegGrp package to go into, in particular, we can instead create a simplified version for our own purposes, like in Listing 6-3.

The full source to base2's RegGrp function can be found here:

- http://code.google.com/p/base2/source/browse/trunk/src/base2/RegGrp.js

Listing 6-3: Merging multiple regular expressions together.

```
function RegMerge() {
  var expr = [];
  for ( var i = 0; i < arguments.length; i++ )
    expr.push( arguments[i].toString().replace(/^\/|\/\w*$/g, "") );
  return new RegExp( "(?:" + expr.join("|") + ")" );
}

var re = RegMerge( /Ninj(a|itsu)/, /Sword/, /Katana/ );
assert( re.test( "Ninjitsu" ),
  "Verify that the new expression works." );
assert( re.test( "Katana" ),
  "Verify that the new expression works." );
```

The pre-construction and compilation of regular expressions is a powerful tool that can be re-used time-and-time again. Virtually all complex regular expression situations will require its use - and the performance benefits of handling these situations in a smart way will be greatly appreciated.

Capturing

The crux of usefulness, in regular expressions, relies upon capturing the results that you've found, so that you can do something with them. Simply matching results (or determining if a string contains a match) is an obvious first step, but determining what you matched can be used in so many situations.

For example in Listing 6-4 we extract the opacity value out of the filter value that Internet Explorer provides:

Listing 6-4: A simple function for getting the current opacity of an element.

```
<div id="opacity" style="opacity:0.5;filter:alpha(opacity=50);"></div>
<script>
function getOpacity(elem){
  var filter = elem.style.filter;
  return filter ?
    filter.indexOf("opacity=") >= 0 ?
      (parseFloat( filter.match(/opacity=([^)]*)/)[1] ) / 100) + "" :
      "" :
    elem.style.opacity;
}

window.onload = function(){
  assert( getOpacity( document.getElementById("opacity") ) == "0.5",
    "Get the current opacity of the element." );
};
</script>
```

The opacity parsing code may seem a little bit confusing, but it's not too bad once you break it down. To start with we need to determine if a filter property even exists for us to parse (if not, we try to access the opacity style property instead). If the filter is available, we need to verify that it will contain the opacity string that we're looking for (with the `indexOf` call).

At this point we can, finally, get down to the actual opacity extraction. The `match` method returns an array of values, if a match is found, or null if no match is found (we can be confident that there will be a match, since we already determined that with the `indexOf` call). The array returned by `match` includes the entire match in the first and each, subsequent, capture following. Thus, when we match the opacity value, the value is actually contained in the 1 position of the array.

Note that this return set is not always the case, from `match`, as shown in Listing 6-5.

Listing 6-5: The difference between a global and local search with match.

```
var html = "<div class='test'><b>Hello</b> <i>world!</i></div>";

var results = html.match(/<(\/?)(\w+)([^>]*?)>/);

assert( results[0] == "<div class='test'>", "The entire match." );
assert( results[1] == "", "The (missing) slash." );
assert( results[2] == "div", "The tag name." );
assert( results[3] == " class='test'", "The attributes." );

var all = html.match(/<(\/?)(\w+)([^>]*?)>/g);

assert( all[0] == "<div class='test'>", "Opening div tag." );
assert( all[1] == "<b>", "Opening b tag." );
assert( all[2] == "</b>", "Closing b tag." );
assert( all[3] == "<i>", "Opening i tag." );
assert( all[4] == "</i>", "Closing i tag." );
assert( all[5] == "</div>", "Closing div tag." );
```

While doing a global search with the `match` method is certainly useful, it is not equivalent to doing a regular, local, `match` - we lose out on all the useful capturing information that the method normally provides.

We can regain this functionality, while still maintaining a global search, by using the regular expression `exec` method. This method can be repeatedly called against a regular expression - causing it to return the next matched set of information every time it gets called. A typical pattern, for how it is used, looks like something seen in Listing 6-6.

Listing 6-6: Using the exec method to do both capturing and a global search.

```
var html = "<div class='test'><b>Hello</b> <i>world!</i></div>";
var tag = /<(\/?)(\w+)([^>]*?)>/g, match;
var num = 0;

while ( (match = tag.exec(html)) !== null ) {
  assert( match.length == 4,
  "Every match finds each tag and 3 captures." );
  num++;
}

assert( num == 6, "3 opening and 3 closing tags found." );
```

Using either `match` or `exec` (where the situation permits) we can always find the exact matches (and captures) that we're looking for. However, we find that we'll need to dig further when we need to begin referring back to the captures themselves.

References to Captures

There are two ways in which you can refer back to portions of a match that you've captured. One within the match, itself, and one within a replacement string (where applicable).

For example, let's revisit the match in Listing 6-6 (where we match an opening, or closing, HTML tag) and modify it to, also, match the inner contents of the tag, itself, in Listing 6-7.

Listing 6-7: Using back-references to match the contents of an HTML tag.

```
var html = "<b class='hello'>Hello</b> <i>world!</i>";
var tag = /<(\w+)([^>]+)>(.*?)<\/\1>/g;

var match = tag.exec(html);

assert( match[0] == "<b class='hello'>Hello</b>",
  "The entire tag, start to finish." );
assert( match[1] == "b", "The tag name." );
assert( match[2] == " class='hello'", "The tag attributes." );
assert( match[3] == "Hello", "The contents of the tag." );

match = tag.exec(html);

assert( match[0] == "<i>world!</i>",
  "The entire tag, start to finish." );
assert( match[1] == "i", "The tag name." );
assert( match[2] == "", "The tag attributes." );
assert( match[3] == "world!", "The contents of the tag." );
```

In Listing 6-7 we use `\1` in order to refer back to the first capture within the expression (which, in this case, is the name of the tag). Using this information we can match the appropriate closing tag - referring back to the right one (assuming that there aren't any tags of the same name within the current tag, of course). These back-reference codes continue up, referring back to each, individual, capture.

Additionally, there's a way to get capture references within the replace string of a `replace` method. Instead of using the back-reference codes, like in the previous example, we use the syntax of `$1`, `$2`, `$3`, up through each capture number, show in Listing 6-8.

Listing 6-8: Using a capture reference, within a replace.

```
assert( "fontFamily".replace( /([A-Z])/g, "-$1" ).toLowerCase() ==
  "font-family", "Convert the camelCase into dashed notation." );
```

Having references to regular expression captures helps to make a lot of difficult code quite easy. The expressive nature that it provides ends up allowing for some terse statements that would, otherwise, be rather obtuse and convoluted.

Non-capturing Groups

In addition to the ability to capture portions of a regular expression (using the parentheses syntax) you can also specify portions to group - but not capture. Typically this is done with the modified syntax: `(?:...)` (where ... is what you're attempting to match, but not capture).

Probably the most compelling use case for this method is the ability to optionally match entire words (multiple-character strings) rather than being restricted to just single character optional matches, an example of which is shown in Listing 6-9.

Listing 6-9: Matching multiple words using a non-capturing group

```
var re = /((?:ninja-)+)sword/;
var ninjas = "ninja-ninja-sword".match(re);

assert( ninjas[1] == "ninja-ninja-",
  "Match both words, without extra capture." );
```

Wherever possible, in your regular expressions, you should strive to use non-capturing groups in place of capturing. The expression engine has to do much less work in remembering and returning your captures. If you don't actually need to results, then there's no need to ask for them!

Replacing with Functions

Perhaps the most powerful feature presented by JavaScript regular expression support is the ability to provide a function as the value to to replace with, in a `replace` method call.

For example, in Listing 6-10 we use the function to provide a dynamic replacement value to adjust the case of a character.

Listing 6-10: Converting a string to camel case with a function replace regular expression.

```
assert( "font-family".replace( /-(\w)/g, function(all, letter){
    return letter.toUpperCase();
  }) == "fontFamily", "CamelCase a hyphenated string." );
```

The second argument, the function, receives a number of arguments which correspond to the results from a similar `match` method call. The first argument is always the entire expression match and the remaining arguments are represented by the captured characters.

The replace-function technique can even be extended beyond doing actual replacements and be used as a means of string traversal (as an alternative to doing the typical `exec`-in-a-while-loop technique).

For example, if you were looking to convert a query string like `"foo=1&foo=2&blah=a&blah=b&foo=3"` into one that looks like this: `"foo=1,2,3&blah=a,b"` a solution using regular expressions and replace could result in some, especially, terse code as shown in Listing 6-11.

Listing 6-11: A technique for compressing a query string.

```
function compress(data){
  var q = {}, ret = [];

  data.replace(/([^=&]+)=([^&]*)/g, function(m, key, value){
    q[key] = (q[key] ? q[key] + "," : "") + value;
    return "";
  });

  for ( var key in q )
    ret.push( key + "=" + q[key] );

  return ret.join("&");
```

```
}

assert( compress("foo=1&foo=2&blah=a&blah=b&foo=3") ==
 "foo=1,2,3&blah=a,b", "Verify the compression." );
```

The interesting aspect of Listing 6-11 is in using the string replace function as a means of traversing a string for values, rather than as an actual search-and-replace mechanism. The trick is two-fold: Passing in a function as the replace value argument to the .replace() method and, instead of returning a value, simply utilizing it as a means of searching.

Let's examine this piece of code:

```
data.replace(/([^=&]+)=([^&]*)/g, function(m, key, value){});
```

The regular expression, itself, captures two things: A key in the query string and its associated value. This match is performed globally, locating all the key-value pairs within the query string.

The second argument to the replace method is a function. It's not uncommon to utilize this function-as-an-argument technique when attempting to replace matches with complex values (that is values that are dependent upon their associated matches). The return value of the function is injected back into the string as its replacement. In this example we return an empty value from the function therefore we end up clearing out the string as we go.

We can see this behavior in Listing 6-12.

Listing 6-12: Replacing a value with a empty string, in a function.

```
assert( "a b c".replace(/a/, function(){ return ""; }) == " b c",
  "Returning an empty result removes a match." );
```

Now that we've collected all of our key values pairs (to be re-serialized back into a query string) the final step isn't really a step at all: We simply don't save the search-and-replaced data query string (which, most likely, looks something like `"&&"`).

In this manner we can use a string's replace method as our very-own string searching mechanism. The result is, not only, fast but also simple and effective.

Using a function as a replace, or even a search, mechanism should not be underestimated. The level of power that it provides for the amount of code is quite favorable.

Common Problems

A few idioms tend to occur again, and again, in JavaScript - but their solutions aren't always obvious. A couple of them are outlined here, for your convenience.

Trimming a String

Removing extra whitespace from a string (generally from the start and end) is used all throughout libraries - especially so within implementations of selector engines.

The most-commonly seen solution looks something like the code in Listing 6-13.

Listing 6-13: A common solution to stripping whitespace from a string.

```
function trim(str){
  return str.replace(/^\s+|\s+$/g, "");
```

```
}

assert( trim(" #id div.class ") == "#id div.class",
  "Trimming the extra whitespace from a selector string." );
```

Steven Levithan has done a lot of research into this subject, producing a number of alternative solutions, which he details:

- http://blog.stevenlevithan.com/archives/faster-trim-javascript

It's important to note, however, that in his test cases he works against an incredibly-large document (certainly the fringe case, in most applications).

Of those solutions two are particularly interesting. The first is done using regular expressions, but with no `\s+` and no `|` or operator, shown in Listing 6-14.

Listing 6-14: A trim method breaking down into two expressions.

```
function trim(str){
  return str.replace(/^\s\s*/, '').replace(/\s\s*$/, '');
}
```

The second technique completely discards any attempt at stripping whitespace from the end of the string using a regular expression and does it manually, as seen in Listing 6-15.

Listing 6-15: A trim method which slices at the rear of the string.

```
function trim(str) {
  var str = str.replace(/^\s\s*/, ''),
    ws = /\s/, i = str.length;
  while (ws.test(str.charAt(--i)));
  return str.slice(0, i + 1);
}
```

Looking at the final breakdown in performance the difference becomes quite noticeable - and easy to see how that even when a particular method of trimming strings is particular scalable (as is seen in the third trim method) and in Table 6-1.

Table 6-1: All time in ms, for 1000 iterations.

Table 7.1.

	Selector Trim	Document Trim
Listing 6-13	8.7	2075.8
Listing 6-14	8.5	3706.7
Listing 6-15	13.8	169.4

Ultimately, it depends on the situation in which you're going to find the trim method to be used. Most libraries use the first solution (and use it primarily on small strings) so that seems to be a safe assumption.

Matching Endlines

When performing a search it's frequently desired that the `.` (which normally matches any character, except for endlines) would also include endline characters. Other regular expression implementations, in other languages, frequently include an extra flag for making this possible. In JavaScript, there are two ways, as shown in Listing 6-16.

Listing 6-16: Matching all characters, including endlines.

```
var html = "<b>Hello</b>\n<i>world!</i>";
assert( /.*/.exec(html)[0] === "<b>Hello</b>",
 "A normal capture doesn't handle endlines." );
assert( /[\S\s]*/.exec(html)[0] === "<b>Hello</b>\n<i>world!</i>",
  "Matching everything with a character set." );
assert( /(?:.|\s)*/.exec(html)[0] === "<b>Hello</b>\n<i>world!</i>",
  "Using a non-capturing group to match everything." );
```

Obviously, due to its simplicity (and implicit speed benefits), the solution provided by `[\S\s]` is the optimal one. It works by simultaneously matching all non-whitespaces characters (`\S`) and whitespace characters - endlines included (`\s`).

Unicode

Frequently, for an expression, you'll want to match alphanumeric characters (such as for an ID selector in a CSS selector engine implementation) for later use. However, assuming that the alpha characters will only be in English is a little short-sited. Expanding the set to include unicode characters ends up becoming quite useful - explicitly supporting multiple languages not covered by the traditional alphanumeric character set, as seen in Listing 6-17.

Listing 6-17: Matching Unicode characters in a CSS selector.

```
var str = "\u0130\u0131\u0132";
assert( ("#" + str).match(new RegExp("#([\\w\u0128-\uFFFF_-]+)")),
  "Verify that our RegExp matches a Unicode selector." );
```

There's a specific trick that we have to use in order to get the Unicode range of characters to work, however. Specifically, we have to create the regular expression dynamically using `new RegExp(...)` and populate it with a singly-escaped version of each of the unicode characters that we want to represent the range. Starting at 128 gives us some high ascii characters and all Unicode characters. We need to create the expression in this manner because doing it with a traditional `/.../` expression creation it ends up causing older versions of Safari to crash - which is undesirable in any situation.

Escaped Characters

Another, common, desire for matching in an implementation of a CSS selector engine is to support escaped characters. This allows the user to specify complex names that don't conform to typical naming conventions (either specified by the HTML or CSS specification) an example of which is in Listing 6-18.

Listing 6-18: Matching escaped characters in a CSS selector.

```
assert( "#form\\:update".match(/#((?:\w|\\.)+)/)[1] == "form:update",
  "Matching an escaped expression." );
```

These particular expression works by using a non-capturing group to allow for the match of either an alphanumeric character or a sequence of a backslash followed by any character (which is acceptable - since we're assuming that the user will be escaping their desired characters).

Summary

Regular expressions permeate modern JavaScript development. Virtually every aspect of development depends on their use in some way. With a good understanding of the advanced concepts that have been

covered in this chapter, any developer should feel comfortable in tackling a challenging piece of regular expression-using JavaScript code. The techniques including compiling regular expressions, capturing values, and replacing with functions - not to mention the many minor tricky points, such as dealing with Unicode. Together these these techniques make for a formidable development armada.

Chapter 8. With Statements

Covered in this chapter:

- How `with(){}` statements work
- Code simplification
- Tricky gotchas
- Templating

`with(){}` statements are a powerful, and frequently misunderstood, feature of JavaScript. They allow you to put all the properties of an object within the current scope - as if they were normal JavaScript variables (allowing you to reference them and assign to them - while having their values be maintained in the original object). Figuring out how to use them correctly can be tricky, but doing so gives you a huge amount of power in your application code.

To start, let's take a look at the basics of how they work, as shown in Listing 7-1.

Listing 7-1: Exposing the properties of the katana object using `with(){}`.

```
var use = "other";
var katana = {
  isSharp: true,
  use: function(){
    this.isSharp = !this.isSharp;
  }
};

with ( katana ) {
  assert( true, "You can still call outside methods." );

  isSharp = false;
  use();

  assert( use !== "other",
  "Use is a function, from the katana object." );
  assert( this !== katana,
  "this isn't changed - it keeps its original value" );
}

assert( typeof isSharp === "undefined",
  "Outside the with, the properties don't exist." );
assert( katana.isSharp,
  "Verify that the method was used correctly in the with." );
```

In the above listing we can see how the properties of the katana object are selectively introduced within the scope of the `with(){}` statement. You can use them directly as if they were first-class variables or methods. Note that within the scope of the statement the properties introduced by `with(){}` take absolute precedence over the other variables, of the same name, that might exist (as we can see with the `use` property and variable). You can also see that the `this` context is preserved while you are within the statement - as is true with virtually all variables keywords - only the ones specified by the original object are introduced. Note that the `with(){}` statement is able to take any JavaScript object, it's not restricted in any sense.

Let's take a look at a different example of assignment within a `with(){}` statement:

Listing 7-2: Attempting to add a new property to a `with(){}`'d object.

```
var katana = {
  isSharp: true,
  use: function(){
    this.isSharp = !!this.isSharp;
  }
};

with ( katana ) {
  isSharp = false;
  cut = function(){
    isSharp = false;
  };
}

assert( !katana.cut, "The new cut property wasn't introduced here." );
assert( cut, "It was made as a global variable instead." );
```

One of the first things that most people try, when using `with(){}`, is to assign a new property to the original object. However, this does not work. It's only possible to use and assign existing properties within the `with(){}` statement - anything else (assigning to new variables) is handled within the native scope and rules of JavaScript (as if there was no `with(){}` statement there at all).

In the above listing we can see that the cut method - which we presumed would be assigned to the katana object - is instead introduced as a global variable.

This should be implied with the results from above but misspelling a property name can lead to strange results (namely that a new global variable will be introduced rather than modifying the existing property on the `with(){}`'d object). Of course, this is, effectively, the same thing that occurs with normal variables, so you'll just need to carefully monitor your code, as always.

There's one major caveat when using `with(){}`, though: It slows down the execution performance of any JavaScript that it encompasses - and not just objects that it interacts with. Let's look at three examples, in Listing 7-3.

Listing 7-3: Examples of interacting with a with statement.

```
var obj = { foo: "bar" }, value;

// "No With"
for ( var i = 0; i < 1000; i++ ) {
  value = obj.foo;
}

// "Using With"
with(obj){
  for ( var i = 0; i < 1000; i++ ){
    value = foo;
  }
}

// "Using With, No Access"
```

```
with(obj){
  for ( var i = 0; i < 1000; i++ ){
    value = "no test";
  }
}
```

The results of running the statements in Listing 7-3 result in some rather dramatic performance differences, as seen in Table 7-1.

Table 7-1: All time in ms, for 1000 iterations, in a copy of Firefox 3.

Table 8.1.

	Average	Min	Max	Deviation
No With	0.14	0	1	0.35
Using With	1.58	1	2	0.50
Using With, No Access	1.38	1	2	0.49

Effectively, any variables accesses must now run through an additional with-scope check which provides an extra level of overhead (even if the result of the with statement isn't being used, as in the third case). If you are uncomfortable with this extra overhead, or if you aren't interested in any of the conveniences, then `with(){}` statements probably aren't for you.

Convenience

Perhaps the most common use case for using `with(){}` is as a simple means of not having to duplicate variable usage or property access. JavaScript libraries frequently use this as a means of simplification to, otherwise, complex statements.

Here are a few examples from a couple major libraries, starting with Prototype in Listing 7-4.

Listing 7-4: A case of using `with(){}` in the Prototype JavaScript library.

```
Object.extend(String.prototype.escapeHTML, {
  div:  document.createElement('div'),
  text: document.createTextNode('')
});

with (String.prototype.escapeHTML) div.appendChild(text);
```

Prototype uses `with(){}`, in this case, as a simple means of not having to call `String.prototype.escapeHTML` in front of both `div` and `text`. It's a simple addition and one that saves three extra object property calls.

The example in Listing 7-5 is from the base2 JavaScript library.

Listing 7-5: A case of using `with(){}` in the base2 JavaScript library.

```
with (document.body.style) {
  backgroundRepeat = "no-repeat";
  backgroundImage =
    "url(http://ie7-js.googlecode.com/svn/trunk/lib/blank.gif)";
```

```
  backgroundAttachment = "fixed";
}
```

Listing 7-5 uses `with(){}` as a simple means of not having to repeat `document.body.style` again, and again - allowing for some super-simple modification of a DOM element's style object. Another example from base2 is in Listing 7-6.

Listing 7-6: A case of using `with(){}` in the base2 JavaScript library.

```
var Rect = Base.extend({
  constructor: function(left, top, width, height) {
    this.left = left;
    this.top = top;
    this.width = width;
    this.height = height;
    this.right = left + width;
    this.bottom = top + height;
  },

  contains: function(x, y) {
    with (this)
      return x >= left && x <= right && y >= top && y <= bottom;
  },

  toString: function() {
    with (this) return [left, top, width, height].join(",");
  }
});
```

This second case within base2 uses `with(){}` as a means of simply accessing instance properties. Normally this code would be much longer but the terseness that `with(){}` is able to provide, in this case, adds some much-needed clarity.

The final example, in Listing 7-7, is from the Firebug developer extension for Firefox.

Listing 7-7: An example of `with(){}` use within the Firebug Firefox extension.

```
const evalScriptPre = "with(__scope__.vars){ with(__scope__.api){" +
  " with(__scope__.userVars){ with(window){";
const evalScriptPost = "}}}}";
```

The lines in Listing 7-7, from Firebug, are especially complex - quite possibly the most complex uses of `with(){}` in a publicly-accessible piece of code. These statements are being used within the debugger portion of the extension, allowing the user to access local variables, the firebug API, and the global object all within the JavaScript console. Operations like this are generally outside the scope of most applications, but it helps to show the power of `with(){}` and in how it can make incredibly complex pieces of code possible, with JavaScript.

One interesting takeaway from the Firebug example, especially, is the dual-use of `with(){}`, bringing precedence to the window object over other, introduced, objects. Performing an action like the following:

```
with ( obj ) { with ( window ) { ... } }
```

Will allow you to have the `obj` object's properties be introduced by `with(){}`, while having the global variables (available by `window`) take precedence, exclusively.

Importing Namespaced Code

As shown previously one of the most common uses for the `with(){}` statement is in simplifying existing statements that have excessive object property use. You can see this most commonly in namespaced code (objects within objects, providing an organized structure to code). The side effect of this technique is that it becomes quite tedious to re-type the object namespaces again-and-again.

Note Listing 7-8, both performing the same operation using the Yahoo UI JavaScript library, but also gaining extra simplicity and clarity by using `with(){}`.

Listing 7-8: Binding click events to two elements using Yahoo UI - done both normally and using `with(){}`.

```
YAHOO.util.Event.on(
  [YAHOO.util.Dom.get('item'), YAHOO.util.Dom.get('otheritem')],
  'click', function(){
    YAHOO.util.Dom.setStyle(this,'color','#c00');
  }
);

with ( YAHOO.util.Dom ) {
  YAHOO.util.Event.on([get('item'), get('otheritem')], 'click',
   function(){ setStyle(this,'color','#c00'); });
}
```

The addition of this single `with(){}` statement allows for a considerable increase in code simplicity. The resulting clarity benefits are debatable (especially when , although the reduction in code size

Clarified Object-Oriented Code

When writing object-oriented JavaScript a couple issues come into play - especially when creating objects that include private instance data. `with(){}` statements are able to change the way this code is written in a particularly interesting way. Observe the constructor code in Listing 7-9.

Listing 7-9: Defining an object which has both private and public data, equally exposed using `with(){}`.

```
function Ninja(){with(this){
  // Private Information
  var cloaked = false;

  // Public property
  this.swings = 0;

  // Private Method
  function addSwing(){
    return ++swings;
  }

  // Public Methods
  this.swingSword = function(){
    cloak( false );
    return addSwing();
  };
```

```
  this.cloak = function(value){
    return value != null ?
      cloaked = value :
      cloaked;
  };
}}

var ninja = new Ninja();

assert( !ninja.cloak(), "We start out uncloaked." );
assert( ninja.swings == 0, "And with no sword swings." );

assert( ninja.cloak( true ),
  "Verify that the cloaking worked correctly." );
assert( ninja.swingSword() == 1, "Swing the sword, once." );
assert( !ninja.cloak(),
  "The ninja became uncloaked with the sword swing." );
```

A couple points to consider about the style of the code in Listing 7-9:

- There is an explicit difference between how private and public instance data are defined (as is the case with most object-oriented code).
- There is absolutely no difference in how private data is accessed, from public data. All data is accessed in the same way: by its name. This is made possible by the use of `with(this){}` at the top of the code - forcing all public properties to become local variables.
- Methods are defined similarly to variables (publicly and privately). Public methods must include the `this.` prefix when assigning - but are accessed in the same ways as private methods.

The ability to access properties and methods using a unified naming convention (as opposed to some being through direct variable access and others through the instance object) is quite valuable.

Testing

When testing pieces of functionality in a test suite there's a couple things that you end up having to watch out for. The primary of which is the synchronization between the assertion methods and the test case currently being run. Typically this isn't a problem but it becomes especially troublesome when you begin dealing with asynchronous tests. A common solution to counteract this is to create a central tracking object for each test run. The test runner used by the Prototype and Scriptaculous libraries follows this model - providing this central object as the context to each test run. The object, itself, contains all the needed assertion methods (easily collecting the results back to the central location). You can see an example of this in Listing 7-10.

Listing 7-10: An example of a test from the Scriptaculous test suite.

```
new Test.Unit.Runner({
  testSliderBasics: function(){with(this){
    var slider = new Control.Slider('handle1', 'track1');
    assertInstanceOf(Control.Slider, slider);
    assertEqual('horizontal', slider.axis);
    assertEqual(false, slider.disabled);
    assertEqual(0, slider.value);
    slider.dispose();
```

```
  }},
  // ...
});
```

You'll note the use of `with(this){}` in the above test run. The instance variable contains all the assertion methods (`assertInstanceOf`, `assertEqual`, etc.). The above method calls could've, also, been written explicitly as `this.assertEqual` - but by using `with(this){}` to introduce the methods that we wish to use we can get an extra level of simplicity in our code.

Templating

The last - and likely most compelling - example of using `with(){}` exists within a simplified templating system. A couple goals for a templating system are usually as follows:

- There should be a way to both run embedded code and print out data.
- There should be a means of caching the compiled templates
- and, perhaps, most importantly: It should be simple to easily access mapped data.

The last point is where `with(){}` becomes especially useful. To begin to understand how this is possible, let's look at the templating code in Listing 7-11.

Listing 7-11: A simple templating solution using `with(){}` for simplified templates.

```
(function(){
  var cache = {};

  this.tmpl = function tmpl(str, data){
    // Figure out if we're getting a template, or if we need to
    // load the template - and be sure to cache the result.
    var fn = !/\W/.test(str) ?
      cache[str] = cache[str] ||
        tmpl(document.getElementById(str).innerHTML) :

      // Generate a reusable function that will serve as a template
      // generator (and which will be cached).
      new Function("obj",
        "var p=[],print=function(){p.push.apply(p,arguments);};" +

        // Introduce the data as local variables using with(){}
        "with(obj){p.push('" +

        // Convert the template into pure JavaScript
        str
          .replace(/[\r\t\n]/g, " ")
          .split("<%").join("\t")
          .replace(/((^|%>)[^\t]*)'/g, "$1\r")
          .replace(/\t=(.*?)%>/g, "',$1,'")
          .split("\t").join("');")
          .split("%>").join("p.push('")
          .split("\r").join("\\'")
      + "');}return p.join('');");
```

```
    // Provide some basic currying to the user
    return data ? fn( data ) : fn;
  };
})();

assert( tmpl("Hello, <%= name %>!", {name: "world"}) ==
  "Hello, world!", "Do simple variable inclusion." );

var hello =  tmpl("Hello, <%= name %>!");
assert( hello({name: "world"}) == "Hello, world!",
  "Use a pre-compiled template." );
```

This templating system provides a quick-and dirty solution to simple variable substitution. By giving the user the ability to pass in an object (containing the names and values of template variables that they want to populate) in conjunction with an easy means of accessing the variables, the result is a simple, reusable, system. This is made possible largely in part due to the existence of the `with(){}` statement. By introducing the data object all of its properties immediately become first-class accessible within the template - which makes for some simple, re-usable, template writing.

The templating system works by converting the provided template strings into an array of values - eventually concatenating them together. The individual statements, like `<%= name %>` are then translated into the more palatable `, name, ` - folding them inline into the array construction process. The result is a template construction system that is blindingly fast and efficient.

Additionally, all of these templates are generated dynamically (out of necessity, since inline code is allowed to be executed). In order to facilitate re-use of the generated templates we can place all of the constructed template code inside a `new Function(...)` - which will give us a template function that we can actively plug our needed data into.

A good question should now be: How do we use this templating system in a practical situation, especially within an HTML document? We can see this in Listing 7-12.

Listing 7-12: Using our templating system to generate HTML.

```
<html>
<head>
  <script type="text/tmpl" id="colors">
    <p>Here's a list of <%= items.length %> items:</p>
    <ul>
      <% for (var i=0; i < items.length; i++) { %>
        <li style='color:<%= colors[i % colors.length] %>'>
          <%= items[i] %></li>
      <% } %>
    </ul>
    and here's another...
  </script>
  <script type="text/tmpl" id="colors2">
    <p>Here's a list of <%= items.length %> items:</p>
    <ul>
      <% for (var i=0; i < items.length; i++) {
        print("<li style='color:", colors[i % colors.length], "'>",
          items[i], "</li>");
      } %>
    </ul>
  </script>
```

```
  <script src="tmpl.js"></script>
  <script>
    var colorsArray = ['red', 'green', 'blue', 'orange'];

    var items = [];
    for ( var i = 0; i < 10000; i++ )
      items.push( "test" );

    function replaceContent(name) {
      document.getElementById('content').innerHTML =
        tmpl(name, {colors: colorsArray, items: items});
    }
  </script>
</head>
<body>
  <input type="button" value="Run Colors"
    onclick="replaceContent('colors')">
  <input type="button" value="Run Colors2"
    onclick="replaceContent('colors2')">
  <p id="content">Replaced Content will go here</p>
</body>
</html>
```

Listing 7-12 shows a couple examples of functionality that's possible with this templating system, specifically the ability to run inline JavaScript in the templates, a simple print function, and the ability to embed the templates inline in HTML.

To be able to run full JavaScript code inline we break out of the array construction process and defer to the user's code. For example, the final result of the loop in the above colors template would look something like this:

```
p.push('<p>Here's a list of', items.length, ' items:</p> <ul>');
for (var i=0; i < items.length; i++) {
  p.push('<li style=\'color', colors[i % colors.length], '\'>',
    items[i], '</li>');
}
p.push('</ul> and here's another...');
```

The `print()` method plays into this nicely, as well, being just a simple wrapper pointing back to `p.push()`.

Finally, the full templating system is pulled together with the use of embedded templates. There's a great loophole provided by modern browsers and search engines: <script/> tags that specify a `type=""` that they don't understand are completely ignored. This means that we can specify scripts that contain our templates, given them a type of "text/tmpl", along with a unique ID, and use our system to extract the templates again, later.

The total result of this templating system is one that is easy to use (due, in large part, to the abilities of `with(){}`), fast, and cacheable: a sure win for development.

Summary

If anything has become obvious, over the course of this chapter, is that the primary goal of `with(){}` is to make complex code simple: simplifying access to namespaced code, improving the usability of test

suites, building usable templating utilities, and even improving the use of existing JavaScript libraries. Discretion should be applied when using it, but having a good understanding of how it works can only lead to better-quality code.

Chapter 9. Code Evaluation

Covered in this chapter:

- We examine how code evaluation works.
- Different techniques for evaluating code.
- How to be use evaluation in your applications.

JavaScript includes the ability to dynamically execute pieces of code at runtime. Code evaluation is simultaneously the most advanced and most frequently misused feature of JavaScript. Understanding the situations in which it can and should be used (along with the best techniques for using it) can give you a definite advantage when creating advanced applications.

Code Evaluation

Within JavaScript there are a number of different ways to evaluate code. Each have their own advantages and disadvantages and should be chosen carefully based upon the context in which they are to be used.

eval()

The `eval()` function is the most commonly used means of evaluating code. The most consistent means of accessing it is as a global function. The eval function executes all code passed in to it, within the current context, returning the result of the last evaluated expression, as in Listing 8-1.

Listing 8-1: A basic example of the `eval` function executing and returning a result.

```
assert( eval("5 + 5") === 10,
  "The code is evaluated and result returned " );
assert( eval("var t = 5;") === undefined, "No result returned." );
assert( t === 5, "But the t variable now exists." );

(function(){
  eval("var t = 6;");
  assert( t === 6, "eval is executed within the current scope." );
})();

assert( t === 5,
  "Verify that the scope execution was done correctly." );
```

The simplest way to determine if a result will be returned from eval is to pretend that the last expression is wrapped in parentheses and still syntactically correct. For example, the code in Listing 8-2 will return values when wrapped in parentheses.

Listing 8-2: Examples of values returned from eval.

```
assert( (true) === true,
  "Variables, objects, and primitives all work." );
assert( (t = 5), === 5, "Assignment returns the assigned value." );
assert( typeof (function(){}) === "function",
  "A function is returned." );
```

It should be noted that anything that isn't a simple variable, primitive, or assignment will actually need to be wrapped in a parentheses in order for the correct value to be returned, for example in Listing 8-3.

Listing 8-3: Getting a value from an evaluated object.

```
var obj = eval("({name:'Ninja'})");
assert( obj.name === "Ninja",
  "Objects require the extra parentheses." );

var fn = eval("(function(){return 'Ninja';})");
assert( fn() === "Ninja", "As do functions." );
```

Now the last one is interesting because is technically should work that simply (and does in Firefox, Safari, and Opera) however Internet Explorer has a problem executing the function with that particular syntax. Instead we are forced to use some boolean-expression trickery to get the function to be generated correctly. For example Listing 8-4 shows a technique found in jQuery to create a function of that nature.

Listing 8-4: Dynamically creating a function with `eval()`.

```
var fn = eval("false||function(){return true;}");
assert( fn() === true,
  "Verify that the function was created correctly." );
```

Now one might wonder why we would even want to create a function in this manner in the first place. One important point is that all code executed by the eval is done so within the current scope and, thus, it has all the properties of it as well - including any enclosed variables. Thus, using the above code, we can dynamically create a function that has a closure to all the variables within the current scope and all the externally enclosed variables as well.

Of course, if we don't need that additional closure, there's a proper alternative that we can make use of.

new Function

All functions in JavaScript are an instance of `Function`. Traditionally the `function name(...) {...}` syntax is used to define a new function however we can go a step further and instantiate functions directly using `new Function`, like in Listing 8-5.

Listing 8-5: Dynamically create a function using `new Function`.

```
var add = new Function("a", "b", "return a + b;");
assert( add(3, 4) === 7, "Function is created and works." );
```

The last argument to `new Function` is always the code that will be contained within the function. Any arguments that go before it will represent the name of the arguments passed in to the function itself.

It's important to remember that no closures are created when functions are created in this manner. This can be a good thing if you don't want to incur any of the overhead associated with possible variable lookups.

It should be noted that code can also be evaluated is asynchronously, using timers. Normally one would pass in a function reference to a timer (which is the recommended behavior) however `setTimeout` and `setInterval` also accept strings which can be evaluated. It's very rare that you would need to use this behavior (it's roughly equivalent to using `new Function`) and it's essentially unused at this point.

Evaluate in the global scope

There's one problem that is frequently encountered, when working with eval, and it's the desire to evaluate a piece of code within the global scope. For example having the following code execute and become global variables, like in Listing 8-6.

Listing 8-6: Attempting to evaluate code into a global variable.

```
(function(){
  eval("var test = 5;");
})();

assert( typeof test === "undefined",
  "Variable is confined to temporary enclosures." );
```

However there is one trick that we can use in modern browsers to achieve an acceptable result: Injecting a dynamic `<script/>` tag into the document with the script contents that we need to execute.

Andrea Giammarchi developed the specific technique for making this work properly cross-platform. An adaptation of his work can be found in Listing 8-7.

- Original Example: http://webreflection.blogspot.com/2007/08/global-scope-evaluation-and-dom.html

Listing 8-7: An example implementation of evaluating code in the global scope.

```
<script>
function globalEval( data ) {
  data = data.replace(/^\s*|\s*$/g, "");

  if ( data ) {
    var head = document.getElementsByTagName("head")[0] ||
        document.documentElement,
      script = document.createElement("script");

    script.type = "text/javascript";
    script.text = data;

    head.appendChild( script );
    head.removeChild( script );
  }
}

window.onload = function(){
  (function(){
    globalEval("var test = 5;");
  })();

  assert( test === 5, "The code is evaluated globally." );
};
</script>
```

The most common place for using this code is in dynamically executing code returned from a server. It's, almost always, a requirement that code of that nature be executed within the global scope (making the above a necessity).

Safe Code Evaluation

One question that frequently arrives is related to the safe execution of JavaScript code, namely: It is possible to safely execute untrusted JavaScript on your page without compromising the integrity of your site. Generally the answer to that question is: No. There are simply too many ways to skirt around any barriers put forth that can result in code getting access to information that it's not supposed to.

There is one project, though, produced by Google called Caja. It's an attempt to create a translator for JavaScript that converts JavaScript into a safer form (and immune to malicious attacks).

- Google Caja: http://code.google.com/p/google-caja/

For example, look at Listing 8-8 and the (relatively simple) JavaScript code and the final Caja code that it gets converted into.

Listing 8-8: The result of a Caja conversion process.

```
// Original Code:
var test = true;
(function(){ var foo = 5; })();
Function.prototype.toString = function(){};

// Cajoled Code:
{
___.loadModule(function (___, IMPORTS___) {
{
  var Function = ___.readImport(IMPORTS___, 'Function');
  var x0___;
  var x1___;
  var x2___;
  var test = true;
  ___.asSimpleFunc(___.primFreeze(___.simpleFunc(function () {
    var foo = 5;
  })))();
  IMPORTS___[ 'yield' ] ((x0___ = (x2___ = Function,
    x2___.prototype_canRead___?
  x2___.prototype: ___.readPub(x2___, 'prototype')),
  x1___ = ___.primFreeze(___.simpleFunc(function () {}))),
  x0___.toString_canSet___? (x0___.toString = x1___):
  ___.setPub(x0___, 'toString', x1___)));
}
});
}
```

Note the extensive use of built-in methods and properties to verify the integrity of the data (most of which is verified at runtime, however a number of possible loopholes are discovered during the conversion process). For example the following snippet throws an immediate exception during the conversion process:

```
(function(){ return this; })();
// FATAL_ERROR :4+2 - 28: Implicit xo4a only allowed in warts mode
```

The desire for secure JavaScript stems from wanting to create mashups and safe AD embedding without worrying about your user's security becoming compromised. We're certainly going to see a lot of work in this realm and Google Caja is leading the way.

Function Decompilation

Most JavaScript implementations also provide access to decompiling already-compiled JavaScript code. Specifically this is done with the .toString() method of functions. A sample result will look something like the following in Listing 8-9.

Listing 8-9: Decompiling a function into a string.

```
function test(a){
  return a + a;
}

assert( test.toString() ===
  "function test(a) {\n    return a + a;\n}",
  "A decompiled function." );
```

This act of decompilation has a number of potential uses (especially in the area of macros and code rewriting) however one of the most popular is one presented in the Prototype JavaScript library. In there they decompile a function in order to read out its arguments (resulting in an array of named arguments). This is frequently used to introspect into functions to determine what sort of values they are expecting to receive.

Below is a simple rewrite of the code in Prototype to become more generic in nature, in Listing 8-10.

Listing 8-10: A function for extracting the argument names for a function.

```
function argumentNames(fn) {
  var found = /^[\s\(]*function[^(]*\(\s*([^)]*?)\s*\)/.exec(fn);
  return found && found[1] ? found[1].split(/,\s*/) : [];
}

assert( argumentNames( argumentNames )[0] === "fn",
  "Get the first argument name." );

function temp(){}

assert( argumentNames( temp ).length === 0, "No arguments found." );
```

There is one point to take into consideration when working with functions in this manner: It's possible that a browser may not support decompilation. There aren't many that don't, but one such browser is Opera Mini.

We can do two things to counteract this: One we can write our code in a resilient manner (like was done above with the `argumentNames` function) or two we can perform a quick test to see if decompilation works in the browser, like in Listing 8-11.

Listing 8-11: Testing to see if function decompilation works as we expect it to.

```
assert( /abc(.|\n)*xyz/.test(function(abc){xyz;}),
 "Decompilation works as we except it to." );
```

We used this test in the chapter on Function Prototypes when we built our simple Class implementation in order to determine if we could analyze methods for if they were accessing a super method.

Examples

There are a number of ways in which code evaluation can be used for both interesting, and practical, purposes. Examining some examples of evaluation is used can give us a better understanding of when and where we should try to use it in our code.

JSON

Probably the most wide-spread use of eval comes in the form of converting JSON strings into their JavaScript object representations. Since JSON data is simply a subset of the JavaScript language it is perfectly capable of being evaluated.

There is one minor gotcha that we have to take into consideration, however: We need to wrap our object input in parentheses in order for it to evaluate correctly. However, the result is quite simple, as seen in Listing 8-12.

Listing 8-12: Converting a JSON string into a JavaScript object using the eval function.

```
var json = '{"name":"Ninja"}';
var object = eval("(" + json + ")");

assert( object.name === "Ninja",
  "Verify that the object evaluated and came out." );
```

Not only does the `eval` function win in simplicity but it also wins in performance (running drastically faster than any manual JSON parser).

However there's a major caveat to using `eval()` for JSON parsing: Most of the time JSON data is coming from a remote server and blindly executing code from a remote server is rarely a good thing.

The most popular JSON script is written by Douglas Crockford (the original specifier of the JSON markup) in it he does some initial parsing of the JSON string in an attempt to prevent any malicious information from passing through. The full code can be found here:

- http://json.org/json2.js

The important pieces of the JSON script (that pre-parses the JSON string) can be found in Listing 8-13.

Listing 8-13: The json2.js pre-parsing code (enforcing the integrity of incoming JSON strings).

```
var cx = /[\u0000\u00ad\u0600-\u0604\u070f\u17b4\u17b5\
  \u200c-\u200f\u2028-\u202f\u2060-\u206f\ufeff\ufff0-\uffff]/g;

cx.lastIndex = 0;
if (cx.test(text)) {
  text = text.replace(cx, function (a) {
    return '\\u' + ('0000' +
      (+(a.charCodeAt(0))).toString(16)).slice(-4);
  });
}

if (/^[\],:{}\s]*$/.test(
  text.replace(/\\(?:["\\\/bfnrt]|u[0-9a-fA-F]{4})/g, '@')
  .replace(/"[^"\\\n\r]*"|true|false|null|-?\d+\
    (?:\.\d*)?(?:[eE][+\-]?\d+)?/g, ']')
  .replace(/(?:^|:|,)(?:\s*\[)+/g, ''))) {

  j = eval('(' + text + ')');
}
```

The pre-parsing works in a couple stages, in the words on Douglas Crockford:

- First we replace certain Unicode characters with escape sequences. JavaScript handles many characters incorrectly, either silently deleting them, or treating them as line endings.
- Next we run the text against regular expressions that look for non-JSON patterns. We are especially concerned with '()' and 'new' because they can cause invocation, and '=' because it can cause mutation. But just to be safe, we want to reject all unexpected forms.

- We then split the second stage into 4 regexp operations in order to work around crippling inefficiencies in IE's and Safari's regexp engines. First we replace the JSON backslash pairs with '@' (a non-JSON character). Second, we replace all simple value tokens with ']' characters. Third, we delete all open brackets that follow a colon or comma or that begin the text. Finally, we look to see that the remaining characters are only whitespace or ']' or ',' or ':' or '{' or '}'. If that is so, then the text is safe for eval.
- Finally we use the eval function to compile the text into a JavaScript structure. The '{' operator is subject to a syntactic ambiguity in JavaScript: it can begin a block or an object literal. We wrap the text in parens to eliminate the ambiguity.

Once all the steps have been taken to protect our incoming JSON strings `eval` then becomes the preferred means of converting JSON data into JavaScript objects.

Importing Namespace

Importing namespaced code into the current context can be a challenging problem - especially considering that there is no simple way to do it in JavaScript. Most of the time we have to perform an action similar to the following:

```
var DOM = base2.DOM;
var JSON = base2.JSON;
// etc.
```

The base2 library provides a very interesting solution to the problem of importing namespaces into the current context. Since there is no way to automate this problem, traditionally, we can make use of eval in order to make the above trivial to implement.

Whenever a new class or module is added to a package a string of executable code is constructed which can be evaluated to introduce the function into the context, like in Listing 8-14.

Listing 8-14: Examining how the base2 namespace property works.

```
base2.namespace ==
  "var Base=base2.Base;var Package=base2.Package;" +
  "var Abstract=base2.Abstract;var Module=base2.Module;" +
  "var Enumerable=base2.Enumerable;var Map=base2.Map;" +
  "var Collection=base2.Collection;var RegGrp=base2.RegGrp;" +
  "var Undefined=base2.Undefined;var Null=base2.Null;" +
  "var This=base2.This;var True=base2.True;var False=base2.False;" +
  "var assignID=base2.assignID;var detect=base2.detect;" +
  "var global=base2.global;var lang=base2.lang;" +
  "var JavaScript=base2.JavaScript;var JST=base2.JST;" +
  "var JSON=base2.JSON;var IO=base2.IO;var MiniWeb=base2.MiniWeb;" +
  "var DOM=base2.DOM;var JSB=base2.JSB;var code=base2.code;" +
  "var doc=base2.doc;"

assert( typeof DOM === "undefined", "The DOM object doesn't exist." );

eval( base2.namespace );

assert( typeof DOM === "object",
  "And now the namespace is imported." );
assert( typeof Collection === "object",
  "Verifying the namespace import." );
```

and we can see how this list is constructed in the base2 Package code:

```
this.namespace = format("var %1=%2;", this.name,
  String2.slice(this, 1, -1));
```

This is a very ingenious way of tackling this complex problem. Albeit it isn't, necessarily, done in the most graceful manner but until implementations of future versions of JavaScript exist we'll have to make do with what we have.

Compression and Obfuscation

A popular piece of JavaScript software is that of Dean Edwards' Packer. This particular script compresses JavaScript code, providing a resulting JavaScript file that is significantly smaller while still being capable of executing and self-extracting itself to run again.

- Dean Edwards' Packer: http://dean.edwards.name/packer/

The result is an encoded string which is converted into a string of JavaScript code and executed, using `eval()`. The result typically looks something like the code in Listing 8-15.

Listing 8-15: An example of code compressed using Packer.

```
eval(function(p,a,c,k,e,r){e=function(c){return(c<a?'':e(
 parseInt(c/a)))+((c=c%a)>35?String.fromCharCode(c+29):
 c.toString(36))};if(!''.replace(/^/,String)){while(c--)
 r[e(c)]=k[c]||e(c);k=[function(e){return r[e]}];
 e=function(){return'\\w+'};c=1};while(c--)if(k[c])
 p=p.replace(new RegExp('\\b'+e(c)+'\\b','g'),k[c]);
 return p}(' // ... long string ...
```

While this technique is quite interesting there are some fundamental flaws with it: Namely that the overhead of uncompressing the script every time it loads is quite costly.

When distributing a piece of JavaScript code it's traditional to think that the smallest (byte-size) code will download and load the fastest. This is not true - and is a fascinating result of this survey. Looking at the speed of loading jQuery in three forms: normal, minified (using Yahoo! Minifier - removing whitespace and other simple tricks), and packed (using Dean Edwards' Packer - massive rewriting and uncompression using eval). By order of file size, packed is the smallest, then minifed, then normal. However, the packed version has an overhead: It must be uncompressed, on the client-side, using a JavaScript decompression algorithm. This unpacking has a tangible cost in load time. This means, in the end, that using a minifed version of the code is much faster than the packed one - even though its file size is quite larger.

It breaks down to a simple formula:

```
Total_Speed = Time_to_Download + Time_to_Evaluate
```

You can see the result of a study that was done on the jQuery library, analyzing thousands of file downloads:

- http://ejohn.org/blog/library-loading-speed/

The results of which can be seen in Table 8-1.

Table 8-1: A comparison of transfer speeds, in various formats, of the jQuery JavaScript library.

Table 9.1.

Minified	Time Avg	# of Samples

minified	519.7214	12611
packed	591.6636	12606
normal	645.4818	12589

This isn't to say that using code from Packer is worthless (if you're shooting for performance then it may be) but it's still quite valuable as a means of code obfuscation.

If nothing else Packer can serve as a good example of using `eval()` gratuitously - where superior alternatives exist.

Dynamic Code Rewriting

Since we have the ability to decompile existing JavaScript functions using a function's .toString() method that means that we can create new functions, extracting the old function's contents, that take on new functionality.

One case where this has been done is in the unit testing library Screw.Unit.

- Screw.Unit: http://github.com/nkallen/screw-unit/tree/master

Screw.Unit takes the existing test functions and dynamically re-writes their contents to use the functions provided by the library. For example, Listing 8-16 shows what a typical Screw.Unit test looks like.

Listing 8-16: A sample Screw.Unit test.

```
describe("Matchers", function() {
  it("invokes the provided matcher on a call to expect", function() {
    expect(true).to(equal, true);
    expect(true).to_not(equal, false);
  });
});
```

Especially note the describe, it, and expect methods - all of which don't exist in the global scope. To counteract this Screw.Unit rewrites this code on the fly to wrap all the functions with multiple `with() {}` statements - injecting the function internals with the functions that it needs in order to execute, as seen in Listing 8-17.

Listing 8-17: The code used by Screw.Unit to introduce new methods into an existing function.

```
var contents = fn.toString().match(/^[^{]*{((.*\n*)*)}/m)[1];
var fn = new Function("matchers", "specifications",
  "with (specifications) { with (matchers) { " + contents + " } }"
);

fn.call(this, Screw.Matchers, Screw.Specifications);
```

This is a case of using code evaluation to provide a simpler user experience to the end-user without having to negative actions (such as introducing many variables into the global scope).

Aspect-Oriented Script Tags

We've previously discussed using script tags that have invalid type attributes as a means of including new pieces of data in the page that you don't want the browser to touch. We can take that concept one step further and use it to enhance existing JavaScript.

Let's say that we created a new script type called "onload" - it contained normal JavaScript code but was only executed whenever that page was already loaded (as opposed to being normally executed inline). The script in Listing 8-18 achieves that goal.

Listing 8-18: Creating a script tag type that executes only after the page has already loaded.

```
<script>
window.onload = function(){
  var scripts = document.getElementsByTagName("script");
  for ( var i = 0; i < scripts.length; i++ ) {
    if ( scripts[i].type == "onload" ) {
      // You may want to use a globalEval implementation here
      eval( scripts[i].innerHTML );
    }
  }
};
</script>
<script type="onload">
  ok(true, "I will only execute after the page has loaded.");
</script>
```

Obviously this technique could be extended beyond this simple example: Executing scripts on user interaction, when the DOM is ready to be manipulated, or even relatively based upon adjacent elements.

Meta-Languages

The most poignant example of the power of code evaluation can be seen in the implementation of other programming languages on top of the JavaScript language: dynamically converting these languages into JavaScript source and evaluating them.

There have been two such language conversions that have been especially interesting.

Processing.js

This was a port of the Processing Visualization Language (which is typically implemented using Java) to JavaScript, running on the HTML 5 Canvas element - by John Resig.

- Processing Visualization Language: http://processing.org/
- Processing.js: http://ejohn.org/blog/processingjs/

The result is a full programming language that you can usual to manipulate the visual display of a drawing area. Arguably Processing is particularly well suited towards it making it an effective port.

An example of Processing.js code can be seen in Listing 8-19.

Listing 8-19: An example of a class defined using Processing.

```
<script type="application/processing">
class SpinSpots extends Spin {
  float dim;
  SpinSpots(float x, float y, float s, float d) {
    super(x, y, s);
```

```
    dim = d;
  }
  void display() {
    noStroke();
    pushMatrix();
    translate(x, y);
    angle += speed;
    rotate(angle);
    ellipse(-dim/2, 0, dim, dim);
    ellipse(dim/2, 0, dim, dim);
    popMatrix();
  }
}
</script>
```

The above Processing code is then converted into the following JavaScript code and executed using an `eval()`, the resulting code can be seeing in Listing 8-20.

Listing 8-20: The end JavaScript result of the above Processing code.

```
function SpinSpots() {with(this){
  var __self=this;function superMethod(){
  extendClass(__self,arguments,Spin);
  this.dim = 0;
  extendClass(this, Spin);
  addMethod(this, 'display', function() {
    noStroke();
    pushMatrix();
    translate(x, y);
    angle += speed;
    rotate(angle);
    ellipse(-dim/2, 0, dim, dim);
    ellipse(dim/2, 0, dim, dim);
    popMatrix();
  });
  if ( arguments.length == 4 ) {
    var x = arguments[0];
    var y = arguments[1];
    var s = arguments[2];
    var d = arguments[3];
    superMethod(x, y, s);
    dim = d;
  }
}}
```

Using the Processing language you gain a few immediate benefits over using JavaScript alone:

- You get the benefits of Processing advanced features (Classes, Inheritance)
- You get Processing's simple drawing API
- You get all of the existing documentation and demos on Processing

The important point, though: All of this is occurring in pure JavaScript - all made possible because of code evaluation.

Objective-J

The second major project put forth was a port of the Objective-C programming language to JavaScript, called Objective-J, by the company 280 North for the product 280 Slides (an online slideshow builder).

- 280 Slides (uses Objective-J): http://280slides.com/

The 280 North team had extensive experience developing applications for OS X (which are primarily written in Objective-C). To create a more-productive environment to work in they ported the Objective-C language to JavaScript, providing a thin layer over the JavaScript language. An important point, though: They allow JavaScript code mixed in with their Objective-C code; the result of which they dub Objective-J, an example of which is in Listing 8-21.

Listing 8-21: An example of some Objective-J code.

```
// DocumentController.j
// Editor
//
// Created by Francisco Tolmasky.
// Copyright 2005 - 2008, 280 North, Inc. All rights reserved.

import <AppKit/CPDocumentController.j>
import "OpenPanel.j"
import "Themes.j"
import "ThemePanel.j"
import "WelcomePanel.j"

@implementation DocumentController : CPDocumentController
{
    BOOL    _applicationHasFinishedLaunching;
}

- (void)applicationDidFinishLaunching:(CPNotification)aNotification
{
    [CPApp runModalForWindow:[[WelcomePanel alloc] init]];
    _applicationHasFinishedLaunching = YES;
}

- (void)newDocument:(id)aSender
{
    if (!_applicationHasFinishedLaunching)
        return [super newDocument:aSender];

    [[ThemePanel sharedThemePanel]
        beginWithInitialSelectedSlideMaster:SaganThemeSlideMaster
          modalDelegate:self
            didEndSelector:@selector(themePanel:didEndWithReturnCode:)
              contextInfo:YES];
}

- (void)themePanel:(ThemePanel)aThemePanel
  didEndWithReturnCode:(unsigned)aReturnCode
{
    if (aReturnCode == CPCancelButton)
```

```
        return;

    var documents = [self documents],
        count = [documents count];

    while (count--)
        [self removeDocument:documents[0]];

    [super newDocument:self];
}
```

In their parsing application (which is written in JavaScript and converts the Objective-J code on-the-fly at runtime) they use light expressions to match and handle the Objective-C syntax, without disrupting the existing JavaScript. The result is a string of JavaScript code which is evaluated.

While this implementation has less far-reaching benefits (it's a specific hybrid language that can only be used within this context). It's potential benefits to users who are already familiar with Objective-C, but wish to explore web programming, will become greatly endowed.

Summary

In this chapter we've looked at the fundamentals of code evaluation in JavaScript. Exploring a number of different techniques for evaluating code and looking at what makes each method well suited to its conditions. We then looked at a variety of use cases for code evaluation along with some examples of how it has been best used. While potential for misuse will always remain, the incredible power that comes with harnessing code evaluation correctly is sure to give you an excellent tool to wield in your daily development.

Chapter 10. Strategies for Cross-Browser Code

In this chapter:

- A sound strategy for developing reusable JavaScript code.
- Analyzing the types of issues that need to be tackled.
- How to tackle issues in a smart way.

When developing capable cross-browser code there is a wide field of points, and counter-points, that need to be taken into consideration. Everything from the basic level of development to planning for future browser releases and web pages that have yet to be encountered. This is certainly a non-trivial task - the result of which must be balanced in a way that best works for your development methodologies. It doesn't matter if you wish your site to work in every browser that ever existed - and ever will exist - if you only have a finite amount of development resources. You must plan your resources appropriately and carefully; getting the maximum result from your effort.

Picking Your Core Browsers

Your primary concern should be taking into account which browsers will be the primary target on which your development will take place. As with virtually any aspect of web development you need to end up picking a number of browsers on which you'll want your users to have an optimal experience. When you choose to support a browser you are typically promising a couple things:

1. That you'll actively test against that browser with your test suite.
2. You'll fix bugs and regressions associated with that browser.
3. That the browser will have comparably reasonable performance.

For example, most JavaScript libraries end up supporting about 12 browser: The previous release, the current release, and the upcoming release of Internet Explorer, Firefox, Safari, and Opera. The choice of the JavaScript library is generally independent of the actual time required to support these browsers or their associated market share (as they are trying to, simultaneously, match actual market share and developer market share - which are often quite the opposite) - however this may not be the case in your development, an example of which is seen in Figure 9-1.

Figure 9-1: An example break-down of browser support for a JavaScript library.

Sample Browser Support Grid

	IE	Firefox	Safari	Opera
Previous	6.0	2.0	2.0	9.2
Current	7.0	3.0	3.1	9.5
Next	8.0	3.1	4.0	10.0

A JavaScript library (or, really, any piece of reusable JavaScript code) is in a unique situation: It must be willing to encounter a large number of situations in which their code may not work correctly. Thus it is optimal to try and work on as many platforms as possible, simultaneously, without sacrificing quality or efficiency.

In order to understand what we're up against we must break down the different types of situations that JavaScript code will encounter and then examine the best ways to write preventative code which will assuage any potential problems that might occur.

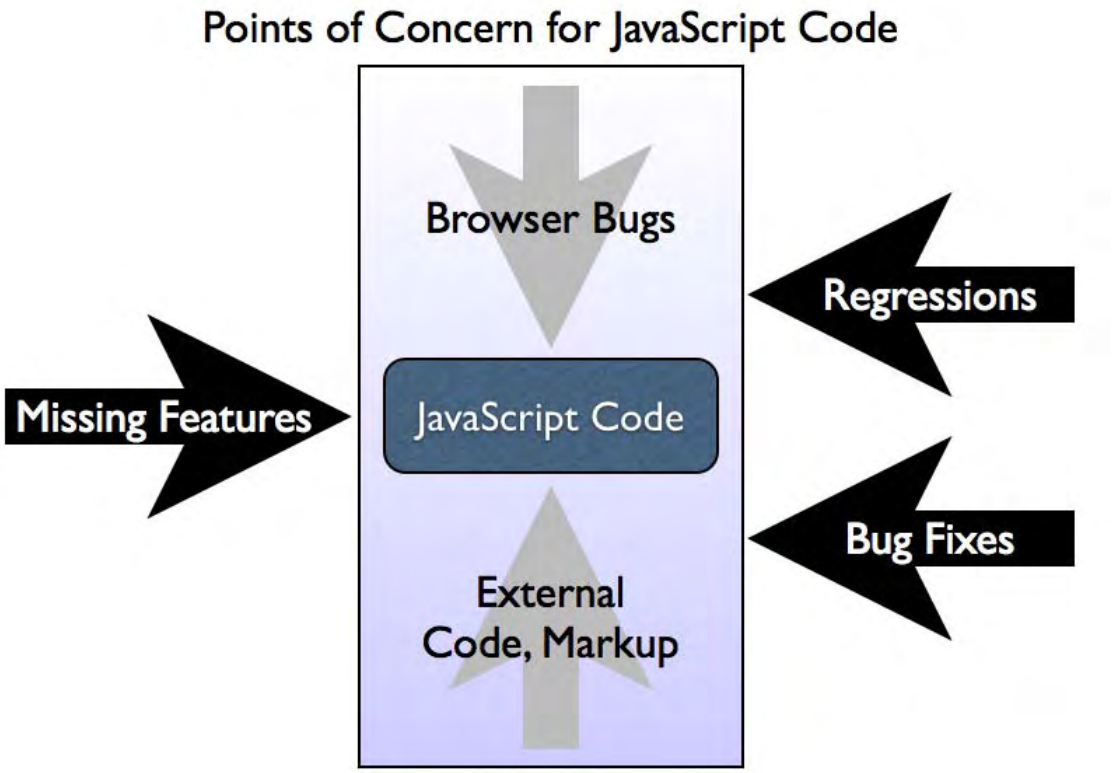
Development Concerns

There are about five major points on which your reusable JavaScript code will be challenged. You'll want to balance how much time you spend on each point with how much benefit you'll receive in the end result. Is an extra 40 hours of development time worth better support for Netscape Navigator 4? Ultimately it's a question that you'll have to answer yourself, but it's one that can be determined rather concretely after some simple analysis.

There's one axiom that can be used when developing: **Remember the past, consider the future - test the present.**

When attempting to develop reusable JavaScript code you must take all points into consideration: You have to pay primary attention to the most-popular browsers that exist right now (as stated previously), then you have to take into concern what changes are upcoming in the next versions of the browsers, and then try to keep good compatibility with old browser versions (supporting as many features as you can without being susceptible to becoming unusable), summarized in Figure 9-2.

Figure 9-2: A breakdown of the different points that need to be considered when developing reusable JavaScript code.



Let's break down the various concerns so that we can have a better understanding of what we're up against.

Browser Bugs

The primary concern of reusable JavaScript code development should be in handling the browser bugs and API inequality associated with the set of browsers that you support. This means that any features that you provide in your application should be completely - and verifiably - usable in all of those browsers.

The solution to this is quite straight forward, albeit challenging: You need a comprehensive suite of tests to cover the common (and fringe) use cases of your code. With good coverage you can feel safe in knowing that the code that you develop will, both, work in the most commonly-used browsers (hopefully covering some 95-99% of your total browser market share) and in the next versions of those browsers (giving you that guaranteed percentage many years into the future).

The trick becomes, however: How can you implement fixes for current browser bugs in a way that is resistant to fixes implemented in future versions of the browser? Assuming that a browser will forever contain a bug is quite foolhardy and is a dangerous development strategy. Its best to use a form of Feature Simulation (as we discuss later on) to determine if a bug has been fixed (and that a browser feature works as you expect it to).

Where the line blurs, though, is in determining what exactly a browser bug is. If a browser implements a private API for handling a piece of functionality can you safely assume that the API will be maintained? We'll look at a number of these issues when we examine how to handle regressions and bug fixes coming in from future browser versions.

External Code and Markup

A tricky point in developing reusable JavaScript code is in the code and markup that surrounds it. If you're expecting your code to work on any web site then you need to expect it to be able to handle the random code associated with it. This has two meanings: Your code must strive to be able to survive poorly written external code and not affect any external code.

This point of concern depends a lot upon which environments you expect your code to be used. For example, if you are using your reusable code on a limited number of web sites, but targeting a large number of browsers, it's probably best to worry less about external code (as you have the power and control to fix it yourself). However if you're trying to develop code that'll have the broadest applicability you'll need to make sure that your code is robust and effective.

To keep your code from affecting other pieces of code on the page it's best to practice encapsulation. Keep an incredibly-small footprint when introducing your code into a page (usually a single global variable). This includes modifying existing variables, function prototypes, or even DOM elements. Any place that your code modifies is a potential area for collision and confusion.

You can even extend these practices further and introduce other libraries in your test suite - making sure that their code doesn't affect how yours performs, and vice versa.

Secondly, when expecting outside code or markup you need to assume the worst (that they will be modifying function prototypes and existing DOM element methods, etc.). There aren't many steadfast rules when dealing when situations of this nature, but here are a couple tips:

hasOwnProperty: When looping through the properties of an object you'll want to verify that the properties that you're expecting are from the object itself and not introduced by an extension to Object.prototype. Thankfully the number of scripts that use this technique is very small but the harm is quite large. You can counter this by using the method `hasOwnProperty` to determine if the property is coming from this object or another one, as shown in Listing 9-1.

Listing 9-1: Using hasOwnProperty to build resilient object looping.

```
Object.prototype.otherKey = "otherValue";

var object = { key: "value" };
for ( var prop in object ) {
  if ( object.hasOwnProperty( prop ) ) {
    assert( prop, "key",
  "There should only be one iterated property." );
  }
}
```

Greedy IDs: Internet Explorer and Opera both have the concept of "greedy DOM IDs" - in which elements become properties of other objects based upon their ID names. Listing 9-2 shows two examples of this nasty trick in action.

Listing 9-2: A simple example of markup that's capable of causing great confusion in development.

```
<form id="form">
  <input type="text" id="length"/>
  <input type="submit" id="submit"/>
</form>
```

If you were to call `document.getElementsByTagName("input").length` you'd find that the length property (which should contain a number) would now contain a

reference to the input element with the ID of length. Additionally if you were to call `document.getElementById("form").submit()` that would no longer work as the submit method will have be overwritten by the submit input element. This particular "feature" of the browser can cause numerous problems in your code and will have to be taken in to consideration when calling default properties in the above situations.

Order of Stylesheets: The best way to ensure that CSS rules (provided by stylesheets) are available when the JavaScript code on the page executes is to specifying the external stylesheets prior to including the external script files. Not doing so can cause unexpected results as the script attempts to access the incorrect style information. Unfortunately this isn't an issue that can be easily rectified with pure JavaScript and should be, instead, relegated to user documentation.

These are just some basic examples of how externalities can truly affect how your code works - frequently in a very unintentional manner. Most of the time these issues will pop up once users try to integrate your code into their sites - at which point you'll be able to diagnose the issue and build appropriate tests to handle them. It's unfortunate that there are no better solutions to handling these integration issues other than to take some smart first steps and to write your code in a defensive manner.

Missing Features

For a large segment of the browsers that exist, but aren't in your list of actively supported browsers, they will most-likely be missing some key features that you need to power your code (or have enough bugs to make them un-targetable).

The fundamental problem, in this area, is that the harder you work to support more browsers the less your ultimate result will be worth. For example, if you worked incredibly hard and were able to bring Internet Explorer 4 support to your library what would the end result be? There would be so few users and the time that you spent to create that result would be so much as to be ludicrous. And that's saying nothing of determining if the CSS of your page would render in this situation, as well.

The strategy usually becomes, at this point: How can we get the most functionality delivered to the user while failing gracefully when we cannot. However there is one flaw in this, take this example: What if a browser is capable of initializing and hiding a number of pieces of navigation on a page (in hopes of creating a drop-down menu) but then the event-related code doesn't work. The result is a half-functional page, which helps no one.

The better strategy is to design your code to be as backwards-compatible as possible and then actively direct known failing browsers over to a tailored version of the page. Yahoo! adopts this strategy with their web sites, breaking down browsers with graded levels of support. After a certain amount of time they "blacklist" a browser (usually when it hits an infinitesimal market-share - like 0.05%) and direct users of that browser (based upon the detected user agent) to a pure-HTML version of the application (no CSS or JavaScript involved).

This means that their developers are able to target, and build, an optimal experience for the vast majority of their users (around 99%) while passing off antiquated browsers to a functional equivalent (albeit with a less-optimal experience).

The key points of this strategy:

- No assumptions are made about the user experience of old browsers. After a browser is no longer able to be tested (and has a negligible market-share) it is simply cut off and served with a simplified page.
- All users of current and past browsers are guaranteed to have a page that isn't broken.
- Future/unknown browsers are assumed to work.

With this strategy most of the extra development effort (beyond the currently-targeted browsers and platforms) is focused towards handling future browsers. This is a smart strategy as it will allow your applications to last longer with only minimal changes.

Bug Fixes

When writing a piece of reusable JavaScript code you'll want to make sure that it's able to last for a long time. As with writing any aspect of a web site (CSS, HTML, etc.) it's undesirable to have to go back and change a broken web site (caused by a new browser release).

The most common form of web site breakage comes in when making assumptions about browser bugs. This can be generalized as: Specific hacks that are put in place to workaround bugs introduced by a browser, which break when the browsers fix the bugs in future releases. The issue is frequently circumvented by building pieces of feature simulation code (discussed in this chapter) instead of making assumptions about the browser.

The issue with handling browser bugs callously is two-fold:

1. First, your code will break if the browser ever resolves the issue.

2. Second, browser vendors become more inclined to not fix the bugs, for fear of causing web sites to break.

An interesting example of the above situation occurred during the development of Firefox 3. A change was introduced which forced DOM nodes to be adopted by a DOM document if they were going to be injected into them (in accordance with the DOM specification), like in Listing 9-3.

Listing 9-3: Examples of having to use adoptNode before injecting a node into a new document.

```
// Shouldn't work
var node = documentA.createElement("div");
documentB.documentElement.appendChild( node );

// Proper way
var node = documentA.createElement("div");
documentB.adoptNode( node );
documentB.documentElement.appendChild( node );
```

However, since there was a bug in Firefox (it allowed the first situation to work, when it shouldn't have) users wrote their code in a manner that depended on that code working. This forced Mozilla to rollback their change, for fear of breaking a number of web sites.

This brings up an important point concerning bugs, though: When determining if a piece of functionality is, potentially, a bug - go back to the specification. In the above case Internet Explorer was actually more forceful (throwing an exception if the node wasn't in the correct document - which was the correct behavior) but users just assumed that it was an error with Internet Explorer, in that case, and wrote conditional code to fall back. This caused a situation in which users were following the specification for only a subset of browsers and forcefully rejecting it in others.

A browser bug should be differentiated from an unspecified API. It's important to point back to browser specifications since those are the exact standards that browsers will be using in order to develop and improve their code. Whereas with an unspecified API the implementation could change at any point (especially if the implementation ever attempts to become specified - and changed in the process). In the case of inconsistency in unspecified APIs you should always test for your expected output, running additional cases of feature simulation. You should always be aware of future changes that could occur in these APIs as they become solidified.

Additionally, there's a distinction between bug fixes and API changes. Whereas bug fixes are easily foreseen (a browser will eventually fix the bugs with its implementation - even if it takes a long amount of time) API changes are much harder to spot. While you should always be cautious when using an unspecified API it is possible for a specified API to change. While this will rarely happen in a way that will massively break most web applications the result is effectively undetectable (unless, of course, you tested every single API that you ever touched - but the overhead incurred in such an action would be ludicrous). API changes in this manner should be handled like any other regression.

In summary: Blanketing a bug to a browser is very dangerous. Your web application will break, it's only a matter of time before it occurs. Using feature simulation to gracefully fall back to a working implementation is the only smart way to handle this issue.

Regressions

Regressions are the hardest problem that you can encounter in reusable, sustainable, JavaScript development. These are bugs, or API changes, that browsers have introduced that caused your code to break in unpredicted ways.

There are some API changes that can be easily detected and handled. For example, if Internet Explorer introduces support for DOM event handlers (bound using addEventListener) simple object detection will be able to handle that change, as shown in Listing 9-4.

Listing 9-4: A simple example of catching the implementation of a new API.

```
function attachEvent( elem, type, handle ) {
  // bind event using proper DOM means
  if ( elem.addEventListener )
    elem.addEventListener(type, handle, false);

  // use the Internet Explorer API
  else if ( elem.attachEvent )
    elem.attachEvent("on" + type, handle);
}
```

However most API changes aren't that easy to predict and neither are bugs. For this reason the best that you can hope for is to be diligent in your monitoring and testing in upcoming browser releases.

For example, in Internet Explorer 7 a basic XMLHttpRequest wrapper around the native ActiveX request object. This caused virtually all JavaScript libraries to default to using the XMLHttpRequest object to perform their Ajax requests (as it should - opting to use a standards-based API is nearly always optimal). However, in Internet Explorer's implementation they broke the handling of requesting local files (a site loaded from the desktop could no longer request files using the XMLHttpRequest object). No one really caught this bug (or really could've pre-empted it) until it was too late, causing it to escape into the wild and breaking many pages in the process. The solution was to opt to use the ActiveX implementation primarily for local file requests.

Keeping close track of upcoming browser releases is absolutely the best way to avoid future regressions of this nature. It doesn't have to be a complete tax on your normal development cycle - it can be as simple as opening your applications in each of the upcoming browsers every couple weeks to make sure no major regressions have taken place.

You can get the upcoming browser releases from the following locations:

- Internet Explorer (their blog - but releases are announced here): http://blogs.msdn.com/ie/
- Firefox: http://ftp.mozilla.org/pub/mozilla.org/firefox/nightly/latest-trunk/

- WebKit (Safari): http://nightly.webkit.org/
- Opera: http://snapshot.opera.com/

Diligence is important in this respect. Since you can never be fully precogniscent of the bugs that will be introduced by a browser it is best to make sure that you stay on top of your code and avert any crises that may arrive.

Thankfully, browser vendors are doing a lot to make sure that regressions of this nature do not occur. Both Firefox and Opera have test suites from various JavaScript libraries integrated into their main browser test suite. This allows them to be sure that no future regressions will be introduced that affect those libraries directly. While this wont catch all regressions (and certainly wont in all browsers) it's a great start and shows good progress by the browser vendors towards preventing as many issues as possible.

Implementing Cross-Browser Code

Knowing which issues to be aware of is only half the battle. Figuring out effective strategies for implementing cross-browser code is a whole other aspect to development. There exist a range of strategies that can be used. While not every strategy will work in every situation - the combination should provide good coverage for most concerns with a code base.

Safe Cross-Browser Fixes

The simplest, and safest, class of cross-browser fixes are those that will have no negative effects on other browsers whatsoever, while simultaneously using no forms of browser or feature detection.

These instances are, generally, rather rare - but it's a result that should always be strived for in your applications. To give an example, examine Listing 9-5.

Listing 9-5: Preventing negative values to be set on CSS height and width properties.

```
// ignore negative width and height values
if ( (key == 'width' || key == 'height') && parseFloat(value) < 0 )
  value = undefined;
```

This change, in jQuery, came about when working with Internet Explorer. The browser throws an exception when a negative value is set on height or width style properties. All other browsers simply ignore this input. The workaround that was introduced, shown in the above listing, was to simply ignore all negative values - in all browsers. This change prevented an exception from being thrown in Internet Explorer and had no effect on any other browser. This was a painless addition which provided a unified API to the user (throwing unexpected exceptions is never desired).

Another example of this simplification can be seen in jQuery in the attribute manipulation code is shown in Listing 9-6.

Listing 9-6: Disallow attempts to change the type attribute on input elements in all browsers.

```
if ( name == "type" && elem.nodeName.toLowerCase() == "input" &&
     elem.parentNode )
  throw "type attribute can't be changed";
```

Internet Explorer doesn't allow you to manipulate the type attribute of input elements that have already been inserted into the document. Attempts to do so result in an unpredictable exception being thrown. jQuery came to a middle-ground solution: It simply disallowed all attempts to manipulate the type attribute on injected input elements in all browsers equally (throwing an informational exception).

This change to the jQuery code base required no browser or feature detection - it was simply introduced as a means of unifying the API across all browsers. Certainly this particular feature addition is quite controversial - it actively limits the features of the library in browsers that are unaffected by this bug. The jQuery team weighed the decision carefully and decided that it was better to have a unified API that worked consistently than an API would break unexpectedly when developing cross-browser. It's very possible that you'll come across situations like this when developing your own reusable code bases.

The important thing to remember from these style of code changes: They provide a solution that works seamlessly across browsers with no browser or feature detection (effectively making them immune to changes going forward). One should always strive to use solutions that work in this manner - even if they are few and far-between.

Object Detection

Object detection is one of the most commonly used ways of writing cross-browser code. It's simple and generally quite effective. It works by simply determining if a certain object or object property exists (and, if so, assuming that it provides the functionality implies).

Most common object detection is used to toggle between multiple APIs that provide duplicate pieces of functionality. For example, like in the code shown in Listing 9-4 and Listing 9-7, object detection is frequently used to choose the appropriate event-binding APIs provided by the browser.

Listing 9-7: Binding an event listener using the W3C DOM Events API or the Internet Explorer-specific API using object detection.

```
function attachEvent( elem, type, handle ) {
  // bind event using proper DOM means
  if ( elem.addEventListener )
    elem.addEventListener(type, handle, false);

  // use the Internet Explorer API
  else if ( elem.attachEvent )
    elem.attachEvent("on" + type, handle);
}
```

In this case we look to see if a property exists named `addEventListener` and, if so, we assume that it's a function that we can execute and that it'll bind an event listener to that element. Note that we start with `addEventListener` (the method provided by the W3C DOM Events specification). This is intentional. Whenever possible we should strive to default to the specified way of performing an action. As mentioned before this will help to make our code advance well into the future and encourage browser vendors to work towards the specified way of performing actions.

One of the most important uses of object detection is in detecting the basic features of a library, or application, and providing a fallback. As discussed previously you'll want to target your code to a specific set of browsers. The best way to do this is to figure out the APIs that the browser provides, that you need, and test for their existence.

Listing 9-9 has a basic example of detecting a number of browser features, using object detection, in order to determine if we should be providing a full application or a reduced fallback experience.

Listing 9-9: An example of using object detection to provide a fallback experience for a browser.

```
if ( typeof document !== "undefined" &&
     (document.addEventListener || document.attachEvent) &&
     document.getElementsByTagName && document.getElementById ) {
```

```
  // We have enough of an API to work with to build our application
} else {
  // Provide Fallback
}
```

What is done in the fallback is up to you. There are a couple options:

1. You could do further object detection and figure out how to provide a reduce experience that still uses some JavaScript.

2. Simply opt to not execute any JavaScript (falling back to the HTML on the page).

3. Redirect a user to a plain version of the site (Google does this with GMail, for example).

Since object detection has very little overhead associated with it (it's just simple property/object lookups) and is relatively simplistic in its implementation, it makes for a good candidate to provide basic levels of fallbacks - both at the API and at the application level. It absolutely should serve as a good first line of defense in your reusable code authoring.

Feature Simulation

The final means that we have of preventing regressions - and the most effective means of detecting fixes to browser bugs - exists in the form of feature simulation. Contrary to object detection, which are just simple object/property lookups, feature simulation runs a complete run-through of a feature to make sure that it works as one would expect it to.

Typically object detection isn't enough to know if a feature will behave as is intended but if we know of specific bugs we can quickly build contrived tests for future knowledge. For example, Internet Explorer will return both elements and comments if you do `getElementsByTagName("*")`. Doing simple object detection isn't enough to determine if this will happen. Additionally this bug will, most likely, be fixed by the Internet Explorer team at some future release of the browser.

We can see an example of using feature simulation to determine if the `getElementsByTagName` method will work as we expect it to in Listing 9-10.

Listing 9-10: Defining a global variable that captures information about how a browser feature works and using it later on.

```
// Run once, at the beginning of the program
var ELEMENTS_ONLY = (function(){
  var div = document.createElement("div");
  div.appendChild( document.createComment("test" ) );
  return div.getElementsByTagName("*").length === 0;
})();

// Later on:
var all = document.getElementsByTagName("*");

if ( ELEMENTS_ONLY ) {
  for ( var i = 0; i < all.length; i++ ) {
    action( all[i] );
  }
} else {
  for ( var i = 0; i < all.length; i++ ) {
    if ( all[i].nodeType === 1 ) {
      action( all[i] );
```

```
    }
  }
}
```

Feature simulation works in two ways. To start a simple test is run to determine if a feature work as we expect it to. It's important to try and verify the integrity of a feature (making sure it works correctly) rather than explicitly testing for the presence of a bug. It's only a semantic distinction but it's one that its important to maintain in your coding.

Secondly we use the results of the test run later on in our program to speed up looping through an array of elements. Since a browser that returns only elements doesn't need to perform these element checks on every stage of the loop we can completely skip it and get the performance benefit in the browsers that work correctly.

The is the most common idiom used in feature simulation: Making sure a feature works as you expect it to and providing browsers, that work correctly, with optimized code - an example of which can be seen in Listing 9-11.

Listing 9-11: Figure out the attribute name used to access textual element style information.

```
<div id="test" style="color:red;"></div>
<div id="test2"></div>
<script>
// Perform the initial attribute check
var STYLE_NAME = (function(){
  var div = document.createElement("div");
  div.style.color = "red";

  if ( div.getAttribute("style") )
    return "style";

  if ( div.getAttribute("cssText") )
    return "cssText";
})();

// Later on:
window.onload = function(){
  document.getElementsById("test2").getAttribute( STYLE_NAME ) =
      document.getElementById("test").getAttribute( STYLE_NAME );
};
</script>
```

In this example we, again, break down into two parts. We start by creating a dummy element, setting a style property, and then attempt to get the complete style information back out again. We start by using the standards-compliant way (accessing the style attribute) then by trying the Internet Explorer-specific way. In either case we return the name of the property that worked.

We can then use that property name at any point in our code when it comes time to get or set style attribute values, which is what we do when the page fully loads.

There's a couple points to take into consideration when examining feature simulation. Namely that it's a trade-off. For the extra performance overhead of the initial simulation and extra lines of code added to your program you get the benefit of knowing exactly how a feature works in the current browser and you become immune of future bug fixes. The immunity can be absolutely priceless when dealing with reusable code bases.

Untestable

Unfortunately there are a number of features in JavaScript and the DOM that are either impossible or prohibitively expensive to test. These features are, generally, quite rare. If you encounter an issue in your code that appears to be untestable it will always pay to investigate the matter further.

Following are some issues that are impossible to be tested using any conventional JavaScript interactions.

Determining if an event handler has been bound. Browsers do not provide a way of determining if if any functions have been bound to an event listener on an element. Because of this there is no way to remove all bound event handlers from an element unless you have foreknowledge of, and maintain references to, all bound handlers.

Determining if an event will fire. While it's possible to determine if a browser supports a means of binding an event (such as using `bindEventListener`) it's not possible to know if a browser will actually fire an event. There are a couple places where this becomes problematic.

First, if a script is loaded dynamically, after the page has already loaded, it may try to bind a listener to wait for the window to load when, in fact, that event happened some time ago. Since there's no way to determine that the event already occurred the code may wind up waiting indefinitely to execute.

Second, if a script wishes to uses specific events provided by a browser as an alternative. For example Internet Explorer provides mouseenter and mouseleave events which simplify the process of determining when a user's mouse enters and leaves an element. These are frequently used as an alternative to the mouseover and mouseout events. However since there's no way of determining if these events will fire without first binding the events and waiting for some user interaction against them it becomes prohibitive to use them for interaction.

Determining if changing certain CSS properties affects the presentation. A number of CSS properties only affect the visual representation of the display and nothing else (don't change surrounding elements or event other properties on the element) - like color, backgroundColor, or opacity. Because of this there is no way to programmatically determine if changing these style properties are actually generating the effects that are desired. The only way to verify the impact is through a visual examination of the page.

Testing script that causes the browser to crash. Code that causes a browser to crash is especially problematic since, unlike exceptions which can be easily caught and handled, these will always cause the browser to break.

For example, in older versions of Safari, creating a regular expression that used Unicode characters ranges would always cause the browser to crash, like in the following example:

```
new RegExp("[\\w\u0128-\uFFFF*_-]+");
```

The problem with this occurring is that it's not possible to test how this works using regular feature simulation since it will always produce an undesired result in that older browser. Additionally bugs that cause crashes to occur forever become embroiled in difficulty since while it may be acceptable to have JavaScript be disabled in some segment of the population using your browser, it's never acceptable to outright crash the browser of those users.

Testing bugs that cause an incongruous API. In Listing 9-6 when we looked at disallowing the ability to change the type attribute in all browsers due to a bug in Internet Explorer. We could test this feature and only disable it in Internet Explorer however that would give us a result where the API works differently from browser-to-browser. In issues like this - where a bug is so bad that it causes an API to break - the only option is to write around the affected area and provide a different solution.

The following are items that are not impossible to test but are prohibitively difficult to test effectively.

The performance of specific APIs. Sometimes specific APIs are faster or slower in different browsers. It's important to try and use the APIs that will provide the fastest results but it's not always obvious which API will yield that. In order to do effective performance analysis of a feature you'll need to make it difficult enough as to take a large amount of time in order

Determining if Ajax requests work correctly. As mentioned when we looked at regressions, Internet Explorer broke requesting local files via the XMLHttpRequest object in Internet Explorer 7. We could test to see if this bug has been fixed but in order to do so we would have to perform an extra request on every page load that attempted to perform a request. Not only that but an extra file would have to be included with the library whose sole purpose would be to exist for these extra requests. The overhead of both these matters is quite prohibitive and would, certainly, not be worth the extra effort of simply doing object detection instead.

While untestable features are a significant hassle (limiting the effectiveness of writing reusable JavaScript) they are almost always able to be worked around. By utilizing alternative techniques, or constructing your APIs in a manner as to obviate these issues in the first place, it will most likely be possible that you'll be able to build effective code, regardless.

Implementation Concerns

Writing cross-browser, reusable, code is a battle of assumptions. By using better means of detection and authoring you're reducing the number of assumptions that you make in your code. When you make assumptions about the code that you write you stand to encounter problems later on. For example, if you assume that a feature or a bug will always exist in a specific browser - that's a huge assumption. Instead, testing for that functionality proves to be much more effective.

In your coding you should always be striving to reduce the number of assumptions that you make, effectively reducing the room that you have for error. The most common area of assumption-making, that is normally seen in JavaScript, is that of user agent detection. Specifically, analyzing the user agent provided by a browser (`navigator.userAgent`) and using it to make an assumption about how the browser will behave. Unfortunately, most user agent string analysis proves to be a fairly large source of future-induced errors. Assuming that a bug will always be linked to a specific browser is a recipe for disaster.

However, there is one problem when dealing with assumptions: it's virtually impossible to remove all of them. At some point you'll have to assume that a browser will do what it proposes. Figuring out the best point at which that balance can be struck is completely up to the developer.

For example, let's re-examine the event attaching code that we've been looking at, in Listing 9-12.

Listing 9-12: A simple example of catching the implementation of a new API.

```
function attachEvent( elem, type, handle ) {
  // bind event using proper DOM means
  if ( elem.addEventListener )
    elem.addEventListener(type, handle, false);

  // use the Internet Explorer API
  else if ( elem.attachEvent )
    elem.attachEvent("on" + type, handle);
}
```

In the above listing we make three assumptions, namely:

1. That the properties that we're checking are, in fact, callable functions.

2. That they're the right functions, performing the action that we expect.

3. That these two methods are the only possible ways of binding an event.

We could easily negate the first assumption by adding checks to see if the properties are, in fact, functions. However how we tackle the remaining two points is much more problematic.

In your code you'll need to decide how many assumptions are a correct level for you. Frequently when reducing the number of assumptions you also increase the size and complexity of your code base. It's fully possible to attempt to reduce assumptions to the point of insanity but at some point you'll have to stop and take stock of what you have and work from there. Even the least assuming code is still prone to regressions introduced by a browser.

Summary

Cross-browser development is a juggling act between three points:

- Code Size: Keeping the file size small.
- Performance Overhead: Keeping the performance hits to a palatable minimum.
- API Quality: Making sure that the APIs that are provided work uniformly across browsers.

There is no magic formula for determining what the correct balance of these points are. They are something that will have to be balanced by every developer in their individual development efforts. Thankfully using smart techniques like object detection and feature simulation it's possible to defend against any of the numerous directions from which reusable code will be attacked, without making any undue sacrifices.

Chapter 11. CSS Selector Engine

In this chapter:

- The tools that we have for building a selector engine
- The strategies for engine construction

CSS selector engines are a, relatively, new development in the world of JavaScript but they've taken the world of libraries by storm. Every major JavaScript library includes some implementation of a JavaScript CSS selector engine.

The premise behind the engine is that you can feed it a CSS selector (for example "div > span") and it will return all the DOM elements that match the selector on the page (in this case all spans that are a child of a div element). This allows users to write terse statements that are incredibly expressive and powerful. By reducing the amount of time the user has to spend dealing with the intricacies of traversing the DOM it frees them to handle other tasks.

It's standard, at this point, that any selector engine should implement CSS 3 selectors, as defined by the W3C:

- http://www.w3.org/TR/css3-selectors/

There are three primary ways of implementing a CSS selector engine:

1. Using the W3C Selectors API - a new API specified by the W3C and implemented in most newer browsers.
2. XPath - A DOM querying language built into a variety of modern browsers.
3. Pure DOM - A staple of CSS selector engines, allows for graceful degradation if either of the first two don't exist.

This chapter will explore each of these strategies in depth allowing you to make some educated decisions about implementing, or at least understanding, a JavaScript CSS selector engine.

Selectors API

The W3C Selectors API is a, comparatively, new API that is designed to reduce much of the work that it takes to implement a full CSS selector engine in JavaScript. Browser vendors have pounced on this new API and it's implemented in all major browsers (starting in Safari 3, Firefox 3.1, Internet Explorer 8, and Opera 10). Implementations of the API generally support all selectors implemented by the browser's CSS engine. Thus if a browser has full CSS 3 support their Selectors API implementation will reflect that.

The API provides two methods:

- `querySelector`: Accepts a CSS selector string and returns the first element found (or null if no matching element is found).
- `querySelectorAll`: Accepts a CSS selector string and returns a static NodeList of all elements found by the selector.

and these two methods exist on all DOM elements, DOM documents, and DOM fragments.

Listing 10-1 has a couple examples of how it could be used.

Listing 10-1: Examples of the Selectors API in action.

```
<div id="test">
  <b>Hello</b>, I'm a ninja!
</div>
<div id="test2"></div>
<script>
window.onload = function(){
  var divs = document.querySelectorAll("body > div");
  assert( divs.length === 2, "Two divs found using a CSS selector." );

  var b = document.getElementById("test")
    .querySelector("b:only-child");
  assert( b,
  "The bold element was found relative to another element." );
};
</script>
```

Perhaps the one gotcha that exists with the current Selectors API is that it more-closely capitulates to the existing CSS selector engine implementations rather than the implementations that were first created by JavaScript libraries. This can be seen in the matching rules of element-rooted queries, as seen in Listing 10-2.

Listing 10-2: Element-rooted queries.

```
<div id="test">
  <b>Hello</b>, I'm a ninja!
</div>
<script>
window.onload = function(){
  var b = document.getElementById("test").querySelector("div b");
  assert( b, "Only the last part of the selector matters." );
};
</script>
```

Note the issue here: When performing an element-rooted query (calling `querySelector` or `querySelectorAll` relative to an element) the selector only checks to see if the final portion of the selector is contained within the element. This will probably seem counter-intuitive (looking at the previous listing we can verify that there are no div elements with the element with an ID of test - even though that's what the selector looks like it's verifying).

Since this runs counter to how most users expect a CSS selector engine to work we'll have to provide a work-around. The most common solution is to add a new ID to the rooted element to enforce its context, like in Listing 10-3.

Listing 10-3: Enforcing the element root.

```
<div id="test">
  <b>Hello</b>, I'm a ninja!
</div>
<script>
(function(){
  var count = 1;

  this.rootedQuerySelectorAll = function(elem, query){
    var oldID = elem.id;
    elem.id = "rooted" + (count++);
```

```
    try {
      return elem.querySelectorAll( "#" + elem.id + " " + query );
    } catch(e){
      throw e;
    } finally {
      elem.id = oldID;
    }
  };
})();

window.onload = function(){
  var b = rootedQuerySelectorAll(
    document.getElementById("test"), "div b");
  assert( b.length === 0, "The selector is now rooted properly." );
};
</script>
```

Looking at the previous listing we can see a couple important points. To start we must assign a unique ID to the element and restore the old ID later. This will ensure that there are no collisions in our final result when we build the selector. We then prepend this ID (in the form of a "#id " selector) to the selector.

Normally it would be as simple as removing the ID and returning the result from the query but there's a catch: Selectors API methods can throw exceptions (most commonly seen for selector syntax issues or unsupported selectors). Because of this we'll want to wrap our selection in a try/catch block. However since we want to restore the ID we can add an extra finally block. This is an interesting feature of of the language - even though we're returning a value in the try, or throwing an exception in the catch, the code in the finally will always execute after both of them are done executing (but before the value is return from the function or the object is thrown). In this manner we can verify that the ID will always be restored properly.

The Selectors API is absolutely one of the most promising APIs to come out of the W3C in recent history. It has the potential to completely replace a large portion of most JavaScript libraries with a simple method (naturally after the supporting browsers gain a dominant market share).

XPath

A unified alternative to using the Selectors API (in browsers that don't support it) is the use of XPath querying. XPath is a querying language utilized for finding DOM nodes in a DOM document. It is significantly more powerful than traditional CSS selectors. Most modern browsers (Firefox, Safari 3+, Opera 9+) provide some implementation of XPath that can be used against HTML-based DOM documents. Internet Explorer 6, 7, and 8 provide XPath support for XML documents (but not against HTML documents - the most common target).

If there's one thing that can be said for utilizing XPath expressions: They're quite fast for complicated expressions. When implementing a pure-DOM implementation of a selector engine you are constantly at odds with the ability of a browser to scale all the JavaScript and DOM operations. However, XPath loses out for simple expressions.

There's a certain indeterminate threshold at which it becomes more beneficial to use XPath expressions in favor of pure DOM operations. While this might be able to be determined programatically there are a few gives: Finding elements by ID ("#id") and simple tag-based selectors ("div") will always be faster with pure-DOM code.

If you and your users are comfortable using XPath expressions (and are happy limiting yourself to the modern browsers that support it) then simply utilize the method shown in Listing 10-4 and completely ignore everything else about building a CSS selector engine.

Listing 10-4: A method for executing an XPath expression on an HTML document, returning an array of DOM nodes, from the Prototype library.

```
if ( typeof document.evaluate === "function" ) {
  function getElementsByXPath(expression, parentElement) {
    var results = [];
    var query = document.evaluate(expression,
      parentElement || document,
      null, XPathResult.ORDERED_NODE_SNAPSHOT_TYPE, null);
    for (var i = 0, length = query.snapshotLength; i < length; i++)
      results.push(query.snapshotItem(i));
    return results;
  }
}
```

While it would be nice to just use XPath for everything it simply isn't feasible. XPath, while feature-packed, is designed to be used by developers and is prohibitively complex, in comparison to the expressions that CSS selectors make easy. While it simply isn't feasible to look at the entirety of XPath we can take a quick look at some of the most common expressions and how they map to CSS selectors, in Table 10-1.

Table 10-1: Map of CSS selectors to their associated XPath expressions.

Table 11.1.

Goal	CSS 3	XPath
All Elements	*	//*
All P Elements	p	//p
All Child Elements	p > *	//p/*
Element By ID	#foo	//*[@id='foo']
Element By Class	.foo	//*[contains(concat(" ", @class, " "),," foo ")]
Element With Attribute	*[title]	//*[@title]
First Child of All P	p > *:first-child	//p/*[0]
All P with an A descendant	Not possible	//p[a]
Next Element	p + *	//p/following-sibling::*[0]

Using XPath expressions would work as if you were constructing a pure-DOM selector engine (parsing the selector using regular expressions) but with an important deviation: The resulting CSS selector portions would get mapped to their associated XPath expressions and executed.

This is especially tricky since the result is, code-wise, about as large as a normal pure-DOM CSS selector engine implementation. Many developers opt to not utilize an XPath engine simply to reduce the complexity of their resulting engines. You'll need to weigh the performance benefits of an XPath engine (especially taking into consideration the competition from the Selectors API) against the inherent code size that it will exhibit.

DOM

At the core of every CSS selector engine exists a pure-DOM implementation. This is simply parsing the CSS selectors and utilizing the existing DOM methods (such as getElementById or getElementsByTagName) to find the corresponding elements.

It's important to have a DOM implementation for a number of reasons:

1. Internet Explorer 6 and 7. While Internet Explorer 8 has support for querySelectorAll the lack of XPath or Selectors API support in 6 and 7 make a DOM implementation necessary.

2. Backwards compatibility. If you want your code to degrade in a graceful manner and support browsers that don't support the Selectors API or XPath (like Safari 2) you'll have to have some form of a DOM implementation.

3. For speed. There are a number of selectors that a pure DOM implementation can simply do faster (such as finding elements by ID).

With that in mind we can take a look at the two possible CSS selector engine implementations: Top down and bottom up.

A top down engine works by parsing a CSS selector from left-to-right, matching elements in a document as it goes, working relatively for each additional selector segment. It can be found in most modern JavaScript libraries and is, generally, the preferred means of finding elements on a page.

For example, given the selector "div span" a top down-style engine will find all div elements in the page then, for each div, find all spans within the div.

There are two things to take into consideration when developing a selector engine: The results should be in document order (the order in which they've been defined) and the results should be unique (no duplicate elements returned). Because of these gotchas developing a top down engine can be quite tricky.

Take the following piece of markup, from Listing 10-5, into consideration, as if we were trying to implement our "div span" selector.

Listing 10-5: A simple top down selector engine.

```
<div>
  <div>
    <span>Span</span>
  </div>
</div>
<script>
window.onload = function(){
  function find(selector, root){
    root = root || document;

    var parts = selector.split(" "),
      query = parts[0],
      rest = parts.slice(1).join(" "),
      elems = root.getElementsByTagName( query ),
      results = [];

    for ( var i = 0; i < elems.length; i++ ) {
      if ( rest ) {
        results = results.concat( find(rest, elems[i]) );
      } else {
        results.push( elems[i] );
      }
    }

    return results;
```

```
  }

  var divs = find("div");
  assert( divs.length === 2, "Correct number of divs found." );

  var divs = find("div", document.body);
  assert( divs.length === 2,
  "Correct number of divs found in body." );

  var divs = find("body div");
  assert( divs.length === 2,
  "Correct number of divs found in body." );

  var spans = find("div span");
  assert( spans.length === 2, "A duplicate span was found." );
};
</script>
```

In the above listing we implement a simple top down selector engine (one that is only capable of finding elements by tag name). The engine breaks down into a few parts: Parsing the selector, finding the elements, filtering, and recursing and merging the results.

Parsing the Selector

In this simple example our parsing is reduced to converting a trivial CSS selector ("div span") into an array of strings (["div", "span"]). As introduced in CSS 2 and 3 it's possible to find elements by attribute or attribute value (thus it's possible to have additional spaces in most selectors - making our splitting of the selector too simplistic. For the most part this parsing ends up being "good enough" for now.

For a full implementation we would want to have a solid series of parsing rules to handle any expressions that may be thrown at us (most likely in the form of regular expressions).

Listing 10-6 has an example of a regular expression that's capable of capturing portions of a selector and breaking it in to pieces (splitting on commas, if need be):

Listing 10-6: A regular expression for breaking apart a CSS selector.

```
var selector = "div.class > span:not(:first-child) a[href]"
var chunker = /((?:\([^\)]+\)|\[[^\]]+\]|[^ ,\(\[]+)+)(\s*,\s*)?/g;
var parts = [];

// Reset the position of the chunker regexp (start from beginning)
chunker.lastIndex = 0;

// Collect the pieces
while ( (m = chunker.exec(selector)) !== null ) {
  parts.push( m[1] );

  // Stop if we've countered a comma
  if ( m[2] ) {
    extra = RegExp.rightContext;
    break;
  }
}
```

```
assert( parts.length == 4,
  "Our selector is broken into 4 unique parts." );
assert( parts[0] === "div.class", "div selector" );
assert( parts[1] === ">", "child selector" );
assert( parts[2] === "span:not(:first-child)", "span selector" );
assert( parts[3] === "a[href]", "a selector" );
```

Obviously this chunking selector is only one piece of the puzzle - you'll need to have additional parsing rules for each type of expression that you want to support. Most selector engines end up containing a map of regular expressions to functions - when a match is made on the selector portion the associated function is executed.

Finding the Elements

Finding the correct elements on the page is one piece of the puzzle that has many solutions. Which techniques are used depends a lot on which selectors are being support and what is made available by the browser. There are a number of obvious correlations, though.

getElementById: Only available on the root node of HTML documents, finds the first element on the page that has the specified ID (useful for the ID CSS selector "#id"). Internet Explorer and Opera will also find the first element on the page that has the same specified name. If you only wish to find elements by ID you will need an extra verification step to make sure that the correct result is being found.

If you wish to find all elements that match a specific ID (as is customary in CSS selectors - even though HTML documents are generally only permitted one specific ID per page) you will need to either: Traverse all elements looking for the ones that have the correct ID or use `document.all["id"]` which returns an array of all elements that match an ID in all browsers that support it (namely Internet Explorer, Opera, and Safari).

getElementsByTagName: Tackles the obvious result: Finding elements that match a specific tag name. It has a dual purpose, though - finding all elements within a document or element (using the "*" tag name). This is especially useful for handling attribute-based selectors that don't provide a specific tag name, for example: ".class" or "[attr]".

One caveat when finding elements comments using "*" - Internet Explorer will also return comment nodes in addition to element nodes (for whatever reason, in Internet Explorer, comment nodes have a tag name of "!" and are thusly returned). A basic level of filtering will need to be done to make sure that the correct nodes are matched.

getElementsByName: This is a well-implemented method that serves a single purpose: Finding all elements that have a specific name (such as for input elements that have a name). Thus it's really only useful for implementing a single selector: "[name=NAME]".

getElementsByClassName: A relatively new method that's being implemented by browsers (most prominently by Firefox 3 and Safari 3) that finds elements based upon the contents of their class attribute. This method proves to be a tremendous speed-up to class-selection code.

While there are a variety of techniques that can be used for selection the above methods are, generally, the primary tools used to what you're looking for on a page. Using the results from these methods it will be possible to

Filtering

A CSS expression is generally made up of a number of individual pieces. For example the expression "div.class[id]" has three parts: Finding all div elements that have a class name of "class" and have an attribute named "id".

The first step is to identify a root selector to being with. For example we can see that "div" is used - thus we can immediately use `getElementsByTagName` to retrieve all div elements on the page. We must, then, filter those results down to only include those that have the specified class and the specified id attribute.

This filtering process is a common feature of most selector implementations. The contents of these filters primarily deal with either attributes or the position of the element relative to its siblings.

Attribute Filtering: Accessing the DOM attribute (generally using the getAttribute method) and verifying their values. Class filtering (".class") is a subset of this behavior (accessing the className attribute and checking its value).

Position Filtering: For selectors like ":nth-child(even)" or ":last-child" a combination of methods are used on the parent element. In browser's that support it `.children` is used (IE, Safari, Opera, and Firefox 3.1) which contains a list of all child elements. All browsers have `.childNodes` (which contains a list of child nodes - including text nodes, comments, etc.). Using these two methods it becomes possible to do all forms of element position filtering.

Constructing a filtering function serves a dual purpose: You can provide it to the user as a simple method for testing their elements, quickly checking to see if an element matches a specific selector.

Recursing and Merging

As was shown in Listing 10-1, we can see how selector engines will require the ability to recurse (finding descendant elements) and merge the results together.

However our implementation is too simple, note that we end up receiving two spans in our results instead of just one. Because of this we need to introduce an additional check to make sure the returned array of elements only contains unique results. Most top down selector implementations include some method for enforcing this uniqueness.

Unfortunately there is no simple way to determine the uniqueness of a DOM element. We're forced to go through and assign temporary IDs to the elements so that we can verify if we've already encountered them, as in Listing 10-7.

Listing 10-7: Finding the unique elements in an array.

```
<div id="test">
  <b>Hello</b>, I'm a ninja!
</div>
<div id="test2"></div>
<script>
(function(){
  var run = 0;

  this.unique = function( array ) {
    var ret = [];

    run++;

    for ( var i = 0, length = array.length; i < length; i++ ) {
      var elem = array[ i ];

      if ( elem.uniqueID !== run ) {
        elem.uniqueID = run;
        ret.push( array[ i ] );
```

```
      }
    }

    return ret;
  };
})();

window.onload = function(){
  var divs = unique( document.getElementsByTagName("div") );
  assert( divs.length === 2, "No duplicates removed." );

  var body = unique( [document.body, document.body] );
  assert( body.length === 1, "body duplicate removed." );
};
</script>
```

This unique method adds an extra property to all the elements in the array - marking them as having been visited. By the time a complete run through is finished only unique elements will be left in the resulting array. Variations of this technique can be found in all libraries.

A longer discussion on the intricacies of attaching properties to DOM nodes see the chapter on Events.

Bottom Up Selector Engine

If you prefer not to have to think about uniquely identifying elements there is an alternative style of CSS selector engine that doesn't require its use.

A bottom up selector engine works in the opposite direction of a top down one. For example, given the selector "div span" it will first find all span elements then, for each element, navigate up the ancestor elements to find an ancestor div element. This style of selector engine construction matches the style found in most browser engines.

This engine style isn't as popular as the others. While it works well for simple selectors (and child selectors) the ancestor travels ends up being quite costly and doesn't scale very well. However the simplicity that this engine style provides can end up making for a nice trade-off.

The construction of the engine is simple. You start by finding the last expression in the CSS selector and retrieve the appropriate elements (just like with a top down engine, but the last expression rather than the first). From here on all operations are performed as a series of filter operations, removing elements as they go, like in Listing 10-8.

Listing 10-8: A simple bottom up selector engine.

```
<div>
  <div>
    <span>Span</span>
  </div>
</div>
<script>
window.onload = function(){
  function find(selector, root){
    root = root || document;

    var parts = selector.split(" "),
      query = parts[parts.length - 1],
```

```
      rest = parts.slice(0,-1).join("").toUpperCase(),
      elems = root.getElementsByTagName( query ),
      results = [];

    for ( var i = 0; i < elems.length; i++ ) {
      if ( rest ) {
        var parent = elems[i].parentNode;
        while ( parent && parent.nodeName != rest ) {
          parent = parent.parentNode;
        }

        if ( parent ) {
          results.push( elems[i] );
        }
      } else {
        results.push( elems[i] );
      }
    }

    return results;
  }

  var divs = find("div");
  assert( divs.length === 2, "Correct number of divs found." );

  var divs = find("div", document.body);
  assert( divs.length === 2,
  "Correct number of divs found in body." );

  var divs = find("body div");
  assert( divs.length === 2,
  "Correct number of divs found in body." );

  var spans = find("div span");
  assert( spans.length === 1, "No duplicate span was found." );
};
</script>
```

Listing 10-8 shows the construction of a simple bottom up selector engine. Note that it only works one ancestor level deep. In order to work more than one level deep the state of the current level would need to be tracked. This would result in two state arrays: The array of elements that are going to be returned (with some elements being set to undefined as they don't match the results) and an array of elements that correspond to the currently-tested ancestor element.

As mentioned before, this extra ancestor verification process does end up being slightly less scalable than the alternative top down method but it completely avoids having to utilize a unique method for producing the correct output, which some may see as an advantage.

Summary

JavaScript-based CSS selector engines are deceptively powerful tools. They give you the ability to easily locate virtually any DOM element on a page with a trivial amount of selector. While there are many nuances to actually implementing a full selector engine (and, certainly, no shortage of tools to help) the situation is rapidly improving.

With browsers working quickly to implement versions of the W3C Selector API having to worry about the finer points of selector implementation will soon be a thing of the past. For many developers that day cannot come soon enough.

Chapter 12. DOM Modification

In this chapter:

- Injecting HTML strings into a page
- Cloning elements
- Removing elements
- Manipulating element text

Next to traversing the DOM the most common operation required by most pieces of reusable JavaScript code is that of modifying DOM structures - in addition to modifying DOM node attributes or CSS properties (which we'll discuss later) - the brunt of the work falls back to injecting new nodes into a document, cloning nodes, and removing them again.

Injecting HTML

In this chapter we'll start by looking at an efficient way to insert an HTML string into a document at an arbitrary location. We're looking at this technique, in particular, since it's frequently used in a few ways: Injecting arbitrary HTML into a page, manipulating and inserting client-side templates, and retrieving and injecting HTML sent from a server. No matter the framework it'll have to deal with, at least, some of these problems.

On top of those points it's technically very challenging to implement correctly. Compare this to building an object-oriented style DOM construction API (which are certainly easier to implement but require an extra layer of abstraction from injecting the HTML).

There already exists an API for injecting arbitrary HTML strings - introduced by Internet Explorer (and in the process of being standardized in the W3C HTML 5 specification). It's a method that exists on all HTML DOM elements called `insertAdjacentHTML`. We'd spend more time looking at this API but there are a few problems.

1. It currently only exists in Internet Explorer (thus an alternative solution would have to be implemented anyway).
2. Internet Explorer's implementation is incredibly buggy (only working on a subset of all available elements).

For these reasons we're going to have to implement a clean API from scratch. An implementation is broken down into a couple steps:

1. Converting an arbitrary, valid, HTML/XHTML string into a DOM structure.
2. Injecting that DOM structure into an arbitrary location in the DOM as efficiently as possible.
3. Executing any inline scripts that were in the string.

All together, these three steps will provide a user with a smart API for injecting HTML into a document.

Converting HTML to DOM

Converting an HTML string to a DOM structure doesn't have a whole lot of magic to it - it uses the exact tool that you are already familiar with: `innerHTML`. It's a multi-step process:

- Make sure the HTML string contains valid HTML/XHTML (or, at least, tweak it so that it's closer to valid).
- Wrap the string in any enclosing markup required
- Insert the HTML string, using `innerHTML`, into a dummy DOM element.
- Extract the DOM nodes back out.

The steps aren't too complex - save for the actual insertion, which has some gotchas - but they can be easily smoothed over.

Pre-Process XML/HTML

To start, we'll need to clean up the HTML to meet user expectations. This first step will certainly depend upon the context, but within the construction of jQuery it became important to be able to support XML-style elements like "`<table/>`".

The above XML-style of elements, in actuality, only works for a small subset of HTML elements - attempting to use that syntax otherwise is able to cause problems in browsers like Internet Explorer.

We can do a quick pre-parse on the HTML string to convert elements like "`<table/>`" to "`<table>></table>`" (which will be handled uniformly in all browsers).

Listing 11-1: Make sure that XML-style HTML elements are interpreted correctly.

```
var tags = /^(abbr|br|col|img|input|link|meta|param|hr|area|embed)$/i;

function convert(html);
  return html.replace(/(<(\w+)[^>]*?)\/>/g, function(all, front, tag){
    return tags.test(tag) ?
      all :
      front + "></" + tag + ">";
  });
}

assert( convert("<a/>") === "<a></a>", "Check anchor conversion." );
assert( convert("<hr/>") === "<hr/>", "Check hr conversion." );
```

HTML Wrapping

We now have the start of an HTML string - but there's another step that we need to take before injecting it into the page. A number of HTML elements must be within a certain container element before they must be injected. For example an option element must be within a select. There are two options to solve this problem (both require constructing a map between problematic elements and their containers).

- The string could be injected directly into a specific parent, previously constructed using `createElement`, using `innerHTML`. While this may work in some cases, in some browsers, it is not universally guaranteed to work.
- The string could be wrapped with the appropriate markup required and then injected directly into any container element (such as a div).

The second technique is the preferred one. It involves very little browser-specific code in contrast with the other one, which would be mostly browser-specific code.

The set of elements that need to be wrapped is rather manageable (with 7 different groupings occurring).

- option and optgroup need to be contained in a `<select multiple="multiple">...</select>`
- legend need to be contained in a `<fieldset>...</fieldset>`
- thead, tbody, tfoot, colgroup, and caption need to be contained in a `<table>...</table>`
- tr need to be in a `<table><thead>...</thead></table>`, `<table><tbody>...</tbody></table>`, or a `<table><tfoot>...</tfoot></table>`
- td and th need to be in a `<table><tbody><tr>...</tr></tbody></table>`
- col in a `<table><tbody></tbody><colgroup>...</colgroup></table>`
- link and script need to be in a `div<div>...</div>`

Nearly all of the above are self-explanatory save for the multiple select, the col, and the link and script ones. A multiple select is used (instead of a regular select) because it won't automatically check any of the options that are placed inside of it (whereas a single select will auto-check the first option). The col fix includes an extra tbody, without which the colgroup won't be generated properly. The link and script fix is a weird one: Internet Explorer is unable to generate link and script elements, via innerHTML, unless they are both contained within another element and there's an adjacent node.

Generating the DOM

Using the above map of characters we not have enough information to generate the HTML that we need to insert in to a DOM element.

Listing 11-2: Generate a list of DOM nodes from some markup.

```
function getNodes(htmlString){
  var map = {
    "<td": [3, "<table><tbody><tr>", "</tr></tbody></table>"],
    "<option": [1, "<select multiple='multiple'>", "</select>"]
    // a full list of all element fixes
  };

  var name = htmlString.match(/<\w+/),
    node = name ? map[ name[0] ] || [0, "", ""];

  var div = document.createElement("div");
  div.innerHTML = node[1] + htmlString + node[2];

  while ( node[0]-- )
    div = div.lastChild;

  return div.childNodes;
}

assert( getNodes("<td>test</td><td>test2</td>").length === 2,
  "Get two nodes back from the method." );
assert( getNodes("<td>test</td>").nodeName === "TD",
  "Verify that we're getting the right node." );
```

There are two bugs that we'll need to take care of before we return our node set, though - and both are bugs in Internet Explorer. The first is that Internet Explorer adds a tbody element inside an empty table

(checking to see if an empty table was intended and removing any child nodes is a sufficient fix). Second is that Internet Explorer trims all leading whitespace from the string passed to `innerHTML`. This can be remidied by checking to see if the first generated node is a text and contains leading whitespace, if not create a new text node and fill it with the whitespace explicitly.

After all of this we now have a set of DOM nodes that we can begin to insert into the document.

Inserting into the Document

Once you have the actual DOM nodes it becomes time to insert them into the document. There are a couple steps the need to take place - none of which are particularly tricky.

Since we have an array of elements that we need to insert and, potentially, into any number of locations into the document, we'll need to try and cut down on the number of operations that need to occur.

We can do this by using DOM Fragments. Fragments are part of the W3C DOM specification and are supported in all browsers. They give you a container that you can use to hold a collection of DOM nodes. This, in itself, is useful but it also has two extra advantages: The fragment can be injected and cloned in a single operation instead of having to inject and clone each individual node over and over again. This has the potential to dramatically reduce the number of operations required for a page.

In the following example, derived from the code in jQuery, a fragment is created and passed in to the clean function (which converts the incoming HTML string into a DOM). This DOM is automatically appended on to the fragment.

Listing 11-3: Inserting a DOM fragment into multiple locations in the DOM, using the code from Listing 11-1 and Listing 11-4.

```
<div id="test"><b>Hello</b>, I'm a ninja!</div>
<div id="test2"></div>
<script>
window.onload = function(){
  function insert(elems, args, callback){
    if ( elems.length ) {
      var doc = elems[0].ownerDocument || elems[0],
        fragment = doc.createDocumentFragment(),
        scripts = getNodes( args, doc, fragment ),
        first = fragment.firstChild;

      if ( first ) {
        for ( var i = 0; elems[i]; i++ ) {
          callback.call( root(elems[i], first),
            i > 0 ? fragment.cloneNode(true) : fragment );
        }
      }
    }
  }

  var divs = document.getElementsByTagName("div");

  insert(divs, ["<b>Name:</b>"], function(fragment){
    this.appendChild( fragment );
  });

  insert(divs, ["<span>First</span> <span>Last</span>"],
```

```
    function(fragment){
      this.parentNode.insertBefore( fragment, this );
    });
};
</script>
```

There's another important point here: If we're inserting this element into more than one location in the document we're going to need to clone this fragment again and again (if we weren't using a fragment we'd have to clone each individual node every time, instead of the whole fragment at once).

One final point that we'll need to take care of, albeit a relatively minor one. When users attempt to inject a table row directly into a table element they normally mean to insert the row directly in to the tbody that's in the table. We can write a simple mapping function to take care of that for us.

Listing 11-4: Figure out the actual insertion point of an element.

```
function root( elem, cur ) {
  return elem.nodeName.toLowerCase() === "table" &&
    cur.nodeName.toLowerCase() === "tr" ?
    (elem.getElementsByTagName("tbody")[0] ||
      elem.appendChild(elem.ownerDocument.createElement("tbody"))) :
    elem;
}
```

Altogether we now have a way to both generate and insert arbitrary DOM elements in an intuitive manner.

Script Execution

Aside from the actual insertion of HTML into a document a common requirement is the execution of inline script elements. This is mostly used when a piece of HTML is coming back from a server and there's script that needs to be executed along with the HTML itself.

Usually the best way to handle inline scripts is to strip them out of the DOM structure before they're actually inserted into the document. In the function that's used to convert the HTML into a DOM node the end result would look something like the following code from jQuery.

Listing 11-5: Collecting the scripts from a piece an array of

```
for ( var i = 0; ret[i]; i++ ) {
  if ( jQuery.nodeName( ret[i], "script" ) &&
      (!ret[i].type ||
        ret[i].type.toLowerCase() === "text/javascript") ) {
    scripts.push( ret[i].parentNode ?
      ret[i].parentNode.removeChild( ret[i] ) :
      ret[i] );
  } else if ( ret[i].nodeType === 1 ) {
    ret.splice.apply( ret, [i + 1, 0].concat(
      jQuery.makeArray(ret[i].getElementsByTagName("script"))) );
  }
}
```

The above code is dealing with two arrays: ret (which holds all the DOM nodes that have been generated) and scripts (which becomes populated with all the scripts in this fragment, in document order). Additionally, it takes care to only remove scripts that are normally executed as JavaScript (those with no type or those with a type of 'text/javascript').

Then after the DOM structure is inserted into the document take the contents of the scripts and evaluate them. None of this is particularly difficult - just some extra code to shuffle around.

But it does lead us to the tricky part:

Global Code Evaluation

When users are including inline scripts to be executed, they're expecting them to be evaluated within the global context. This means that if a variable is defined it should become a global variable (same with functions, etc.).

The built-in methods for code evaluation are spotty, at best. The one fool-proof way to execute code in the global scope, across all browsers, is to create a fresh script element, inject the code you wish to execute inside the script, and then quickly inject and remove the script from the document. This will cause the browser to execute the inner contents of the script element within the global scope. This technique was pioneered by Andrea Giammarchi and has ended up working quite well. Below is a part of the global evaluation code that's in jQuery.

Listing 11-6: Evaluate a script within the global scope.

```
function globalEval( data ) {
  data = data.replace(/^\s+|\s+$/g, "");

  if ( data ) {
    var head = document.getElementsByTagName("head")[0] ||
        document.documentElement,
      script = document.createElement("script");

    script.type = "text/javascript";
    script.text = data;

    head.insertBefore( script, head.firstChild );
    head.removeChild( script );
  }
}
```

Using this method it becomes easy to rig up a generic way to evaluate a script element. We can even add in some simple code for dynamically loading in a script (if it references an external URL) and evaluate that as well.

Listing 11-7: A method for evaluating a script (even if it's remotely located).

```
function evalScript( elem ) {
  if ( elem.src )
    jQuery.ajax({
      url: elem.src,
      async: false,
      dataType: "script"
    });

  else
    jQuery.globalEval( elem.text || "" );

  if ( elem.parentNode )
    elem.parentNode.removeChild( elem );
}
```

Note that after we're done evaluating the script we remove it from the DOM. We did the same thing earlier when we removed the script element before it was injected into the document. We do this so that scripts won't be accidentally double-executed (appending a script to a document, which ends up recursively calling itself, for example).

Cloning Elements

Cloning an element (using the DOM `cloneNode` method) is straightforward in all browsers, except Internet Explorer. Internet Explorer has three behaviors that, when they occur in conjunction, result in a very frustrating scenario for handling cloning.

First, when cloning an element, Internet Explorer copies over all event handlers on to the cloned element. Additionally, any custom expandos attached to the element are also carried over.

In jQuery a simple test to determine if this is the case.

Listing 11-8: Determining if a browser copies event handlers on clone.

```
<script>
var div = document.createElement("div");

if ( div.attachEvent && div.fireEvent ) {
  div.attachEvent("onclick", function(){
    // Cloning a node shouldn't copy over any
    // bound event handlers (IE does this)
    jQuery.support.noCloneEvent = false;
    div.detachEvent("onclick", arguments.callee);
  });
  div.cloneNode(true).fireEvent("onclick");
}
</script>
```

Second, the obvious step to prevent this would be to remove the event handler from the cloned element - but in Internet Explorer, if you remove an event handler from a cloned element it gets removed from the original element as well. Fun stuff. Naturally, any attempts to remove custom expando properties on the clone will cause them to be removed on the original cloned element, as well.

Third, the solution to all of this is to just clone the element, inject it into another element, then read the `innerHTML` of the element - and convert that back into a DOM node. It's a multi-step process but one that'll result in an untainted cloned element. Except, there's another IE bug: The `innerHTML` (or `outerHTML`, for that matter) of an element doesn't always reflect the correct state of an element's attributes. One common place where this is realized is when the name attributes of input elements are changed dynamically - the new value isn't represented in the `innerHTML`.

This solution has another caveat: `innerHTML` doesn't exist on XML DOM elements, so we're forced to go with the traditional `cloneNode` call (thankfully, though, event listeners on XML DOM elements are pretty rare).

The final solution for Internet Explorer ends up becoming quite circuitous. Instead of a quick call to `cloneNode` it is, instead, serialized by `innerHTML`, extracted again as a DOM node, and then monkey-patched for any particular attributes that didn't carry over. How much monkeying you want to do with the attributes is really up to you.

Listing 11-9: A portion of the element clone code from jQuery.

```
function clone() {
  var ret = this.map(function(){
    if ( !jQuery.support.noCloneEvent && !jQuery.isXMLDoc(this) ) {
      var clone = this.cloneNode(true),
        container = document.createElement("div");
      container.appendChild(clone);
      return jQuery.clean([container.innerHTML])[0];
    } else
      return this.cloneNode(true);
  });

  var clone = ret.find("*").andSelf().each(function(){
    if ( this[ expando ] !== undefined )
      this[ expando ] = null;
  });

  return ret;
}
```

Note that the above code uses jQuery's `jQuery.clean` method which converts an HTML string into a DOM structure (which was discussed previously).

Removing Elements

Removing an element from the DOM should be simple (a quick call to removeChild) - but of course it isn't. We have to do a lot of preliminary cleaning up before we can actually remove an element from the DOM.

There's usually two steps of cleaning that need to occur on an DOM element before it can be removed from the DOM. Both of them relate to events, so we'll be discussing this in more detail when we cover it in the Events chapter.

The first thing to clean up are any bound event handlers from the element. If a framework is designed well it should only be binding a single handler for an element at a time so the cleanup shouldn't be any harder than just removing that one function. This step is important because Internet Explorer will leak memory should the function reference an external DOM element.

The second point of cleanup is removing any external data associated with the element. We'll discuss this more in the Events chapter but a framework needs a good way to associate pieces of data with an element (especially without directly attaching the data as an expando property). It is a good idea to clean up this data simply so that it doesn't consume any more memory.

Now both of these points need to be done on the element that is being removed - and on all descendant elements, as well (since all the descendant elements are also being removed - just in a less-obvious way).

For example, here's the relevant code from jQuery:

Listing 11-10: The remove element function from jQuery.

```
function remove() {
  // Go through all descendants and the element to be removed
  jQuery( "*", this ).add([this]).each(function(){
    // Remove all bound events
    jQuery.event.remove(this);
```

```
    // Remove attached data
    jQuery.removeData(this);
  });

  // Remove the element (if it's in the DOM)
  if ( this.parentNode )
    this.parentNode.removeChild( this );
}
```

The second part to consider, after all the cleaning up, is the actual removal of the element from the DOM. Most browsers are perfectly fine with the actual removal of the element from the page -- except for Internet Explorer. Every single element removed from the page fails to reclaim some portion of its used memory, until the page is finally left. This means that long-running pages, that remove a lot of elements from the page, will find themselves using considerably more memory in Internet Explorer as time goes on.

There's one partial solution that seems to work quite well. Internet Explorer has a proprietary property called `outerHTML`. This property will give you an HTML string representation of an element. For whatever reason `outerHTML` is also a setter, in addition to a getter. As it turns out, if you set `outerHTML = ""` it will wipe out the element from Internet Explorer's memory more-completely than simply doing `removeChild`. This step is done in addition to the normal `removeChild` call .

Listing 11-11: Set outerHTML in an attempt to reclaim more memory in Internet Explorer.

```
// Remove the element (if it's in the DOM)
if ( this.parentNode )
  this.parentNode.removeChild( this );

if ( typeof this.outerHTML !== "undefined" )
  this.outerHTML = "";
```

It should be noted that it isn't successful in reclaiming all of the memory that was used by the element, but it absolutely reclaims more of it (which is a start, at least).

It's important to remember that any time an element is removed from the page that you should go through the above three steps - at the very least. This includes emptying out the contents of an element, replacing the contents of an element (with either HTML or text), or replacing an element directly. Remember to always keep your DOM tidy and you won't have to work so much about memory issues later on.

Text Contents

Working with text tends to be much easier than working with HTML elements, especially since there are built-in methods, that work in all browsers, for handling this behavior. Of course, there are all sorts of bugs that we end up having to work around, making these APIs obsolete in the process.

There are typically two desired scenarios: Getting the text contents out of an element and setting the text contents of an element.

W3C-compliant browsers provide a `.textContent` property on their DOM elements. Accessing the contents of this property gives you the textual contents of the element (both its direct children and descendant nodes, as well).

Internet Explorer has its own property, `.innerText`, for performing the exact-same behavior as `.textContent`.

Listing 11-12: Using textContent and innerText.

```
<div id="test"><b>Hello</b>, I'm a ninja!</div>
<div id="test2"></div>
<script>
window.onload = function(){
  var b = document.getElementById("test");
  var text = b.textContent || b.innerText;

  assert( text === "Hello, I'm a ninja!",
  "Examine the text contents of an element." );
  assert( b.childNodes.length === 2,
  "An element and a text node exist." );

  if ( typeof b.textContent !== "undefined" ) {
    b.textContent = "Some new text";
  } else {
    b.innerText = "Some new text";
  }

  text = b.textContent || b.innerText;

  assert( text === "Some new text", "Set a new text value." );
  assert( b.childNodes.length === 1,
  "Only one text nodes exists now." );
};
</script>
```

Note that when we set the `textContent`/`innerText` properties the original element structure is removed. So while both of these properties are very useful there are a certain number of gotchas. First, as when we discussed removing elements from the page, not having an sort of special consideration for element memory leaks will come back to bite you. Additionally, the cross-browser handling of whitespace is absolutely abysmal in these properties. No browser appears capable of returning a consistent result.

Thus, if you don't care about preserving whitespace (especially endlines) feel free to use textContent/ innerText for accessing the element's text value. For setting, though, we'll need to devise an alternative solution.

Setting Text

Setting a text value comes in two parts: Emptying out the contents of the element and inserting the new text contents in its place. Emptying out the contents is straightforward - we've already devised a solution in Listing 11-10.

To insert the new text contents we'll need to use a method that'll properly escape the string that we're about to insert. An important difference between inserting HTML and inserting text is that the inserted text will have any problematic HTML-specific characters escaped. For example '<' will appear as <

We can actually use the built-in `createTextNode` method, that's available on DOM documents, to perform precisely that.

Listing 11-13: Setting the text contents of an element.

```
<div id="test"><b>Hello</b>, I'm a ninja!</div>
<div id="test2"></div>
<script>
window.onload = function(){
```

```
  var b = document.getElementById("test");

  // Replace with your empty() method of choice
  while ( b.firstChild )
    b.removeChild( b.firstChild );

  // Inject the escaped text node
  b.appendChild( document.createTextNode( "Some new text" ) );

  var text = b.textContent || b.innerText;

  assert( text === "Some new text", "Set a new text value." );
  assert( b.childNodes.length === 1,
  "Only one text nodes exists now." );
};
</script>
```

Getting Text

To get an accurate text value of an element we have to ignore the results from textContent and innerText. The most common problem is related to endlines being unnecessarily stripped from the return result. Instead we must collect all the text node values manually to get an accurate result.

A possible solution would look like this, making good use of recursion:

Listing 11-14: Getting the text contents of an element.

```
<div id="test"><b>Hello</b>, I'm a ninja!</div>
<div id="test2"></div>
<script>
window.onload = function(){
  function getText( elem ) {
    var text = "";

    for ( var i = 0, l = elem.childNodes.length; i < l; i++ ) {
      var cur = elem.childNodes[i];

      // A text node has a nodeType === 3
      if ( cur.nodeType === 3 )
        text += cur.nodeValue;

      // If it's an element we need to recurse further
      else if ( cur.nodeType === 1 )
        text += getText( cur );
    }

    return text;
  }

  var b = document.getElementById("test");
  var text = getText( b );

  assert( text === "Hello, I'm a ninja!",
  "Examine the text contents of an element." );
  assert( b.childNodes.length === 2,
```

```
  "An element and a text node exist." );
};
</script>
```

In your applications if you can get away with not worrying about whitespace definitely stick with `textContent`/`innerText` as it'll make your life so much simpler.

Summary

We've taken a comprehensive look at the best ways to tackle the difficult problems surrounding DOM manipulation. While these problems should be easier than they are - the cross-browser issues introduced make their actual implementations much more difficult. With a little bit of extra work we can have a unified solution that will work well in all major browsers - which is exactly what we should strive for.

Chapter 13. Attributes and CSS

In this chapter:

- blah

DOM Attributes

Attributes vs. DOM properties

Attribute names (cssText, className, etc.)

URL resolution (IE) .getAttribute("...", 2);

ID override in IE/Opera

Form Values

CSS

Style property names

Computed Style

Height and Width

Elements

document

window

Opacity

Summary

Chapter 14. Events

In this chapter:

- blah

'this'

Event Propagation

Keyboard Events

Blocking arrow keys

Mouse Events

Cursor position position relative to element

Custom Events

Triggering

Delegation

... throttling?

Summary

Chapter 15. Ajax

In this chapter:

- blah

XMLHttpRequest

IE7 local file

File Loading

XML JavaScript (eval in global scope) JSON (eval) XML

Caching

Cross-Domain Requests

JSONP getScript

Summary

Chapter 16. Animation

In this chapter:

- blah

Tweening

Value interpolation

Animatable Values (height, width, top, left, etc.)

Smooth Animations

Stop/Pause

Summary

Chapter 17. Performance

In this chapter:

- blah

Performance Test Suite

Firebug Profiling

Techniques

Looping Variable caching

File Distribution

JSMin YUIMin Packer GZip

Summary