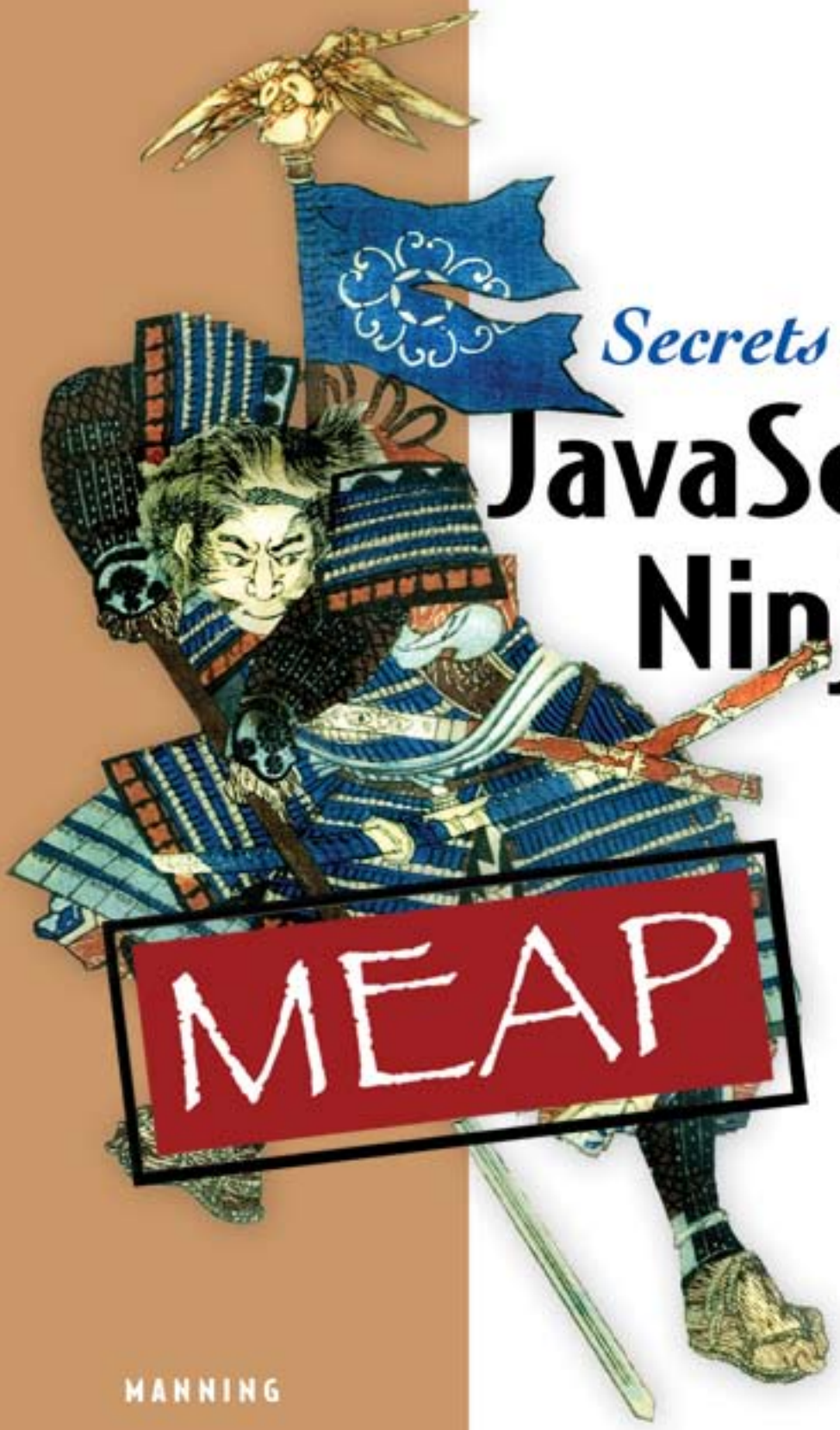

Secrets of the JavaScript Ninja

MEAP

John Resig
Bear Bibeault

MANNING

MEAP Edition
Manning Early Access Program
Secrets of the JavaScript Ninja version 7

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

Table of Contents

1 Enter the ninja

In this chapter:

- A look at the purpose and structure of this book
- Which libraries we will focus upon
- What is advanced JavaScript programming?
- Cross-browser authoring
- Test suite examples

If you are reading this book, you know that there is nothing simple about creating effective and cross-browser JavaScript code. In addition to the normal challenges of writing clean code, we have the added complexity of dealing with obtuse browser differences and complexities. To deal with these challenges, JavaScript developers frequently capture sets of common and reusable functionality in the form of JavaScript libraries. These libraries vary widely in approach, content and complexity, but one constant remains: they need to be easy to use, incur the least amount of overhead, and be able to work across all browsers that we wish to target.

It stands to reason then, that understanding how the very best JavaScript libraries are constructed can provide us with great insight into how your own code can be constructed to achieve these same goals. This book sets out to uncover the techniques and secrets used by these world-class code bases, and to gather them into a single resource.

In this book we'll be examining the techniques that are used to create two of the more popular JavaScript libraries. Let's meet them!

1.1 Our key JavaScript libraries

The techniques and practices used to create two modern JavaScript libraries will be the focus of our particular attention in this book. They are:

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

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 and feedback these libraries been refined over the years into the optimal code bases that they are today. In addition to detailed examination of 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! and released to the public in February of 2006. Yahoo! UI 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. This library supports 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 target problem areas comprehensively. For these reasons they'll serve as a good basis for further analysis, and understanding the fundamental construction of these code bases gives us insight into the process of world-class JavaScript library construction.

But these techniques won't only be useful for constructing large libraries, but can be applied to all JavaScript coding, regardless of size.

The make up of a JavaScript library can be broken down into three aspects: advanced use of the JavaScript language, meticulous construction of cross-browser code, and a series of best practices that tie everything together. We'll be carefully analyzing these three aspects to give us a complete knowledge base with which we can create our own effective JavaScript code.

1.2 Understanding the JavaScript Language

Many JavaScript coders, as they advance through their careers, may get to the point at which they're actively using the vast array of elements comprising the language: including objects and functions, and, if they've been paying attention to coding trends, even anonymous inline functions, throughout their code. In many cases, however, those skills may not be taken beyond fundamental skill levels. Additionally there is generally a very poor understanding of the purpose and implementation of ***closures*** in JavaScript, which fundamentally and irrevocably binds the importance of functions to the language.

JavaScript consists of a close relationship between objects, functions – which in JavaScript are first class elements – and closures. Understanding the strong relationship between these three concepts vastly improves our JavaScript programming ability, giving us a strong foundation for any type of application development.

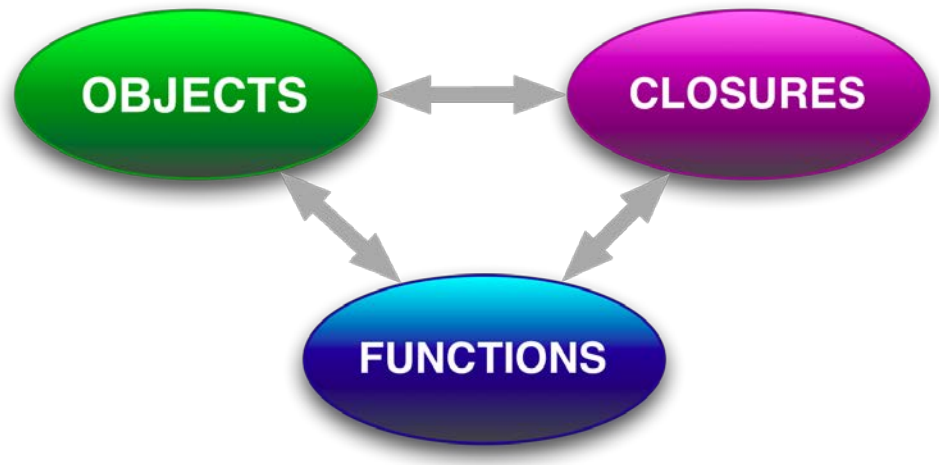

Figure 1.1 JavaScript consists of a close relationship between objects, functions and closures

Many JavaScript developers, especially those coming from an object-oriented background, may pay a lot of attention to objects, but at the expense of understanding how functions and closures contribute to the big picture.

In addition to these fundamental concepts, there are two features in JavaScript that are woefully underused: timers and regular expressions. These two concepts have applications in virtually any JavaScript code base, but aren't always used to their full potential due to their misunderstood nature.

A firm grasp of how timers operate within the browser, all too frequently a mystery, gives us the ability to tackle complex coding tasks such as long-running computations and smooth animations. And an advanced of how regular expressions work allows us to simplify what would otherwise be quite complicated pieces of code.

As another high point of our advanced tour of the JavaScript language, we'll take a look at the `with` statement later on in chapter 8, and the crucially important `eval()` method in chapter 9

All too often these two important language features are trivialized, misused, and even condemned outright by many JavaScript programmers. But by looking at the work of some of the best JavaScript coders we can see that, when used appropriately, these useful features allow for the creation of some fantastic pieces of code that wouldn't be otherwise

possible. To a large degree they can also be used for some interesting meta-programming exercises, molding JavaScript into whatever we want it to be.

Learning how to use these features responsibly and to their best advantage can certainly elevate your code to higher levels.

Honing our skills to tie these concepts and features together gives us a level of understanding that puts the creation of any type of JavaScript application within our reach, and gives us a solid base for moving forward starting with writing solid, cross-browser code.

1.3 Cross-browser considerations

Perfecting our JavaScript programming skills will get us far, but when developing browser-based JavaScript applications sooner, rather than later, we're going to run face first into *The Browsers* and their maddening issues and inconsistencies.

In a perfect world, all browsers would be bug-free and support Web Standards in a consistent fashion, but we all know that we most certainly do not live in that world.

The quality of browsers has improved greatly as of late, but it's a given that they all still have some bugs, missing APIs, and specific quirks that we'll need to deal with. Developing a comprehensive strategy for tackling these browser issues, and becoming intimately familiar with their differences and quirks, is just as important, if not more so, than proficiency in JavaScript itself.

When writing browser applications, or JavaScript libraries to be used in them, picking and choosing which browsers to support is an important consideration. We'd like to support them all, but development and testing resources dictates otherwise. So how do we decide which to support, and to what level?

Throughout this book, an approach that we will employ is one that we'll borrow from Yahoo! that they call ***Graded Browser Support***.

This technique, in which the level of browser support is graded as one of A, C, or X, is described at http://developer.yahoo.com/yui/articles/gbs, and defines the three level of support as:

- **A**: Modern browsers that take advantage of the power capabilities of web standards. These browsers get full support with advanced functionality and visuals.
- **C**: Older browsers that are either outdated or hardly used. These browsers receive minimal support; usually limited to HTML and CSS with no scripting, and bare-bones visuals.
- **X**: Unknown or fringe browsers. Somewhat counter-intuitively, rather than being unsupported, these browsers are given the benefit of the doubt, and assumed to be as capable as A-grade browsers. Once the level of capability can be concretely ascertained, these browsers can be assigned to A or C grade.

As of early 2011, the Yahoo! Graded Browser Support matrix was as shown in Table 1.1. Any ungraded browser/platform combination, or unlisted browser, is assigned a grade of X.

Table 1.1 This early 2011 Graded Browser Support matrix shows the level of browser support from Yahoo!

	Windows XP	Windows 7	Mac OS 10.6	iOS 3	iOS 4	Android 2.2
IE <6	C					
IE 6, 7, 8	A					
Firefox <3	C					
Firefox 3	A	A	A			
Firefox 4		A	A			
Safari < 5			C			
Safari 5+			A			
Safari for iOS				A	A	
Chrome	A					
WebKit for Android						A
Opera <9.5	C					
Netscape <8	C					

Note that this is the support chart as defined by Yahoo! for YUI 2 and YUI 3 – it is not a recommendation on what we, in this book, or you, in your own projects, should support. But by using this approach to come up with our own support matrix, we can determine the balance between coverage and pragmatism to come up with the optimal set of browsers and platforms that we should support.

It's impractical to develop against a large number of platform/browser combinations, so we must weigh the cost versus benefit of supporting the various browsers, and create our own resulting support matrix.

This analysis must take in account multiple considerations, the primary of which are:

- The market share of the browser
- The amount of effort necessary to support the browser

Figure 1.2 shows a sample chart that represents your authors' personal choices when developing for some browsers (not all browsers included for brevity) based upon March 2011 market share:

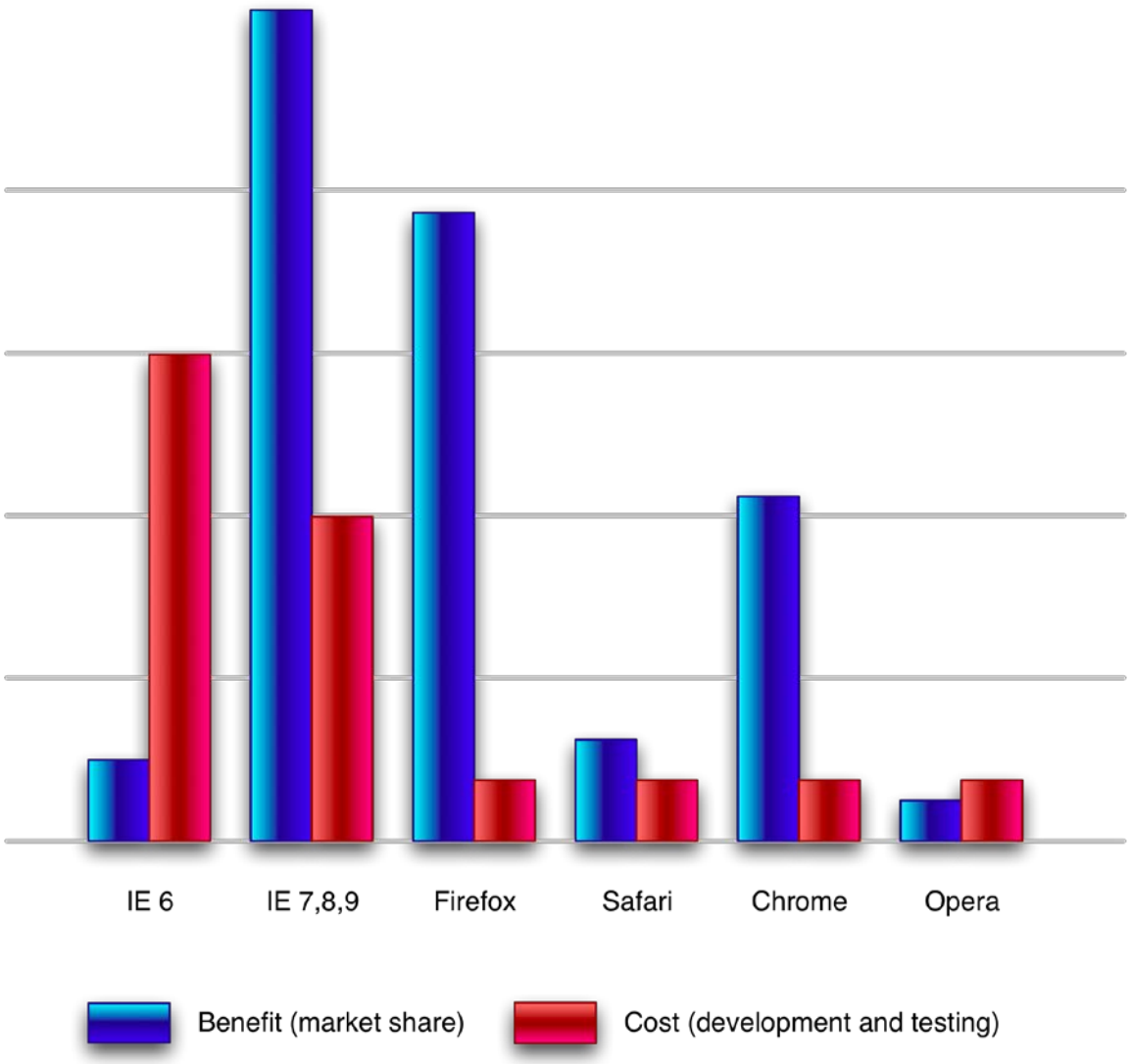

Figure 1.2 Analyzing the cost versus benefit of supporting various browsers tells us where to put our effort

Charting the benefit versus cost in this manner shows us at a glance where we should put our effort to get the most "bang for the buck". Things that jump out of this chart:

- Even though it's relatively a lot more effort to support Internet Explorer 7 and later than the standards-compliant browsers, it's large market share makes the extra effort worthwhile.
- Supporting Firefox and Chrome is a no-brainer since that have large market share and are easy to support.
- Even though Safari has a relatively low market share, it still deserves support, as its standard-compliant nature makes its cost small.
- Opera, though not much more effort than Safari, loses out because of its minuscule market share.
- Nothing really need be said about IE 6.

Of course, nothing is ever quite so cut-and-dried. It might be safe to say that benefit is more important than cost; it ultimately comes down to the choices of those in the decision-making process, taking into account factors such as the skill of the developers, the needs of the market, and other business concerns. But quantifying the costs versus benefits is a good starting point for making these important support decisions.

Minimizing the cost of cross-browser development is significantly affected by the skill and experience of the developers, and this book is intended to boost your skill level, so let's get to it by looking at best practices as a start.

1.4 Best practices

Mastery of the JavaScript language and a grasp of cross-browser coding issues are important parts of becoming an expert web application developer, but they're not the complete picture. To enter the Big Leagues you also need to exhibit the traits that scores of previous developers have proved are beneficial to the development of quality code. These traits, which we will examine in depth in chapter 2, are known as ***best practices*** and, in addition to mastery of the language, include such elements as:

- Testing
- Performance analysis
- Debugging skills

It is vitally important to adhere to these practices in our coding, *and* frequently. The complexity of cross-browser development certainly justifies it. Let's examine a couple of these practices.

1.4.1 Best practice: testing

Throughout this book, we'll be applying a number of testing techniques that serve to ensure that our example code operates as intended, as well as to serve as examples of how to test general code. The primary tool that we will be using for testing is an `assert()` function, whose purpose is to assert that a premise is either true or false. The general form of this function is:

```
assert(condition,message);
```

where the first parameter is a condition that should be true, and the second is a message that will be raised if it is not.

Consider, for example:

```
assert(a == 1, "Disaster! A is not 1!");
```

If the value of variable a is not equal to one, the assertion fails and the somewhat overly-dramatic) message is raised.

Note that the `assert()` function is not an innate feature of the language (as it is in some other language, such as Java), so we'll be implementing it ourselves. We'll be discussing its implementation and use in chapter 2.

1.4.2 Best practice: performance analysis

Another important practice is performance analysis. The JavaScript engine in the browsers have been making astounding strides in the performance of JavaScript itself, but that's no excuse for us to write sloppy and inefficient code. Another function we'll be implementing and using in this book is the `perf()` function for collecting performance information.

An example of its use would be:

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

We'll examine the implementation and use of `perf()` in chapter 2.

These best-practice techniques, along with the others that we'll learn along the way, will greatly enhance our JavaScript development. Developing applications with the restricted resources that a browser provides, coupled with the increasingly complex world of browser capability and compatibility, makes having a robust and complete set of skills a necessity.

1.5 Summary

Cross-browser web application development is hard; harder than most people would think.

In order to pull it off, we need not only a mastery of the JavaScript language, but a thorough knowledge of the browsers, along with their quirks and inconsistencies, and a good grounding in accepted best practices.

While JavaScript development can certainly be challenging, there are those brave souls who have already gone down this torturous route: the developers of JavaScript libraries. We'll be distilling the knowledge gained during the construction of these code bases, effectively fueling our development skills, and raising them to world class level.

This exploration will certainly be informative and educational – let's enjoy the ride!

2

Testing and debugging

In this chapter:

- Tools for Debugging JavaScript code
- Techniques for generating tests
- Building a test suite
- How to test asynchronous operations

Constructing effective test suites for our code is always important, so we're actually going to discuss it now, before we go into any discussions on coding. As important as a solid testing strategy is for *all* code, it can be crucial for situations where external factors have the potential to affect the operation of our code; which is *exactly* the case we are faced with in cross-browser JavaScript development.

Not only do we have the typical problems of ensuring the quality of the code, especially when dealing with multiple developers working on a single code base, and guarding against regressions that could break portions of an API (generic problems that all programmers need to deal with), but we also have the problem of determining if our code works in all the browsers that we choose to support.

We'll further discuss the problem of cross-browser development in-depth when we look at cross-browser strategies in chapter 10, but for now, it's vital that the importance of testing be emphasized and testing strategies defined, as we'll be using these strategies 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 those tests.

Let's get started.

2.1 Debugging Code

Remember when debugging JavaScript meant using `alert()` to verify the value of variables? Well, the ability to debug JavaScript code has dramatically improved in the last few years, in no small part due to the popularity of the Firebug developer extension for Firefox.

Similar tools have been developed for all major browsers:

- **Firebug**: The popular developer extension for Firefox that got the ball rolling. See http://getfirebug.org/
- **IE Developer Tools**: Included in Internet Explorer 8 and 9.
- **Opera Dragonfly**: Included in Opera 9.5 and newer. Also works with Mobile versions of Opera.
- **WebKit Developer Tools**: Introduced in Safari 3, dramatically improved in Safari 4, and now in Chrome.

There are two important approaches to debugging JavaScript: logging and breakpoints. They are both useful for answering the important question “What’s going on in my code?”, but each tackling it from a different angle.

Let’s start by looking at logging.

2.1.1 Logging

Logging statements (such as using the `console.log()` method in Firebug, Safari, Chrome and IE) are part of the code (even if perhaps temporarily) and useful in a cross-browser sense. We can write logging calls in our code, and we can benefit from seeing the messages in the console of all modern browsers (with the exception of Opera).

These browser consoles have dramatically improved the logging process over the old 'add an alert' technique. All our logging statements can be written to the console and be browsed immediately or at a later time without impeding the normal flow of the program – something not possible with `alert()`.

For example, if we wanted to know what the value of a variable named `x` was at a certain point in the code, we might write:

```
console.log(x);
```

If we were to assume that the value of `x` is 213, then the result of executing this statement in the Chrome browser with the JavaScript console enabled would appear as shown in figure 2.1.

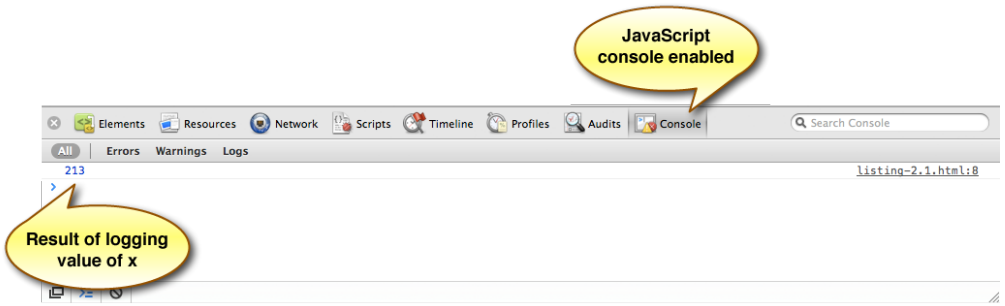

Figure 2.1 Logging lets us see the state of things in our code as it is running

Because Opera chose to go its own way when it comes to logging, implementing a proprietary `postError()` method, we'll get all suave and implement a higher-level logging method that works across all modern browsers as shown in Listing 2.1.

Listing 2.1: A simple logging method that works in all modern browsers

```
function log() {
 try {
  console.log.apply(console, arguments);                    #1
 }
 catch(e) {                                                 #2
  try {
   opera.postError.apply(opera, arguments);                 #3
  }
  catch(e){
   alert(Array.prototype.join.call( arguments, " "));       #4
  }
 }
}
```

#1 Tries to log using most common method
#2 Catches failure
#3 Tries to log the Opera way
#4 Uses an alert if all else fails

In this method, we first try to log a message using the method that works in most modern browsers (#1). If that fails, an exception will be thrown that we catch (#2), and then try to log a message using Opera's proprietary method (#3). If both of those methods fail, we fall back to using old-fashioned alerts (#4).

NOTE Within our method we used the `apply()` and `call()` methods of the JavaScript Function to relay the arguments passed to *our* function to the logging function. These Function methods are designed to help us make precisely controlled calls to JavaScript functions and we'll be seeing much more of them in chapter 3.

Logging is all well and good to see what the state of things might be as the code is running, but sometimes we want to stop the action and take a look around.

That's where breakpoints come in.

2.1.2 Breakpoints

Breakpoints, a somewhat more complex concept than logging, possess a notable advantage over logging: they halt the execution of a script at a specific line of code, pausing the browser. This allows us to leisurely investigate the state of all sorts of things at the point of the break. This includes all accessible variables, the context, and the scope chain.

Let's say that we have a page that employs our new `log()` method as shown in listing 2.2.

Listing 2.2 A simple page that uses our custom log() method

```
<!DOCTYPE html>
<html>
  <head>
    <title>Listing 2.2</title>
    <script type="text/javascript" src="log.js"></script>
    <script type="text/javascript">
      var x = 213;
      log(x);                                                    #1
    </script>
  </head>
  <body>
  </body>
</html>
```

#1 Line upon which we will break

If we were to set a breakpoint using Firebug on the annotated line (#1) in listing 2.2 (by clicking on the line number margin in the Script display) and refresh the page to cause the code to execute, the debugger would stop execution at that line and show us the display in figure 2.2.

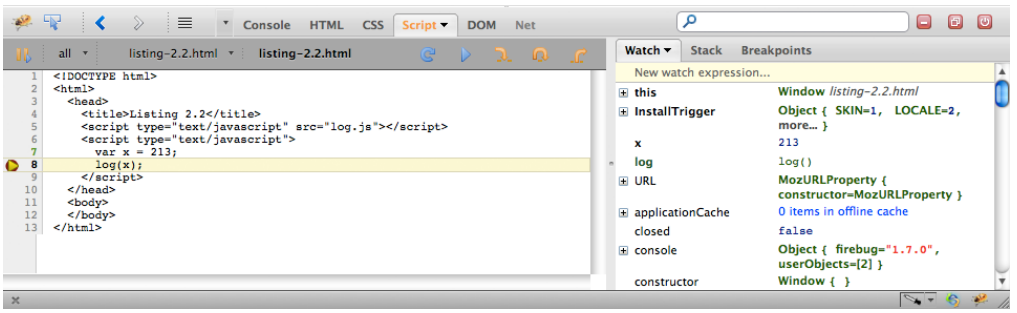

Figure 2.2 Breakpoints allow us to halt execution at a specific line of code so we can take a gander at the state

Note how the rightmost pane allows us to see the state within which our code is running, including the value of x.

The debugger breaks on a line *before* that line is actually executed; so in this example, the call to our `log()` method has yet to be executed. If we were to imagine that we were trying to debug a problem with our new method, we might want to *step into* that method to see what's going on inside it.

Clicking on the "step into" button (left-most gold arrow button) causes the debugger to execute up to the first line of our method, and we'd see the display of figure 2.3.

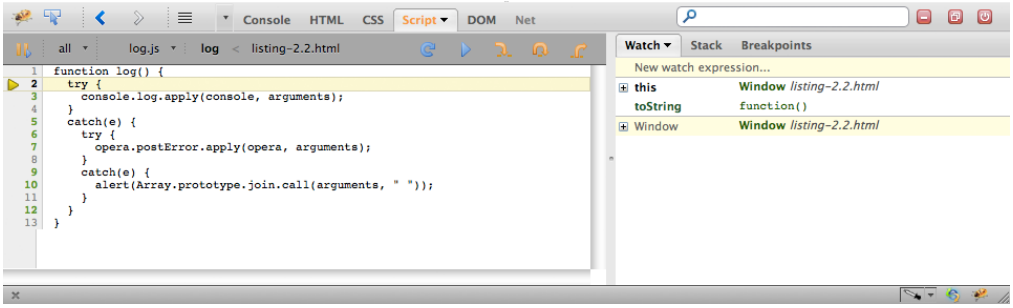

Figure 2.3 Stepping into our method lets us see the new state within which the method executes

Note how the displayed state has changed to allow us to poke around the new state within which our `log()` method executes.

Any fully featured debugger with breakpoint capabilities is highly dependent upon the browser environment in which it is executing. For this reason, the aforementioned developer tools were created as the functionality provided by them would not be otherwise possible. It is a great boon and relief to the entire web development community that all the major browser implementers have come on board to create effective utilities for allowing debugging activities.

Debugging code not only serves its primary and obvious purpose (detecting and fixing bugs), it also can helps us achieve the good practice goal of generating effective test cases.

2.2 Test generation

Robert Frost wrote that good fences make good neighbors, but in the world of web applications, indeed any programming discipline, good tests make good code.

Note the emphasis on the word *good*. It's quite possible to have an extensive test suite that doesn't really help the quality of our code one iota if the tests are poorly constructed. Good tests exhibit three important characteristics:

- **Repeatability** – our test results should be highly reproducible. Tests run repeatedly should always produce the exact same results. If test results are nondeterministic, how would we know what are valid results versus invalid results? Additionally this

helps to make sure that your tests aren't dependent upon external factors issues like network or CPU loads.

• **Simplicity** – our tests should focus on testing one thing**.** We should strive to remove as much HTML markup, CSS, or JavaScript as we can *without* disrupting the intent of the test case. The more that we remove, the greater the likelihood that the test case will only be influenced by the specific code that we are trying to test.

• **Independence** – our tests should execute in isolation. We must strive to not make the results from one test be dependent upon another. We should break tests down into their smallest possible unit, which helps us to determine the exact source of a bug when an error occurs.

There are a number of approaches that can be used for constructing tests, with the two primary approaches being: deconstructive tests and constructive tests. Let's examine what each of these approaches entails:

- **Deconstructive test cases**

  Deconstructive test cases are created when existing code is whittled down (deconstructed) to isolate a problem, eliminating anything that's not germane to the issue. This helps us to achieve the three characteristics listed above. We might start with a complete site, but after removing extra markup, CSS, and JavaScript, we arrive at a smaller case that reproduces the problem.

- **Constructive test cases**

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

Let's see an example of constructive testing.

When creating reduced test cases, we can start with a few HTML files with minimum functionality already included in them. We might even have different starting files for various functional areas; for example, one for DOM manipulation, one for Ajax tests, one for animations, and so on.

For example, Listing 2.3 shows a simple DOM test case used to test jQuery.

Listing 2.3: 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 2.4, showing file `gen.sh`.

Listing 2.4: A simple shell script used to generate a new test case

```
#!/bin/sh
# Check out a fresh copy of jQuery
git clone git://github.com/jquery/jquery.git $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
```

The above script would be executed using the command line:

```
./gen.sh mytest dom
```

which would pull in the DOM test case from `dom.html` in the git repository.

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 JSBin is shown in Figure 2.4.

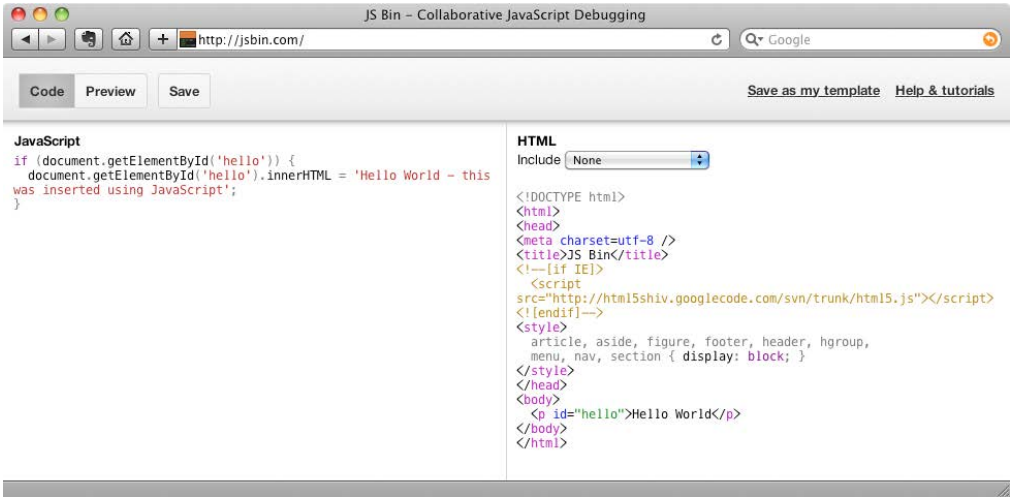

Figure 2.4: A screenshot of the JSBin web site in action

With the tools and knowledge in place for figuring out how to create test cases, we can start to build test suites around these cases so that it becomes easier to run these tests over and over again. Let's look into that.

2.3 Testing frameworks

A test suite should serve as a fundamental part of your development workflow. For this reason you should pick a suite that that works particularly well for you your coding style, and your code base.

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. Testing frameworks can help us reach that goal without us having to worry about anything but creating the tests and organizing them into suites.

There are a number of features that we might want to look for in a JavaScript unit-testing framework, depending upon the needs of the tests. Some of these features include:

- The ability to simulate browser behavior (clicks, key presses, an so on).
- Interactive control of tests (pausing and resuming tests).
- Handling asynchronous test time outs.
- The ability to filter which tests are to be executed.

In mid-2009 a survey was conducted, attempting to determine what JavaScript testing frameworks people used in their day-to-day development. The results were quite illuminating.

The raw results, should you be interested, can be found at http://spreadsheets.google.com/pub?key=ry8NZN4-Ktao1Rcwae-9Ljw&output=html, and the charted results are as shown in figures 2.5, 2.6 and 2.7.

The first figure depicts the disheartening fact that a lot of the respondents don't test at all. In the wild, it's easy to believe that the percentage of non-testers is actually quite higher.

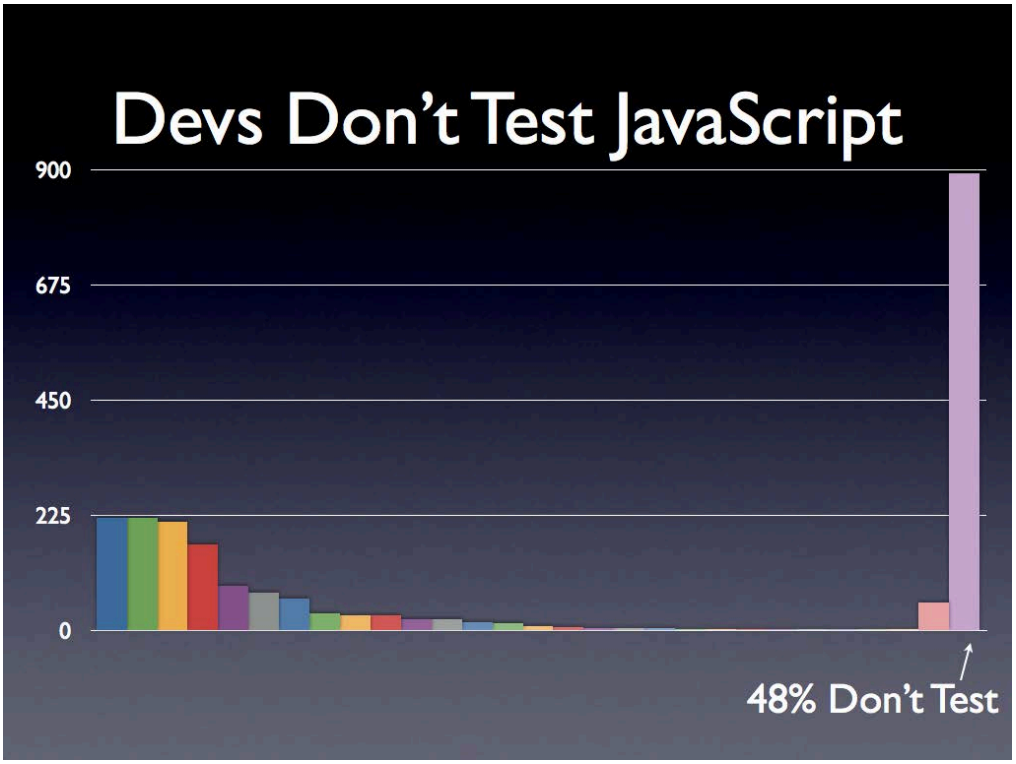

Figure 2.5 A dishearteningly large percentage of script developers don't test at all

Another insight from the results is that the vast majority of scrupt authors that do write tests use one of four tools, all of which were pretty much tied in the results: JSUnit, QUnit, Selenium, and YUITest. The top ten "winners" are shown in figure 2.6.

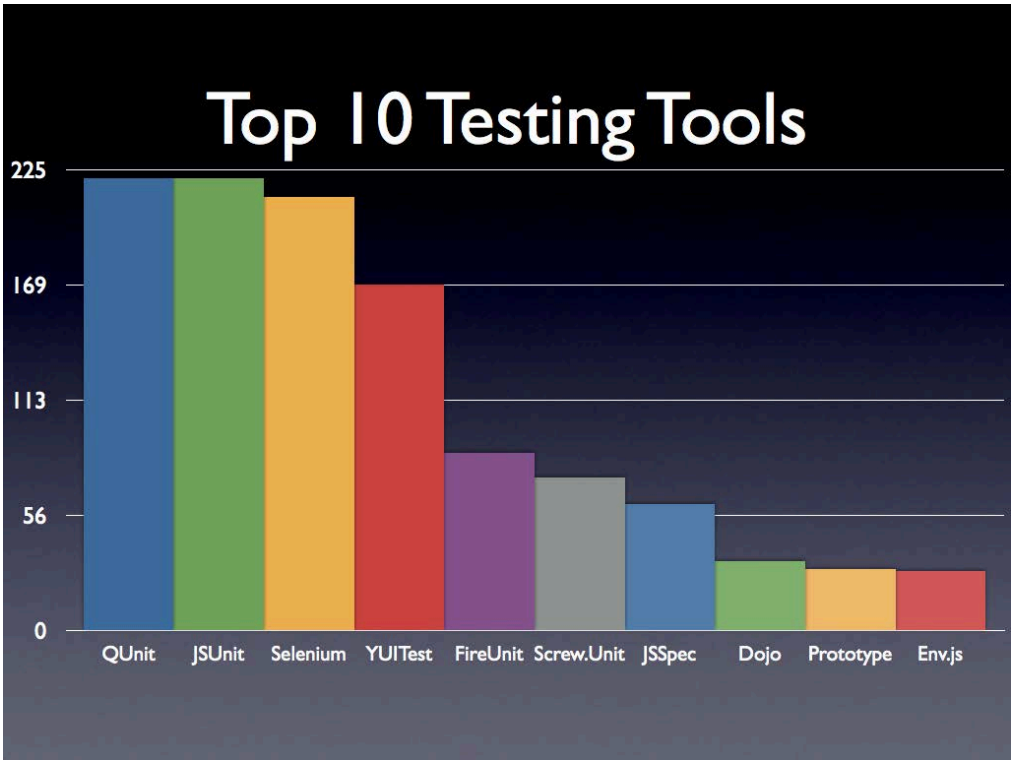

Figure 2.6 Most test-savvy developers favor a small handful of testing tools

An interesting result, showing that there isn't any one definitive preferred testing framework at this point. But even more interesting is the massive "long tail" of one-off frameworks that have one, or very few, users as shown in figure 2.7.

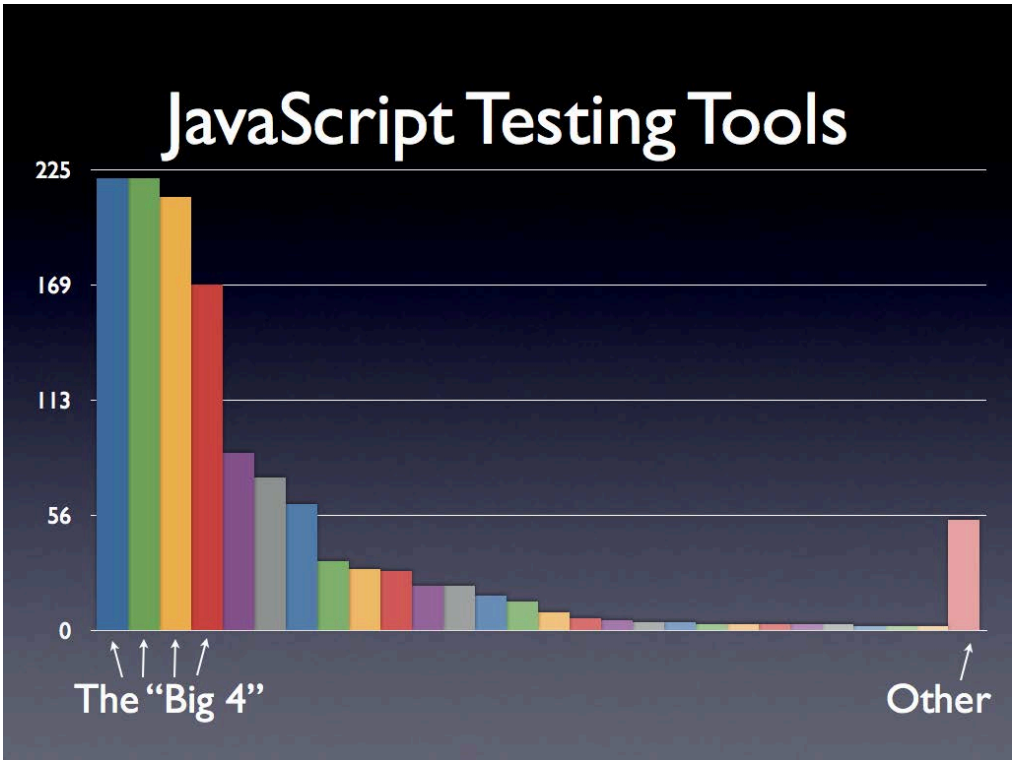

Figure 2.7 The remainder of the testing tools have few users

It should be noted that it's fairly easy for someone to write a testing framework from scratch, and that's not a bad way to help him or her to gain a greater understanding of what a testing framework is trying to achieve. This is an especially interesting exercise to tackle because, when writing a testing framework, typically we'd be dealing with pure JavaScript without having to worry much about dealing with many cross-browser issues. Unless, that is, you're trying to simulate browser events, then good luck!

Obviously, according to the result depicted in figure 2.7, a number of people have come to this same conclusion and have written a large number of one-off frameworks to suite their own particular needs.

General JavaScript unit testing frameworks tend to provide a few basic components: a test runner, test groupings, and assertions. Some also provide the ability to run tests asynchronously.

But while it is quite easy to write a proprietary unit-testing framework, it's likely that we'll just want to use something that's been pre-built. Let's take a brief survey of some of the most popular unit testing frameworks.

2.3.1 QUnit

QUnit is the unit-testing framework that was originally built to test jQuery. It has since expanded beyond its initial goals and is now a standalone unit-testing framework. QUnit is primarily designed to be a simple solution to unit testing, providing a minimal, but easy to use, API.

Distinguishing features:

- Simple API
- Supports asynchronous testing.
- Not limited to jQuery or jQuery-using code
- Especially well-suited for regression testing

More Information can be found at http://docs.jquery.com/Qunit

2.3.2 YUITest

YUITest is a testing framework built and developed by Yahoo! and released in October of 2008. It was completely rewritten in 2009 to coincide with the release of YUI 3. YUITest provides an impressive number of features and functionality that is sure to cover any unit testing case required by your code base.

Distinguishing features:

- Extensive and comprehensive unit testing functionality
- Supports asynchronous tests
- Good event simulation

More Information is available at http://developer.yahoo.com/yui/3/test/

2.3.3 JSUnit

JSUnit is a port of the popular Java JUnit testing framework to JavaScript. While it's still one of the most popular JavaScript unit testing frameworks around, JSUnit is also one of the oldest (both in terms of the code base age and quality). The framework hasn't been updated much recently, so for something that's known to work with all modern browsers, JSUnit may not be the best choice.

More information can be found at http://www.jsunit.net/

Next, we'll take a look at creating test suites.

2.4 The Fundamentals of 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.

To better understand how a test suite works it makes sense to look at how a test suite is constructed. Perhaps surprisingly JavaScript test suites are really easy to construct and a functional one can be built in only about 40 lines of code.

One would have to ask, though: Why would I want to build a new test suite? 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 (as already shown). It can serve as a good learning experience though, especially when looking at how asynchronous testing works.

2.4.1 The assertion

The core of a unit-testing framework is its assertion method; usually named `assert()`. This method usually takes a value – an expression whose premise is *asserted* – and a description that describes the purpose of the assertion. If the value evaluates to `true`, in other words is "truth-y", then the assertion passes, otherwise it is considered a failure. The associated message is usually logged with an appropriate pass/fail indicator.

A simple implementation of this concept can be seen in Listing 2.8.

Listing 2.8: A simple implementation of a JavaScript assertion

```
<html>
  <head>
    <title>Test Suite</title>
    <script>

      function assert(value, desc) {                              #1
        var li = document.createElement("li");                    #1
        li.className = value ? "pass" : "fail";                   #1
        li.appendChild(document.createTextNode(desc));            #1
        document.getElementById("results").appendChild(li);       #1
      }                                                           #1

      window.onload = function() {
        assert(true, "The test suite is running.");               #2
        assert(false, "Fail!");                                   #2
      };
    </script>

    <style>
      #results li.pass { color: green; }                          #3
      #results li.fail { color: red; }                            #3
    </style>
  </head>

  <body>
    <ul id="results"></ul>                                        #4
  </body>
</html>
```

#1 Defines the assert() method
#2 Executes tests using assertions
#3 Defines styles for results
#4 Holds test results

The function named `assert()` (#1) is almost surprisingly straight-forward. It creates a new `<li>` element containing the description, assigns a class name pass or fail, depending

upon the value of the assertion parameter (`value`), and appends the new element to a list element in the document body (#4).

The simple test suite consists of two trivial tests (#2): one that will always succeed, and one that will always fail.

Style rules for the `pass` and `fail` classes (#3) visually indicate success or failure using color.

This function is simple - but it will serve as a good building block for future development, and we'll be using this `assert()` method throughout this book to test various code snippets, verifying their integrity.

2.4.2 Test groups

Simple assertions are useful, but really begin to shine when they are grouped together in a testing context to form ***test groups***.

When performing unit testing, a test group will likely represent a collection of assertions as they relate to a single method in our API or application. If you were doing behavior-driven development the group would collect assertions by task. Either way the implementation is effectively the same.

In our sample test suite, a test group is built in which individual assertions are inserted into the results. Additionally if any assertion fails then the entire test group is marked as failing. The output in Listing 2.8 is kept pretty simple, some level dynamic control would prove to be quite useful in practice (contracting/expanding the test groups and filtering test groups if they have failing tests in them).

Listing 2.9: An implementation of test grouping

```
<html>
  <head>
    <title>Test Suite</title>
    <script>

      (function() {
        var results;
        this.assert = function assert(value, desc) {
          var li = document.createElement("li");
          li.className = value ? "pass" : "fail";
          li.appendChild(document.createTextNode(desc));
          results.appendChild(li);
          if (!value) {
            li.parentNode.parentNode.className = "fail";
          }
          return li;
        };
        this.test = function test(name, fn) {
          results = document.getElementById("results");
          results = assert(true, name).appendChild(
              document.createElement("ul"));
          fn();
        };
```

```
    })();

    window.onload = function() {
      test("A test.", function() {
        assert(true, "First assertion completed");
        assert(true, "Second assertion completed");
        assert(true, "Third assertion completed");
      });
      test("Another test.", function() {
        assert(true, "First test completed");
        assert(false, "Second test failed");
        assert(true, "Third assertion completed");
      });
      test("A third test.", function() {
        assert(null, "fail");
        assert(5, "pass")
      });
    };
  </script>
  <style>
    #results li.pass { color: green; }
    #results li.fail { color: red; }
  </style>
 </head>
 <body>
  <ul id="results"></ul>
 </body>
</html>
```

As we cab see seen in Listing 2.9, the implementation is really not much different from our basic assertion logging. The one major difference is the inclusion of a results variable, which holds a reference to the current test group (that way, the logging assertions are inserted correctly).

Beyond simple testing of code, another important aspect of a testing framework is handling of asynchronous operations.

2.4.3 Asynchronous Testing

A daunting and complicated tasks that many developers encounter while developing a JavaScript test suite, is handling asynchronous tests. These are tests whose results will come back *after* a non-deterministic amount of time has passed; common examples of this situation could be Ajax requests or animations.

Often handling this issue is over-though and made much more complicated than it needs be. To handle asynchronous tests we need to follow a couple of simple steps:

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

Let's look at the example of Listing 2.10.

Listing 2.10: A simple asynchronous test suite

```
<html>
  <head>
    <title>Test Suite</title>
    <script>
      (function() {
        var queue = [], paused = false, results;
        this.test = function(name, fn) {
          queue.push(function() {
            results = document.getElementById("results");
            results = assert(true, name).appendChild(
                document.createElement("ul"));
            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();
            }
          }
        }

        this.assert = function assert(value, desc) {
          var li = document.createElement("li");
          li.className = value ? "pass" : "fail";
          li.appendChild(document.createTextNode(desc));
          results.appendChild(li);
          if (!value) {
            li.parentNode.parentNode.className = "fail";
          }
          return li;
        };
      })();
      window.onload = function() {
        test("Async Test #1", function() {
          pause();
          setTimeout(function() {
            assert(true, "First test completed");
            resume();
          }, 1000);
        });
        test("Async Test #2", function() {
          pause();
```

```
            setTimeout(function() {
              assert(true, "Second test completed");
              resume();
            }, 1000);
          });
        };
      </script>
      <style>
        #results li.pass {
          color: green;
        }

        #results li.fail {
          color: red;
        }
      </style>
    </head>
    <body>
      <ul id="results"></ul>
    </body>
  </html>
```

Let's break down the functionality exposed in in Listing 2.10. There are three publicly accessible functions: `test()`, `pause()`, and `resume()`. These three functions have the following capabilities:

- `test(fn)` takes a function which contains a number of assertions, that will be run either synchronously or asynchronously, and places it on the queue to await execution.
- `pause()` should be called from within a test function and tells the test suite to pause executing tests until the test group is done.
- `resume()` unpauses the tests and starts the next test running after a short delay pu inyo place to avoid long-running code blocks.

The one internal implementation function, `runTest()`, is called whenever a test is queued or dequeued. It checks to see if the suite is currently unpaused and if there's something in the queue; in which case 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.

We'll be taking a closer look in chapter 6 which focuses on Timers, where we'll make an in-depth examination of much of the nitty-gritty relating to delayed execution.

2.5 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.

We started off by examining how to use logging to observe the actions of our code as it is running and even implemented a convenience method that we can use to make sure that we can successfully log information in all modern browsers, despite their differences.

We then explored how to use breakpoints to halt the execution of our code at a certain point, allowing us to take a look around at the state within which the code is executing.

Our attention then turned to test generation, defining and focusing on the attributes of good tests: *repeatability*, *simplicity* and *independence*. The two major types of testing, *deconstructive* and *constructive* testing were then examined.

Data collected regarding how the JavaScript community is using testing was presented, and we took a brief survey of existing test frameworks that you might want to explore and adopt should you want use a formalized testing environment.

Building upon that, we introduced the concept of the assertion, and created a simple implementation that will be used throughout the remainder of this book to verify that our code does what we intend for it to do.

Finally, we looked at how to construct a simple test suite capable of handling asynchronous test cases. Altogether, these techniques will serve as an important cornerstone to the rest of your development with JavaScript.

3

Functions are fundamental

In this chapter:

- Why understanding functions is so crucial
- How functions are *first-class* objects
- The context within a function
- Handling variable argument lists
- Checking for functions

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 objects: they coexist with, and can be treated like, any other JavaScript object. Just like more mundane JavaScript data types, they can be referenced by variables, declared with literals, and even passed as function parameters.

One of the most important features of the JavaScript language is the ability to create anonymous functions just about anywhere within the code. In addition to making the code more compact and easy to understand (by putting function declarations near where they are used), it eliminates the need to pollute the global namespace with unnecessary names when a function isn't going to be referenced multiple places within the code.

But regardless of how functions are declared, they can be referenced as values, and be used as the fundamental building blocks for reusable code libraries. Understanding how functions, including anonymous functions, work at their most fundamental level will drastically improve our ability to write clear, concise and reusable code.

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

3.1 Declaring functions

There are a number of ways that JavaScript allows us to declare functions. While it's likely that the most common method for JavaScript developers to declare a function is as a named object, to be accessed elsewhere within the same scope, this isn't the most powerful and versatile way of doing things. We'll be looking at how declaring functions as variables, or even object properties, gives us the means to construct better and reusable JavaScript code.

Let's take a look, in Listing 3.1, at some different ways of declaring functions.

Listing 3.1: Three different ways to declare functions

```
<!DOCTYPE html>
<html>
  <head>
    <title>Listing 3.1</title>
    <script type="text/javascript">
      function isNimble(){ return true; }               #1
      var canFly = function(){ return true; };          #2
      window.isDeadly = function(){ return true; };     #3

      console.log(window.isNimble);                     #4
      console.log(window.canFly);                       #4
      console.log(window.isDeadly);                     #4
    </script>
  </head>
</html>
```

#1 Declared as named
#2 Declared as a variable
#3 Declared as a property
#4 Emits window properties

In this code block we declare a function named `isNimble()` using the customary `function` keyword (#1), an anonymous inline function assigned to a variable named `canFly` (#2), and a second anonymous function which is assigned to a property of window anmed `isDeadly` (#3).

> **NOTE** If this is the first time that you've seen a function declared in an inline and anonymous fashion as a ***function literal***, then lines (#2) and (#3) may look a bit odd to you. But the syntax is actually rather simple: the keyword `function`, followed by the parameter list for the function (empty in these cases), followed by the function body. In other words, just like a named function declaration but leaving off the name!
>
> This literal can be used as value anywhere that a value is allowed within the language. It's no different, really, than a numeric literal such as 213 for declaring a numeric value, just a lot wordier!

When loaded into a browser, the logging shows the output of figure 3.1.

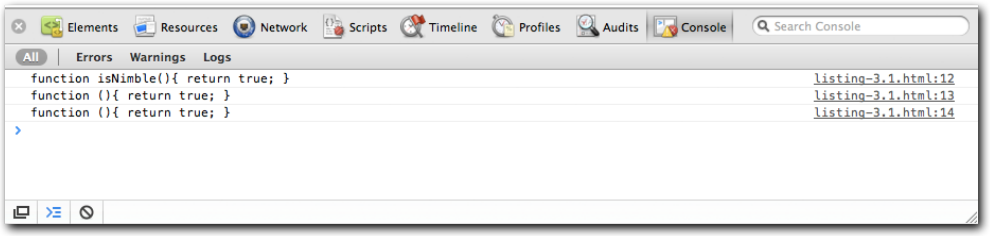

Figure 3.1 Regardless of how we declare a function, it ends up as a property on the `window` object

All these different means of declaring a function are essentially equivalent when evaluated within the global scope: the window object. As we can see by the logging (#4), all three declarations end up creating a property on window that references the declared function. We'll be seeing more on this in a little bit.

But while these methods of function declaration appear to be functionally equivalent (no pun intended), there are some subtle differences. One you may have already noticed in the output of figure 3.1 – the function declared as a named function retains its name as part of its declaration while the anonymous functions are, well, anonymous (nameless).

Another difference becomes noticeable when we change when we shift the order of the declarations, as shown in Listing 3.2.

Listing 3.2: A look at the location of named functions

```
<!DOCTYPE html>
<html>
  <head>
    <title>Listing 3.2</title>
    <script type="text/javascript"
            src="../scripts/assert.js"></script>            <!--#1-->
  </head>
  <body>
    <ul id="results"></ul>
    <script type="text/javascript">
      var canFly = function(){ return true; };
      window.isDeadly = function(){ return true; };

      assert(isNimble() && canFly() && isDeadly(),            //#2
             'Works, even though isNimble is declared below.');//#2

      function isNimble(){ return true; }                     //#3
    </script>
  </body>
</html>
```

#1 Includes assert()
#2 Uses all 3 functions
#3 Declares function after its use

When we load the page of listing 3.2 into a browser, we see the display of figure 3.2.

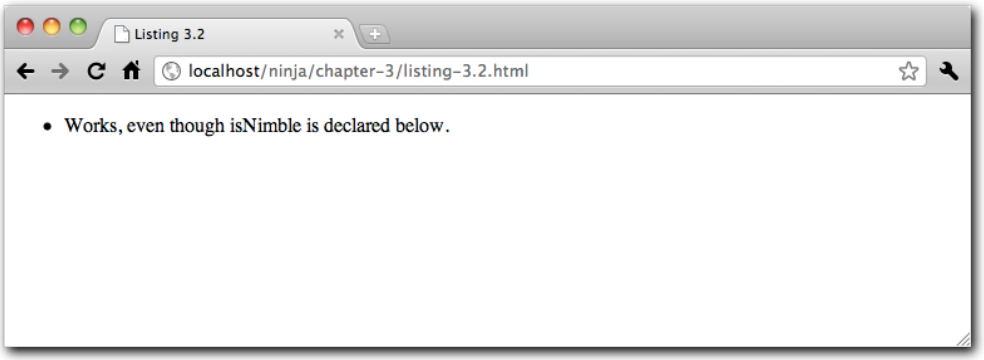

Figure 3.2 The output of the `assert()` proves that named functions can be forward referenced

This result shows us that, even though the `isNimble()` function is declared *after* the `assert()` that references it, the reference is valid! What's up with that?

Turns out that we're able to forward reference the `isNimble()` function because of a simple property of named function declartions: that is, no matter where within a scope that we define a function, it will be accessible throughout that scope.

This is not true of the anonymous functions that we declared and assigned to a variable and a property. As the only way to access those functions is through the variable or property, and because neither variables nor properties can be forward-referenced, we must assign the anonymous functions to the variable or property prior to their first reference.

We can convince ourselves of that by modifying the code of listing 3.2 such that the variable assignment to `canFly` is moved to after the assert. Upon loading this page, the JavaScript console gives us the bad news shown in figure 3.3.

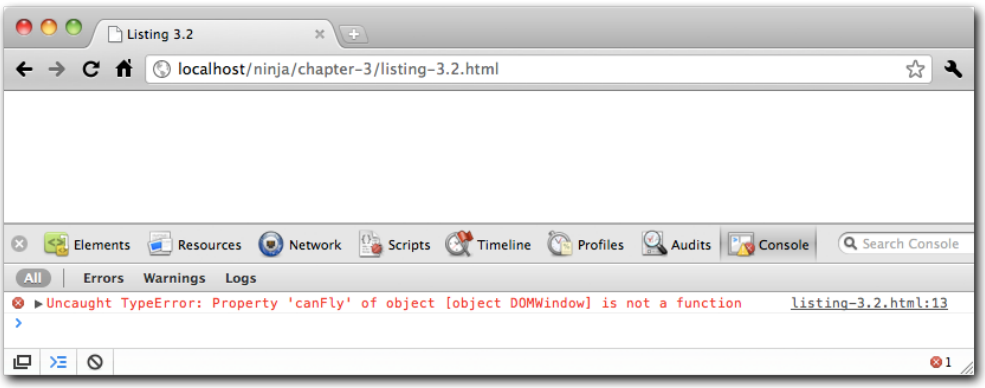

Figure 3.3 Attempting to forward-reference variables doesn't make the browser happy

That ability to forward-reference named functions is demonstrated even more dramatically by the code of listing 3.3.

Listing 3.3: Declaring a function below a return statement

```
<!DOCTYPE html>
<html>
  <head>
    <title>Listing 3.3</title>
  </head>
  <body>
    <ul id="results"></ul>
    <script type="text/javascript">
      function stealthCheck(){
        assert(stealth(),                                          #1
               "We'll never get below the return, but that's OK!");
        return true;
        function stealth(){ return true; }                         #2
      }
      stealthCheck();                                              #3
    </script>
  </body>
</html>
```

#1 Invokes yet-to-be-declared function
#2 Declares function
#3 Invokes test function

In the listing, the declaration of `stealth()` will never be reached in the normal flow of code execution as it's after an unconditional return statement. However, because it's specially privileged as a named function, it'll still be defined within our scope, and we can invoke it successfully.

As previously noted, anonymous functions, being bound by the rules regarding the variables and properties that reference them, do not get this benefit. They can only be referenced after their point of declaration, as proven by the code of Listing 3.4.

Listing 3.4: Types of function definition that can't be placed anywhere.

```
<!DOCTYPE html>
<html>
  <head>
    <title>Listing 3.4</title>
    <script type="text/javascript" src="../scripts/assert.js"></script>
  </head>
  <body>
    <ul id="results"></ul>
    <script type="text/javascript">
      assert(typeof canFly == "undefined",                         #A
             "canFly can't be forward referenced.");               #A
      assert(typeof isDeadly == "undefined", "Nor can isDeadly." );#A
      var canFly = function(){ return true; };                     #B
      window.isDeadly = function(){ return true; };                #B
    </script>
  </body>
```

```
</html>
```

#A Tests our supposition
#B Declares functions after references

In this test, we assert, successfully, as shown in figure 3.4, that the anonymous functions assigned to variables and properties cannot be forward referenced.

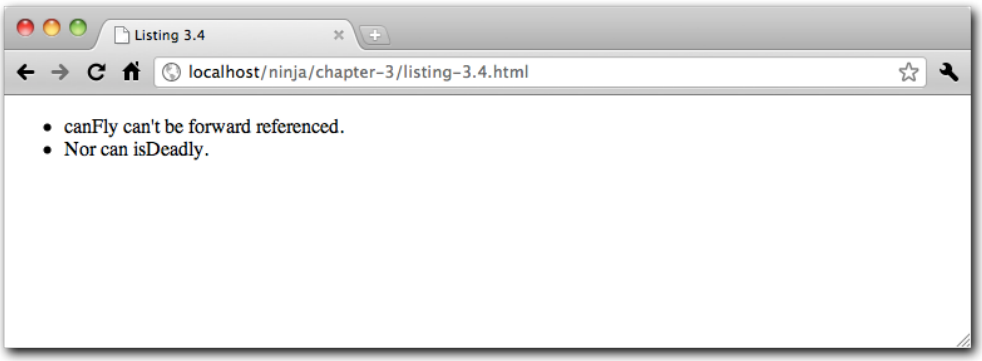

Figure 3.4 We assert, correctly, that anonymous functions cannot be forward referenced

This concept is very important as it begins to lay down the fundamentals for the naming, flow, and structure that functional code exists within, and begins to establish the framework through which we will understand anonymous functions.

3.2 Anonymous Functions

You may or may not have been familiar with anonymous functions prior to their introduction in the previous section, but they are a crucial concept with which we all need to be familiar. They are an important and logical feature for a language that takes a great deal of inspiration from functional languages like Scheme. Anonymous functions are typically used in cases where we wish to create a function for later use, such as storing it in a variable or as a property of an object, or using it as a callback, but where it doesn't need to have a name for later reference. We'll see plenty of examples throughout the rest of the book so don't panic if that seems a bit daunting at the moment.

Listing 3.5 shows a few common examples of anonymous function declarations.

Listing 3.5: Examples of declaring anonymous functions

```
<!DOCTYPE html>
<html>
  <head>
    <title>Listing 3.5</title>
    <script type="text/javascript">
      var obj = {
```

```
      someMethod: function(){                                    #1
        console.log('Hi!');                                      #1
      }                                                          #1
    };

    obj.someMethod();                                            #2

    setInterval(
      function(){ console.log('Here!'); },                       #3
      1000);

  </script>
 </head>
</html>
```

#1 Declares function property
#2 Invokes function via property
#3 Declares function parameter

In listing 3.5, we declare an anonymous function as a property of an object (#1) and then invoked it using the property reference (#2). Then we supply an anonymous function as a parameter to the `setInterval()` method (of `window`) (#3), which is invoked every second when the interval expires.

The results (after letting things run for about 5 seconds) can be seen in figure 3.5.

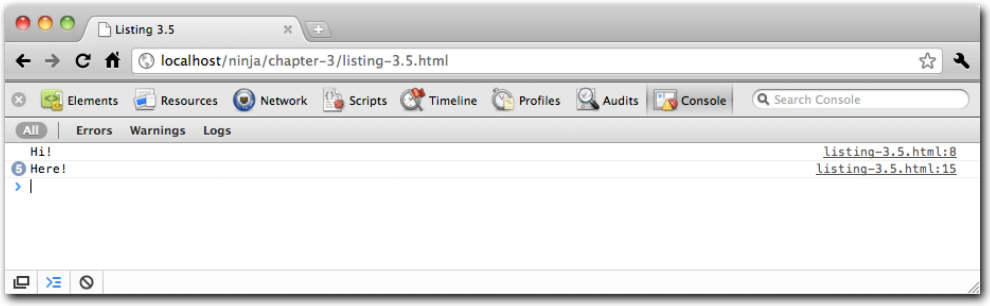

Figure 3.5 Anonymous functions can be called at later times even without being named

Note how in both of these cases, the functions didn't need to be named in order to be used after their declarations.

NOTE You might argue that by assigning an anonymous function to a property named `someMethod` that we give the function a name, but that's not a valid way of thinking about it. The `someMethod` name is the name of the *property*, not of the function itself. Review the results of listing 3.1 in figure 3.1 to see that anonymous functions do not possess names in the same manner that named function do (which, by the way, is stored in a property of the function instance named `name`).

We're going to see that anonymous functions are going to be able to solve many of the challenges that we'll face when developing JavaScript applications. In the rest of this chapter we'll be expanding on their use, and show some alternative ways in which they can be employed, starting with recursion.

3.2.1 Recursion

We can learn even more about how functions work within JavaScript by considering ***recursion***.

Generally speaking, recursion is a well understood concept. When a function calls itself, or calls a function that in turn calls the original function anywhere in the call tree, recursion occurs. Things get a bit more interesting, and somewhat less clear, when we begin dealing with anonymous functions. Let's consider the code of listing 3.6.

Listing 3.6: Recursion in a named function

```
<script type="text/javascript">
  function yell(n) {                                    #1
    return n > 0 ? yell(n-1) + "a" : "hiy";             #1
  }                                                     #1

  assert(yell(4) == "hiyaaaa",                          #2
         "Calling the named function comes naturally.");

</script>
```

#1 Declares recursive function
#2 Asserts that it works as planned

In this listing, we declare a function named `yell()` that employs recursion by calling itself by name. Our test verifies that the function works as intended (#2).

> **NOTE** This function satisfies two criteria for recursion: a reference to self, and convergence towards termination.
>
> The function clearly calls itself, so the first criterion is satisfied. And because the value of parameter `n` decreases with each iteration, it will sooner or later reach a value less than zero and stop the recursion, satisfying the second criterion.
>
> Note that a "recursive" function that does *not* converge towards termination is better known as an infinite loop!

It's pretty clear how all this works with a named function, but what if we were to use anonymous functions? Let's complicate the situation a bit by declaring a recursive anonymous function as an object's property as in listing 3.7.

Listing 3.7: Function recursion within an object.

```
<script type="text/javascript">
      var ninja = {
        yell: function(n) {                                          #1
          return n > 0 ? ninja.yell(n - 1) + "a" : "hiy";
        }
      };

      assert(ninja.yell(4) == "hiyaaaa",
            "An object property isn't too bad, either.");
    </script>
```

#1 Declares function as a property

In this listing we defined our recursive e function as an anonymous function referenced by the `yell` property of the `ninja` object (#1).Within the function, we invoke the function recursively via a reference to the object's property: `ninja.yell()`.

That's all fine until we decide to create a new object, let's say `samurai`, referencing the anonymous function from the `ninja`. Consider the code of listing 3.8.

Listing 3.8: Recursion using a missing function reference

```
<script type="text/javascript">
  var ninja = {
    yell: function(n) {
      return n > 0 ? ninja.yell(n - 1) + "a" : "hiy";
    }
  };

  var samurai = { yell: ninja.yell };                        #1

  ninja = {};                                                #2

  try {
    assert(samurai.yell(4) == "hiyaaaa",                     #3
          "Is this going to work?");
  }
  catch(e){
    console.log("Uh, this isn't good! Where'd ninja.yell go?");
  }
</script>
```

#1 References ninja function
#2 Redefines ninja
#3 Tests if it still works

We can see how things start to break down in this scenario. After we redefine `ninja` with an empty object (#2) the anonymous function still exists, and can be referenced through the `samurai.yell` property, but the `ninja.yell` property no longer exists. And as the function recursively calls itself through that reference, things go badly awry (#3) when the function is invoked.

We can rectify this problem by fixing the initially sloppy definition of the recursive function. Rather than explicitly referencing `ninja` in the anonymous function, we should have used the function context (`this`) as follows:

```
var ninja = {
    yell: function(n) {
      return n > 0 ? this.yell(n - 1) + "a" : "hiy";
    }
  };
```

When we use `this` in a method – a term frequently applied to functions that are referenced through object properties – the function context refers to the object through which the method was invoked.

So when invoked as `ninja.yell()`, this refers to `ninja`, but when invoked by `samurai.yell()`, this refers to samurai and all is well.

Using this makes our `yell()` method much more robust, and is the way it should have been declared in the first place.Problem solved. But…

Let's consider another approach, if for no other reason to introduce more functional concepts. What if we give the anonymous function a name?

This may seem completely contradictory; how is a function anonymous if it has a name. Well, as it turns out, the function literal syntax allows us to supply a name to the declared function by adding a name before the parameter list (just like when declaring named functions). Arguably, it might be better to call these ***inline functions***, rather than "anonymous" to avoid the contradiction.

Observe the use of this technique in Listing 3.9:

Listing 3.9: Using a named anonymous (inline) function

```
<script type="text/javascript">
  var ninja = {
    yell: function shout(n){                                    #1
      return n > 0 ? shout(n-1) + "a" : "hiy";
    }
  };

  assert(ninja.yell(4) == "hiyaaaa",                            #2
         "Works as we would expect it to!");

  var samurai = { yell: ninja.yell };
  ninja = {};                                                   #3

  assert(samurai.yell(4) == "hiyaaaa",                          #4
         "The method correctly calls itself.");
</script>
```

#1 Declares inline function
#2 Test that it works
#3 Wipes out the reference
#4 Tests that it still works

Here (#1) we assign the name `shout` to the inline function, and use that name for the recursive reference within the function body, and then test (#2) that calling as a method of `ninja` still works.

As before, we copy the reference to the function to `samurai.yell`, and wipe out the original `ninja` object (#3).

Upon testing calling the function as a method of `samurai` (#4), we find that everything still works because wiping out the `yell` property of `ninja` had no effect on the name we gave to the inline function (and used to perform the recursive call).

This ability to name an inline function extends even further. It can even be used within normal variable assignments with some seemingly bizarre results., like in Listing 3.10:

Listing 3.10: Verifying the identity of a named, anonymous, function.

```
<script type="text/javascript">
  var ninja = function myNinja(){                                    #1
    assert(ninja == myNinja,                                         #2
           "This function is named two things at once!");            #2
  };

  ninja();                                                           #3

  assert(typeof myNinja == "undefined",                              #4
         "But myNinja isn't defined outside of the function.");      #4

</script>
```

#1 Declares named inline function
#2 Tests internal referencing
#3 Invokes function and test
#4 Test external referencing

This listing brings up the most important point regarding inline (named anonymous) functions: **inline functions can be named, but those names are only visible within the functions themselves.**

We declare an inline function with the name `myNinja` (#1) and internally test to be sure that the name and the reference to which the function is assigned refer to the same thing (#2). Calling the function invokes this test (#3).

Then, we test that the function name is not externally visible (#4). And, as expected, when we run the code, the test passes.

So while givinganonymous functions a name may provide a means to clearly allow recursive references within the function – arguably, this approach provides more clarity than using `this` – it has limited utility elsewhere.

Let's look at yet another way to approach the issue that introduces another concept: the callee property of the `arguments` object. Consider the code of listing 3.11.

Listing 3.11: Using `arguments.callee` to reference a function itself.

```
<script type="text/javascript">
```

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

#1 References the argument.callee property

The `arguments` reference is available within every function (pointing to the argument list) and the `arguments.callee` property can serve as a reliable way to always access the function itself. Later on in this chapter, and in the next chapter (chapter 4, on closures), we'll take a closer look at what can be done with this particular property.

All together, these different techniques for handling anonymous functions will be of great benefit to us as we start to scale in complexity, providing us with various means to reference functions without resorting to hard-coded and fragile dependencies like variable and property names.

The next step in our functional journey is understanding how the object-oriented nature of functions in JavaScript helps us take this knowledge to the next level.

3.3 Functions as objects

Functions in JavaScript are not like functions in many other languages. JavaScript gives functions many capabilities, not the least of which is that they are treated as, and behave like, other objects in the language.

In JavaScript, functions can have properties, have an object prototype, can be assigned to variables and properties (as we have already seen) and generally have all the abilities of plain vanilla objects, but with an amazing super-power: they are callable.

Let's look at some of the similarities functions share with other object types and how they can be made especially useful. To start with, let's recap the assignment of functions to variables.

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

Just as we can assign an object to a variable, we can do so with a function. This also applies to assigning functions to object properties.

> **Note** One thing that's important to remember is the semicolon after `function(){}` definitions. It's a good practice to have semicolons at the end of all statements; especially so after variable assignments. Doing so with anonymous functions is no exception. When compressing code (which will be discussed in chapter ??? on distribution), having properly placed semicolons will allow for greater flexibility in compression techniques.

Another capability that may come as a surprise to many people is that, just like with an object, we can attach properties to a function, as in:

```
var obj = {};
var fn = function(){};
obj.prop = "hitsuke (distraction)";
fn.prop = "tanuki (climbing)";
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 or general on-page code. This is especially so when it comes to topics like event callback management. For now, let's look at a couple of the more interesting things that can be done such as storing functions and "memoizing".

3.3.1 Storing Functions

There are times when we may want to store a collection of related but unique functions; event callback management being the most obvious example. When adding functions to such a collection, a challenge we face is determining which functions are actually new to the collection, and should be added, and which is already resident and should not be added.

An obvious technique would be to store all the functions in an array, and loop through the array checking for duplicate functions. However, this performs poorly and is impractical for most applications. Rather, we can make good use of function properties to achieve an acceptable result, as shown in in Listing 3.12.

Listing 3.12: Storing a collection of unique functions

```
<script type="text/javascript">
  var store = {
    nextId: 1,                                    //#1
    cache: {},                                    //#2
    add: function(fn) {                           //#3
      if (!fn.id) {                               //#3
        fn.id = store.nextId++;                   //#3
        return !!(store.cache[fn.id] = fn);       //#3
      }                                           //#3
    }
  };

  function ninja(){}

  assert(store.add(ninja),                        //#4
         "Function was safely added.");           //#4
  assert(!store.add(ninja),                       //#4
         "But it was only added once.");          //#4
</script>
```

#1 Keeps track of available ids
#2 Stores the functions
#3 Adds only unique functions
#4 Tests that it works

In Listing 3.12, we create an object, assigned to variable `store`, in which we will store a unique set of functions. This object has two data properties: one which stores a next available id value (#1), and one within which we will cache the stored functions (#2). Functions are added to this cache via the `add()` method (#3).

Within `add()`, we first check to see if an `id` property has been added to the function, and if so, we assume that the function has already been processed and we ignore it. Otherwise, we assign an `id` property to the function (incrementing the `nextId` property along the way, and add the function as a property of the `cache`, using the `id` value as the property name.

We then return the value `true`, which we compute the hard way by converting the function to its Boolean equivalent, so that we can tell when the function was added after a call to `add()`.

> **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 into its Boolean equivalent, which will always be `true`. (Sure we could have hard-coded true, but then we wouldn't have had a chance to introduce `!!`).

Running the page in the browser shows that when our tests try to add the `ninja()` function twice (#4), the function is only added once, as shown in figure 3.6.

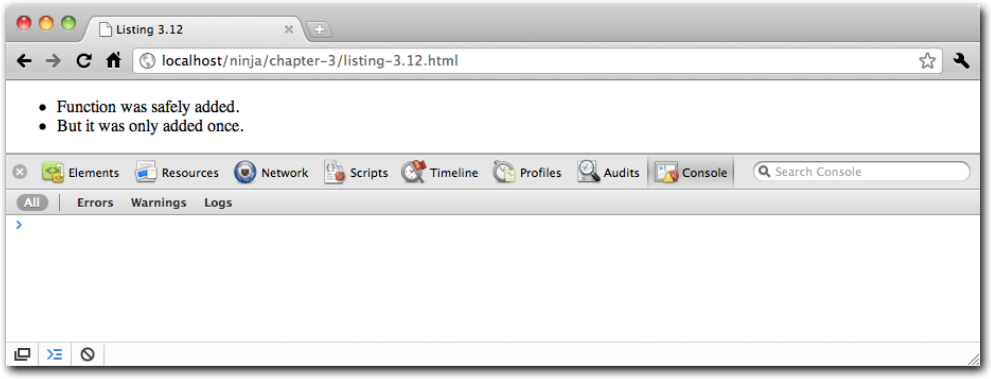

Figure 3.6 By tacking a property onto a function, we can keep track of it

Another useful trick that we can pull out of our sleeves using function properties, is giving a function the ability to modify itself. This technique could be used to remember previously-computed values; saving time during future computations.

3.3.2 Self-memoizing functions

Memoization (no, that's not a typo) is the process of building a function that 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 3.15.

Listing 3.15: A prime computation function which memorizes its previously-computed values.

```
<script type="text/javascript">
  function isPrime(value) {
    if (isPrime.answers[value] != null) {                    //#1
      return isPrime.answers[value];                         //#1
    }                                                        //#1
    var prime = value != 1; // 1 can never be prime
    for (var i = 2; i < value; i++) {
      if (value % i == 0) {
        prime = false;
        break;
      }
    }
    return isPrime.answers[value] = prime;                   //#2
  }

  isPrime.answers = {};                                      //#3

  assert(isPrime(5), "5 is prime!" );                        //#4
  assert(isPrime.answers[5], "The answer was cached!" );      //#4
</script>
```

#1 Checks for cached values
#2 Stores the computed value
#3 Creates the cache
#4 Tests that it all works

Within the `isPrime()` function, we start by checking to see if the answer has already been cached (#1) in a property named `answers`, in which we will store the computed answer (`true` or `false`) using the value as the property key. If we find a cached answer, we simply return it.

If no cached value is found, we go ahead and perform the calculations needed to determine if the value is prime (which can be an expensive operation for larger values) and store the result in the cache as we return it (#2).

After the function is declared, we add the `answers` property to it. This may seem an odd place to do this, but we can't add the property to the function until it actually exists. An alternative, which may seem more intuitive to some, is to create it lazily as part of the function itself by adding a statement such as:

```
if (!isPrime.anwers) isPrime.answers = {};
```

to the beginning of the function.

This approach has two major advantages:

- First, the user gets improved performance benefits on function calls for a previously computed value.
- Secondly, it happens completely seamlessly as the user doesn't need to perform any special request or do any extra initialization in order to make it all work.

It should be noted that any sort of caching will certainly sacrifice memory in favor of performance so should only be used when the tradeoff is deemed appropriate and helpful.

Let's take a look at another example. Querying for a set of DOM elements by tag name is a fairly common operation. We can take advantage of our newfound function memoization superpowers by building a cache that we can store the matched element sets within. Consider:

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

The memoization (caching) code is quite simple and doesn't add that much extra complexity to the overall querying process. However, if were to do some performance analysis upon the function, with the results shown in Table 3.1, we find that this simple layer of caching yields us a 7x performance increase! Not a bad superpower to have.

Table 3.1: All time in ms, for 1000 iterations, in a copy of Firefox 3

	Average	Min	Max	Deviation
non-cached version	12.58	12	13	0.50
cached version	1.73	1	2	0.45

Even without digging too deeply, the usefulness of function properties should be quite evident: we can store state and cache information in a single location, gaining not only organizational advantages, but performance benefits without any extra storage or caching objects polluting the scope. We'll be revisiting this concept throughout the upcoming chapters, as the applicability of this technique extends throughout the JavaScript language.

The ability to possess properties, just like the other objects in JavaScript, isn't the only superpower that functions have. Much of a function's power is related to its ***context***, which we'll explore next.

3.4 Context

A function's context is one of its most powerful and (as often happens) one of its most confusing features. Having an implicit `this` variable defined within every function gives us a great deal of flexibility regarding how our functions can be written, called and executed.

There's a lot to wrap our head's around regarding context – but a thorough understanding of this concept can make a drastic improvement in the quality of our functional code.

Let's start with some basics.

3.4.1 Contexts in methods

To start, it's important to realize what that the function context represents: namely, the object within which the function is being executed. For example, defining a function as a method of an object (in other words, as a property of that object) usually ensures that the method's context refers to that object. We say *usually* because, as we'll see later in this section, there are ways to override this default behavior. But for now, let's keep it simple and consider the code of listing 3.14.

Listing 3.14: Examining context within a function

```
<script type="text/javascript">
  var katana = {
    isSharp: true,                                #1
    use: function(){                              #2
      this.isSharp = !this.isSharp;               #2
    }                                             #2
  };

  katana.use();                                   #3

  assert(!katana.isSharp,                         #4
         "The value of isSharp has been changed!");
</script>
```

#1 Sets initial property value
#2 Toggles value of context proeprty
#3 Executes method
#4 Runs test

In this test, we define an object with a Boolean property named `isSharp` (#1), and a method `use()`. Within `use()`, the context variable `this` is used to reference the `katana` object, and toggle the value of its `isSharp` property (#2). Note how we did not need to use the specific reference `katana` in order to access the object – it's available as the context without needing a hard-coded name.

We then invoke the function as a method of the `katana` object (#3), and verify that the object's `isSharp` property is toggled as expected (#4).

While you might think that using the `katana` name rather than the context variable would be just as good, just different, think again. What if we decided to change the name of the katana variable to `sword`? We'd also need to remember to change all the references lest the method fail. By using the context, the method is less fragile and more versatile. As we see more usage of the context in upcoming examples, we'll see how this versatility pays off in spades.

The example of listing 3.14 shows that when a function is invoked as a method of an object, the context of the function invocation is the method's object, but what about a function that isn't explicitly declared as a property of an object?

3.4.2 Contexts in standalone functions

Functions aren't always methods of objects – frequently they are defined as standalone (top-level) elements. In such cases, the function's context refers to the global object (`window`). Let's verify that using the code of listing 3.15.

Listing 3.15: Context in a standalone function

```
<script type="text/javascript">
  function katana(){                              #1
    this.isSharp = true;                          #1
  }                                               #1

  katana();                                       #2

  assert(isSharp === true,                        #3
         "A global property now exists.");
</script>
```

#1 Declares standalone function
#2 Invokes function
#3 Test for global value

This test is similar to the example in the previous section except that, this time, we define a function named `katana()` as a standalone (top-level) function (`#1`). After invoking the function (`#2`) we test to see if, as predicted, a global value named `isSharp` has been created (`#3`).

All of this becomes quite important when we begin to deal with functions in a variety of situations – knowing what the context represents within a function has a profound effect on our functional code.

> **Note** In ECMAScript 3.1 strict mode, a slight change has taken place: when a function isn't defined as a property of an object no longer means that its context will be the global object. The change is that a function defined within another 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 contexts, it's time to dig deeper and explore more complex usages.

3.4.3 Defining the context

It might be easy to think that the context of a function is an intrinsic feature of the function itself, but in reality, it's not! What we will discover is that the context of a function has little to do with how the function is declared, but is determined by how the function is *invoked*.

That's important enough to bear repeating: *the context of a function is determined by how the function is invoked*.

We'll discover that the same function, invoked by different means, can have different contexts! It turns out we have the power to redefine the context anytime that we call a function.

JavaScript provides two methods, `call()` and `apply()`, on every function, that we can use to invoke the function. These methods not only allow use to specify the arguments to be passed to the function invocation, they allow us to define the context.

Let's see how in listing 3.16.

Listing 3.16: Choosing the context of a function call

```
<script type="text/javascript">
  function fn(){ return this; }                                    //#1

  var ronin = {};                                                  //#2

  assert(fn() == this, "The context is the global object.");       //#3
  assert(fn.call(ronin) == ronin,                                  //#4
         "The context is changed to a specific object." );
</script>
```

#1 Defines a simple function
#2 Creates an object
#3 Tests normal call
#4 Tests use of call() method

In this test, we create a function (#1) that simply returns its context as its value; we can use this to easily discover the context of any invocation. Then we declare an empty object (#2) that we'll later attempt to force as the context.

The first test (#3) invokes the function using a normal function call. From our previous discussion we'd expect the context in this case to be the global object, and our assertion shows that this is indeed the case.

In the second test, we employ the `call()` method of the function (#4) rather than a normal function call. When using `call()`, the first argument passed to the method is set as the invocation's context. Additional arguments to `call()` can be used to pass arguments to the functoin.

The two methods, `call()` and `apply()`, are quite similar to each other, each used to invoke a function with a specified context, but differ in how they pass incoming argument values. The `call()` method passes arguments individually, whereas `.apply()` passes in arguments as an array. These differences are shown in Listing 3.17.

Listing 3.17: Two methods of invoking a function with custom context

```
<script type="text/javascript">
  function add(a, b){                                  #1
    return a + b;                                      #1
  }                                                    #1
```

```
  assert(add.call(this, 1, 2) == 3,                    #2
         ".call() takes individual arguments" );

  assert(add.apply(this, [1, 2]) == 3,                 #3
         ".apply() takes an array of arguments" );
</script>
```

#1 Declares an add function
#2 Invokes function via call()
#3 Invokes function via apply()

In this example, we define a simple function (#1) to add two numbers passed as parameters. Then we call the function in two ways:

1. First we invoke the function using `call()` (#2). Note how the arguments to the target function are passed individually in the arguments to the `call()` method.
2. We then invoke the function using `apply()`, in which the arguments are specified as an array passed as the second argument to the `apply()` method.

In both of these cases, we pass the global object as the context, but the function makes no use of it.

Let's look at some simple examples of how these concepts can be practically used in JavaScript code.

3.4.4 Looping and callbacks

At this point you're likely wondering what real utility the `call()` and `apply()` methods bring to the party. After all, what's wrong with good old-fashioned function calls, and why would we possibly want to pick and choose what object serves as the context?

In all honesty, most often a straightforward function call is exactly what we need, especially if we're just making a direct call to a function. But what about when we're not? Ponder, for a moment, the concept of the ***callback function***.

When using callback functions, we pass a reference to the function to another function, and at some point that other function is responsible for invoking the callback. In this case, an indirect invocation of the function is needed. Let's examine a concrete example.

A frequent element of most JavaScript libraries is a looping function that helps to simplify the process of looping through an array. Consider the implementation of listing 3.18.

Listing 3.18: A looping function with a callback

```
<script type="text/javascript">
  function loop(array,fn){                                #1
    for (var i = 0; i < array.length; i++)                #1
      if (fn.call(array, array[i], i) === false) break;   #1
  }                                                       #1

  var num = 0;                                            #2
  var numbers = [4,5,6];                                  #2

  loop(numbers, function(value, n){
```

```
      assert(this === numbers,                                    #3
             "Context is correct.");
      assert(n == num++,                                          #4
             "Counter is as expected.");
      assert(value == numbers[n],                                 #5
             "Value is as expected.");
    });
  </script>
```

#1 Implements the looping function
#2 Declares test data
#3 Tests callback context
#4 Tests callback counter parameter
#5 Tests callback value parameter

In this code, we implement a looping function (#1) that iterates over each element of the array passed as the first argument, invoking the callback function passed as the second argument for each iteration element. The callback is invoked with the array as its context, and two arguments: the current iteration item, and the loop counter.

We then set up two variables to use as test data (#2): `num` to mirror the loop counter, and `numbers`, the array that we will iterate over.

To test, we call the `loop()` function passing a callback which contains 3 assertions that verify that the context (#3), the counter parameter (#4), and the value parameter (#5) are all set as expected.

The key to all of this is the following line from `loop()`:

```
fn.call(array, array[i], i)
```

that uses the callback's `call()` method to set array as the context, and pass the iteration element and loop counter as arguments.

Use of such a looping function is, perhaps arguably, syntactically cleaner than a traditional `for` loop, and can be of great utility for when you want to simplify the rote looping process, or segregate the inner code into its own function. Additionally, the callback is always able to access the original array via the `this` context, regardless of its actual source. This means that the reference even works on anonymous arrays, as in the following example:

```
loop([4,5,6], function(value,n){ /* whatever */ });
```

One thing that's especially important about the example of listing 3.18 is that it serves as a demonstration of the use of a function, passed as an argument, and being called in response to some action. Typically, these callbacks are declared as anonymous functions directly in the argument list of the function to which they are being passed.

Such callbacks are a staple of JavaScript libraries. Quite often callbacks are employed in relation to asynchronous, or nondeterministic, 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.

Let's look at one more practical use of precisely controlling function invocation.

3.4.5 Faking array methods

Let's imagine 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 arrays per se.

One option might be to create a new array every time you wish to create a new version of such an object and imbue it with our desired properties and methods – remember we can add properties and methods to an object as we please. Generally, however, this can be quite slow, not to mention tedious. Rather, let's examine the possibility of using a normal object and just giving it the functionality that we desire. For example, take a look at the code of Listing 3.19:

Listing 3.19: Simulating array-like methods

```
<script type="text/javascript">
  var elems = {
    length: 0,                                          #1
    add: function(elem){                                #2
      Array.prototype.push.call(this, elem);
    },
    find: function(id){                                 #3
      this.add(document.getElementById(id));
    }
  };

  elems.find("first");                                   #4
  assert(elems.length == 1 && elems[0].nodeType,        #4
         "Verify that we have an element in our stash");#4

  elems.find("second");                                 #4
  assert(elems.length == 2 && elems[1].nodeType,        #4
         "Verify the other insertion");                 #4
</script>
```

#1 Stores the count of elements
#2 Implements adding method
#3 Implements finding method
#4 Tests adding elements

In this example, we're creating a "normal" object and instrumenting it to mimic some of the behaviors of an array. First, we define a length property to record the number of element that are stored (#1); just like an array.

The we define a method to add an element to the end of our simulated array, calling this method simply `add()`(#2). Rather than writing our own code, we've decided to leverage a native object method of JavaScript arrays: `Array.prototype.push`. Normally, this method would operate on its context array, but here we are tricking the method to use *our* object as its context by using the `call()` method, and forcing our object to be the context of the `push()` method, which increments the `length` property (thinking that it's the `length` property of an array), and adds a numbered property to the object referencing passed element.

This behavior is almost subversive, in a way, but it's one that exemplifies what we're capable of doing with mutable object contexts.

Our `add()` method expects an element reference to be passed for storage. While there may be times that we have such a reference around, more often than not, we won't. So we also define a convenience method, `find()`, that looks up the element by its `id` value and adds it to storage (#3).

Finally we run two tests that each add an item to the object via `find()`, and check that the length was correctly adjusted, and that elements were added at the appropriate points. (#4). (These tests assume that elements with the `id` values of `first` and `second` exist on the page.)

The borderline nefarious behavior we demonstrated in this section not only reveals the power that malleable functions contexts gives us, it serves as an excellent segue into discussing the complexities of dealing with function arguments.

3.5 Variable-length argument lists

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. One of these flexible and powerful features of JavaScript is the ability for functions to accept an arbitrary number of arguments. This flexibility offers the developer great control over how their functions, and therefore their applications, can be written.

Let's take a look at a few prime examples of how we can use flexible argument lists to our advantage. We'll take a look at how to supply multiple arguments to functions that can accept any number of them, how to use variable-length argument lists to implement function overloading, and how to understand and use the length designation of argument lists.

3.5.1 Min/Max Number in an Array

Finding the smallest or the largest values contained within an array is an interesting problem. There is no predefined method for performing this action in the JavaScript language, so we'll need to write our own from scratch.

Initially, we'd likely think that it might be necessary to loop through the entire contents of the array, performing comparisons along the way, in order to determine the correct answers. However we have a trick up our sleeve. You see, the `call()` and `apply()` methods also exist as methods of the built-in JavaScript functions, And, some built-in methods are able to take any number of arguments.

For our purposes, the methods `Math.min()` and `Math.max()`,able to take any number of input arguments, can do most of the work for us. (And who isn't happy to find someone else to do the heavy lifting?) For example, all of the following uses of `Math.max()` to find the largest value of a set of numbers are valid:

```
var biggest = Math.max(1,2);
var biggest = Math.max(1,2,3);
var biggest = Math.max(1,2,3,4);
var biggest = Math.max(1,2,3,4,5,6,7,8,9,10,2058);
```

Note how this function is able to accept any number of values as input. Let's see how we can use that ability to our advantage in defining our array-inspecting functions. Consider listing 3.20.

Listing 3.20: Generic min and max functions for arrays

```
<script type="text/javascript">
  function smallest(array){                          //#1
    return Math.min.apply(Math, array);              //#1
  }                                                  //#1

  function largest(array){                           //#2
    return Math.max.apply(Math, array);              //#2
  }                                                  //#2

  assert(smallest([0, 1, 2, 3]) == 0,                //#3
         "Located the smallest value.");             //#3
  assert(largest([0, 1, 2, 3]) == 3,                 //#3
         "Located the largest value.");              //#3
</script>
```

#1 Implements function to find smallest
#2 Implements function to find largest
#3 Tests the implementations

In this code we define two functions; one to find the smallest value within an array (#1), and one to find the largest value (#2). Notice how both functions use the `apply()` method to supply the value in the passed arrays as variable-length argument lists to the `Math` functions.

A call to `smallest()`, passing the array `[0,1,2,3]` (as we did in our tests (#3)) results in a call to `Math.min()` that is functionally equivalent to:

```
Math.min(0,1,2,3);
```

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.

Now that we know how to use variables-length argument lists when calling functions, let's take a look at how we can declare our own functions to accept them!

3.5.2 Function overloading

Back in section 3.2, we mentioned the built-in `arguments` variable defined for all functions. We're now ready to take a closer look.

All functions are provided access to this important local variable, `arguments`., This variable will give our functions the power to handle any number of passed arguments. Even if we only ask for, or expect, a certain number of arguments, we'll always be able to access *all* passed arguments through the `arguments` variable.

Let's take a quick look at an example of using this power to implement effective function overloading.

DETECTING AND TRAVERSING ARGUMENTS

In other, more pure, object-oriented languages, method overloading is usually effected by declaring distinct implementations of methods of the same name with differing argument lists. That's not how it's done in JavaScript. In JavaScript, we're going to overload functions with a single implementation that modifies its behavior by inspecting the number and nature of the passed arguments.

In the following code, we're going to merge the contents of multiple objects into a single root object. This can be an essential utility for effecting multiple inheritance (which we'll discuss more when we talk about object prototypes in chapter 5).

Take a look at listing 3.21.

Listing 3-21: Traversing variable-length argument lists

```
<script type="text/javascript">
  function merge(root){                                    //#1
    for (var i = 1; i < arguments.length; i++) {
      for (var key in arguments[i]) {
        root[key] = arguments[i][key];
      }
    }
    return root;
  }

  var merged = merge(                                      //#2
    {name: "Batou"},                                       //#2
    {city: "Niihama"});                                    //#2

  assert(merged.name == "Batou",                           //#3
         "The original name is intact.");                  //#3
  assert(merged.city == "Niihama",                         //#3
         "And the city has been copied over.");            //#3
</script>
```

#1 Implements the merge function
#2 Calls the implemented functions
#3 Tests that it did the right things

The first thing that you will notice about the implementation of the `merge()` function (#1) is that its signature only declares a single parameter: `root`. This does *not* mean that we are limited to calling the function with a single parameter. Far from it! We can, in fact, call `merge()` with any number of parameters, including none. There is no proscription in JavaScript that enforces passing the same number of arguments to a function as there are declared parameters in the function declaration.

Whether the function can successfully deal with those (or no) arguments is entirely up to the definition of the function itself, but JavaScript imposes no rules in this regard.

The fact that we declared the function with a single parameter, `root`, means that only one of the possible passed argument can be referenced by name; the first one.

TIP If no argument that corresponds to a named parameter is passed, the named parameter can be checked for this with: `paramname === undefined`.

So we can get at the first passed argument via `root`, but how do we access the rest of any passed arguments? Why, the `arguments` variable, or course, which references an array of the passed arguments. (We'll find out that this isn't completely technically accurate in just a short while, but let's let that slide for now.)

Remember that what we're trying to do is to merge the properties of any object passed as the second through n^{th} arguments into the object passed as `root` (the first argument). So we iterate through the argument in the list, starting at index 1 in order to skip the first argument.

During each iteration, in which the iteration item is an object passed to the function, we loop through the properties of that passed object, and copy the located properties to the `root` object.

TIP If you haven't seen a `for-in` statement before, it simply iterates through all properties of an object, setting the property name (key) as the iteration item.

It should be clear at this point that this can be a powerful mechanism for creating complex and intelligent methods. Let's take a look at another example where the use of the `arguments` variable isn't so clean-cut.

SLICING AND DICING AN ARGUMENTS LIST

For our next example, we'll build a function that multiplies the first argument with the largest of the remaining arguments.

Seems simple enough – we'll grab the first argument, and multiply it by the result of using the `Math.max()` function, which we've already become familiar with, on the remainder of the argument values. Because we only want to pass the array that starts with the second element in the arguments list to `Math.max()`, we'll use the `slice()` method of arrays to create an array that omits the first element.

So, we go ahead and write up the code shown in listing 3.22.

Listing 3.22: Slicing the arguments list

```
<script type="text/javascript">
  function multiMax(multi){
    return multi * Math.max.apply(Math, arguments.slice(1));
  }

  assert(multiMax(3, 1, 2, 3) == 9, "3*3=9 (First arg, by largest.)");
</script>
```

But when we execute this script, we get a surprise as shown in figure 3.7.

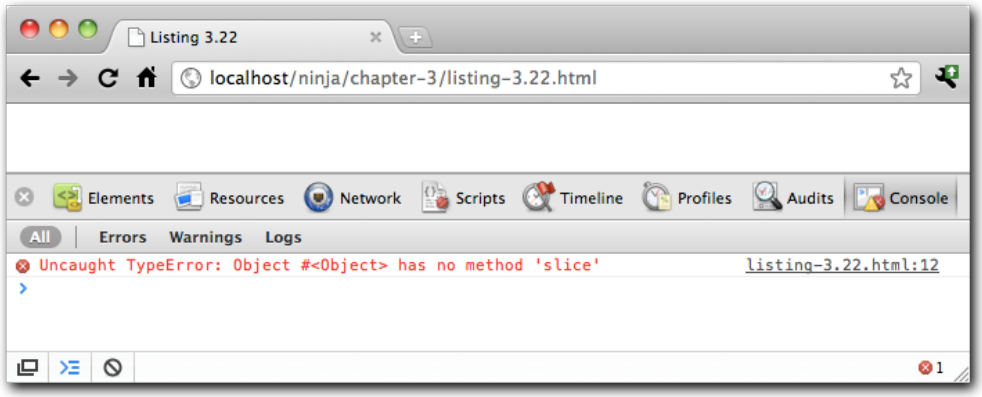

Figure 3.7 Something's rotten in the state of Denmark, and with our code!

What's up with that?

Well, as it turns out, the `arguments` variable doesn't reference a true array. Even thought it looks and feels a lot like one – we can iterate over it with a for loop, for example – it lacks all of the basic array methods, including `slice()`.

Well, we could create our own array by copying the value into a *true* array, but before we resort to that, recall the lesson of listing 3.19 in which we fooled an Array function to treat a non-array as an array. Let's use that knowledge and rewrite the code as shown in listing 3.23.

Listing 3.23: Slicing the arguments list – successfully this time

```
<script type="text/javascript">
  function multiMax(multi){
    return multi * Math.max.apply(Math,
      Array.prototype.slice.call(arguments, 1));          //#1
  }

  assert(multiMax(3, 1, 2, 3) == 9,
         "3*3=9 (First arg, by largest.)");

</script>
```

#1 Fools the `slice()` method

We use the same technique that we applied in listing 3.19 to coerce the Array's `slice()` method into treating the `arguments` "array" as a true array, even if it isn't one.

Segue needed

3.5.3 Function overloading

When it comes to operator overloading – the technique of defining a function that does different things base upon what is passed to it – it's easy to imagine that such a function could be easily implemented by inspecting the argument list using the mechanisms we've learned so far, and to simply perform different actions in `if-then` and `else-if` clauses. And often, that approach will serve us well, especially if the actions to be taken are on the simple side.

But once things start getting a bit more complicated, lengthy functions using many such clauses can quickly become unwieldy. In this section we're going to explore a technique by which we can create multiple functions seemingly with the same name, each differentiated from each other by the number of arguments they expect, that can be written as distinct and separate anonymous functions rather than as a monolithic `if-then-else-if` block.

All of this hinges on a little-known property of functions that we need to learn about first.

THE FUNCTION'S LENGTH PROPERTY

There's an interesting property on all functions that isn't very well known, but gives us an insight into how the function was declared: the `length` property. This property, not to be confused with the `length` property of the `arguments` variable, equates to the number of named parameters with which the function was declared. Thus, if we declare a function that accepts a single argument, it's `length` property will have a value of 1.

Examine the following code:

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

So within a function, we can determine two things about its arguments:

- How many named parameters it was declared with, via the `length` property.
- How many arguments were actually passed on the invocation, via `arguments.length`.

Let's see how this property can be used to building a function that we can use to create overloaded functions, differentiated by argument count.

OVERLOADING FUNCTIONS BY ARGUMENT COUNT

Yes, you read that correctly. We're going to create a function whose purpose is to declare other functions programmatically. More correctly, we could say that we're going to create a function that *binds* anonymous functions to a common name.

Let's say, for example, that we had an object upon which we wanted to define overloaded `find()` methods. Depending upon how many argument we passed to the `find()` method, we'd want it to act in different ways. Again, we could simply opt for if`-then-else-if` statements, but we're too suave for that. Besides, we've already noted that that can get messy quickly.

Keep your arms in the cart at all times, as this one's going to be a bit of a wild ride.

Let's look at Listing 3.24.

Listing 3.24 A function overloading function

```
function addMethod(object, name, fn) {
  var old = object[name];                          //#1
  object[name] = function(){
    if (fn.length == arguments.length)             //#2
      return fn.apply(this, arguments)             //#2
    else if (typeof old == 'function')             //#3
      return old.apply(this, arguments);           //#3
  };
}
```

#1 Stores previous function
#2 Invokes new function
#3 Invokes previous functions

Our `addMethod()` function accepts three arguments:

1. An object upon which a method is to be bound.
2. The name of the property to which the method will be bound.
3. The declaration of the method to be bound.

For example, the statement:

```
addMethod(myObject,'whatever',function(){ /* do something */ });
```

would bind the passed anonymous function as a method named `whatever` on `myObject`.

That's nothing special – we could have done that without a fancy function – but subsequently we could execute:

```
addMethod(myObject,'whatever',function(p1){ /* do something else */ });
```

that adds *another* method named `whatever` to `myObject` that, unlike our first method, accepts a single argument. The "magic" part is that if we call `myObject.whatever()`, the first method will be called, but if we call `myObject.whatever(1)`, the second method would be called.

How can this be? Let's find out.

In the `addMethod()` function, the first thing that we do is store a copy of any function that is already declared in the property identified by `name` in a variable named `old` (`#1`). We then redefine that property (the one identified by `name`) with an anonymous function of our own design.

This is the function that will be called when we invoke the method by the name that we passed in. So when we call `myObject.whatever()` (with any number of arguments) our inner anonymous function will be invoked.

Within this anonymous function we check the number of arguments actually *passed* to the invocation, and if it matches the number of parameters *expected* by the passed function, it is invoked. If not, the previously stored function in `old` is invoked (assuming it is a function).

There's a bit of sleight of hand going on here with regard to how the inner function accesses `old` and `fn`, that involves a concept called closures, which we'll be taking a close

look at in the next chapter. For now, just accept that when it executes, the inner function has access to `old` and `fn`.

Let's test our new function as shown in Listing 3.25

Listing 3.25Testing the addMethod function

```
<script type="text/javascript">
  var ninjas = {                                                    #1
   values: ["Dean Edwards", "Sam Stephenson", "Alex Russell"]
  };

  addMethod(ninjas, "find", function(){                             #2
    return this.values;
  });

  addMethod(ninjas, "find", function(name){                         #3
    var ret = [];
    for (var i = 0; i < this.values.length; i++)
      if (this.values[i].indexOf(name) == 0)
        ret.push(this.values[i]);
    return ret;
  });

  addMethod(ninjas, "find", function(first, last){                   #4
    var ret = [];
    for (var i = 0; i < this.values.length; i++)
      if (this.values[i] == (first + " " + last))
        ret.push(this.values[i]);
    return ret;
  });

  assert(ninjas.find().length == 3,                                  #5
         "Found all ninjas");
  assert(ninjas.find("Sam").length == 1,
         "Found ninja by first name");
  assert(ninjas.find("Dean", "Edwards").length == 1,
         "Found ninja by first and last name");
  assert(ninjas.find("Alex", "X", "Russell") == null,
         "Found nothing");
</script>
```

#1 Declares base object with test data
#2 Binds no-argument method
#3 Binds single-argument method
#4 Binds dual-argument method
#5 Tests bound methods

To test our method-overloading function, we define a base object (#1), containing some test data consisting of well-known JavaScript ninjas, to which we will bind three methods, all with the name `find`. The purpose of all these methods will be to find a ninja based upon criteria passed to the methods.

We declare and bind three versions of a `find` method:

1. One expecting no arguments (#2) that returns all ninjas.
2. One that expects a single argument (#3) that returns any ninjas whose name contains the passed text.
3. One that expects two arguments (#4) that returns any ninjas whose first and last name matches the passed strings.

Loading this page to run the test shows that we have succeeded as shown in figure 3.8

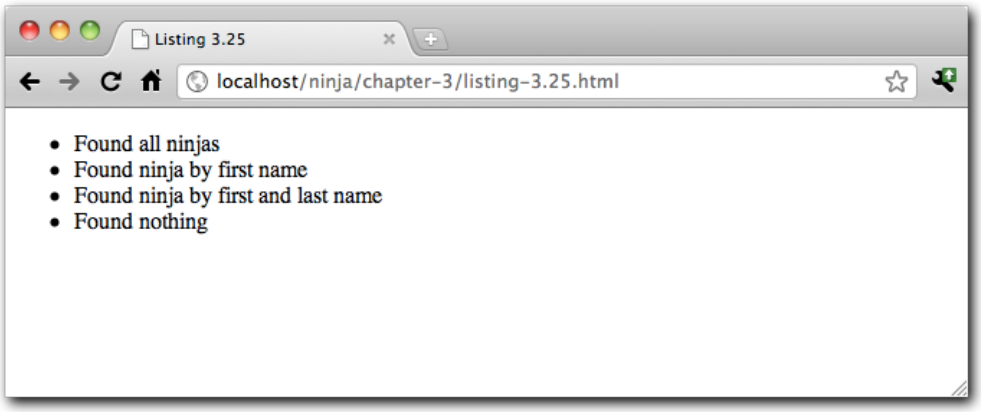

Figure 3.8 Ninjas found! All using the same overloaded method name `find()`

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

It should be noted that there are some caveats to be aware of 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. Which is frequently what we will want to do.

• Such overloaded methods will have some function call overhead. Thus, we'll want to take that into consideration in high performance situations.

Nonetheless, this function provides a good example of some functional techniques, as well as an opportunity to introduce the `length` property of functions.

So far in this chapter, we've seen how functions are treated as first-class objects by JavaScript, now let's look at one more thing we might do to functions-as-objects: checking to see of an object is a function.

3.6 Checking for functions

To close out our look at functions in JavaScript, let take a look at how we can detect when an object is an instance of a function, and therefore something that can be called; a seemingly simple task, but not with out its cross-browser issues.

Typically the `typeof` statement is more than sufficient to get the job done in most cases; for example in the following code:

```
function ninja(){}

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

This should be the typical way that we check if a value is a function, and this will always work if what we are testing is indeed a function. However there exist a few cases where this test may yield some *false-positives* that we need to be aware of:

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

For situations in which these specific cases cause our code to trip up, 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 are 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 Explorer's problematic functions. One technique that *does* work fairly well is to convert the function to a string and determine its type based upon its serialized value, as in the following code:

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

Even this function isn't perfect, but in situations like the above, it'll pass all the cases that we listed, giving us a correct value to work with.

There is one notable exception however (isn't there always?): Internet Explorer reports methods of DOM elements with a type of "object", like so: `typeof`

`domNode.getAttribute == "object"` and `typeof inputElem.focus == "object"` – so this particular technique does not cover this case.

The implementation of `isFunction()` requires a little bit of magic in order to make it work correctly. We access 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 an 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 directly call `fn.toString()` to try and get this result is two-fold:

1. Individual objects are likely to have their own `toString()` implementation.
2. Most types in JavaScript already have a pre-defined `toString()` method that overrides the method provided by `Object.prototype`.

By accessing the `Object.prototype` method directly we ensure that we're not getting an overridden version of `toString()` and we 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 us to painlessly write cross browser applications with little concern for the painful minutia. We'll explore lots more cross-browser strategies as we go along, and especially in chapter 10.

3.7 Summary

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

Specifically, within this chapter, we took a look at different techniques for defining functions, along with how these different techniques can benefit clarity and organization. We also examined anonymous functions, and how recursion can be a tricky problem, looking at how to mitigate these issues 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 so.

We also worked up an understanding of how function contexts work, and how they can be manipulated using the `apply()` and `call()` methods of functions themselves.

We looked at a number of examples dealing with a variable number of arguments to functions, in order to make them more powerful and versatile. We also looked at binding overloaded functions based upon the number of passed and expected arguments.

Finally, we closed with an examination of detecting when an object is a function, and the cross-browser issues that relate to that.

In all, we made 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.

4

Closing in on Closures

In this chapter:

- What are closures and how do they work?
- Using closures to simplify development
- improving the speed of code using closures
- Solving common scoping issues with closures

Closely tied to the functions that we learned all about in the previous chapter, closures are a defining feature of JavaScript. While scores of page authors get along writing on-page script without their benefit, the use of closures can not only reduce the amount and complexity of the script necessary to add advanced features to our pages, but also allow us to do things that would simply not be possible, or simply too complex to be feasible, without them. The landscape of the language, and how we write our code using it, is forever shaped by the inclusion of closures.

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

In this chapter we'll learn what closures are all about, and how to use them to elevate our on-page script to world-class levels.

4.1 How closures work

Simply put, a closure is the scope created when a function is declared that allows the function to access and manipulate variables that are external to itself. Put another way,

closures allow a function to access all the variables as well as other functions, that are declared in the same scope that the function itself is declared.

That may seem rather intuitive until you remember that a declared function can be called at any later time, even *after* the scope in which it was declared has gone away.

This concept is probably best explained through code, so let's start small with listing 4.1.

Listing 4.1: A simple closure

```
<!DOCTYPE html>
<html>
  <head>
    <title>Listing 4.1</title>
    <script type="text/javascript" src="../scripts/assert.js"></script>
  </head>
  <body>
    <ul id="results"></ul>
  </body>
  <script type="text/javascript">

    var outerValue = 'ninja';                                    #1

    function outerFunction() {
      assert(outerValue == "ninja","I can see the ninja.");      #2
    }

    outerFunction();                                             #3

  </script>
</html>
```

#1 Defines a value
#2 Declares a function body
#3 Executes function

In this code example, we declare a variable (#1) and a function (#2) in the same scope. Afterwards, we cause the function to execute (#3). As can be seen in figure 4.1, the function as able to "see" and access the `outerValue` variable.

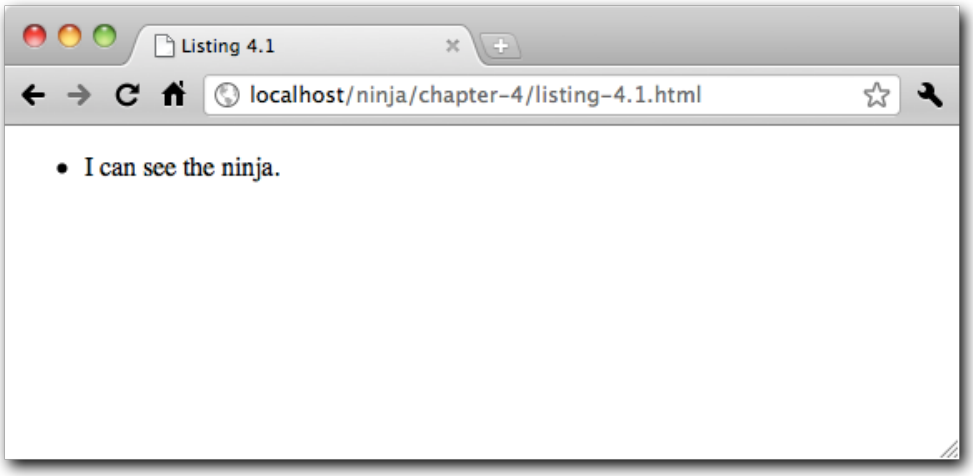

Figure 4.1: Our function has found the ninja, which is hiding in plain sight

You've likely written code such as this hundreds of times without realizing that you were creating a closure!

Not impressed? I guess that's not surprising. Because both of the outer value and the outer function are declared in global scope, that scope never goes away (as long as the page is loaded) and it's not surprising that the function can access the variable as it's still in scope and viable. So even though the closure exists, its benefits aren't yet clear.

Let's spice it up a little as shown in listing 4.2.

Listin 4.2: A not-so-simple closure

```
<script type="text/javascript">

  var outerValue = 'ninja';

  var later;                                                    #1

  function outerFunction() {
    var innerValue = 'samurai';                                 #2

    function innerFunction() {                                  #3
      assert(outerValue,"I can see the ninja.");                #3
      assert(innerValue,"I can see the samurai");               #3
    }                                                           #3

    later = innerFunction;                                      #4
  }

  outerFunction();                                              #5
```

```
  later();                                                    #6
</script>
```

#1 Declares an empty variable
#2 Declares a value inside the function
#3 Declares an inner function
#4 Invokes the outer function
#5 Invokes the inner function

The first thing we added to our example is an uninitialized variable named `later` (#1) that we'll use, well, later. Then we added a variable *inside* the outer function (#2). This limits the scope to the body of `outerFunction`.

We then declare a new function inside the outer function (#3) which accessing both of the variables that we've declared. Yes, we can do that. Remember that functions are first-class objects and can be created just about anywhere a variable can.

We also added an assignment of the inner function to the `later` variable, so that we can call it later.

With that set up, we run our test. We call our outer function (#5), which causes the inner function to be declared and to be assigned to `later` (in the global scope), and then call the inner function from the global scope.

What do we expect to happen?

Certainly the inner function has access to `outerValue` – it's declared in the global scope, after all – but what about `innerValue`? `innerValue` was declared inside the scope of `outerFunction` and is *not* available from the global scope where we call the inner function through the reference in `later` (#6). Surely that will spark some sort of error!

But when we execute the test, we see the display of figure 4.2.

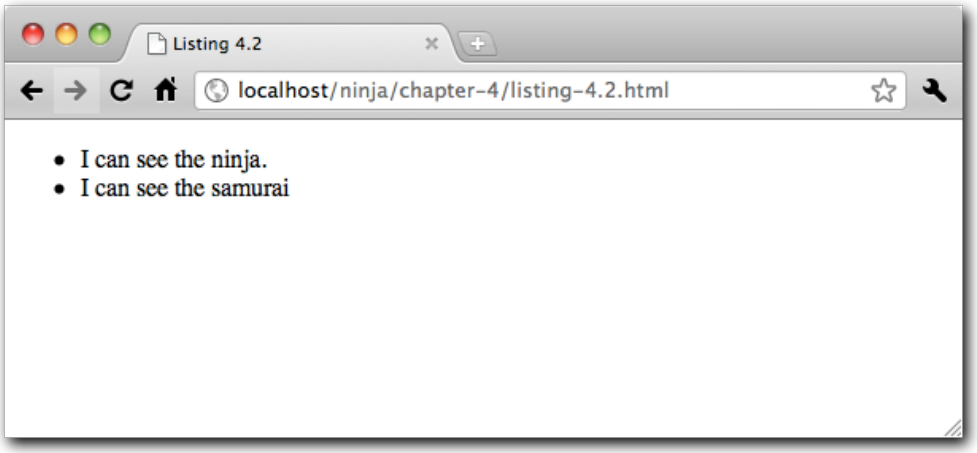

Figure 4.2: Despite trying to hide inside a function, the samurai has been spied

How can that be? The answer, of course, is closures.

When we declared `innerFunction` inside the outer function, not only was the function declaration defined, but a closure was also defined that encompasses the function declaration, but also all variables in scope *at the time of the declaration*.

When `innerFunction` eventually executes, even if it is executed *after* the scope in which it is declared goes away, it has access to the original scope in which it was declared through its closure.

Let's augment the example with a few more additions to observe a few core principles of closures. See the code of listing 4.3.

Listing 4.3 What else closures can see

```
<script type="text/javascript">

  var outerValue = 'ninja';
  var later;

  function outerFunction() {
    var innerValue = 'samurai';

    function innerFunction(paramValue) {                        #1
      assert(outerValue,"Inner can see the ninja.");
      assert(innerValue,"Inner can see the samurai");
      assert(paramValue,"Inner can see the wakizashi");         #2
      assert(tooLate,"Inner can see the ronin");                #2
    }

    later = innerFunction;
  }

  assert(!tooLate,"Outer can't see the ronin");                 #3

  var tooLate = 'ronin';                                        #4

  outerFunction();
  later('wakizashi');                                           #5

</script>
```

#1 Accepts a parameter
#2 Tests late variable and parameter
#3 Looks for a later value
#4 Declares a value after the function
#5 Passes parameter

To our previous code we have added a number of interesting additions. We added a parameter (#1) to the inner function, and pass a value to the function when it is invoked through `later` (#4). We also added a variable that declared after the outer function declaration (#3).

When the tests inside the inner function execute (#2), we can see the display of figure 4.3.

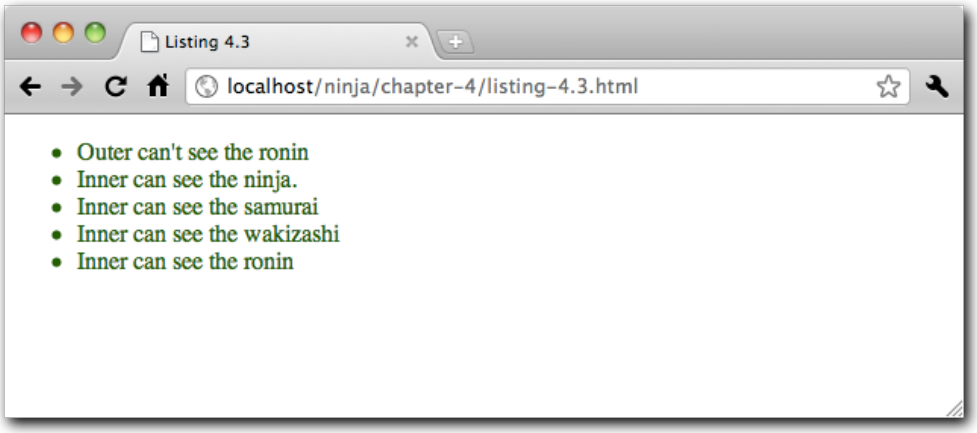

Figure 4.3: Turns out that inner can see farther than outer

This shows three more interesting concepts regarding closures:

1. Function parameters are included in the closure of that function.
2. All variables in an outer scope, even those declared after the function declaration, are included.
3. Within the same scope, variables not yet defined cannot be forward referenced.

Points 2 and 3 explain why the inner closure can see variable `tooLate`, but the outer closure cannot.

It's important to note that while all of this information isn't readily visible anywhere (there's no "closure" object that you can inspect that's holding all of this information) there is a direct cost to storing and referencing information in this manner. It's important to remember that each function that accesses information via a closure has a "ball and chain," if you will, attached to them carrying this information around. So while closures are incredibly useful, they certainly aren't free of overhead. All that information needs to be held in memory until it's absolutely clear to the JavaScript engine that it will no longer be needed, or the page unloads.

4.2 Putting closures to work

Now that we understand what a closure is and how it works (at least at a high level), let's see how we can actually put them to work on our pages.

4.2.1 Private Variables

A 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 approximation, as demonstrated by the code of listing 4.4.

Listing 4.4: Using closures to approximate private variables

```
<script type="text/javascript">

  function Ninja() {                                          #1
    var slices = 0;                                           #2

    this.getSlices = function(){                              #3
      return slices;                                          #3
    };                                                        #3

    this.slice = function(){                                  #4
      slices++;                                               #4
    };                                                        #4
  }

  var ninja = new Ninja();                                    #5

  ninja.slice();                                              #6

  assert(ninja.getSlices() == 1,                              #7
         "We're able to access the internal slice data." );   #7
  assert(ninja.slices === undefined,                          #7
         "And the private data is inaccessible to us." );     #7
</script>
```

#1 Defines constructor
#2 Declares "private" variable
#3 Creates an accessor method
#4 Declares increment method
#5 Constructs the object
#6 Calls the increment method
#7 Tests the private-ness

In listing 4.4 we create a function that is to serve as a constructor (#1). We're getting a little bit ahead of ourselves here, as *constructors* are a topic we'll be looking at in chapter 5, but it serves our purpose to peek ahead at them a bit here. For now, just be aware that when using the `new` keyword on a function (#5), a new Object instance is created and the function is called, with that new object as its context, to serve as a constructor to that object. So `this` within the function is a newly instantiated object.

Within the constructor, we define a variable to hold state, `slices` (#2). The scope of this variable limits its accessibility to within the constructor. To give access to the value of variable from outside the scope, we define an *accessor* method, `getSlices()`, which can be used to read, but not write the private variable.

An implementation method, `slice()`, was added to give us control over the value of the variable in a controlled fashion (#4). In a real-world application this might be some business method; in this example, it merely increments the value of `slices`.

Outside of the constructor, we invoke it with the `new` operator (#5), and then call the `slice()` method (#6).

Our tests (#7) show that we can use the accessor method to obtain the value of the private variable, but that we cannot access it directly. This effectively prevents us from being able to make uncontrolled changes to the value of the variable, just as if it were a private variable in a fully object-oriented language.

State can maintained within the function, without letting it be directly accessed by a user, because the variable is available to the inner methods via their closures, but not to code that lies outside the constructor.

Now let's focus on another common use of closures.

4.2.2 Callbacks and Timers

Another one of the most common places we can use closures is when we're dealing with callbacks or timers. In both cases a function is being asynchronously called at an unspecified later time, and within the functions we have the need to access outside data.

Closures act as an intuitive way of accessing that data, especially when we 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 4.5.

Listing 4.5: Using closures from a callback for an Ajax request

```
<div></div>

<script>
  var elem$ = jQuery("div");                                    #1
  elem$.html("Loading...");

  jQuery.ajax({
    url: "test.html",
    success: function(html){                                    #2
      assert(elem$,
             "We can see elem$, via the closure for this function.");
      elem$.html(html);
    }
  });
</script>
```

#1 Declares a variable
#2 Creates a callback and closure

There're a number of interesting things going on in listing 4.5. We start with an empty `<div>` element, which we want to load with the text "Loading…" while an Ajax request is under way that will fetch new content from the server to load into that `<div>` when the response returns.

We need to reference the `<div>` element twice: once to preload it, and once to load it with the server content. We *could* look up a reference to the `<div>` element each time, but we want to be stingy regarding performance so we'll just look it up once, and store it away in a variable named `elem$` (`#1`) (using the `$` sign as a suffix is a jQuery convention to indicate that the variable holds a jQuery wrapped set reference).

Within the call to the jQuery `.ajax()` method, we define an anonymous function (`#2`) to serve as the response callback. Within this callback, we reference the `elem$` variable via the closure, and use it to stuff the response text into the `<div>`.

That was pretty straightforward. Let's look at a slightly more complicated example in listing 4.6, which creates a simple animation.

Listing 4.6: Using a closure in a timer interval callback

```
<div id="box" style="position:absolute;">Box!</div>            #1
<ul id="results"></ul>

<script type="text/javascript">
  var elem = document.getElementById("box");                   #2
  var tick = 0;                                                #3

  var timer = setInterval(function(){                          #4
    if (tick < 100) {
      elem.style.left = elem.style.top = tick + "px";
      tick++;
    }
    else {
      clearInterval(timer);
      assert(tick == 100,                                      #5
             "Tick accessed via a closure.");
      assert(elem,
             "Element also accessed via a closure.");
      assert(timer,
             "Timer reference also obtained via a closure." );
    }
  }, 10);

</script>
```

#1 Creates element to be animated
#2 References animation element
#3 Counts animation ticks
#4 Creates and starts timer
#5 Tests closure

Loading the example into a browser, we see the display of figure 4.4 when the animation has completed.

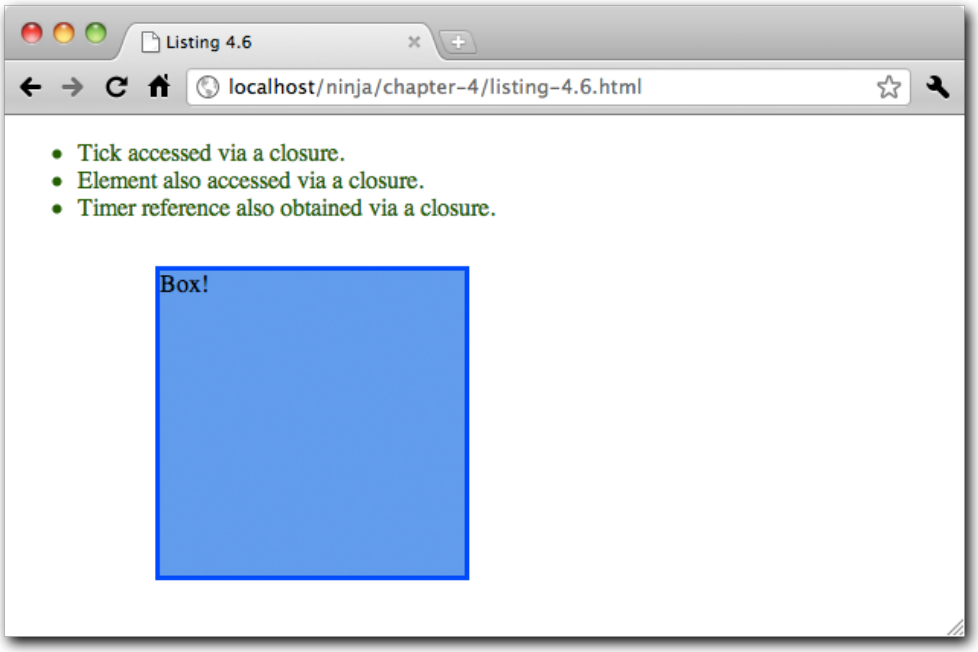

Figure 4.4: Closures used to keep track of the steps of an animation

What's especially interesting about the code of listing 4.4 is that it uses a single anonymous function (#4) to accomplish the animation, which accesses three variables, accessed via a closure, to control the animation process.

This structure is particularly important as the three variables (the reference to the DOM element (#2), the tick counter (#3), and the timer reference (#4)) all must be maintained across the steps of the animation. This example is a particularly good one for demonstrating how the concept of closures is capable of producing some surprisingly intuitive and concise code.

Now that we've seen closures used in various callbacks, let's take a look at some of the other ways in which they can be applied, starting with using them to help us bend function contexts to our will.

4.3 Binding function contexts

In the previous chapter, during our discussion of function contexts, we saw 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 the code of listing 4.7 in which an object method is bound to an element as an event listener.

Listing 4.7: Binding a specific context to a function

```
<button id="test">Click Me!</button>                                    #1

<script>
  var button = {                                                        #2
    clicked: false,
    click: function(){                                                  #3
      this.clicked = true;
      assert(button.clicked,"The button has been clicked");             #4
    }
  };

  var elem = document.getElementById("test");                           #5
  elem.addEventListener("click",button.click,false);                    #5

</script>
```

#1 Creates a button element
#2 Defines a backing object
#3 Declares the click handler
#4 Tests the state
#5 Establishes the click handler

In this example, we have a button (#1) and we want to know whether it has ever been clicked or not. In order to retain that state information, we create a "backing object" (#2) in which we will store the clicked state, as well as define an event handler (#3) that will fire when the button is clicked in order to record the click.

Within the handler, which we establish as a click handler for the button (#5), we set the clicked property to `true`, and then test that the state was properly recorded in the backing object.

But when we load the example into a browser and click the button, we see by the display of figure 4.5 that something is amiss (the stricken text indicates that the test has failed).

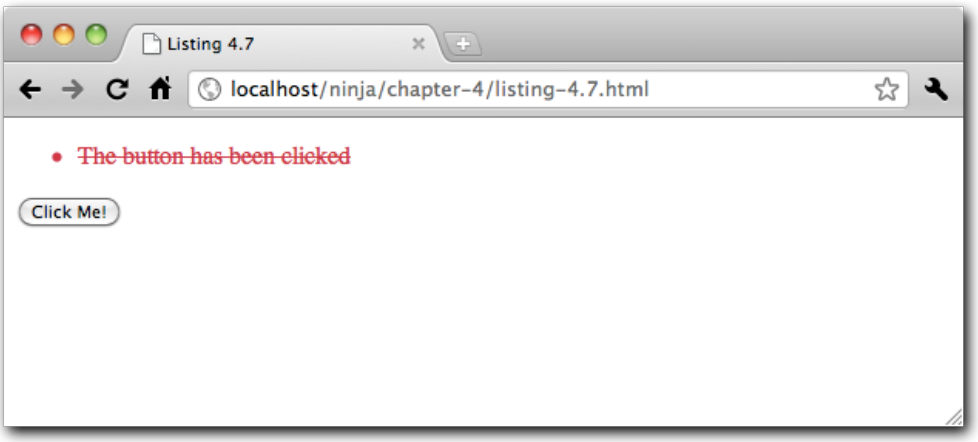

Figure 4.5; Why did our test fail? Where did the change of state go?

The code of listing 4.5 fails because the context of the `click` function is not referring to the `button` object as we intended. Recalling the lessons of chapter 3, if we were to have called the function via:

```
button.click()
```

the context would indeed have been button. But in our example, `addEventListener` redefines the context to be the target element of the event, which causes the context to be the <button> element, not the button object. So we set our state on the wrong object!

Setting the context to the target element is a perfectly reasonable default, and one that we can count on in many situations, but in this instance, it's in our way. Luckily, closures give us a way around this.

We can force a particular function invocation to always have a desired context using a mix of anonymous functions, `.apply()`, and closures. Take a look at the code of listing 4.8, which updates the code of listing 4.7 with additions to bend the function context to our wills.

Listing 4.8: Binding a specific context to an event handler

```
<script>
  function bind(context,name){                                    #1
    return function(){                                            #1
      return context[name].apply(context,arguments);              #1
    };                                                            #1
  }                                                               #1

  var button = {
    clicked: false,
    click: function(){
      this.clicked = true;
      assert(button.clicked,"The button has been clicked");
      console.log(this);
    }
  };

  var elem = document.getElementById("test");
  elem.addEventListener("click",bind(button,"click"),false);      #2

</script>
```

#1 Defines binding function
#2 Binds a specific context to the handler

The secret sauce that we've added here is the `bind()` method (#1) This method is designed to create and return a new anonymous function that calls the original function enforcing the context to be the object that we pass to `bind()`. This context, along with the information of what to call as the end function, is remembered though the anonymous function's closure, which includes the parameters passed to `bind()`.

Later, when we establish the event handler, we use the `bind()` method (#2) to specify the event handler rather than using `button.click` directly.

This particular implementation of a binding function makes the assumption that we're going to be using an existing method (a function attached as a property) of an object, and that we want that object to be the context. With that assumption, `bind()` only needs two pieces of information: a reference to the object containing the method, and the name of the method.

This `bind()` function is a simplified version of a function popularized by the Prototype JavaScript Library, which promotes writing code in a clean and classicial object-oriented manner. Thus, most object methods have this particular "incorrect context" problem when established as event handlers. The original version of the method looks something like the code in Listing 4.9.

Listing 4.9: An example of the function binding code used in the Prototype library

```
Function.prototype.bind = function(){                              #1
  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" );
```

#1 Adds method to all functions

Note that this method is quite similar to the function we implemented in Listing 4.8, but with a couple of notable additions. To start, it attaches itself to all functions, rather than presenting itself as a globally-accessible function (#1) by adding itself as a property of the `prototype` of JavaScript's Function. We'll be exploring prototypes in chapter 5, but for now just think of it as the central blueprint for a JavaScript type.

We would use this function, bound as a method to all functions (via the prototype) like so: `myFunction.bind(myObject)`. Additionally, with this method, we are able to bind arguments to the anonymous function. This allows us to pre-specify some of the arguments, in a form of partial function application (which we'll discuss in the very next section).

It's important to realize that Prototype's `.bind()` (or our own) isn't meant to be a replacement for methods like `.apply()` or `.call()`. Remember, the underlying purpose is controlling the context for delayed execution via the anonymous function and closure. This

important distinction makes them especially useful for delayed execution callbacks for event handlers and timers.

Now, what about those pre-filled function arguments we mentioned earlier?

4.4 Partially applying functions

"Partially applying" a function is a particularly interesting technique in which we can prefill arguments, to a function before it is even executed. In effect, partially applying a function returns a new function, which we can call. This technique of filling in the first couple of arguments of a function (and returning a new function) is typically called ***currying***.

As usual, this is best understood through examples. Before we look at how we'll actually implement currying, let's look at how we might want to use it.

Let's say that we wanted to split a CSV (comma-separated value) string into its component parts, ignoring extraneous whitespace. We can easily do that with the String's split() method, supplying an appropriate regular expression:

```
var elements = "val1,val2,val3".split(/,\s*/);
```

But having to remember that regular expression all the time can be tiresome. So let's implement a `csv()` method to do it for us. And let's imagine how we'd like to do it using currying, as shown in listing 4.10.

Listing 4.10: Partially applying arguments to a native function

```
String.prototype.csv = String.prototype.split.partial(/,\s*/);    //#1

var results = ("Mugan, Jin, Fuu").csv();                           //#2

assert(results[0]=="Mugan" &&                                      //#3
       results[1]=="Jin" &&                                        //#3
       results[2]=="Fuu",                                          //#3
       "The text values were split properly");                     //#3
```

#1 Creates a new String function
#2 Invokes the curried function
#3 Tests the results

In listing 4.8 we've taken the String's `.split()` method and have imagined a `partial()` method (not yet implemented) that we can use to prefill the regular expression upon which to split (`#1`). The result is a new function named `.csv()` that we can call at any point to convert a list of comma-separated values into an array without having to deal with messy regular expressions.

With that in mind, let's look at how a curry method is, more or less, implemented in the Prototype library in listing 4.11.

Listing 4.11: 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);   #1
  return function() {                                            #2
    return fn.apply(this, args.concat(
      Array.prototype.slice.call(arguments)));
```

```
  };
};
```

#1 Remebers "prefill" arguments
#2 Create anonymous curried function

This technique is another good example of using a closure to remember state. In this case we want to remember the arguments to be prefilled (#1) and transfer them to the newly constructed function (#2). This new function will have the filled-in `arguments` and the new arguments concatenated together and passed. The result is a method that allows us to prefill arguments, giving us a new, simpler function that we can use.

While this style of partial function application is perfectly useful, we can do better. What if we wanted to fill in any missing argument from a given function, not just those at the beginning of the argument list?

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 4.12 shows a possible implementation.

Listing 4.12: A more-complex "partial" function

```
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 Prototype's `.curry()` method, but has a couple of important differences. Notably, the user can specify arguments anywhere in the parameter list that will be filled in later by specifying the `undefined` value for "missing" arguments. To accommodate this we have increased the abilities of our arguments-merging technique. Effectively, we loop through the arguments that are passed in and look for the appropriate gaps (the `undefined` values), filling in the missing pieces as we go along.

Thinking back to the example of constructing a string splitting function,, let's look at some other ways in which this new functionality could be used. To start we could construct a function that is able to be easily delayed:

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

delay(function(){
  assert(true,
         "A call to this function will be delayed 10 ms.");
});
```

This snippet creates a new function, named `delay()`, into which we can pass another function to have it be called asynchronously after 10 milliseconds.

We could, also create a simple function for binding events:

```
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 simpler function call.

Up to this point, we've used closures to reduce the complexity of our code, handily demonstrating some of the power that functional JavaScript programming has to offer.

Now let's continue to explore using closures in code to add advanced behaviors and further simplifications.

4.5 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.

Remember memorization from chapter 3? Let's take another look.

4.5.1 Memoization

As we learned in chapter 3, memoization is the process of building a function that is capable of remembering its previously computed answers. As we demonstrated in that chapter, it's pretty straightforward to introduce memorization 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()`, shown in Listing 4.13, that we can use to remember return values from an existing function. This implementation does not involve closures.

Listing 4.13: A memorization method for functions.

```
<<script type="text/javascript">

  Function.prototype.memoized = function(key){
    this._values = this._values || {};                          #1
    return this._values[key] !== undefined ?                    #2
      this._values[key] :
      this._values[key] = this.apply(this, arguments);
  };

  function isPrime( num ) {                                     #3
```

```
    var prime = num != 1;
    for ( var i = 2; i < num; i++ ) {
      if ( num % i == 0 ) {
        prime = false;
        break;
      }
    }
    return prime;
  }

  assert(isPrime.memoized(5),                                    #4
         "The function works; 5 is prime.");                     #4
  assert(isPrime._values[5],                                     #4
         "The answer has been cached.");                         #4

</script>
```

#1 Creates cache if non exists
#2 Returns cached or computer value
#3 Defines memoizable functions
#4 Tests the function and cache

In this code, we use the familiar `isPrime()` function (#3) from the previous chapter, and 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, we can add a new `.memoized()` method to all functions that gives us the ability to wrap the functions and attach properties that are associated with the function itself. This will allow us to create a data store (cache) in which all of our pre-computed values can be saved. Let's see how that 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 expression (#1):

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

If the `_values` property already exists, then we just re-save that reference to the property, otherwise we create the new data store (a simple object) and store its reference in the `_values` property.

When we call a function through this method, we look into the data store (#2) to see if the stored value already exists and if so, return that value. Otherwise, we compute the value and store it in the cache for any subsequent calls.

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.

Testing the code (#4) shows that we can compute values, and that the value is cached.

The problem with what we've come up with so far, is that a caller of the `isPrime()` function must remember to call it through its .memoized() method in order to reap the benefits of memoization. That won't do at all.

With the memoizing method at our disposal to monitor 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 memorized automatically (without the caller having to do anything weird like remember to call `.memoized()`), as shown in Listing 4.14.

Listing 4.14: A technique for memorizing functions using closures

```
<script type="text/javascript">

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

  Function.prototype.memoize = function(){
    var fn = this;                                    #1
    return function(){                                #2
      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();

</script>
```

#1 Brings context into closure
#2 Wraps original function in memoization

Listing 4.14 builds upon our previous example, in which we created the `memoized()` method, adding yet another new method, `memoize()`. This method returns a function which wraps the original function with the `memoized()` method applied, such that it will always return the memoized version of the original function (#2). This eliminates the need for the caller to apply `memoized()` themselves.

Note that within the `memoize()` method we construct a closure remembering the original function (obtained via the context) that we want to memorize (#1) by copying the context into a variable. This is a common technique: each function has its own context, so contexts are never part of a closure. But context values can become part of a closure by establishing

a variable reference to the value. By remembering the original function we can return a new function which will always call our `memoized()` method; giving us direct accessto the memorized instance of the function.

In Listing 4.14 we also show a comparatively strange means of defining a new function when we define `isPrime()`. As we want `isPrime` to always be memorized, we need to construct a temporary function whose results won't be memorized. We take this anonymous, prime-figuring function and memoize it immediately, giving us a new function which is assigned to the `isPrime` variable. We'll discuss this construct in depth, in section 4.5.3. 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 4.14 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 counteract this. We'll discuss this matter in depth later in the book.

4.5.2 Function Wrapping

Function wrapping is a technique for encapsulating the functionality of a function, while overwriting it with new or extended functionality, in a single step. It is best used when we wish to override some previous behavior of a function, while still allowing certain use cases to still execute.

A common use is when implementing pieces of cross-browser code in situations where a deficiency in a browser must be accounted for. Consider, for example, working around bug in Opera's implementation of access title attributes. In the Prototype library, the function wrapping technique is employed to work around this bug.

As opposed to having a large if/else block within its `readAttribute()` function (a technique that is debatably messy and not a particularly good separation of concerns) Prototype, instead opted to completely override the old method using function wrapping and deferring the rest of the functionality to the original function.

Let's take a look at that in listing 4.15.

Listing 4.15: Wrapping an old function with a new piece of functionality

```
function wrap(object, method, wrapper) {
  var fn = object[method];                                    #1
  return object[method] = function() {                        #2
    return wrapper.apply(this, [ fn.bind(this) ].concat(
        Array.prototype.slice.call(arguments)));
  };
}

// Example adapted from Prototype
if (Prototype.Browser.Opera) {                                #3
  wrap(Element.Methods, "readAttribute",
```

```
        function(original, elem, attr) {
    return attr == "title" ?
        elem.title :
        original(elem, attr);
  });
}
```

#1 Remembers original function
#2 Returns new wrapper function
#3 Employs function wrapping

Let's dig in to how the `wrap()` function works, which is is passed a base object, the name of the method in that object to wrap, and the new wrapper function. To start, we save a reference to the original method in `fn` (#1); we'll access it later via the anonymous function's closure. We then proceed to overwrite the method with a new function (#2). This new function executes the passed wrapper function (brought to us via a closure), passing it an augmented arguments list. We want the first argument to be the original function that we're overriding, so we create an array containing a reference to the original function (whose context is bound, using the `bind()` method that we created in section 4.3, to be the same as the wrapper's), and add the original arguments to this array. The `apply()` method, as we've seen, uses this array as the argument list..

Prototype uses the `wrap()` function to override an existing method (in this case `readAttribute`) replacing it with a new function (#3). 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 overridden while still retaining access to the original functionality.

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

Now let's look at an often-used syntax that looks odd, but is an important part of functional programming.

4.6 Immediate functions

A large part of advanced functional JavaScript, not to mention the very use of closures, centers around one simple construct:

```
(function(){})()
```

This single pattern of code is incredibly versatile and ends up giving the JavaScript language a ton of unforeseen power. But as its syntax, with all those braces and parentheses, may seem a little strange, let's deconstruct what's going on step by step.

First, let's ignore the contexts of the first set of parentheses, and examine the construct:

```
(...)()
```

We know that we can call any function using the `functionName()` syntax, but in place of the function name, we can use any expression that references a function instance. That's why we could call a function from a variable that refers to a function using the `variableName()` syntax. As in other expressions, if we want an operator (in this case the

function call operator `()` to be applied to an entire expression, we'd enclose that expression in a set of parentheses. Consider how the expressions `(3 + 4) * 5` and `3 + (4 * 5)` differ from each other.

So in `(…)()`, the first set of parentheses is merely a set of delimiters enclosing an expression while the second set is an operator. It's just a bit confusing that each set of parentheses has a very different meaning. If the function call operator were something like `~~` rather than `~~`, the expression `(…)~~` would likely be less confusing.

In the end, whatever is within the fist set of parentheses will be expected to be a reference to a function to be executed.

The following is also valid:

```
(functionName)()
```

Although the first set of parentheses are not needed in this case.

Now, rather than a function name, if we provided an anonymous function within the first set of parentheses, we end up with the syntax:

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

where the anonymous function has no body. Providing a body, the syntax expands to:

```
(function(){
  statement-1;
  statement-2;
   …
  statement-n;
})();
```

The result of this code is an expression that creates, executes, and discards a function all in the same statement. Additionally, since we're dealing with a function that can have a closure as any other, we also have access to all outside variables in the same scope as the statement. As it turns out, this simple construct, called an ***immediate function***, ends up becoming immensely useful, as we'll soon see.

Let's start by looking at how scope interacts with immediate functions.

4.6.1 Temporary Scope and Private Variables

Using immediate functions, we can start to build up interesting enclosures for our work. Because the function is executed immediately, and all the variables inside of it are confined to its inner scope, we can use it to create a temporary scope within which our state can be maintained,

> **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.

Consider the following snippet:

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

```
})();
```

As the immediate function is executed immediately (hence its name), the click handler is also bound right away. Additionally, note that a closure is created allowing the `numClicks` variable to persist with the handler *but nowhere else*. This is the most common way in which immediate functions are used, just as a simple self-contained wrapper for functionality.

However it's important to remember that since they are functions they can be used in interesting ways, as in:

```
document.addEventListener("click", (function(){
  var numClicks = 0;
  return function(){
    alert( ++numClicks );
  };
})(), false);
```

This is an alternative, and debatably more confusing, version of our previous snippet.

In this case we're again creating an immediate function, but this time we return a value from it: a function to serve as the event handler. Because this is just like any other expression, the returned value is passed to the `addEventListener` method. However, the inner function that we created stil 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 scoped based upon the closure that they're in.

Moreover, using 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 4.16 has a quick example from the Prototype JavaScript library:

Listing 4.16: Using an immediate function 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, Prorotype is extending an object with a number of new properties and methods. In this code, a temporary variable could have been created for

`Element._attributeTranslations.read.values`, but instead, it was chosen 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.

Which we'll be doing without further delay.

4.6.2 Loops

Another very useful application of immediate functions is the ability to solve a nasty issue with loops and closures. Listing 4.17 shows a common piece of problematic code:

Listing 4.17: A problematic piece of code in which the iterator is not maintained in the closure

```
<div>DIV 0</div>
<div>DIV 1</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>
```

Our intention is that clicking each `<div>` element will show its ordinal value. But when we load the page and click on "DIV 0", we see the display of figure 4.7.

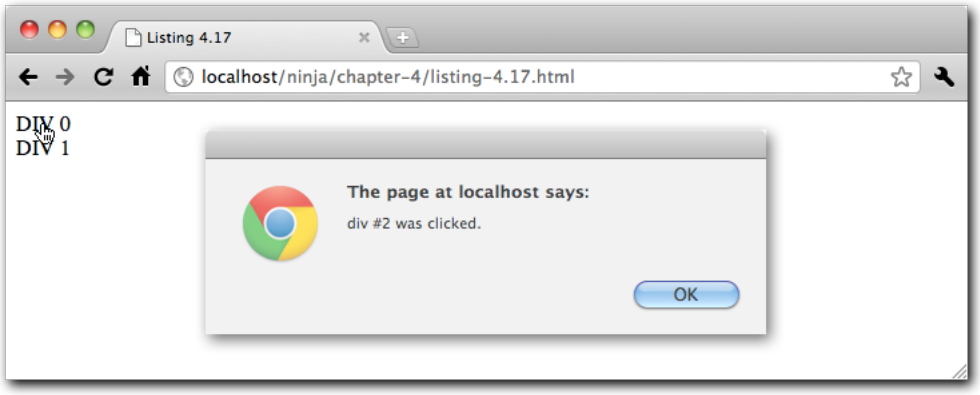

Figure 4.7: Where did we go wrong? Why does DIV 0 think it's 2?

In listing 4.19 we encounter a common issue with closures and looping; namely that the variable that's being closured (`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 are 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 (fighting fire with fire, so to speak), as shown in Listing 4.18 (changes noted in bold).

Listing 4.18: Using an anonymous function wrapper to persist the iterator properly.

```
<div>DIV 0</div>
<div>DIV 1</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>
```

By using an immediate function as the body of the `for` loop (replacing the previous block) we enforce the correct value for the handlers by passing that value into the immediate function (and hence, the closure of the inner function). This means that within the scope of each step of the `for` loop, the `i` variable is defined anew, giving the click handler closure the value that we expects.

Running the updated page shows the expected display of figure 4.8.

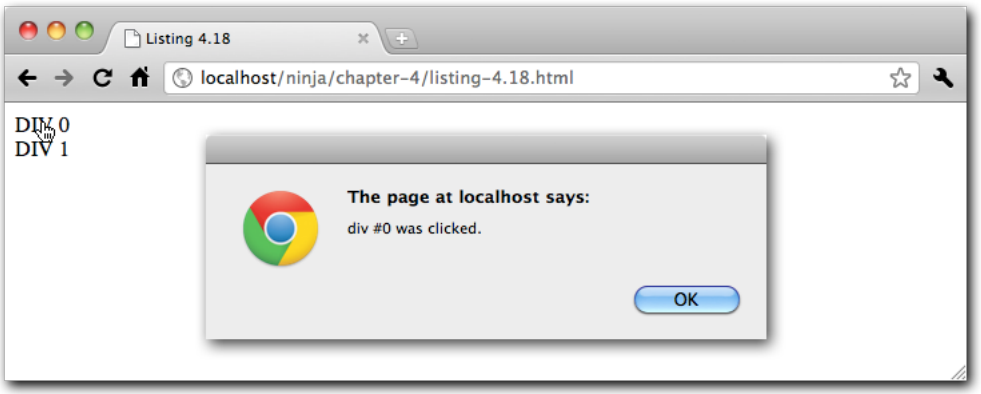

Figure 4.8 That's more like it

This example clearly points out how we can control the scope of variables and values using immediate functions and closures. Let's see how that can help us be good on-page citizens.

4.6.3 Library Wrapping

Another important concept that closures and immediate functions enable is an important one to JavaScript library development. When developing a library it's incredibly important that we don't pollute the global namespace with unnecessary variables, especially ones that are only temporarily used.

To this end, the concept of closures and immediate functions is especially useful: helping us to keep as much of the library as private as possible, and to only selectively introduce variables into the global namespace. The jQuery library takes great care to heed this principle, completely enclosing all of its functionality and only introducing the variables it needs, like `jQuery`, as shown in Listing 4.19.

Listing 4.19: Placing a variable outside of a function wrapper

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

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

Note that there is a double assignment performed, completely intentionally. First, the jQuery constructor (as an anonymout function) is assigned to `window.jQuery` which introduces it as a global variable. However, that does not guarantee that that variable will be the only one 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. We won't care; within our own world that we created via the outer immediate function, the name `jQuery` means only what we want it to mean.

As 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 implemented; another is shown in Listing 4.20.

Listing 4.20: An alternative means of putting a variable in the outer scope

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

  // ...
```

```
  return jQuery;
})();
```

The code in Listing 4.20 has the same effect as that shown in the Listing 4.19, but structured in a different manner. Here we define a `jQuery` function within our anonymous scope, use it freely within that scope, then return it such that it is assigned to a global variable, also named `jQuery`. Oftentimes this particular technique is preferred if you're only exporting a single variable, as the intention of the assignment is somewhat clearer.

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 structure any particular application to your preferences.

4.7 Summary

In this chapter we dove in to how closures, a key concept of functional programming, work in JavaScript.

We started with the basics, looking at how closures are 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 immediate functions which, as we learned, has the power to redefine 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 inevitably encounter.

In this chapter's example code, we lightly introduced the concept of prototypes. Now it's time to dig into prototypes in earnest.

5

Object-orientation with prototypes

In this chapter:

- Using functions as constructors
- Exploring prototypes
- Extending objects with prototypes
- Avoiding common gotchas
- Building classes with inheritance

Now that we've learned how functions are first-class objects in JavaScript, and how closures make them incredibly versatile and useful, we're ready to tackle another important aspect of functions: function prototypes.

Function prototypes are a convenient way to easily define properties to be attached to instances of objects. They can be used for multiple purposes, the primary of which is enabling Object-Oriented programming in JavaScript.

Prototypes are used throughout JavaScript as a convenient means of defining functionality, and automatically applying it to instances of objects. Once defined, the prototype's property automatically become properties of instantiated objects, serving as a blueprint of sorts for the creating of complex objects, and a manner similar to that of classes in classical object-oriented languages.

Indeed, the predominant use of prototypes in JavaScript is in producing a classical-style form of object-oriented code and inheritance.

5.1 Instantiation and prototypes

All functions have a `prototype` property, which by default references an empty object. This property doesn't serve much purpose until the function is used as a ***constructor***. In order to

understand what that means, and what that does, it's important to remember a simple truth about JavaScript: functions serve a dual purpose. They can behave both as regular functions, and as a constructor for a "class", which can be instantiated.

5.1.1 Instantiation

The simplest way to create a new object is with a statement such as:

```
var o = {};
```

This creates a new empty object, which we can then populate with properties via assignment statements.

But those coming from an object-oriented background might miss the convenience and encapsulation that arises from concept of a class constructor: a function that serves to initialize the object to a known initial state.

JavaScript provides such a mechanism, though in a very different form than most other languages. Like object-oriented languages such as Java and C++, JavaScript employs the `new` operator to instantiate new objects, but there is no class definition in JavaScript. Rather, the new operator is applied to a function, and that function serves as the constructor for the newly allocated object.

Let's examine a simple case of using function, both with and without the `new` operator, and see how the prototype property adds functionality to the new instance. Consider the code of listing 5.1.

Listing 5.1: Creating an new instance with prototyped method

```
<script type="text/javascript">

  function Ninja(){}                                    #1

  Ninja.prototype.swingSword = function(){              #2
    return true;
  };

  var ninja1 = Ninja();                                 #3
  assert(ninja1 === undefined,
         "No instance of Ninja created.");

  var ninja2 = new Ninja();                             #4
  assert(ninja2 && ninja2.swingSword(),
         "Instance exists and method is callable." );

</script>
```

#1 Defines dual-purpose function
#2 Adds prototypes method
#3 Calls as a function
#4 Calls as a constructor

In this code we define a function named `Ninja()` that we'll use in tow ways: as a "normal" function, and as a constructor (#1). After the function exists, we add a method, `swingSword()`, to its prototype. Then we put the function through its paces.

First, we call the function normally (#3) and store its result in variable `ninja1`. Looking at the function body (#1) we see that it returns no value, so we'd expect `ninja1` to test as undefined, which we assert to be true.

Then we call the function via the `new` operator, and something completely different happens. The function is once again called, but this time a newly allocate object has been created and set as the context of the function. The result returned from the `new` operator is a reference to this new object. We test for two things: that ninja2 has a reference to an object, and that that object has a method `swingSword()` that we can call.

This shows that the function's prototype serves as a sort of blueprint for the new object when the function is used as a constructor.

When the function is called as a constructor via the `new` operator its ocntext is defined as the new object instance. This means that in addition to adding properties via the prototype, we can initialize values within the constructor function. Let's examine this a little bit more in Listing 5.2.

Listing 5.2: Observing the precedenc of initialization activities

```
<script type="text/javascript">

  function Ninja(){
    this.swung = false;                                    #1
    this.swingSword = function(){                          #2
      return !this.swung;
    };
  }

  Ninja.prototype.swingSword = function(){                 #3
    return this.swung;
  };

  var ninja = new Ninja();                                 #4
  assert(ninja.swingSword(),
         "Called the instance method, not the prototype method.");

</script>
```

#1 Creates an instance variable
#2 Creates an instance method
#3 Defines a prototypes method
#4 Constructs a Ninja

Listing 5.2 is very similar to the previous example: we defined a method by adding it to the prototype property (#3). However, we also add an identically named method within the constructor function itself (#2). The two methods are defined to return opposing results so we can tell which is called. The precedence of the initialization operations is important and goes 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.

When we test (#4) we discover that `swingSword()` method returns true as the binding operations within the constructor always take precedence. Because the `this` context within the constructor refers to the instance itself, we can perform any initialization actions in the constructor to our heart's content.

An important thing to realize about a function's prototype is that, unlike instance properties that are static, any updates to it are reflected in all objects created from that prototype. For example, let's rearrange the code a bit and see what happens, as shown in Listing 5.3.

Listing 5.3: Observing the behavior of changes to the prototype

```
<script type="text/javascript">

  function Ninja(){                                          #1
    this.swung = true;
  }

  var ninja = new Ninja();                                   #2

  Ninja.prototype.swingSword = function(){                   #3
    return this.swung;
  };

  assert(ninja.swingSword(),                                 #4
         "Method exists, even out of order.");

</script>
```

#1 Defines constructor
#2 Instantiates object
#3 Adds a prototyped method
#4 Tests if method was applied

In this example, we define a constructor (#1) and proceed to use it to create an object instance (#2). *After* the instance has been created, we add a method to the prototype (#3). If what we said above is true (that changes to prototypes are reflected in all its instantiated objects) then we'd expect the method to exist on the object even though the method was added to the prototype after the instantiate.

Our test (#4) succeeds, showing that the assertion is true as shown in figure 5.1.

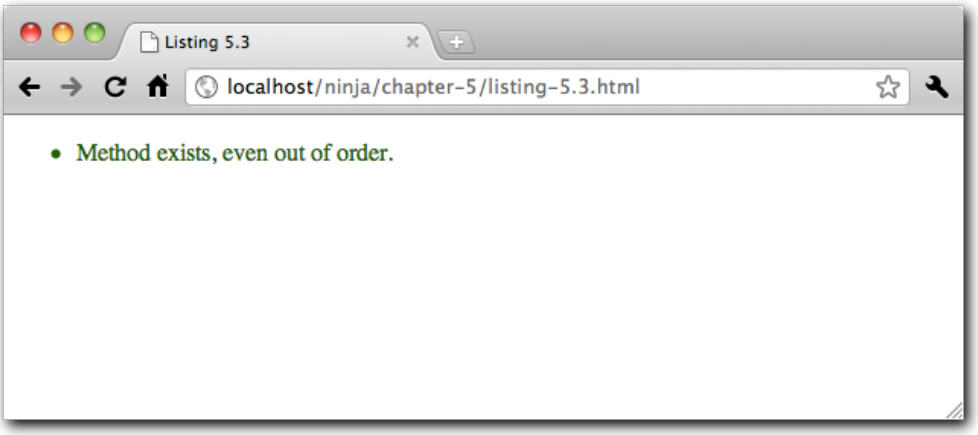

Figure 5.1: Our test proves that prototype changes are applied live!

These seamless live updates give 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 makes it possible for us to create a functional framework which users can extend with further functionality - even well after any objects have been instantiated.

Before we move on, let's try one more variation on this theme. Observe to code of listing 5.4.

Listing 5.4: Fruther observing the behavior of changes to the prototype

```
<script type="text/javascript">

  function Ninja(){
    this.swung = true;
    this.swingSword = function(){                          #1
      return !this.swung;
    };
  }

  var ninja = new Ninja();

  Ninja.prototype.swingSword = function(){                 #2
    return this.swung;
  };

  assert(ninja.swingSword(),                               #3
         "Called the instance method, not the prototype method.");

</script>
```

#1 Define instance method
#2 Defines prototyped method
#3 Tests precedence

In this example we re-introduce an instance method (#1) with the same name as the prototyped method (#2) as we did back in listing 5.2. In that example, the instance method took precedence over the prototyped method. This time, however, the prototyped method is added after the constructor has been executed. Which method will reign supreme in this case?

Our test (#3) shows that, even when the prototyped method is added after the instance method has been added, that the instance method takes precedence.

Now that we know how to instantiate objects via function constructors, let's learn a bit more about the nature of those objects.

5.1.2 Object typing

Once we have new instances of constructed objects, it would be handy to know which function constructed the object instance, so we can refer back to it later, possibly even performing a form of type checking, as shown in Listing 5.5.

Listing 5.5: Examining the type of an instance and its constructor

```
<script type="text/javascript">

  function Ninja(){}

  var ninja = new Ninja();

  assert(typeof ninja == "object",                                  #1
        "The type of the instance is object.");
  assert(ninja instanceof Ninja,                                    #2
         "instance of identifies the constructor." );
  assert(ninja.constructor == Ninja,                                #3
         "The ninja object was created by the Ninja function.");

</script>
```

#1 Tests the type
#2 Tests the instance
#3 Tests the constructor

In Listing 5.5 we define a constructor and create an object instance using it. Then we examining the type of the instance using the `typeof` operator. (#1) This isn't very revealing, as all instances will be objects, thus always returning `"object"` as the result. Much more interesting, however, is the `instanceof` operator (#2), which is really helpful in that it gives us a clear way to determine if an instance was created by a particular function constructor.

On top of this we can also make use of an object property named `constructor` that is added to all instances, and 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 can with the `instanceof` operator).

Additionally, since this is just a reference back to the original function, we can instantiate a new Ninja object using it, as shown in Listing 5.6.

Listing 5.6: Instantiating a new object using a reference to a constructor

```
<script type="text/javascript">

  function Ninja(){}
  var ninja = new Ninja();
  var ninja2 = new ninja.constructor();                    #1

  assert(ninja2 instanceof Ninja, "It's a Ninja!");        #2

</script>
```

#1 Construts a 2nd Ninja
#2 Proves its Ninja-ness

We define a constructor and create an instance using that constructor. Then, we use the constructor property of the created instance to construct a second instance (#1). Testing (#2) shows that a second Ninja has been constructed.

What's especially interesting about this is that we can do this without even having access to the original function; we can use the reference completely behind the scenes.

NOTE

While the `.constructor` property of an object can be changed, it doesn't have any immediate or obvious, purpose as its reason for being is to inform as to where the object was constructed from. The original value will simply be overwritten and lost.

That's all very interesting, but we've just scratched the surface of the super-powers that prototypes confer to us. Now things get really interesting.

5.1.3 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. But in order to make use of it, we need to understand how inheritance works in JavaScript and what role the ***prototype chain*** plays. Let's consider the example of listing 5.7, in which we will attempt to add inheritance to an instance.

Listing 5.7: Trying to achieve inheritance with prototypes

```
<script type="text/javascript">

  function Person(){}                                      #1
  Person.prototype.dance = function(){};                   #1

  function Ninja(){}                                       #2

  Ninja.prototype = Person.prototype;                      #3
  Ninja.prototype = { dance: Person.prototype.dance };     #3

  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" );
</script>
```

#1 Defines a dancing Person
#2 Defines a Ninja
#3 Attempt to make a Ninja a Person

As the prototype of a function is just an object, there are multiple ways of copying functionality (such as properties or methods). So we can achieve inheritance by *chaining* the prototype of objects. In this code, we define a Person (#1), and then a Ninja (#2). And because a Ninja is clearly a person, we want Ninja to inherit the attributes of Person. We attempt to do so in this code by copying (#3) the Person prototype and its method to the Ninja prototype.

Running our test reveals that we failed, as shown in figure 5.2.

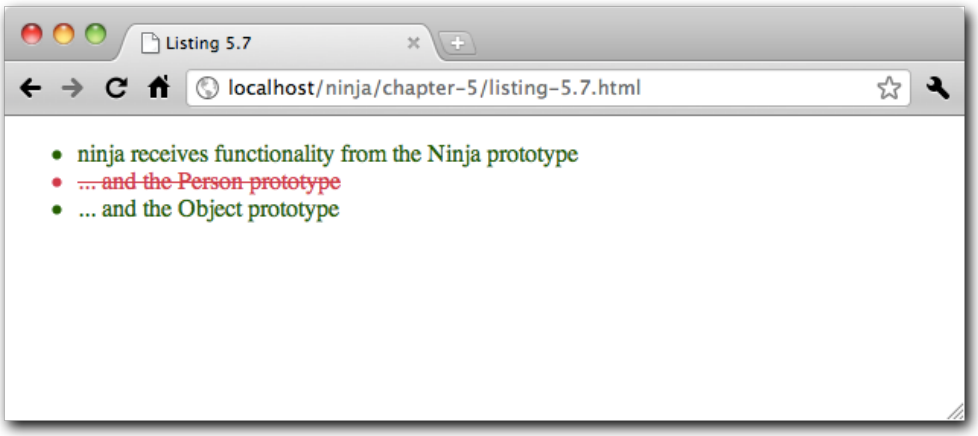

Figure 5.2: Our ninja isn't a Person. No dancing!

NOTE

It's interesting to note that even without doing anything overt, all objects are instances of Object. Perform the statement `console.log({}.constructor)` in a browser, and see what you get.

The only technique capable of creating a prototype chain is by using an instance of an object as the other object's prototype as in:

```
SubClass.prototype = new SuperClass();
```

This will preserve the prototype chain as the prototype of the SubClass instance will be an instance of the SuperClass, which has a prototype with all the properties of SuperClass, and which will in turn have a prototype pointing to an instance of its superclass, and on and on.

Let's change the code of listing 5.7 to use this technique as shown in listing 5.8.

Listing 5.8 Achieving inheritance with prototypes

```
<script type="text/javascript">

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

  function Ninja(){}

  Ninja.prototype = new Person();                          #1

  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" );

</script>
```

#1 Makes a Ninja a Person

The only change we made to the code was to use an instance of Person as the prototype for Ninja (#1). Running the tests shows that we've succeeded, as shown in figure 5.3.

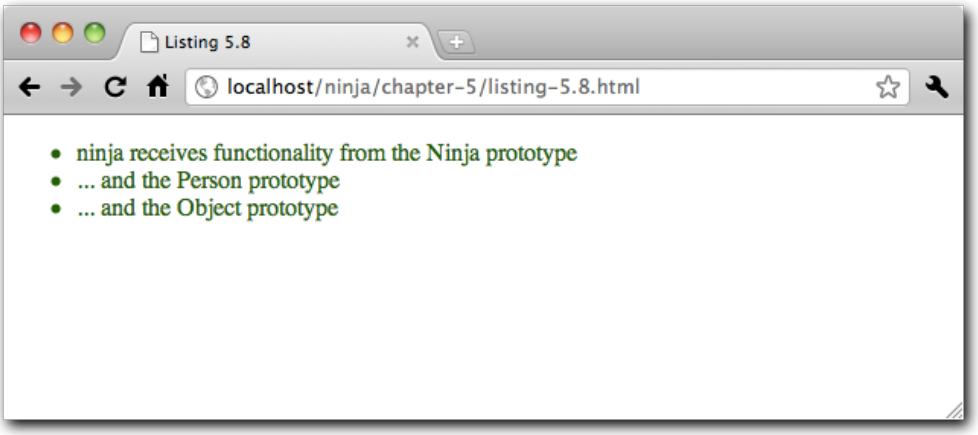

Figure 5.3: Our ninja is a Person! Let the dancing begin.

The implications are that when we perform an `instanceof` operation we can determine if the function inherits the functionality of any object in its prototype chain.

NOTE

Make sure not to 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 undesirable side effects.

An additional side effect of doing prototype inheritance in this manner is that all inherited function prototypes will continue to live update. The manner in which the prototype chain is applied for our example is something akin to what's shown in Figure 5.4.

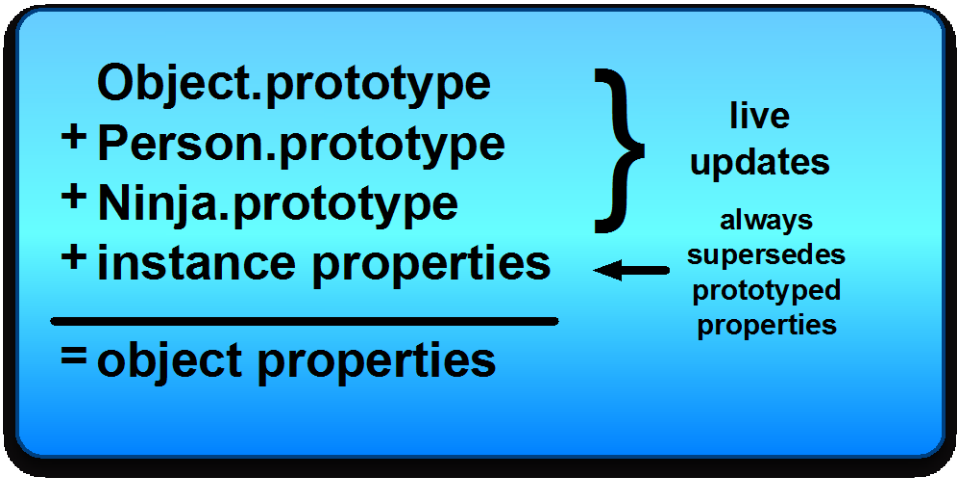

Figure 5.4: The order in which properties are bound to an instantiated object

It's important to note from Figure 5.4 is that our object has properties that are inherited from the Object prototype. 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, as each of those object constructors are functions themselves. This proves to be an incredibly powerful feature of the language. Using it we can extend the functionality of the language *itself*, introducing new or missing pieces of the language.

For example, one such case where this would be quite useful is in anticipating some of the features of future versions of JavaScript. For example, JavaScript 1.6 introduced a couple of useful helper methods, including some for Arrays.

One such method is `forEach()` which allows us to iterate over the entries in an array, calling a function for every entry. 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 implement future functionality, eliminating the need to wait until the next version of the language is ready. Listing 5.9 shows a possible future-compatible implementation of `forEach()` that we could have defined prior to JavaScript 1.6, or for use in older browsers.

Listing 5.9: Implementing the JavaScript 1.6 array forEach method in a future-compatible manner

```
<script type="text/javascript">

  if (!Array.prototype.forEach) {                              #1
    Array.prototype.forEach = function(fn, callback) {         #2
      for (var i = 0; i < this.length; i++) {
        fn.call(callback || null, this[i], i, this);           #3
      }
    };
  }

  ["a", "b", "c"].forEach(function(value, index, array) {      #4
    assert(value,
           "Is in position " + index + " out of " +
           (array.length - 1));
  });

</script>
```

#1 Tests for pre-existence
#2 Adds method
#3 Calls callback for each entry
#4 Puts it through its paces

Before we stomp on an implementation that might already be there, we check to make sure that Array doesn't already have a `forEach()` method defined (#1), and skip the whole thing if so. This makes the code future-compatible as when it executes in an environment where the new method is defined, it will defer to the native method.

If we determine that the method does not exist, we go ahead an add it ti the Array prototype (#2), simply looping through the array using a traditional for-loop, and calling the callback method for each entry (#3). The values passed to the callback are: the entry, the index, and the original array.

Note that the expression `callback || null` prevents us from passing a possible `undefined` value to `call()`.

Because all the built-in objects include prototypes, we have all the power necessary to extend the language to our desires.

But 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 is significant possibility for naming collisions to occur.

Also, when implementing features on native prototypes that are forward-looking (such as our `forEach()` implementation) there's a danger that our anticipated implementation may not exactly match the final implementation, causing issues to occur when a browser finally does implement the method. We should always take great care when treading in these waters.

We've seen that we can use prototypes to augment the native JavaScript objects; now let's turn our attention to the DOM.

5.1.4 HTML DOM prototypes

A fun feature in modern browsers, including Internet Explorer 8+, Firefox, Safari, and Opera, is that all DOM elements inherit from an HTMLElement constructor. By making the HTMLElement prototype accessible, the browser is providing us with the ability to extend any HTML node of our choosing,. Let's explore that in listing 5.10.

Listing 5.10: 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() {     #1
    if (this.parentNode)
      this.parentNode.removeChild(this);
  };

  var a = document.getElementById("a");            #2
  a.parentNode.removeChild(a);                     #2

  document.getElementById("b").remove();           #3

  assert(!document.getElementById("a"),"a is gone.");
  assert(!document.getElementById("b"),"b is gone too.");
</script>
```

#1 Adds a new method to all elements
#2 Does it the old-fashioned way
#3 Uses the new method

In this code, we add a new remove() method to all DOM elements by augmenting the prototype of the base HTMLElement constructor (#1).

Then we remove element `a` using the native means (#2), and then we remove `b` using our new method. In both cases, we assert that the elements are removed from the DOM.

More information about this particular feature can be found in the HTML 5 specification at http://www.whatwg.org/specs/web-apps/current-work/multipage/section-elements.html.

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

The most important thing to realize, when working with these prototypes, is that they don't exist in versions of Internet Explorer prior to IE8. If older version of IE aren't a target platform for you, then these features should serve you well.

Another point that we need to be aware of is whether HTML elements can be instantiated directly from their constructor function. Consider something like this:

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

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

Save for the gotcha that this feature presents with regards to platform compatibility with older browsers, the benefits with respect to clean code can be quite dramatic and should be strongly investigated in applicable situations.

And speaking of gotchas…

5.2 The Gotchas!

As with most things in life, in JavaScript there are a series of gotchas associated with prototypes, instantiation, and inheritance of which we need to be aware. Some of them can be worked around, but a number of them will simply require a dampening of our excitement.

Let's take a look at some of them.

5.2.1 Extending Object

Perhaps the most egregious mistake that we can make with prototypes is to extend the native `Object.prototype`. The difficulty is that when we extend this prototype *all* objects receive those additional properties. This is especially problematic as when we iterate over the properties of the object these new properties appear, causing all sorts of unexpected behavior. Let's illustrate that with an example.

Let's say that we wanted to do something seemingly innocuous such as adding a `keys()` method to Object that would return an array of all the names (keys) of the properties in the object, as shown in Listing 5.11.

Listing 5.11: Unexpected behavior of adding extra properties to the Object prototype

```
<script type="text/javascript">

  Object.prototype.keys = function() {                    #1
    var keys = [];
    for (var p in this)
      keys.push(p);
    return keys;
  };

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

```
  assert(obj.keys().length == 3,                          #3
        "There are three properties in this object.");

</script>
```

#1 Defiens new method
#2 Creates test subject
#3 Tests the new method

First, we define the new method (#1) by simply iterating over the properties and collecting the keys into an array, which we return,

We define a test subject with three properties (#2), and then test that we get a three-elements array as a result (#3).

But the test fails, as shown in figure 5.5.

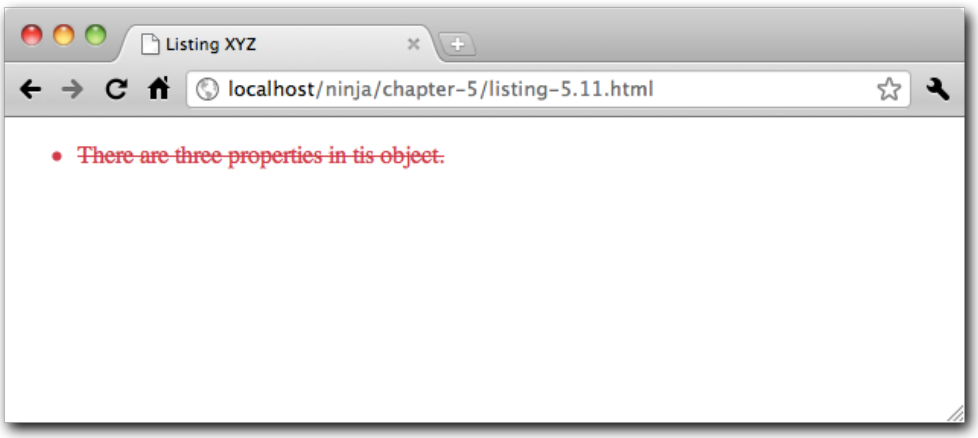

Figure 5.5: Whoa! We screwed up a fundamental assumption of objects

What went wrong, of course, is that in adding the `keys()` method to Object, we introduced another property that will appear on all objects and is included in the count. This affects all objects and would force any code to have to account for the extra property. This could break code based upon reasonable assumptions made by page authors who are using our code. This is obviously unacceptable.

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. Let's observe its use in listing 5.12.

Listing 5.10: Using the hasOwnProperty() method to tame Object prototype extensions

```
<script type="text/javascript">

  Object.prototype.keys = function() {
    var keys = [];
```

```
    for (var i in this)
      if (this.hasOwnProperty(i)) keys.push(i);        #1
    return keys;
  };

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

  assert(obj.keys().length == 3,                        #2
      "There are three properties in this object.");

</script>
```

#1 Ignores prototyped properties
#2 Tests method

Our redefined method ignores non-instance properties (#1) so that this time, the test (#2) succeeds.

But just because it's possible for us to work around this issue doesn't mean that it should be abused and become a burden for the users of our code. Looping over the properties of an object is an incredibly common behavior and it's uncommon for people to use `hasOwnProperty()` within their own code – most page authors probably do not even know of its existence. Generally we should avoid using such workarounds except in the most controlled situations (such as a single developer working on a single site with code that is completely controlled).

Now we'll look at another pitfall that we could trap us.

5.2.2 Extending Number

It's generally safe to extend most native prototypes (save for Object, as previously mentioned), but one other problematic native is Number. Due to how numbers, and properties of numbers, are parsed by the JavaScript engine some results can be rather confusing, as in Listing 5.13.

Listing 5.13: Adding a method to the Number prototype.

```
<script type="text/javascript">

  Number.prototype.add = function(num){                 #1
    return this + num;
  };

  var n = 5;                                            #2
  assert(n.add(3) == 8,
        "It works when the number is in a variable.");

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

  assert(5.add(3) == 8, "What about a simple literal?");  #4

</script>
```

#1 Defined new method
#2 Tests variable format

#3 Tests expression format
#4 tests literal format

Here we define a new `add()` method on Number (#1) that will take the argument, add it to the number's value, and return the result. The we test the new method using a number of formats: with the number in a variable (#2), with a number in an expression (#3) and directly on a literal (#4).

But when we try to load the page into a browser, the page won't even load as shown in figure 5.6.

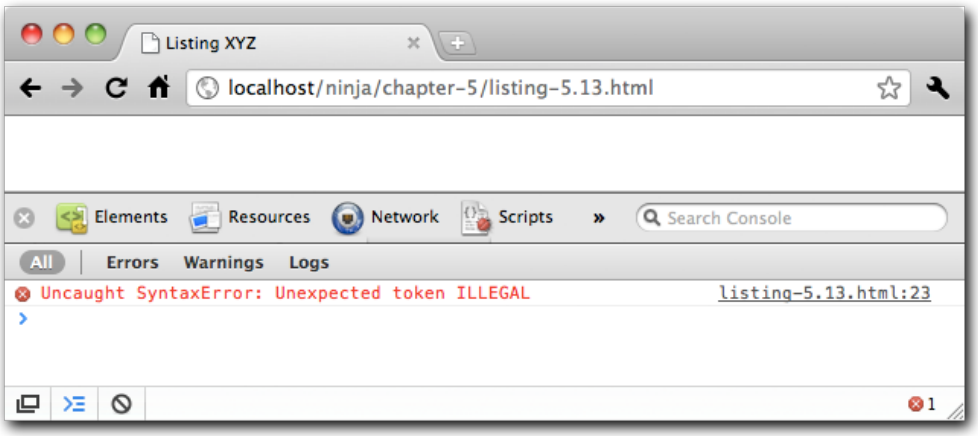

Figure 5.6: When tests won't even load, we know there's a big problem

It turns out that the syntax parser can't handle the literal case.

This can be a frustrating issue to deal with as the logic behind it can be rather obtuse. There have been libraries that have continued to include Number prototype functionality, regardless of these issues, simply stipulating how they should be used (Prototype being one of them). That's certainly an option, albeit one that requires the library to explain the issues with good documentation and clear tutorials.

Now let's look at some issues we can encounter when we subclass, rather than augment, native objects.

5.2.3 Subclassing native objects

Another tricky point that we might stumble across regards the subclassing of native objects. The one object that's quite simple to subclass is Object (as it's the root of all prototype chains to begin with). However, once we start wanting to subclass other native objects, the situation becomes less clear-cut. For example, with Array, everything might seem to work as we might expect it to. Let's take the code of listing 5.14.

Listing 5.14: Subclassing the Array object

```
<script type="text/javascript">

  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.");


</script>
```

We subclass Array with a new constructor of our own, MyArray, and it all works fine and dandy, unless, that is, you tried to load this into Internet Explorer. For whatever reason, the native Array object is not allowed to be subclassed in Internet Explorer (the length property is immutable – causing all other pieces of functionality to become broken).

When faced with such situations, it's a better strategy to implement individual pieces of functionality from native objects, rather than attempt to sub-class them completely. Let's take a look at this approach as outlined in listing 5.15.

Listing 5.13: Simulating Array functionality but without the true sub-classing.

```
<script type="text/javascript">

  function MyArray() {}                                          #1
  MyArray.prototype.length = 0;                                  #1

  (function() {                                                  #2
    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();                                      #3
  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 subclassing Array, though.");

</script>
```

#1 Defines new class

#2 Copies array functionality
#3 Tests the new class

In Listing 5.13 we define a new class, and give it its own `length` property. Them rather than trying to subclass Array, which we've already learned won't work across all browsers, we use an immediate function (#2) to a add selected methods from Array to our class using the `apply()` trick we learned back in chapter 3.

Note the use of the array of method names to keep things tidy and easy to extend.

The only property that we had to implement ourselves is the `length` (because that's the one property that must remain mutable - the feature that Internet Explorer does not provide).

Now let's see what we can do about a common problem people trying to use our code might run into.

5.2.4 Instantiation issues

We've already noted that functions serve a dual purpose: as "normal" functions, and as constructors. Because of this, it may not always be clear to users of the code which is which.

Let's start by looking at a simple case of what happens when someone gets it wrong, as shown in listing 5.16.

Listing 5.14: The result of leaving off the new operator from a function call.

```
<script type="text/javascript">

  function User(first, last){                          #1
    this.name = first + " " + last;
  }

  var user = User("Ichigo", "Kurosaki");              #2

  assert(user, "User instantiated");                  #3
  assert(user.name == "Ichigo Kurosaki",              #4
         "User name correctly assigned");

</script>
```

#1 Defines User class
#2 Create test user
#3 Tests instantiation
#4 Tests for proper construction

In the code, we define a User class (#1) whose constructor accepts a first and last name and concatenates them to form a full name. We then create an instance of the class in the user variable (#2), and test that the object was instantiated (#3) and that the constructor performed correctly (#4).

But things go awry when try it out, as shown in figure 5.7.

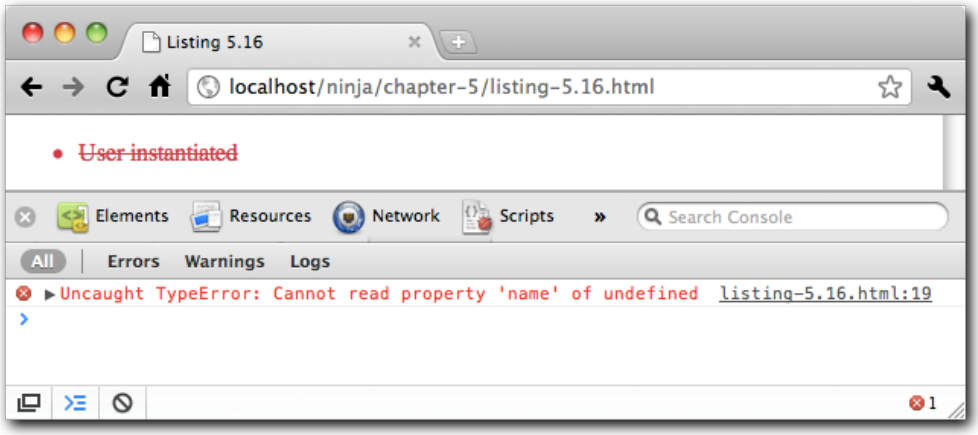

Figure 5.7: Our object didn't even get instantiated!

The test reveals that the first test fails, indicating that object wasn't even instantiated, which causes the second test to throw an error.

On a quick inspection of the code it may not have been immediately obvious that the `User()` function is actually something that is meant to be instantiated with the `new` operator, or maybe we just slipped up and forgot. In either case, the absence of the `new` operator caused the function to called in a normal fashion, without instantiation. A new user might easily fall into this trap, trying to call the function without the operator, causing severely unexpected results (e.g. `user` would be undefined).

NOTE

You may have noticed that since the beginning of this book, and notably in the this and the previous chapter, we have user a naming convention in which some functions start with a lowercase letter and others start with an uppercase character. This is a common convention in which functions serving as constructors use an uppercase opening character, and normal functions do not.

Moreover, constructors tend to be nouns that identify the "class" that they are constructing: Ninja, User, Samurai, and so on. Whereas normal functions are named as verbs, or verb/object pairs, that describe what they do: throwShuriken, swingSword, hideBehindAPlant.

More than merely causing unexpected errors, when a function meant to be instantiated isn't, it can have subtle side-effects such as polluting the current scope (frequently the global

namespace), causing even more unexpected results. For example, inspect the code of Listing 5.17.

Listing 5.17: An example of accidentally introducing a variable into the global namespace

```
<script type="text/javascript">

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

  var name = "Rukia";                                #1

  var user = User("Ichigo", "Kurosaki");             #2

  assert(name == "Rukia",                            #3
         "Name was set to Rukia.");

</script>
```

#1 Creates global variable
#2 Calls constructor incorrectly again
#3 Tests the global variable

This code is similar to that of the previous example, except that this time there just happens to be a global variable named `name` in the global namespace (#1), and makes the same mistake (#2) as the previous example. But this time we don't have a test that catches that mistake. Rather, the test we have shows that the value of the global name variable has been overwritten (#3) as it fails when executed.

To find out why, look at the code of the constructor. When called as a constructor, the context of the function invocation is the newly allocated object. But what is it when called as a normal function? Recall from chapter 3 that it's the global scope. Which means that the reference `this.name` refers not to the `name` instance property of an allocated object, but the `name` variable of the global scope!

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 misusing 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 begin pulled out from underneath their feet?).

As JavaScript ninjas, we want to be sensitive to the needs of our user base, so let's ponder on what we can do about the situation. First, in order to do anything about it, we need a way to determine when the situation comes up. Is there a way that we can determine whether a function that we intend to be used as a constructer is being incorrectly called?

Consider the code of listing 5.18

Listing 5.18: Determining if we're called as a constructor

```
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." );
```

Recall a few important concepts:

- We can get a reference to the currently executing function via `arguments.callee` (we learned this in chapter 3)
- The context of a regular function is the global scope (unless someone tried hard not to make it so)
- The `instanceof` operator for a constructed object test for its constructor

Using these facts we can see that the expression:

```
this instanceof arguments.callee
```

will evaluate to `true` when executed within a constructor, but `false` when executed within a regular function.

This means that, within a function that we intend to be called as a constructor, that we can test to see if someone called us without the new operator! Neat! But what do we do about it?

If we weren'r ninjas, we ight just throw an Error telling the user to do it right next time. But we're better than that. Let's see if we can just fix the problem for them.

Consider the changes to the User constructor shown in listing 5.19.

Listing 5.19: Fixing things on the caller's behalf

```
<script type="text/javascript">

  function User(first, last) {
    if (!(this instanceof arguments.callee)) {          #1
      return new User(first,last);                      #1
    }                                                   #1
    this.name = first + " " + last;
  }

  var name = "Rukia";

  var user = User("Ichigo", "Kurosaki");                 #2

  assert(name == "Rukia","Name was set to Rukia.");      #3
  assert(user instanceof User, "User instantiated");     #3
  assert(user.name == "Ichigo Kurosaki",                 #3
         "User name correctly assigned");                #3

</script>
```

#1 Fixes things up
#2 Calls constructor incorrectly
#3 tests everything

By using the expression we developed in listing 5.18 to determine if the user has called us incorrectly, we instantiate a User ourselves (#1) and return it as the result of the function. The outcome is that regardless of whether the caller invokes us as a normal function (#2) or not, they end up with a User instance, which our tests (#3) verify.

Now *that's* user friendly! Who says ninjas are mean?

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.

Enough of problems. Let's take a look at how we use these new-found powers to write more class-like code.

5.3 Enabling class-like code

While it's great that JavaScript lets us use a form of inheritance via prototypes, a common desire for many developers, especially those from a classical object-oriented background, is a simplification or abstraction of JavaScript's inheritance system into one that they are more familiar with.

This inevitably leads us towards the realm of classes; what a typical object-oriented developer would consider to be expected, even though JavaScript doesn't support classical inheritance natively.

Generally there are a handful of features that such developers crave:

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

There is a number of existing JavaScript libraries that simulate classical inheritance. Out of all them there are two that stand up above the others: the implementations within base2 and Prototype. While they each contain a number of advanced features, their object-oriented core is an important part of these libraries, and we'll distill what they offer to come up with a proposed syntax that would make things a tad more natural for classically trained object-oriented developers.

Listing 5.20 shows an example of a syntax that could achieve the above listed goals.

Listing 5.20: An example of somewhat classical-style inheritance syntax

```
<script type="text/javascript">
  var Person = Object.subClass({                          #1
    init: function(isDancing) {
      this.dancing = isDancing;
    },
    dance: function() {
      return this.dancing;
    }
  });
```

```
  var Ninja = Person.subClass({                                    #2
    init: function() {
      this._super(false);                                          #3
    },
    dance: function() {
      // Ninja-specific stuff here
      return this._super();
    },
    swingSword: function() {
      return true;
    }
  });

  var person = new Person(true);                                    #4
  assert(person.dance(),                                            #4
         "The person is dancing.");                                 #4

  var ninja = new Ninja();                                          #5
  assert(ninja.swingSword(),                                        #5
         "The sword is swinging.");                                 #5
  assert(!ninja.dance(),                                            #5
         "The ninja is not dancing.");                              #5

  assert(person instanceof Person,                                  #6
         "Person is a Person.");                                    #6
  assert(ninja instanceof Ninja &&                                  #6
         ninja instanceof Person,                                   #6
         "Ninja is a Ninja and a Person.");                         #6

</script>
```

#1 Subclasees Object
#2 Subclasses Person
#3 Calls superclass constructor
#4 Tests Person class
#5 Tests Ninja class
#6 Tests class hierarchy

There a number of important things to note about this example:

- Creating a new “class” is accomplished by calling a `subclass()` method of the existing constructor function for the superclass, as we did here by creating a Person class from Object (#1), and creating a Ninja class from Person (#2).
- We wanted the creation of a constructor to be simple. In our proposed syntax, we simply provide an `init()` method for each class, as we did for Person and for Ninja.
- All our “classes” eventually inherit from a single ancestor: Object. Therefore if we want to create a brand new class it must be a subclass of Object or a class that inherets from Object in its class hierarchy (completely mimicking the current prototype system).
- The most challenging aspect of this syntax is enabling access to overridden methods with their context properly set. We can see this with the use of `this._super()`,

calling the original `init()`(#3) and `dance()` methods of the Person superclass.

Proposing a syntax that we'd like to use to accomplish an inheritance scheme was the easy part. Now we need to implement it!

The code in Listing 5.21 enables the notion of 'classes' as a structure, maintains simple inheritance, and allows for the super-method calling. Be warned that this is pretty involved code – but we're all here to become ninjas, and this is Master Ninja Territory.

In fact, we're going to present the code in complete form in listing 5.21, so that we can see how all the parts fit together, but then we'll dissect it piece by piece in the subsections that follow.

Listing 5.21: A sub-classing method

```
(function() {
 var initializing = false,
     fnTest = /xyz/.test(function() { xyz; }) ? /\b_super\b/ : /.*/; #1

 Object.subClass = function(prop) {                                  #2
   var _super = this.prototype;

   initializing = true;                                              #3
   var proto = new this();                                           #3
   initializing = false;                                             #3

   for (var name in prop) {                                          #4
     proto[name] = typeof prop[name] == "function" &&
         typeof _super[name] == "function" && fnTest.test(prop[name]) ?
         (function(name, fn) {                                        #5
           return function() {
             var tmp = this._super;

             this._super = _super[name];

             var ret = fn.apply(this, arguments);
             this._super = tmp;

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

   function Class() {                                                #6
     if (!initializing && this.init)
       this.init.apply(this, arguments);
   }

   Class.prototype = proto;                                          #7

   Class.constructor = Class;                                        #8

   Class.subClass = arguments.callee;                                #9

   return Class;
```

```
  };
})();
```

#1 Determines if functions can be serialized
#2 Creates a subclass
#3 Instantiates the superclass
#4 Copies properties into prototype
#5 Defines overriding function
#6 Creates a dummy class constructor
#7 Populates class prototype
#8 Overrides constructor reference
#9 Makes class extendable

The two most important parts of this implementation are the initialization and super-method portions. Having a good understanding of what's being achieved in these areas will help with understanding of the full implementation. But as it'd be confusing to jump right into the middle of this rather complex code, we'll start at the top and work or way through the code from top to bottom.

Let's start with something you might not ever have seen before.

5.3.1 Checking for function serializability

Unfortunately, the code that starts out our implementation is something that's rather esoteric, and could be confusing to most.

Later on in the code, we're going to need to know if the browser supports function serialization. But as the test for that is one with rather complex syntax, we're going to get it out of the way now, and store the result so that we don't have to complicate the later code that will already be complicated enough in its own right.

Function serialization is simply the act of taking a function, and getting its text source back. We'll need it later to check if a function has a specific reference within it that we'll be interested in. In most modern browsers, the function's `toString()` method will do the trick. So, generally, a function is serialized simply by using it in a context that expects a string, causing it's `toString()` method to be invoked. And such it is with our code to test if it works.

After we set a variable named `initializing` to false (we'll see why in just a bit), we test if function serialization works with the expression:

```
/xyz/.test(function() { xyz; })
```

This expression creates a function that contains the text "xyz", and passes it to the test() method of a regular expression that tests for the string "xyz". If the function is correctly serialized (the `test()` method expects a string triggering the function's `toString()` method), the result will be `true`.

Using this text expression we set up a regular expression to be used later in the code with:

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

This establishes a variable named `superPattern` that we'll use later to check if a function contains the string "_super". We can only do that if function serialization is

supported, so we substitute a pattern that matched anything in browsers that don't allow us to serialize functions.

We'll be using this result later on, but by doing the check now, we don't have to embed this expression, with its rather complicated syntax, in the later code.

Now let's move on to the actual implementation of the sub-classing method.

5.3.2 Initialization of subclasses

At this point, we're ready to declare the method that will subclass a superclass (#2), which we accomplish with:

```
Object.subClass = function(properties) {
  var _super = this.prototype;
```

This adds a `subclass()` method to Object that accepts a single parameter that we'll expect to be hash of the properties to be added to the subclass.

In order to simulate inheritance with a function prototype, we use the previously discussed technique of creating an instance of the superclass and assigning it to the prototype. *Without* using our above implementation it could look something like this code :

```
function Person(){}
function Ninja(){}
Ninja.prototype = new Person();
assert((new Ninja()) instanceof Person,
       "Ninjas are people too!");
```

What's challenging about this snippet, is that all we *really* want is the benefits of `instanceof`, but 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 an instance, we can 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 circumvent any unnecessary and expensive startup code, when we're simply creating an instance to serve as a prototype.

What we need to do next is to copy any subclass-specific properties that were passed to the method to the prototype instance. But that's not quite so easy as it sounds.

5.3.3 Preserving super methods

In most languages supporting inheritance, when a method is overridden we retain the ability to access the overridden method. This is useful because sometimes we want to completely replace a method's functionality, but sometimes we just want to augment it. In our particular implementation, we create new temporary method named `.__super`, which is only

accessible from within a subclassed method, that references the original method in the superclass.

For example, recall from listing 5.20, when we wanted to call a superclasses' constructor, we did that with the following code (parts omitted for brevity):

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

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

Within the constructor for Ninja, we call the constructor for Person, passing an appropriate value. This prevents us from having to copy code – we can leverage the code within the superclass that already does what we need it to do.

Implementing this functionality (in the code of listing 5.21) is a multi-step process. In order to augment our subclass with the object hash that is passed into the `subclass()` method, we simply need to merge the superclass properties and the passed properties. To start, we create an instance of the superclass to use as a prototype (#3) with the following code:

```
initializing = true;
var proto = new this();
initializing = false;
```

Note how we "protect" the initialization code as we discussed in the previous section, with the value of the `initializing` variable.

Now we are ready to merge the passed properties into this `proto` object (a prototype prototype, if you will). If we were unconcerned with superclass functions, that would be an almost trivial task:

```
for (var name in properties) proto[name] = properties[name];
```

But we *are* concerned with superclass functions, so the above code will work for all properties except functions that want to call their superclass equivalent. So when we're overriding a function with one that will be calling it via `_super`, we'll need to wrap the subclass function with one that defines a reference to the superclass function via a property named `_super`.

But before we can do that, we need to detect the condition under which we need to wrap the subclass function. We can do that with the following conditional expression:

```
typeof properties[name] == "function" &&
typeof _super[name] == "function" &&
superPattern.test(properties[name])
```

This expression contains three clauses that check:

1. Is the subclass property a function?

2. Is the superclass property a function?

3. Does the subclass function contain a reference to `_super()`?

Only if all three clauses are `true` do we need to do anything other than copy the property value. Note that we use the regular expression pattern that we set up in section 1.3.1, along with function serialization, to test if the function calls its superclass equivalent.

If the conditional expression indicates that we must wrap the function, we do so by assigning the result of the following immediate function(#5) to the subclass property:

```
(function(name, fn) {
  return function() {
    var tmp = this._super;

    this._super = _super[name];

    var ret = fn.apply(this, arguments);
    this._super = tmp;

    return ret;
  };
})(name, properties[name])
```

This immediate function creates and returns a new function that wraps and executes the subclass function, while making the superclass function available via the `_super` property. 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 (we 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 in the superclass 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 that of the superclass).

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

There are any 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 a good mix of usability and simplicity.

5.4 Summary

Adding object-orientation to JavaScript via function prototypes and prototypal inheritance ia a features that can provide an incredible amount of wealth to developers who prefer an object-oriented slant to theif code. By allowing for the greater degree of control and structure that object-orientation can bring to the code, JavaScript applications can improve in clarity and quality.

In this chapter, we looked at how using the `prototype` property of functions allows us to bring object-orientation to JavaScript code.

We started by examining exactly what `prototype` is, and what role it plays when a function is paired with the `new` operator to become a constructor. We observed how functions behave when used as constructors and how it differs from direct invocation of the function.

Then, we learned how to determine the type of an object, along with discovering which constructor resulted in its coming into being.

We then dug into the object-oriented concept of inheritance, and learned how to use the prototype chain to effect inheritance in JavaScript code.

In order to avoid common pitfalls, we looked at some common "gotchas" that could trap the unwary, with regards to extending Object and other native objects, as well as how to guard against instantiation issues caused by the improper use of our constructors.

We wrapped up the chapter by proposing a syntax that could be used to enable the subclassing of objects in JavaScript, and then created a method that implements that syntax. (Not for the faint of heart, that example!)

Due to the inherit extensibility that prototypes provide, they afford a versatile platform to build off of for future development.

Another fundamental feature, whose mastery will help promote our code to ninja status, is the subject of our next chapter: timers.

6

Tangling with Timers

In this chapter:

- How timers work
- An examination of timer execution
- Processing large tasks using timers
- Managing animations with timers
- Better testing with timers

Timers are an often misused and poorly understood feature available to us in JavaScript that can provide great benefit to the developer in complex applications; when used properly, that is.

Note that we called timers a feature that is *available* to us in JavaScript, but did not call them a feature of JavaScript itself; as they're not. Rather, timers are provided as part of the objects and methods that the web browser makes available. This means that if we choose to use JavaScript in a non-browser environment it's very likely that timers will not exist, and we'd have to implement our own version of them using implementation-specific features (such as threads in Rhino).

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 a timer will never interrupt another currently running timer.

Let's take a look at how this all works.

6.1 How Timers Work

Due to their sheer usefulness, it's important to understand how timers work at a fundamental level. They may seem to behave unintuitively at times because of the single thread within which they execute; we're most probably used to things like timers working in a multi-threaded environment.

We'll examine the ramifications of JavaScript's single-threaded restrictions in a moment, but let's start by examining the functions by which we can construct and manipulate timers.

6.1.1 Setting and clearing timers

JavaScript provides us with a two methods to create timers, and two to clear (remove) them. All are methods of the `window` (global context) object.

They are described in table 6.1.

Table 6.1: JavaScript's timer manipulation methods

Method	Format	Description
`window.setTimeout`	`id = setTimeout(fn,delay)`	Initiates a timer that will fire exactly once, executing the passed function, after the delay has elapsed. A value that uniquely identifies the timer is returned.
`window.clearTimeout`	`clearTimeout(id)`	Cancels (clears) the timer identified by the passed value if the timer has not yet fired.
`window.setInterval`	`id = setInteval(fn,delay)`	Initiates a timer that will continually fire, executing the passed function, at the specified delay interval until canceled. A value that uniquely identifies the timer is returned.
`window.clearInterval`	`clearInterval(id)`	Cancels (clears) the interval timer identified by the passed value.

These methods allow us to set and clear timers that either fire a single time, or periodically as a set interval. In practice, most browser allow you to use either clear method to cancel a timer (either of `clearTimeout()` or `clearInterval()` can be used to cancel

a timer or an interval timer), but it's recommended that the methods be used in matched pairs if for nothing other than clarity.

There's an important concept that needs to be understood with regard to JavaScript timers: the timer delay is not guaranteed. Let's explore that concept.

6.1.2 Timer execution

As all of the JavaScript code in a browser executes within a single thread, the handlers for asynchronous events, such as mouse clicks and timers, are only executed when there's nothing else already executing.

This is likely best demonstrated with a timing diagram, as shown in figure 6.1.

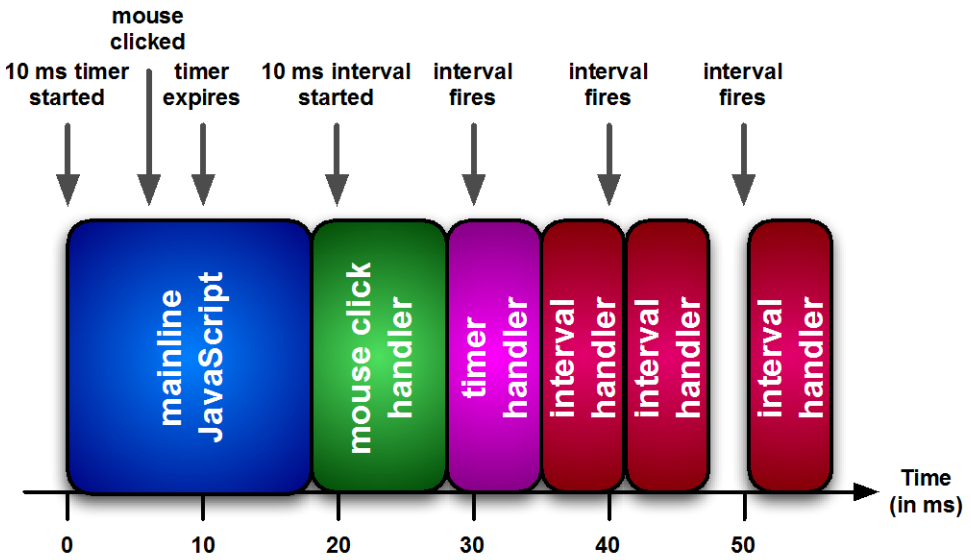

Figure 6.1: A timing diagram that shows how mainline code and handlers execute within a single thread

There's a lot of information to digest from figure 6.1, but understanding it completely gives us a better realization of how asynchronous JavaScript execution works. This diagram is one dimensional, with time (in milliseconds) running from left to right along the x axis. The boxes represent portions of JavaScript code under execution, extending for the amount of time they are running. For example, the first block of mainline JavaScript code executes for approximately 18ms, the mouse click block for approximately 10ms, and so on.

Because JavaScript can only execute one block of code at a time (due to its single-threaded nature), each of these units of execution are blocking the progress of other asynchronous events. This means that when an asynchronous event occurs (like a mouse

click, a timer firing, or even an XMLHttpRequest completing), it gets queued up to be executed when the thread frees up. How this queuing actually occurs varies from browser-to-browser, so consider this to be a simplification, but one that's close enough to understand the concepts.

Starting out, within the execution of the first block of JavaScript, a number of important events occur:

- At 0 milliseconds, a timeout timer is initiated with a 10 ms delay, and an interval timer is initiated with a 10 ms delay
- At 6 milliseconds, the mouse is clicked
- At 10 ms, the timeout timer and the interval expire

Under normal circumstances, if there were no code currently under execution, we'd expect the mouse click handler to be executed immediately at 6 ms, and the timer handlers to execute when they expire at 10ms. Note, however, that none of these handlers can execute at those times because the initial block of code is still executing. Due to the single-threaded nature of JavaScript, the handlers are queued in order to be executed at the next available moment.

When the initial block of code ends execution at 18 ms, there are three code blocks queued up for execution: the click handler, the timeout handler, and the first invocation of the interval handler. We'll assume that the browser is going to use a FIFO technique (first in, first out) – but remember, the browser may choose a more complicated algorithm if it so chooses – and so the waiting click handler (which we'll assume takes 10 ms to execute) begins execution.

While the timeout handler is executing, the second interval expires at 20 ms. Again, because the thread is occupied executing the timeout handler, the interval handler cannot execute. But this time, because an instance of an interval callback is already queued and awaiting execution, this invocation is dropped. The browser will not queue up more than one instance of a specific interval handler.

The click handler completes at 28 ms, and the waiting timeout handler, which we expected to run at the 10 ms mark, actually ends up starting at the 28 ms mark. That's what was meant earlier by there being no guarantee that the delay that is specified in any way can be counted on to determine exactly when the handler will execute.

At 30 ms, the interval fires again, but once more, no additional instance is queued because there is already a queue instance for this interval timer.

At 34 ms, the timeout handler finishes, and the queued interval handler begins to execute. But that handler takes 6 ms to execute, so while it is executing, another interval expires at the 40 ms mark, causing the this invocation of the interval handler to be queued. When the first invocation finishes at 42 ms, this queued handler executes.

This time, the handler finishes (at 47 ms) before the next interval expires at 50 ms. So the fifth firing of the interval does not have its handler queued, but executes as soon as the interval expires.

The important concept to take away from this is that, because JavaScript is single-threaded, only one unit of execution can ever be running at a time, and that we can never be certain that timer handlers will execute exactly when we expect.

This is especially true of interval handlers. We saw in this example, that even though we scheduled an interval that we expected to fire at the 10, 20, 30, 40 and 50 ms marks, only three of those instances executed at all, and at the 35, 42, and 50 ms marks.

As we can see, intervals have some special considerations that do not apply to timeouts. Let's look at those a tad more closely.

6.1.3 Differences between timeouts and intervals

At first glance, an interval may just seem to be a timeout that periodically repeats itself. But the differences are a little deeper than that. Let's take a look at an example to better illustrate the differences between `setTimeout` and `setInterval`; see listing 6.1.

Listing 6.1: Two ways to create repeating timers

```
setTimeout(function() {                          //#1
  /* Some long block of code... */               //#1
  setTimeout(arguments.callee, 10);              //#1
}, 10);                                          //#1

setInterval(function() {                         //#2
  /* Some long block of code... */               //#2
}, 10);                                          //#2
```

#1 Sets up a timeout that reschedules itself
#2 Sets up an interval

The two pieces of code in listing 6.1 may *appear* to be functionally equivalent, but they are not. Notably the `setTimeout` variant of the 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.

Recall from the example of the previous section how the timeout callback is never guaranteed to execute exactly when it is fired. So rather than being fired every 10 ms, as the interval is, it will reschedule itself for 10 ms after it gets around to executing.

Let's recap:

- JavaScript engines execute only a single thread at a time, forcing asynchronous events to queue up, awaiting execution.
- If a timer is blocked from immediately executing, it will be delayed until the next available time of execution (which may be longer, but never shorter, than the specified delay).
- Intervals may end up executing back-to-back with no delay if they get backed up enough, and multiple instances of the same interval handler will never be queued up.

- `setTimeout()` and `setInterval()` are fundamentally different in how their firing frequency are determined.

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

In this section we used delay values that are fairly small; 10 ms showed up a lot, for example. We'd like to find out whether or not those values are overly optimistic, so let's turn our attention to examining the granularity with which we can specify those delays.

6.2 Minimum timer delay and reliability

While it's pretty obvious that we can specify timer delays of seconds, minutes, hours – or whatever interval values we desire – what isn't obvious is what the smallest practical timer delay that we can choose might be.

At a certain point, a browser is simply incapable of providing a fine-enough resolution on the timers in order to handle them accurately as they, themselves, are restricted by the timing restrictions of the operating system.

Up until just a few years ago, specifying delays as short as 10 ms was rather laughably overoptimistic. But there's been a lot of recent focus on improving the performance of JavaScript in the browsers, so we put it to the test. We set off an interval timer, specifying a delay of 1 ms, and for the first 100 "ticks" of the timer, measure the actual delay between interval invocations.

The results are displayed in figures 6.2 and 6.3.

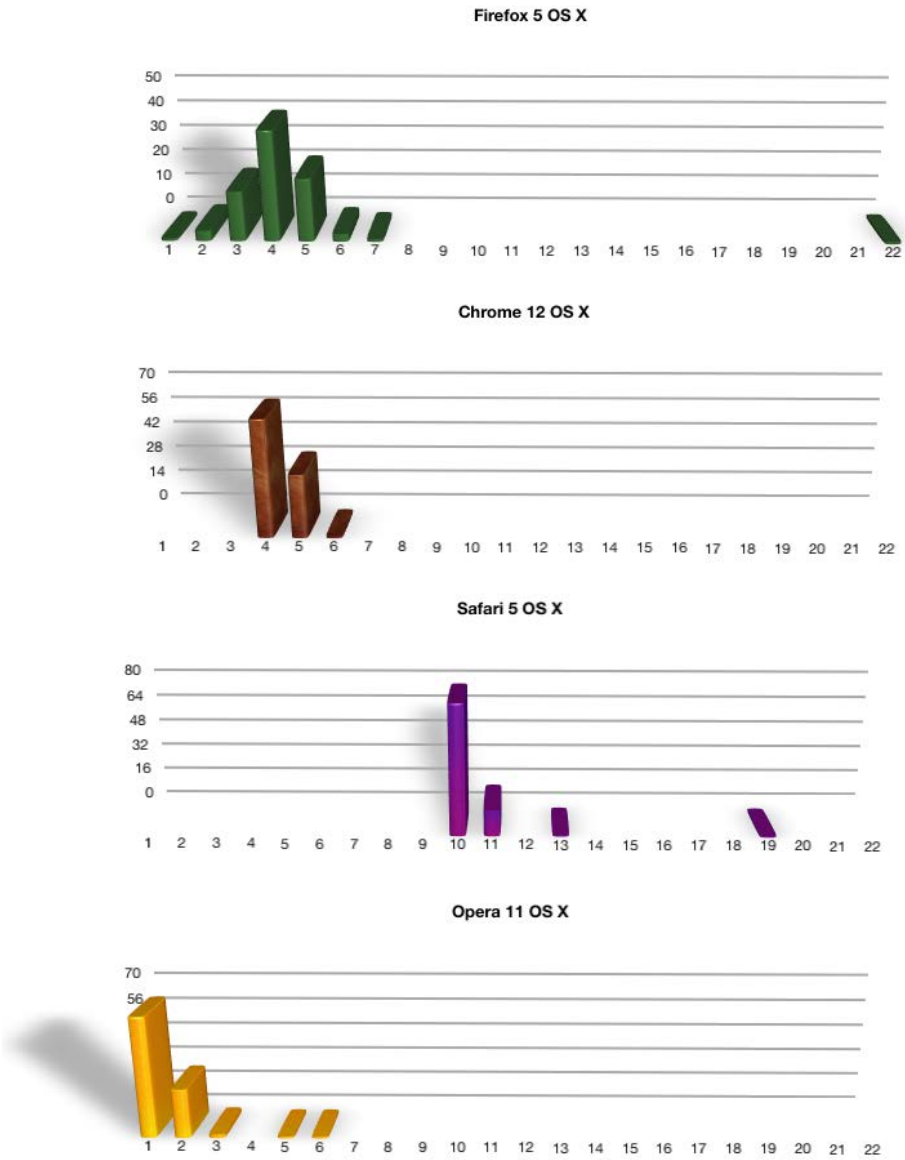

Figure 6.2: Interval timer performance measured on OS X browsers shows that some browsers get pretty close to 1 ms granularity

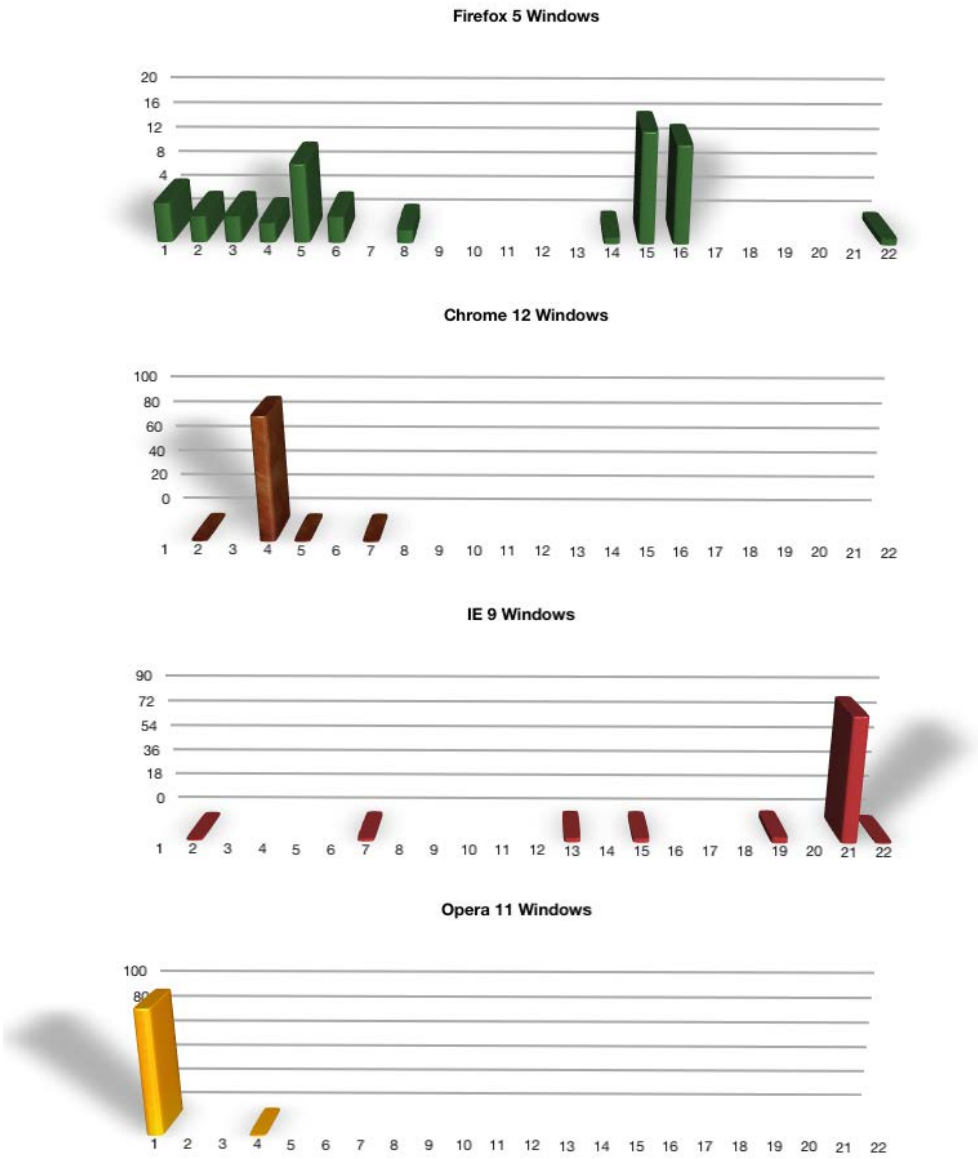

Figure 6.2 Interval timer performance as measured on Windows browsers is equally all over the place

These charts plot the number of times, out of the 100 tick run, that each interval value was achieved.

Under OS X, for Firefox we found that the average value was around 4 ms, almost always with some much longer outlying results, such as the single interval that took 22 ms. Chrome was much more consistent and also averaged around 4 or 5 ms, while Safari was rather slower, averaging out at 10 ms. Opera 11 proved to be the fastest browser with a whopping 56 intervals out of 100 taking the prescribed 1 ms delay.

The Windows results showed Firefox once again the most sporadic, with results all over the board and no clear peak. Chrome fared well with an average 4 ms, while IE 9 clocked in with a rather miserable peak at 21 ms. Opera once again took the commanding lead delivering all but one interval in the specified 1 ms.

NOTE

These tests were conducted on a MacBook Pro with a 2.5 GHz Intel Core 2 Duo processor, and 4GB of RAM running OS X 10.6.7, and a Windows 7 laptop with an Intel Quad Q9550 2.83GHz processor and 4GB RAM.

We can draw the conclusion that modern browsers are generally not yet able to realistically and sustainably achieve interval delays to the granularity level of 1 ms, but some of them are getting really, really close.

In our tests, we specified a delay of 1 ms, but you can also specify a value of 0 to get the smallest possible delay. There is one catch, though: Internet Explorer fumbles when we provide a 0 ms delay to `setInterval()`; whenever a 0 ms delay is specified for `setInterval()`, the interval executes the callback only once, just as if we had used `setTimeout()` instead.

There are a few other things that we can learn from these charts. The most important is simply a reinforcement of what we learned previously: browsers do not guarantee the exact delay interval that we specify. So while specific delay values can be asked for, the exact accuracy is not always guaranteed, especially with smaller values.

This needs to be taken into account in our applications when using timers. If the difference between 10 ms and 15 ms is problematic, or you require finer granularity than the browsers are capable of delivering, then you might have to rethink your approach, as the browsers just aren't capable of delivering that accurate a level of timing.

With all that under our belts, let's take a look at how our understanding of timers can help us avoid some performance pitfalls.

6.3 Dealing with computationally-expensive processing

The single-threaded nature of JavaScript is probably the largest "gotcha" in complex JavaScript application development. While JavaScript is busy executing, user interaction in the browser can become, at best, sluggish and, at worst, unresponsive. This can cause the browser to stutter or seem to hang, as all updates to the rendering of a page are suspended while JavaScript is executing.

Because of this, reducing all complex operations that take any more than a few hundred milliseconds into manageable portions becomes a necessity if we want to keep the interface responsive. Additionally, some browsers (such as Firefox and Opera) will produce a dialog warning the user that a script has become “unresponsive” if it has run non-stop for at least 5 seconds. Other browsers, such as that on the iPhone, will actually silently kill any script running for more than 5 seconds.

These outcomes are, obviously, less than desirable – producing an unresponsive user interface is never good. However there will almost certainly arise situations in which we'll need to process a significant amount of data; situations such as manipulating a couple of thousand DOM elements, for example.

This is a situation where timers can come to the rescue and become especially useful. As timers are capable of effectively suspending execution of a piece of JavaScript until a later time, they can also prevent the browser from hanging when the individual pieces of code aren't long enough to cause the browser to hang.

Taking this into account, we can convert intensive loops and operations into non-blocking operations.

Let's look at the example of listing 6.2, in which a task is likely to take a long time.

Listing 6.2: A long-running task.

```
var table = document.getElementsByTagName("tbody")[0];
for (var i = 0; i < 20000; 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);
}
```

In this example we're creating a total of 140,000 DOM nodes, populating a table with a large number of cells. This is incredibly expensive and will likely hang the browser for a noticeable period while executing, preventing the user from performing normal interactions. We can introduce timers into this situation to achieve a different, and better result, as shown in listing 6.3.

Listing 6.3: Using a timer to break up a long-running task.

```
<script type="text/javascript">

  var rowCount = 20000;                                    #1
  var divideInto = 4;                                      #1
  var iteration = 0;                                       #1

  var table = document.getElementsByTagName("tbody")[0];

  setTimeout(function(){
    var base = (rowCount / divideInto) * iteration;        #2
```

```
    for (var i = 0; i < rowCount / divideInto; i++) {
      var tr = document.createElement("tr");
      for (var t = 0; t < 6; t++) {
        var td = document.createElement("td");
        td.appendChild(
          document.createTextNode((i + base) + "," + t +
                                  "," + iteration));
        tr.appendChild(td);
      }
      table.appendChild(tr);
    }
    iteration++;                                              #3
    if (iteration < divideInto)                               #3
      setTimeout(arguments.callee,0);                         #3
  },0);

</script>
```

#1 Sets up the parameters
#2 Computes where we left off last time
#3 Schedules the next phase

In this modification to our example we've broken up our lengthy operation into 4 smaller operations, each creating 35,000 DOM nodes. These smaller operations are much less likely to interrupt the flow of the browser.

Note how we've set it up so that the parameters controlling the operation are collected into easily tweak-able variables (#1) should we find that we need to break the operations up into, let's say, 10 parts instead of 4.

Also important to note is the little of math we needed to do to keep track of where we left off in the previous iteration (#2), and how we automatically schedule the next iteration until we determine that we're done (#3). (Our knowledge of the `arguments.callee` property comes in mighty handy here!)

What's rather impressive is just how little our code had to change in order to accommodate this new, asynchronous approach. We have to do a *little* more work in order to keep track of what's going on, ensure that the operation is correctly conducted, and to schedule the execution parts. But beyond that, the core of the code looks very similar to what we started off with.

The most perceptible change made evident by adopting this technique, as compared to the original example, is that a long browser hang is now replaced with four (or however many we choose) visual updates of the page. For while the browser will attempt to execute our code segments as quickly as possible, it will also render the DOM changes after each step of the timer. In the original, it needed to wait for one large bulk update.

Much of the time it's imperceptible to the user to see these types of updates occur, but it's important to remember that they do occur, and we should strive to make sure that any code we introduce into the page does not perceptibly interrupt the normal operation of the browser.

One situation in which this technique has served one of your authors particularly well was in an application 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 it was converted to move all schedule computation to the client side. A view of the schedule computation screen can be seen in figure 6.4.

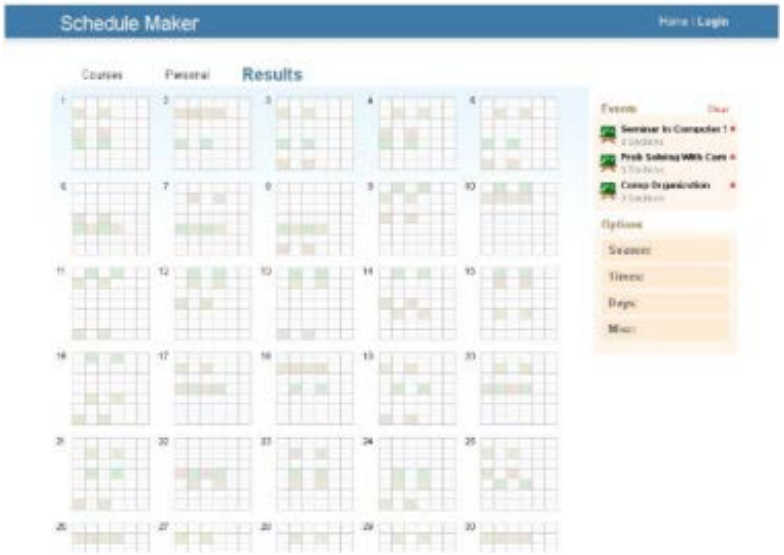

Figure 6.4: A web-based schedule generation application with client-side computation

These operations were quite expensive (running through thousands of permutations in order to find correct results). The resulting performance problems were solved by breaking up clumps of schedule computation into tangible bites, updating the user interface with a percentage of completion as it went along. 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 such as 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 single-threaded browser environment using timers, while still providing a useful experience to the user.

But all is not completely rosy; handling large numbers of timers can get unwieldy. Let's see what we can do about that.

6.4 Central timer control

A problem that can arise in the use of timers is in how to manage them when dealing with a large number of timers. This is especially critical when dealing with animations as we'll likely

be attempting to manipulate a large number of properties simultaneously, and we'll need a way to manage that.

Many multiple timers are problematic for a number of reasons. There's not only the issue of needing to retain references to lots of interval timers that, sooner or later, must be canceled (though we know how to help tame that kind of mess with closures), but also of interfering with the normal operation of the browser. We saw previously that, by making sure that no one timer handler invocation performs excessively lengthy operations that we can prevent our code from blocking other operations, but there are other browser considerations. One of these regards garbage collection.

It's important to realize that firing off a large number of simultaneous timers is likely to increase the likelihood of a garbage collection task occurring in the browser. Garbage collection, roughly speaking, is when the browser goes through its allocated memory and tries to tie up any loose ends by removing unused variables, and objects. Timers are particularly a problem as they are generally managed outside of the flow of the normal single-threaded JavaScript engine (through other browser threads).

While some browsers are more capable of handling this situation, others can exhibit long garbage collection cycles. You might have noticed this when you see a nice, smooth animation in one browser, but 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 situation, and is the reason why all modern animation engines utilize a technique called a ***central timer control***.

Having a central control for our timers gives us a lot of power and flexibility, namely:

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

Let's take a look at an example that uses this technique for managing multiple functions that are animating separate properties. First, we'll create a facility for managing multiple handler functions with a single timer. Consider the code of listing 6.4.

Listing 6.4: A central timer control to manage multiple handlers

```
var timers = {                                                    //#1

  timerID: 0,                                                     //#2
  timers: [],                                                     //#2

  add: function(fn) {                                             //#3
    this.timers.push(fn);
  },

  start: function() {                                             //#4
    if (this.timerID) return;
    (function() {
```

```
      if (timers.timers.length > 0) {
        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() {                                          //#5
    clearTimeout(this.timerID);
    this.timerID = 0;
  }

};
```

#1 Declares timer control object
#2 Records state
#3 Creates function to add handlers
#4 Creates function to start timer
#5 Creates function to stop timer

In listing 6.4 we've created a central control structure (#1) to which we can add any number of timer callback functions, and through which we can start and stop their execution. Additionally, we allow the callback functions to remove themselves at any time by simply returning `false`,which is much more convenient than the typical `clearTimeout()` call).

Let's step through the code to see how it works.

To start, all of the callback functions are stored in an array named `timers` along with the ID of any current timer (#2). These variables constitute the only state that our timer construct needs to maintain.

The `add()` method accepts a callback handler (#3), and simply adds it to the `timers` array.

The real meat comes in with in the `start()` method (#4). In this method, we first verify that there isn't already a timer running (by checking if the `timerID` member has a value), and if we're in the clear, we execute an immediate function to start our central timer.

Within the immediate function, if there are any registered handlers, we run though a loop and execute each handler. If a handler returns `false`, we remove it from the array of handlers, and schedule the next "tick" of the animation.

Putting this construct to use, we create an element to animate:

```
<div id="box">Hello!</div>
```

Then, we start the animation with:

```
var box = document.getElementById("box"), x = 0, y = 20;

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

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

timers.start();
```

We get a reference to the element, add a handler that moves the element horizontally, another handler that moves it vertically, and start the whole shebang.

The result, after the animation completes, is shown in figure 6.5.

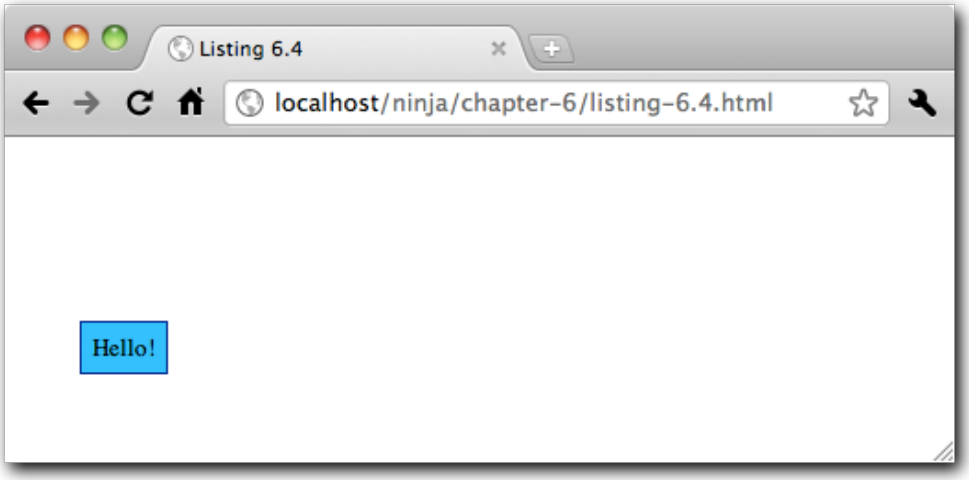

Figure 6.5: After running multiple animation handlers, the element has moved down and across the page

An important point 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, where the browser could choose to execute one before another.

This manner of timer organization is critical for large applications, or any form of JavaScript animations. Having a solution in place will certainly help in any form of future application development and especially when creating animations.

In addition to animations, central timer control can help us on the testing front. Let's see how.

6.5 Asynchronous testing

Another situation in which a centralized timer control comes in mighty handy is when we wish to perform asynchronous testing. The issue here is that when we want to perform

testing on actions that may not complete immediately (such as handlers for a timer, or even an XMLHttpRequest) we need to break our test suite out such that it works completely asynchronously.

As we saw in test examples in the previous chapters, we can easily just run the tests as we come to them, and most of the time, this is fine. However once we introduce the desire to do asynchronous testing we need to break all of those tests out and handle them separately. Listing 6.5 has some code that does so. You should not be surprised to find that this code looks somewhat familiar.

Listing 6.5: A simple asynchronous test suite

```
<script type="text/javascript">

  (function() {

    var queue = [], paused = false;                 //#1

    this.test = function(fn) {                      //#2
      queue.push(fn);
      runTest();
    };

    this.pause = function() {                       //#3
      paused = true;
    };

    this.resume = function() {                      //#4
      paused = false;
      setTimeout(runTest, 1);
    };

    function runTest() {                            //#5
      if (!paused && queue.length) {
        queue.shift()();
        if (!paused) resume();
      }
    }
  })();
</script>
```

#1 Retains state
#2 Defines test registration function
#3 Defines function to pause testing
#4 Defines resume function
#5 Runs tests

The single most important aspect in listing 6.5 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, very similar to the code we introduces with listing 6.4, necessary to make this behavior possible. The bulk of the functionality is contained within the `resume()` and `runTest()` functions. It behaves very similarly to the `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 it completely stops the interval from running. The important point 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 the test 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 reliable asynchronous testing to an existing test suite, with the effective use of timers.

6.6 Summary

Learning about how JavaScript timers function has been illuminating! Seemingly simple features, timers are actually quite complex in their implementation. Taking all their intricacies into account, however, gives us great insight into how we can best exploit them for our gain.

In this chapter, it's become apparent that timers end up being especially useful in complex applications including computationally-expensive code, animations, or 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 manage.

So far, we've discussed a number of features and techniques that we can use to create sophisticated code while keeping its complexity in check. In the next chapter we'll take a look at another feature of JavaScript that is rather underused, but can help us reign in what might be otherwise overly complex code: regular expressions.

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.

7.1 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 7.1.

Listing 7.1: 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 7.2.

Listing 7.2: 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 7.2. 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 7.3.

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 7.3: 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.

7.2 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 7.4 we extract the opacity value out of the filter value that Internet Explorer provides:

Listing 7.4: 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 7.5.

Listing 7.5: 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 7.6.

Listing 7.6: 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.

7.2.1 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 7.6 (where we match an opening, or closing, HTML tag) and modify it to, also, match the inner contents of the tag, itself, in Listing 7.7.

Listing 7.7: 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 7.7 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 7.8.

Listing 7.8: 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.

7.2.2 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 7.9.

Listing 7.9: 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!

7.3 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 7.10 we use the function to provide a dynamic replacement value to adjust the case of a character.

Listing 7.10: 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 7.11.

Listing 7.11: 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 7.11 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 7.12.

Listing 7.12: 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.

7.4 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.

7.4.1 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 7.13.

Listing 7.13: 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 7.14.

Listing 7.14: 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 7.15.

Listing 7.15: 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 7.1.

Table 7.1: All time in ms, for 1000 iterations.

	Selector Trim	Document Trim
Listing 7.13	8.7	2075.8
Listing 7.14	8.5	3706.7
Listing 7.15	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.

7.4.2 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 7.16.

Listing 7.16: 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`).

7.4.3 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-sighted. 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 7.17.

Listing 7.17: 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.

7.4.4 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 7.18.

Listing 7.18: 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).

7.5 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.

8

Developing cross-browser strategies

In this chapter:

- Strategies for developing reusable, cross-browser JavaScript code.
- Analyzing the issues needing to be tackled
- Tackling those issues in a smart way

Anyone who's been developing on-page JavaScript code for more than five minutes knows that there are a wide range of points and counter-points that need to be taken into consideration in order to make sure that the code works flawlessly across the field of supported browsers. These considerations span everything from the basic development for the immediate needs, to planning for future browser releases, all the way to the reuse of the code on web pages that have yet to be envisioned.

This is certainly a non-trivial task, and one that must be balanced according to the development methodologies that you have in place, as well as the resources available to your project. As much as we'd love for our pages to work perfectly in every browser that ever existed or will ever exist, reality rears its ugly head and we must realize that we only have a finite amount of development resources. Therefore, we must intelligently plan to apply those resources appropriately and carefully, extracting the best practical results from our efforts.

This starts with choosing our browsers carefully.

8.1 Choosing which browsers to support

The primary concern that we need to take into account, when deciding where to direct our limited resources, is deciding which browsers will be the primary targets that we will support with our code.

As with virtually any aspect of web development, we need to carefully choose the browsers upon which we'll want our users to have an optimal experience. When we choose to support a browser, we are typically making the following promises:

1. That we'll actively test against that browser with our test suite.
2. That we'll fix bugs and regressions associated with that browser.
3. That the browser will execute our code with a reasonable level of performance.

As an example, most JavaScript libraries end up supporting about a dozen or so browsers. This set is usually the previous release, the current release, and the upcoming beta release (if available) of the *Big Five* browsers: Internet Explorer, Firefox, Safari, Chrome and Opera.

That's actually an enormous browser set to support. The mainstream JavaScript libraries have the luxury of large staff (even if most are volunteers) that the average page author does not have at his or her disposal. So, realistic choices must be made regarding which browsers to support.

Of course, we can choose to leverage the work already done by the big JavaScript libraries and automatically gain browser support, but this book doesn't make the assumption that you'll be using alibrary and aims to help you choose which browsers to support in your own code.

You might want to make a "browser support matrix", as shown in figure 8.1, and fill it in for your own purposes. (The selections in this figure are just an example and do not reflect any judgment values on the depicted browsers.)

The remainder of this chapter should help you decide which boxes to check, and which to "x out".

Figure 8.1: An example "browser support" chart – fill one in with your own decisions!

Any piece of reusable JavaScript code, to include mass-consumption JavaScript libraries through your own on-page code, should be developed to work in as many environments as feasible, but it's important not to bite off more than can be chewed, and quality should never be sacrificed for coverage.

That's important enough to repeat; in fact, we urge you to read it out loud: *Quality should never be sacrificed for coverage.*

In order to understand what we're up against, we're going to examine the different situations that JavaScript code will find itself up against throughout this chapter, and then examine some of the best ways to write that code with the aim of assuaging any potential problems that those situations pose.

This should go a long way in helping you decide which of these techniques it is worth your time to adopt, and help you fill out your own browser-support chart.

8.2 The five major development concerns

With any piece of non-trivial code, there are myriad development concerns for us to worry about. But there are about five major generally accepted points which pose the biggest challenges to our reusable JavaScript code.

The five points of concern for JavaScript development are shown in figure 8.2.

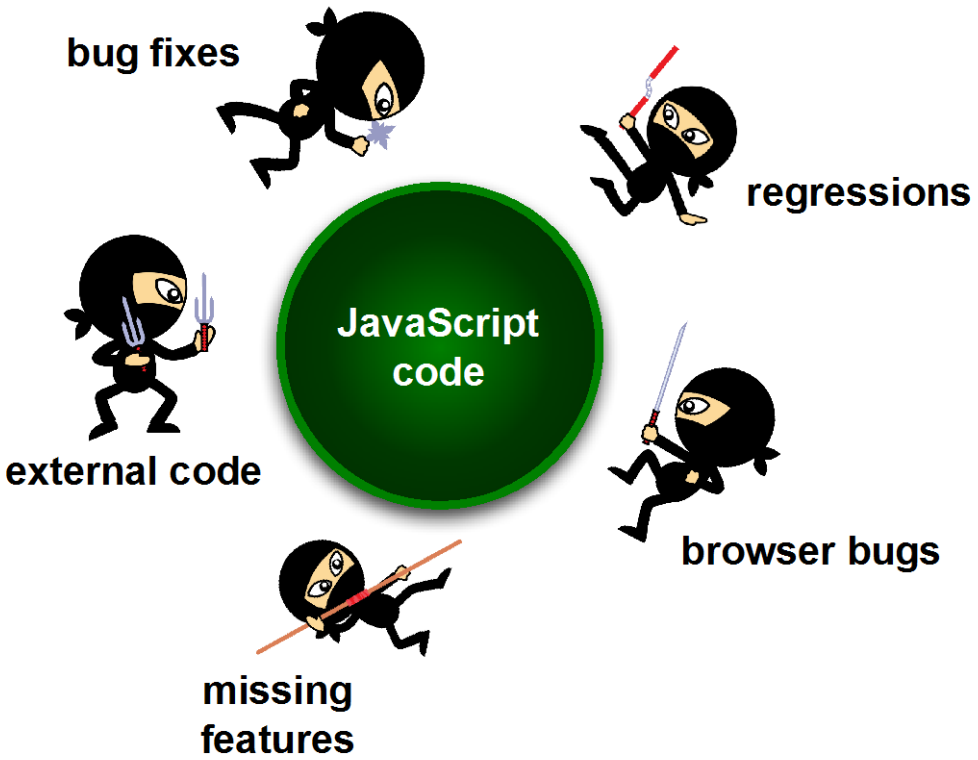

Figure 8.2 The five major points of concern for development of reusable JavaScript

The points are:

1. Browser bugs
2. Bug fixes
3. Missing features
4. External code
5. Browser regressions

We'll want to balance how much time we spend on each point with how much benefit we get as an end result. For example, is an extra 40 hours of development time worth better support for an antiquated (and unsupported) browser such as IE6?

Ultimately these are questions that you'll have to answer yourself, applying them to your own situation. The answer to the previous question could be radically different for web

applications destined for general Internet access, versus an in-house application used by workers chained to IE6 by a Luddite IT department!

Analysis of our intended audience, our development resources, and schedule are all factors that go into our decisions. There's one axiom that can be used when pondering these points: *remember the past, consider the future, test the present*.

When striving to develop reusable JavaScript code, we must take all of the points into consideration, but paying closest attention to the most-popular browsers that exist right now. Then, we'll have to take into concern the changes that are coming in the next versions of the browsers. And then we'll try to maintain compatibility with older browser versions, supporting as many features as we can without sacrificing quality or features for the entire support set.

In the following sections, we'll break down these various concerns so that we can have a better understanding of what we're up against, and how to combat them.

8.2.1 Browser bugs and differences

A primary concern we'll need to deal with when developing reusable JavaScript code is the handling of the various browser bugs and API differences associated with the set of browsers that we've decided to support.

This means that any features that we provide in our code should be completely and *verifiably* correct in all of those browsers.

The means to achieving this is quite straight-forward, having already been presented in chapter 2: we need a comprehensive suite of tests to cover both the common and fringe use cases of the code. With good test coverage, we can feel safe in knowing that the code that we develop will work in the supported set of browsers. And, assuming no changes that break backwards compatibility, we'll have a warm fuzzy feeling that our code will even work in future versions of those browsers.

We'll be looking at specific strategies for dealing with browser bugs and differences in section 8.3.

A tricky point in all of this is: how can we implement fixes for current browser bugs in a way that is resistant to the fixes for those bugs that will be implemented in future versions of the browser?

8.2.2 Bug Fixes

Assuming that a browser will forever present a bug is rather foolhardy – most bugs eventually get fixed – and is a dangerous development strategy. It's best to use the techniques that we'll discuss in section 8.3 to make sure that we future-proof our workarounds as much as possible.

When writing a piece of reusable JavaScript code, we want to make sure that it's able to last for a good long time. As with writing any aspect of a web site (CSS, HTML, etc.), it's undesirable to have to go back and fix broken code caused by a new browser release.

Making assumptions about browser bugs causes a common form of web site breakage. That is: specific hacks put in place to workaround bugs presented by a browser that break when the browsers fix the bugs in future releases.

The issue can be circumvented by building pieces of feature simulation code (which we'll discuss at length in section 8.3.3) instead of making assumptions about the browser.

The problem with handling browser bugs callously is two-fold:

1. Firstly, our code is liable to break when the bug fix is eventually instituted.
2. Secondly, we can actual train browser vendors to *not* fix 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 created within one document to be *adopted* by another DOM document if they were going to be injected into the other document (which is in accordance with the DOM specification), like in Listing 10.3.

The following bit code should actually not work:

```
var node = documentA.createElement("div");
documentB.documentElement.appendChild( node );
```

The proper way, should be:

```
var node = documentA.createElement("div");
documentB.adoptNode(node);
documentB.documentElement.appendChild(node);
```

However, since there was a bug in Firefox that allowed the first situation to work – when it shouldn't have – users wrote their code in a manner that depended upon that code working. This forced Mozilla to rollback their change, for fear of breaking a number of web sites.

This brings up another important point concerning bugs: when determining if a piece of functionality is potentially a bug, always verify it with the specification. In the above case, Internet Explorer was actually more forceful (throwing an exception if the node wasn't in the correct document - the correct behavior), but users just assumed that it was an error with Internet Explorer, and in that case, wrote conditional code to provide a fallback. 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 also be differentiated from an unspecified API. It's important to refer back to browser specifications since those are the exact standards that the browsers use 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 standardized and changes in the process). In the case of inconsistencies in unspecified APIs, you should always test for your expected output, running additional cases of feature simulation. And always be aware that 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 in 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 also 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, we test every single API that we ever touch - but the overhead incurred in such an action would be ludicrous). API changes in this manner should be handled like any other regression.

For our next point of concern, we know that no man is an island, and neither is our code. Let's explore the ramifications of that.

8.2.3 Living with external code and markup

Any reusable code must co-exist with the code that surrounds it. Whether we're expecting our code to work within pages that we write ourselves, or on web sites developed by others, we need to ensure that it is able to co-exist with any other random code sharing the same page.

And this is a double-edged sword: our code must not only be able to withstand living with poorly written external code, it must itself take care not to have adverse effects on the code with which it lives.

Exactly how much we need to be vigilant about this point of concern depends a great deal upon the environment in which we expect the code to be used. For example, if we are writing reusable code for a single or limited number of web sites that we have some level of control over, it might be safe to worry less about any effects from external code as we know what the end environment within which the code will operate will be, and have some level of control to fix any problems ourselves.

However if we're developing code that will have a broad level of applicability in unknown (and uncontrollable) environments, we'll need to make doubly sure that our code is robust.

Let's discuss some strategies to achieve that.

ENCAPSULATING OUR CODE

To keep our code from affecting other pieces of code on the pages upon which it is loaded, it's best to practice ***encapsulation***.

One dictionary definition of encapsulation reads "to place in or as if in a capsule", while a more domain-focused definition could be "a language mechanism for restricting access to some of the object's components". Your Aunt Mathilda might summarize it more succinctly as "keep your nose in your own business!"

Keeping an incredibly small global footprint when introducing our code into a page can go a long way to making Aunt Mathilda happy. In fact, keeping our global footprint to a handful of global variables, or better yet *one*, is actually fairly easy.

The jQuery library is a good example of this. It introduces one global variable (a function) named `jQuery`, and one alias for that global variable, `$`. It even has a supported means to give up the $ alias back to whatever other on-page code or other library may want to use it.

Almost all operations in jQuery are made via the `jQuery` function. And any other functions that it provides (so-called *utility functions*) are defined as properties of `jQuery` (remember from chapter 3 how easy it is to define functions that are properties of other functions) thus using the name `jQuery` as a namespace for all its definitions.

We can use the same strategy.

Let's say that we are defining a set of functions for our own use, or for the use of others, that we'll group under a namespace of own choosing – let's say `ninja`.

We could, like jQuery, define a global function named `ninja()` that performs various operations based upon what we pass to the function. For example:

```
var ninja = function(){ /* implementation code goes here */ }
```

Defining our own utility functions that use this function as their namespace is as easy as:

```
ninja.hitsuke = function(){ /* code to distract guards with fire here */ }
```

If we didn't want or need `ninja` to be a function and to just serve as a namespace, we could define it as:

```
var ninja = {};
```

This creates an empty object upon which we can define properties, functions and otherwise, in order to keep from adding any other names to the global namespace.

Other activates that we wish to avoid, in order to keep our code encapsulated, are modifying any existing variables, function prototypes, or even DOM elements. Any aspect of the page that our code modifies, outside of itself, is a potential area for collision and confusion.

The other side of that two-way street is that, even if we follow best practices and carefully encapsulate our code, we cannot be assured that code that we haven't written ourselves is going to be as well behaved.

DEALING WITH LESS-THAN-EXEMPLARY CODE

There's an old joke that's been going around likely since Grace Hopper removed that moth from a relay back in the Cretaceous period that "the only code that doesn't suck is the code you write". While your authors do not ascribe to that cynical view, when our code co-exists with code over which we have no control, we should assume the worst, just to be safe.

Other code – even if well written, rather than just buggy – might intentionally be doing things like modifying function prototypes, object properties, as well as DOM element methods.

In such circumstances, our code could be doing something innocuous such as using JavaScript arrays, and no one could fault us for making the simple assumption that JavaScript arrays are going to act like JavaScript arrays. But if some other on-page code goes and modifies the manner in which arrays work, our code could not work as intended through no fault of our own!

Unfortunately, there aren't many steadfast rules when dealing when situations of this nature, but there are some steps we can take to mitigate these type of problems.

The next few sections will introduce these defensive steps.

AVOIDING IMPLANTED PROPERTIES

The first of these defensive steps is to take advantage of the `hasOwnProperty()` function. This function is inherited from Object by all JavaScript objects, and tests if the object possesses a specified property. This is similar to JavaScript's `in` operator, with the important difference that it does *not* check up the prototype chain.

We can therefore us this function to detect properties that have been introduced by an extension to `Object.prototype`.

We can observe the behavior of this function by inspecting the tests of listing 8.1.

Listing 8.1: Using hasOwnProperty() to test for inherited properties

```
<script type="text/javascript">

  Object.prototype.ronin = "ronin";             //#1

  var object = { ninja: 'value' };              //#2
  object.samurai = 'samurai';                   //#2

  assert(object.hasOwnProperty('ronin'),"ronin is a property");
  assert(object.hasOwnProperty('ninja'),"ninja is a property");
  assert(object.hasOwnProperty('samurai'),"samurai is a property");

</script>
```

#1 Sets up an inherited property
#2 Sets up non-inherited properties

The results of running the test are shown in figure 8.3.

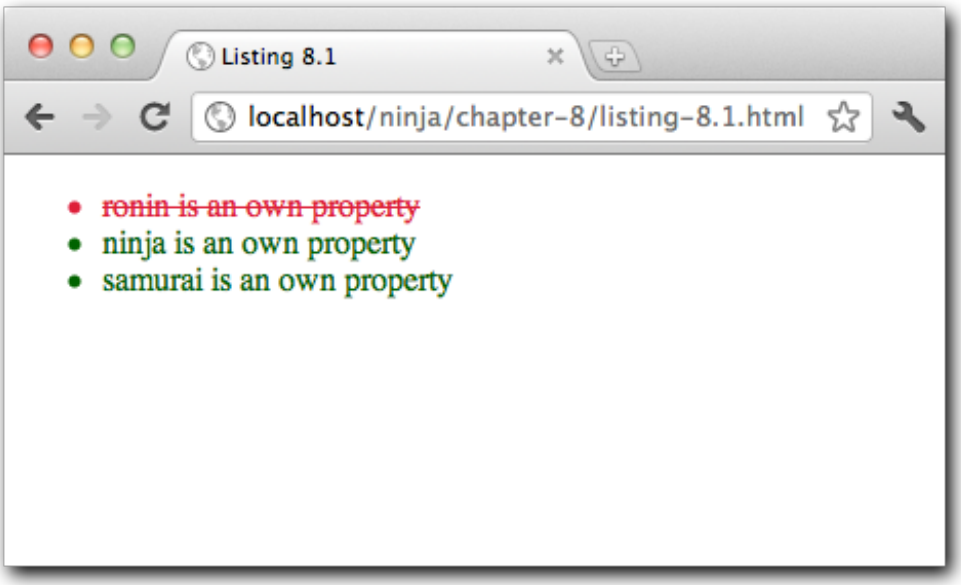

Figure 8.3 Results of tests show how we can use `hasOwnProperty()` can detect inherited properties

The test results clearly show that the `ronin` property, added to the Object prototype, is not considered an "own property" of the created objects.

Thankfully the number of scripts that use this technique is very small but the harm can be great if extra properties added to the prototype confuse our code. This can be especially problematic when iterating through the properties of an object using a for-in clause. But we can counter this complication by using `hasOwnProperty()` to determine if we should ignore a property or not.

COPING WITH GREEDY IDS

Both Opera and Internet Explorer exhibit an anti-feature (we can't really call it a *bug* because the behavior is as intended) that can cause our code to trip and fall unexpectedly. This feature causes element references to be added to other elements using the `id` of the original element. And when that `id` conflicts with properties that are already part of the element, bad things can happen.

Consider the following HTML snippet to observe what nastiness can ensue as a result of these so-called "greedy IDs":

```
<form id="form">
  <input type="text" id="length"/>
  <input type="submit" id="submit"/>
</form>
```

In the naughty browsers, if we were to call `document.getElementsByTagName("input").length`, we'd find that the `length` property, which we'd expect to contain a numeric length value, instead contains a reference to the input element with the `id` of length.

Additionally, if we were to all `document.getElementById("form").submit()`, that would also no longer work as the `submit()` method will have been overwritten by a reference to the input element with an `id` of `submit`.

This particular "feature" of the browsers can cause numerous and mystifying problems in our code, and will have to be kept in mind when debugging in these browsers. When we encounter properties that have seemingly been inexplicably transformed into something other than what we expect them to be, greedy IDs are a cause we will need to check for.

Luckily, we can avoid this problem in our own markup by avoiding simple `id` values that can conflict with standard property names. The name `submit` is especially to be avoided for `id` and `name` values as it is a common source of frustrating and perplexing buggy behavior.

ORDER OF STYLESHEETS

Often we expect CSS rules to already be available by the time our code executes. One of the best ways to ensure that CSS rules provided by stylesheets are defined when our JavaScript code executes is to specify the external stylesheets prior to including the external script files. Not doing so can cause unexpected results as the script attempts to access the as-yet-undefined style information. Unfortunately this isn't an issue that can be easily rectified with pure JavaScript and should instead be handled with user documentation.

These sections covered just some basic examples of how externalities can affect how our code works – frequently in unintentional and confounding manners. Many times, issues with our code will pop up when other users try to integrate our code into *their* sites, at which point we should be able to diagnose the issue and build appropriate tests to handle them. At other times, we'll discover such problems when we integrate others' code into our pages, and hopefully the tips in these sections will help to identify what's causing the issues.

It's unfortunate that there are no better and deterministic solutions to handling these integration issues other than to take some smart first steps and to write our code in a defensive manner.

Now let's move on to the next point of concern.

8.2.4 Missing Features

For browsers that aren't lucky enough to survive our support matrix – and therefore benefiting from the testing that our code will get for the "A List" browsers – there are likely to be some missing key features that our code needs in order to operate as expected.

There may even be browsers that *are* on our support matrix that we need to support (perhaps for some political or business reasons) that lack key features.

Even if we are not going to give those browsers full support, especially those that did not make the cut, it'd be best if we could write our code defensively such that it degrades

gracefully, or provide some other type of fallback for end users who choose (or are forced) to use browsers other than those we have the resources to test upon.

Our strategy at this point becomes: how can we get the most functionality delivered to our users, while failing gracefully when we cannot. This is known as ***graceful degradation***.

Graceful degradation should be approached cautiously, and with due consideration. Take the case where a browser is capable of initializing and hiding a number of pieces of navigation on a page, perhaps in hopes of creating a drop-down menu, but the event-related code to power the menu doesn't work. The result is a half-functional page, which helps no one.

A better strategy would be to design our code to be as backwards-compatible as possible and to actively direct known failing browsers to an alternate version of the page or site tailored to the capabilities of the lesser browser. Yahoo! adopts this strategy with most of their web sites, breaking down browsers into graded levels of support. After a certain amount of time they "blacklist" a browser (usually when it hits an infinitesimal market-share such as 0.05%) and direct users of that browser (based upon the detected user agent) to a pure-HTML version of the application (one with no CSS or JavaScript involved).

This means that their developers are able to provide an optimal experience for the vast majority of their users (around 99%), by passing off antiquated browsers to a functional equivalent (albeit with a less modern 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, or not at all.
- All users of current and past browsers are guaranteed to have a page that isn't broken.
- Future/unknown browsers are assumed to work.

The primary downside of this strategy is that extra development effort (beyond the currently-targeted browsers and platforms) must be expended to focus on handling future browsers. Despite the cost, this is a smart strategy as it will allow your applications to stay viable longer with only minimal updates and changes.

8.2.5 Regressions

Regressions are one of the hardest problems that we will encounter in the creation of reusable and maintainable JavaScript code. These are bugs, or non-backward-compatible API changes, that browsers have introduced that cause our code to break in unpredictable ways.

There are some API changes that, with some foresight, we can proactively detect and handle. For example, with Internet Explorer 9, Microsoft introduced support for DOM Level 2 event handlers (bound using the `addEventListener()` method). For code written prior to IE9, simple object detection was able to handle that change, as shown in Listing 8.2.

Listing 8.2: Anticipating an upcoming API change

```
function bindEvent(element, type, handle) {
  if (element.addEventListener) {
    element.addEventListener(type, handle, false);                //#1
  }
  else if (element.attachEvent) {
    element.attachEvent("on" + type, handle);                     //#2
  }
}
```

#1 Binds using standard API
#2 Binds using proprietary API

In this example, we future-proofed our code knowing (or hoping against hope) that someday, Microsoft would bring Internet Explorer into line with DOM standards. If the browser supports the standards-compliant API, we user object detection to infer that and use the standard API (#1). If not, then we check to see if the IE-proprietary method is available, and use that (#2). All else failing, we simply do nothing.

However most API future changes aren't that easy to predict and, of course, there is no way to predict upcoming bugs. This is but one of the very important reasons that we have stressed testing throughout this book. In the face of unpredictable changes that will affect our code, the best that you can hope for is to be diligent in monitoring our tests for each browser release, and to quickly address issues that regressions may introduce..

Let's consider an example of an unpredictable bug: Internet Explorer 7 introduced a basic XMLHttpRequest wrapper around the native ActiveX request object. As a result, virtually all JavaScript libraries opted to default to using the XMLHttpRequest object to perform their Ajax requests (as they should – choosing to use a standards-based API is nearly always preferred).

However, in Internet Explorer's implementation, Microsoft 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 predicted 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.

Having a good suite of tests, and keeping close track of upcoming browser releases is absolutely the best way to deal with future regressions of this nature. It doesn't have to be taxing on your normal development cycle – which should already include routine testing. Running these tests on new browser releases should always be factored into the planning of any development cycle.

You can get the upcoming browser releases from the following locations:

- Internet Explorer: 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/
- Chrome: http://chrome.blogspot.com/

Diligence is important in this respect. Since we can never fully predict the bugs that will be introduced by a browser, it's best to make sure that we stay on top of our code and quickly 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 won't catch all regressions (and certainly won't in all browsers) it's a great start and shows good progress by the browser vendors towards preventing as many issues as possible.

8.3 Implementation strategies

Knowing which issues to be aware of is only half the battle – figuring out effective strategies for dealing with them, and implementing robust cross-browser code, is another matter.

There are a wide a range of strategies that we can use, and while not every strategy will work in every situation, the range that we'll examine should provide a good set of tools for covering most of the concerns that we need to address within our code bases.

8.3.1 Safe cross-browser fixes

The simplest (and safest) class of cross-browser fixes are those that exhibit two important traits:

1. They have no negative effects or side effects on other browsers
2. They use no form of browser or feature detection

The instances in which we can apply such fixes may be generally rather rare, but they're a tactic that we should always strive for in our applications. To give an example, examine the following code snippet:

```
// ignore negative width and height values
if ((key == 'width' || key == 'height') && parseFloat(value) < 0)
  value = undefined;
```

This code represents a change (plucked from jQuery) that came about when working with Internet Explorer. Some versions of that browser throw an exception when a negative value is set on the `height` or `width` style properties. All other browsers simply ignore negative input.

The workaround, shown above, 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 change that provided a unified API to the user (as throwing unexpected exceptions is never desired).

Another example of this type of fix (also from jQuery) appears in the attribute manipulation code. Consider:

```
if (name == "type" &&
    elem.nodeName.toLowerCase() == "input" &&
    elem.parentNode)
  throw "type attribute can't be changed";
```

Internet Explorer doesn't allow us to manipulate the `type` attribute of input elements that are already part of the DOM; attempts to change this attribute result in a proprietary exception being thrown. jQuery came to a middle-ground solution: simply disallow *all* attempts to manipulate the `type` attribute on injected input elements in all browsers equally, throwing a uniform 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. The action still results in an exception, but that exception is uniform across all browser types.

Certainly this particular approach could be considered quite controversial – it purposefully limits the features of the library in all browsers because of a bug that exists in only one. The jQuery team weighed this 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 code. It's very possible that you'll come across situations like this when developing your own reusable code bases, and will carefully need to consider whether a limiting approach such as this is appropriate for your audience.

The important thing to remember for these types of code changes: they provide a solution that works seamlessly across browsers without the need for browser or feature detection, effectively making them immune to changes going forward. One should always strive to use solutions that work in this manner when feasible, even if the applicable instances are few and far-between.

8.3.2 Object detection

As we've previously discussed, *object detection* is a commonly used approach when writing cross-browser code, being not only simple, but generally quite effective. It works by determining if a certain object or object property exists, and if so, assuming that it provides the implied functionality. (We'll see what to do about cases where this assumption fails in the next section.)

Most commonly object detection is used to choose between multiple APIs that provide duplicate pieces of functionality. For example, as in the code that we saw in listing 8.2 where object detection was used to choose the appropriate event-binding APIs provided by the browser, repeated here:

```
function bindEvent(element, type, handle) {
  if (element.addEventListener) {
    element.addEventListener(type, handle, false); }
  else if (element.attachEvent) {
    element.attachEvent("on" + type, handle); }
```

```
}
```

In this example we checked to see if a property named `addEventListener` exists, 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. We then proceed to test other APIs, such as `attachEvent`, for existence.

Note that we tested for `addEventListener`, the *standard* method provided by the W3C DOM Events specification, first. This is deliberate and intentional.

Whenever possible we should default to the standard way of performing any action. As mentioned before, this will help to make our code as future-proof as possible, and encourage browser vendors to work towards providing the standard means of performing actions.

An important uses of object detection is detecting the facilities provided by the browser environment in which the code is executing, so that we can provide features that use those facilities in our code or to determine if we need to provide a fallback.

The following code snippet shows a basic example of detecting the presence of a browser feature using object detection, in order to determine if we should be providing full application functionality, or a reduced-experience fallback.

```
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
}
```

Here, we test that:

- The browser has a document loaded
- The browser provides a means to bind event handlers
- The browser can find elements given a tag name
- The browser can find elements by id

Failing any of these tests causes us to resort to a fallback position. What is done in the fallback is up to the expectations of the consumers of the code, and the requirements placed upon the code. There are a couple of options that can be considered:

- We could perform further object detection to figure out how to provide a reduced experience that still uses some JavaScript.
- We could simply opt to not execute any JavaScript, falling back to the unscripted HTML on the page.
- We could redirect the user to a plainer version of the site. Google does this with GMail, for example.

Because object detection has very little overhead associated with it (it's just a simple property/object lookup) and is relatively simple in its implementation, it makes for a good

candidate to provide basic levels of fallback; both at the API and at the application level. It's a good choice for the first line of defense in your reusable code authoring.

But what if our assumption about an API working correctly just because it exists proves to be overly optimistic?

8.3.3 Feature Simulation

Another means that we have of dealing with regressions, as well as the most effective means of detecting fixes to browser bugs, exists in the form of *feature simulation*.

In contrast to object detection, which is simply an object/property lookup, feature simulation performs a complete run-through of a feature to make sure that it works as we would expect it to.

While object detection is a good way to know if a feature exists, it isn't enough to know if the feature will *behave* as is intended. However, if we know of specific bugs, we can quickly build tests to check at what point in the future the feature bug is fixed, as well as write code to work around the bug until such a time.

As an example, Internet Explorer will erroneously return both elements *and* comments if we execute `getElementsByTagName("*")`. No amount of object detection is going to determine if this will happen or not. Additionally, this bug will, most likely (hopefully) be fixed by the Internet Explorer team in some future release of the browser.

Let's write an example of using feature simulation to determine if the `getElementsByTagName()` method will work as we expect it to:

```
window.findByTagWorksAsExpected = (function(){
  var div = document.createElement("div");
  div.appendChild(document.createComment("test"));
  return div.getElementsByTagName("*").length === 0;
})();
```

In this example, we have written an immediate function that returns `true` if a call to `getElementsByTagName("*")` functions as expected, and `false` otherwise.

The steps of this test function are fairly simple:

- Create a dis-attached `<div>` element.
- Add a comment node to the `<div>`.
- Call the function and see how many values are returned, and return `true` or `false` depending upon the result.

Well, knowing that there's a problem is only half the battle. So what can we do with this knowledge to make things better for our code? Listing 8.3 shows a use of the feature simulation snippet presented above in a useful context: working around the bug.

Listing 8.3 Putting feature simulation into practice to work around a browser bug

```
<!DOCTYPE html>
<html>
  <head>
```

```
    <title>Listing 8.3</title>
    <script type="text/javascript" src="../scripts/assert.js"></script>
    <link href="../styles/assert.css" rel="stylesheet" type="text/css">
  </head>
  <body>

    <div><!-- comment #1--></div>
    <div><!-- comment #2--></div>

    <script type="text/javascript">

      function getAllElements(name) {

        if (!window.findByTagWorksAsExpected) {                //#1
          window.findByTagWorksAsExpected = (function(){       //#2
            var div = document.createElement("div");           //#2
            div.appendChild(document.createComment("test"));   //#2
            return div.getElementsByTagName("*").length === 0; //#2
          })();                                                //#2
        }

        var allElements = document.getElementsByTagName('*');  //#3
        if (!windowfindByTagWorksAsExpected) {                 //#4
          for (var n = allElements.length -1; n >= 0; n--) {   //#4
            if (allElements[n].nodeType === 1)                 //#4
              allElements.splice(n,1);                         //#4
          }                                                    //#4
        }                                                      //#4

        return allElements;

      }

      var elements = getAllElements();                         //#5
      var elementCount = elements.length;                      //#5

      for (var n = 0; n < elementCount; n++) {                 //#6
        assert(elements[n].nodeType === 1,                     //#6
               "Node is an element node");                     //#6
      }                                                        //#6

    </script>

  </body>
</html>
```

#1 Tests if we already know the answer
#2 Determines if the feature works
#3 Calls the suspect feature
#4 Fixes things up if buggy
#5 Sets up for testing
#6 Tests the feature with work-around

In this code we set up some `<div>` elements containing comment nodes that we'll later use for testing. Then we get down to business with some script.

Because using `document.getElementsByTagName('*')` directly is suspect, we define an alternate method, `getAllElements()` to use in its place. We want this method to just

factor down into a call to `document.getElementsByTagName('*')` on browsers that implement it correctly, but to use a fallback that produces the correct results on browsers that do not.

So the first thing that our method does is to use the immediate function that we developed above to determine if the feature works as expected (#2). Note that we store the result in a window-scoped variable so that we can refer to it later, and we check to see if it's already been set so that we only run the (relatively expensive) feature simulation check once (#1).

After the check, we run the call to `document.getElementsByTagName('*')` and store the result in a variable (#3).

At this point, we have the node list of all elements, and we know whether we're operating in a browser that has the "comment node problem" or not. If we had determined that the problem exists, we run through the nodes stripping out any that aren't element nodes (#4). This process is skipped for browsers that don't have the problem.

NOTE

The `nodeType` of element nodes is 1, while that of comment nodes is 8. You can find a complete list of `nodeType` values at: https://developer.mozilla.org/en/nodeType.

Finally, we test our new method by using it (#5) and asserting that the returned node list only contains element nodes (#6).

This example demonstrates how feature simulation works in two phases. To start, a simple test is run to determine if a feature works 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. While that may be a semantic distinction, it's one that is important to keep in mind.

Secondly, the results of the test are later used in our program to speed up looping through an array of elements. As a browser that works correctly (returns only elements) doesn't need to perform the element checks on every stage of the loop, we can completely skip it and not pay any performance penalties in the browsers that work correctly.

That is the most common idiom used in feature simulation: making sure a feature works as expected, and providing fallback code in non-working browsers.

Let's look at another example. In standards-compliant browsers, the name used to retrieve the value of the `style` attribute of DOM elements is (unsurprisingly) `style`. However, Internet Explorer uses `cssText`. Listing 8.4 shows how to use feature simulation to deal with this situation.

Listing 8.4: Figure out the attribute name used to access textual element style information.

```
<div id="test1" style="color:red;">Red</div>
<div id="test2" style="color:blue;">Blue</div>

<script type="text/javascript">

  var STYLE_NAME = (function(){                          //#1
    var div = document.createElement("div");
    div.style.color = "red";
    if (div.getAttribute("style")) return "style";       //#2
    if (div.getAttribute("cssText")) return "cssText";   //#2
  })();

  window.onload = function(){                            //#3
    assert(
      document.getElementById("test1").getAttribute(STYLE_NAME)
        == 'color:red;',
      "Correctly fetched style attribute");
    assert(
      document.getElementById("test2").getAttribute(STYLE_NAME)
        == 'color:blue;',
      "Correctly fetched style attribute");
  };
```

#1 Tests feature
#2 Records result
#3 Tests the test

This example again demonstrates the two parts of feature simulation. We start by executing an immediate function (#1) that tests which name must be used to access the attribute, and record it in a variable for later access (#2). Note that once again, we first check the standards-compliant way, and *then* try the Internet Explorer-specific way.

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 we test (#3) once the page finishes loading.

The most important point to take into consideration when using feature simulation is that it's a trade-off. Paying the extra performance overhead of the initial simulation, along with the extra lines of code added to our programs, gives us the benefit of knowing that a suspect feature will work as expected in all supported browsers, and makes it immune to breaking upon future bug fixes. This immunity can be absolutely priceless when creating reusable code bases.

Feature simulation is great when we can test whether a browser is broken or not, but what about browser problems that stubbornly resist being tested?

8.3.4 Untestable browser issues

Unfortunately there are a number of possible problem areas in JavaScript and the DOM that are either impossible or prohibitively expensive to test. These situations are, fortunately, rather rare, but when we encounter them, it will always pay to investigate the matter further.

Following are some known issues that are impossible to test using any conventional JavaScript interactions.

EVENT HANDLER BINDINGS

One of these infuriating lapses in the browsers is the inability to determine if an event handler has been bound. The browsers do not provide any way of determining 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 we have maintained references to all bound handlers.

EVENT FIRING

Another of these aggravations is in determining if an event will fire. While it's possible to determine if a browser supports a means of binding an event, as we've seen a few times earlier in this chapter, 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 itself 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 forever to execute.

The second situation occurs if a script wishes to use custom 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 or leaves an element's boundaries. These are frequently used as an alternative to the `mouseover` and `mouseout` events as they act slightly more intuitively than the standard 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's hard to use them in reusable code.

CSS PROPERTY EFFECTS

Yet another pain point is determining whether changing certain CSS properties actually affects the presentation. A number of CSS properties only affect the visual representation of the display and nothing else (they don't change surrounding elements or affect other properties on the element). Examples are `color`, `backgroundColor`, and `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.

BROWSER CRASHES

Testing script that causes the browser to crash is another annoyance. Code that causes a browser to crash is especially problematic since, unlike exceptions that 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, as in the following example:

```
new RegExp("[\\w\u0128-\uFFFF*_-]+");
```

The problem with this is that it's not possible to test whether this problem exists using feature simulation as the test itself will always produce a crash 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.

INCONGRUOUS APIS

Back in section 8.3, we saw how jQuery decided to disallow 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 set up an incongruity in which the API works differently from browser to browser. In situations such as this, where a bug is so bad that it causes an API to break, the only option is to work around the affected area and provide a different solution.

In addition to impossible-to-test problems, there are issues that are not *impossible* to test, but are prohibitively difficult to test effectively. Let's look at some of them.

API PERFORMANCE

Sometimes specific APIs are faster or slower in different browsers. When writing reusable and robust code, it's important to try and use the APIs that provide good performance. But it's not always obvious which API that would be!

Effectively conducting performance analysis of a feature usually entails throwing a large amount of data at it, and is something that usually takes a relatively long time. Therefore, it's not something we can just do in our code in the same way that we used feature simulation.

AJAX ISSUES

Determining if Ajax requests work correctly is another thorn in our sides. As was 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 optimum.

And not only that, but an extra file would have to be included with the library whose sole reason for being is to serve as a target for these extra requests. The overhead of both these matters is too prohibitive and would certainly not be worth the extra time and resources.

While untestable features are a significant hassle that hinders our goal of writing reusable JavaScript, they are frequently able to be worked around with a bit of effort and cleverness. By utilizing alternative techniques, or constructing our APIs in a manner as to obviate these issues in the first place, it will most likely be possible that we'll be able to build effective code, despite the odds stacked against us.

8.4 Assumptions

Writing cross-browser, reusable code is a battle of assumptions. But by using clever detection and authoring, we reducing the number of assumptions that we make in our code.

When we make assumptions about the code that we write, we stand to encounter problems later down the road.

For example, if you assume that an issue or a bug will always exist in a specific browser, that's a huge and dangerous assumption. Instead, testing for the problem (as we've done throughout this chapter) proves to be much more effective. In our coding, we should always be striving to reduce the number of assumptions that we make, effectively reducing the room that we have for error, and the probability that something's going to come back and bite us in the butt.

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. In other words: browser detection.

Unfortunately, most user agent string analysis proves to be a superb source of future-induced errors. Assuming that a bug, issue or proprietary feature will always be linked to a specific browser is a recipe for disaster.

However, reality intervenes when it comes to minimizing assumptions: it's virtually impossible to remove all of them. At some point we'll have to assume that a browser will do what it's supposed to do. Figuring out the best point at which that balance can be struck is completely up to the developer, and is what, as they say, "Separates the men from the boys" (with apologies to our female readers).

For example, let's re-examine the event attaching code that we've already seen a number of times:

```
function bindEvent(element, type, handle) {
  if (element.addEventListener) {
    element.addEventListener(type, handle, false);
  }
  else if (element.attachEvent) {
    element.attachEvent("on" + type, handle);
  }
}
```

Before peeking below, see if you can spot three assumptions that are made by this code. Go on, we'll wait.

(Jeopardy theme plays.)

How'd you do? In the above code, we made at least the following three assumptions:

1. That the properties that we're checking are, in fact, callable functions.
2. That they're the correct functions, performing the action that we expect.
3. That these two methods are the only possible ways of binding an event.

We could easily get rid of the first assumption by adding checks to see if the properties are, in fact, functions. Tackling the remaining two points is much more difficult.

In our code, we always need to decide how many assumptions are optimal for our requirements, our target audience, and for us. Frequently, when reducing the number of assumptions we also increase the size and complexity of the code base. It's fully possible,

and rather easy, to attempt to reduce assumptions to the point of complete insanity, but at some point we'll have to stop and take stock of what we have, say “good enough”, and work from there. Remember that even the least assuming code is still prone to regressions introduced by a browser.

8.5 Summary

Reusable cross-browser development is a juggling act between three points:

- Code size: keeping the file size small.
- Performance overhead: keeping the performance level to a palatable minimum.
- API quality: making sure that the APIs provided work uniformly across browsers.

There is no magic formula for determining the correct balance of these points. 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.

9 Ninja alchemy: run-time code evaluation

What's in this chapter:

- How run-time code evaluation works
- Different techniques for evaluating code
- Using evaluation in applications

One of the many powerful abilities that separates JavaScript from many other languages is its ability to dynamically interpret and execute pieces of code at runtime.

Code evaluation is simultaneously a most powerful, as well as a 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 us a marked advantage when creating advanced application code.

In this chapter we'll explore that various ways of interpreting code at run-time and the situations in which this powerful ability can lift our code into the big leagues.

9.1 Code evaluation mechanisms

Within JavaScript there are a number of different mechanisms for evaluating code. Each has its own advantages and disadvantages and which to use should be chosen carefully based upon the context in which it is being employed.

Let's take a look at the various ways that we can cause code to be evaluated.

9.1.1 Evaluation with the eval() method

The `eval()` method is likely the most commonly used means of evaluating code at run time. Defined as a function in global scope, the `eval()` method executes the code passed into it in string form, within the current context. The result returned from the method is the result of the last evaluated expression.

BASIC FUNCTIONALITY

Let's see the basic functionality of `eval()` in action. We expect two basic things from `eval()`:

- It will evaluate the code passed to it as a string
- It will execute that code in the scope within which `eval()` is called

Take a look at the code of listing 9.1.

Listing 9.1: Basic test of the eval() method

```
<script type="text/javascript">

  assert(eval("5 + 5") === 10,                                  //#1
         "5 and 5 is 10");

  assert(eval("var ninja = 5;") === undefined,                  //#2
         "no value was returned" );
  assert(ninja === 5, "The variable ninja was created");        //#3

  (function(){
    eval("var ninja = 6;");                                     //#4
    assert(ninja === 6,                                         //#4
           "evaluated within the current scope.");              //#4
  })();

  assert(ninja === 5,                                           //#5
        "the global scope was unaffected");                     //#5

</script>
```

#1 Tests simple expression
#2 Tests valueless evaluation
#3 Verifies side effect
#4 Tests evaluation scope
#5 Tests for scope "leakage"

In the code of this listing, we test a number of basic assumptions about `eval()`. The result of these test are shown in figure 9.1.

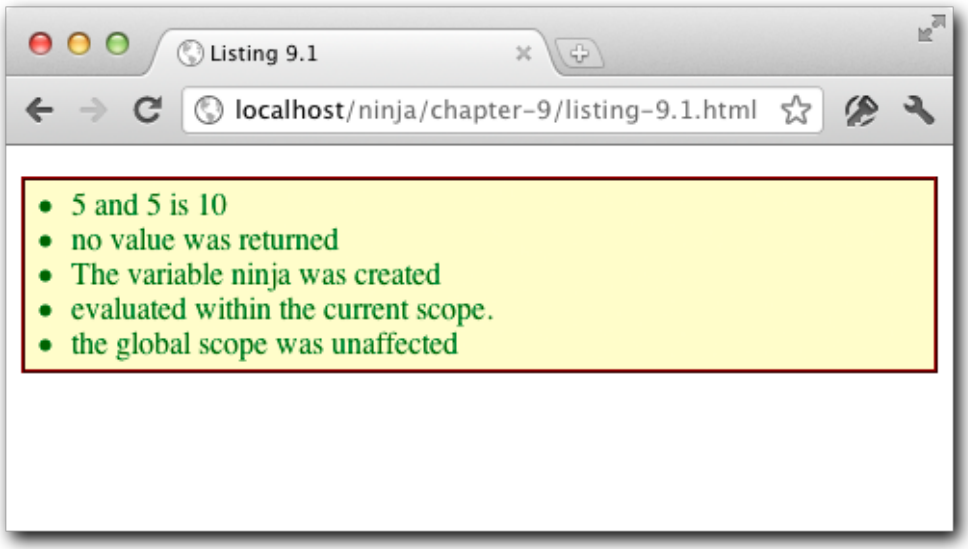

Figure 9.1 Proving that `eval()` can evaluated various expressions and is confined to the local scope

First, we send a string containing a simple expression into the `eval()` method (#1) and verify that it produces the expected result.

Then, we try a statement that produces no value: the assignment "ninja = 5", and verify that the expected value (none) is returned (#2). But wait, that's not enough of a test. We expected no result, but was that because the expression was evaluated and produced no result, or because nothing happened at all? A further test is needed.

We expect the code to be evaluated in the current scope, in this case the global scope, and so we'd expect a side effect of the evaluation to be the creation of a globally-scoped variable named `ninja`. And indeed, a simple test (#3) bears that out.

Next we want to test that an evaluation in a non-global scope works as expected. So we create an immediate function, and evaluate the phrase "var ninja = 6;" within it (#4). A test that the variable exists with the expected value is conducted. But once again, that's not quite enough. Is `ninja` evaluating to 6 because we created a new variable in the local scope? Or did we modify the global `ninja` variable?

One further test (#5) proves that the global scope was untouched.

EVALUATION RESULTS

The eval() method will return the result of the last expression in the passed code string. For example, if we were to call:

```
eval('3+4;5+6')
```

the result would be 11.

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, if we wanted to create a simple object using eval(), we might be tempted to write:

```
var o = eval('{ninja: 1}');
```

But that wouldn't do what we want. Rather, we'd need to surround the object literal with parentheses as follows:

```
var o = eval('({ninja: 1})');
```

Let's run some more tests as shown in listing 9.2.

Listing 9.2: Testing returned values from eval()

```
<script type="text/javascript">

  var o = eval("({name:'Ninja'})");                  //#1
  assert(o != undefined,"o was created");
  assert(o.name === "Ninja",
         "and with the expected property");

  var fn = eval("(function(){return 'Ninja';})");    //#2
  assert(typeof fn === 'function',
         "function created");
  assert(fn() === "Ninja",
         "and returns expected value" );

</script>
```

#1 Creates an object
#2 Creates a function

Here we create a object (#1) and a function (#2) on the fly using `eval()`. Not how in either case, the phrases needed to be enclosed in parentheses.

As an exercise, make a copy of `listing-9.2.html`, remove the parentheses, and load the file. See how far you get!

If you ran this test under Internet Explorer, you may have gotten a nasty surprise. IE has a problem executing that particular syntax. We are forced to use some boolean-expression trickery to get the call to eval() to execute correctly. The following shows a technique found in jQuery to create a function using `eval()` in broken version of IE.

```
var fn = eval("false||function(){return true;}");
assert(fn() === true,
       "The function was created correctly.");
```

Now you might be wondering why we would even want to create a function in this manner in the first place. Well, we usually wouldn't. If we know what function we want to create, we'd usually just create it using one of the mans we explored in chapter 3. But what if we didn't know in advance what the syntax of the function might be We might want to generate the code at run-time, or perhaps obtain the code from someone else. (If that that latter possibility sets off your alarms, fear not, we'll explore security considerations in just a bit.)

Just as when we create a function in a particular scope using “normal” means, functions create with `eval()` inherit the closure of that scope – a ramification of the fact that `eval()` executes within the local scope.

Of course, if we don't need that additional closure, there's another alternative that we can make use of.

9.1.2 Evaluation via the Function constructor

All functions in JavaScript are an instance of `Function`.; we learned that back in chapter 3. There we saw how we could create named functions using syntax such as `function name(...){...}`, or create anonymous function via function literals.

However, we can also instantiate functions directly using the Function constructor., as shown in the following code:

```
var add = new Function("a", "b", "return a + b;");
assert(add(3,4) === 7, "Function created and working!");
```

The last argument to the Function constructor is always the code that will become the body of the function. Any preceding arguments represent the names of the arguments that are to be passed into the function.

So our example is equivalent to:

```
var add = function(a,b) { return a + b; }
```

The main difference being, of course, that in the former example, the function body is provided by a run-time string.

Another difference is that it's important to realize that no closures are created when functions are created via the Function constrictor. This can be a good thing when we don't want to incur any of the overhead associated with possible variable lookups.

9.1.3 Evaluation with timers

Another way that we can cause strings of code to be evaluated, and in this case asynchronously, is through the user of timers.

Normally, as we saw in chaper 6, we’d pass an inline function or a function reference to a timer. This is the recommended usaged of the `setTimeout()` and `setInterval()` methods, but these methods also accept strings which will be evaluated when the timers fire.

It's very rare that we would need to use this behavior (it's roughly equivalent to using `new Function`) and it's essentially unused at this point except in the cases where the code to be evaluated must be a run-time string.

9.1.4 Evaluation in the global scope

We stressed, when discussing the `eval()` method, that the evaluation executes in the scope within which `eval()` is called. And we proved it with the test of listing 9.1. But frequently, we may want to evaluate strings of code in the global scope despite the fact that that may not be the current execution scope.

For example, within some function we may want to execute code in the global scope, as in:

```
(function(){
```

```
  eval("var test = 5;");
})();

assert(test === 5,
       "Variable created in global scope");
```

If we expected the variable `test` to be created in the global scope, our test results are discouraging as the test fails. Because the execution scope of the evaluation is within the immediate function, so is the variable scope.

A naïve answer would be to change the code to be evaluated to:

```
eval("window.test = 5;");
```

While this will cause the variable to be defined in the global scope, it does *not* change the scope in which the evaluation takes place and any other expectations we have about scope will still be local rather than global.

However, there is a tactic that we can use in modern browsers to achieve our goal: injecting a dynamic `<script>` tag into the document with the script contents that we want to execute.

Andrea Giammarchi (self-professed JavaScript Jedi) developed a technique for making this work properly across multiple platforms. His original work can be found at:

http://webreflection.blogspot.com/2007/08/global-scope-evaluation-and-dom.html

and an adaptation can be found in Listing 9.3.

Listing 9.3: Evaluating code in the global scope

```
<script type="text/javascript">

  function globalEval(data) {                                  //#1
    data = data.replace(/^\s*|\s*$/g, "");
    if (data) {
      var head = document.getElementsByTagName("head")[0] ||
                 document.documentElement,
          script = document.createElement("script");           //#2

      script.type = "text/javascript";
      script.text = data;

      head.appendChild(script);                                //#3
      head.removeChild(script);                                //#4
    }
  }

  window.onload = function() {
    (function() {
      globalEval("var test = 5;");
    })();

    assert(test === 5, "The code was evaluated globally.");
  };
```

```
</script>
```

#1 Defines a global eval function
#2 Creates a script node
#3 Attaches it to the DOM
#4 Blows it away

The code for this is surprisingly simple. To use in place of `eval()`, we define a function named `globalEval()` (#1) that we can call whenever we want an evaluation to take place in the global scope.

This function strips any leading and trailing whitespace from the passed string (review chapter 7 if the statement doesn't make sense to you), and then we locate the `<head>` element of the DOM, and create a dis-attached `<script>` element (#2).

We then set the type of the script element, and load its body with the passed string to be evaluated.

Attaching the script element to the DOM as a child of the head element (#3) causes the script to be evaluated in the global scope. Then, having done its duty, the script element is unceremoniously discarded (#4). The results of the test are shown in figure 9.2.

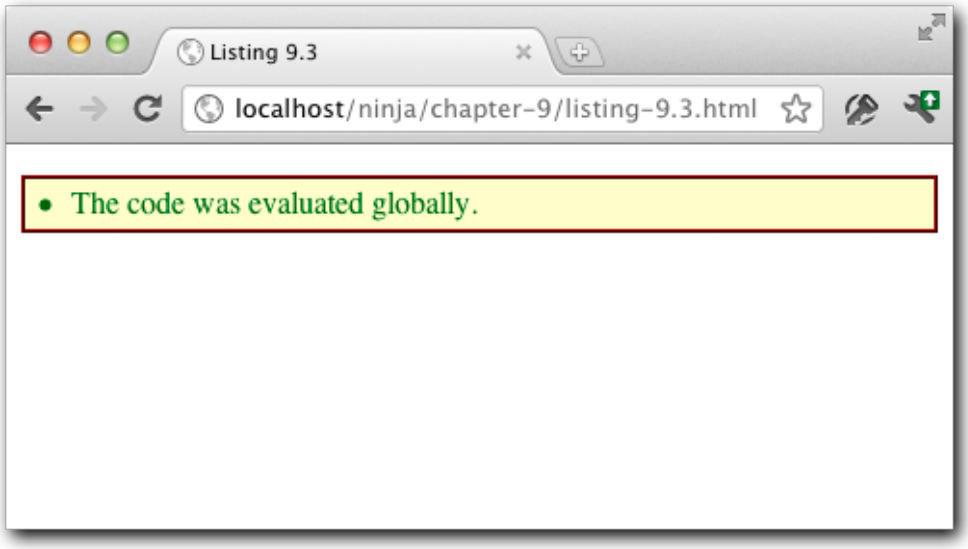

Figure 9.2 We can execute evaluated code in the global context using a bit of DOM manipulation trickery

A common use case for this code is 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 use of our new function a necessity.

But, can we trust that server?

9.1.5 Safe Code Evaluation

One question that frequently arises, with respect to code evaluation, concerns the safe execution of JavaScript code. In other words: is it possible to safely execute untrusted JavaScript on our pages without compromising the integrity of the site? After all, if we didn't provide the code to be evaluated, goodness knows what it could contain!

Some naïve user might supply a string of code that puts execution into an infinite loop, or removes necessary DOM elements, or tromps all over vital data. Or, even worse, malicious users could purposefully try to inject code that compromises the security of the site.

So generally the answer to the question is: no. There are simply too many ways that arbitrary code can skirt around any barriers put forth, and can result in code getting access to information that it's not supposed to or causing other problems.

There is hope, however. A Google project named Caja attempts to create a translator for JavaScript that converts JavaScript into a safer form, and immune to malicious attacks. You can find more information on Caja at http://code.google.com/p/google-caja/

As an example, look at the following code

```
var test = true;
(function(){ var foo = 5; })();
Function.prototype.toString = function(){};
```

Caja will cajole that code into:

```
___.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.

The desire for securely executing random JavaScript code frequently stems from wanting to create mashups and safe advertisement embedding without worrying about security becoming compromised. We're certainly going to see a lot of work in this realm and Google Caja is leading the way.

OK, so now we know a number of ways to take a string and get it converted to code that is immediately evaluated. What about the other direction?

9.2 Function decompilation

Most JavaScript implementations also provide a means to decompe already-compiled JavaScript code.

As complicated as this sounds, this is simply provided by the `.toString()` method of functions. Let's test this with code of listing 9.4.

Listing 9.4: Decompiling a function into a string

```
<script type="text/javascript">

  function test(a){ return a + a; }                    //#1

  assert(test.toString() ===                           //#2
         "function test(a){ return a + a; }",          //#2
         "Function decompiled");                       //#2

</script>
```

#1 Defines a function
#2 Tests decompilation

In this test, we create a simple function named `test` (#1), and then assert that the function's `toString()` method returns the original text of the function (#2).

One thing to be aware of: the value returned by `toString()` will contain all the original whitespace of the original declaration, to include line terminators. For testing purposes, we punted in listing 9.4, defining a simple function on a single line. If you make a copy of the file and fool around with the formatting of the function declaration, you will find that the test fails until you change the test string to match the formatting of the declaration.

The act of decompilation has a number of potential uses, especially in the area of macros and code rewriting. One of the more interesting uses is one presented in the Prototype JavaScript library. In the Prototype library, the code decompiles 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.

Listing 9. 5 shows a simplification of the code in Prototype to infer function parameter names.

Listing 9.5: A function for finding the argument names of a function

```
<script type="text/javascript">

  function argumentNames(fn) {
    var found = /^[\s\(]*function[^(]*\(\s*([^)]*?)\s*\)/   //#1
                .exec(fn.toString());                       //#1
    return found && found[1] ?                              //#2
             found[1].split(/,\s*/) :                       //#2
             [];                                            //#2
  }

  assert(argumentNames(function(){}).length === 0,          //#3
         "Works on zero-arg functions.");
```

```
  assert(argumentNames(function(x){})[0] === "x",          //#4
         "Single argument working.");

  var results = argumentNames(function(a,b,c,d,e){});     //#5
  assert(results[0] == 'a' &&
         results[1] == 'b' &&
         results[2] == 'c' &&
         results[3] == 'd' &&
         results[4] == 'e',
         "Multiple arguments working!");

</script>
```

#1 Finds argument list
#2 Splits the list
#3 Tests zero-arg case
#4 Tests single-arg case
#5 Tests multi-arg case

The function comprises just a few lines of code, but uses a lot of advanced JavaScript features in those scant few characters. First, the function decompiles the function and uses a regular expression to extract the comma-delimited argument list (#1). (We covered regular expressions in chapter 6, if you need a refreshed.)

Note that because the `exec()` method *expects* a string, we could have left the `.toString()` off the function argument, and it would have been implied. But we included it here explicitly for clarity.

Then the result of that extraction is split into its component values, performing check to make sure that cases such as zero-argument lists are accounted for (#2).

Finally, we test that zero-argument (#3), single-argument (#4) and multi-argument (#5) cases work as expected, as shown in figure 9.3.

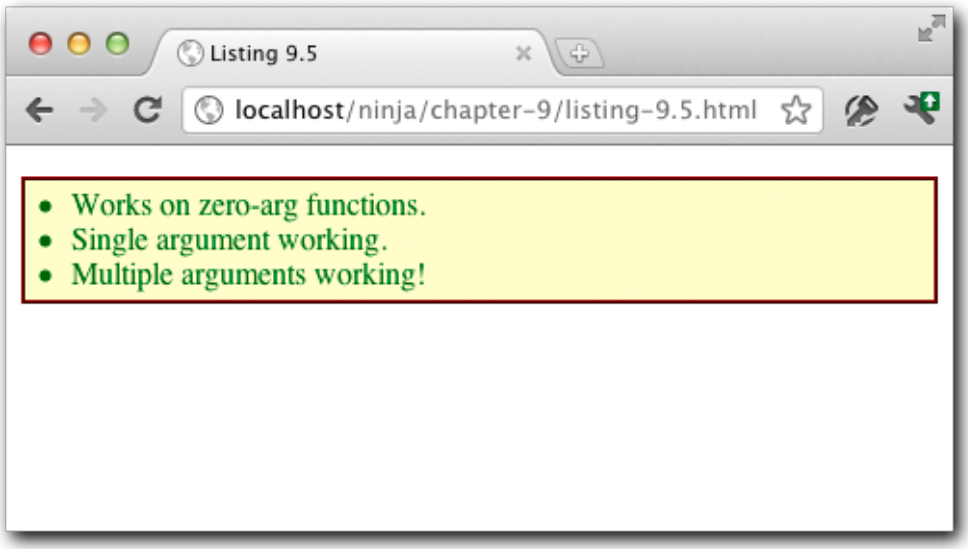

Figure 9.3 We can use function decompilation to do fancy things such as inferring the names of arguments to a function

There's an important point to take into consideration when working with functions in this manner: it's possible that a browser may not support decompilation. While there aren't many that don't, one such browser is Opera Mini. If that's on your list of supported browsers, you'll need to take that into consideration in code that uses function decompilation.

As emphasized previously in this book (particularly in chapter 8), we certainly don't want to resort to browser detection to determine if function decompilation is supported. Rather, we'll use feature simulation (see chapter 8) to test whether a browser supports decompilation. One means could be:

```
var FUNCTION_DECOMPLIATION = /abc(.|\n)*xyz/.test(function(abc){xyz;});
assert(FUNCTION_DECOMPLIATION,
      "Function decompilation works in this browser");
```

Again using regular expressions (which are a sadly underused workhorse in JavaScript), we pass a function to the test() method (here letting the invocation of `toString()` happened implicitly as the method expects a string) and store the result in a variable for later use (or testing as shown here).

At this point, we've covered the various means of performing run-time code evalution; now let's put that knowledge into action.

9.3 Code evaluation in action

We saw in section 9.1 that there are a number of ways in which code evaluation can be performed. We can use this ability for both interesting and practical purposes throughout our code. Let's examine some examples of evaluation in order to give us a better understanding of when and where we can or should use it in our code.

9.3.1 Converting JSON

Probably the most widespread use of run-time evaluation comes in the form of converting JSON strings into their JavaScript object representations. As JSON data is simply a subset of the JavaScript language, it is perfectly capable of being evaluated as JavaScript code.

As frequently happens in the best laid of plans, there *is* one minor gotcha that we have to take into consideration. We need to wrap our object input in parentheses in order for it to evaluate correctly. that's a quite simple to do, as seen in Listing 9.6; we just need to remember to do it..

Listing 9.6: Converting a JSON string into a JavaScript object

```
<script type="text/javascript">

  var json = '{"name":"Ninja"}';                //#1

  var object = eval("(" + json + ")");          //#2

  assert(object.name === "Ninja",               //#3
         "My name is Ninja!");                  //#3

</script>
```

#1 Defines source JSON
#2 Converts JSON
#3 Tests the conversion

Pretty simple stuff – and it performs well in most JavaScript engines.

However, there is a major caveat to using `eval()` for JSON parsing: often, JSON data is coming from a remote server and blindly executing untrusted code from a remote server is rarely a good thing.

The most popular JSON converter script is written by Douglas Crockford, the original creator 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:

- https://github.com/douglascrockford/JSON-js

Some important pre-processing that Mr. Crockford's function performs prior to the actual evaluation include:

- Guarding against certain Unicode characters that can cause problems in some browsers.

- Guards against non-JSON patterns that could indicate malicious intent; including the assignment operator and usage of the new operator.
- Makes sure that only JSON-legal characters are included.

If the JSON that is to be evaluated comes from your own code and servers, or some other trusted source, we usually don't need to worry about malicious code injection. But when we have no reason to trust the JSON that we're going to evaluate, using safeguards such as those provided by Mr. Crockford is just prudent.

Now let's look at another common usage of run-time evaluation.

9.3.2 Importing namespaced code

In chapters 3 and 8, we talked about name-spacing code to keep from polluting the current context – usually the global context. And that's a good thing. But what about when we want to take code that's been namespaced and bring it into the current context deliberately?

This can be a challenging problem, considering that there is no simple or supported way to do it in the JavaScript language. Most of the time, we have to resort to actions 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, we can make use of run-time evaluation to make the above easier to implement.

Whenever a new class or module is added to a base2 package, a string of executable code is constructed which can be evaluated to introduce the functions into the current context, as shown in Listing 9.14.

Listing 9.14: Examining how the base2 namespace importing works

```
<script type="text/javascript">

  base2.namespace ==                                          //#1
    "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",                          //#2
         "The DOM object doesn't exist." );

  eval(base2.namespace);                                      //#3
```

```
        assert(typeof DOM === "object",                                  //#4
               "And now the namespace is imported." );
        assert(typeof Collection === "object",
               "Verifying the namespace import." );

    </script>
```

#1 Defines imported names
#2 Tests "before" condition
#3 Evaluates importees
#4 Tests "after" conditions

This is a very ingenious way of tackling a complex problem. Albeit, it may not necessarily be done in the most graceful manner, but until implementations of future versions of JavaScript exist that support this, we'll have to make do with what we have.

And speaking of ingenious, another usage of evaluation is the packing of JavaScript code. Let's learn about that.

9.3.3 JavaScript compression and obfuscation

One of the realities of client-side code is that it needs to somehow actually *get* to the client side. As such, keeping the transmission footprint as small as possible is a worthy goal. We could try to write our code in as few characters as possible, but that leads to crappy and unreadable code. Rather, it's best to write our code with as much clarity as possible, and then to compress it for transmission.

A popular piece of JavaScript software that does just that is Dean Edwards' Packer. This clever script compresses JavaScript code, providing a resulting JavaScript file that is significantly smaller that the original, while still being capable of executing and self-extracting itself to run again. Dean Edwards' Packer can be found at:

- http://dean.edwards.name/packer/

The result of using this tool is an encoded string which is converted into a string of JavaScript code and executed using the `eval()` function. The result typically looks something like this:

```
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 clever and quite interesting, it has some fundamental flaws, the most debilitating being that the overhead of uncompressing the script every time it loads is quite costly.

When distributing a piece of JavaScript code, it's normal to think that the smallest code (byte-wise) will download and load the fastest. But this is not always true – smaller code may download faster, but doesn't always *evaluate* faster. And when all is said and done, it's

the combination of downloading *and* evaluation that's important to the performance of our pages. It breaks down to a simple formula:

```
load time = download time + evaluation time
```

Let's take a look at the speed of loading jQuery in three forms:

- normal (uncompressed)
- minimized, using Yahoo!'s YUI Compressor which removes whitespace and performs a few other other simple tricks
- packed using Dean Edwards' Packer, with massive rewriting and uncompression using `eval()`.

By order of file size, packed is the smallest, then minimized, followed by uncompressed, and we'd rightly expect their download times to be proportional to the file size. However, the packed version has significant overhead: it must be uncompressed and evaluated on the client-side. This unpacking has a tangible cost in load time. and means, in the end, that using a minified version of the code is much faster than the packed one, even though its file size is quite larger.

The results of the study (more information on which can be found at http://ejohn.org/blog/library-loading-speed/) is shown in table 9.1.

Table 9.1: A comparison of load speeds for various formats of the jQuery JavaScript library

Compression scheme	Average time	# of samples
minimized	519.7214	12,611
Packed	591.6636	12,606
Normal	645.4818	12,589

This isn't to say that using code from Packer is worthless – far from it. But if you're goals are limited to performance, it may not be your best choice.

However, even with the additional overhead, the Packer can be a valuable tool if *obfuscation* is one of your goals. Unlike server-side code, which in a reasonably secured web application is completely inaccessible from the client, JavaScript code must be sent to the client for execution. After all, the browser can't execute any code that it doesn't receive.

Back when the most complicated script on web pages were for trivial activates such as image roll-overs, no one much cared that the code was shipped off the client and available for viewing by anyone on the receiving end.

But these days, in the era of highly functional Ajax pages and so-called Single Page Applications, the amount and complexity of code can be high and some organization can be leery of exposing that code to the public.

The obfuscation provided by scripts such a Packer may be just the answer such organizations are looking for. If nothing else, Packer serves as a good example of using `eval()` to effect run-time evaluation.

TIP

If you are interested in the YUI Compressor, visit its site at: http://developer.yahoo.com/yui/compressor/. Yahoo! also provides some other interesting performance information at: http://developer.yahoo.com/performance/rules.html.

9.3.4 Dynamic code rewriting

Because we have the ability to decompile existing JavaScript functions using a function's `.toString()` method as described in section 9.2, that means that we can create new functions from existing functions by extracting and massaging the old function's contents.

One case where this has been done is in the unit testing library Screw.Unit. (Information can be found at http://github.com/nkallen/screw-unit/tree/master.)

Screw.Unit takes existing test functions and dynamically re-writes their contents to use the functions provided by the library. For example, a typical Screw.Unit test looks like the following:

```
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);
  });
});
```

Note the methods: `describe()`, `it()` and `expect()`, none of which exist in the global scope. To make this code possible, Screw.Unit *rewrites* this code on the fly to wrap all the functions with multiple `with(){}` statements (which we'll talk about in chapter 12), injecting the function internals with the functions that it needs in order to execute, as seen in:

```
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 for the test writers without having to introduce a bunch of variables into the global scope.

Next, the term AOP has been bandied about lately in the world of server-side code. Why should they have all the fun?

9.3.5 Aspect-Oriented script tags

AOP, or Aspect-Oriented Programming is defined by Wikipedia as:

> A programming paradigm which aims to increase modularity by allowing the separation of cross-cutting concerns.

Yeah, that made our heads hurt too.

Stripped down to bare bones AOP is a technique by which code is injected and executed at run-time to handle “cross-cutting” things like logging. Rather than weighing down code with a bunch of logging statements, an AOP engine will add the logging code at run-time keeping it out of the programmer’s face during development.

> **TIP**
>
> For more information on AOP, visit the Wikipedia article at http://en.wikipedia.org/wiki/Aspect-oriented_programming, or the tutorial at http://tim.oreilly.com/pub/a/onjava/2004/01/14/aop.html. And if you’re interested in using AOP in Java, take a gander at *AspectJ in Action* at http://www.manning.com/laddad2/.

Well, the injection and evaluation of code at run-time sounds right up our alley in this chapter, doesn’t it? Let’s see how we might use the ideas of AOP to our advantage.

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, for whatever reason, we’ll create a new script type called "onload".

What? A new script *type*? How can we do that?

As it turns out, defining custom script types is easy because the browsers will just ignore any script type that it does not understand. So we can force the browser to completely ignore a script block (and use it for whatever nefarious purposes we want) by using a type value that’s not standard.

So, if we want to create new type called “onload”, we could do so easily by specifying a script block as follows:

```
<script type=″x/onload″>   … custom script here … </script>
```

(following the convention of using “x” to mean “custom”).

We’ll intend such blocks to contain normal JavaScript code that will be executed whenever the page is loaded, as opposed to being normally executed inline.

Examine the code of listing 9.8.

Listing 9.18: Creating a script tag type that executes only after the page has already loaded.

```
<script>
  window.onload = function(){
    var scripts = document.getElementsByTagName("script"); //#1
    for (var i = 0; i < scripts.length; i++) {             //#2
```

```
      if (scripts[i].type == "x/onload") {                    //#2
        globalEval(scripts[i].innerHTML);                     //#2
      }
    }
  };
</script>

<script type="x/onload">
  assert(true,"Executed on page load");                       //#3
</script>
```

#1 Finds all script blocks
#2 Locates and executes "onload" blocks
#3 Provides custom script

In this example, we provide a custom script block (#3) that is ignored by the browser. In the onload handler for the page, we search for all script blocks (#1) and upon finding any that are of our customer type, we use the `globalEval()` function that we developed earlier in this chapter to cause the script to be evaluated in the global context (#2).

This is obviously a simple example, but this technique could be used for more complex and meaningful purposes. For example, customer script blocks are used with the jQuery `.tmpl()` method to provide run-time templates. This technique could be used for such varying purposes as: executing scripts on user interaction, or when the DOM is ready to be manipulated, or even relatively based upon adjacent elements.

The application of this technique is limited on my the imagination of the page author.

Nw let's see another advanced used of run-time evaluation.

9.3.6 Meta-languages

One of the most poignant examples of the power of run-time code evaluation can be seen in the implementation of other programming languages on top of the JavaScript language; *meta-languages*, if you will, that can be dynamically converted into JavaScript source and evaluated.

There have been two such meta-languages that have been especially interesting.

PROCESSING.JS

Processing.js is a port of the Processing Visualization Language (see http://processing.org/), which is typically implemented using Java, to JavaScript and running on the HTML 5 Canvas element by John Resig.

This port is a full programming language that can be used to manipulate the visual display of a drawing area. Arguably Processing.js is particularly well suited to this task, making it an effective port.

An example of Processing.js code, utilizing a script block with a type of "application/processing", follows:

```
<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,js code is converted into JavaScript code and executed using a call to `eval()`. The resulting JavaScript is the following:

```
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;
  }
}}
```

The details of the translation from a meta-language to JavaScript would require a chapter of its own (or maybe even a whole book), and is beyond the scope of this discussion.

But why use a meta-language at all?

By using the Processing.js language, we gain a few immediate benefits over using just JavaScript:

- We get the benefits of Processing advanced language features (such as classes and inheritance)
- We get Processing's simple but powerful drawing API
- We get all of the existing documentation and demos on Processing

More information can be found at http://ejohn.org/blog/processingjs/.

The important point to take away from all this is that all of this advance processing is made possible through the code evaluation capabilities of the JavaScript language.

Let's look at another such project.

OBJECTIVE-J

A second major project using these capabilities is Objective-J, a port of the Objective-C programming language to JavaScript by the company 280 North, and used for the product 280 Slides (an online slideshow builder). See http://280slides.com/.

The 280 North team had extensive experience developing applications for OS X, which are primarily written in Objective-C, so in order.to create a more-productive environment for themselves to work within, they ported the Objective-C language to JavaScript. In addition to providing a thin layer over the JavaScript language, Objective-J allows JavaScript code to be mixed in with the Objective-C code. An example is shown here:

```
// 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 the Objective-J 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 using run-time evaluation.

While this implementation has less far-reaching benefits (it's a specific hybrid language that can only be used within this context), its potential benefits to users who are already familiar with Objective-C, but wish to explore web programming, will be self-evident.

9.4 Summary

In this chapter we've learned the fundamentals of run-time code evaluation in JavaScript.

There are a number of mechanisms that JavaScript provides for evaluating strings of JavaScript code at run time:

- The `eval()` method
- Function constructors
- Timers
- Dynamic `<script>` blocks

JavaScript also provides a means to go in the opposite direction, that is, obtaining a string for the code of a function, via a function's `toString()` method – a process known as *function decompilation*.

We also explored a variety of use cases for run-time evaluation including such activities as: JSON conversion, moving definitions between namespaces, minimization and obfuscation of JavaScript code, dynamic code rewriting and injection, and even the creation of meta-languages.

While potential for misuse of this powerful feature is possible, the incredible power that comes with harnessing code evaluation gives you an excellent tool to wield in our quest for JavaScript ninja-hood.

10
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 10.1.

Listing 10.1: 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 10.2: 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 10.3.

Listing 10.3: 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 10.3 result in some rather dramatic performance differences, as seen in Table 10.1.

Table 10.1: All time in ms, for 1000 iterations, in a copy of Firefox 3.

	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.

10.1 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 10.4.

Listing 10.4: 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 10.5 is from the base2 JavaScript library.

Listing 10.5: 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 10.5 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 10.6.

Listing 10.6: 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 10.7, is from the Firebug developer extension for Firefox.

Listing 10.7: 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 10.7, 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.

10.2 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 10.8, both performing the same operation using the Yahoo UI JavaScript library, but also gaining extra simplicity and clarity by using `with(){}`.

Listing 10.8: 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

10.3 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 10.9.

Listing 10.9: 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 10.9:

- 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.

10.4 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 10.10.

Listing 10.10: 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.

10.5 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 10.11.

Listing 10.11: 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 10.12.

Listing 10.12: 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 10.12 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.

10.6 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.

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.

11.1 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 11.1 has a couple examples of how it could be used.

Listing 11.1: 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 11.2.

Listing 11.2: 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 11.3.

Listing 11.3: 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 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).

11.2 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 11.4 and completely ignore everything else about building a CSS selector engine.

Listing 11.4: 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 11.1.

Table 11.1: Map of CSS selectors to their associated XPath expressions.

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.

11.3 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 (suchas 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 11.5, into consideration, as if we were trying to implement our "div span" selector.

Listing 11.5: 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.

11.3.1 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 11.6 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 11.6: 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.

11.3.2 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

11.3.3 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.

11.3.4 Recursing and Merging

As was shown in Listing 11.1, 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 11.7.

Listing 11.7: 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.

11.3.5 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 11.8.

Listing 11.8: 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 11.8 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.

11.4 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.

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.

12.1 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 of 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.

12.1.1 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 12.1: 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 12.2: 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>")[0].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 remedied 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.

12.1.2 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 12.3: Inserting a DOM fragment into multiple locations in the DOM, using the code from Listing 12.1 and Listing 12.4.

```
<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 12.4: 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.

12.1.3 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 12.5: 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 12.6: 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 12.7: 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).

12.2 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 12.8: 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 12.9: 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).

12.3 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 things 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 12.10: 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 12.11: 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.

12.4 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 12.12: 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 12.10.

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 12.13: 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 12.14: 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.

12.5 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.

13
Attributes and CSS

In this chapter:

- The difference between DOM Attributes and DOM properties.
- Dealing with cross-browser attributes and styles.
- Techniques for handling element dimensions.

DOM attributes and properties are an important piece of creating a solid piece of reusable JavaScript code. They act as the primary means through which additional information can be attached to a DOM element. There likely isn't an area in the DOM that is more riddled with bug and cross-browser issues than this.

The same goes for CSS and styling of elements. Many of the complications that arise when constructing a dynamic web application come from setting and getting element styling properly. While this book won't cover all that is known about handling element styling (that's enough for an entire other book) the core essentials will be considered.

13.1 DOM Attributes and Expandos

When accessing the values of element attributes you have two possible options: Using the traditional DOM methods (such as `getAttribute` and `setAttribute`) or using properties of the DOM objects themselves (frequently called an 'expando' property). For example, you can see the different styles in Listing 13.1.

Listing 13.1: An example of using traditional DOM attribute setting and an expando.

```
<div></div>
<script>
window.onload = function(){
  var div = document.getElementsByTagName("div")[0];
```

```
  // Set the ID
  div.setAttribute("id", "somediv");
  div.id = "somediv";
};
</script>
```

Besides the obvious simplicity that an expando gives you (less typing is always a plus) there are five major differences between the two that need to be understood:

First, the naming of expandos are generally more consistent across browsers. If you're able to access an expando in one browser you have a good chance of it having the the same in other browsers, as well.

Listing 13.2 shows a case where there's a naming discrepancy when using normal DOM attributes.

Listing 13.2: A naming discrepancy that occurs between browsers when using traditional DOM attribute methods.

```
<div class="oldclass"></div>
<script>
window.onload = function(){
  var div = document.getElementsByTagName("div")[0];

  // Set the class
  div.setAttribute("class", "newclass");

  // The above doesn't work in IE, this does
  div.setAttribute("className", "newclass");
};
</script>
```

Obviously having naming discrepancies like this can be quite frustrating so attempting to minimize the difference is generally a good idea.

Second, expandos have some name limitations as to what names they are capable of using. The ECMAScript 3.0 specification states that certain keywords aren't able to be used (such as 'for' and 'class') so alternatives must be used ('htmlFor' and 'className'). Additionally, attribute names that are two words long (e.g. 'readonly') switch to camel case ('readOnly'). Some examples of these translations can be found in Table 13.1.

Table 13.1: Cases where the expando name of an attribute is different from the original name.

Original Name	Expando Name
for	htmlFor
class	className
readonly	readOnly
maxlength	maxLength

cellspacing	cellSpacing
rowspan	rowSpan
colspan	colSpan
tabindex	tabIndex

Third, expando properties don't exist on XML nodes. Expando properties are primarily only available on HTML documents, thus you'll need to use the traditional DOM attribute methods. This isn't so bad because on XML documents there really doesn't exist the normal litany of naming mistakes that you see from DOM attributes in HTML documents.

You'll probably want some form of a check in your code to determine if an element (or document) is an XML element (or document). An example function can be seen in Listing 13.3.

Listing 13.3: A function for determining if an element (or document) is from an XML document.

```
function isXML( elem ) {
  return (elem.ownerDocument || elem)
    .documentElement.nodeName !== "HTML";
}
```

Fourth, it should be noted that not all attributes become an expando. It's generally the case for attributes that are natively specified by the browser - but not so for custom attributes specified by the user. There's a relatively simple way around this, though: If you know that you're accessing a custom attribute, just use the normal DOM attribute technique. Otherwise, you should check to see if the expando is undefined, and if so, use the normal DOM attribute technique for getting the attribute, as in Listing 13.4.

Listing 13.4: A way for getting the value of a custom attribute value.

```
<div custom="value"></div>
<script>
window.onload = function(){
  var div = document.getElementsByTagName("div")[0];

  var val = div.custom;

  if ( typeof val === "undefined" ) {
    val = div.getAttribute("custom");
  }
};
</script>
```

Last, and perhaps most importantly, expandos are much, much, faster than their corresponding DOM attribute operations (especially so in Internet Explorer). You can see the stark difference in Table 13.2.

Table 13.2: The results of running the expando and attribute performance tests, results in milliseconds.

Browser	Expando (ms)	Attribute (ms)
Firefox 3.5	55	71
Safari 4	11	15
IE 6	103	1202

Internet Explorer 6 ends up being almost 12 times slower at traditional DOM attribute access than through an expando.

For a simple test of the speed differences, as shown in Listing 13.5, the class and id attributes are accessed in a tight loop.

Listing 13.5: Comparing the performance difference between access attributes values through the DOM `getAttribute` method and an expando.

```
<html>
<head>
  <title>Attribute vs. Expando Speed</title>
</head>
<body>
<div id="test" class="test"></div>
<script>
var elem = document.getElementsByTagName("div")[0];

var start = (new Date).getTime();
expando();
var test1 = (new Date).getTime() - start;

start = (new Date).getTime();
attr();
var test2 = (new Date).getTime() - start;

alert( test1 + " " + test2 );

function expando(){
  for ( var i = 0; i < 10000; i++ ) {
    if ( elem.id === "test" ) {}
    if ( elem.className === "test" ) {}
  }
}

function attr(){
  for ( var i = 0; i < 10000; i++ ) {
    if ( elem.getAttribute("id") === "test" ) {}
    if ( elem.getAttribute("class") === "test" ) {}
  }
```

```
}
</script>
</body>
</html>
```

This difference in speed can be crippling, especially in cases where a number of queries take place (like in a CSS selector engine, for example).

The end result of having these two systems (traditional DOM attributes and expandos) is a hybrid which attempts to take the best of both worlds. An example implementation can be seen in Listing 13.6.

Listing 13.6: A simple implementation for setting and accessing attributes (using both expandos and normal DOM attributes).

```
(function(){

  // Map expando names
  var map = {
    "for": "htmlFor",
    "class": "className",
    readonly: "readOnly",
    maxlength: "maxLength",
    cellspacing: "cellSpacing",
    rowspan: "rowSpan",
    colspan: "colSpan",
    tabindex: "tabIndex"
  };

  this.attr = function( elem, name, value ) {
    var expando = map[name] || name,
      expandoExists = typeof elem[ expando ] !== "undefined";

    if ( typeof value !== "undefined" ) {
      if ( expandoExists ) {
        elem[ expando ] = value;
      } else {
        elem.setAttribute( name, value );
      }
    }

    return expandoExists ?
      elem[ expando ] :
      elem.getAttribute( name );
  };

})();
```

It should be noted, though, that the above implementation doesn't take into account many of the cross-browser issues that occur in attribute access, which will be covered in the next section.

It should be noted that expandos are also capable of holding non-string values. You can add an extra expando to an element, holding an object, a function, or some other value, and

it'll persist along with the DOM object. This will be discussed in more depth in the chapter on Events.

13.2 Cross-Browser Attributes

As mentioned before, writing cross-browser attribute-handling code can be quite harrowing - the number of cross-browser issues at play is nearly limitless. A few of the major, and most commonly encountered, issues are outlined in this section.

13.2.1 DOM ID/Name Expansion

The nastiest bug to deal with is a mis-implmentation of the original DOM code in browsers. The premise is that browsers take inputs that have a specific name or ID and add them as expandos properties to a form element. (This problem actually occurs elsewhere, as well, but this is, by far, its most prevalent form.) Both Internet Explorer and Opera take these expandos and actively overwrite any existing expandos that might already be on the DOM object.

Additionally, Internet Explorer goes a step further and sets that element as the attribute value even when you use the traditional DOM attribute methods.

An example of both of these problems can be seen in Listing 13.7.

Listing 13.7: A case where form elements expand on to a form overwriting the original ID and action attributes.

```
<html>
<head>
<title>Expansion Test</title>
<script>
window.onload = function(){
  var form = document.getElementById("form");

  // Both are DOM elements in Internet Explorer and Opera
  alert( form.id );
  alert( form.action );

  // Both are DOM elements in Internet Explorer
  alert( form.getAttribute("id") );
  alert( form.getAttribute("action") );
};
</script>
</head>
<body>
<form id="form" action="/">
  <input type="text" id="id"/>
  <input type="text" name="action"/>
</form>
</body>
</html>
```

The fact that it's now impossible to get at any of the attributes of the form element in a reliable manner an alternative technique for access needs to be devised.

One technique is to gain access to the DOM node representing the element attribute itself. This node ends up remaining untainted from the inner DOM elements. An example of this technique can be seen in Listing 13.8.

Listing 13.8: Getting access to the original values of the form element attributes.

```
<html>
<head>
<title>Expansion Test</title>
<script>
function attr( elem, name ) {
  if ( elem.nodeName.toUpperCase() === "form" &&
       elem.getAttributeNode(name) )
    return elem.getAttributeNode(name).nodeValue;
  else
    return elem.getAttribute(name);
}

window.onload = function(){
  var form = document.getElementById("form");

  // Both are 'form' and '/' accordingly
  alert( attr(form, "id") );
  alert( attr(form, "action") );
};
</script>
</head>
<body>
<form id="form" action="/">
  <input type="text" id="id"/>
  <input type="text" name="action"/>
</form>
</body>
</html>
```

If you're interested in the sort of problems that arise from these element expansions I recommend checking out Juriy Zaytsev's DOMLint tool, which is capable of analyzing a page for potential problems and Garrett Smith's write-up the issue, as a whole.

- DOMLint: http://yura.thinkweb2.com/domlint/
- Unsafe Names for HTML Form Controls: http://jibbering.com/faq/names/

13.2.2 URL Normalization

There's a bug in Internet Explorer that violates the principle of least surprise: When accessing an attribute that references a URL (such as href, src, or action) that URL is automatically converted from its specified form into a full, canonical, URL. For example if `href="/test/"` was specified then the value retrieved from `.href` or `.getAttribute("href")` would be the full URL: `http://example.com/test/`.

Incidentally, Internet Explorer (the only browser to fail this measure), also provides an addition to the `getAttribute` method. If you pass in an extra attribute at the end with a

value of '2' it'll leave the URL value as it was originally entered. When can use this fact to develop a method that handles this workaround, as in Listing 13.9.

Listing 13.9: An example workaround for handling attributes that URLs in them.

```
<html>
<head>
<title>Attribute URL Test</title>
<script>
(function(){
  var div = document.createElement("div");
  div.innerHTML = "<a href='/'></a>";

  var urlok = div.firstChild.href === "/";

  this.urlAttr = function(elem, name, value) {
    if ( typeof value !== "undefined" ) {
      elem.setAttribute(name, value);
    }

    return urlok ?
      elem[ name ] :
      elem.getAttribute(name, 2);
  };
})();

window.onload = function(){
  var a = document.getElementById("a");

  // Alerts out the full URL 'http://example.com/'
  alert( a.getAttribute("href") );

  // Alerts out the input '/'
  alert( urlAttr( "href", a ) );
};
</script>
</head>
<body>
  <a id="a" href="/"></a>
</body>
</html>
```

A brief note: Be sure that you don't use `getAttribute(..., 2)` indiscriminately. Older versions of Opera would crash whenever the extra '2' was passed in (for no apparent reason).

13.2.3 Style Attribute

One attribute that's particularly challenging to set and get the value of is the style attribute of an element. HTML DOM elements have a style object property which you can access to gain information about the style information of an element (e.g. `elem.style.color`) however if you want to get the full style string that was set

on the element it becomes much more challenging (`<div style='color:red;'></div>`).

To start, expandos don't work here because the 'style' expando is already taken up by the 'style' object. So while `get/setAttribute("style")` work in most browsers, it doesn't work in Internet Explorer. Instead, IE has a property of the style object which you can get/set to affect the style attribute: `elem.style.cssText`.

Listing 13.10 has an example of a method dedicated to accessing and setting the style attribute of a DOM element.

Listing 13.10: Create a method for getting and setting the style attribute string across browsers. (References code from Listing 13.3.)

```
<html>
<head>
<title>Style Attribute Test</title>
<script>
(function(){
  // Create a dummy div to test access the style information
  var div = document.createElement("div");
  div.innerHTML = '<b style="color:red;"></b>';

  // Get the style information from getAttribute
  // (IE uses .cssText insted)
  var hasStyle = /red/.test( div.firstChild.getAttribute("style") );

  this.styleAttr = function(elem, value) {
    var hasValue = typeof value !== "undefined";

    // Use the isXML method from Listing 13.3
    if ( !hasStyle && !isXML(elem) ) {
      if ( hasValue ) {
        elem.style.cssText = value + "";
      }

      return elem.style.cssText;
    } else {
      if ( hasValue ) {
        elem.setAttribute("style", value);
      }

      return elem.getAttribute("style");
    }
  };
})();

window.onload = function(){
  var a = document.getElementById("a");

  // Fails in Internet Explorer
  alert( a.getAttribute("style") );

  // Alerts out "color:red;"
  alert( styleAttr( a ) );
```

```
};
</script>
</head>
<body>
  <a id="a" href="/" style="color:red;"></a>
</body>
</html>
```

While directly getting and setting the style attribute is comparatively uncommon (compared to accessing/mutating the style property object directly) it is still quite useful for getting a full serialization of all the style information currently on an element.

13.2.4 Type Attribute

There's another Internet Explorer gotcha related to the type attribute of form inputs - to which there isn't a reasonable fix. Once an input element has been inserted into a document its type attribute can no longer be changed (and, in fact, throws an exception if you attempt to change it).

An example of the problem can be seen in Listing 13.11.

Listing 13.11: Attempting to change an input element's type attribute after it has already been inserted into a document.

```
<html>
<head>
<title>Type Attribute Test</title>
<script>
window.onload = function(){
  var input = document.createElement("input");
  input.type = "text";

  document.getElementById("form").appendChild( input );

  // Errors out in Internet Explorer
  input.type = "hidden";
};
</script>
</head>
<body>
<a id="a" href="/"></a>
<form id="form" action="/">
  <input type="text" id="id"/>
  <input type="text" name="action"/>
</form>
</body>
</html>
```

There are two possible stopgap solutions:

1. Create a new input and copy over any existing properties, attributes, and event handlers (assuming that you've been tracking the event handlers to begin with). Unfortunately any

references to the old element will still exist, you would need to have a way to change all of those, as well.
2. Simply reject any attempts to change the type attribute at the API level.

jQuery employs the second method (throwing an informative exception if the users attempts to change the type attribute after it has already been inserted into the document). Obviously this isn't a terribly good 'solution' but at least the user experience is consistent across all platforms.

13.2.5 Tab Index

Determining the tab index of an element is another weird attribute - and one where there's little consensus at to how it should work. While it's possible to get the tab index of an element using the traditional means (`.tabIndex` and `.getAttribute("tabindex")`) the result is quite inconsistent for elements that haven't had a tab index explicitly defined.

For example, a div that has no tab index specified should return undefined - but Internet Explorer 6 and 7 return 0 and Firefox 2 and 3 return -1 (when using `.tabIndex`).

There is a workaround, though. We can use the `getAttributeNode` method to get at the node representing the tabindex attribute. Once we have that we can check its 'specified' property to see if the user has ever specified a tab index (this will be false if the browser is trying to use guesswork and give you a bogus number instead of undefined).

Listing 13.12 shows how we can use this technique to get at the correct tab index value and how we can report the correct tab index for attributes that should (or shouldn't) have one.

Listing 13.12: A method for getting the correct tab index of an element.

```
(function(){
  var check = /^(button|input|object|select|textarea)$/i,
      a = /^(a|area)$/i;

  this.getIndex = function(elem){
    var attributeNode = elem.getAttributeNode("tabIndex");
    return attributeNode && attributeNode.specified ?
      attributeNode.value :
      check.test( elem.nodeName ) ||
          a.test( elem.nodeName ) && elem.href ?
        0 :
        undefined;
  };
})();
```

This is a complex issue and is one that is especially important in the world of usability and accessibility. The Fluid Project has been doing a lot of research into the area and has written up some comprehensive information on the subject matter.

- http://fluidproject.org/blog/2008/01/09/getting-setting-and-removing-tabindex-values-with-javascript/

13.2.6 Node Name

While this isn't related to an attribute, per se, determining the name of a node can be slightly tricky. Specifically, the case sensitivity of the name changes depending upon which type of document you are in. If it's a normal HTML document then the `nodeName` property will return the name of the element in all uppercase (e.g. "HTML" or "BODY"). However, if it's in an XML or XHTML document then the `nodeName` will return the name as specified by the user (meaning that it could be lowercase, uppercase, or a mix of both).

If you have a DOM node, and you know which `nodeName` you're looking for, you can something like in Listing 13.13.

Listing 13.13: Normalizing the nodeName of an element in order to simplify node name checks.

```
<div></div>
<ul></ul>
<script>
window.onload = function(){
  var all = document.getElementsByTagName("*")[0];

  for ( var i = 0; i < all.length; i++ ) {
    var nodeName = all[i].nodeName.toLowerCase();
    if ( nodeName === "div" || nodeName === "ul" ) {
      all[i].className = "found";
    }
  }
};
</script>
```

If you already know all the documents in which you'll be checking `nodeNames` then you probably won't have to worry about this case sensitivity. However, if you're writing reusable code that could run in any environment, it's best to be adaptive and gracefully handle the different pages.

13.3 CSS

As with attributes, getting and setting CSS attributes can be quite tricky. Similarly to attributes we are, again, presented with two APIs for handling style values. The most commonly used API is the `.style` property of an element (which is an object holding properties corresponding to set style properties). On top of this there is an API for accessing the current, computed, style information of an element.

Style information (located on the `.style` property of a DOM element object) comes from two places: If an inline style attribute is declare (e.g. `style="color:red;"`) then that style information will be put in the style object. Also, if a user sets a value on the style object it will affect the display of the element and store the value on the object.

No values from internal, or external, stylesheets will be available on the element's style object (computed style will be required to access that information). Listing 13.14 has an example of the limitations of the style object.

Listing 13.14: An example of where the properties on an element's style property come from.

```
<div style="color:red;">text</div>
<style>div { font-size: 10px; }</style>
<script>
window.onload = function(){
  var div = document.getElementsByTagName("div")[0];

  // Will alert 'red' or '#ff0000' (in Opera)
  alert( div.style.color );

  // Will be an empty alert
  alert( div.style.fontSize );

  div.style.backgroundColor = "blue";

  // Will alert 'blue' or '#0000ff' (in Opera)
  alert( div.style.backgroundColor );
};
</script>
```

It should be noted that any values in an element's style object will take precedence over anything specified by a stylesheet (even if the stylesheet rule uses `!important`). In the end it is the ultimate arbiter of what an element looks like. For this reason, in your CSS property access code, you should always check the value of the `.style` property first as it'll have the most definite state in it.

With CSS attributes there are relatively few cross-browser difficulties when it comes to accessing the values provided by the browser. CSS attributes frequently use names like "font-weight" and "background-color" - all of which need to be converted into camel case in order to be accessed correctly (two examples of this were shown in Listing 13.14).

Note browser-specific attributes. They generally start with a prefix that corresponds to the browser, for example: `"-moz-border-radius"`. When accessing the value of this attribute through JavaScript it'll need to be camel cased as such: `"MozBorderRadius"`. However, while Mozilla allows you to both read and write custom attributes WebKit only allows you to write to them (at least via `.style`, it allows you to read the computed value).

Some simple code for converting the attribute name to camel case can be see in Listing 13.15.

Listing 13.15: A simple method for access style information of an element, handling correct camel-casing.

```
<div style="color:red;font-size:10px;-moz-border-radius:3px;"></div>
<script>
function style(elem, name, value){
  name = name.replace(/-([a-z])/ig, function(all, letter){
    return letter.toUpperCase();
  });

  if ( typeof value !== "undefined" ) {
```

```
    elem.style[ name ] = value;
  }

  return elem.style[ name ];
}

window.onload = function(){
  var div = document.getElementsByTagName("div")[0];

  // Will alert 'red' or '#ff0000' (in Opera)
  alert( style(div, "color") );

  // Will alert '10px'
  alert( style(div, "font-size") );

  // Will alert '3px 3px 3px 3px' in Firefox
  alert( style(div, "-moz-border-radius") );
};
</script>
```

13.3.1 Float Property

The one, major, naming bug that exists with CSS attributes relates to how the float attribute is called. Since the float property conflicts with the built in float keyword in JavaScript the browsers have to provide an alternative name.

Nearly all browsers choose to use `'cssFloat'` as the alternative name whereas Internet Explorer uses `'styleFloat'`.

A simple function for handling these differences can be seen in Listing 13.16.

Listing 13.16: An example of retrieving the float style attribute across browsers.

```
<html>
<head>
<title>Float Style Test</title>
<script>
(function(){
  var div = document.createElement("div");
  div.innerHTML = "<div style='float:left;'></div>";

  var floatName = div.firstChild.style.cssFloat === "left" ?
    "cssFloat" :
    "styleFloat";

  this.floatStyle = function(elem, value) {
    if ( typeof value !== "undefined" ) {
      elem.style[ floatName ] = value;
    }

    return elem.style[ floatName ];
  };
})();

window.onload = function(){
```

```
    var div = document.getElementById("div");

    // Alerts out 'left'
    alert( floatStyle( div ) );
  };
  </script>
  </head>
  <body>
    <div style="float:left;"></div>
    <a id="a" href="/"></a>
  </body>
  </html>
```

As seen in Listing 13.16 it's pretty easy to use feature detection to make sure that the property exists before using attempting to access it. After that we can make a fairly safe assumption that if the spec-compatible `cssFloat` doesn't work then `styleFloat` will.

13.3.2 Raw Pixel Values

The last point to consider when setting a style value is the setting of numbers to properties that normal consume pixels. When setting a number into a style property you must specify the unit in order for it to work across all browsers.

```
// Works across browsers
elem.style.height = "10px";
elem.style.height = 10 + "px";

// Doesn't work across browsers
elem.style.height = 10;
```

We can work around this by watching for when a number comes in as a value and automatically add on a the "px" suffix. Of course, we have to be careful of cases where we actually don't want to add a suffix, for which there are a few style attributes:

- z-index
- font-weight
- opacity
- zoom
- line-height

Additionally, when attempting to read a pixel value out of a style attribute the `parseFloat` method should be used.

Altogether a unified method can be created for handling both of these cases, as in Listing 13.17.

Listing 13.17: A method for getting and setting pixel-based style attributes using numbers.

```
<html>
<head>
<title>Pixel Style Test</title>
<script>
(function(){
  var check = /z-?index|font-?weight|opacity|zoom|line-?height/i;

  this.pxStyle = function(elem, name, value) {
    var nopx = check.test( name );

    if ( typeof value !== "undefined" ) {
      if ( typeof value === "number" ) {
        value += nopx ? "" : "px";
      }
      elem.style[ name ] = value;
    }

    return nopx ?
      elem.style[ name ] :
      parseFloat( elem.style[ name ] );
  };
})();

window.onload = function(){
  var div = document.getElementById("div");

  pxStyle( div, "top", 5 );

  // Alerts out '5'
  alert( pxStyle( div, "left" ) );
};
</script>
</head>
<body>
  <div style="top:10px;left:5px;"></div>
</body>
</html>
```

These concerns cover most of what you have to worry about when it comes to handling the `.style` property of an element. There's still one, large, missing gap though: Retrieving the current computed style of an element.

13.3.3 Computed Style

The computed style of an element is a combination of the styles applied to it via stylesheets, the inline `style` attribute, and any manipulations of the `.style` property.

The primary API specified by the W3C (and implemented by all browsers but Internet Explorer) is the `getComputedStyle` method on the document's defaultView. The method takes an element and returns an interface through which property queries can

be made. The interface provides the `getPropertyValue( name )` for retrieving the computed style of a specific property.

Contrary to the `.style` object the `getPropertyValue( name )` method only accepts method names that are in the original CSS property style (e.g. "font-size").

As alluded to before, Internet Explorer has a completely different technique for accessing the computed style of an element. Incidentally, it's also a dramatically simpler API. A single .currentStyle property is provided on all elements and it behaves exactly like the .style property, except providing live, computed, style information.

Listing 13.18 has an example function which is able to access the computed style information for an element using the W3C-compatible API or Internet Explorer's `currentStyle` property.

Listing 13.18: A function for accessing computed style information using the W3C-compatible API or Internet Explorer's `currentStyle`.

```
<div style="color:red;">text</div>
<style>div { font-size: 10px; display: inline; }</style>
<script>
function computedStyle( elem, name ) {
  var defaultView = elem.ownerDocument.defaultView;

  if ( defaultView && defaultView.getComputedStyle ) {
    var computedStyle = defaultView.getComputedStyle( elem, null );

    if ( computedStyle ) {
      name = name.replace(/([A-Z])/g, "-$1").toLowerCase();
      return computedStyle.getPropertyValue( name );
    }
  } else if ( elem.currentStyle ) {
    name = name.replace(/-([a-z])/ig, function(all, letter){
      return letter.toUpperCase();
    });

    return elem.currentStyle[ name ];
  }
}

window.onload = function(){
  var div = document.getElementsByTagName("div")[0];

  // Will alert '10px'
  alert( computedStyle( div, "font-size" ) );

  // Will alert '10px'
  alert( computedStyle( div, "fontSize" ) );

  // Will alert 'inline'
  alert( computedStyle( div, "display" ) );
};
</script>
```

Internet Explorer's `currentStyle` property has an added benefit. The W3C way of accessing a computed value will always return a numerical result in pixels. Thus if you have a font-size measured in '2em' the computed font-size that you'll get back will be something like '14.45px'. This isn't terribly useful, especially if you want to get at the original value provided by a stylesheet.

Internet Explorer's `currentStyle` has no such problem. A font size of '2em' will come out as '2em'. This ends up being simultaneously useful and frustrating. Since the API is different from the W3C-style API (that every other browser provides) we're forced to bang it into shape (forcing it to return pixel values whenever it returns numbers based in another unit).

There's one very interesting hack written by Dean Edwards that is able to handle quite a few of the common conversion cases. The code for it is shown in Listing 13.19.

- Conversion trick by Dean Edwards: http://erik.eae.net/archives/2007/07/27/18.54.15/#comment-102291

Listing 13.19: Dean Edward's technique for getting at pixel values for different united properties.

```
// If we're not dealing with a regular pixel number
// but a number that has a different ending, we need to convert it
if ( /^[0-9.]+(em|ex|pt|%)?$/i.test( value ) ) {
  // Remember the original values
  var left = elem.style.left, rsLeft = elem.runtimeStyle.left;

  // Put in the new values to get a computed value out
  elem.runtimeStyle.left = elem.currentStyle.left;
  elem.style.left = value || 0;
  ret = elem.style.pixelLeft + "px";

  // Revert the changed values
  elem.style.left = left;
  elem.runtimeStyle.left = rsLeft;
}
```

This technique works by using a couple Internet Explorer-specific features. We're already familiar with `.currentStyle` but there's also `.runtimeStyle` and `.style.pixelLeft`.

`.runtimeStyle` is analogous to `.style` with one exception: Any style properties set using .runtimeStyle.prop will override `.style.prop` (which still leaving the value contained in `.style.prop` intact).

`.pixelLeft` (and Top, Bottom, Right, Height, and Width) are properties for directly accessing the pixel value of the respective CSS properties. (Note: These could be used as alternatives to even using Dean Edwards' technique in the first place for these properties - Dean's technique is more useful for things like font-size, line-height, etc.).

The technique works by setting the current computed left to the runtime left ('left' is chosen arbitrarily - it could also work with top, right, etc.) - this will keep the positioning of the element intact while we compute the pixel value. The pixel value is computed by setting

the `.style.left` and then reading the resulting pixel value out using `.style.pixelLeft`.

Dean's technique isn't perfect - especially when attempting to handle percentages on elements that are a child of the body element (they end up expanding to consume a large portion of the document - 10% becomes 200px, for example) but it ends up working well enough to handle most of the common use cases.

AUTO VALUES

One aspect that is particularly hard to solve using Internet Explorer's `currentStyle` technique is that of 'auto' values. For example if an element has no specific height or width defined then it defaults to 'auto' (automatically expanding and contracting based upon its contents). Thankfully it's possible to find the correct value of those properties in other ways, which we'll discuss later in this chapter.

However there do still exist some properties that fall prey to this problem, like margin and font-size.

If you want to figure out the current pixel value that corresponds to an auto margin, as it currently appears, you'll have to perform some extra calculations and gymnastics - many of which will have to depend upon the specific code base. For example, figuring out the pixel margin of an element that's center-aligned with a margin of auto would be a piece of work in and of itself.

Font size can actually be a little bit easier since a font-size of auto corresponds to a size of 1em. You can work your way up the tree to figure out precisely what '1em' means (in pixel values) or use Dean's trick from the previous section.

One library that obsessively tackles the problem of computed font size is called CSSStyleObject and can be found here:

- http://www.strictly-software.com/CSSStyleObject.asp

CATCH-ALL PROPERTIES

CSS properties that are actually amalgams of sub-properties (like margin, padding, border, and background-position) don't provide consistent results across all browsers (more than likely they simply don't provide any results).

Thus, in order to figure out what the actual border is for an element you'll need to check the border-top-width, border-bottom-width, border-left-width, and border-right-width to get a clear picture.

It's a hassle, especially when all four values have the same result (and especially when something like `border: 5px;` was used in the stylesheet directly) but that's the hand we've been dealt.

13.3.4 Height and Width

As mentioned before, when looking at properties that could have an auto value, it was mentioned that height and width are two properties that frequently have these values.

Because of this we'll need to utilize a different technique to get at the actual height and width of an element.

Thankfully the `offsetHeight` and `offsetWidth` properties provide just that: A fairly reliable place to access the correct height and width of an element. The numbers returned by the two properties include the padding of the element as well. This information may be desired (if you're attempting to position an element over another one, for example, you'll want to leave the padding in). In reality you'll probably want to have a few methods for accessing an elements height or width (height with no padding, height with padding, and height with padding and border).

There is one gotcha, though: If you want to find out the height and width of an element that is hidden (by its display property being set to 'none'), offsetWidth and offsetHeight will both return 0. One way to workaround this problem is to temporarily make the element visible, but in a hidden state).

We can achieve this by setting the `display` property to `block` (making it visible and able to be measured) and immediately set its `visibility` to `hidden` (making it invisible to the user) and finally set its `position` to `absolute` (taking it out of the flow of the document, stopping it from pushing any other elements out of the way).

Listing 13.20 has an example of setting these properties on an element, getting its height or width, and immediately setting the properties back.

Listing 13.20: An example of getting the height and width of an element (both visible and invisible).

```
<html>
<head>
  <title>Height/Width Test</title>
  <script>
  (function(){

    this.height = function( elem ) {
      return this.offsetHeight ||
        swap( elem, props, function(){
          return elem.offsetHeight;
        });
    };

    this.width = function( elem ) {
      return this.offsetWidth ||
        swap( elem, props, function(){
          return elem.offsetWidth;
        });
    };

    var props = {
      position: "absolute",
      visibility: "hidden",
      display: "block"
    };
```

```
    function swap( elem, options, callback ) {
      var old = {}, ret;
      // Remember the old values, and insert the new ones
      for ( var name in options ) {
        old[ name ] = elem.style[ name ];
        elem.style[ name ] = options[ name ];
      }

      ret = callback();

      // Revert the old values
      for ( var name in options ) {
        elem.style[ name ] = old[ name ];
      }

      return ret;
    }

  })();

  window.onload = function(){
    var block = document.getElementById("block");
    var none = document.getElementById("none");

    // Both will alert out the same values
    alert( width(block) + " " + height(block) );
    alert( width(none) + " " + height(none) );
  };
  </script>
</head>
<body>
  <div id="block">Test</div>
  <div id="none" style="display:none;">Test</div>
</body>
</html>
```

One thing to note in the code of Listing 13.20 is the use of `offsetHeight` or `offsetWidth` as a crude means of determining the visibility of an element. As it turns out this serves to be an incredibly efficient way to do just that. We can assume that any element that has both an `offsetHeight` and `offsetWidth` not equal to zero then it must be visible. Likewise if they're both zero then it's definitely not visible. Ambiguity comes in when either one or the other is not zero (but we can fall back to our `computedStyle` method from Listing 13.18).

The one gotcha that exists is that a tr element in Internet Explorer will always return zero (even if it's visible). Thus we work around this by always checking the computed style of that element.

Listing 13.21 has a sample method for determining the visibility of an element.

Listing 13.21: An example of getting the visibility of an element (both visible and invisible). Uses code from Listing 13.18.

```
<html>
<head>
  <title>Visibility Test</title>
  <script>
  function isVisible( elem ) {
    var isTR = elem.nodeName.toLowerCase() === "tr",
      w = elem.offsetWidth, h = elem.offsetHeight;
    return w !== 0 && h !== 0 && !isTR ?
      true :
      w === 0 && h === 0 && !isTR ?
        false :
        computedStyle( elem, "display" ) === "none";
  }

  window.onload = function(){
    var block = document.getElementById("block");
    var none = document.getElementById("none");

    // Alerts out 'true'
    alert( isVisible(block) );

    // Alerts out 'false'
    alert( isVisible(none) );
  };
  </script>
</head>
<body>
  <div id="block">Test</div>
  <div id="none" style="display:none;">Test</div>
</body>
</html>
```

While height and width do end up needing to be handled in a unique manner that means through which that occurs isn't overly difficult. Additionally the results work across all platforms without much browser-specific trickery.

13.3.5 Opacity

Opacity is a special case that needs to be handled differently in Internet Explorer from other browsers. All other modern browsers support the opacity CSS property whereas Internet Explorer uses its proprietary alpha filter.

An example of how the rules might be defined in a stylesheet:

```
opacity: 0.5;
filter: alpha(opacity=50);
```

The problem would be pretty simple if `alpha` was the only type of filter allowed, but it's not (there are many different filters, including transformations, available in Internet Explorer). Thus we'll have to do some extra parsing to arrive at our desired result.

Listing 13.21 has an example of how to handle opacity across all browsers.

Listing 13.21: An example of retrieving the opacity style attribute across browsers.

```
<html>
<head>
<title>Opacity Style Test</title>
<script>
(function(){
  var div = document.createElement("div");
  div.innerHTML = "<div style='opacity:.5;'></div>";

  // If .5 becomes 0.5, we assume the browser knows how
  // to handle native opacity
  var native = div.firstChild.style.opacity === "0.5";

  this.opacityStyle = function(elem, value) {
    if ( typeof value !== "undefined" ) {
      if ( native ) {
        elem.style.opacity = value;
      } else {
        // .filter may not exist
        elem.style.filter = (elem.style.filter || "")
          // Need to replace any existing alpha
          .replace( /alpha\([^)]*\)/, "" ) +
          // If we have a number, set that as the value (0-100)
          (parseFloat( value ) + '' == "NaN" ?
            "" :
            "alpha(opacity=" + value * 100 + ")");
      }

      elem.style[ floatName ] = value;
    }

    if ( native ) {
      return elem.style.opacity;
    } else {
      // Only attempt to get a value if one might exist
      var match = elem.style.filter.match(/opacity=([^)]*)/);
      return match ?
        (parseFloat( match[1] ) / 100) + "" :
        "";
    }
  };
})();

window.onload = function(){
  var div = document.getElementById("div");

  // Alerts out '0.5'
  alert( opacityStyle( div ) );
};
```

```
</script>
</head>
<body>
  <div style="opacity:0.5;filter:alpha(opacity=50);"></div>
</body>
</html>
```

Getting at the computed opacity works in a very similar manner. Instead of access the `.style.filter` property we look at `.currentStyle.filter` instead.

The only other change to accessing computed opacity is that when no value is found (in Listing 13.21 we just returned an empty string) we need to return "1" instead (since the default value for opacity is completely opaque).

13.3.6 Colors

Handling color values in CSS can be tricky. When accessing them via `.style` you're at the mercy of whatever names/syntaxes the user chose. When you're accessing them via the different computed style methods there is no particular census around what values the method should return.

Because of this, any attempts to gain access to the useful parts of a color (presumably its red, blue, green color channel information) will require some legwork.

In total there are five separate ways in which color information can be represented:

- `rgb(R,G,B)` - Where R, G, B are numbers from 0 to 255.
- `rgb(R%,G%,B%)` - Where R, G, B are numbers from 0% to 100%.
- #RRGGBB - Where RR, GG, BB are hexadecimal representations of the numbers 0 through 255.
- #RGB - Where R, G, B are hexadecimal representations similar to the previous one, but in shorthand (e.g. #F54 is equal to #FF5544).
- `red`, `blue`, etc. - The name representing a set of RGB values.

All together this can result in a lot of code dedicated to handling these particulars. There are a lot of implementations floating around that tackle the same problem, one such one is shown in Listing 13.22. This code comes from the jQuery Color plugin with code written by Blair Mitchelmore (for the highlightFade plugin) and Stefan Petre (for the Interface plugin).

- jQuery Color Plugin: http://plugins.jquery.com/project/color
- highlightFade Plugin: http://jquery.offput.ca/highlightFade/
- Interface Plugin: http://interface.eyecon.ro/

Listing 13.22: A method for converting a string value into an array of RGB values (including the conversion of color names to values - full list is available in the source links).

```
var num = /rgb\(\s*([0-9]+)\s*,\s*([0-9]+)\s*,\s*([0-9]+)\s*\)/,
    pc = /rgb\(\s*([0-9.]+)%\s*,\s*([0-9.]+)%\s*,\s*([0-9.]+)%\s*\)/,
    hex = /#([a-fA-F0-9]{2})([a-fA-F0-9]{2})([a-fA-F0-9]{2})/,
    hex2 = /#([a-fA-F0-9])([a-fA-F0-9])([a-fA-F0-9])/;

// Parse strings looking for color tuples [255,255,255]
function getRGB(color) {
  var result;

  // Look for rgb(num,num,num)
  if (result = num.exec(color))
    return [parseInt(result[1]), parseInt(result[2]),
      parseInt(result[3])];

  // Look for rgb(num%,num%,num%)
  if (result = pc.exec(color))
    return [parseFloat(result[1])*2.55, parseFloat(result[2])*2.55,
      parseFloat(result[3])*2.55];

  // Look for #a0b1c2
  if (result = hex.exec(color))
    return [parseInt(result[1],16), parseInt(result[2],16),
      parseInt(result[3],16)];

  // Look for #fff
  if (result = hex2.exec(color))
    return [parseInt(result[1]+result[1],16),
      parseInt(result[2]+result[2],16),
      parseInt(result[3]+result[3],16)];

  // Otherwise, we're most likely dealing with a named color
  return colors[color.replace(/\s+/g, "").toLowerCase()];
}

// Map color names to RGB values
var colors = {
  aqua:[0,255,255],
  azure:[240,255,255],
  beige:[245,245,220],
  black:[0,0,0],
  blue:[0,0,255],
  // ... snip ...
  silver:[192,192,192],
  white:[255,255,255],
  yellow:[255,255,0]
};
```

Some libraries choose to not handle this functionality natively and push it off into an add-on (as is the case in jQuery). Generally color conversion is most useful when doing

animations (animating from one color to another) so its inclusion really only occurs when that functionality is desired.

13.4 Summary

Getting and setting DOM attributes and CSS is the area of JavaScript development most fraught with cross-browser compatibility issues. Thankfully the issues can be handled in a way that is particularly cross-browser compliant without resorting to browser-specific code.

14 Events

In this chapter:

- Techniques for binding events
- Custom events and event triggering
- Event bubbling and delegation

DOM event management is a problem that should be relatively simple, all browsers provide relatively stable APIs for managing events (although, with different implementations). Unfortunately the features provided by browsers are insufficient for most of the tasks that need to be handled by sufficiently-complex applications. Because of these shortcomings the end result is a near-full duplication of the existing DOM APIs in JavaScript.

14.1 Binding and Unbinding Event Handlers

When binding and unbinding event handlers you typically use with the browser's native methods: addEventListener and removeEventListener for DOM-compliant browsers and attachEvent and detachEvent in Internet Explorer. For the most part the two techniques behave very similarly but with one exception: Internet Explorer's event system doesn't provide a way to listen for the capturing stage of an event, only the bubbling phase (after the event occurs and begins to traverse back up the DOM tree).

Additionally, Internet Explorer's event implementation doesn't properly set a context on the bound handler (thus if you try to access the 'this' inside the handler you'll just get the global object instead of the current element). For this reason the handler provided by the user shouldn't be bound directly to the element in Internet Explorer and should, instead, bind an intermediary handler that corrects the context before executing the handler.

These techniques can be seen together in Listing 14.1.

Listing 14.1: Simple methods for binding and unbinding an event handler while preserving the context. Also a proxy method for changing the context for any given function.

```
<script>
(function(){
  if ( document.addEventListener ) {
    this.addEvent = function( elem, type, fn ) {
      elem.addEventListener( type, fn, false );
      return fn;
    };

    this.removeEvent = function( elem, type, fn ) {
      elem.removeEventListener( type, fn, false );
    };

  } else if ( document.attachEvent ) {
    this.addEvent = function( elem, type, fn ) {
      var bound = function() {
        return fn.apply( elem, arguments );
      };

      elem.attachEvent( "on" + type, bound );
      return bound;
    };

    this.removeEvent = function( elem, type, fn ) {
      elem.detachEvent( "on" + type, fn );
    };
  }
})();

addEvent( window, "load", function() {
  var li = document.getElementsByTagName("li");
  for ( var i = 0; i < li.length; i++ ) (function(elem){
    var handler = addEvent( elem, "click", function() {
      this.style.backgroundColor = "green";
      removeEvent( elem, "click", handler );
    });
  })(li[i]);
});
</script>
<ul>
<li>Click</li>
<li>me</li>
<li>once.</li>
</ul>
```

The code in Listing 14.1 exposes three new methods: `addEvent`, `removeEvent`, and `proxy.addEvent` and `removeEvent` work exactly as advertised: You provide a DOM element, document, or window object, the type of the event, and the function that you wish to bind/unbind and it'll attach an event to that object during the bubbling phase.

The one bit of trickiness is with the context-fixing function for Internet Explorer. Since we've used an intermediary function to correct the context in Internet Explorer the normal

handler will no longer be able to remove the same handler function that we put in. Instead the addEvent function now returns the new function that has been bound (which we will use to unbind the handler later on). A better workaround is to use a separate object to store all the event handlers, which we will discuss in the next section.

14.2 The Event Object

Many browsers, especially Internet Explorer, miss out on a number of the W3C event object properties and methods. The only reasonable way to work around this is to create a new object that simulates the browser's native event object, fixing issues on it where appropriate (it's not always possible to modify the existing object, many properties can't be overwritten).

To start, it's important to note that the event object isn't passed in as the first argument to the bound handler in Internet Explorer instead it's contained within a global object named `event`. Having a single event object makes it all the more important to clone it and transfer the properties to a new object, since once a new event begins the old event object will go away.

A function for event normalization can be found in Listing 14.2.

Listing 14.2: Implement some basic event object normalization.

```
function fixEvent( event ) {
  if ( !event || !event.stopPropagation ) {
    var old = event || window.event;

    // Clone the old object so that we can modify the values
    event = {};

    for ( var prop in old ) {
      event[ prop ] = old[ prop ];
    }

    // The event occurred on this element
    if ( !event.target ) {
      event.target = event.srcElement || document;
    }

    // Handle which other element the event is related to
    event.relatedTarget = event.fromElement === event.target ?
      event.toElement :
      event.fromElement;

    // Stop the default browser action
    event.preventDefault = function() {
      event.returnValue = false;
      event.isDefaultPrevented = returnTrue;
    };

    event.isDefaultPrevented = returnFalse;

    // Stop the event from bubbling
    event.stopPropagation = function() {
```

```
      event.cancelBubble = true;
      event.isPropagationStopped = returnTrue;
    };

    event.isPropagationStopped = returnFalse;

    // Stop the event from bubbling and executing other handlers
    event.stopImmediatePropagation = function() {
      this.isImmediatePropagationStopped = returnTrue;
      this.stopPropagation();
    };

    event.isImmediatePropagationStopped = returnFalse;

    // Handle mouse position
    if ( event.clientX != null ) {
      var doc = document.documentElement, body = document.body;

      event.pageX = event.clientX +
        (doc && doc.scrollLeft || body && body.scrollLeft || 0) -
        (doc && doc.clientLeft || body && body.clientLeft || 0);
      event.pageY = event.clientY +
        (doc && doc.scrollTop || body && body.scrollTop || 0) -
        (doc && doc.clientTop || body && body.clientTop || 0);
    }

    // Handle key presses
    event.which = event.charCode || event.keyCode;

    // Fix button for mouse clicks:
    // 0 == left; 1 == middle; 2 == right
    if ( event.button != null ) {
      event.button = (event.button & 1 ? 0 :
        (event.button & 4 ? 1 :
          (event.button & 2 ? 2 : 0)));
    }
  }

  return event;

  function returnTrue() { return true; }
  function returnFalse() { return false; }
}
```

The `fixEvent` function in Listing 14.2 handles many of the common discrepancies between the W3C DOM event object and the one provided by Internet Explorer. You could use it within your handlers to get a usable event object back.

A few of the concerns that are fixed in this process:

- `.target` The commonly-used property denoting the original source of the event. IE frequently stores this in `.srcElement`.
- `.relatedTarget` comes into use when it's used on an event that works in

conjunction with another element (such as "mouseover" or "mouseout"). The `.toElement` and `.fromElement` are IE's counterparts.

• `.preventDefault()` doesn't exist in the IE implementation, which would normally prevent the default browser action from occurring, instead the `.returnValue` property needs to be set to false.

• `.stopPropagation()` also doesn't exist, which would normally stop the event from bubbling further up the tree. Setting the `.cancelBubble` property to true will make this happen.

• `.pageX` and `.pageY` don't exist in IE (provide the position of the mouse relative to the whole document) but can be easily duplicated using other information (`clientX/Y` provides the position of the mouse relative to the window, `scrollTop/Left` gives the scrolled position of the document, and `clientTop/Left` gives the offset of the document itself - combining these three will give you the final `pageX/Y` values).

• `.which` is equivalent to the key code pressed during a keyboard event, this can be duplicated by accessing the `.charCode` and `.keyCode` properties.

• `.button` matches the mouse button clicked by the user on a mouse event. Internet Explorer uses a bitmask (1 for left client, 2 for right click, 4 for middle click) so it needs to be converted to their equivalent values (1, 2, and 3).

Some of the best information on the DOM event object and its cross-browser capabilities can be found on the Quirksmode compatibility tables:

• Event object compatibility: http://www.quirksmode.org/dom/w3c_events.html

• Mouse position compatibility: http://www.quirksmode.org/dom/w3c_cssom.html#mousepos

Additionally issues surrounding the nitty-gritty of keyboard and mouse event object properties can be found in the excellent JavaScript Madness guide:

- • Keyboard events: http://unixpapa.com/js/key.html
- • Mouse events: http://unixpapa.com/js/mouse.html

14.2.1 Handler Management

For a number of reasons it's not advisable to bind an event handler directly to an element, you'll want to use an intermediary event handler instead and store all the handlers in a separate object. These reasons include:

• Normalizing the context of handlers.

- Fixing the properties of event objects.
- Handling garbage collection of bound handlers.
- Triggering or removing some handlers by a filter.
- Unbinding all events of a particular type.
- Cloning of event handlers.

You need to have access to the full list of the handlers bound to an element in order to be achieve all of these points. At which time it really makes the most sense to just avoid directly binding the events.

The best way to manage the handlers associated with a DOM node is to give each node that your work a unique ID and then store all data in a centralized object. Keeping the data in a central store, in this manner, helps to avoid potential memory leaks in Internet Explorer (attaching functions to a DOM element that have a closure to a DOM node is capable of causing memory to be lost after navigating away from a page).

Listing 14.3: An example of a central object store for DOM elements.

```
(function(){
var cache = {}, guid = 1, expando = "data" + (new Date).getTime();

this.getData = function( elem ) {
  var id = elem[ expando ];

  if ( !id ) {
    id = elem[ expando ] = guid++;
    cache[ id ] = {};
  }

  return cache[ id ];
};

this.removeData = function( elem ) {
  var id = elem[ expando ];

  if ( !id ) {
    return;
  }

  // Remove all stored data
  delete cache[ id ];

  // Remove the expando property from the DOM node
  try {
    delete elem[ expando ];
  } catch( e ) {
    if ( elem.removeAttribute ) {
      elem.removeAttribute( expando );
    }
  }
};
```

```
})();
```

The two generic functions, `getData` and `setData`, in Listing 14.3 are actually quite useful beyond the scope of managing event handlers. Using these functions you can attach all sorts of data to an object and have it be stored in a central location.

We can then use this data store to build from and create a new set of `addEvent` and `removeEvent` methods that work closer to our desired outcome in all browsers, as seen in Listing 14.4.

Listing 14.4: Combining the fix element method from Listing 14.2 and the data store from Listing 14.3 we can now build a new way to add and remove events.

```
<script>
(function(){
var guid = 1;

this.addEvent = function( elem, type, fn ) {
  var data = getData( elem ), handlers;

  // We only need to generate one handler per element
  if ( !data.handler ) {
    // Our new meta-handler that fixes
    // the event object and the context
    data.handler = function( event ) {
      event = fixEvent( event );

      var handlers = getData( elem ).events[ event.type ];

      // Go through and call all the real bound handlers
      for ( var i = 0, l = handlers.length; i < l; i++ ) {
        handlers[i].call( elem, event );

        // Stop executing handlers since the user requested it
        if ( event.isImmediatePropagationStopped() ) {
          break;
        }
      }
    };
  }

  // We need a place to store all our event data
  if ( !data.events ) {
    data.events = {};
  }

  // And a place to store the handlers for this event type
  handlers = data.events[ type ];

  if ( !handlers ) {
    handlers = data.events[ type ] = [];

    // Attach our meta-handler to the element,
    // since one doesn't exist
```

```
    if ( document.addEventListener ) {
      elem.addEventListener( type, data.handler, false );

    } else if ( document.attachEvent ) {
      elem.attachEvent( "on" + type, data.handler );
    }
  }

  if ( !fn.guid ) {
    fn.guid = guid++;
  }

  handlers.push( fn );
};

this.removeEvent = function( elem, type, fn ) {
  var data = getData( elem ), handlers;

  // If no events exist, nothing to unbind
  if ( !data.events ) {
    return
  }

  // Are we removing all bound events?
  if ( !type ) {
    for ( type in data.events ) {
      cleanUpEvents( elem, type );
    }

    return;
  }

  // And a place to store the handlers for this event type
  handlers = data.events[ type ];

  // If no handlers exist, nothing to unbind
  if ( !handlers ) {
    return;
  }

  // See if we're only removing a single handler
  if ( fn && fn.guid ) {
    for ( var i = 0; i < handlers.length; i++ ) {
      // We found a match
      // (don't stop here, there could be a couple bound)
      if ( handlers[i].guid === fn.guid ) {
        // Remove the handler from the array of handlers
        handlers.splice( i--, 1 );
      }
    }
  }

  cleanUpEvents( elem, type );
};

// A simple method for changing the context of a function
```

```
this.proxy = function( context, fn ) {
  // Make sure the function has a unique ID
  if ( !fn.guid ) {
    fn.guid = guid++;
  }

  // Create the new function that changes the context
  var ret = function() {
    return fn.apply( context, arguments );
  };

  // Give the new function the same ID
  // (so that they are equivalent and can be easily removed)
  ret.guid = fn.guid;

  return ret;
};

function cleanUpEvents( elem, type ) {
  var data = getData( elem );

  // Remove the events of a particular type if there are none left
  if ( data.events[ type ].length === 0 ) {
    delete data.events[ type ];

    // Remove the meta-handler from the element
    if ( document.removeEventListener ) {
      elem.removeEventListener( type, data.handler, false );

    } else if ( document.detachEvent ) {
      elem.detachEvent( "on" + type, data.handler );
    }
  }

  // Remove the events object if there are no types left
  if ( isEmpty( data.events ) ) {
    delete data.events;
    delete data.handler;
  }

  // Finally remove the expando if there is no data left
  if ( isEmpty( data ) ) {
    removeData( elem );
  }
}

function isEmpty( object ) {
  for ( var prop in object ) {
    return false;
  }

  return true;
}
})();

addEvent( window, "load", function() {
```

```
  var li = document.getElementsByTagName("li");
  for ( var i = 0; i < li.length; i++ ) (function(elem){
    addEvent( elem, "click", function handler(e) {
  // Only do this on left click
  if ( e.button === 1 ) {
        this.style.backgroundColor = "green";
        removeEvent( elem, "click", handler );
  }
    });
  })(li[i]);
});
</script>
<ul>
<li>Click</li>
<li>me</li>
<li>once.</li>
</ul>
```

There is a lot of code in Listing 14.4 but none of it is particularly challenging. Let's go through the logic function by function.

14.2.2 addEvent

We're now storing all the incoming handlers in the central data store (the event types are within the "events" property and each event type stores an array of all the handlers that are bound) instead of binding the handler directly to the element. We use an array because it allows us to bind multiple copies of the same handler and ensure that the handlers will be executed in the same order every time they're run. If we wish to implement a cross-browser version of the W3C DOM method `stopImmediatePropagation` we can easily do that with all these handlers stored in this way.

Since we still have to bind something to the element in order to listen for the event to occur we instead bind a single, generic, handler. We only need to generate one of these handlers per element (since we capture the element via a closure which then use to look up the event data and correct the handler contexts).

While the handler comes in we also give it a unique ID. This isn't explicitly required in order to be able to remove the handler later (we could always just compare the handler directly with what's stored in the data store) but it does give us the ability to create a nice proxy function for changing the context of the handler. The proxy function is similar to the context changing functions that we looked at in previous chapters but with an important distinction: You can bind a handler run through the proxy handler but still unbind the original handler. This can be best seen in Listing 14.5.

Listing 14.5: How to use the proxy function to count the number of clicks.

```
<script>
var obj = {
  count: 0,
  method: function(){
    this.count++;
```

```
  }
};

addEvent( window, "load", function() {
  // Attach the handler while enforcing the context to 'obj'
  addEvent( document.body, "click", proxy( obj, obj.method ) );

  // Remove on right click
  addEvent( document.body, "click", function(e) {
    if ( e.button === 3 ) {
      // Note that we can remove the obj.method handler no problem
      removeEvent( document.body, "click", obj.method );
    }
  });
});
</script>
```

14.2.3 removeEvent

The `removeEvent` function is much more advanced than the simple one that we built previously. Not only is it capable of removing a single event handler (even one bound using the proxy function) but it can also remove all handlers of a particular type or even all events bound to the element.

```
// Remove all bound events from the body
removeEvent( document.body );

// Remove all click events from the body
removeEvent( document.body, "click" );

// Remove a single function from the body
removeEvent( document.body, "click", someFn );
```

The important aspect behind `removeEvent` is in making sure that we clean up all our event handlers, data store, and meta-handler properly. Most of the logic for handling that is contained within the `cleanUpEvents` function. This function makes sure to systematically go through the element's data store and clean up any loose ends - eventually removing the meta-handler from the element itself if all handlers of a particular type have been removed from the element.

14.3 Triggering Events

Normally events occur when a user action triggers them (such as clicking or moving the mouse). However there are many cases where it's appropriate to simulate a native event occurring (even more so when working with custom events).

In a triggering function there are a few things that we want to have happen. We want to trigger the bound handlers on the element that we target, we want the event to bubble up the tree triggering any other bound handlers, and finally we want the native event to be

triggered on the target element (where available). A function that handles all of these cases can be seen in Listing 14.6.

Listing 14.6: Trigger a bubbling event on an element.

```
function triggerEvent( elem, event ) {
  var handler = getData( elem ).handler,
    parent = elem.parentNode || elem.ownerDocument;

  if ( typeof event === "string" ) {
    event = { type: event, target: elem };
  }

  if ( handler ) {
    handler.call( elem, event );
  }

  // Bubble the event up the tree to the document,
  // Unless it's been explicitly stopped
  if ( parent && !event.isPropagationStopped() ) {
    triggerEvent( parent, event );

  // We're at the top document so trigger the default action
  } else if ( !parent && !event.isDefaultPrevented() ) {
    var targetData = getData( event.target ),
      targetHandler = targetData.handler;

    if ( event.target[ type ] ) {
      // Temporarily disable the bound handler,
      // don't want to execute it twice
      if ( targetHandler ) {
        targetData.handler = function(){};
      }

      // Trigger the native event (click, focus, blur)
      event.target[ type ]();

      // Restore the handler
      if ( targetHandler ) {
        targetData.handler = targetHandler;
      }
    }
  }
}
```

The `triggerEvent` function in Listing 14.6 takes two parameters: The element on which the event will be triggered and the event that is being executed (which can be either an event object or just an event type, from which a simple event object is generated).

To trigger the event all we have to do is walk from the initial starting DOM node, find the meta-handler bound to it (if there is one), execute it and pass in the event object. All the rest of the handler execution is taken care of by the code that was originally in `addEvent` and `fixEvent`. We continue to walk up the tree by looking for new `parentNode` properties

until we reach the top of the tree and move to the document. Once the document execution has completed we can now execute the default browser action.

Note that during the event bubbling up the tree we make sure that propagation hasn't been stopped. Since we're doing our own triggering and bubbling it's important that we also simulate those actions. Additionally we make sure that we don't execute the default browser action if that has also been prevented.

To trigger the default browser action we use the appropriate method on the original target element. For example if we triggered a focus event we check to see if the original target element has a `.focus()` method and if so, execute it. We need to make sure that when that happens we don't execute the handlers again (we already did that once the first time through - all we care about this time is the execute of the default browser action).

Probably the best part of all of this code though is that it implicitly allows custom events to just work.

14.3.1 Custom Events

With all the work that's gone in to integrating cross-browser events, thus far, supporting custom events ends up being relatively simple. Custom events are a way of simulating the experience (to the end user) of a real event without having to use the browser's underlying event structure. Custom events can be an effective means of indirectly communicating actions to elements in a one-to-many way.

With the code that we've written for `addEvent`, `removeEvent`, and `triggerEvent` - nothing has to change in order to support custom events. Functionally there is no difference between an event that does exist and will be fired by the browser and an event that doesn't exist and will only fire when triggered manually. You can see an example of triggering a custom event in Listing 14.7.

Listing 14.7: Triggering a custom event across a number of elements.

```
<script>
addEvent( window, "load", function() {
  var li = document.getElementsByTagName("li");

  for ( var i = 0; i < li.length; i++ ) {
    addEvent( li[i], "update", function() {
      this.innerHTML = parseInt(this.innerHTML) + 1;
    });
  }

  var input = document.getElementsByTagName("input")[0];

  addEvent( input, "click", function() {
    var li = document.getElementsByTagName("li");

    for ( var i = 0; i < li.length; i++ ) {
      triggerEvent( li[i], "update" );
    }
  });
```

```
});
</script>
<input type="button" value="Add 1"/>
<ul>
  <li>1</li>
  <li>2</li>
  <li>3</li>
</ul>
```

The important aspect behind custom events, in contrast with just using regular functions, is that you're associating the functionality directly with the elements themselves and in a one-to-many fashion. In Listing 14.7 there is one update handler bound to each list item but that doesn't have to be the case, there could be any number of update handlers bound to the item and it wouldn't make any difference, from an execution and triggering perspective. This one-to-many relationship is the fundamental advantage behind using custom events in your code and should allow you to develop applications in a much more expressive and flexible manner.

14.4 Bubbling and Delegation

Event delegation is one of the best techniques available for developing high performance, scalable, web applications. Traditionally one would bind event handlers directly to the elements that they wish to watch - with event delegation you bind to an ancestor element (such as a table or the entire document) and watch for all events that come in. In this way you only have to bind one event handler for a large number of elements. This technique makes good use of the event bubbling provided by the browser. Since the events will bubble up the tree we can simply bind to that once ancestor element and wait for the events to come in.

Since event bubbling is the only technique available across all browsers (event capturing only works in W3C compatible browsers) it's important that all events consistently bubble or work in a way that the user expects. Unfortunately the `submit`, `change`, `focus`, and `blur` events all have serious problems with their bubbling implementations, in various browsers, and must be worked around.

To start, the `submit` and `change` events don't bubble at all in Internet Explorer (unlike in the W3C DOM-capable browsers where they bubble consistently). Presumably, however, they would implement support for those events at some point so we need to use a technique that is capable of gracefully determining if an event is capable of bubbling up to a parent element.

One such piece of detection code was written by Juriy Zaytsev and can be seen in Listing 14.8.

- Event detection code: http://perfectionkills.com/detecting-event-support-without-browser-sniffing/

Listing 14.8: Event bubbling detection code originally written by Juriy Zaytsev.

```
function isEventSupported(eventName) {
  var el = document.createElement('div'), isSupported;

  eventName = 'on' + eventName;
  isSupported = (eventName in el);

  if ( !isSupported ) {
    el.setAttribute(eventName, 'return;');
    isSupported = typeof el[eventName] == 'function';
  }

  el = null;
  return isSupported;
}
```

The event detection technique works by checking to see if an existing ontype (where type is the name of the event) property exists on a div. We check a div explicitly because divs typically have more events bubbled up to them (such as `change` and `submit`). If the ontype property doesn't exist then we create the ontype attribute, putting in a bit of code, and check to see if the element knows how to translate that into a function. If it does then it's a pretty good indicator that it knows how to interpret that particular event on bubble.

We can now use this detection code as the basis for further implementing properly-working events across all browsers.

14.4.1 submit

The `submit` event is one of the few that doesn't bubble in Internet Explorer. Thankfully though it is one of the easiest events to simulate. A `submit` event can be triggered in a one of two ways:

- Clicking (or focusing on and hitting a trigger key, like enter or spacebar) a submit or image submit button.
- Pressing enter while inside a text or password input.

Knowing these two cases we can bind to the two appropriate events (click and keypress - both of which bubble normally) to achieve our desired result, as seen in Listing 14.9.

Listing 14.9: An implementation of a cross-browser, bubbling, `submit` event.

```
(function(){
  // Check to see if the submit event works as we expect it to
  if ( !isEventSupported("submit") ) {
    this.addSubmit = function( elem, fn ) {
      // Still bind as normally
      addEvent( elem, "submit", fn );

      // But we need to add extra handlers if we're not on a form
```

```
      // Only add the handlers for the first handler bound
      if ( elem.nodeName.toLowerCase() !== "form" &&
           getData( elem ).events.submit.length === 1 ) {

        addEvent( elem, "click", submitClick );
        addEvent( elem, "keypress", submitKeypress );
      }
    };

    this.removeSubmit = function( elem, fn ) {
      removeEvent( elem, "submit", fn );

      var data = getData( elem );

      // Only remove the handlers when there's nothing left to remove
      if ( elem.nodeName.toLowerCase() !== "form" &&
           !data || !data.events || !data.events.submit ) {

        addEvent( elem, "click", submitClick );
        addEvent( elem, "keypress", submitKeypress );
      }
    };

  // Otherwise the event works perfectly fine
  // so we just behave normally
  } else {
    this.addSubmit = function( elem, fn ) {
      addEvent( elem, "submit", fn );
    };

    this.removeSubmit = function( elem, fn ) {
      removeEvent( elem, "submit", fn );
    };
  }

  // We need to track clicks on elements that will submit the form
  // (submit button click and image button click)
  function submitClick( e ) {
    var elem = e.target, type = elem.type;

    if ( (type === "submit" || type === "image") &&
         isInForm( elem ) ) {
      return triggerEvent( this, "submit" );
    }
  }

  // Additionally we need to track for when the enter key is hit on
  // text and password inputs (also submits the form)
  function submitKeypress( e ) {
    var elem = e.target, type = elem.type;

    if ( (type === "text" || type === "password") &&
         isInForm( elem ) && e.keyCode === 13 ) {
      return triggerEvent( this, "submit" );
    }
  }
```

```
    // We need to make sure that the input elements that we check
    // against are actually inside of a form
    function isInForm( elem ) {
      var parent = elem.parentNode;

      while ( parent ) {
        if ( parent.nodeName.toLowerCase() === "form" ) {
          return true;
        }

        parent = parent.parentNode;
      }

      return false;
    }
  })();
```

In Listing 14.9 we can see that we don't need to bind a `submit` event to a form element as that'll continue to work as we would expect it to. Instead we should only bind our special handlers in the case where we're not binding on a form, likely farther up the document tree. Note that we continue to bind to the submit event even though it won't have any immediate effect - we use this as a means to easily store our handlers for later triggering.

Inside the `submitClick` function we verify that the user is in fact clicking on a submit or image input (both of which can submit the form). Additionally we verify that those inputs are actually inside a form (it's theoretically possible that someone might have these inputs outside a form, in which case there obviously isn't a form submission occurring).

Inside `submitKeypress` we first verify that we're working against either a text or password input and that the input is inside a form (like with the click verification). We have the additional check to make sure that the keyCode is equal to 13 (the enter key), which would trigger a `submit` event to occur.

Note that in the `addSubmit` and `removeSubmit` functions we're careful to only bind these additional handlers once and only remove them after all of the submit events have been unbound from the element. Binding them more than once would cause too many events to fire and unbinding them too early would break future submits.

All of this logic though is rather quite generic and tends to apply well to fixing other DOM bubbling events, such as the `change` event.

14.4.2 change

The `change` event is another event that doesn't bubble properly in Internet Explorer. Unfortunately it's significantly harder to implement properly, compared to the `submit` event. In order to implement the bubbling change event we must bind to a number of different events.

- The `focusout` event for checking the value after moving away from the form

element.

- The `click` and `keydown` event for checking the value the instant it's changed.
- The `beforeactivate` for getting the previous value before a new one is set.

Listing 14.10 has a full implementation of the change that uses all of the above events to create the final result in all browsers.

Listing 14.10: An implementation of a cross-browser bubbling change event.

```
(function(){
  // We want to simulate change events on these elements
  var formElems = /textarea|input|select/i;

  // Check to see if the submit event works as we expect it to
  if ( !isEventSupported("change") ) {
    this.addChange = function( elem, fn ) {
      addEvent( elem, "change", fn );

      // Only add the handlers for the first handler bound
      if ( getData( elem ).events.change.length === 1 ) {
        addEvent( elem, "focusout", testChange );
        addEvent( elem, "click", changeClick );
        addEvent( elem, "keydown", changeKeydown );
        addEvent( elem, "beforceactivate", changeBefore );
      }
    };

    this.removeChange = function( elem, fn ) {
      removeEvent( elem, "change", fn );

      var data = getData( elem );

      // Only remove the handlers when there's
      // nothing left to remove
      if ( !data || !data.events || !data.events.submit ) {
        addEvent( elem, "focusout", testChange );
        addEvent( elem, "click", changeClick );
        addEvent( elem, "keydown", changeKeydown );
        addEvent( elem, "beforceactivate", changeBefore );
      }
    };

  // The change event works just fine
  } else {
    this.addChange = function( elem, fn ) {
      addEvent( elem, "change", fn );
    };

    this.removeChange = function( elem, fn ) {
      removeEvent( elem, "change", fn );
    };
  }
```

```
// Check to see that the clickable inputs are updated
function changeClick( e ) {
  var elem = e.target, type = elem.type;

  if ( type === "radio" || type === "checkbox" ||
       elem.nodeName.toLowerCase() === "select" ) {
    return testChange.call( this, e );
  }
}

// Change has to be called before submit
// Keydown will be called before keypress,
// which is used in submit-event delegation
function changeKeydown( e ) {
  var elem = e.target, type = elem.type, key = e.keyCode;

  if ( key === 13 && elem.nodeName.toLowerCase() !== "textarea" ||
       key === 32 && (type === "checkbox" || type === "radio") ||
       type === "select-multiple" ) {
    return testChange.call( this, e );
  }
}

// Beforeactivate happens before the previous element is blurred
// Use it to store information for later checking
function changeBefore( e ) {
  var elem = e.target;
  getData( elem )._change_data = getVal( elem );
}

// Get a string value back for a form element
// that we can use to verify if a change has occurred
function getVal( elem ) {
  var type = elem.type, val = elem.value;

  // Checkboxes and radios only change the checked state
  if ( type === "radio" || type === "checkbox" ) {
    val = elem.checked;

  // Multiple selects need to check all selected options
  } else if ( type === "select-multiple" ) {
    val = "";

    if ( elem.selectedIndex > -1 ) {
      for ( var i = 0; i < elem.options.length; i++ ) {
        val += "-" + elem.options[i].selected;
      }
    }

  // Regular selects only need to check what
  // option is currently selected
  } else if ( elem.nodeName.toLowerCase() === "select" ) {
    val = elem.selectedIndex;
  }

  return val;
```

```
    }

    // Check to see if a change in the value has occurred
    function testChange( e ) {
      var elem = e.target, data, val;

      // Don't need to check on certain elements and read-only inputs
      if ( !formElems.test( elem.nodeName ) || elem.readOnly ) {
        return;
      }

      // Get the previously-set value
      data = getData( elem )._change_data;
      val = getVal( elem );

      // the current data will be also retrieved by beforeactivate
      if ( e.type !== "focusout" || elem.type !== "radio" ) {
        getData( elem )._change_data = val;
      }

      // If there's been no change then we can bail
      if ( data === undefined || val === data ) {
        return;
      }

      // Otherwise the change event should be fired
      if ( data != null || val ) {
        return triggerEvent( elem, "change" );
      }
    }
  })();
```

As you can probably tell by the code in Listing 14.10 we receive virtually no help from Internet Explorer towards having a workable `change` solution. Instead we must implement all aspects of the change events: Serializing the old values, comparing the old values against the new values, and watching for potential change triggers to occur (`click`, `keydown`, and `focusout`).

The core of the functionality is encapsulated within the `getVal` and `testChange` functions. `getVal` is designed to return a simple serialized version of the state of a particular form element. This value will be stored in the `_change_data` property within the element's data object for later use. The `testChange` function is primarily responsible for determining if an actual change has occurred between the previously-stored value and the newly-set value and triggering a real change event if the value is now different.

In addition to checking to see if a change has occurred after a `focusout` (blur) we also check to see if the enter key was hit on something that wasn't a textarea or hitting the spacebar on a checkbox or radio button. We also check to see if a click occurred on a checkbox, radio, or select as that will also trigger a change to occur.

All told there's a lot of code for something that should've been tackled natively by the browser. It'll be greatly appreciated when the day comes that this code doesn't need to exist.

FOCUSIN AND FOCUSOUT

`focusin` and `focusout` are two custom events introduced by Internet Explorer that detect when a `focus` or `blur` has occurred on any element, or element descendant. It actually occurs before the focus or blur take place (making it equivalent to a capturing event rather than a bubbling event).

The reason for even considering these two events is that the `focus` and `blur` events do not bubble (as dictated by the W3C DOM recommendation and as implemented by all browsers). It ends up being far easier to implement `focusin` and `focusout` clones across all browsers then trying to circumvent the intentions of the browsers (whatever that may be) and getting the events to bubble.

The best way to implement the `focusin` and `focusout` events would be to modify the existing `addEvent` function to handle the event types inline.

```
// Attach our meta-handler to the element, since once doesn't exist
if ( document.addEventListener ) {
  elem.addEventListener(
    type === "focusin" ? "focus" :
      type === "focusout" ? "blur" : type,
    data.handler, type === "focusin" || type === "focusout" );

} else if ( document.attachEvent ) {
  elem.attachEvent( "on" + type, data.handler );
}
```

and then modify the `removeEvent` function to unbind the events again properly:

```
// Remove the meta-handler from the element
if ( document.removeEventListener ) {
  elem.removeEventListener(
    type === "focusin" ? "focus" :
      type === "focusout" ? "blur" : type,
    data.handler, type === "focusin" || type === "focusout" );

} else if ( document.detachEvent ) {
  elem.detachEvent( "on" + type, data.handler );
}
```

The end will result will be native support for the `focusin` and `focusout` event in all browsers. Naturally you might want to keep your event-specific logic separate from your `addEvent` and `removeEvent` internals, in that case you could implement some form of extensibility for overriding the native binding/unbinding mechanisms provided by the browser, for specific event types.

Some more information about cross-browser `focus` and `blur` events can be found here:

- http://www.quirksmode.org/blog/archives/2008/04/delegating_the.html

14.4.3 mouseenter and mouseleave

mouseenter and mouseleave are two custom events introduced by Internet Explorer to simplify the process of determining when the mouse is currently positioned within an element, or outside of it. Traditionally one would interact with the mouseover and mouseout events provided by the browser but they don't provide what users are typically looking for: They fire the event when you move between child elements in addition to on the element itself. This is typical of the event bubbling model but it is frequently more than what a user is looking for (they simply want to make sure that they're still within the element and only car about when the element is left).

This is where the `mouseenter` and `mouseleave` events come in handy. They will only fire on the main element on which you bind them and only inform you as to if the mouse is currently inside or outside of the element. Since Internet Explorer is the only browser that currently implments these useful events we need to simulate the full event interaction for other browsers as well.

Listing 14.11 shows some code to implement the `mouseenter` and `mouseleave` events across all browsers.

Listing 14.11: An example of implementing a hover event using the event detection code from Listing 14.8.

```
<div>Hover <strong>over</strong> me!</div>
<style>.over { background: yellow; }</style>
<script>
(function(){
  if ( isEventSupported( "mouseenter" ) ) {
    this.hover = function( elem, fn ) {
      addEvent( elem, "mouseenter", function(){
        fn.call( elem, "mouseenter" );
      });

      addEvent( elem, "mouseleave", function(){
        fn.call( elem, "mouseleave" );
      });
    };

  } else {
    this.hover = function( elem, fn ) {
      addEvent( elem, "mouseover", function(e){
        withinElement( this, e, "mouseenter", fn );
      });

      addEvent( elem, "mouseout", function(e){
        withinElement( this, e, "mouseleave", fn );
      });
    };
```

```
  }

  function withinElement( elem, event, type, handle ) {
    // Check if mouse(over|out) are still
    // within the same parent element
    var parent = event.relatedTarget;

    // Traverse up the tree
    while ( parent && parent != elem ) {
      // Firefox sometimes assigns relatedTarget a XUL element
      // which we cannot access the parentNode property of
      try {
        parent = parent.parentNode;

      // assuming we've left the element since
      // we most likely mousedover a xul element
      } catch(e) { break; }
    }

    if ( parent != elem ) {
      // handle event if we actually just
      // moused on to a non sub-element
      handle.call( elem, type );
    }
  }
})();

window.onload = function(){
  var div = document.getElementsByTagName("div")[0];

  hover( div, function(type){
    if ( type === "mouseenter" ) {
      this.className = "over";
    } else {
      this.className = "";
    }
  });
};
</script>
```

The majority of the logic for handling the `mouseenter` and `mouseleave` events lies inside of the `withinElement` function. This function checks the `relatedTarget` of the event and makes sure that it's not contained within the original element (if it is contained within then we're just seeing `mouseover` and `mouseout` events from internal elements)

14.5 Document Ready

The final important event that needs to be covered is the "ready" event (implemented as `DOMContentLoaded` in W3C DOM-capable browsers). The ready event

fires as soon as the entire DOM document has been loaded and is able to be traversed and manipulated. This event has become an integral part of many modern frameworks, allowing code to be layered in an unobtrusive manner, frequently executing before the page is displayed.

An implementation of the ready event, using the above technique, can be found in Listing 14.12.

Listing 14.12: Implement a cross-browser DOM ready event.

```
(function(){
  var isReady = false, DOMContentLoaded;

  // Catch cases where addReady is called after the
  // browser event has already occurred.
  if ( document.readyState === "complete" ) {
    return ready();
  }

  // Mozilla, Opera and Webkit currently support this event
  if ( document.addEventListener ) {
    DOMContentLoaded = function() {
      document.removeEventListener(
        "DOMContentLoaded", DOMContentLoaded, false );
      ready();
    };

    // Use the handy event callback
    document.addEventListener(
      "DOMContentLoaded", DOMContentLoaded, false );

  // If IE event model is used
  } else if ( document.attachEvent ) {
    DOMContentLoaded = function() {
      if ( document.readyState === "complete" ) {
        document.detachEvent(
          "onreadystatechange", DOMContentLoaded );
        ready();
      }
    };

    // ensure firing before onload,
    // maybe late but safe also for iframes
    document.attachEvent(
      "onreadystatechange", DOMContentLoaded );

    // If IE and not a frame
    // continually check to see if the document is ready
    var toplevel = false;
```

```
    try {
      toplevel = window.frameElement == null;
    } catch(e) {}

    if ( document.documentElement.doScroll && toplevel ) {
      doScrollCheck();
    }
  }

  function ready() {
    if ( !isReady ) {
      triggerEvent( document, "ready" );
      isReady = true;
    }
  }

  // The DOM ready check for Internet Explorer
  function doScrollCheck() {
    if ( isReady ) {
      return;
    }

    try {
      // If IE is used, use the trick by Diego Perini
      // http://javascript.nwbox.com/IEContentLoaded/
      document.documentElement.doScroll("left");
    } catch( error ) {
      setTimeout( doScrollCheck, 1 );
      return;
    }

    // and execute any waiting functions
    ready();
  }
})();
```

The majority of the logic in Listing 14.12 is directly related to figuring out when the DOM is ready in Internet Explorer. Internet Explorer doesn't provide a native means of watching for when the document is finally ready, unlike other browsers. Instead we must resort to a technique originally developed by Diego Perini in which the `doScroll("left")` method is called on the `documentElement`. Calling this method will have effect of scrolling the viewport to the left side (which is the normal starting position anyway) - but more importantly, it will throw an exception if the document isn't ready to be manipulated yet. Thus we can use a timer to constantly loop and try to run the `doScroll` method in a try/catch block. Whenever an exception is no longer thrown then it is safe to continue. More information about this particular technique can be found on Diego's site:

- http://javascript.nwbox.com/IEContentLoaded/

It should be noted that the `doScroll` technique alone doesn't work inside iframe'd documents. As a fallback we use a secondary technique: Watching the `onreadystatechange` event on the document. This particular event is more inconsistent than the `doScroll` technique - while it'll always fire after the DOM is ready it'll sometimes fire quite a while after (but always before the final window load event). That being said it serves as a good backup for IE, making sure that the at least something will fire before the window load event.

One additional check this is performed, on load, is examining the `document.readyState` property. This property, available in all browsers, notes how fully-loaded the DOM document is at that point (note that long delays in loading, especially in Internet Explorer, may cause the `readyState` to report "complete" too early, hence why we use the `doScroll` technique instead). Checking this property on load can help to avoid unnecessary event binding if the DOM is already in a ready-to-use state (as might be the case if the script was dynamically loaded at a later time).

Note: While `document.readyState` is available in all browsers it was only added to Firefox in version 3.6. This may not matter, depending upon your application, but it's good to note.

With a complete ready event implementation we now have all the tools in place for a complete DOM event system.

14.6 Summary

As we can see, DOM event systems are anything but simple. Internet Explorer's divergent system ends up causing much of the overhead that we see and have to write. Even so the lack of extensibility in the native API means that we still have to circumvent, or improve upon, most of the event system in order to arrive at a solution that is more universally applicable.

In this chapter we learned the fundamentals behind adding and removing events from a DOM node and the hidden object store that's used to make it all possible. Additionally we looked at triggering events and working with custom events and how applications can benefit from their unique model. Finally we examined event bubbling and delegation and all the quirks that go along with it, working to normalize the bubbling of events across all browsers.

All told we now have the knowledge necessary to implement a complete, and useful, DOM event system that is capable of tackling even the greatest challenge presented to us by the browser.