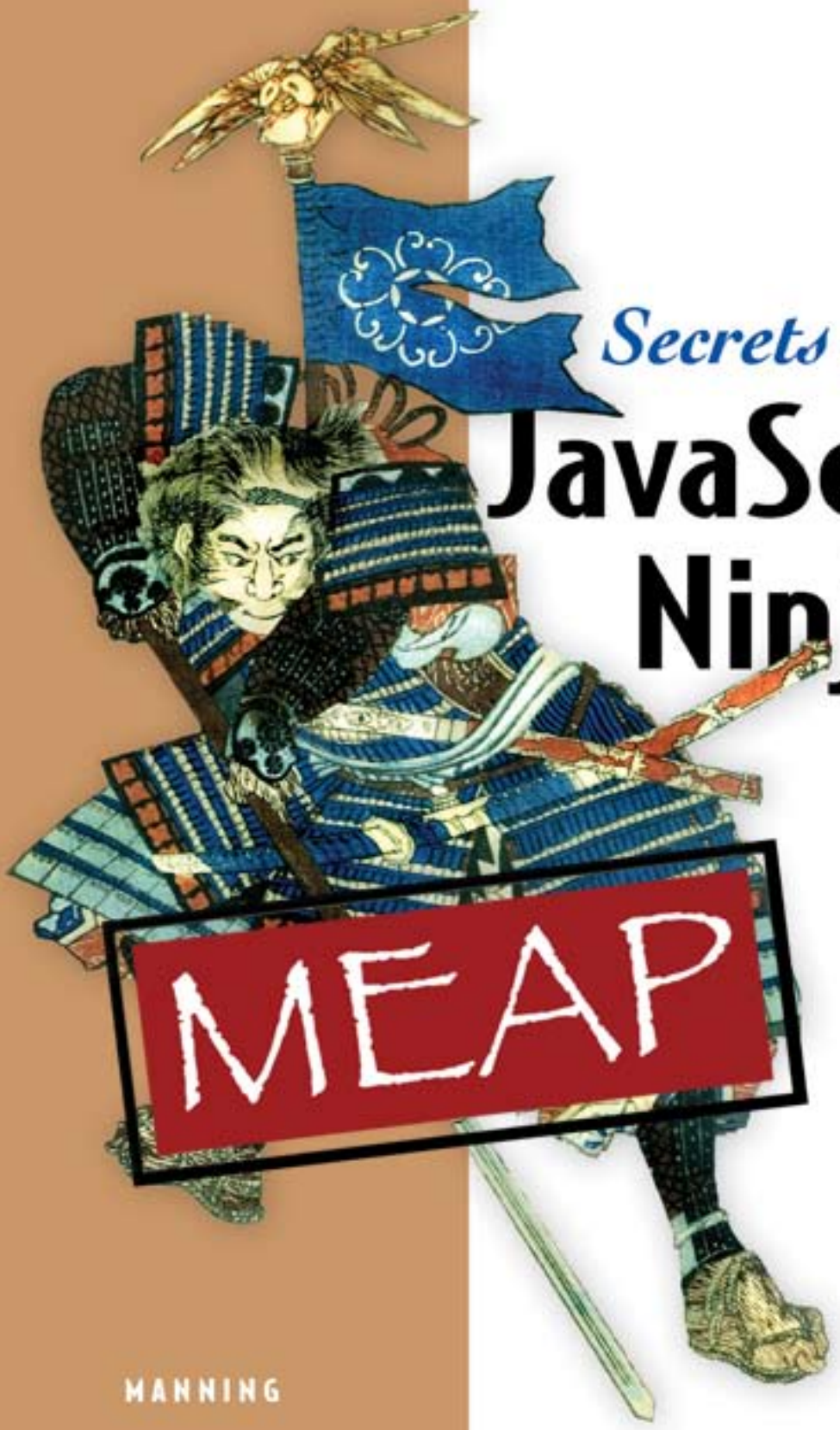

Secrets of the JavaScript Ninja

MEAP

John Resig
Bear Bibeault

MANNING

MEAP Edition
Manning Early Access Program
Secrets of the JavaScript Ninja version 10

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

brief 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:

- 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.
- 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.

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.

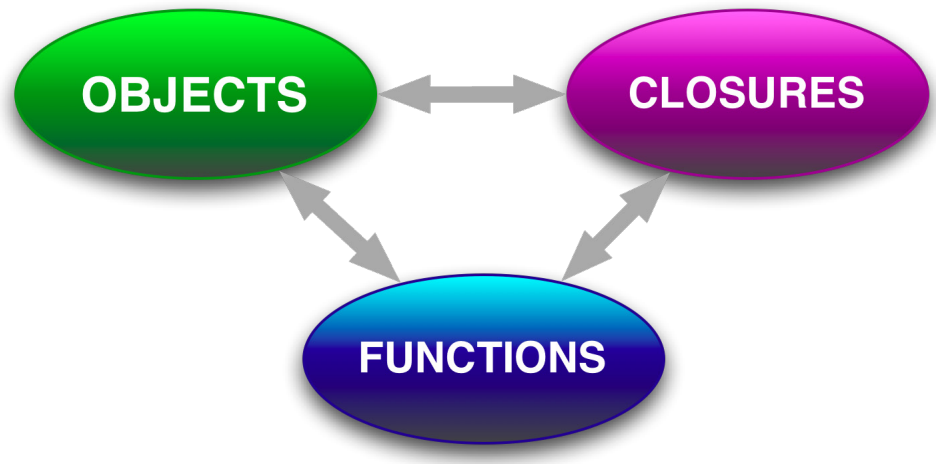

Figure 1.1 JavaScript consists of a close relationship between objects, functions and closures

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 10, 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:

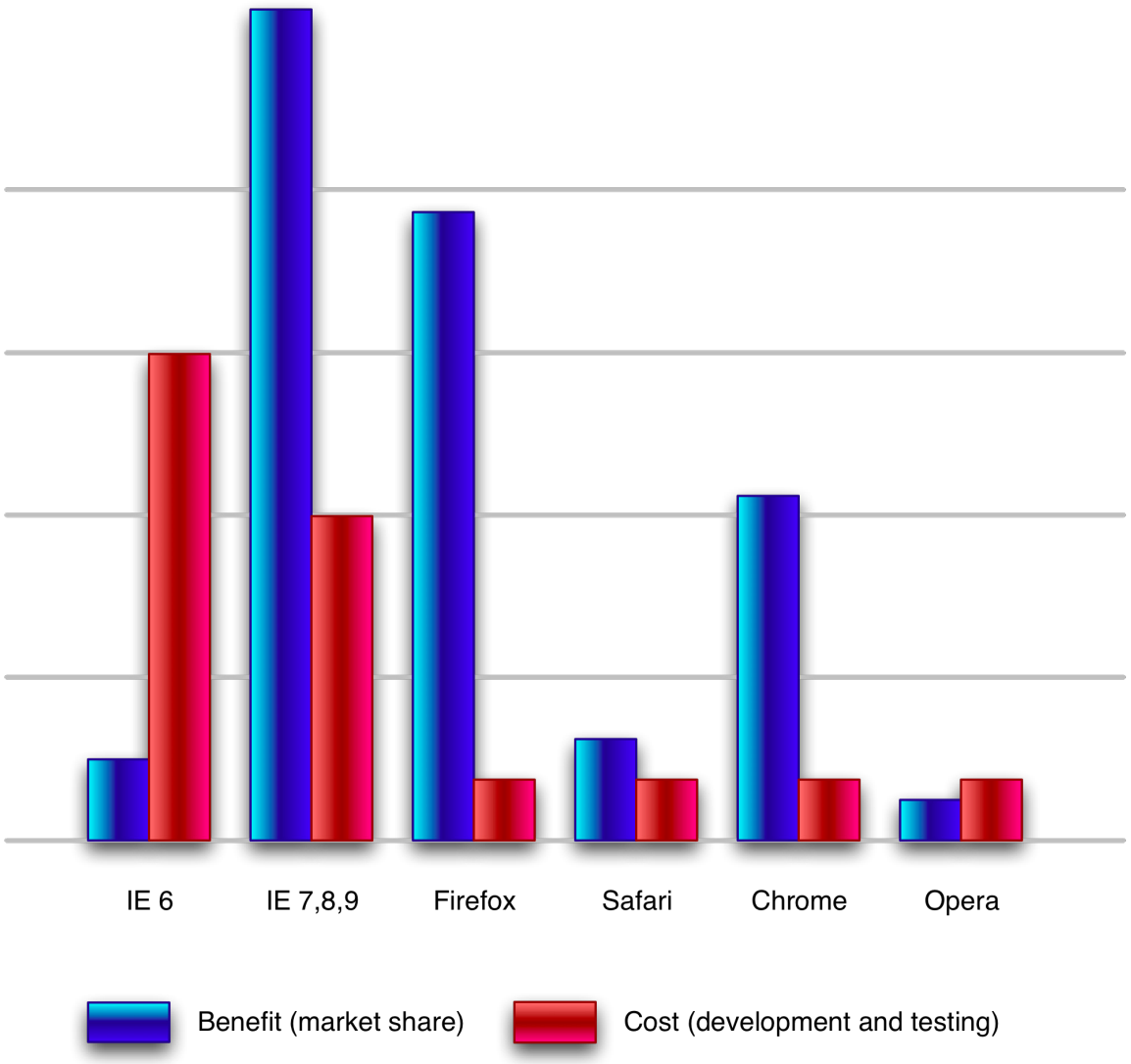

Figure 1.2 Analyzing the cost versus benefit of supporting various browsers tells us where to put our effort

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 no 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 languages, 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;
  }
});
```

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 11, 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.

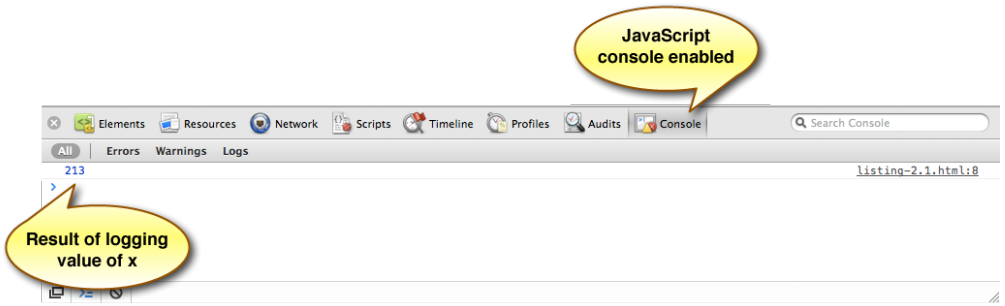

Figure 2.1 Logging lets us see the state of things in our code as it is running

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.

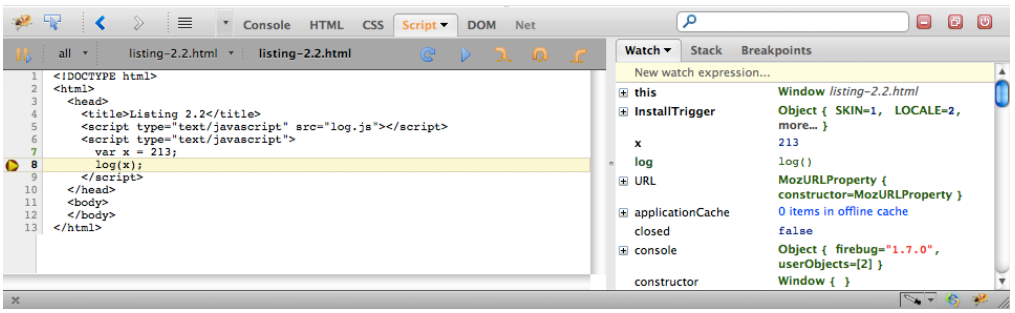

Figure 2.2 Breakpoints allow us to halt execution at a specific line of code so we can take a gander at the state

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.

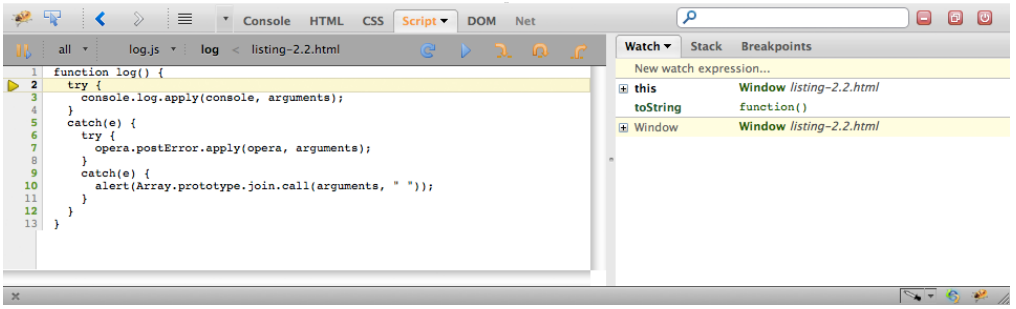

Figure 2.3 Stepping into our method lets us see the new state within which the method executes

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.

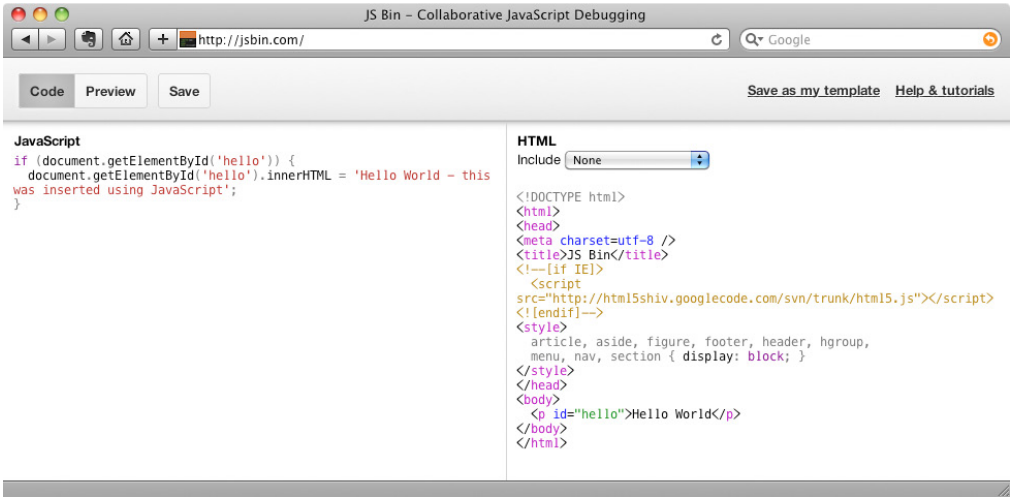

Figure 2.4: A screenshot of the JSBin web site in action

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.

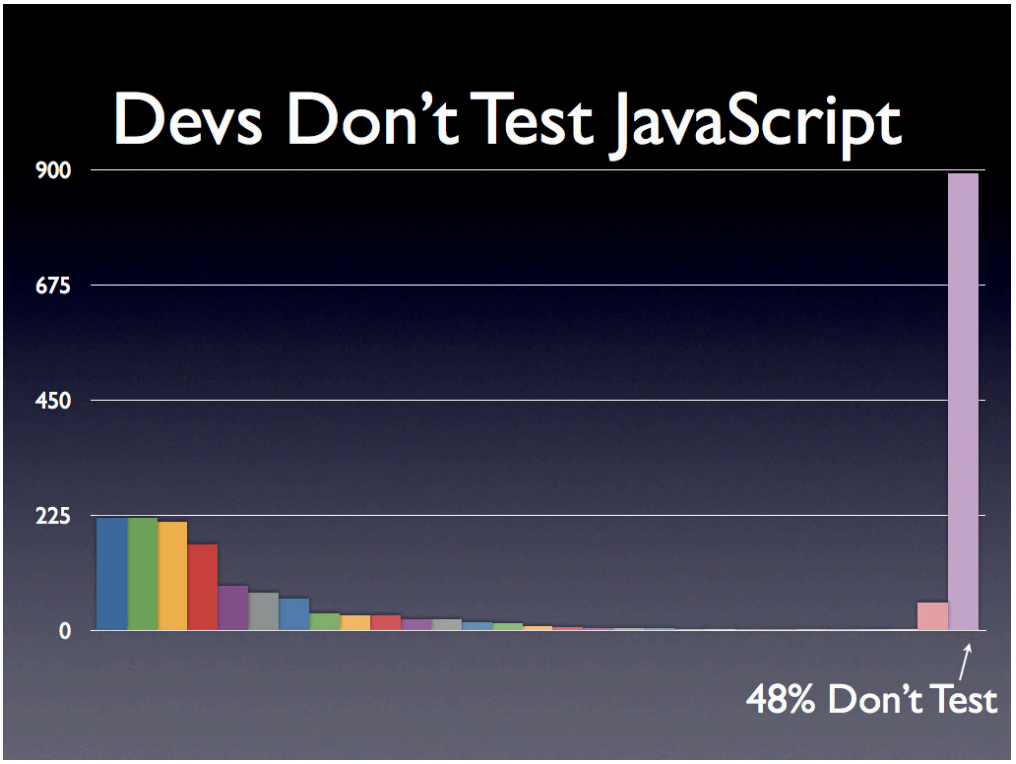

Figure 2.5 A dishearteningly large percentage of script developers don't test at all

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.

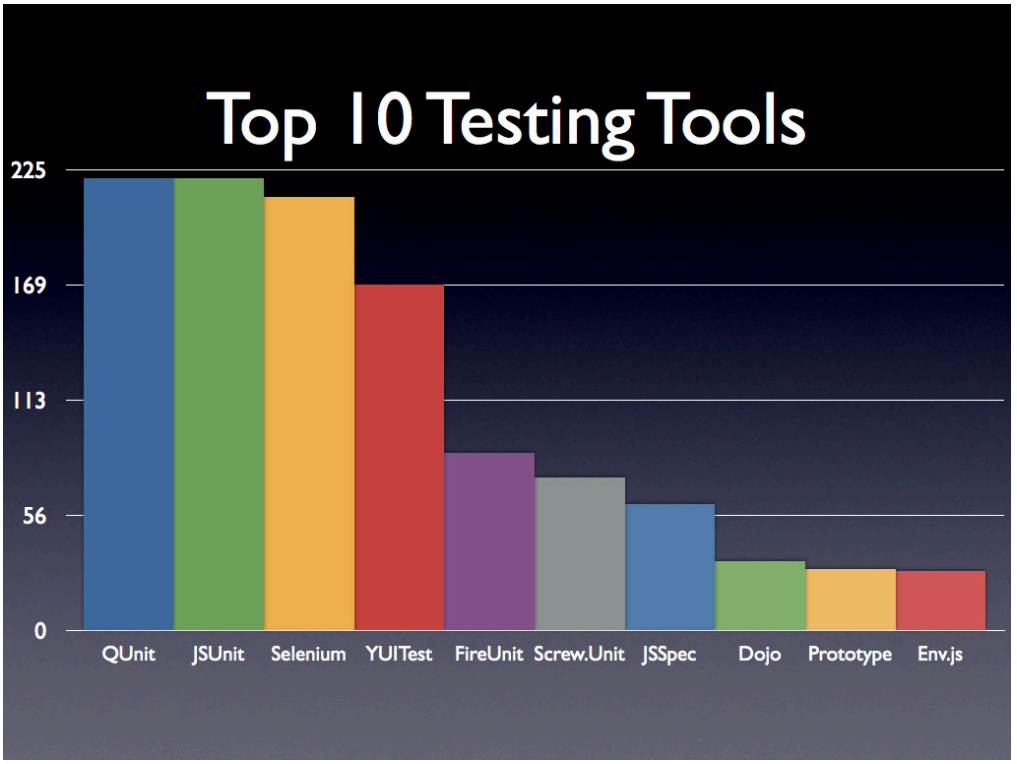

Figure 2.6 Most test-savvy developers favor a small handful of testing tools

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.

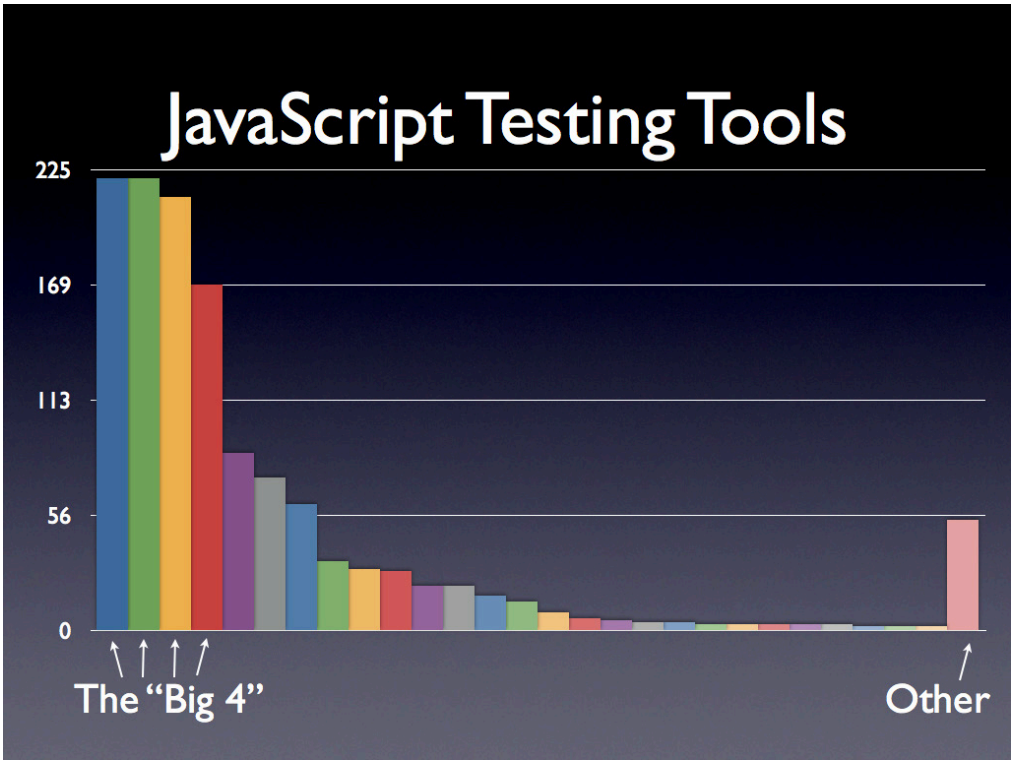

Figure 2.7 The remainder of the testing tools have few users

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 8 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
- How the browser invokes function
- Declaring functions
- The secrets of how functions are invoked
- The context within a function

You might have been somewhat surprised, upon turning to this first page of the part of this book dedicated to JavaScript fundamentals, to see that the topic of discussion is to be functions rather than objects.

We'll certainly be paying plenty of attention to objects (particularly in chapter 6), but when it comes down to brass tacks, the main difference between writing JavaScript code like the average Joe (or Jill) and writing it like a JavaScript Ninja, is understand JavaScript as a ***functional language***. The level of the sophistication of all the code that you will ever write in JavaScript hinges upon this realization.

If you're reading this book, you're not a rank beginner and we're assuming that you know enough object fundamentals to get by for now (and we'll be taking a look at more advanced object concepts in chapter 6), but *really* understanding functions in JavaScript is the single most important weapon we can wield. So important, in fact, that this and the following two chapters are going to be devoted to thoroughly understanding functions in JavaScript.

Most importantly, in JavaScript, functions are *first-class objects*; that is, they coexist with, and can be treated like, any other JavaScript object. Just like the more mundane JavaScript data types, they can be referenced by variables, declared with literals, and even passed as function parameters.

The fact that JavaScript treats functions as first-class objects is going to be important on a number of levels, but one significant advantage comes in the form of code terseness. To take a sneak-peek ahead to some code that we'll examine in greater depth in section 3.1.2, some imperative code (in Java) to perform a collection sort could be:

```
Arrays.sort(values,new Comparator<Integer>(){
  public int compare(Integer value1, Integer value2) {
    return value2 - value1;
  }
});
```

The JavaScript equivalent written using a functional approach:

```
values.sort(function(value1,value2){ return value2 - value1; });
```

Don't be too concerned if the notation seems odd – you'll be an old hand at it by the end of this chapter. We just wanted to give you a glimpse of one of the advantages that understanding JavaScript as a functional language will bring to the table.

This chapter will thoroughly examine JavaScript's focus on functions, and give us a sound basis on which to bring our JavaScript code to a level that any master would be proud of.

3.1 What's with the functional difference?

How many times have you heard someone moan "I hate JavaScript!"?

We're willing to bet that nine times out of ten (or perhaps even greater), this is a direct consequence of someone trying to use JavaScript as if it were another language that the lamenter is more familiar with, and frustrated by the fact that it's *not* that other language.

This is probably most common with those coming to JavaScript from a language such as Java, a decidedly non-functional language, but one that a lot of developers learn before their exposure to JavaScript.

Making matters even worse for these developers is the unfortunate naming choice of ***Java***Script. Without belaboring the history behind that lamentable naming decision, perhaps developers would have fewer incorrect preconceived notions about JavaScript if it had retained the name *LiveScript* or been given some other less confounding name.

Because JavaScript, as the old joke depicted in figure 3.1 goes, has as much to do with Java as a hamburger has to do with ham.

Figure 3.1 JavaScript is to Java as hamburger is to ham; both delicious, but not much in common except a name

Hamburgers and ham are both foods that are meat products, just as JavaScript and Java are both programming languages with a C-influenced syntax, but other than that, they don't have much in common, and are fundamentally different right down to their DNA.

NOTE

Another factor that plays into some developers' poor initial reaction to JavaScript may be that most developers are introduced to JavaScript in the browser. So rather than reacting to JavaScript, *The Language*, they may be recoiling from the JavaScript bindings to the DOM API. And the DOM API… well, let's just say that it isn't going to win any *Friendliest API of the Year* awards. But that's not JavaScript's fault.

Before we learn about how functions are such a central and key concept in JavaScript, let's understand *why* the functional nature of JavaScript is so important, especially for code written for the browser.

3.1.1 Why is JavaScript's functional nature important?

It's likely that if you've done any amount of scripting in a browser that you probably know all that we're going to discuss in this section, but let's go over it anyways just in case, and to make sure we're all in the same vernacular.

One of the reasons that functions and functional concepts are so important in JavaScript is that the function is the primary modular unit of execution. Except for inline script that runs while the markup is being evaluated, all of the script code that we'll write for our pages will be within a function.

NOTE

Back in the Dark Ages, inline script was used to add dynamicity to pages via `document.write()`. These days, `document.write()` is considered a dinosaur and its use is not recommended. There are better ways to make pages dynamic, be they the use of server-side templating, or client-side DOM manipulation (or a healthy combination of both).

Because most of our code will run as the result of a function invocation, we will see that having functions that are versatile and powerful constructs will give us a great deal of flexibility and sway when writing our code. We'll spend the rest of this chapter examining just how the nature of functions as first-class objects can be exploited to our great benefit.

Now, that's the *second* time that we've used the term "first class object", and it's an important concept, so before we go on, let's make sure we know what it really means.

FUNCTIONS AS FIRST-CLASS OBJECTS

Objects in JavaScript enjoy certain capabilities. They can be:

- created via literals
- assigned to variables, array entries, and properties of other objects
- passed as arguments to functions
- returned as values from functions
- possess properties that can be dynamically created and assigned

Functions in JavaScript possess all of these capabilities, and are thus treated like any other object. Therefore, we say that they are ***first-class*** objects, just like any other object in the language.

And more than just being treated with the same respect as other objects types, functions have a special capability in that they can be *invoked*. And that invocation is frequently discharged in an *asynchronous* manner.

Let's talk a little about why that is.

THE BROWSER EVENT LOOP

If you've done any programming to create GUI (graphical user interface) desktop applications, you'll know that most are written in a similar fashion:

- Set up the user interface
- Enter a loop waiting for events to occur
- Invoke handlers (also called *listeners*) for those events

Well, programming for the browser is no different *except* that *our* code is not responsible for running the event loop and dispatching events; the browser handles that for us.

Our responsibility is to set up the handlers for the various events that can occur in the browser. These events are placed in an event queue (a FIFO list, more on that later) as they occur, and the browser dispatches these events by invoking any handlers that have been established for them.

Because these events happen at unpredictable times and in an unpredictable order, we say that the handling of the events, and therefore the invocation of their handling functions, is ***asynchronous***.

The types of events that can occur include:

- Browser events, such as when a page is finished loading or when it is to be unloaded.
- Network events, such as responses to an Ajax request.
- User events, such as mouse clicks, mouse moves, or key presses.
- Timer events, such as when a timeout expires or an interval fires.

The vast majority of our code executes as a result of such events. Consider the following:

```
function startup(){
  /* do something wonderful */
}
window.onload = startup;
```

Here, we establish a function to serve as a handler for the `load` event. The establishing statement executes as part of the inline script (assuming it appears at top level and not within any other function), but the wonderful things that we're going to do *inside* the function don't execute until the browser finishes loading the page and fires off a `load` event.

In fact, we can simplify this to a single line if we like. Consider:

```
window.onload = function() { /* do something wonderful */ };
```

(If the notation used to create the function looks odd to you, be assured that we'll be making it crystal clear in section 3.2.)

Unobtrusive JavaScript

This approach of assigning a function, named or otherwise, to the `onload` property of the `window` instance may not be the way that you are used to setting up a load handler. You may be more accustomed to using the `onload` attribute of the `<body>` tag.

Either approach achieves the same effect, but the `window.onload` approach is vastly preferred by JavaScript ninjas as it adheres to a popular principle known as *Unobtrusive JavaScript*.

Remember when the advent of CSS pioneered moving style information out of the document markup? Few would argue that segregating style from structure was a bad move. Well, Unobtrusive JavaScript does the same thing for *behavior*; moving script out of the document markup.

This results in pages having their three primary components: structure, style and behavior, nicely partitioned into their own locations. Structure is defined in the document markup, style in `<style>` elements or external stylesheets, and behavior in `<script>` blocks or external script files.

You will not see any script embedded into document markup in the examples in this book unless it is to make a specific point or to vastly simplify the example.

It's important to note that the browser event loop is *single-threaded*. Every event that is placed into the event queue is handled in the order that it is placed onto the queue. This is known as a *FIFO list* (first-in, first-out), or perhaps a *silo* to the old-timers. Each event is processed in its own "turn" and all other events have to wait until the current event's turn is over. Under no circumstances are two handlers executing simultaneously in separate threads.

Think of a line at the bank. Everyone gets into a single line and has to wait their turn to be "processed" by the tellers. But with JavaScript, there's only *one* teller window open! So the customers only get processed one at a time, as their turn comes. All it takes is one person, who thinks it's appropriate to do their financial planning for the fiscal year while they are at the teller's window (we've all run into them!), to gum up the whole works.

This execution model, and ways of dealing with its challenges, is one that we'll explore in great depth in chapter 8.

A vastly simplified overview of this process is shown in figure 3.2.

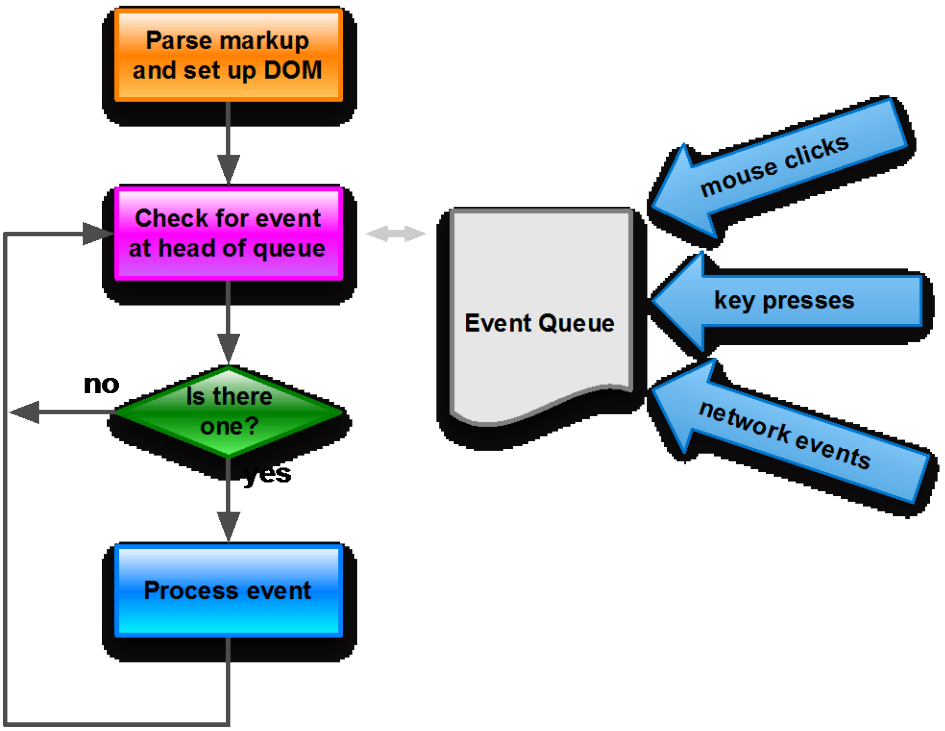

Figure 3.2 A simplified view of how the browsers process the event loop, handling each event in its own turn within a single thread

This concept is central to on-page JavaScript and is something we'll see again and again throughout the examples of this book: code is set up in advance in order to execute at a later time. Except for in-line setup code, the vast majority of the code that we place onto a page is going to execute as the result of an event (in other words as part of the "Process event" box).

It's important to note that whatever browser mechanism is putting the events *onto* the queue is external to this event loop model. The processing necessary to determine when events have occurred and to push them onto the event queue does not participate in the thread that's *handling* the events.

For example, when the end user waves the mouse around on the page, the browser will detect these motions and push a bunch of `mousemove` events onto the event queue. The event loop will eventually come across these events and trigger any handlers established for that type of event.

Such event handlers are a case of a more general concept known as ***callback functions***. Let's explore that very important concept.

THE CALLBACK CONCEPT

Whenever we set up a function for something else to call at a later time, be it the browser or other code, we are setting up what is termed a ***callback***. The term stems from the fact that we establish a function that some other code will later "call back" into at an appropriate point of execution.

We'll find that callbacks are an essential part of using JavaScript effectively and we're about to see a real-world example of how callbacks are used. But it's a tad complex, so before we dive into it, let's strip the callback concept completely naked and examine it in its simplest form.

We'll see callbacks used extensively as event handlers throughout the remainder of this book, but event handlers are just one example of callbacks; we can even employ callbacks ourselves in our own code. Here's a completely useless example of a function that accepts a reference to another function as a parameter and calls that function as a callback:

```
function useless(callback) { return callback(); }
```

But as useless as this example is, it clearly demonstrates the ability to pass a function as an argument to another function, and to subsequently invoke that function.

We can test our useless function with:

```
var text = 'Domo arigato!';
assert(useless(function(){ return text; }) === text,
       "The useless function works! " + text);
```

Here, we use the `assert()` testing function that we set up in the previous chapter to verify that the callback function is invoked and returns the expected value, which is in turn returned as the useless value. The result is shown in figure 3.3.

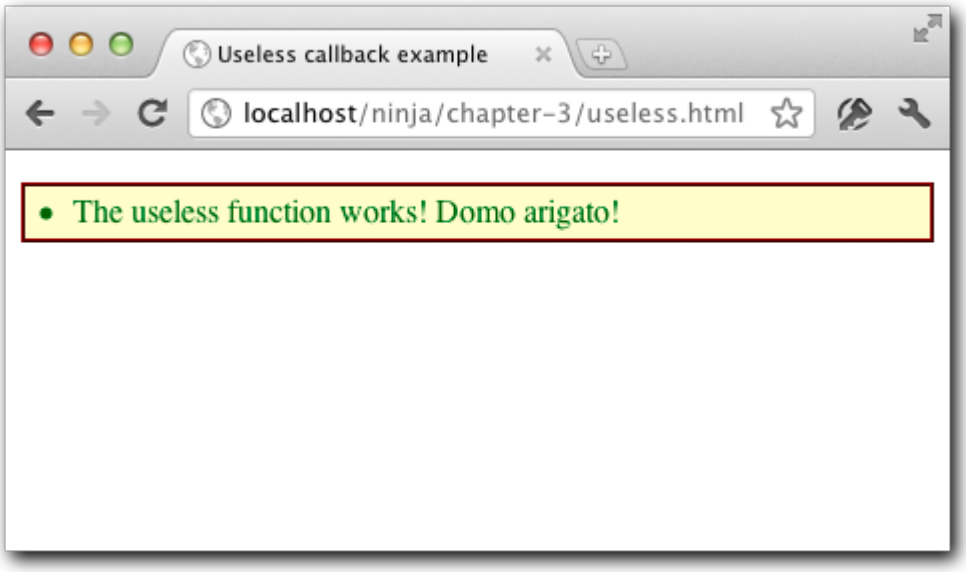

Figure 3.3 Our useless function may not do much, but it shows that functions can be passed around and invoked at any later time

That was really, really easy. And the reason is because of JavaScript's functional nature that lets us deal with functions as first-class objects.

Now let's consider a not-so-useless example, and compare it with using callbacks in a non-functional language.

3.1.2 Sorting with a comparator

Almost as soon as we *have* a collection of data, odds are we're going to need to sort it in some fashion. And as it turns out, we're going to need a callback in order to do anything but the most simple of sort operations.

Lets say that we had an array of some numbers in a random order: 213, 16, 2058, 54, 10, 1965, 57, 9. That order might be just fine, but chances are that, sooner or later, we're going to want to have them sorted into some non-random order.

Both Java and JavaScript provide a simple means to sort arrays into ascending order. In Java:

```
Integer[] values = { 213, 16, 2058, 54, 10, 1965, 57, 9 };
Arrays.sort(values);
```

In JavaScript:

```
var values = [ 213, 16, 2058, 54, 10, 1965, 57, 9 ];
values.sort();
```

NOTE We're not *picking* on Java – really, we're not. It's a fine language. We're just using Java as the crutch here because it's a good example of a language *without* functional capabilities, and one that lots of developers coming to JavaScript are familiar with.

There are some minor differences between the implementations of sorting in these languages – most notably, Java supplies a utility class with a static function, while JavaScript provides the capability as a method on the array itself – but both approaches are straightforward and easy to understand. But if we decide we want a sorting order *other* than ascending, something as simple as descending for example, things start to diverge rather markedly.

In order to provide a means to allow us to sort the values into *any* order we want, both languages let *us* provide a comparison algorithm that tells the sort algorithm how the values should be ordered. So instead of just letting the sort algorithm decide what values go before other values, *we'll* provide a function that performs the comparison. We'll give the sort algorithm access to this function as a callback, and it will call it whenever it needs to make a comparison.

The concept is similar in both languages, but the implementations couldn't be more different.

In non-functional Java, methods cannot exist on their own, and cannot be passed as arguments to other methods. Rather, they must be declared as a member of an object that *can* be instantiated and passed to a method. So the `Arrays.sort()` method has an overload that accepts an object containing the comparison method that it will call as a callback whenever a comparison needs to be made. This object and its method must conform to a known format (Java being strongly typed), so the following interface needs to be defined:

```
public interface Comparator<T> {
  int compare(T t, T t1);
  boolean equals(java.lang.Object o);
}
```

A novice Java developer might create a concrete class that implements this interface, but to make a fair comparison, we're going to assume a fair level of Java savvy-ness and use an inline anonymous implementation. So a usage of the `Arrays.sort()` static method to sort the values in descending order could look like the following code:

```
Arrays.sort(values,new Comparator<Integer>(){
  public int compare(Integer value1, Integer value2) {
    return value2 - value1;
  }
});
```

The `compare()` method of the inline Comparator implementation is expected to return a negative number if the order of the passed values should be reversed, a positive number if not, and zero if the values are equal; so simply subtracting the values produces the desired return value to sort the array into descending order.

The result of running the above code is the re-sorted array:

```
2058, 1965, 213, 57, 54, 16, 10, 9
```

Well, that wasn't overly complicated, but it did involve a fair amount of syntax, especially if you include the declaration of the required interface, to perform an operation that's fairly simple in nature.

The wordiness of this approach becomes even more apparent when we consider the equivalent JavaScript code that takes advantage of JavaScript's functional capabilities:

```
var values = [ 213, 16, 2058, 54, 10, 1965, 57, 9 ];
values.sort(function(value1,value2){ return value2 - value1; });
```

No interfaces. No extra object. One line.

We simply declare an inline anonymous function that we directly pass to the `sort()` method of the array.

The functional difference in JavaScript is the ability to create a function as a standalone entity, just as we can any other object type, and to pass it as an argument to a method, just like any other object type, which can accept it as a parameter, just like any other object type. It's that "first class" status coming into play.

That's something not even remotely possible in non-functional languages such as Java.

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

But regardless of how functions are declared (much more on this in the upcoming section), 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.

Now let's take a more in-depth look at how functions are declared and invoked. On the surface it may seem that there's not much to the acts of declaring and invoking functions, but there's actually a lot going on that we need to be aware of.

3.2 Declarations

JavaScript functions are declared using a ***function literal*** that creates a function value in the same way that a numeric literal creates a numeric value. Remember that, as first class objects, functions are values that can be used in the language just like other values such as strings and numbers.

And whether you realize it or not, you've been doing that all along.

Function literals are composed of four parts:

1. The `function` keyword.
2. An *optional* name that, if specified, must be a valid JavaScript identifier.

3. A comma-separated list of parameter names enclosed in parentheses; the names must be valid identifiers and the list can be empty. The parentheses must always be present, even with an empty parameter list.
4. The body of the function as a series of JavaScript statements enclosed in braces. The body can be empty, but the braces must always be present.

The fact that the function name is optional may come as a surprise to some developers, but we've seen ample examples if just such ***anonymous functions*** in the previous section. If there's no need for a function to be referenced by its name, we don't have to give it one. (Sort of like the joke about cats: *why give a cat a name if it's not going to come when called?*)

When a function is named, that name is valid throughout the scope within which the function is declared. Additionally, if a named function is declared at top-level, a property using the function name is created on `window` that references the function.

And lastly, all functions have a property named `name` that stores its name as a string. Functions with no name still possess this property, set to the empty string.

Why just *say* all that, when we can *prove* it?

We can write tests to assert that what we've said about functions is true. Examine the code of listing 3.1.

Listing 3.1: Proving things about the way that functions are declared

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

  function isNimble(){ return true; }                              //#1

  assert(typeof window.isNimble === 'function',                    //#2
         "isNimble() defined");
  assert(isNimble.name === 'isNimble',
         "isNimble() has a name");


  var canFly = function(){ return true; };                         //#3

  assert(typeof window.canFly === 'function',                      //#4
         "canFly() defined");
  assert(canFly.name === '',
         "canFly() has no name");


  window.isDeadly = function(){ return true; };                    //#5

  assert(typeof window.isDeadly === 'function',                    //#6
         "isDeadly() defined");

  function outer(){                                                //#7
    assert(typeof inner === 'function',
           "inner() in scope before declaration");
    function inner(){}
    assert(typeof inner === 'function',
```

```
            "inner() in scope after declaration");
      assert(window.inner === undefined,
             "inner() not in global scope");
    }

    outer();                                                    //#8
    assert(inner === undefined,
           "inner() still not in global scope");

</script>
```

#1 Declares a named function. The name is available throughout the current scope and is implicitly added as a property of `window`.
#2 The first test asserts that the `window` property is established, and the second that the `name` property of the function is recorded.
#3 Creates an anonymous function that is assigned to variable `canFly`. This does not create a named function. Rather, the variable is an explicit window property, and the `name` property of the function is empty.
#4 Tests that the variable references the anonymous function, and that the `name` property is set to the empty string (not `null`).
#5 Create an anonymous function referenced by a property of `window`.
#6 Tests that the property reference the function. We could also test that the function has an empty `name` property here.
#7 Defines an `inner` function inside an `outer` function. Tests that `inner()` is able to be referenced before and after its declaration, and that no global name is created for `inner()`.
#8 Tests that `outer()` can be referenced in the global scope, but that `inner()` cannot.

In this test page, we declare globally scoped functions in three different ways:

1. The `isNimble()` function is declared as a named function (#1). This is likely the most common declaration style that most developers have seen. That will change as you progress through this book.
2. An anonymous function is created and assigned to a global variable named `canFly` (#3). Because of JavaScript's functional nature, the function can be invoked through this reference as `canFly()`. In this respect, it is *almost* functionally equivalent (no pun intended) to declaring a named function named "canFly", but not quite. One major difference: the function's `name` property is "", not "canFly".
3. Another anonymous function is declared, and assigned to a `window` property named `isDeadly` (#5). Again, we can invoke the function through this property (`window.isDeadly()` or simply `isDeadly()`), and this is again *almost* functionally equivalent to a named function named "isDeadly".

Throughout the example, we placed assertions that verify that what we said about functions is true, the results of these tests being shown in figure 3.4.

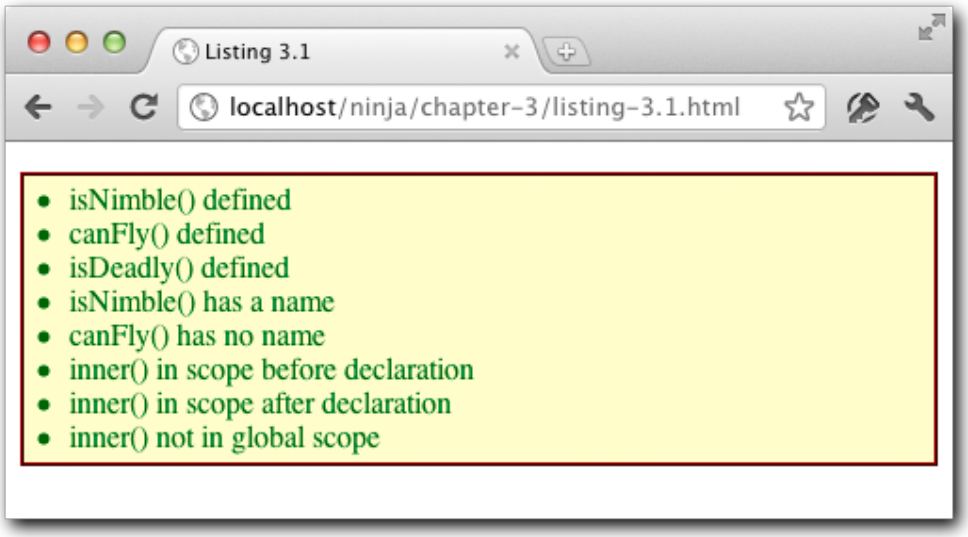

Figure 3.4 Running our test page shows that all those things that that we said about functions are true!

The test prove that:

- That `window.isNimble` is defined as a function. This proves that named functions are added as properties to `window` (#2).
- That the named function `isNimble()` has a `name` property that contains the string "isNimble" (#2).
- That `window.canFly` is defined as a function, proving that global variables, even those containing functions, end up on `window` (#4).
- That the anonymous function assigned to `canFly` has a name property consisting of the empty string (#4).
- That `window.isDeadly` is defined as a function (#6).

This is far from a complete test set of *everything* that we said about functions so far. How would you extend this test code to assert the suppositions for the declared functions?

Then comes the time to test non-global functions. We create a function, appropriately named `outer()` in which we will test our assertions regarding functions declared in a non-global scope (#7). We declare an inner function named `inner()`, but before it is declared, we assert that the function is in scope. This tests our assertion that a function is available throughout the scope within which it is declared, even when forward-referenced.

Then we declare the function, check that it is within scope inside the function, and check that it is *not* within the global scope.

Finally, we execute the inner test and once again assert that the inner function did not creep its way out into the global scope (#8).

These concepts are very important as they lay down the foundations for the naming, flow, and structure that functional code provides, and begins to establish the framework through which we employ functional programming to our great benefit.

The point we made with the inner function (#8) – namely, that the function is forward-referencable within the out function – may have you wondering: "When we declare a function, what scope is that function available within?" It's a good question and one that we'll answer next.

3.2.1 Scoping and functions

When we declare a function, not only do we need to be concerned with the scope within which that function is available, but also what scopes the function itself *creates*, and how declarations within the function are affected by those scopes.

Scopes in JavaScript act somewhat differently than in most other languages whose syntax is influenced by C; namely, those that use *braces* (`{` and `}`) as block delimiters. In most of such languages, each block creates its own scope; not so in JavaScript!

In JavaScript, scopes are declared by *functions*, and not by blocks. So, the scope of a declaration that is created inside a block is *not* terminated (as it is in other languages) by the end of the block.

Consider the following code:

```
if (window) {
  var x = 213;
}
alert(x);
```

In most other languages, one would expect the scope of the declaration for `x` to terminate at the end of the block created by the `if` statement, and for the alert to fail with an undefined value. But if we were to run the above code in a page, the value 213 would be alerted, because JavaScript does not terminate scopes at the end of blocks.

That seems simple enough, but there are a few nuances to the scoping rules depending upon what is being declared. In general:

- Variable declarations are in scope from their point of declaration to the end of the function within which they are declared, regardless of block nesting.
- Named functions are in scope within the entire function within which they are declared, regardless of block nesting.
- For the purposes of declaration scopes, the global context acts like one big function encompassing the code on the page.

Once again, instead of just saying it, we're going to *prove* it. Take a look at the code snippet below:

```
function outer(){

  var a = 1;
```

```
  function inner(){ /* does nothing */ }

  var b = 2;

  if (a == 1) {
    var c = 3;
  }

}

outer();
```

In the code, we declare five items: an outer function named `outer()`, a function inside that function named `inner()`, and three numeric variables inside the outer function named `a`, `b`, and `c`.

To test where the various items are in scope, and perhaps more importantly, where they are not, we'll intersperse a block of tests throughout this code. We'll put the same block of tests, with one test for each of these declarations, at strategic places in the code. Each test asserts that one of the items that we're declaring is in scope (except for the first, which isn't at test at all, but just a label that will help keep the code and output more readable.)

This test block is:

```
assert(true,"some decriptive text");
assert(typeof outer==='function',
      "outer() is in scope");
assert(typeof inner==='function',
      "inner() is in scope");
assert(typeof a==='number',
      "a is in scope");
assert(typeof b==='number',
      "b is in scope");
assert(typeof c==='number',
      "c is in scope");
```

Note that in many circumstances some of these tests will **fail**! Under normal circumstances, we'd expect our asserts to always pass; but in this code, which is only for demonstration, it suits our purposes to show where the tests pass and where they fail, which directly corresponds to whether the tested item is in scope or not.

Listing 3.2 shows the code with the repeated test as shown above removed so that we can see the forest for the trees (wherever the test code has been removed, we show the comment `/* test code here */` so you'll know where the test code appears in the actual page file).

The completely assembled code, with declarations and embedded tests, is shown in listing 3.2

Listing 3.2: Observing the scoping behavior of declarations

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

  assert(true,"|----- BEFORE OUTER -----|");
  /* test code here */                                    #1
```

```
  function outer(){
  assert(true,"|----- INSIDE OUTER, BEFORE a -----|");
  /* test code here */                                    #2

    var a = 1;

    assert(true,"|----- INSIDE OUTER, AFTER a -----|");
   /* test code here */                                   #3

    function inner(){ /* does nothing */ }

    var b = 2;

    assert(true,"|----- INSIDE OUTER, AFTER inner() AND b -----|");
    /* test code here */                                  #4

    if (a == 1) {
      var c = 3;
      assert(true,"|----- INSIDE OUTER, INSIDE if -----|");
      /* test code here */                                #5
    }

    assert(true,"|----- INSIDE OUTER, OUTSIDE if -----|");
    /* test code here */                                  #6

  }

  outer();

  assert(true,"|----- AFTER OUTER -----|");
  /* test code here */                                    #7

</script>
```

#1 Runs our test block before we've defined anything at all! As all our tests assert that each item is in scope, all but the tests for items than can be forward referenced will fail. As such, only the top-level function `outer()` is in scope at his point. See figure 3.5 (or better yet, run the code in your browser so that you don't have to flip pages as much) to verify that all test but that for `outer()` fail.

#2 Runs our test block inside function `outer()` but before anything else has been declared. The `outer()` function is still in scope as is the `inner()` function which is defined within `outer()` function. Functions can be forward referenced, but not variable declarations, so all other tests fail.

#3 Runs the test block inside `outer()` and after the variable `a` has been declared. Test results show that `a` has been added to the scope at this point.

#4 Runs test code after `inner()` and `b` have been declared. The testing shows that `b` has been added to the scope. The fact that `inner()` was declared at this point is moot. Its scope extends back to the beginning of the containing function and its declaration certainly doesn't remove it from the scope!

#5 Runs the tests inside an if-block after variable `c` has been declared in that block. Tests show that all items are in scope at this point.

#6 Runs the test code inside `outer()` but after the if-block has been closed. Test show that all items are in scope; even `c` although the if-block within which it was declared is closed! Unlike most other block-structured languages, variable declarations extend from the point of declaration to the end of the function, crossing any block boundaries.

#7 Runs test in global scope, after `outer()` has been declared. Once again, only `outer()` is in scope as the scope of anything declared *within* `outer()` is confined to within it.

Running this code results in the display of figure 3.5.

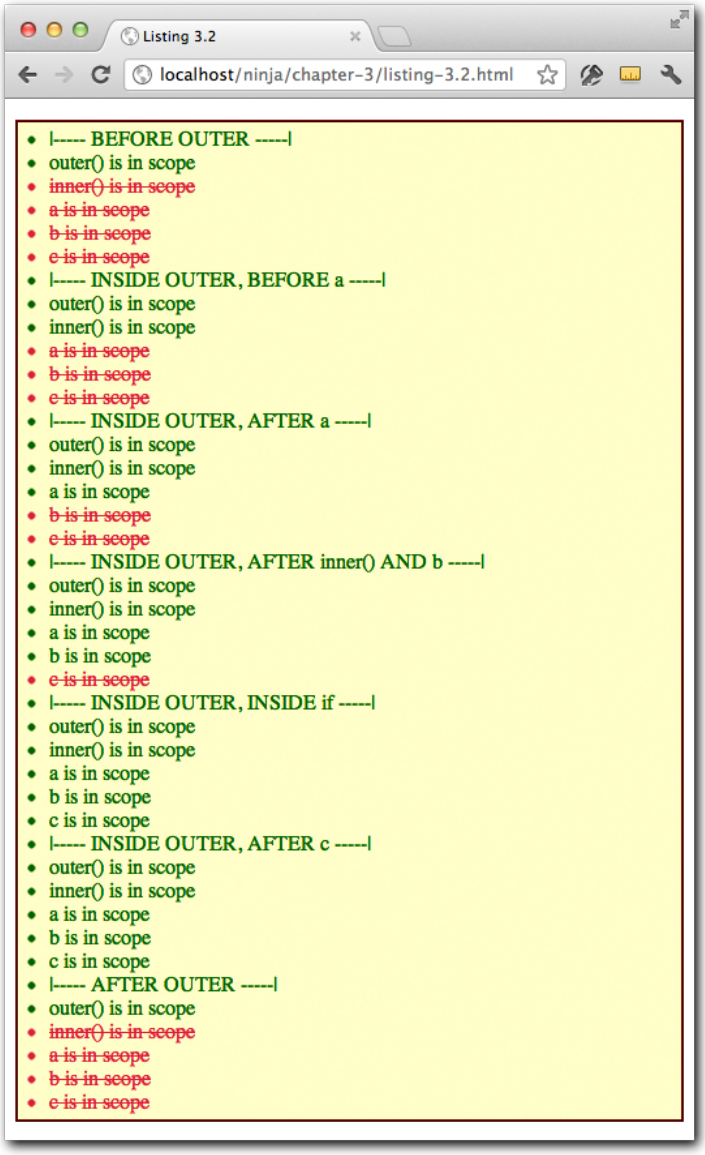

Figure 3.5 Running our scope tests clearly shows where the declared items are in scope, and where they are not

As expected, there are many failures as not all of the items are in scope at every position where we placed the block of tests.

Of particular note, see how the declaration of `inner()` is available through the entire `outer()` function, while the numeric variables `a`, `b` and `c` are only available from their point of declaration to the end of `outer()`. This clearly shows how function declarations can be forward referenced within their scope, while variables cannot.

Also take particular note how the closing of the `if` statement block within which `c` is declared does *not* terminate the scope of `c`. Variable `c`, despite being nested in a block, is available from its point of declaration to the end of `outer()` just like the variables not defined in a nested block.

The scopes of the various declared items are graphically depicted in figure 3.6.

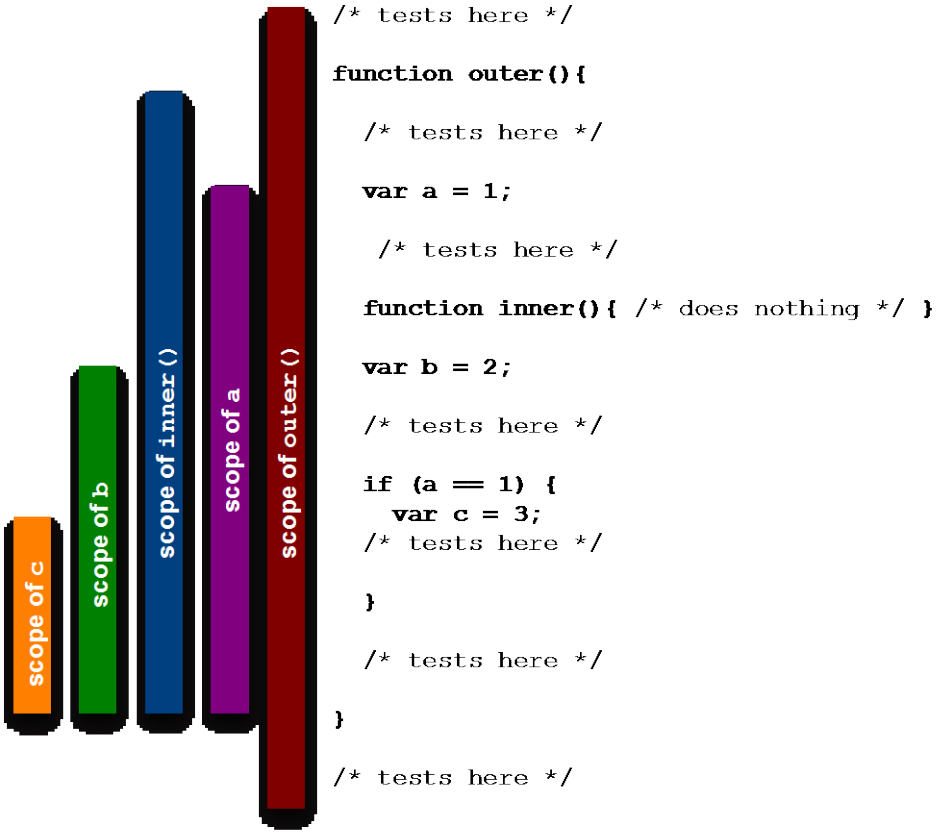

Figure 3.6 The scope of each declared item depends not only on where it is declared, but by whether it is a variable or a function

PONDER THIS Now that you understand a bit about scope, you should be able to answer the following question: rather than cutting and pasting the block of tests over and over again, why did we just not create a function to hold them once and call as needed?

Knowing how functions are declared, let's see how we can invoke them.

3.3 Invocations

We've all called JavaScript functions, but have you ever stopped to wonder what really happens when a function is called? In this section, we'll examine the various ways that functions can be invoked.

As it turns out, the manner in which a function is invoked has a huge impact on how the code within it operates; primarily in how the `this` parameter is established. This difference is much more important than it might seem at first, and it's one that we'll examine within this section and be exploiting throughout the rest of this book to help elevate our code to ninja level.

There are actually four different ways to invoke a function, each with their own nuances. They are:

1. As a function, in which the function is invoked in a straightforward manner.
2. As a method, which ties the invocation to an object, enabling object-oriented programming.
3. As a constructor, in which a new object is brought into being.
4. Via their `apply()` or `call()` methods, which, well, is kind of complicated, so we'll take that when we get to it.

For all but the last of these, the function invocation operator is a set of parentheses following any expression that evaluates to a function reference. Any arguments to be passed to the function are including inside the parentheses characters as a comma-separated list.

Before we take a close look at those four ways to make our functions execute, let's examine what happens to these arguments that are to be passed to the incovations.

3.3.1 From arguments to function parameters

When a list of arguments is supplied as part of a function invocation, these arguments are assigned to the parameters specified in the function declaration in the same order that each was specified. The first argument gets assigned to the first parameter, the second argument to the second parameter, and so on.

If there are a different number of arguments than there are parameters, no error is raised; JavaScript is perfectly fine with this situation, and deals with it as follows:

- If more arguments are supplied than there are parameters, the "excess" arguments are simply not assigned to parameter names. We'll see in just a bit that even though the arguments aren't assigned to parameter names, we still have a way to get at them.

- If there are more parameters than there are arguments, the parameters that have no corresponding argument are set to `undefined`.

And, very interestingly, all function invocations are also passed two implicit parameters: `arguments` and `this`.

By *implicit*, we mean that these parameters are not explicitly listed in the function signature, but that they are silently passed to the function, and in scope within the function. Thus, they can be referenced within the function just like any other explicitly named parameter.

Let's take a look at each of these implicit parameters in turn.

THE ARGUMENTS PARAMETER

The `arguments` parameter is set as a collection of all of the arguments passed to the function. The collection has a property named `length` that contains the count of arguments, and the individual argument values can be obtained using array indexing notation; `arguments[2]` to fetch the 3rd parameter, for example.

But note that we went out of our way to avoid calling the `arguments` parameter an array. You may be fooled into to thinking that it is an array; after all, it has a `length` parameter, its entries can be fetched using array notation, and we can even iterate over it with a `for` loop. But it is *not* a JavaScript array, and if you try to use array methods on `arguments` you'll find nothing but heartbreak and disappointment. Just think of `arguments` as an "array-like" construct, and exhibit restraint in its use.

The `this` parameter is even more interesting.

THE `THIS` PARAMETER

Whenever a function is invoked, in addition to the parameters that represent the explicit arguments that were provided on the function call, an implicit parameter named `this` is also passed to the function. The `this` parameter refers to an object that is implicitly associated with the function invocation and is termed the ***function context***.

The function context is a notion that those coming from OO languages such as Java will think that they understand – that `this` points to an instance of the class within which the method is defined. But beware! As we will see, invocation as a *method* is only one of the four ways that a function can be invoked. And as it turns out, what the `this` parameter points to is not, as in Java, defined by how the function is declared, but by how it is *invoked*!

Because of this fact, it might have been clearer to call `this` the *invocation* context, but we were never consulted about the name.

We're about to look at how the four invocation mechanisms differ, and we'll see that one of the primary differences between them is how the value of `this` is determined for each invocation means. And then we'll be taking a long and hard look at function contexts again in section 3.4; so don't worry if things don't gel right away, we'll be discussing `this` at great length.

Now let's see how functions can be invoked.

3.3.2 Invocation as a function

"Invocation as a function"? Well, of course functions are invoked as *functions*. How silly to think otherwise.

But in reality, we say that a function is invoked "as a function" to distinguish it from the other invocation mechanisms: methods, constructors and apply/call. If a function is not invoked as a method, as a constructor, or via `apply()` or `call()`, it is simply invoked "as a function".

This type of invocation occurs when a function is invoked using the () operator, and the expression to which the () operator is applied does not reference the function as a property of an object. (In that case, we'd have a method invocation, but we'll discuss that next.)

Some simple examples:

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

var samurai = function(){};
samurai();
```

When invoked in this manner, the function context is the global context; that is, the `window` object.

We're going to refrain from writing any tests to prove this at the moment as it'll be more interesting to do so when we have something to compare it to, like the method invocations that we'll discuss next.

3.3.3 Invocation as a method

When a function is assigned to a property of an object *and* the invocation occurs by referencing the function using that property, then the function is invoked as a *method* of that object. As in:

```
var o = {};
o.whatever = function(){};
o.whatever();
```

OK, so what? The function is called a "method" in this case, but what makes that interesting or useful?

Well, if you come from any object-oriented background, you'll remember that the object to which a method belongs is available within the body of the method as `this`. The same thing happens here! When we invoke the function as the *method* of an object, that object becomes the function context and is available within the function via the `this` parameter.

This is one of the primary means by which JavaScript allows object-oriented code to be written. (Constructors are another, and we'll be getting to them in short order.)

Contrast this with invocation "as a function" in which the function context isn't set to anything particularly useful.

Let's consider some test code to illustrate the differences between invocation as a function and invocation as a method, shown in listing 3.3.

Listing 3.3 Illustrating the difference between function and method invocations

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

  function creep(){ return this; }                    #1

  assert(creep() === window,                          #2
         "Creeping in the window");                   #2

  var sneak = creep;                                  #3

  assert(sneak() === window,                          #4
         "Sneaking in the window");                   #4

  var ninja1 = {                                      #5
    skulk: creep                                      #5
  };                                                  #5

  assert(ninja1.skulk() === ninja1,                   #6
         "The 1st ninja is skulking");                #6

  var ninja2 = {                                      #7
    skulk: creep                                      #7
  };

  assert(ninja2.skulk() === ninja2,                   #8
         "The 2nd ninja is skulking");                #8
</script>
```

#1 Defines a function that returns its function context. This will allow us to examine the function context of a function from outside of it, *after* is has been invoked.
#2 Tests an invocation "as a function" and verifies that the function context was the window object (the global scope). Figure 3.7 shows that this test passes.
#3 Creates a reference to the same function in variable `sneak`.
#4 Invokes the function using the `sneak` variable. Even though we've used a variable, the function is still invoked as a function, and the function context is `window`.
#5 Creates an object in `ninja1` and creates a `skulk` property that reference the original `creep()` function.
#6 Invokes the function through the skulk property, thus invoking it as a method of `ninja1`. The function context is no longer `window` but is `ninja1`. That's object oorientation!
#7 Creates another object, `ninja2`, which also has a `skulk` property referencing `creep()`.
#8 Invokes the function as a method of `ninja2`, and behold, the function context is `ninja2`.

Figure 3.7 shows that all our test assertions pass.

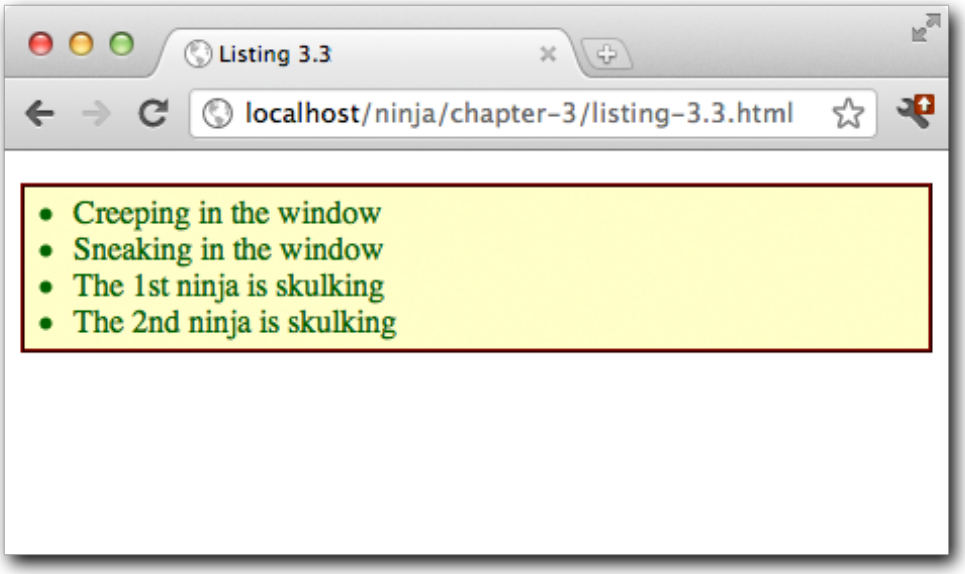

Figure 3.7 A single function, invoked in various ways, can serve as either a "normal" function or a method

In this test, we set up a single function named `creep` (#1) that we'll use throughout the rest of the code. The only thing that this function does is to return its function context so that we can see, from outside the function, what the function context for the invocation was. (Otherwise, we'd have no way of knowing.)

When we call the function by its name, this is a case of invoking the function "as a function", so we'd expect that the function context would be the global context – in other words, the `window`. We assert that this is so (#2), and as we see in figure 3.7, this assertion passes.

So far, so good.

Then we create a reference to the function in a variable named `sneak` (#3). Note that this does *not* create a second instance of the function; it merely creates a reference to the same function. You know, first-class object and all.

When we invoke the function via the variable – something we can do because the function invocation operator can be applied to any expression that evaluates to a function – we'd once again be invoking the function as a function. As such, we'd once again expect that the function context would be the `window` (#4), and it is.

Next, we get a bit trickier and define an object in variable `ninja1` with a property named `skulk` that receives a reference to the `creep()` function #(5). By doing so, we say that we have created a *method* named `skulk` on the object. We do *not* say that `creep()` has

become a method of `ninja1`; it hasn't. We've already seen that `creep()` is its own independent function that can be invoked in numerous ways.

According to what we stated earlier, when we invoke the function via a method reference, we expect the function context to be the method's object; in this case, `ninja1`, and assert as much (#6). Again figure 3.5 shows us that this is borne out. We're on a roll!

This particular ability is crucial to writing JavaScript in an object oriented-manner. It means that we can, within any method, use `this` to reference the method's owning object – a fundamental concept in object-oriented programming.

To drive that point home, we continue our testing by creating yet another object, `ninja2`, also with a property named `skulk` that references the `creep()` function (#7). Upon invoking this method through its object, we correctly assert that its function context is `ninja2`.

Note that even though the *same* function is used throughout all these examples, that the function context for each invocation of the function changes depending upon how the function is *invoked*, rather than on how it was declared.

For example, the exact same function instance is shared by both `ninja1` and `ninja2`, yet when executed, the function has access to, and can perform operations upon, the object through which the method was invoked. This means that we don't need to create separate copies of a function to perform the exact same processing on different objects – a tenet of object-oriented programing.

This is a powerful capability, yet the manner in which we used it in this example has limitations. Foremost, when we created the two ninja objects, we were able to share the same function to use as a method in each, but we had to use a bit of repeated code to set up the separate objects and their `skulk` methods.

But that's nothing to despair over – JavaScript provides mechanisms to make creating objects from a single pattern much easier than in this example. And we'll be exploring those capabilities in depth in chapter 6. But for now, let's consider a part of that mechanism that relates to function invocations: the ***constructor***.

3.3.4 Invocation as a constructor

There's nothing special about a function that's going to be used as a constructor; constructor functions are declared just like any other function. The difference is in how the function is invoked.

To invoke the function *as a constructor*, we precede the function invocation with the `new` keyword.

For example, let's recall the `creep()` function from the previous section:

```
function creep(){ return this; }
```

If we want to invoke the `creep()` function from the previous section as a constructor, we would write:

```
new creep();
```

But even though we can invoke `creep()` as a constructor, that function isn't particularly well suited for use as a constructor. Let's find out why by discussing what makes constructor special.

THE SUPER-POWERS OF CONSTRUCTORS

Invoking a function as a constructor is a powerful feature of JavaScript because when a constructor is invoked, the following special actions take place:

- A new empty object is created.
- This object is passed to the constructor as the `this` parameter, and thus becomes the constructor's function context.
- In the absence of any explicit return value, the new object is returned as the constructor's value.

This latter point is why `creep()` makes for a lousy constructor. The purpose of a constructor is to cause a new object to be created, to set it up, and to return it as the constructor value. Anything that interferes with that intent is not appropriate for functions intended for use as a constructor.

Let's consider a more appropriate function: one that will set up the skulking ninjas of listing 3.3 in a more succinct fashion. See the code of listing 3.4.

Listing 3.4 Using a constructor to set up common objects

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

  function Ninja() {                                  #1
    this.skulk = function() { return this; };         #1
  }                                                   #1

  var ninja1 = new Ninja();                           #2
  var ninja2 = new Ninja();                           #2

  assert(ninja1.skulk() === ninja1,                   #3
         "The 1st ninja is skulking");                #3
  assert(ninja2.skulk() === ninja2,                   #3
         "The 2nd ninja is skulking");                #3

</script>
```

#1 Defines the constructor which creates a `skulk` property on whatever object is the function context. The method once again returns the function context so that we can test it externally.
#2 Creates two objects by invoking the constructor with `new`. The newly created objects are referenced by `ninja1` and `ninja2`.
#3 Tests the methods of the constructed object. Each should return their own constructed object.

The results of this test are shown in figure 3.8.

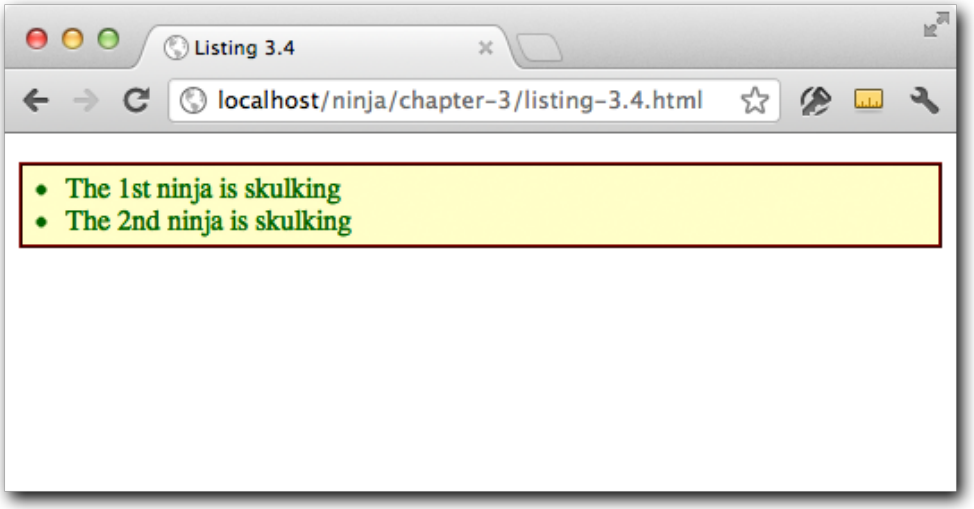

Figure 3.8 Constructors let us create multiple objects following the same pattern with a minimum of fuss and bother

In this example we create a function named `Ninja()` (#1) that we intend to use to construct, well, Ninjas. When invoked with the `new` keyword, an empty object instance will be created and passed to the function as `this`. The constructor creates a property named `skulk` on this object, which is assigned a function, making that property a method of the newly created object.

The method performs the same operation as `creep()` in the previous sections, returning the function context so that we can test it externally.

With the constructor defined, we create two new Ninja objects by invoking the constructor twice (#2). Note that the returned values from the invocations are stored in variables that become references to the newly created Ninjas.

Then we run the same tests copied from listing 3.3 to ensure that each invocation of the method operates upon the expected object (#3).

Functions intended for use as constructors are generally coded differently from other functions. Let's see how.

CODING CONSIDERATIONS FOR FUNCTIONS

The intent of constructors is to initialize the new object that will be created by the function invocation to initial conditions. And while such functions *can* be called as "normal" functions, or even assigned to object properties in order to be invoked as methods, they're generally not very useful as such.

For example, it'd be perfectly valid to call the `Ninja()` function as follows:

```
var whatever = Ninja();
```

But the effect would be for the `skulk` property to be created on `window`, and for `window` to be returned and stored in `whatever`; not a particularly useful operation.

Because constructors are generally coded and used in a manner that's different from other functions, and generally aren't all that useful unless invoked as constructors, a naming convention has arisen to distinguish constructors from run-of-the-mill functions and methods. If you've been paying attention, you may have already noticed it.

Functions and methods are generally named starting with a verb that describes what they do (`skulk()`, `creep()`, `sneak()`, `doSomethingWonderful()` and so on) and start with a lowercase letter. Constructors on the other hand, are usually named as a noun that describes the object that's being constructed, and start with an uppercase character; `Ninja()`, `Samurai()`, `Ronin()`, `KungFuPanda()` etc.

It's pretty easy to see how a constructor makes it much easier to create multiple objects that conform to the same pattern without having to repeat the same code over and over again. The common code is written once, as the body of the constructors. We'll learn much more about using constructors and about the other object-oriented mechanisms that JavaScript provides that make it even easier to set up object patterns in chapter 6.

But we're not done with function invocation yet. There's yet another way that JavaScript lets us invoke functions that gives us a great deal of control over the invocation details.

3.3.5 Invocation with the apply and call methods

So far, we've seen that one of the major differences between the types of function invocation is what object ends up as the function context referenced by the implicit `this` parameter passed to the executing function. For functions, it is always `window`; for methods, the method's owning object; and for constructors, a newly created object instance.

But what if we wanted to make it whatever we wanted?

What if we wanted to set it explicitly?

What if…. well, why would we?

To give a glimpse of why we'd care about this ability, we'll get a bit ahead of ourselves and consider that when an event handler is called, the function context is set to the bound object of the event. We'll examine event handling in detail in chapter 13, but for now just assume that the bound object is the object upon which the event handler is established.

That's usually exactly what we want; but not always. For example, in the case of a method, we might want to force the function context to be the owning object of the method and not the object to which the event is bound.

We'll see this scenario in chapter 13, but for now, the question is, can we do that?

Well, yes we can.

USING THE APPLY AND CALL METHODS

JavaScript provides a means for us to invoke a function, and to explicitly specify any object we want as the function context. The way that we do this is through the use of one of two methods that exist for every function: `apply()` or `call()`.

Yes, we said methods of functions. As first-class objects, functions can have properties, including methods, just like any other object type.

To invoke a function using its `apply()` method, we pass two parameters to `apply()`: the object to be used as the function context, and an array of values to be used as the invocation arguments. The `call()` method is used in a similar manner, except that the arguments are passed directly in the argument list rather than as an array.

Listing 3.5 shows both of these methods in action.

Listing 3.5 Using the apply and call methods to supply the function context

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

  function juggle() {                                    #1
    var result = 0;
    for (var n = 0; n < arguments.length; n++) {         #2
      result += arguments[n];
    }
    this.result = result;                                #3
  }

  var ninja1 = {};                                       #4
  var ninja2 = {};                                       #4

  juggle.apply(ninja1,[1,2,3,4]);                        #5

  juggle.call(ninja2,5,6,7,8);                           #6

  assert(ninja1.result === 10,"juggled via apply");      #7
  assert(ninja2.result === 26,"juggled via call");       #7

</script>
```

#1 Defines a function
#2 Sums up the arguments
#3 Stores result on context
#4 Sets up test subjects
#5 Applies the function
#6 Calls the function
#7 Tests the expected results

The results are shown in figure 3.9.

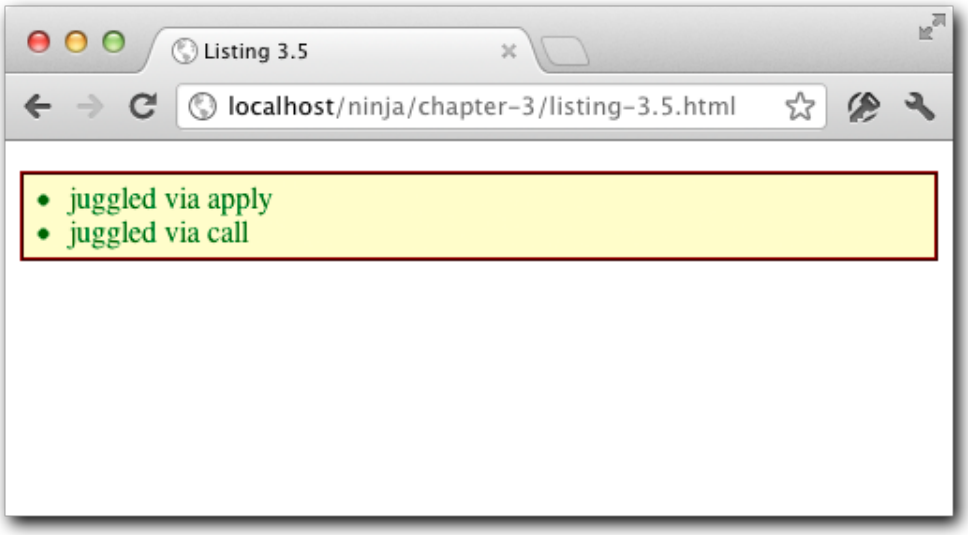

Figure 3.9 The apply and call methods let us set the function context to any object of our choosing

In this example, we set up a function named `juggle()`(#1), in which we define juggling as adding up all the arguments (#2) and storing them as a property named `result` on the function context (#3). That may a rather lame definition of juggling, but it *will* allow us to surmise whether arguments were passed to the function correctly, and which object ended up as the function context.

We then set up two objects that we'll use as function contexts (#4), passing the first to the function's `apply()` method, along with an array of arguments (#5), and passing the second to the function's `call()` method, along with a number of other arguments.

Then we test (#7)!

First, we check that `ninja1`, which was passed via `apply()` received a `result` property that's the result of adding up all the argument values. Then we do the same for `ninja2` which was passed via `call()`.

The results in figure 3.9 show that the tests passed, meaning that we were successfully able to specify arbitrary objects to serve as functions contexts for function invocations.

This can come in handy whenever it would be expedient to usurp what would normally be the function context with an object of our own choosing; something that can be particularly useful when invoking callback functions.

FORCING THE FUNCTION CONTEXT IN CALLBACKS

Let's consider a concrete example of this. Let's take something as simple as a function that will perform an operation one every entry of an array. In imperative programming, it's

common to pass the array to a method and use a for-loop to iterate over every entry, performing the operation on each entry.

```
function(collection) {
  for (var n = 0; n < collection.length; n++) {
    /* do something to collection[n] */
  }
}
```

The functional approach would rather be to create a function that operates on a single element, and pass each entry to that function.

```
function(item){
  /* do something to item */
}
```

The difference lies in thinking at a level where functions are the building blocks of the program rather than imperative statements.

You might think that it's all rather moot, and that all we're doing is moving the for-loop out one level, but we're not done massaging this example yet.

In order to facilitate a more functional style, quite a few of the popular JavaScript libraries provide a "for-each" function that invokes a callback on each element within an array. This is often more succinct, and a style preferred over the traditional `for` statement by those familiar with functional programming. Its organizational benefits will become even more evident once we've covered closures in chapter 5.

Such an iteration function *could* simply pass the "current" element to the callback as a parameter, but most make the current element the function context of the callback.

Let's build our own (simplified) version of such a function in Listing 3.6.

Listing 3.6 Building a for-each function to demonstrate setting an arbitrary function context

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

  function forEach(list,callback) {                    #1
    for (var n = 0; n < list.length; n++) {
      callback.call(list[n],n);                        #2
    }
  }

  var list = ['shuriken','katana','nunchucks'];        #3

  forEach(                                             #4
    list,
    function(index){ console.log(index); console.log(this);
      assert(this == list[index],
             "Got the expected value of " + list[index]);
    }
  );

</script>
```

#1 Defines the for-each function
#2 Invokes the callback
#3 Sets up a test subject

#4 Tests the function

Our iteration function sports a simple signature that expects the array of objects to be iterated over as the first argument, and a callback function as the second (#1). The function iterates over the array entries, invoking the callback function (#2) for each entry.

We use the `call()` method of the callback function passing the current iteration entry as the first parameter, and the loop index as the second. This *should* cause the current entry to become the function context, and the index to be passed as the single parameter to the callback.

Now to test that!

We set up a simple array (#3), and then call the `forEach()` function, passing the test array and a callback within which we test that the expected entry is set as the function context for each invocation of the callback (#4).

Figure 3.10 shows that our function works splendidly.

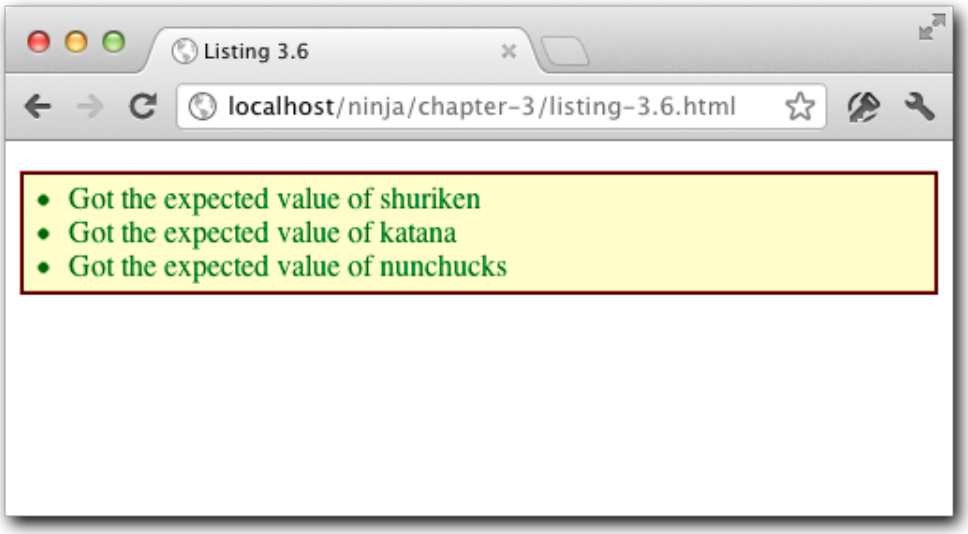

Figure 3.10 The test results show that we have the ability to make any object that we please the function context of a callback invocation

In a production-ready implementation of such a function, there'd be a lot more work to do. For example, what if the first argument isn't an array? What if the second isn't a function? How would you allow the page author to terminate the loop at any point?

As an exercise, augment the function to handle these situations.

Another exercise you could task your self with: enhance the function so that the page author can also pass an arbitrary number of arguments to the callback in addition to the iteration index.

3.4 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 learned:

- Writing sophisticated code hinges upon learning JavaScript as a functional language.
- Functions are first class objects that are treated just like any other objects within JavaScript. Just like any other object type, they can be:
    - Created via literals
    - Assigned to variables
    - Passed as parameters
    - Returned as function results
    - Possess properties and methods
- Each object has a "super power" that distinguishes it from the rest; for functions it's the ability to be invoked.
- Functions are created via literals for which a name is optional.
- The browser invokes functions during the lifetime of a page by invoking then as event handlers of various types.
- The scope of declaration within a function differs from that of most other languages. Specifically:
    - Variables within a function are in scope from their point of declaration to the end of the function, spanning block boundaries.
    - Inner named functions are available anywhere within the enclosing function; even as forward references.
- The parameter list of a function and its actual argument list can be different lengths.
    - Unassigned parameters evaluate as `undefined`.
    - Extra arguments are simply not bound to parameter names.
- Each function invocation is passed two implicit parameters:
    - `arguments`, a collection of the actual passed arguments.
    - `this`, a reference to the object serving as the function context.
- Functions can be invoked in various ways, and the invocation method determines the

function context value:

- When invoked as a simple function, the context is the global object (`window`).
- When invoked as a method, the context is the object owning the method.
- When invoked as a constructor, the context is a newly allocated object.
- When invoked via the `apply()` or `call()` methods of the function, the context can be whatever the heck we want.

In all, we made a thorough examination of the fundamentals of function mechanics. In the next chapter, we'll see how we can take this functional knowledge and put it into use.

4

Wielding functions

In this chapter:

- Why anonymous functions are so important
- Recursion in functions
- The ways that functions can be reference for invocation
- How to store references to functions
- Using function capabilities for memoization
- Using the function context to get our way
- Dealing with variable length argument lists
- Determining if an object is a function

In the previous chapter we focused on how JavaScript treats functions as first-order objects, and how that enables a functional programming style. In this chapter we'll expand on how to use those functions to solve various problems that we might come across when authoring web applications.

Not all of the examples used throughout this chapter are direct reflections of actual problems you will see in your web apps; after all, how many times will you write your own recursive mathematical functions? But we can't directly address each and every problem you might come across; that would take far too much space. Besides, that would turn this book into a "cookbook" and that's certainly *not* what this book is about.

We know that the purpose of this book is to expose you to the secrets that will help you to truly understand JavaScript. So, the examples have been carefully chosen to stay simple in nature, while exposing us to the important concepts that will be broadly applicable to the dilemmas that we are bound to run into in future coding projects.

Without further ado, let's take the functional JavaScript knowledge that we now possess in our two hands, and wield it like the mighty weapon that it is.

4.1 Anonymous functions

You may or may not have been familiar with anonymous functions prior to their introduction in the previous chapter, but they are a crucial concept with which we all need to be familiar if we're striving for JavaScript ninja-hood. They are an important and logical feature for a language that takes a great deal of inspiration from functional languages such as 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, as a method of an object, or using it as a callback; for example, as a timeout or event handler. In all of these situations, it doesn't need to have a name for later reference. We'll see plenty of examples throughout the rest of this chapter and book so don't panic if that still seems a bit strange at the moment.

Listing 4.1 shows some common examples of anonymous function declarations.

Listing 4.1: Common examples of using anonymous functions

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

  window.onload =                                    //#1
    function(){ assert(true, 'power!'); };           //#1

  var ninja = {
    shout: function(){                               //#2
      assert(true,"Ninja");                          //#2
    }                                                //#2
  };

  ninja.shout();

  setTimeout(
    function(){ assert(true,'Forever!'); },          //#3
    500);

</script>
```

#1 Establishes an anonymous function as an event handler. There's no need to create a named function only to reference it in this one location.
#2 Creates a function to be used as a method for `ninja`. We'll be using the property named `shout` to invoked the function, so it doesn't need its own name.
#3 Passes a function to the `timeout()` function as a callback to be invoked when the timer expires. Again, why bother to give it an unneeded name?

In listing 4.1, we do a couple of typical things.

First, we establish a function as a handler for the load event; (#1) we're never going to call this function directly; we're going to let the event handling mechanism do it for us. We could have done the same thing using:

```
function bootMeUp(){ assert(true, 'power!'); };
window.onload = bootMeUp;
```

But why bother to create the separate top-level function with a name when it's not really needed?

Next, we declare an anonymous function as a property of an object (#2), which we know from the previous chapter makes the function a method of the object. We then invoke the method using the property reference.

Another interesting use of an anonymous function, that should look familiar from the previous chapter, is its use as a callback supplied to another function call. In this example, we supply an anonymous function as an argument to the `setTimeout()` method (of `window`) (#3), which is invoked after one half of a second has elapsed.

The results (after letting things run for a second or two) can be seen in figure 4.1.

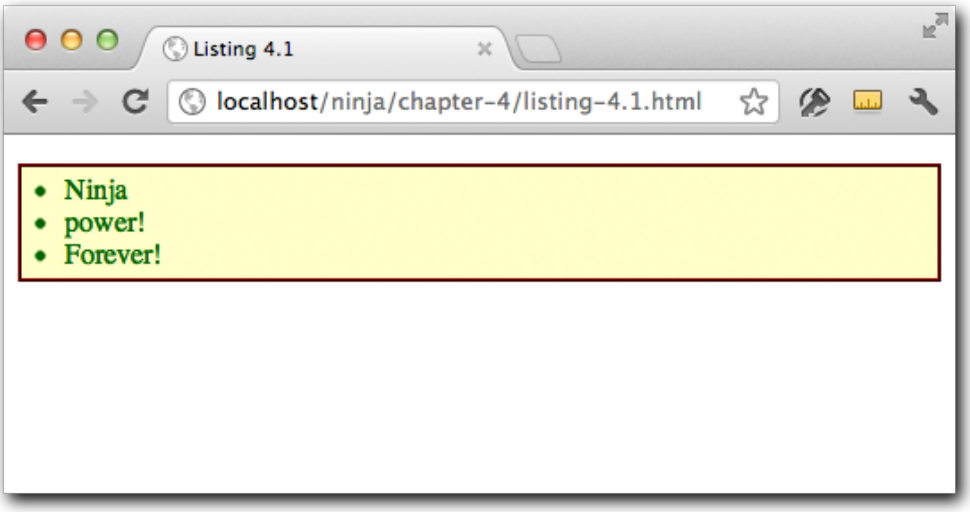

Figure 4.1 Anonymous functions can be called at various times despite not being named

Note how in all of these cases, the functions didn't need to have a name in order to be used after their declarations. Also note our use, once again, of the `assert()` function with a test condition of `true` as a lazy man's means of emitting output. Hey, we wrote the code, why not use it?

> **NOTE** Some might argue that by assigning an anonymous function to a property named `shout` that we give the function a name, but that's not the correct way of thinking about it. The `shout` name is the name of the *property*, not of the function itself. This can be proven by examining the `name` property of the function. Review the results of listing 3.1 in figure 3.4 to see that anonymous functions do not possess names in the same manner that named function do.

We're going to see anonymous functions a lot in the code throughout the rest of this book because prowess with JavaScript directly relies upon using it as a functional language. As such we're going to be using functional programming styles heavily in all the code that follows. Functional programming concentrates on small, usually side-effect-free functions as the basic building blocks of application code. As we go along, we're going to see that this style is essential to the types of things we need to do in web applications.

So in addition to not polluting the global namespace with unnecessary function names, we're going to be creating lots of little functions that get passed around versus large function full of imperative statements.

Functional programming with anonymous functions is 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 various ways in which they can be employed, and we'll start with recursion.

4.2 Recursion

Rcursion is a really useful technique for applications of all types. You might be thinking that recursion is mostly useful in applications that do a lot of math; and that's true – many mathematical formulae are recursive in nature. But it's also useful for doing things like walking trees, and that's a construct that, including the DOM itself, that we're likely to see popping up within web applications.

So recursion is a concept that we'll llkely find useful, and that we can use to develop an even deeper understanding of how functions work within JavaScript.

Recursion is a concept that you've probably run into before. Whenever a function calls itself, or calls a function that in turn calls the original function anywhere in the call tree, recursion occurs.

Let's start by using recursion in its simplest form.

4.2.1 Recursion in named functions

There are any numbers of common examples for recursive functions, and one of those examples is the test for a palindrome – perhaps the "Hello world!" for recursive techniques – which is a phrase that reads the same way in either direction.

That's a non-recursive definition of a palindrome, and we can use it to implement a function that creates a reversed copy of the string and compare it to the original.

But that's not an elegant solution on a number of levels, not the least of which is the need to allocate and create a new String. By using a more mathematical definition of a palindrome, we can come up with a more elegant solution. That definition is:

1. A single or zero-character string is a palindrome.
2. Any other string is a palindrome if the first and last characters are the same, and the remaining string is a palindrome.

Our implementation using this definition:

```
function isPalindrome(text) {
```

```
  if (text.length <= 1) return true;
  if (text.charAt(0) != text.charAt(text.length - 1)) return false;
  return isPalindrome(text.substr(1,text.length - 2));
}
```

Note that the new definition, and our implementation of it, is *recursive* as it uses the definition of a palindrome to determine if a string is a palindrome.

The implementation is straightforward, and we can see that we make a recursive call, using the function's name, in the last line of the function.

Things get a bit more interesting, and a tad less clear, when we begin dealing with anonymous functions, but we'll get to that in a bit. Let's establish a really simple example of recursion that we'll build upon as we go along.

Ninjas frequently need to signal each other in code, often employing natural sounds as a cover. We're going to give out ninja the ability to chirp like a cricket; with the number of chirps used to encode different messages. We'll start with an implementation using the function's name as shown in the code of listing 4.2.

Listing 4.2: Chirping usinfg a named function

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

  function chirp(n) {                                          #1
    return n > 1 ? chirp(n - 1) + "-chirp" : "chirp";          #1
  }                                                            #1

  assert(chirp(3) == "chirp-chirp-chirp",                      #2
         "Calling the named function comes naturally.");       #2

</script>
```

#1 Declares a recursive chirping function that calls itself by name until it determines that it's done.
#2 Asserts that the ninja chirps as planned.

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

About recursion

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 of one or less 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?

4.2.2 Recursion with object methods

In the previous section we said that we were going to give our ninja the ability to chirp; but we really didn't. What we created was a standalone function for chirping.

Let's fix that by declaring the recursive function as a method of a `ninja` object. This complicates things a bit as the recursive function becomes an anonymous function assigned to an object's property as shown in listing 4.3.

Listing 4.3: Method recursion within an object

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

  var ninja = {
    chirp: function(n) {                                          #1
      return n > 1 ? ninja.chirp(n - 1) + "-chirp" : "chirp";
    }
  };

  assert(ninja.chirp(3) == "chirp-chirp-chirp",
        "An object property isn't too confusing, either.");

</script>
```

#1 Declares the recursive chirp function as a property of the ninja object. We now need to call the method from within itself using the reference to the object's method.

In this test, we defined our recursive function as an anonymous function referenced by the `chirp` property of the `ninja` object (#1). Within the function, we invoke the function recursively via a reference to the object's property: `ninja.chirp()`. We can't reference it directly by its name as we did in listing 4.2, because it doesn't have one!

That's all fine as it stands, but because we're relying upon an *indirect* reference to the function – namely, the `chirp` property – we could be standing on thin ice – not a wise move for a ninja of any standing. Let's see how.

4.2.3 The pilfered reference problem

The example of listing 4.3 relied on the fact that we had a reference to the function to be called recursively in the property of an object. But unlike a function's actual name, such references may be transient, and relying upon them can trip us up in confounding ways.

Let's modify the previous example by adding a new object, let's say `samurai`, that also references the anonymous recursive function in the `ninja` object. Consider the code of listing 4.4.

Listing 4.4 Recursion using a missing function reference

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

  var ninja = {
    chirp: function(n) {
```

```
      return n > 1 ? ninja.chirp(n - 1) + "-chirp" : "chirp";
    }
  };

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

  ninja = {};                                              #2

  try {
    assert(samurai.chirp(3) == "chirp-chirp-chirp",        #3
           "Is this going to work?");
  }
  catch(e){
    assert(false,
           "Uh, this isn't good! Where'd ninja.chirp go?");
  }

</script>
```

#1 Creates a `chirp()` method on `samurai` by referencing the existing method of the same name on ninja. Why write the code twice when we already have an implementation?
#2 Redefines `ninja` such that it has no properties. This means that its `chirp()` method goes away!
#3 Tests if things still work. Hint: they don't!

We can see how things can quickly break down in this scenario. We copied a reference to the chirping function into the `samurai` object (#1). So now, both `ninja.yell` and `samurai.yell` reference the same anonymous function. A diagram of the relationsips created is shown in figure 4.2.

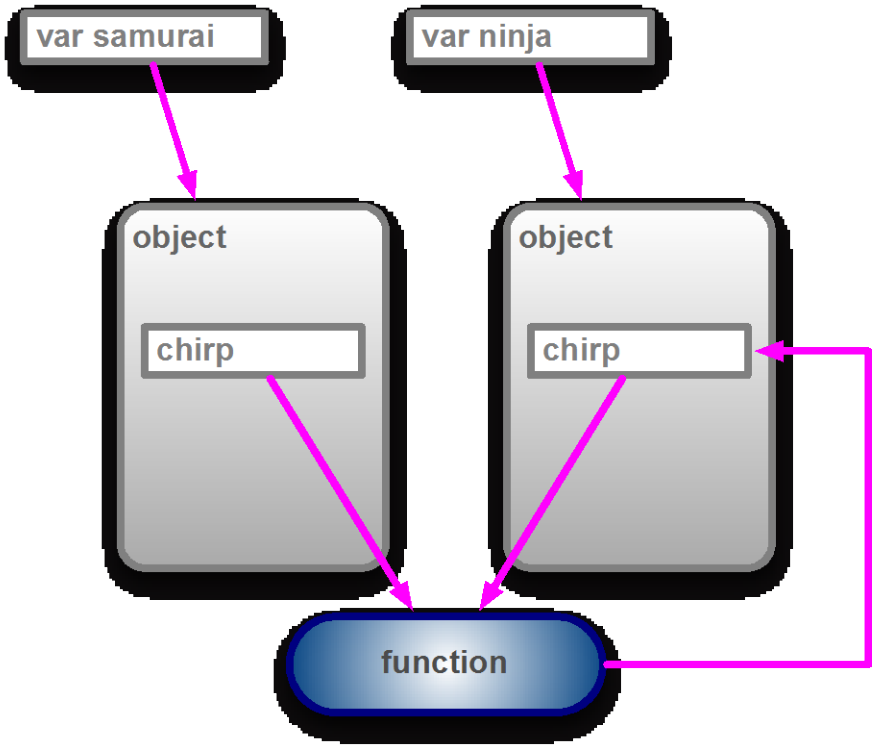

Figure 4.2 The two objects have a reference to the same function, but the function refers to itself through only one of the objects

That's generally not a problem – it's not at all uncommon for functions to be referenced from multiple places. The potential booby trap is that the function is recursive and uses the `ninja.chirp` reference to call itself, regardless of whether the function is invoked as a method of `ninja` or of `samurai`.

So what happens if `ninja` were to go away, leaving `samurai` holding the bag?

To test this, we redefine `ninja` with an empty object (#2). The anonymous function still exists, and can be referenced through the `samurai.chirp` property, but the `ninja.chirp` property no longer exists. And because the function recursively calls itself through that now-defunct 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 = {
```

```
  chirp: function(n) {
    return n > 1 ? this.chirp(n - 1) + "-chirp" : "chirp";
  }
};
```

Remember that when a function is invoked as a method, the function context refers to the object through which the method was invoked. So when invoked as `ninja.chirp()`, `this` refers to `ninja`, but when invoked by `samurai.chirp()`, `this` refers to `samurai` and all is well.

Using the function context (`this`) makes our `chirp()` method much more robust, and is the way that the method should have been declared in the first place.

So, problem solved.

But…

4.2.4 Inline named functions

The solution we came up in the previous section worked perfectly fine when functions are used as methods of an object. In fact, the technique of using the function context, regardless of whether the method is recursive or not, to reference the "owning object" of the method is one that's very common and accepted. And we'll be seeing a lot more about that in chapter 6.

But for now, we have another problem. The solution relied upon the fact that the function would be a method named `chirp()` of any object within which the method is defined..What if the properties don't have the same name? Or what if one of the references to the function isn't even an object property? Our solution only works in the specific case where the function is used as a method, and that the property named of the method is identical in all its uses.

Can we develop a more general technique? Let's consider another approach: what if we give the anonymous function a *name*?

At first, this may seem completely crazy; if we're going to use a function as a method, why would we also give it its own name? Well, remember that when declaring a function literal, the name of the function is optional, and we've been leaving it off for all but top-level functions. But as it turns out, there's nothing wrong with giving *any* function literal a name, even those that are declared as callbacks or methods.

No longer *anonymous*, these functions are better called ***inline functions***, rather than "anonymous named functions" to avoid the oxymoron.

Observe the use of this technique in Listing 4.5:

Listing 4.5: Using a inline function in a recursive fashion

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

  var ninja = {
    chirp: function signal(n) {                              //#1
      return n > 1 ? signal(n - 1) + "-chirp" : "chirp";
    }
  };
```

```
  assert(ninja.chirp(3) == "chirp-chirp-chirp",              //#2
         "Works as we would expect it to!");

  var samurai = { chirp: ninja.chirp };                      //#3

  ninja = {};                                                //#4

  assert(samurai.chirp(3) == "chirp-chirp-chirp",            //#5
         "The method correctly calls itself.");

</script>
```

#1 Declares the named inline function.
#2 Test that it works just like we expect it to.
#3 Wipes out the ninja object just like in the previous example.
#4 Tests that it still works. And it does!

Here (#1) we assign the name `signal` 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.chirp`, 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 `chirp` 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, as shown in Listing 4.6:

Listing 4.6: Verifying the identity of an inline 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 and assigns it to a variable.
#2 Tests that the tow names are equivalent inside the inline function.
#3 Invokes function to perform the internal test.
#4 Tests that the inline function's name isn't available outside the inline function.

This listing brings up the most important point regarding inline functions: even though inline functions can be named, those names are *only* visible within the functions themselves. Remember the scoping rules we talked about earlier in the chapter? Inline function names

act somewhat like variable names, and their scope is limited to the function within which they are declared.

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

Are there other techniques we can employ?

4.2.5 The `callee` property

Let's look at still another way to approach recursion that introduces another concept concerning functions: the `callee` property of the `arguments` parameter.

> **WARNING** The `callee` property may be on the chopping block for an upcoming version of JavaScript. The ECMAscript 5 standard forbids its use in "strict" mode. So while it's OK to use this property in current browsers, its use is not a future-proof and we'd likely not want to use `callee` unless there are no alternatives. Nevertheless, we present it here as you may come across it in existing ocde.

Consider the code of listing 4.7.

Listing 4.7: Using `arguments.callee` to reference the calling function

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

  var ninja = {
    chirp: function(n) {                                  //#1
      return n > 1 ? arguments.callee(n - 1) + "-chirp" : "chirp";
    }
  };

  assert(ninja.chirp(3) == "chirp-chirp-chirp",       //#2
         "arguments.callee is the function itself.");
</script>
```

#1 References the `arguments.callee` property

As we discovered in section 3.3, the `arguments` parameter is implicitly passed to every function and `arguments` has a property named `callee` that refers to the currently executing function. This property can serve as a reliable way to always access the function itself. Later on in this chapter, as well as in the following chapter (chapter 5, on closures), we'll take a closer look at what can be done with this particular property.

All together, these different techniques for referencing 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 in understanding how the object-oriented nature of functions in JavaScript can help take our code to the next level.

4.3 Fun with function as objects

As we've consistently harped on throughout this chapter, functions in JavaScript are not like functions in many other languages. JavaScript gives functions many capabilities, not the least of which is their treatment as first-class objects.

We've seen that functions can have properties, can have methods, can be assigned to variables and properties, and generally enjoy all the abilities of plain vanilla objects, but with an amazing super-power: they are callable.

In this section, we'll examine some ways that we can exploit the similarities that functions share with other object types.

But to start with, let's recap a few key concepts that we're going to take advantage of. Let's start with assigning 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 in order to create methods.

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, having properly placed semicolons will allow for greater flexibility in compression techniques.

Another capability that may have surprised a few people is that, just like any other object, we can attach properties to a function, as in:

```
var obj = {};
var fn = function(){};
obj.prop = "hitsuke (distraction)";
fn.prop = "tanuki (climbing)";
```

This aspect of functions can be used in a number of different ways throughout a library or general on-page code. This is especially true when it comes to topics like event callback management. So, let's look at a couple of the more interesting things that can be done with this capability; first we'll look at storing functions in collections, and then a technique known as "memoizing".

4.3.1 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 (and one that we'll be examining in excruciating detail in chapter 13). When adding functions to such a collection, a challenge we can face is determining which functions are actually new to the collection and should be added, and which are 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 as ninjas, we want to make things work *well*, not merely work.

So, we can make use of function properties to achieve this with an appropriate level of sophistication, as shown in Listing 4.8.

Listing 4.8: 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 4.8, 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 4.2.

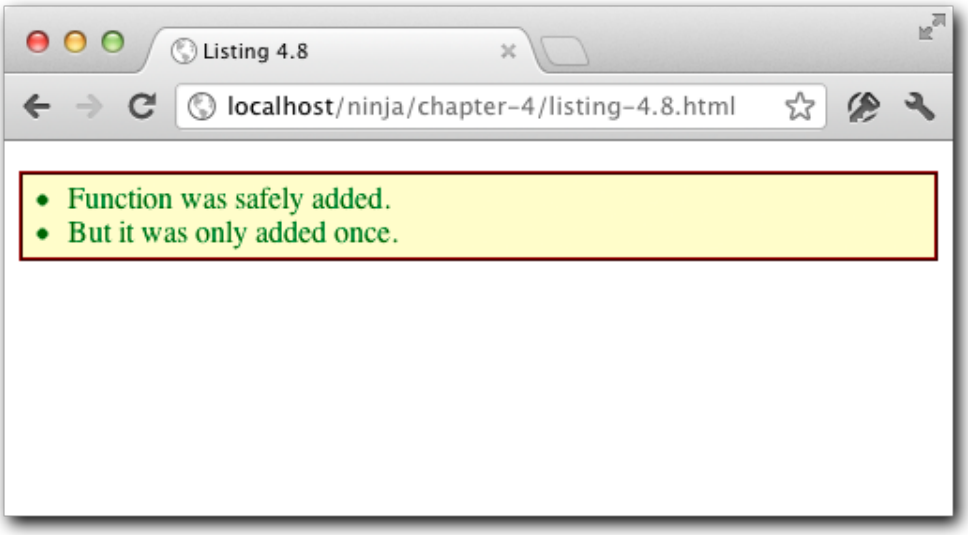

Figure 4.2 By tacking a property onto a function, we can keep track of it

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.

4.3.2 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. This can markedly increase performance by avoiding needless complex computations that have already been performed once.

We'll take a look at this technique in the context of storing the answer to expensive computations, and then with a more real-world example of storing a list of DOM elements that we've looked up.

MEMOIZING EXPENSIVE COMPUTATIONS

As a basic example, let's look at a simplistic algorithm for computing prime numbers. While this is just a simple example of a complex calculation, this technique is readily applicable to other expensive computations, such as deriving the MD5 hash for a string, that are too complex to present as an example here.

From the outside, the function will appear to be just like any normal function, but we'll surreptitiously build in an "answer cache" in which it will save the answers to the computations it performs. Look over the code of Listing 4.9.

Listing 4.9: Memoizing previously-computed values

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

  function isPrime(value) {
    if (!isPrime.anwers) isPrime.answers = {};                    #1
    if (isPrime.answers[value] != null) {                         #2
      return isPrime.answers[value];                              #2
    }                                                             #2
    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;                        #3
  }

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

</script>
```

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

Within the `isPrime()` function, we start by checking to see if the `answers` property that we'll use as a cache has been created, and if not, we create it (#1). The creation of this initially empty object will only occur on the first call to the function; after that, the cache will exist.

Then we check to see if the answer for the passed value has already been cached in `answers` (#2). Within this cache 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 (#3).

Some simple tests (#4) show that the memoization is working!

This approach has two major advantages:

- First, the end user enjoys performance benefits for function calls asking for a previously computed value.
- Secondly, it happens completely seamlessly and behind the scenes; neither the user or the page author 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 kindred example.

MEMOIZING DOM ELEMENTS

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 within which we can store the matched element sets. 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, yielding the results shown in Table 4.1, we find that this simple layer of caching yields us a 5x performance increase! Not a bad superpower to have.

Table 4.1: All time in ms, for 100,000 iterations, in a copy of Chrome 17

	Average	Min	Max	Runs
non-cached version	16.7	18	19	10
cached version	3.2	3	4	10

Even these simple examples give evidence to the usefulness of function properties: we can store state and cache information in a single and encapsulated location, gaining not only organizational advantages, but performance benefits without external storage or caching objects polluting the scope. We'll be revisiting this concept throughout the upcoming chapters, as the utility of this technique is broadly applicable.

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, an example of which we'll explore next.

4.3.3 Faking array methods

There are times that we may want to create an object that contains a collection of data. If the collection was all that we were worried about, we could just use an array. But in certain cases, there may be more state to store than just the collection itself – perhaps some sort of metadata regarding the collected items.

One option might be to create a new array every time you wish to create a new version of such an object and add our metadata properties and methods to it – remember we can add properties and methods to an object as we please. Generally, however, this can be quite slow, not to mention tedious.

Let's examine the possibility of using a normal object and just *giving* it the functionality that we desire. Methods that know how to deal with collections already exist on the Array object; can we trick them into working on our own objects?

Turns out that we can! For an example, take a look at the code of Listing 4.10:

Listing 4.10: Simulating array-like methods

```
<body>

  <input id="first"/>
  <input id="second"/>

  <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>
</body>
```

#1 Stores the count of elements. If we're going to pretend we're an array, we're going to need someplace to store the number of items that we're storing.
#2 Implements the method to add elements to our collection. The prototype for the Array class already has a method to do this, why not use it instead of re0nventing the wheel?
#3 Implements a method to find elements by their `id` values and add them to our collection.
#4 Tests the `find()` and `add()` method.

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.

Then 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`. (Don't worry about the `prototype` part of that reference, we'll be finding out all about that in chapter 6).

Normally, the `Array.prototype.push()` method would operate on its own array via its function context. 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. This method, which increments the `length` property (thinking that it's the `length` property of an array), adds a numbered property to the object referencing the passed element.

This behavior is almost subversive in a way (how fitting for ninjas!), 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).

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.

4.4 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 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 learn:

- 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
- How to understand and use the `length` property of argument lists.

Let's start with using `apply()` to hand off a variable number of arguments.

4.4.1 Using `apply()` to supply variable arguments

With any language, there are often things that we need to do that seem to have been mysteriously overlooked by the developers of the language; and JavaScript is not exception. One of these odd vacuums involves finding the smallest or the largest values contained within an array. Seems like that'd be something done often enough to warrant inclusion in JavaScript, but if we poke around, the closest thing we find is a set of methods on the `Math` object named `min()` and `max()`.

At first we might think that these methods are the answer to our problem, but on examination, we see that each of these methods expects a variable length argument list, and not an array. How silly not to have provided both.

So calls to `Math.max()`, for example, could look like:

```
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);
```

We can't very well resort to something like:

```
var biggest = Math.max(list[0],list[1],list[2]);
```

Unless we knew exactly how big the array would be, how would we know how many arguments to pass? And even if we did, that's far from a satisfactory solution.

Before abandoning `Match.max()` and resorting to looping through the contents ourselves to find min and max values, let's pull on our ninja hoods ponder if there is an easy and supported way to use an array as a variable length argument list.

Eureka! The `apply()` method!

You may recall, the `call()` and `apply()` methods exist as methods of *all* functions, even of the built-in JavaScript functions (we saw this with our "fake array" example).

Let's see how we can use that ability to our advantage in defining our array-inspecting functions. Consider listing 4.11.

Listing 4.11: 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!

4.4.2 Function overloading

Back in section 3.2, we introduced the built-in `arguments` parameter that is implicitly passed to all functions. We're now ready to take a closer look at that parameter.

All functions are implicitly passed this important parameter, which will give our functions the power to handle any number of passed arguments. Even if we only define a certain number of parameters, we'll always be able to access *all* passed arguments through the `arguments` parameter.

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 but differing parameter lists. That's not how it's done in JavaScript. In JavaScript, we "overload" functions with a single implementation that modifies its behavior by inspecting the number and nature of the passed arguments. Let's see how that can be done.

In the following code, we're going to merge the properties of multiple objects into a single root object. This can be an essential utility for effecting inheritance (which we'll discuss more when we talk about object prototypes in chapter 6).

Take a look at listing 4.12:

Listing 4.12: 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 arguments 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 situation with: `paramname === undefined`.

So we can get at the first passed argument via `root`, but how do we access the rest of any arguments that may have been passed? Why, the `arguments` parameter, of course, which references a collection of all of the passed arguments.

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 arguments 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 any 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.

We should be seeing, at this point, that the ability to access and traverse the `arguments` collection is a powerful mechanism for creating complex and intelligent methods. We can use it to inspect the arguments passed to any function in order to allow our function to flexibly operate on the arguments even when we don't know in advance exactly what is going to be passed.

Libraries such as jQuery UI use function overloading extensively. Consider a method to create and manage a UI widget such as a floating dialog. The same method, `dialog()`, is used to both create and to perform operations on the dialog. To create the dialog a call such as the following is made:

```
$("#myDialog").dialog({ caption: "This is a dialog" });
```

The exact same method is used to perform operations, such as opening the dialog:

```
$("#myDialog").dialog("open");
```

What the `dialog()` method actually does is determined by an inspection of exactly what is being passed to it.

Let's take a look at another example where the use of the `arguments` parameters isn't as clear-cut as in the example of listing 4.12..

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. This probably isn't something that's specifically applicable in our applications, but a simple example of yet more techniques of dealing with arguments within a function.

This might 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 4.13.

Listing 4.13: 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 4.3.

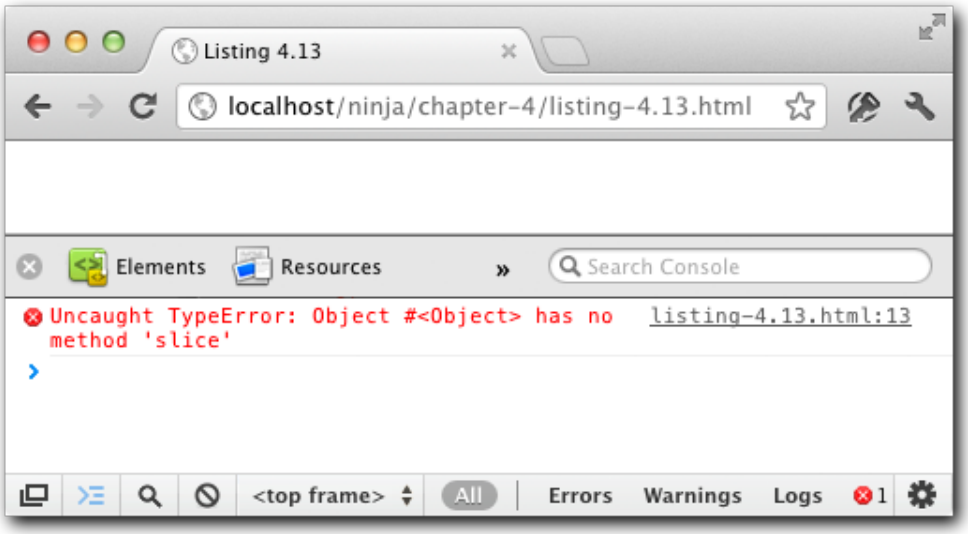

Figure 4.3 Something's rotten in the state of Denmark, and with our code!

What's up with that? Apparently it wasn't as simple as we had first thought.

Well, as we had pointed out earlier in the chapter, the `arguments` parameter doesn't reference a true array. Even though 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 the very handy `slice()`.

Well, we could create our own sets of array slice-and-dice methods; a hand-built *Argu-matic* utensil, if you will. Or, we could create our own array by copying the values into a *true* array. But either of these approaches seems ham-handed and redundant when we know that Array already has the functionality we seek.

So before we resort to copying the data or creating the *Argu-matic*, recall the lesson of listing 4.10 in which we fooled an Array function into treating a non-array as an array. Let's use that knowledge and rewrite the code as shown in listing 4.14.

Listing 4.14: 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 3.15 to coerce the Array's `slice()` method into treating the `arguments` "array" as a true array, even if it isn't one.

Now that we've learned a bit regarding how to deal with the `arguments` parameter a bit, let's look at some techniques for overloading functions based upon what we find there.

FUNCTION OVERLOADING APPROACHES

When it comes to function 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 simpler side.

But once things start getting a bit more complicated, lengthy functions using many such clauses can quickly become unwieldy. In the remainder of 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` parameter, equates to the number of named parameters with which the function was declared. Thus, if we declare a function with a single formal parameter, its `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 build a function that we can use to create overloaded functions, differentiated by argument count.

OVERLOADING FUNCTIONS BY ARGUMENT COUNT

There are any number of ways that we can decide to overload what a function does based upon its arguments. One common approach is to perform different operations based upon the type of the passed arguments. Another could be switching based upon whether certain parameters are present or absent. And still another is based upon the count of the passed arguments.

Let's take an example were we wanted to have a method on an object that performs different operations based upon argument count. If we wanted to have long, monolithic functions, we could simply do something like the following:

```
var ninja = {
  whatever: function() {
    switch (arguments.length) {
      case 0:
        /* do something */
        break;
      case 1:
        /* do something else */
        break;
      case 2:
        /* do yet something else */
        break;
    //and so on…
    }
  }
}
```

In this approach, each case would perform a different operation based upon the argument count, obtaining the actual arguments through the `arguments` parameter.

But that's not very tidy, and certainly not very ninja, is it?

Let's posit another approach, what if we wanted to add the overloaded method using syntax along the lines of the following:

```
var ninja = {};
addMethod(ninja,'whatever',function(){ /* do something */ });
addMethod(ninja,'whatever',function(a){ /* do something else */ });
addMethod(ninja,'whatever',function(a,b){ /* yet something else */ });
```

Here, we create the object and then add methods to it using the same name (whatever), but with separate functions for each overload. Note how each overload has a different number of parameters specified.

This way, we actually create a separate anonymous function for each overload. Nice and tidy!

However, such an `addMethod()` function doesn't exist, so we'll need to create it ourselves.

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

Let's look at Listing 4.15.

Listing 4.15 A method overloading function

```
function addMethod(object, name, fn) {
  var old = object[name];                          #1
  object[name] = function(){                       #2
    if (fn.length == arguments.length)             #3
      return fn.apply(this, arguments)             #3
    else if (typeof old == 'function')             #4
      return old.apply(this, arguments);           #4
  };
}
```

#1 Stores the previous function as we may need to call it if the passed function doesn't have the matching number of arguments.
#2 Creates a new anonymous function that becomes the method.
#2 Invokes the passed function if its parameter and argument count match.
#3 Invokes the previous function if the passed function is not a match,

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.

Look again at our usage example:

```
var ninja = {};
addMethod(ninja,'whatever',function(){ /* do something */ });
addMethod(ninja,'whatever',function(a){ /* do something else */ });
addMethod(ninja,'whatever',function(a,b){ /* yet something else */ });
```

The first call to `addMethod()` will create a new anonymous function that, when called with a zero-length argument list, will call the passed fn function. Because `ninja` is a new object at this point, there's no previously established method to worry about.

But on the *next* call to `addMethod()`, we store a reference to the anonymous function that we created in the previous invocation in the variable `old` (#1), and proceed to create another anonymous function that becomes the method (#2). This newer method will check to see if the number of passed arguments is one, and if so, invoke the function passed as `fn` (#2). Failing that, it will call the function stored in `old` (#3), which as you will recall, will check for zero parameters, and call the version of `fn` with zero parameters.

On the third call to `addMethod()`, we pass an `fn` that takes two parameters. And we go through the process again, creating yet another anonymous function which becomes the method, and calls the 2-parameter `fn` when 2 arguments are passed and deferring to the previously created 1-argument function.

It's almost as if we're winding the function around each other like the layers of an onion, each layer checking for a matching number of arguments, and deferring to a previously created layer if no match is found.

There's a bit of sleight of hand going on here with regard to how the inner anonymous 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 the current values of `old` and `fn`.

Let's test our new function as shown in Listing 4.16:

Listing 4.16 Testing 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 an object to serve as the base, pre-loaded with some test data.
#2 Binds a no-argument method to the base object.
#3 Binds a single-argument method to the base object.
#4 Binds a dual-argument method to the base object.
#5 Tests the bound methods.

Loading this page to run the test shows that all succeed as shown in figure 4.4.

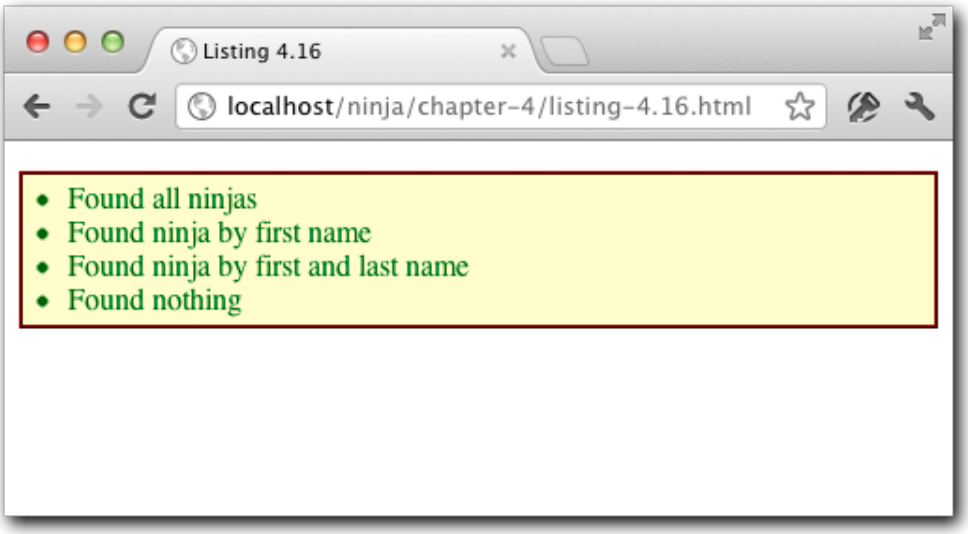

Figure 4.4 Ninjas found! All using the same overloaded method name `find()`

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.

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 much more about closures 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 exactly what we'll be wanting 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 if an object is a function.

4.5 Checking for functions

To close out our look at functions in JavaScript, let's take a look at how we can detect when a particular object is an instance of a function, and therefore something that can be called; a seemingly simple task, but not without its cross-browser issues.

Typically the `typeof` statement is more than sufficient to get the job done; 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: Doing a `typeof` on the HTML `<object>` element yields an inaccurate "function" result, instead of "object" as we might expect..
- 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.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 test 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. Writing code that works seamlessly in multiple browsers can be quite challenging, but it's a necessary skill for anyone who wants to write code that's robust and workable on the web. We'll explore lots more cross-browser strategies as we go along, and it will be the focus of chapter 11.

4.6 Summary

In this chapter we took the knowledge that we gained in chapter 3, and wielded it to solve a number of problems we're likely to find in applications.

In particular:

- Anonymous functions let us create smaller units of execution rather than large functions full of imperative statements.
- Looking at recursive functions, we learned how functions can be referenced in various ways, including:
    - By name
    - As a method (via an object property name)
    - By an inline name
    - Through the `callee` property of `arguments`
- Functions can have properties and those properties can be used to store any information we might wish to use, including

    - Storing functions in function properties for later reference and invocation.
    - Using function properties to create a cache (memoization).
- By controlling what function context is passed to a function invocation, we can "fool" methods into operating on objects that aren't the object that they are methods for. This can be useful for leveraging already existing methods on objects like Array and Math to operate on our own data.
- Functions can perform differing operations based upon the arguments that are passed to it (function overloading). We can inspect the `arguments` list top determine what it is we'd like to do given the type or number of the passed arguments.
- An object can be checked to see if it is an instance of a function by testing if the result of the `typeof` operator is "function". This is not without its cross-browser issues.

One of our examples, listing 4.15 to be precise, made heavy use of a concept known as a closure, that controls what data values are available to a function while it is executing. Let's spend a chapter taking a closer look at this essential concept.

5 Closing in on closures

In this chapter:

- What are closures and how do they work?
- Using closures to simplify development
- Improving performance 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 understanding 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, they 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 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.

5.1 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 that function. Put another way, closures allow a function to access all the variables, as well as other functions, that are declared in the same scope within which 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 5.1.

Listing 5.1: A simple closure

```
<!DOCTYPE html>
<html>
  <head>
    <title>Listing 5.1</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 5.1, the function as able to "see" and access the `outerValue` variable.

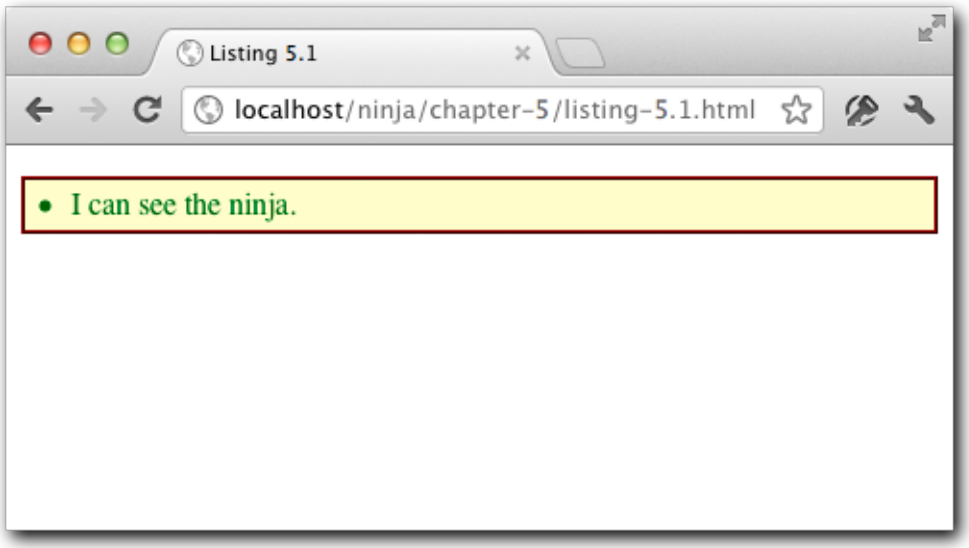

Figure 5.1: Our function has found the ninja, who was hiding in plain sight

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 5.2.

Listing 5.2: 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 named `inside` to the outer function (#2). This limits the scope of `inside` to the body of `outerFunction`.

We then declare a new function inside the outer function (#3) which accesses both of the variables that we've declared. Yes, we can do that. Remember from the previous chapter 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 5.2.

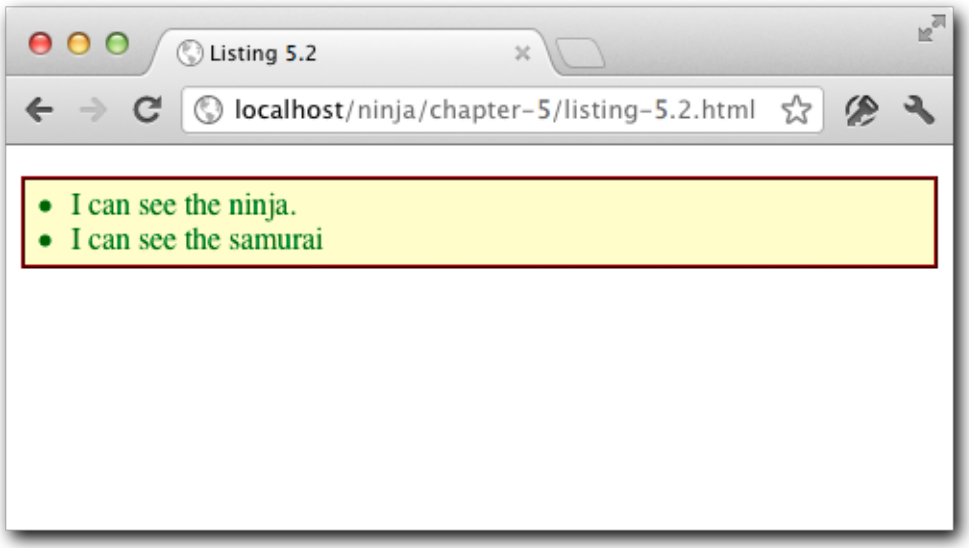

Figure 5.2: Despite trying to hide inside a function, the samurai has been spied!

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 created that encompasses not only 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 5.3.

Listing 5.3 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` (#5). We also added a variable that declared after the outer function declaration (#4).

When the tests inside the inner function execute (#2), we can see the display of figure 5.3.

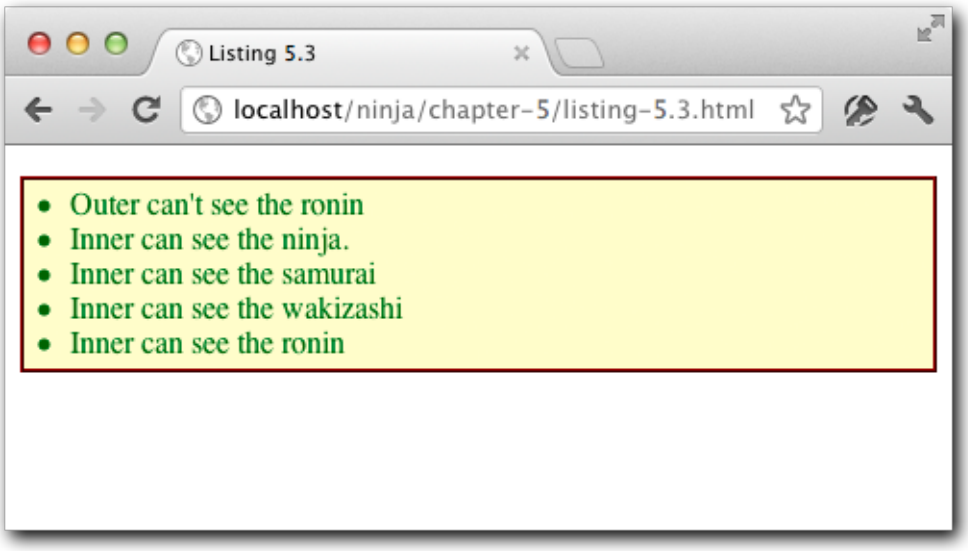

Figure 5.3: Turns out that inner can see farther than outer

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 structure 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 (and is safe to garbage collect), or the page unloads.

5.2 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.

5.2.1 Private variables

A common use of closures is to encapsulate some information as a "private variable" of sorts – in other words, to limit their scope. Object-oriented code written in JavaScript is unable to use traditional private variables: properties of the object that are hidden from outside parties. However, using the concept of a closure, we can achieve an acceptable approximation, as demonstrated by the code of listing 5.4.

Listing 5.4: 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 5.4 we create a function that is to serve as a constructor (#1). We introduced the concept of using a function as a constructor in the previous chapter, and we'll be taking an in-depth look at them again in chapter 6. For now, just recall 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 JavaScript scoping rules for this variable limit its accessibility to within the constructor. To give access to the value of the variable from code that is outside the scope, we define an *accessor* method, `getSlices()`, which can be used to read, but not write the private variable. (Accessor methods are frequently called "getters".)

An implementation method, `slice()`, was created 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.

This allows state to be maintained within the function, without letting it be directly accessed by a user of the function, 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.

5.2.2 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 such functions we frequently 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 5.5.

Listing 5.5: Using closures from a callback for an Ajax request

```
<div></div>

<button type="button" id="testButton">Go!</button>

<script>
  jQuery('#testButton').click(function(){                              #1

    var elem$ = jQuery("div");                                         #2

    elem$.html("Loading...");                                          #3

    jQuery.ajax({
      url: "test.html",
      success: function(html){                                         #4
        assert(elem$,
              "We can see elem$, via the closure for this callback.");
        elem$.html(html);
      }
    });

  });
</script>
```

#1 Establishes the click handler
#2 Declares a variable
#3 Pre-loads div with text
#4 Creates a callback and closure

There're a number of interesting things going on in listing 5.5. We start with an empty `<div>` element, which upon the click of a button (#1) we want to load with the text "Loading…" (#3) 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 whenever the response comes back from the server. 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$` (#2) (using the `$` sign as a suffix is a jQuery convention to indicate that the variable holds a jQuery wrapped set reference).

Within the arguments to the jQuery `.ajax()` method, we define an anonymous function (#4) 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 the slightly more complicated example of listing 5.6, which creates a simple animation.

Listing 5.6: Using a closure in a timer interval callback

```
<div id="box"> ボックス</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 5.4 when the animation has completed.

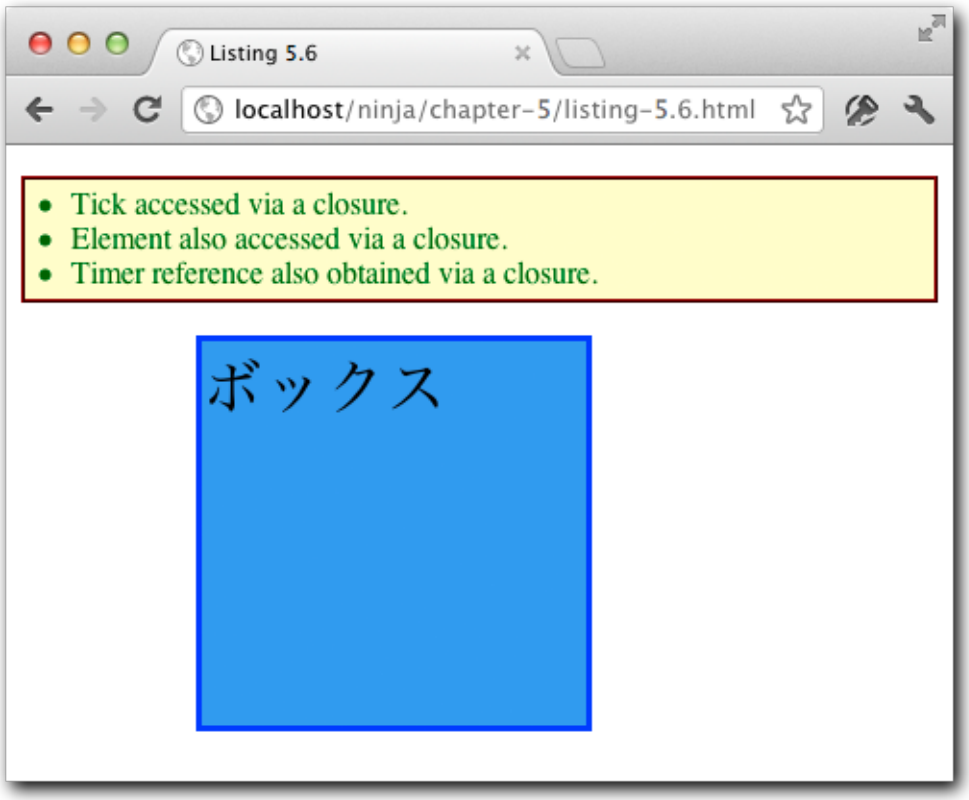

Figure 5.4: Closures used to keep track of the steps of an animation

What's especially interesting about the code of listing 5.4 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. Not only are we seeing the values that these variables had at the time that the closure was created, but we can also update them within the closure as the function within the closure executes. In other words, the closure isn't simply a snapshot of the state of the scope at the time of creation, but an active encapsulation of that state that can be modified as long as the closure exists.

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 wills.

5.3 Binding function contexts

During our discussion of function contexts in the previous chapter, 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 5.7 in which an object method is bound to an element as an event listener.

Listing 5.7: 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" named `button` (#2) in which we will store the clicked state, as well as define a method that will serve as an event handler (#3) that will fire when the button is clicked in order to record the click.

Within the method, 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 5.5 that something is amiss (the stricken text indicates that the test has failed).

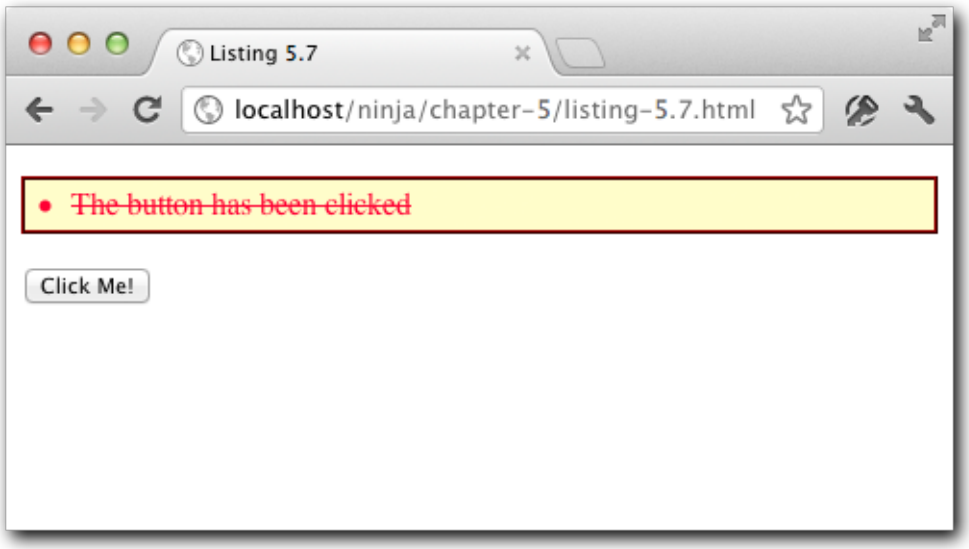

Figure 5.5 Why did our test fail? Where did the change of state go?

The code of listing 5.5 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 the 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 when an event handler is invoked 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 5.8, which updates the code of listing 5.7 with additions to bend the function context to our wills.

Listing 5.8: 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 whatever object that we pass to `bind()` as its first argument. 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 classical 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 5.9.

Listing 5.9: 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 5.8, 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 6, 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 a moment ago?

5.4 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 with predefined arguments, which we can later call.

This technique of filling in the first few 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 5.10.

Listing 5.10: 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 5.10 we've taken the String's `.split()` method and have imagined a `partial()` method (yet to be implemented as listing 5.12) 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 partial/curry method is, more or less, implemented in the Prototype library in listing 5.11.

Listing 5.11: 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 Remembers "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 5.12 shows a possible implementation (and this is the implementation that we used to make listing 5.10 work).

Listing 5.12: 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 has the ability 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.

5.5 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 memoization from chapter 4? Let's take another look.

5.5.1 Memoization

As we learned in chapter 4, 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 straight-forward 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, that we'll name `memoized()`, and shown in Listing 5.13, that we can use to remember return values from an existing function. This implementation does not involve closures.

Listing 5.13: 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, and 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 this approach, 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()`. The result is shown in Listing 5.14.

Listing 5.14: 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 5.14 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 access to the memorized instance of the function.

In Listing 5.14 we also show a comparatively strange means of defining a new function when we define `isPrime()`. As we want `isPrime` to always be memoized, 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 5.5.3. 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 5.14 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.

5.5.2 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 accessing 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 5.15.

Listing 5.15: 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. It 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 5.3, 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.

5.6 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 contents 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 first set of parentheses will be expected to be a reference to a function to be executed. So the following is also valid:

```
(functionName)()
```

Although the first set of parentheses is not needed in this case, the syntax is perfectly valid.

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.

5.6.1 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 are 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 still 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 5.16 has a quick example from the Prototype JavaScript library:

Listing 5.16: 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.

Something we'll be doing without further delay.

5.6.2 Loops

Another very useful application of immediate functions is the ability to solve a nasty issue with loops and closures. Listing 5.17 shows a common piece of problematic code:

Listing 5.17: 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 5.6.

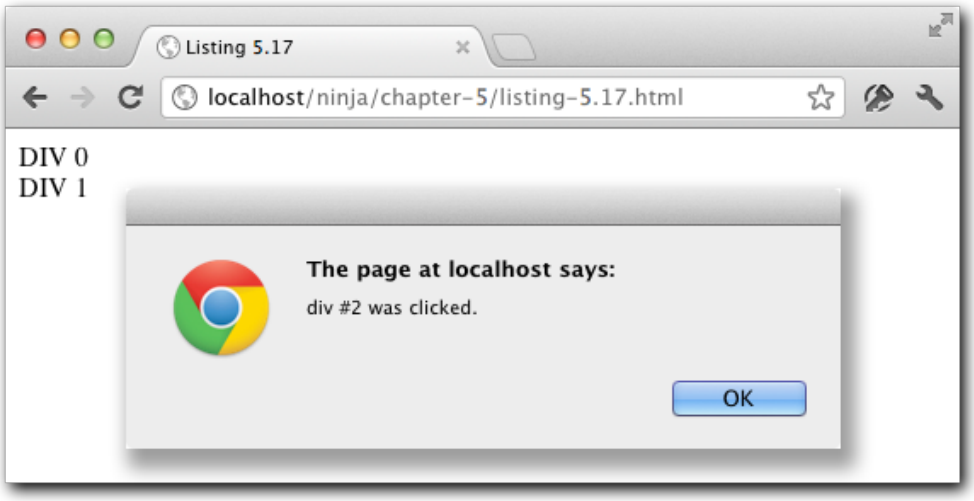

Figure 5.6: Where did we go wrong? Why does DIV 0 think it's 2?

In listing 5.19 we encounter a common issue with closures and looping; namely that the variable that's being closured (`i` in this case) 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 we discovered in section 5.2.2: closures remember *references* to included variables – *not* just their actual values at the time at which they are created. 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) and immediate functions, as shown in Listing 5.18 (changes noted in bold).

Listing 5.18: Using an immediate function to handle 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 expect.

Running the updated page shows the expected display of figure 5.7.

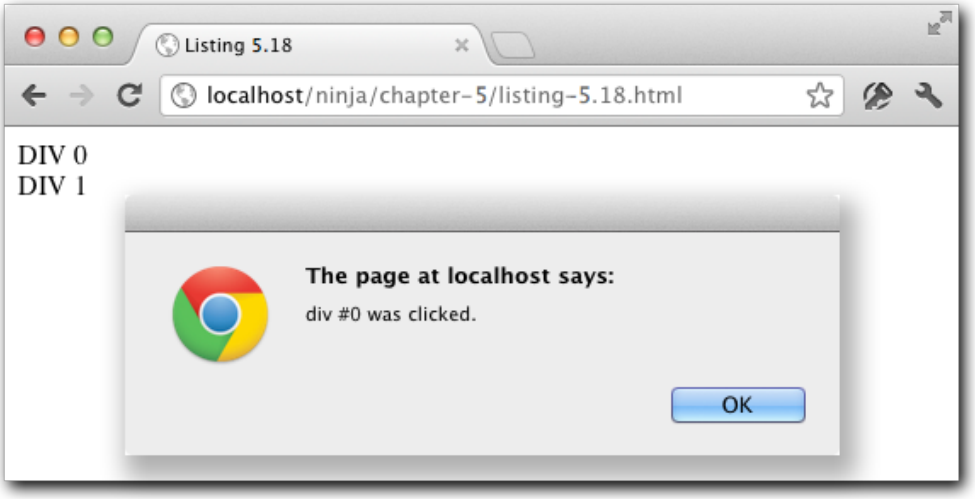

Figure 5.7 That's more like it! Each element now knows its own ordinal.

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.

5.6.3 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 here:

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

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

Note that there is a double assignment performed, completely intentionally. First, the jQuery constructor (as an anonymous 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.

Because 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 here:

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

  // ...

  return jQuery;
})();
```

This code has the same effect as that shown previously, just 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.

5.7 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.

6

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.

Those already somewhat familiar with JavaScript prototypes might think of them as being closely related to objects. But once again it's all about functions. Prototypes are a convenient way to define classes of objects, but they are a feature of functions.

Prototypes are used throughout JavaScript as a convenient means of defining properties and functionality that is to be automatically applied to instances of objects. Once defined, the prototype's properties automatically become properties of instantiated objects, serving as a blueprint of sorts for the creating of complex objects. In other words, they serve a similar function 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.

Let's start exploring how.

6.1 Instantiation and prototypes

All functions have a `prototype` property that initially references an empty object. This property doesn't serve much purpose until the function is used as a ***constructor***. We saw in

chapter 3 that using the `new` keyword to invoke a function calls the function as constructor with a newly instantiated object as its context.

As object instantiation is a large part of what makes constructors useful, let's take a little time at this point to make sure we truly understand it.

6.1.1 Object instantiation

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

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

This creates a new and 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 via constructors, but there is no class definition in JavaScript. Rather, the `new` operator, applied to a constructor function, triggers the creation of a newly allocated object.

PROTOTYPES AS OBJECT BLUEPRINTS

Let's examine a simple case of using a function, both with and without the `new` operator, and see how the `prototype` property adds properties to the new instance. Consider the code of listing 6.1.

Listing 6.1: 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 an empty function
#2 Adds prototyped method
#3 Calls as a function
#4 Calls as a constructor

In this code we define a function named `Ninja()` (#1)that we'll invoke in two ways: as a "normal" function (#3), and as a constructor (#4). After the function is created, we add a method, `swingSword()`, to its prototype (#2). 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 allocated 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. The results of the tests are shown in figure 6.1.

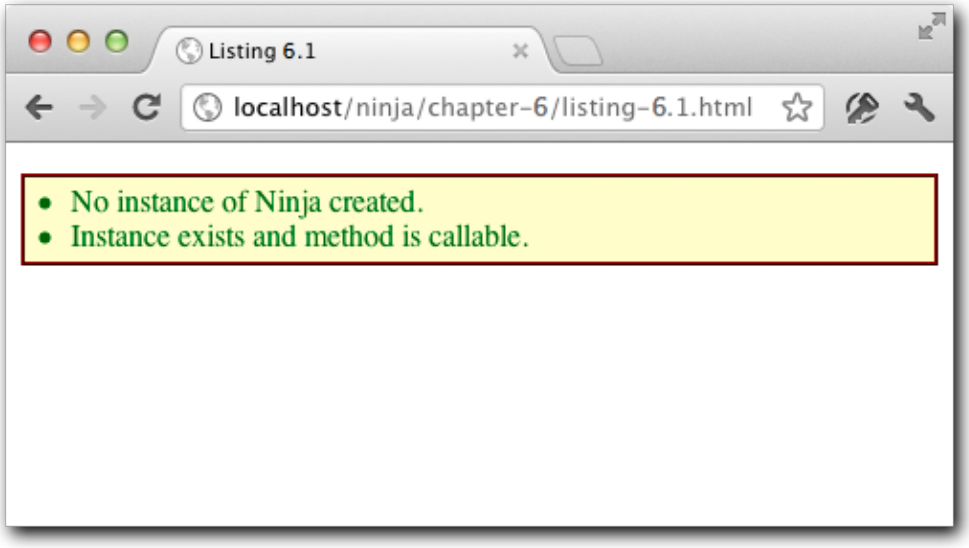

Figure 6.1 A prototype lets us predefine properties, including methods, to be automatically injected into new object instances

INSTANCE PROPERTIES

When the function is called as a constructor via the `new` operator its context 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 creating such instance properties in Listing 6.2.

Listing 6.2: Observing the precedence 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 6.2 is very similar to the previous example in that we define a method by adding it to the `prototype` property (#3) of the constructor. 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.

NOTE

This isn't anything we'd actually advise doing in real-world code. We are doing it here just to demonstrate precedence of initializers.

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 the `swingSword()` method returns `true` as the binding operations within the constructor always take precedence over those in the prototype. 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.

RECONCILING REFERENCES

A vitally important concept to understand about prototypes is how JavaScript goes about reconciling references, and how the prototype comes into play during this process.

The previous examples may have led you to believe that, when a new object is created and passed to a constructor, that the properties of the constructor's prototype are copied to

the object. That would certainly account for the fact that a property assigned within the constructor body overrides the prototype value. But as it turns out, we can observe some behaviors that don't make sense if a simple copy was what was going on.

For example, if the prototype values were simply copied, any change to the prototype made *after* the object was constructed would not be reflected in the object, right? So, let's rearrange the code a bit and see what happens, as shown in Listing 6.3.

Listing 6.3: 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). Then we run a test to see if the change we made to the prototype after the object was constructed takes effect,

Our test (#4) succeeds, showing that the assertion is true as shown in figure 6.2.

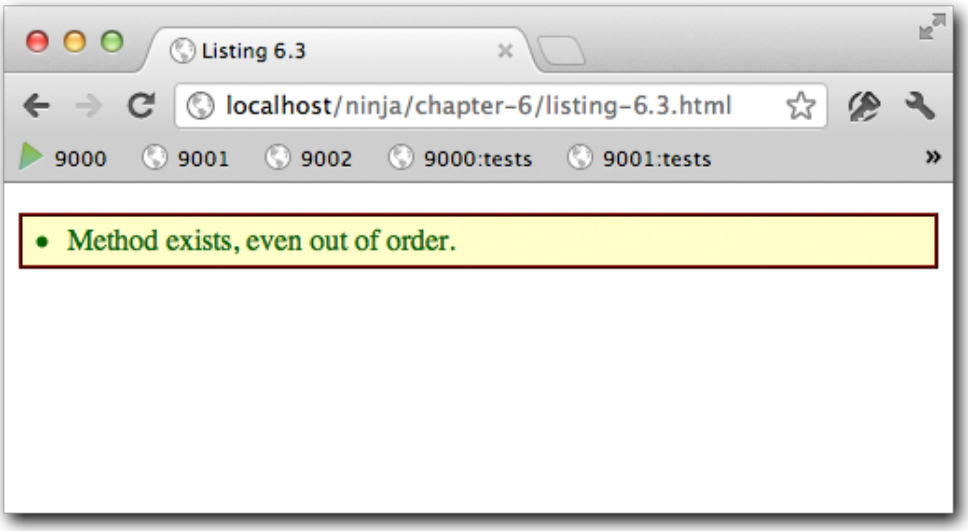

Figure 6.2: Our test proves that prototype changes are applied live!

So clearly there's more to all this than a simple copying of properties.

What's really going on is that prototype properties are not copied anywhere, but rather, the prototype is attached to the constructed object and consulted during the reconciling of property references made to the object.

A simplified overview of the process is as follows:

1. When a property reference to an object is made, first the object itself is checked to see if the property exists. If it does, the value is taken. If not…
2. The prototype associated with the object is located and *it* is checked for the property. If it exists, the value is take. If not…
3. The value is `undefined`.

We'll see later on in the chapter that things get a little more complicated than this, but this is a good enough understanding for now.

How does all this work?

Each object in JavaScript has an implicit property named constructor that references the constructor that was used to create the object. And because the prototype is a property of the constructor, each object has a way to find its prototype.

Take a look at figure 6.3 which a capture of the JavaScript console (in Chrome) when the code of listing 6.3 is loaded into the browser.

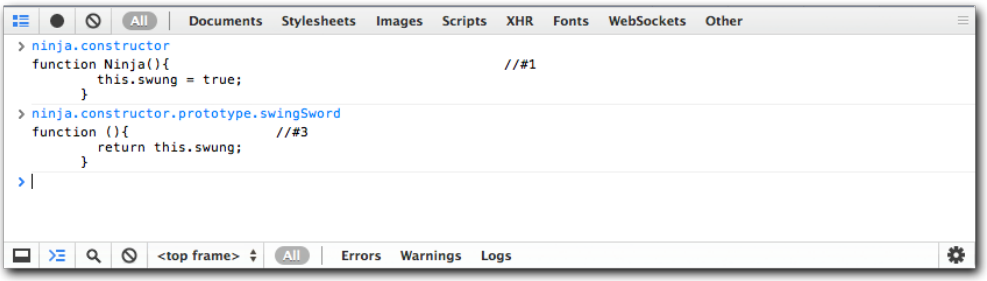

Figure 6.3 Inspecting the structure of an object reveals the path to its prototype

When we type the reference `ninja.constructor` into the console, we see that it references the `Ninja()` function as we'd expect as the object was created by using that function as a constructor.

A deeper reference to `ninja.constructor.prototype.swingSword` shows how we can access prototype properties from the object instance.

This explains why changes to the prototype made after the object has been constructed take effect. The prototype is actively attached to the object and any references made to object properties are reconciles, using the prototype is necessary, at the time of reference.

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 objects have been instantiated.

Before we move on, let's try one more variation on this theme. Observe the code of listing 6.4.

Listing 6.4: 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 6.2. 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. This makes perfect sense. The prototype is only consulted when a property reference on the object itself fails. As the object possesses a `swingSword` property, the prototyped version doesn't come into play, even though it was added to the prototype after object construction.

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

6.1.2 Object typing via constructors

While it's great to know how JavaScript uses the prototype during the reconciliation of property references, it's also handy for us to know which function constructed the object instance. We can refer back to it at any time, possibly even using it as a form of type checking, as shown in Listing 6.5.

Listing 6.5: 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 6.5 we define a constructor and create an object instance using it. Then, we examine 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 the `constructor` property that we now know is added to all instances, as 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 constrcutor, we can instantiate a new Ninja object using it, as shown in Listing 6.6.

Listing 6.6: 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, doing so 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 useful, but we've just scratched the surface of the super-powers that prototypes confer to us. Now things get really interesting.

6.1.3 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 6.7, in which we will attempt to add inheritance to an instance.

Listing 6.7: Trying to achieve inheritance with prototypes

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

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

```
  function Ninja(){}                                          #2

  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). 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's method to the Ninja prototype.

Running our test reveals that we failed, as shown in figure 6.4.

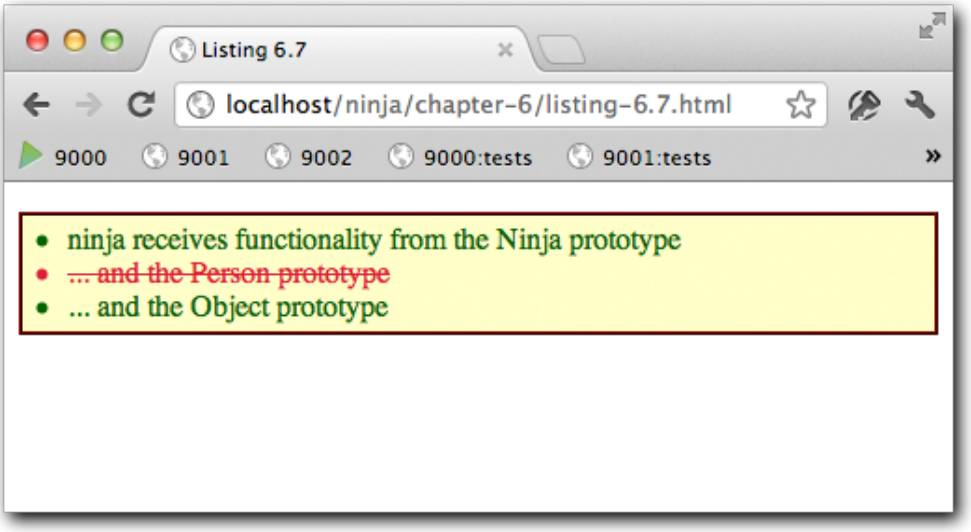

Figure 6.4: Our ninja isn't really a Person. No dancing!

While we have taught the Ninja to mimic the dance of a Person, it hasn't *made* the Ninja a Person. Besides, using this approach, we'd need to copy each property of Person to the Ninja prototype individually. That's no way to do inheritance! Let's keep exploring.

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

What we really want to achieve is a ***prototype chain*** so that a Ninja can be a Person, and a person can be a Mammal, and a Mammal can be an Animal, and so on, all the way to Object. The best technique capable of creating such 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 6.7 to use this technique as shown in listing 6.8.

Listing 6.8 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");
  assert(typeof ninja.dance == "function", "… and can dance!")

</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 6.5.

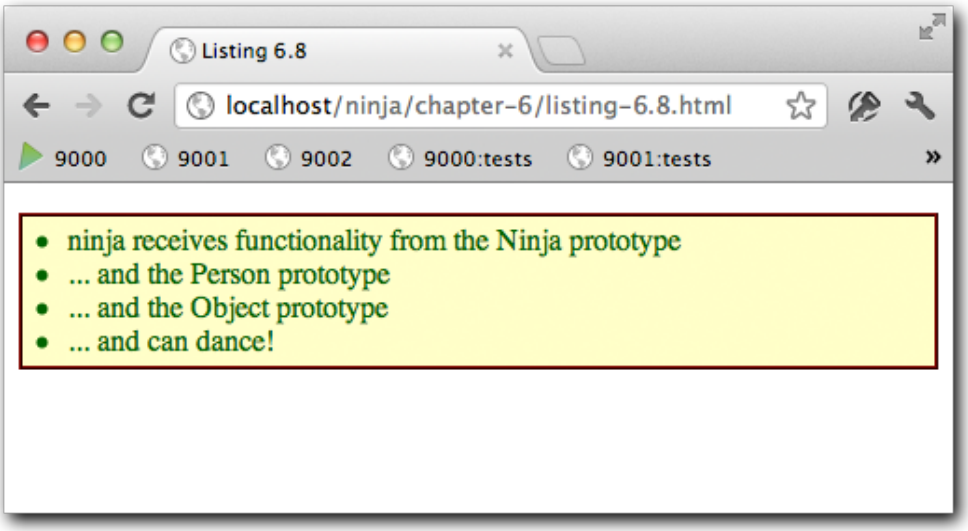

Figure 6.5: Our ninja is a Person! Let the dancing begin.

The very important implications of this 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 happy 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 6.6.

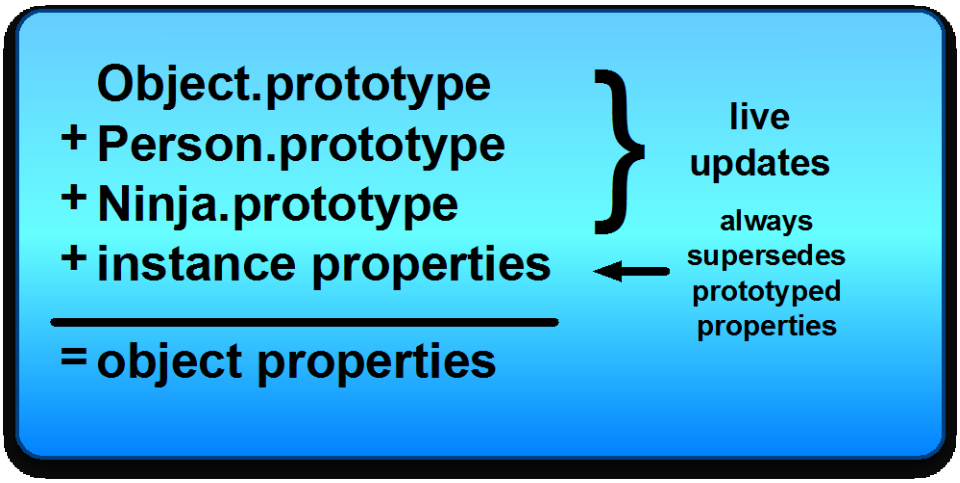

Figure 6.6: The order in which properties are reconcile for an instantiated object

It's important to note from Figure 6.6 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 this future functionality, eliminating the need to wait until the next version of the language is ready. Listing 6.9 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 6.9: 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 method is defined, it will defer to the native method.

If we determine that the method does not exist, we go ahead and add it to 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, like Array, 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 prototype, 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.

6.1.4 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 6.10.

Listing 6.10: 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 versions 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. We might consider doing 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…

6.2 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.

6.2.1 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 when we iterate over the properties of the object and 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 6.11.

Listing 6.11: 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 Defines the 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 6.7.

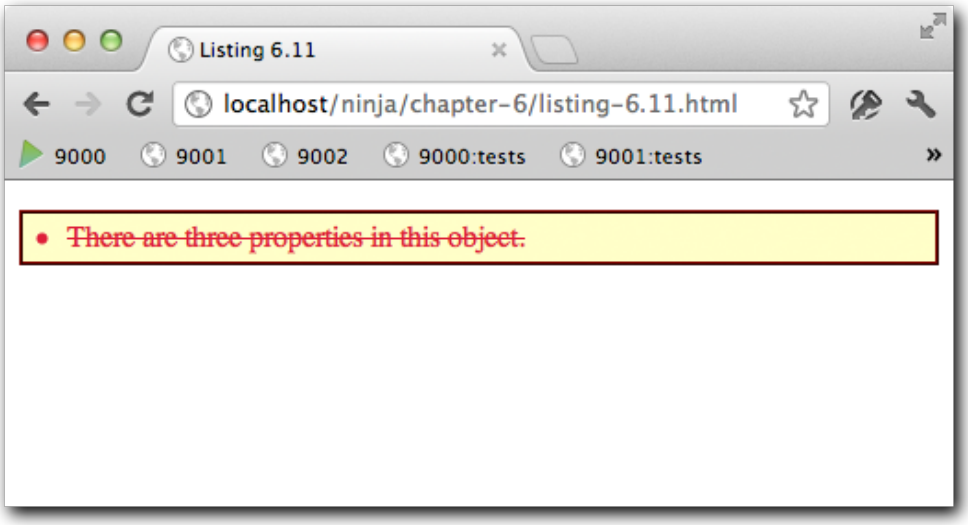

Figure 6.7: Whoa! We screwed up a fundamental assumption of objects!

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 6.12.

Listing 6.12: 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 – many 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 could trap us.

6.2.2 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 6.13.

Listing 6.13: 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. Then 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 6.8.

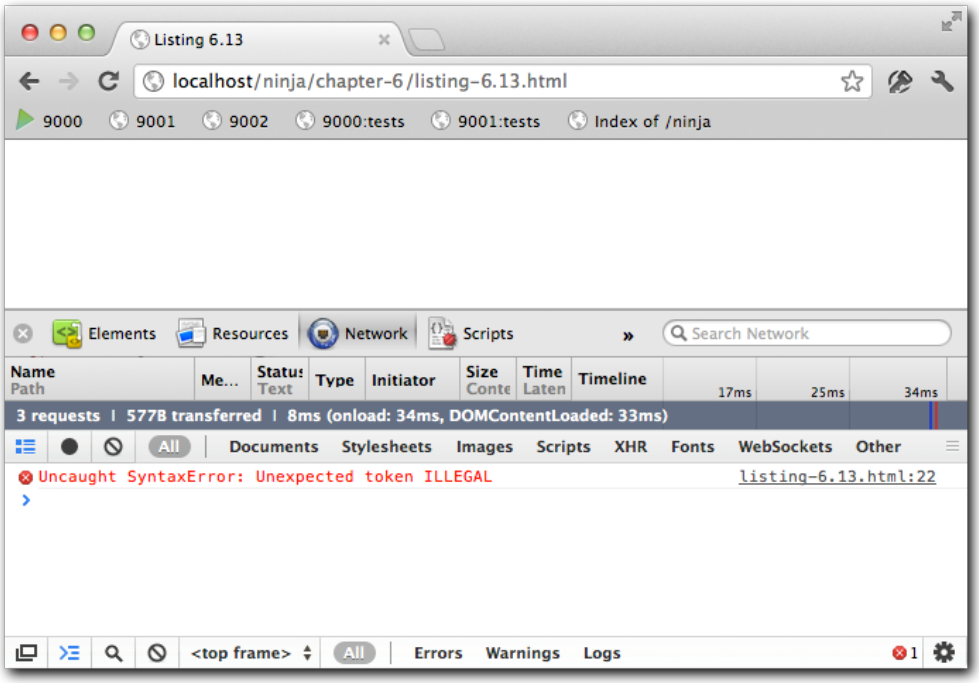

Figure 6.8: When tests won't even load, we know there's a big problem!

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.

6.2.3 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. But let's take a look at the code of listing 6.14.

Listing 6.14: 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 6.15.

Listing 6.15: 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 6.15 we define a new class, and give it its own `length` property. Then, rather than trying to subclass Array, which we've already learned won't work across all browsers, we use an immediate function (#2) to add selected methods from Array to our class using the `apply()` trick we learned back in chapter 4.

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.

6.2.4 Instantiation issues

We've already noted that functions can serve a dual purpose: as "normal" functions, and as constructors. Because of this, it may not always be clear to users of our 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 6.16.

Listing 6.16: 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 6.9.

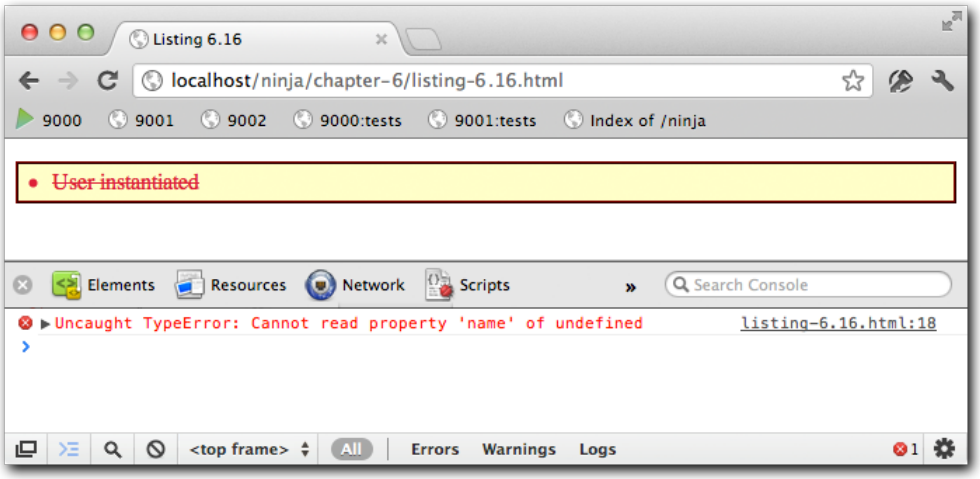

Figure 6.9: Our object didn't even get instantiated!

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 the instantiation of a new object. A novice 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, we have used 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 6.17.

Listing 6.17: 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. D'oh!

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` 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 6.18

Listing 6.18: Determining if we're called as a constructor

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

  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.");

</script>
```

Recall a few important concepts:

- We can get a reference to the currently executing function via `arguments.callee` (we learned this in chapter 4)
- 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 tests 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't ninjas, we might 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 6.19.

Listing 6.19: 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 6.18 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 delineation between functions that are meant to be instantiated, and not, which can be quite useful.

Enough of the problems. Let's take a look at how we use these new-found powers to write more class-like code.

6.3 Writing 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 are 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 6.20 shows an example of a syntax that could achieve the above listed goals.

Listing 6.20: 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 Subclasses Object
#2 Subclasses Person
#3 Calls superclass constructor
#4 Tests Person class
#5 Tests Ninja class
#6 Tests class hierarchy

There are 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 inherits 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 6.21 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. So don't feel bad if it takes a while for you to grok it.

In fact, to make it a bit easier to digest, we're going to present the code in complete form in listing 6.21, 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 6.21: 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.

6.3.1 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 its `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`. (We'll be investigating regular expressions at length in the following chapter.)

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 matches 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.

6.3.2 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 or skip 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 we 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.

6.3.3 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, and that references the original method in the superclass.

For example, recall from listing 6.20, 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 6.21) 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 of a 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 results 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.

6.4 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 their 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.

In the final example of this chapter we caught a glimpse of the use of regular expressions. In the next chapter we'll take an in-depth look at this frequently overlooked, but very powerful, feature of the JavaScript language.

7

Wrangling regular expressions

Covered in this chapter:

- A refresher on regular expressions
- Compiling regular expressions
- Capturing with regular expressions
- Frequently-encountered idioms

Regular expressions are a necessity of modern development. There, we said it.

While many a web developer could go through life happily ignoring regular expressions, there are just some problems that need to be solved in JavaScript code that cannot be addressed in a fashion that would be called elegant without regular expressions.

Sure, there may be other ways to solve the same problems. But frequently, something that might take a half-screen of code can something be distilled down to a single statement with the proper use of regular expressions, Every JavaScript ninja will have the regular expression as an essential part of his or her toolkit.

Regular expressions trivialize the process of tearing apart strings and looking for information. Everywhere you look in mainstream JavaScript libraries, you'll see the prevalent use of regular expressions for spot tasks such as:

- 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
- and more …

Let's start by looking at an example.

7.1 Why regular expressions rock

Let's say that we wanted to validate that a string, perhaps entered into a form by a web site user, follows the format for a 9-digit US postal code. We all know that US Postal Service has little sense of humor, and they insist that a US post code (also known as a ZIP Code) follows a specific format:

```
99999-9999
```

where each 9 represents any decimal digit. So the format is: five decimal digits, followed by a dash, followed by four decimal digits. If you use any other format, your package or letter gets diverted into the black hole of the "hand sorting" department, and good luck predicting how long it will take to emerge again from the event horizon.

So let's create a function that, given a string, will verify that the USPS will stay happy. We could resort to simply performing a comparison on each character, but we're ninjas and that's just too inelegant a solution, resulting in a lot of needless repetition. Rather, consider the solution in listing 7.1:

Listing 7.1: Testing for a specific pattern in a string

```
function isThisAZipCode(candidate) {
  if (typeof candidate !== "string" ||                    #1
      candidate.length != 10) return false;               #1
  for (var n = 0; n < candidate.length; n++) {
    var c = candidate[n];
    switch (n) {                                          #2
      case 0: case 1: case 2: case 3: case 4:
      case 6: case 7: case 8: case 9:
        if (c < '0' || c > '9') return false;
        break;
      case 5:
        if (c != '-') return false;
        break;
    }
  }
  return true;                                            #3
}
```

#1 Short circuits obviously bogus candidates
#2 Performs test based upon character index
#3 If all succeeded, we're good!

This code takes advantage of the fact that we only have two different checks to make depending upon the position of the character within the string. We still need to perform up to nine comparisons at run time, but we only have to write each comparison once.

Even so, would anyone consider this solution *elegant*? Surely, more elegant than the brute-force, non-iterative approach would be, but it still seems like an awful lot of code for such a simple check.

Now consider:

```
function isThisAZipCode(candidate) {
  return /^\d{5}-\d{4}$/.test(candidate);
}
```

Except for some rather esoteric syntax in the body of the function, that's a lot more succinct and elegant, no?

That's the power of regular expressions, and it's just the tip of the iceberg. Don't worry if that syntax looks like someone's pet iguana walked across the keyboard to you, we're about to recap regular expressions before we dive off into seeing how to use them in ninja-like fashion on our pages.

7.2 A regular expression refresher

Much as we'd like to, we can't offer you an exhaustive tutorial on regular expressions in the space we have. After all, entire books have been dedicated to regular expressions. But we'll do our best to hit all the important points.

For more detail than we can offer in this chapter, O'Reilly's *Mastering Regular Expressions* is a popular choice.

Let's dig in.

7.2.1 Regular expressions explained

The term ***regular expression*** stems from mid-century mathematics when a mathematician named Kleene described models of computational automata as "regular sets". But that's not helping us understand anything about regular expressions, so let's simplify things and say that a regular expression is simply way to express a *pattern* for matching strings of text.

The expression itself consists of terms and operators that allow us to define these patterns. We'll see what those terms and operators consist of very shortly.

In JavaScript, we have two ways to create a regular expression: via a regular expression literal, and by constructing an instance of a RegExp object.

For example, if we wanted to create a rather mundane regular expression (or ***regex***, for short) that matches the string "test" exactly, we could do so with a regex literal:

```
var pattern = /test/;
```

Or, we could construct a RegExp instance, passing the regex as a string:

```
var pattern = new RegExp("test");
```

Both of these formats result in the same regex being created in the variable `pattern`.

The literal syntax is preferred when the regex is known at development time, and the constructor approach used when the regex is constructed at run-time by building it up dynamically in a string.

One of the reasons that the literal syntax is preferred over expressing regexs in a string is that (as we shall soon see), the backslash character plays an important part in regular expressions. But, the backslash character is *also* the escape character for string literals. So to express a backslash within a string literal, we need to use `\\` (double backslash). This can make regular expressions, which already possess rather cryptic syntax, even more odd-looking when expressed within strings.

In addition to the expression itself, there are three flags that can be associated with a regex: `i` (insensitive), `g` (global), and `m` (multi-line). These flags are appended to the end of

the literal (example: `/test/ig`), or passed in a string as the second parameter to the RegExp constructor (`new RegExp("test","ig")`).

The meaning of the flags is as follows:

- `i`: makes the regex case insenstive. So `/test/i` matches not only "test", but also "Test", "TEST", "tEsT", and so on.
- `g`: matches all instances of the pattern, as opposed to the default of "local" which matches only the first occurence. More on this later.
- `m`: allows matches across multiple lines as might be obtained from the value of a textarea element.

Simply matching the exact string "test" (even in a case insensitive manner) isn't very interesting – after all, we can do that particular check with simple string comparison. So let's take a look at the terms and operators that give regular expressions their immense power to match more compelling patterns.

7.2.2 Terms and operators

Regular expressions, like most other expressions with which we are familiar, are made up of terms and operators that qualify those terms. In the sections that follow, we'll take a look at these terms and operators and see how they can be used to express patterns.

EXACT MATCHING

Any character that's not a special character or operator (as we'll be introducing as we go along), represents a character that must appear literally in the expression. For example, in our `/test/` regex, there are four terms that represent characters that must appear literally in a string for it to match the expressed pattern.

Placing such characters one after the other implicitly denotes an operation that means "followed by". So `/test/` means 't' followed by 'e' followed by 's' followed by 't'.

MATCHING FROM A CLASS OF CHARACTERS

Many times, we won't want to match a specific literal character, but from a finite set of characters. We can specify this with the set operator (also called the *character class* operator) by placing the set of characters that we wish to match in square brackets.

For example:

```
[abc]
```

would signify that we want to match any of the characters 'a', 'b' or 'c'. Note that even though this expression spans 5 characters, it matches only a single character in the candidate string.

Other times, we want to match anything *but* a finite set. We can specify this by placing a caret character right after the opening bracket of the set operator:

```
[^abc]
```

This changes the meaning to any character *but* 'a', 'b' or 'c'.

There's one more invaluable variation to the set operation: the ability to specify a range of values. For example, if we wanted to match any one of the lower-case characters between 'a' and 'm', we could write `[abcdefghijklm]`. But we can express that much more succinctly with:

```
[a-m]
```

The dash indicates that all characters between 'a' and 'm' (lexicographically) are included in the set.

ESCAPING

Not all characters represent their literal equivalent. Certainly all of the alphabetic and decimal digit characters represent themselves, but as we will shortly see, special characters such as `$` and the period (dot) character either represent matches to something other than themselves, or operators that qualify their preceding term. In fact, we've already seen how the `[`, `]` and `^` characters are used to represent something other than their literal selves.

So how do we specify that we want to match a literal `[` or `$` or `^` or other special character? Within a regex, the backslash character escapes whatever character follows it, making it a literal match term. So `\[` specifies a literal match to the `[` character rather than the opening of a set operation. A double backslash (`\\`) matches a single backslash.

BEGINS AND ENDS

Frequently we may wish to ensure that a pattern matches at the beginning of a string, or perhaps at the end of a string. The carat character, when used as the first character of the regex anchors the match at the beginning of the string, such that `/^test/` only matches if the substring "test' appears at the beginning of the string being matched. (Note that this is an overloading of the ^ character as it is also used to negate a set.)

Similarly, the dollar sign character (`$`) signifies that the pattern must appear at the end of string, such as `/test$/`.

Using both the `^` and the `$` indicates that the specified pattern must encompass the entire candidate string. For example: `/^test$/`.

REPEATED OCCURRENCES

If we wanted to specify that we desire to match a series of 4 'a' characters, we might express that with `/aaaa/`, but what if we wanted to match *any* number of the same character?

Regular expressions give us the means to specify a number of different repetition options:

- We can specify that a character is optional (in other words, can appear either once or not at all) but following it with the `?`. Example: `/t?est/` matches both 'test' and 'est'.
- If we want a character to appear one or many times, we use the `+`, as in `/t+est/` which matches 'test', 'ttest', and 'tttest', but not 'est'.
- If we want the character to appear *zero* or many times, the `*` is used as in `/t*est/`, which matches 'test', 'ttest', 'tttest' *and* 'est'.

- We can specify a fixed number of repetitions with the number of allowed repetitions between braces. For example, `/a{4}/` indicates a match on four consecutive 'a' characters.
- We can also specify a range for the repetition count by specifying the range with a comma separator. For example, `/a{4,10}/` matches any string of four through ten consecutive 'a' characters.
- The second value in a range can be omitted (but leaving the comma) to indicate an open-ended range. The regex `/a{4,}/` matches any string of four or more consecutive 'a' characters.

Any of these repetition operators can be *greedy* or *non-greedy*. By default, they are greedy: they will consume all the possible characters that comprise a match. Annotating the operator with a `?` character, as in `a+?`, makes the operation non-greedy: it will consume only enough characters to make a match.

PREDEFINED CHARACTER CLASSES

There are some characters that we'd like to match that are impossible to specify with literals characters (such as control characters like a carriage return), and there are also character classes that we might often want to match; for example, the set of decimal digits, or the set of whitespace characters. So that we can use control character matching in our regular expressions, and so that we don't need to resort to the character class operator for commonly used sets of characters, the regular expression syntax gives us a number of predefined terms that represent these characters or commonly used classes.

Table 7.1 lists these terms and what character or set of characters they represent.

Table 7.1: Predefined character class and character terms

Predefined term	Matches
\t	Horizontal tab
\b	Backspace
\v	Vertical tab
\f	Form feed
\r	Carriage return
\n	New line
\cA : \cZ	Control characters
\x0000 : \xFFFF	Unicode hexadecimal
\x00 : \xFF	ASCII hexadecimal

.	Any character, except for newline (\n)
\d	Any decimal digit. Equivalent to `[0-9]`.
\D	Any character but a decimal digit. Equivalent to `[^0-9]`.
\w	Any word character. Equivalent to `[A-Za-z0-9_]`.
\W	Any character but a word character. Equivalent to `[^A-Za-z0-9_]`.
\s	Any whitespace character (space, tab, form feed, and so on).
\S	Any character but a whitespace character.
\b	A word boundary.
\B	Not a word boundary (inside a word).

These predefined sets help us to keep our regular expressions from looking excessively cryptic.

GROUPING

So far we've seen that operators (such as `+` or `*`) only affect the preceding term. If we wanted to apply the operator to a group of terms, we can use parentheses for groups just as in a mathematical expression. For example, `/(ab)+/` matches one of more consecutive occurrences of the substring 'ab'.

When a part of a regex is grouped with parentheses, it serves double duty, also creating what's known as a ***capture***. There's a lot to captures, and we'll be discussing them in more depth later on in section 7.4.

ALTERNATION (OR)

Alternatives can be expressed using the | (pipe) character. For example: `/a|b/` matches either of the 'a' or 'b' character, and `/(ab)+|(cd)+/` matches one or more occurrences of either 'ab' or 'cd'.

BACK REFERENCES

The most complex of terms we can express in regular expressions are back references to captures defined in the regex. The notation for such a term is the backslash followed by the number of the capture to be referenced, beginning with 1, for example: `\1`, `\2`, and so on.

An example could be `/^([dtn])a\1/`, which matches a string that starts with any of the 'd', 't' or 'n' characters, followed by an 'a', followed by whatever character matched the first capture. This latter point is important! This is not the same as `/[dtn]a[dtn]/`! The character following the 'a' cannot be any of 'd', or 't', or 'n', but whichever one of those triggered the match for the first character. As such, which character the `\1` will match cannot be known until evaluation time.

A good example of where this might be useful is in matching XML-type markup elements. Consider the following regex:

```
/<(\w+)>(.+)<\/\1>/
```

This allows to match simple elements such as "whatever". Without the ability to specify a back-reference, this would not be possible as we would have no way to know what closing tag would match the opening tag ahead of time.

OK, we know that that was kind of a whirlwind crash course on regular expressions. If they're still making you pull your hair out, we strongly recommend using one of the resources we referenced earlier in this chapter if you find yourself bogging down in the material that follows.

Now that we have a handle on what regular expressions are, let's learn how to use them wisely in our code.

7.3 Compiling regular expressions

Regular expressions go through multiple phases of processing, and understanding what happens during each of these phases can help us to optimize JavaScript code that utilizes regular expressions.

The two prominent phases are compilation and execution. Compilation occurs when the regular expression is first created. 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 created (discounting any internal optimizations performed by the browser).

Frequently browsers *are* smart enough to determine when identical regular expressions are being used, and to cache the compilation results for that particular expression. However, we cannot count on this being 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.

As we learned in our regular expression overview of the previous section, there are two ways of creating a compiled regular expression in JavaScript: via literal, and via constructor. Let's look at an example as shown in Listing 7.2

Listing 7.2: Two ways to create a compiled regular expression

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

  var re1 = /test/i;                          #1

  var re2 = new RegExp("test", "i");          #2

  assert(re1.toString() == "/test/i",
         "Verify the contents of the expression.");
  assert(re1.test("TesT"), "Yes, it's case-insensitive.");
  assert(re2.test("TesT"), "This one is too.");
  assert(re1.toString() == re2.toString(),
        "The regular expressions expressions are equal.");
  assert(re1 != re2, "But they are different objects.");
</script>
```

#1 Creates regex via literal
#2 Creates regex via constructor

In this example, both regular expressions are in their compiled state after creation. If we were to replace every reference to `re1` with the literal `/test/i`, it's likely that the same regex would be compiled time and time again. So compiling a regex *once* and storing it in a variable for later reference can be an important optimization.

Note that each regex has a unique object representation: every time that a regular expression is created (and thus compiled), a new regular expression object is created. This is unlike other primitive types (like string, number, etc.) as the result will always be unique.

Of particular importance is the use of the constructor (`new RegExp(...)`) to create a regular expression. This technique allows us to build and compile an expression from a string that we can dynamically create at run time. This can be immensely useful for constructing complex expressions that will be heavily reused.

For example, let's say that we wanted to determine which elements within a document have a particular class name whose value we won't know until run time. As elements are capable of having multiple class names associated with them (incomprehensibly stored in a space-delimited string) this serves as an interesting example of run-time regular expression compilation, as shown in Listing 7.3.

Listing 7.3: Compiling a run-time regular expression for later use

```
<div class="samurai ninja"></div>                                  #1
<div class="ninja samurai"></div>                                  #1
<div></div>                                                        #1
<div class="samurai ninja ronin"></div>                            #1

<script>
  function findClassInElements(className, type) {

    var elems =                                                    #2
      document.getElementsByTagName(type || "*");

    var regex =                                                    #3
      new RegExp("(^|\\s)" + className + "(\\s|$)");

    var results = [];                                              #4

    for (var i = 0, length = elems.length; i < length; i++)
      if (regex.test(elems[i].className)) {                        #5
        results.push(elems[i]);
      }
    return results;
  }

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

#1 Creates test subjects
#2 Collects elements by type
#3 Compiles regex using passed name

#4 Stores results
#5 Tests for regex match

There are a number of interesting things that we can learn from Listing 7.3. To start, we set up a number of "test subject" `<div>` elements with various combinations of class names (#1). Then we define our class name checking function, which accepts as parameters the class name for which we will check, and the element type to check within.

First, we collect all the elements of the specified type (#2). Then we set up our regular expression (#3). Note the use of the `new RegExp()` constructor to compile a regular expression based upon the class name passed to the function. This is an instance where we are unable to use a regex literal, as the class name for which we will search isn't known in advance.

We construct (and hence, compile) this expression once, in order to avoid frequent and unnecessary recompilation. Because the contents of the expression are dynamic (based upon the incoming `className` argument) we can realize major performance savings by handling the expression in this manner.

The regex itself matches either of the beginning of the string or a whitespace character, followed by our target class name, followed by either of a whitespace character or the end of the string. Something to notice is the use of a double-escape (`\\`) within new regex: `\\s`. When creating literal regular expressions with terms including the backslash, 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 we must be aware of when constructing regular expressions in strings rather than literals.

Once compiled, using the regex to collect (#4) the matching elements is a snap using the `test()` method (#5)

Preconstructing and precompiling regular expressions so that they can be re-used time and time again is a recommended technique that affords us performance gains that cannot be ignored. Virtually all complex regular expression situations can benefit from the use of this technique.

Back in the introductory section, we mentioned that the use of parentheses in regular expressions served not only to group terms for operator application, but also created what is known as ***captures***. Let's find out more about that.

7.4 Capturing matching segments

The height of usefulness with respect to regular expressions is realized when we *capture* the results that are found so that we can do something with those results. Simply determining if a string matches a pattern is an obvious first step and often all that we need, but determining *what* was matched is also useful in many situations.

7.4.1 Performing simple captures

Take a situation in which we want to extract a value that's embedded in a complex string. A good example of such a string might be the manner in which opacity values are specified for

Internet Explorer. Rather than the conventional `opacity` rule with a numerical value employed by the other browsers, IE8 and earlier use a rule such as:

```
filter:alpha(opacity=50);
```

In the example, of Listing 7.4 we extract the opacity value out of this filter string.

Listing 7.4: A simple function for capturing an embedded value

```
<div id="opacity"
     style="opacity:0.5;filter:alpha(opacity=50);">      #1
</div>                                                   #1

<script>
  function getOpacity(elem) {
    var filter = elem.style.filter;
    return filter ?                                      #2
      filter.indexOf("opacity=") >= 0 ?
        (parseFloat(filter.match(/opacity=([^)]*)/)[1]) / 100) + "" :
        "" :
      elem.style.opacity;
  }

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

#1 Defines test subject
#2 Decides what to return

We define an element that specifies both styles for opacity (one for standards-compliant browsers, and one for IE) that we'll use as a test subject (#1). Then we create a function that will return the opacity value as the standards-defined value between 0.0 and 1.0, regardless of how it was defined.

The opacity parsing code may seem a little bit confusing at first (#2), but it's not too bad once we 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` property is resident, we need to verify that it will contain the opacity string that we're looking for. We do that with the `indexOf()` call.

At this point we can get down to the actual opacity value extraction. The `match()` method of regular expressions returns an array of captured values if a match is found, or `null` if no match is found. In this case we can be confident that there *will* be a match, as we already determined that with the `indexOf()` call.

The array returned by `match` always includes the entire match in the first index, and then each subsequent capture following. Remember that the captures are defined by parentheses in the regular expression. Thus, when we match the opacity value, the value is actually contained in the `[1]` position of the array as the only capture we specified in our regex was created by the parentheses that we embedded after the `opacity=` portion of the regex.

This example used a local regular expression and the `match()` method. Things change a bit when we use global expressions. Let's see how.

7.4.2 Matching using global expressions

As we saw in the previous section, using a local regular expression (one without the global flag) with the String object's `match()` methods returns an array containing the entire matched string, along with any matched captures in the operation.

But when we supply a global regular expression (one with the `g` flag included), `match()` returns something rather different. It's still an array of results, but in the case of a global regular expression, which matches all possibilities in the candidate string rather than just the first match, the array returned contains the global matches; captures *within* each match are not returned in this case.

We can see this in action in the code and tests of listing 7.5.

Listing 7.5: Differences between a global and local search with `match()`

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

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

  var results = html.match(/<(\/?)(\w+)([^>]*?)>/);          //#1

  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);              //#2

  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.");

</script>
```

#1 Matches using a local regex
#2 Matches using a global regex

We can see that when we do a local match (#1), a single instance is matched and the captures within that match are also returned, but when we use a global match (#2), what's returned is the list of matches.

If captures are important to us, we can regain this functionality, while still performing a global search, by using the regular expression's `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's called. A typical pattern for how it can be used is shown in the code of Listing 7.6.

Listing 7.6: Using the exec method to do both capturing and a global search

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

  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) {                    #1
    assert(match.length == 4,
           "Every match finds each tag and 3 captures.");
    num++;
  }

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

</script>
```

#1 Repeatedly calls exec()

In this example, we repeatedly call the `exec()` method (#1) which retains state from its previous invocation so that each subsequent call progresses to the next global match. Each call returns the next match *and* its captures.

By using either `match()` or `exec()`, we can always find the exact matches (and captures) that we're looking for. But we discover that we'll need to dig further when we need to begin referring back to the captures themselves within the regex.

7.4.3 Referencing captures

There are two ways in which we can refer back to portions of a match that we'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 (in which we match an opening or closing HTML tag) and modify it to also match the inner contents of the tag itself, as shown in Listing 7.7.

Listing 7.7: Using back-references to match the contents of an HTML tag.

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

  var html = "<b class='hello'>Hello</b> <i>world!</i>";

  var pattern = /<(\w+)([^>]+)>(.*?)<\/\1>/g;                #1

  var match = pattern.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 = pattern.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.");

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

#1 Uses capture back-reference

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 whatever the capture matched. (This all assumes, of course, that there aren't any embedded tags of the same name within the current tag; so this is hardly an exhaustive example of tag matching.)

Additionally, there's a way to get capture references within the replace string of a call to the `replace()` method. Instead of using the back-reference codes, as in the example of listing 7.7, we use the syntax of `$1`, `$2`, `$3`, up through each capture number. An example of such usage is:

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

In this code, the value of the first capture (in this case, the capital letter `F`), is referenced in the *replace string* (via `$1`). This allows us to specify a replace string without even knowing what its value will be until matching time. That's a pretty powerful ninja-esque weapon to wield.

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

As both captures and expression grouping are specified using parentheses, there's no way for the regular expression processor to know which sets of parentheses we added to the regex for grouping and which were intended to indicate captures. So it treats all parentheses sets as both groups and captures, which can result in the capture of more information than we really intended simply because we needed to specify some grouping in the regex. What can we do in such cases?

7.4.4 Non-capturing groups

As we noted, parentheses serve a double duty: they not only group terms for operations, they also specify captures. This is most times not an issue, but in regular expressions in which lots of grouping is going on, it could also cause lots of needless capturing to go on, which may make sorting through the resulting captures tedious.

Consider the following regex:

```
var pattern = /((ninja-)+)sword/;
```

Here, our intent is to create a regex that allows the prefix "ninja-" to appear one or more times before the word sword, and we want to capture the entire prefix. This regex requires two sets of parentheses:

1. The parentheses that define the capture (everything before the string `sword`).
2. The parentheses that group the text `ninja-` for the `+` operator.

This all works fine, but results in more than the single intended capture due to the inner set of grouping parentheses.

To allow us to indicate that a set of parentheses should not result in a capture, the regular expression syntax lets us put the notation `?:` immediately after the opening parenthesis to indicate a non-capture grouping. This is known as a ***passive sub-expression***.

So, changing our regular expression to:

```
var pattern = /((?:ninja-)+)sword/;
```

causes only the outer set of parentheses to create a capture. The inner parentheses have been converted to a passive sub-expression.

To test this, refer to the code of listing 7.8.

Listing 7.8: Grouping without capturing

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

  var pattern = /((?:ninja-)+)sword/;                    //1

  var ninjas = "ninja-ninja-sword".match(pattern);

  assert(ninjas.length == 2,"Only one capture was returned.");
  assert(ninjas[1] == "ninja-ninja-",
         "Matched both words, without any extra capture.");

</script>
```

#1 Uses passive sub-expression

Running these tests, we can see that the passive sub-expression (#1) prevents unnecessary captures.

Wherever possible in our regular expressions, we should strive to use non-capturing (passive) groups in place of capturing when the capture is unnecessary. The expression engine will have to do much less work in remembering and returning the captures; if we don't actually need captured results, then there's no need to ask for them! The price that we pay is that it can make what are likely already-complex regular expressions a tad more cryptic.

Now let's turn our attention to another way that regular expressions give us ninja powers: using functions with the String's `replace()` method.

7.5 Replacing using functions

The `replace()` method of the String object is a powerful and versatile method, which we saw used briefly in our discussion of captures. When a regular expression is provided as the first parameter to `replace()`, it will cause a replacement on a match (or matches if the regex is global) to the pattern rather than a fixed string. For example, let's say that we wanted to replace all uppercase characters in a string with 'X'. We could write:

```
"ABCDEfg".replace(/[A-Z]/g,"X")
```

This results in a value of "XXXXXfg". Nice.

But perhaps the most powerful feature presented by `replace()` is the ability to provide a function as the replacement value rather than a fixed string.

When the replacement value (second argument) is a function, it is invoked for each match found (remember a global search will match all instances of the pattern in the source string) with a variable list of parameters:

- The full text of the match.
- The captures of the match, one parameter for each.
- The index of the match within the original string.
- The source string.

The value returned from the function serves as the replacement value.

This gives us a tremendous amount of leeway to determine what the replacement string should be at run time, with lots of information regarding the nature of the match at our fingertips.

For example, in listing 7.9 we use the function to provide a dynamic replacement value to convert a string with words separated by dashes to its camel-cased equivalent.

Listing 7.9: Converting a dashed string to camel case

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

  assert( "border-bottom-width".replace(
      /-(\w)/g,                                    #1
      function(all,letter) {
        return letter.toUpperCase();               #2
      }) == "borderBottomWidth",
      "Camel cased a hyphenated string.");

</script>
```

#1 Matches dashed characters
#2 Converts to uppercase

Here, we provided a regex that matches any character preceded by a dash character. A capture in the global regex identifies the character that was matched (without the dash). Each time the function is called (twice in this example), it is passed the full match string as the first argument, and the capture (only one for this regex) as the second argument. We aren't interested in the rest of the arguments, so we didn't specify them.

So the first time that the function is called, it is passed "-b" and "b", and the second time it is called it's passed "-w" and "w". In each case, the captured letter is uppercased and returned as the replacement string. We end up "-b" replaced with "B", and "-w" replaced with "W".

Because a global regex will cause such a replace function to be executed for every match in a source string, this technique can even be extended beyond doing rote replacements and be used as a means of string traversal as an alternative to doing the `exec`-in-a-while-loop technique that we saw earlier in this chapter.

For example, let's say that we were looking to take a query string and convert it to an alternate format that suits our purposes. We'd turn a query string such as:

```
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 results in some especially terse code as shown in listing 7.10.

Listing 7.10: A technique for compressing a query string

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

  function compress(source) {
    var keys = {};                                          #1

    source.replace(
      /([^=&]+)=([^&]*)/g,
      function(full, key, value) {                          #2
        keys[key] =
          (keys[key] ? keys[key] + "," : "") + value;
        return "";
      }
    );

    var result = [];                                        #3
    for (var key in keys) {                                 #3
      result.push(key + "=" + keys[key]);                   #3
    }                                                       #3

    return result.join("&");                                #4
  }

  assert(compress("foo=1&foo=2&blah=a&blah=b&foo=3") ==
         "foo=1,2,3&blah=a,b",
         "Compression is OK!");

</script>
```

#1 Stores located keys
#2 Extracts key/value info
#3 Collects key info
#4 Joins results with &

The most interesting aspect of Listing 7.10 is how it uses the string `replace()` method 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 replacement value argument and instead of returning a value, simply utilizing it as a means of searching.

The example code first declares a hash in which we'll store the keys and values that we find in the source query string (#1).

Then we call the `replace()` method (#2) on the source string, passing a regex that will match the key value pairs, and capture the key and the value. We also pass a function that will be passed the full match, the key capture and the value capture. These captured values get stored in the hash for later reference.

Note how we simply return the empty string because we really don't care what substitutions happen to the source string – we're just using the side effects rather than the actual result.

Once `replace()` returns, we declare an array in which to aggregate the results and iterate through the keys that we found, adding each to the array (#3).

Finally, we join each of the results we stored in the array using `&` as the delimiter, and return the result (#4).

Using this technique, we can coopt the String object's `replace()` method as our very-own string searching mechanism. The result is not only fast but also simple and effective. The level of power that this technique provides, especially in light of the small amount of code necessary, should not be underestimated.

In fact, all of these regular expression techniques can have a huge impact on how we write script on our pages. Let's see how we can apply what we've learned to solve common problems we might encounter.

7.6 Solving common problems with regular expressions

In JavaScript, a few idioms tend to occur again and again, but their solutions aren't always obvious. Our knowledge of regular expressions can definitely come to our rescue, and in this section we'll look at a couple of common problems that we can solve with a regex or two.

7.6.1 Trimming a string

Removing extra whitespace from the beginning and the end of a string is a common need but one that was (until recently) omitted from the String object. Almost every JavaScript library provides and uses an implementation of string trimming for older browsers that don't have the `String.trim()` method.

The most-commonly used approach looks something like the code in Listing 7.11.

Listing 7.11: A common solution to stripping whitespace from a string

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

  function trim(str) {
    return str.replace(/^\s+|\s+$/g, "");          #1
```

```
    }

    assert(trim(" #id div.class ") == "#id div.class",
           "Extra whitespace trimmed from a selector string.");

  </script>
```

#1 Trims string without looping

Rather than iterating over characters to determine which need to be trimmed, a single call to the `replace()` method with a regex that matches whitespace at the beginning or end of a string does the job.

Steven Levithan, one of the authors of O'Reilly's *Regular Expressions Cookbook*, has done a lot of research into this subject producing a number of alternative solutions, which he details at 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, which is certainly a fringe case, for most applications.

Of those solutions, two are of particular interest. The first is accomplished using regular expressions, but with no `\s+` and no | or operator, shown in listing 7.12:

Listing 7.12: An alternative double-replacement trim implementation

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

  function trim(str) {
    return str.replace(/^\s\s*/, '')            #1
              .replace(/\s\s*$/, '');           #1
  }

  assert(trim(" #id div.class ") == "#id div.class",
         "Extra whitespace trimmed from a selector string.");

</script>
```

#1 Trims using two replacements

This implementation performs two replacements: one for the leading whitespace, and one for trailing whitespace.

Dave's 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 7.13.

Listing 7.13: A trim method that slices at the end of the string

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

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

  assert(trim(" #id div.class ") == "#id div.class",
```

```
        "Extra whitespace trimmed from a selector string.");

</script>
```

#1 Trims using a regex and slicing

This implementation uses a regex to trim at the leading edge, and a slice operation at the trailing edge.

Comparing performance of these implementations for short strings and document-length strings, the difference becomes quite noticeable as shown in table 7.2, which shows the time in milliseconds to perform 1000 iterations of the `trim()` method.

Table 7.2: Performance comparison of three trim() implementations

	Short string	Document
listing 7.11	8.7	2075.8
listing 7.12	8.5	3706.7
listing 7.13	13.8	169.4

This comparison makes it easy to see which implementation is the most scalable. While the implementation of listing 7.13 fared poorly against the other implementations for short strings, it left the others in the dust for much longer (document length) strings.

Ultimately, which will fare better depends on the situation in which you're going to perform the trimming. Most libraries use the first solution, and it's likely that we'll be using it on smaller strings, so that seems to be a safe bet.

Let's move on to another common need.

7.6.2 Matching newlines

When performing a search, it's sometimes desirable that the . (period) term, which matches any character except for newline, would also include newline characters. Regular expression implementations in other languages frequently include a flag for making this possible, but JavaScript's implementation does not.

Let's look at a couple of ways of getting around this omission in JavaScript, as shown in listing 7.14.

Listing 7.14: Matching *all* characters, including newlines

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

  var html = "<b>Hello</b>\n<i>world!</i>";                    #1

  assert(/.*/.exec(html)[0] === "<b>Hello</b>",                #2
         "A normal capture doesn't handle endlines.");

  assert(/[\S\s]*/.exec(html)[0] ===                           #3
         "<b>Hello</b>\n<i>world!</i>",
         "Matching everything with a character set.");
```

```
assert(/(?:.|\s)*/.exec(html)[0] ===                          #4
       "<b>Hello</b>\n<i>world!</i>",
       "Using a non-capturing group to match everything.");

</script>
```

#1 Defines our test subject
#2 Shows that newlines aren't matched
#3 Matches all using whitespace matching
#4 Matches all using alteration

In this example, we define a test subject string (#1) containing a newline. Then we try a number of ways of matching all of the characters in the string.

In the first test (#2) we verify that newlines, indeed, are not matched by the `.` operator.

Well, ninjas will not be denied, so in the next test (#3) we get our way with an alternate regex, `/[\S\s]/`, in which we define a character class that matches anything that's *not* a whitespace character and anything that *is* a whitespace character. This union is the set of all characters.

Another approach is taken in the next test (#4), where we use an alternation regex, `/(?:.|\s)*/`, in which we match everything matched by ., which is everything but newline, and everything considered whitespace, which includes newline. The resulting union is the set of all characters including newline. Note the use of a passive sub-expression to prevent any unintended captures.

Due to its simplicity (and implicit speed benefits), the solution provided by `/[\S\s]*/` is generally considered optimal.

Next, let's take a step to widen our view to a worldwide scope.

7.6.3 Unicode

Frequently in the use of regular expressions, we want to match alphanumeric characters; for example, an ID selector in a CSS selector engine implementation. However, assuming that the alphabetic characters will only be from the set of English characters is rather shortsighted.

Expanding the set to include Unicode characters is quite desirable, explicitly supporting multiple languages not covered by the traditional alphanumeric character set, as seen in listing 7.15.

Listing 7.15: Matching Unicode characters

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

  var text ="\u5FCD\u8005\u30D1\u30EF\u30FC";

  var matchAll =                                    #1
    /[\w\u0080-\uFFFF_-]+/;                         #1

  assert((text).match(matchAll),
         "Our regexp matches unicode!" );

</script>
```

#1 Matches all including Unicode

The specific technique that is used in this example to include the entire range of Unicode characters in the match is to create a character class that includes the `\w` term, to match all the "normal" word characters, plus a range that spans the entire set of Unicode characters above character code 128 (hex 0x80). Starting at 128 gives us some high ASCII characters along with all Unicode characters.

The astute among you might note that by adding the entire range of Unicode characters above \u0080 that we match not only alphabetic characters, but also all Unicode punctuation and other special characters (arrows, for example). But that's OK, because the point of the example is to show how to match Unicode characters in general. If you have a specific range of characters that you want to match, you can use the lesson of this example to add whatever range you wish to the character class.

Before we move on from our examination of regular expressions, let's tackle one more common issue.

7.6.4 Escaped characters

It's common for page authors to use names that conform to program identifiers when assigning `id` values to page elements, but that's just a convention; id values can contain other than "word" characters, including punctuation. For example, a web devo might use the `id` value `form:update` for an element.

A library developer, when writing an implementation for, say, a CSS selector engine, would like support this via escaped characters. This allows the user to specify complex names that don't conform to typical naming conventions. So let's develop a regex that will allow us to match escaped characters. Consider the code of listing 7.16.

Listing 7.16: Matching escaped characters in a CSS selector

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

  var pattern = /^(?:(\w+)|(\W))+$/;

  var test = "form.update.whatever";

  var r = test.match(pattern);
  console.log(r);

  r = pattern.exec(test);
  console.log(r);

  assert(pattern.test("form\\:update"),
         "Matching an escaped expression." );

</script>
```

This 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 as we're assuming that the user will be escaping their non-word characters).

7.7 Summary

Regular expressions permeate modern JavaScript development with virtually every aspect of modern development depends on their use in some way. With a good understanding of the advanced regex concepts that have been covered in this chapter, any developer should feel comfortable in tackling a challenging piece of code that uses regular expressions.

During the course of this chapter, we first took a whirlwind tour through regular expressions that gave us a good overview of exactly what they are, and how they are constructed. We learned what that various terms and operators that go into a regex mean, and how to combine them to form pattern-matching regular expressions.

Then we learned how to pre-compile regular expressions, and how doing so can give us an enormous performance gain over just letting a regex get recompiled every time that it is needed.

We learned how to use regular expression to test a string for a match against the pattern that the expression represents, and even more importantly, we learned how to capture the segments of the source string that were matched.

We learned how to use useful methods like the `exec()` method of regular expressions, as well as regex-oriented methods of String such as `match()` and `replace()`. And we did so while learning the difference between local and global regular expressions.

How the segments that we captured could be used as back-references and replacement strings was explained, along with how to avoid unnecessary captures with passive sub-expressions.

We examined the utility of providing a function to dynamically determine a replacements string, and then we rounded out the chapter with solutions to some common idioms such as string trimming and matching such characters as newlines and Unicode.

All told, that’s quite an arsenal of powerful tools to stuff into our ninja backpacks.

Back in the beginning of chapter 3, we talk about the event loop and stated that JavaScript executed all event callbacks in a single thread, each in their own turn. In the next chapter, we’re going to examine JavaScript threading in details, and discuss its effects upon timers and intervals.

8

Taming threads and timers

In this chapter:

- How JavaScript handles threading
- 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.

This chapter will take a look at how this all works.

8.1 How timers and threading 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.

8.1.1 Setting and clearing timers

JavaScript provides us with two methods to create timers, and two corresponding methods to clear (remove) them. All are methods of the `window` (global context) object.

They are described in table 8.1.

Table 8.1: JavaScript's timer manipulation methods

Method	Format	Description
`window.setTimeout`	`id = setTimeout(fn,delay)`	Initiates a timer that will fire exactly once after the delay has elapsed, executing the passed callback,. 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 at the specified delay interval, executing the passed function, 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 that fire periodically at a specified interval. In practice, most browsers 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.

An important concept that needs to be understood with regard to JavaScript timers is that the timer delay is not guaranteed. The reason for this has a great deal to do with the nature of JavaScript threading.

Let's explore that concept.

8.1.2 Timer execution within the execution thread

All JavaScript code in a browser executes in a single thread. One. Just one.

An intrinsic result of this fact is that the handlers for asynchronous events, such as interface events and timers, are only executed when there's nothing else already executing. This means that handlers must queue up for executing when a slot is available, and that no handler will ever interrupt the execution of another.

This is likely best demonstrated with a timing diagram, as shown in figure 8.1.

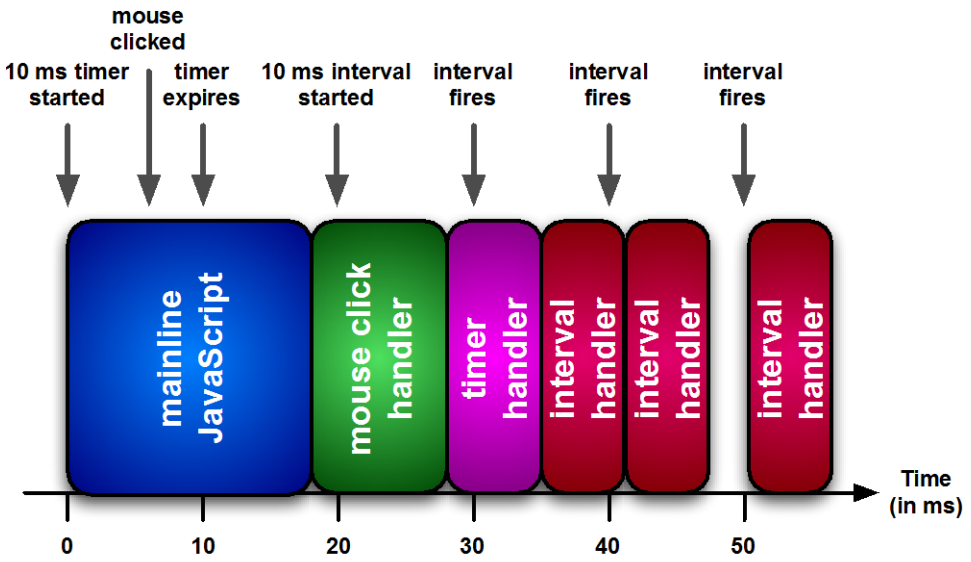

Figure 8.1: A timing diagram that shows how mainline code and handlers execute within a single thread

There's a lot of information to digest from figure 8.1, but understanding it completely gives us a better understanding 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 next 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 up 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 all of 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.

8.1.3 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 8.1.

Listing 8.1: 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 8.1 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.

8.2 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 8.2 and 8.3.

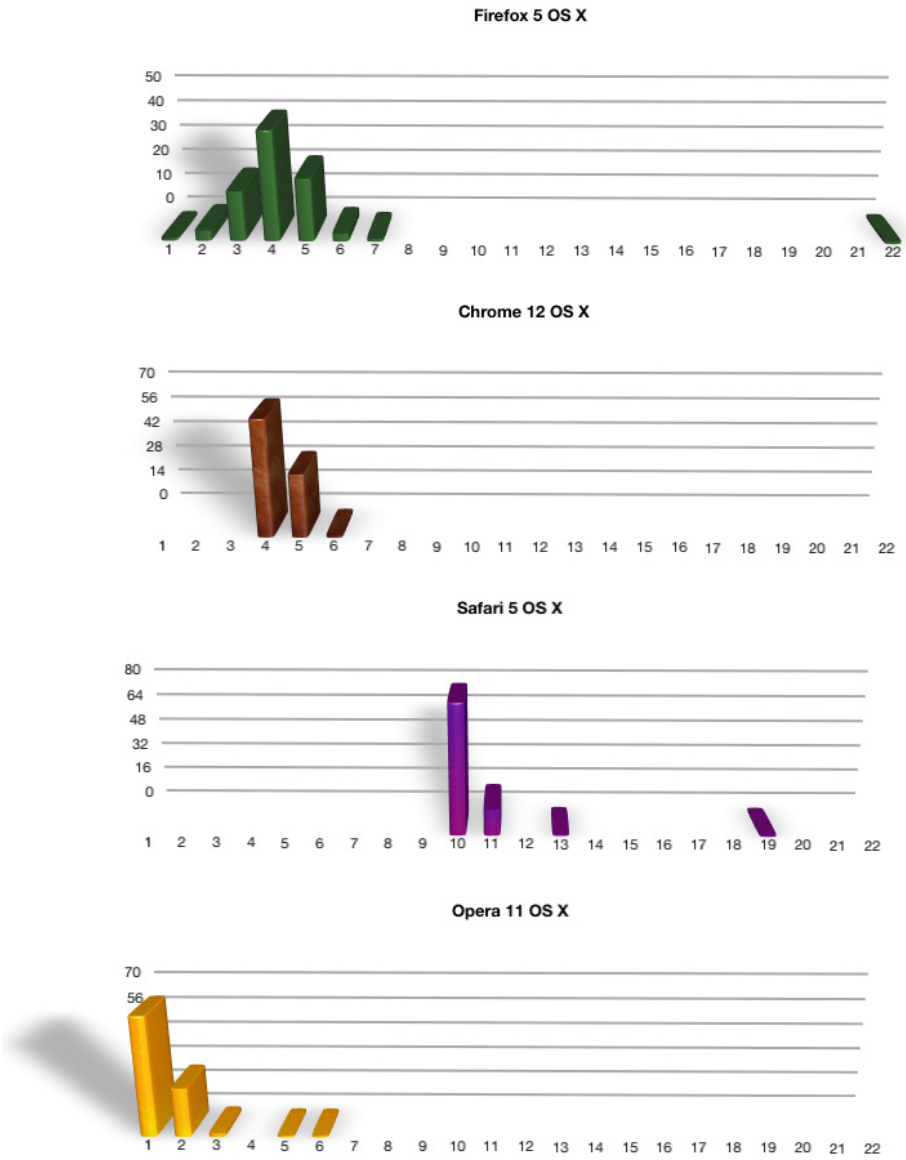

Figure 8.2: Interval timer performance measured on OS X browsers shows that some browsers get pretty close to 1 ms granularity

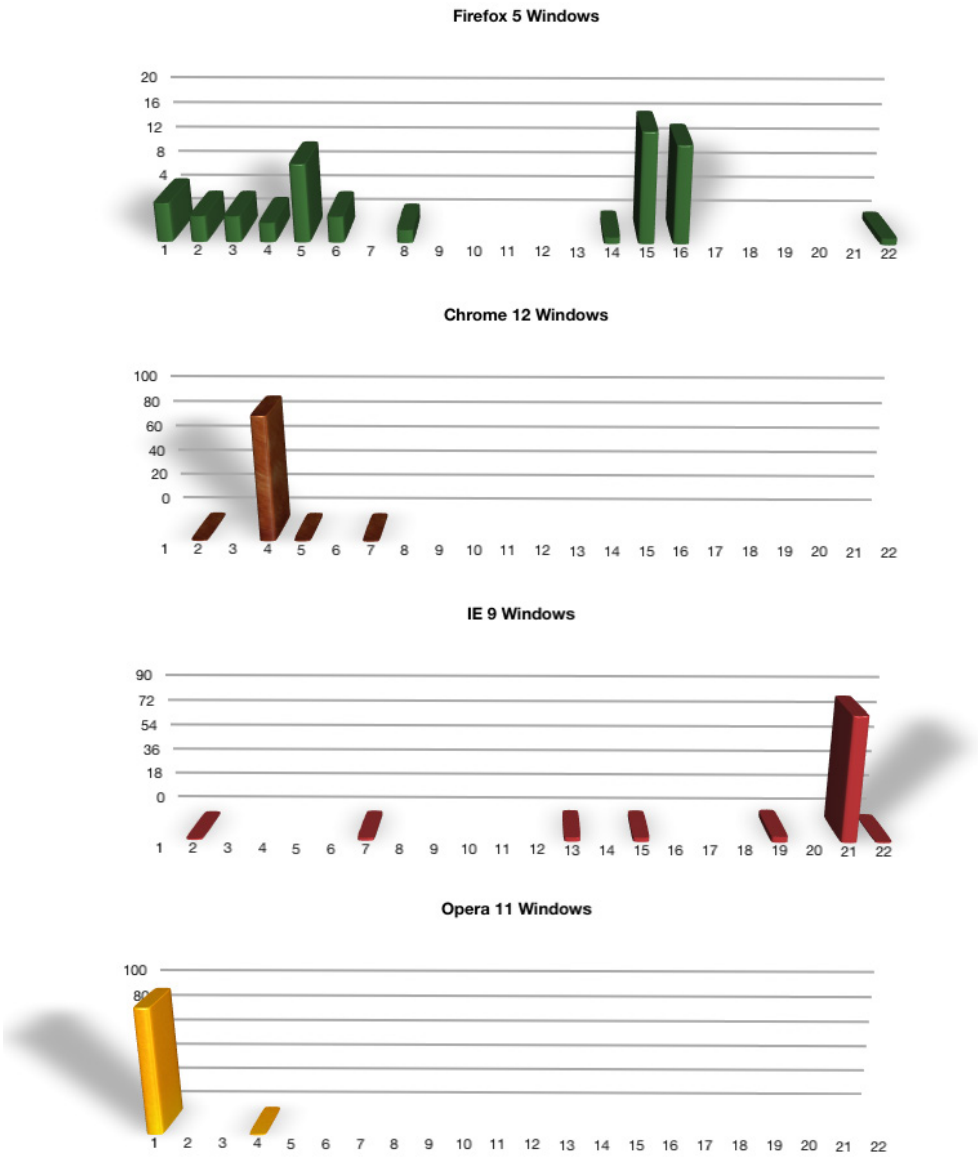

Figure 8.3 Interval timer performance as measured on Windows browsers is equally all over the place

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.

8.3 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 by breaking up the individual pieces of code into fragments that 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 8.2, in which a task is likely to take a long time.

Listing 8.2: 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 8.3.

Listing 8.3: 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, ten parts instead of four.

Also important to note is the little bit of math that 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 8.4.

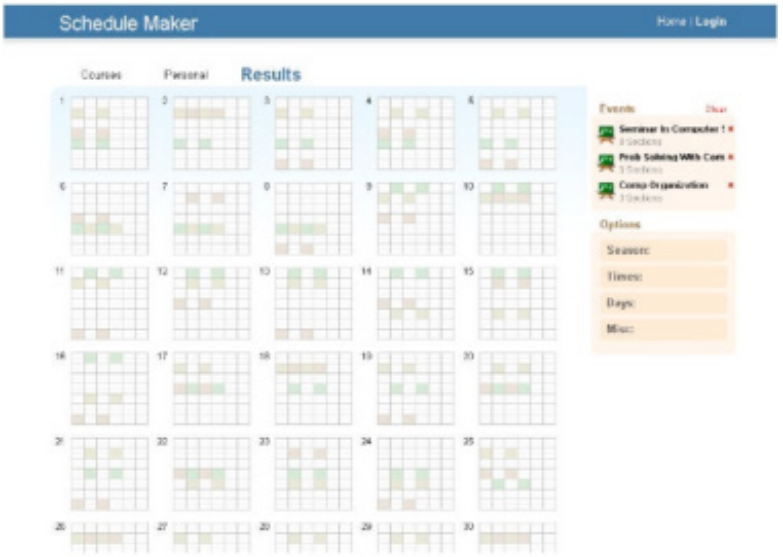

Figure 8.4: A web-based schedule generation application with client-side computation

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.

8.4 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 objects. Timers are a particular 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 stutter 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 our 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 8.4.

Listing 8.4: 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 8.4 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'll 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 8.5.

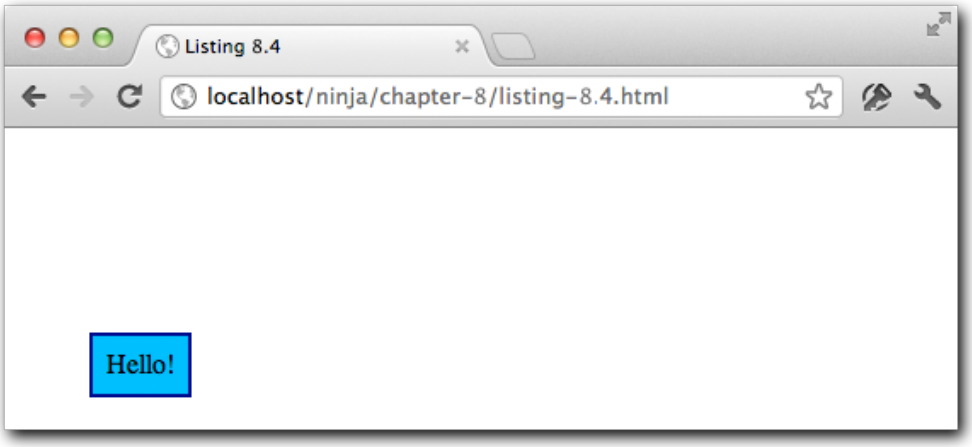

Figure 8.5: After running multiple animation handlers, the element has moved down and across the page

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.

8.5 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 need to do asynchronous testing, we need to break all of those tests out and handle them separately. Listing 8.5 has some code that does so. You should not be surprised to find that this code looks somewhat familiar.

Listing 8.5: 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 8.5 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 introduced with listing 8.4, 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 forces 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.

8.6 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 how JavaScript performs run-time evaluations and how we can harness that power to our own ends.

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 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. We'll learn about the various mechanisms that JavaScript provides to cause code to be evaluated at run time, then …

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.

These various means include:

- The `eval()` function

- Function constructors
- Timers
- The `<script>` element

While we examine each of these mechanisms, we'll discuss evaluation scope and then learn safe practices to keep in mind when evaluating code at run time.

Let's start by examining the most common way that page authors trigger code evaluation.

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, which attempts to prove these assertions.

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.

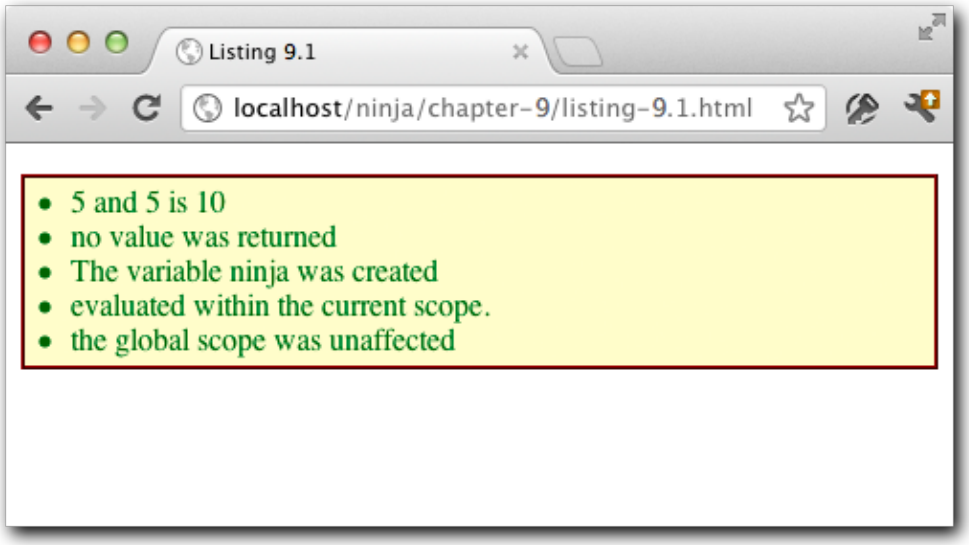

Figure 9.1 Proving that `eval()` can evaluated various expressions and is confined to the local scope

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, another 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()`. Note how in both cases, 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 8 or earlier, you may have gotten a nasty surprise. Versions of IE prior to IE9 have 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 versions of IE.

```
var fn = eval("false||function(){return true;}");
assert(fn() === true,
       "The function was created correctly.");
```

This particular issue is fixed in IE9.

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 define it using one of the means that 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 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 created with `eval()` inherit the closure of that scope – a ramification of the fact that `eval()` executes within the local scope.

It turns out that 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 omit the name to create anonymous functions.

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 of a variable argument list to the Function constructor is always the code that will become the body of the function. Any preceding arguments represent the names of the parameters for the function.

So our example is equivalent to:

```
var add = function(a,b) { return a + b; }
```

While these are functionally equivalent, a glaring difference is that in the Function constructor approach, the function body is provided by a run-time string.

Another difference that's vitally important to realize is 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 unneeded closures.

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 chapter 8, we'd pass an inline function or a function reference to a timer. This is the recommended usage of the `setTimeout()` and `setInterval()` methods, But these methods *also* accept strings which will be evaluated when the timers fire.

For example:

```
var tick = window.setTimeout('alert("Hi!")',100);
```

It's rather rare that we would need to use this behavior (it's roughly equivalent to using the `new Function()` approach) and it's use is discouraged 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; proving 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 it 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"); //fails!
```

If we expected the variable `test` to be created in the global scope as a result of the execution of the immediate function, 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 solution would be to change the code to be evaluated to:

```
eval("window.test = 5;");
```

While this *would* 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, a self-professed JavaScript Jedi and PHP Ninja,) developed a technique for making this work properly across multiple platforms.

> NOTE We will not even attempt to distinguish between what constitutes a Jedi versus a Ninja. We'll just leave it at that both exhibit a mastery of their chosen craft.

His original work can be found at:
http://webreflection.blogspot.com/2007/08/global-scope-evaluation-and-dom.html
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. For 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 on regular expressions if the statement doesn't make sense to you), and then, locating either the `<head>` element of the DOM or the document itself, we create a dis-attached `<script>` element (#2).

We 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.

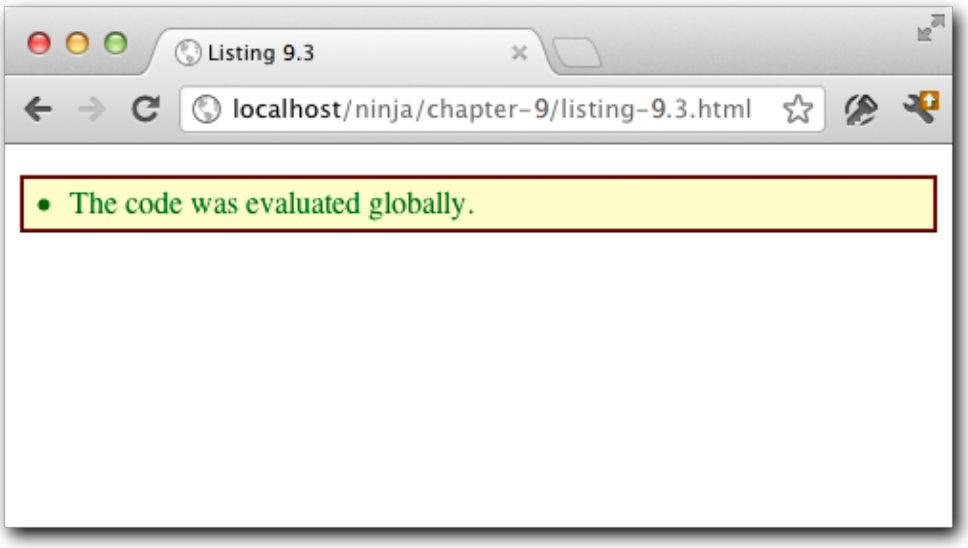

Figure 9.2 We can execute evaluated code in the global context using a bit of DOM manipulation trickery

A common use case for this code is when 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 coder might supply us with 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 hooligan 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 view, or cause other problems for us.

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 opposite direction?

9.2 Function decompilation

Most JavaScript implementations also provide a means to decompile already-compiled JavaScript code.

As complicated as this may sound, it's actually quite 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 exact 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 statements. First, the function decompiles the passed function

and uses a regular expression to extract the comma-delimited argument list (#1). (We covered regular expressions in chapter 7, if you need a refresher.)

Note that because the `exec()` method *expects* a string, we could have left the `.toString()` off the function argument, and it would have been implicitly called. But we included it here explicitly for clarity.

Then, the result of that extraction is split into its component values, performing checks 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.

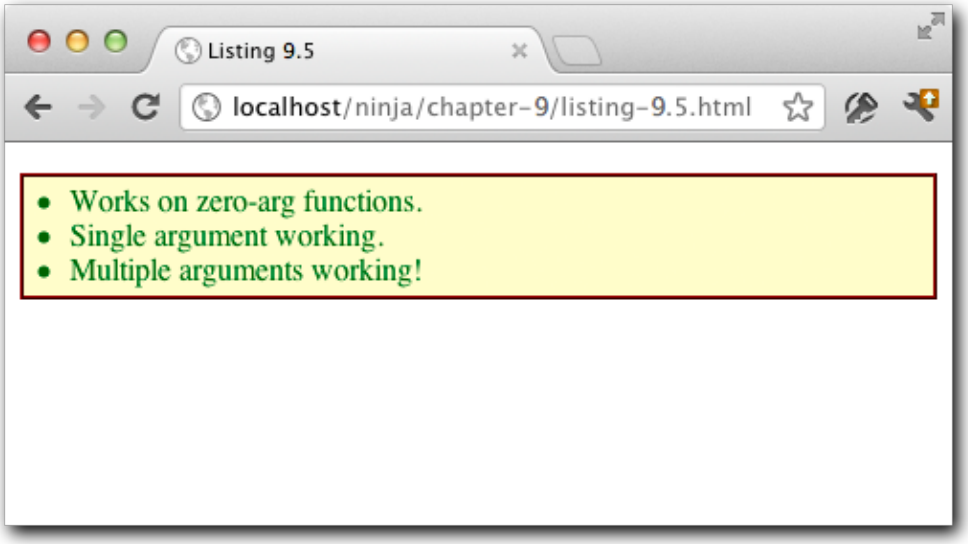

Figure 9.3 We can use function decompilation to do fancy things such as inferring the names of arguments to a function

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 (and particularly in upcoming chapters), we certainly don't want to resort to browser detection to determine if function decompilation is supported. Rather, we'll use feature simulation (which we'll discuss at length in chapter 11) 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 evaluation; 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.

Most modern browsers support the native JSON object with its `parse()` and `stringify()` methods, but a number of earlier browsers that do not provide this object are still in the wild. And for these browsers, it's still important to know how to deal with JSON without `window.JSON`.

But 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, as pointed out earlier, 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.
- Guarding 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 name-spaced code

In chapter 3, 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 name-spaced 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 simple tricks
- packed using Dean Edwards' Packer, with massive rewriting and decompression 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	518.7214	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 your goals are limited to performance, it may not be your best choice.

However, performance may not always be your number one focus. Even with the additional overhead, the Packer can be a valuable tool if *obfuscation* is what you are after. 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 activities such as image roll-overs, no one much cared that the code was shipped off to 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.

Let's move on to another activity that we might use run-time code evaluation for: rewriting code..

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, 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 10), 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>
```

Note that we're 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 9.8: 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 custom 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, custom 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 only the imagination of the page author.

Now 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:

- Why the with statement is controversial
- How `with` statements work
- Code simplification with `with`
- Tricky `with` gotchas
- Templating with `with`

The `with` statement is a powerful, frequently misunderstood, and controversial feature of JavaScript. A `with` statement allows us to put all the properties of an object within the current scope, allowing us to reference and assign to them *without* having to prefix them with the reference to their owning object.

Figuring out how to use this statement correctly can be tricky, but many feel that doing so gives us many advantages when writing our application code.

The `with` statement is, however, not without its detractors. One of these high-profile skeptics is none less than Douglas Crockford (JavaScript Ninja Extraordinaire, and inventor of JSON, and author of *JavaScript: the Good Parts*) who, in 2006, published a famous blog post titled "with Statement Considered Harmful" in which he states:

> If you can't read a program and be confident that you know what it is going to do, you can't have confidence that it is going to work correctly. For this reason, the `with` statement should be avoided.
>
> Douglas Crockford, April 2006

The full blog post is available at http://yuiblog.com/blog/2006/04/11/with-statement-considered-harmful/

Mr. Crockford is not alone. Many JavaScript editors and IDEs (Integrated Development Environments) will flag usages of `with` as warnings, advising against their use.

We're not going to come down on either side of the fence here. You're likely the run up against the `with` statement in code that's out in the wild, so it's something you should be familiar with even if you aren't planning to use it in your own code.

Besides, part of being a language Ninja is learning how to make your own decisions based upon facts and industry opinions. We urge you to read Mr. Crockford's post, read this chapter, and scour the Internet for other information and make up your own mind with regards to the advantages or harmfulness of `with` for yourself.

With that said (pun absolutely intended), let's learn about the `with` statement.

10.1 What's with `with`?

A with statement creates a scope within which the properties of a specified object can be referenced without the need of a prefix. Let's see how that all works.

10.1.1 Referencing properties within a `with` scope

To start off, let's jump right into a look at the basics of how the with statement works, as shown in Listing 10.1.

Listing 10.1: Creating a `with` scope using an object

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

  var use = "other";                                                    #1

  var katana = {                                                        #2
    isSharp: true,
    use: function(){
      this.isSharp = !this.isSharp;
    }
  };

  with (katana) {                                                       #3

    assert(use !== "other" && typeof use == "function",                 #4
           "use is a function from the katana object.");
    assert(this !== katana,
           "context isn't changed; keeps its original value");

  }

  assert(use  === "other",                                              #5
         "outside the with use is unaffected.");
  assert(typeof isSharp === "undefined",
         "outside the with the properties don't exist.");

</script>
```

#1 Defines a top-level variable
#2 Creates an object
#3 Establishes a `with` scope
#4 Tests inside the scope
#5 Tests outside the scope

In the code of the above listing we can see how the properties of the katana object are introduced into the scope created by the `with` statement. Within the scoped, we can reference the properties directly without the `katana` prefix as if they were top-level variables and methods.

To verify this we start by defining a top-level variable with the name `use` (#1). Then we create an inline object that has a property with that same name, `use`, along with another named `isSharp` (#2). This object is reference by a variable named `katana`.

Things get interesting when we establish a `with` scope using `katana` (#3). Within this scope, the properties of katana can be reference directly without the `katana` prefix. We test this (#4) by verifying that a reference to `use` does not have the value of the top-level variables named `use`, and that it is a function as we'd expect if the reference to use pointed to the `katana.use()` method.

The display of figure 10.1 shows that the assertions pass.

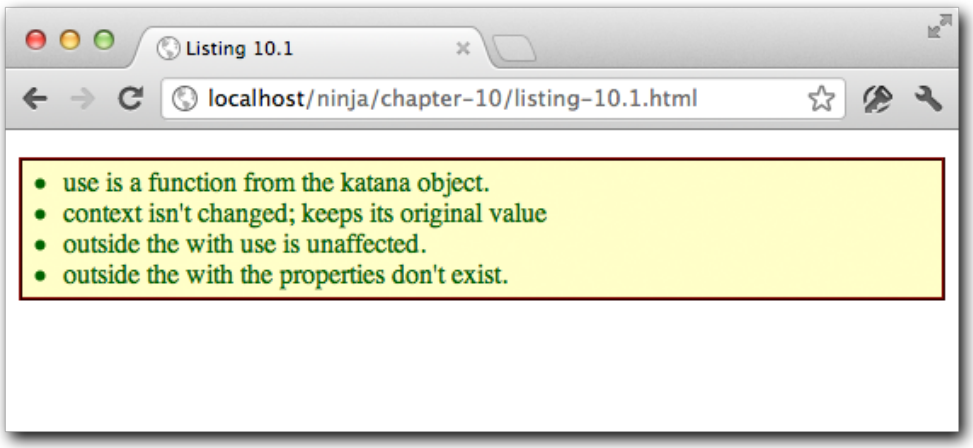

Figure 10.1 With a `with` statement, we can cause simple references to resolve to an object's properties

Our testing continues outside the scope of the `with` statement (#4) verifying that references to `use` refer to the top-level variable, and that the `isSharp` property is no longer available.

Note that within the scope of the `with` statement the object's properties take absolute precedence over variables of the same name defined in higher-level scopes.

We have also proven that the function context (`this`) is unaffected by the statement.

OK, that covers reading the values of properties, what about writing to them?

10.1.2 Assignments within a `with` scope

Let's take a look at an example of assignments within a `with` scope in listing 10.2:

Listing 10.2: Assignments in `with` scopes

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

  var katana = {                                                  //#1
    isSharp: true,
    use: function(){
      this.isSharp = !this.isSharp;
    }
  };

  with (katana) {
    isSharp = false;                                              //#2

    assert(katana.isSharp === false,                              //#3
        "properties can be assigned");

    cut = function(){                                             //#4
      isSharp = false;
    };

    assert(typeof katana.cut == "function",                       //#5
        "new properties can be created on the scoped object");
    assert(typeof window.cut == "function",
        "new properties are created in the global scope");

  }

</script>
```

#1 Creates object
#2 Assigns to existing property
#3 Tests assignment
#4 Attempts to create a new property
#5 Tests assignment

In this code we create the same `katana` object as in out previous test, with `use` and `isSharp` properties (#1), then we once again create a `with` scope using that object. But instead of referencing the properties, we're going to try some assignments.

First, we assign a value of `false` to the `isSharp` property (#2). If `isSharp` resolves to the `katana` property, we'd expect that the value of the property would be flipped from its initial value of `true` to `false`. We explicitly test the property (#3) and peeking ahead to figure 10.2, we see that this test passes. This proves that we can use un-prefixed references to the object's properties for both reading and for assignments.

Then we try something a little less straightforward: we create a function and assign it to a new reference named `cut` (#4). The question is: which scope will this new property be created within? Will it be created on `katana` because the assignment is within the `with`

scope? Or will it be created in the global scope (the `window` object) as we'd have expected outside of any `with` scope?

To find out which of these situations transpires, we write two tests (#5) only one of which can succeed. The first test asserts that the property will be created within `katana`, while the second test asserts that the property will be created in the global scope.

Figure 10.2 clearly shows that, because the second test is the one that passes, that unreferenced assignments that are not to an existing property on the with scopes' object are made to the global context.

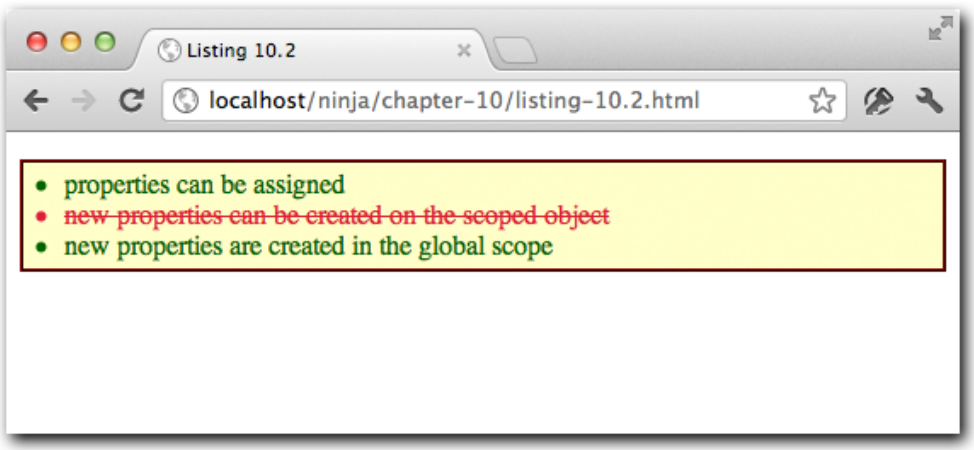

Figure 10.2 Test results show that un-prefixed references can't be used to create new properties

If we wanted to create the new property on `katana`, we'd need to use the object reference prefix, even though we are within a `with` scope, as follows:

```
katana.cut = function(){
  isSharp = false;
};
```

OK, that's easy enough – that's what we'd need to do without a `with` scope in any case. But it points out that care needs to be taken within `with` scopes as a simple typo in a property name can lead to strange and unexpected results; namely that a new global variable will be introduced rather than modifying the intended existing property on the `with` scope's object. Of course, this is something we need to generally be aware of, so we'll just need to carefully test our code, as always.

What other considerations should we be aware of?

10.1.3 Performance considerations

There's another major factor that we must be aware of when using `with`: it slows down the execution performance of any JavaScript that it encompasses. And that's not just limited to objects that it interacts with. Let's look at a timing test as shown in Listing 10.3.

Listing 10.3: Performance testing the `with` statment

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

  var ninja = { foo: "bar" },                          #1
      value,
      maxCount = 1000000,
      n,
      start,
      elapsed;

  start = new Date().getTime();                        #2
  for (n = 0; n < maxCount; n++) {
    value = ninja.foo;
  }
  elapsed = new Date().getTime() - start;
  assert(true,"Without with: " + elapsed);

  start = new Date().getTime();                        #3
  with(ninja){
    for (n = 0; n < maxCount; n++) {
      value = foo;
    }
  }
  elapsed = new Date().getTime() - start;
  assert(true,"With (with access): " + elapsed);

  start = new Date().getTime();                       #4
  with(ninja){
    for (n = 0; n < maxCount; n++) {
      foo = n;
    }
  }
  elapsed = new Date().getTime() - start;
  assert(true,"With (with assignment): " + elapsed);

  start = new Date().getTime();                       #5
  with(ninja){
    for (n = 0; n < maxCount; n++) {
      value = "no test";
    }
  }
  elapsed = new Date().getTime() - start;
  assert(true,"With (without access): " + elapsed);

</script>
```

#1 Sets up variables
#2 Tests without `with`
#3 Tests referencing
#4 Tests assignments

#5 tests with no access

For these performance tests, we set up a number of variables, including one (`ninja`) that will be the target of the `with` scope (#1). Then we run 4 performance tests, each performing an action a quarter of a million times and displaying its results:

- (#2) This test assigns the value of the `ninja.foo` property to the variable value without declaring any with scope.
- (#3) This test performs the same assignment as the first test, except that the assignment takes place within a with scope, and does not prefix the reference to property foo.
- (#4) This test assigns a value (the loop counter) to the foo property within a with scope, and without prefixing the property reference.
- (#5) This final test performs an assignment to the variable value within a with scope, but without any access to the ninja object at all.

The results of running theses tests is shown in table 10.1. All tests were run on the listed browsers executing on a MacBook Pro running OS X Lion 10.7.3 with a 2.8 GHz Core i7 processor and 8GB of RAM. The IE test was executed on a Windows 7 instance running in a Parallels virtual machine. All times are in milliseconds.

Table 10.1: Results of running the performance tests of listing 10.3

	no with scope	with reference	with assignment	with no access
Chrome 17	22	1386	1265	1240
Safari 1.5.3	6	266	265	251
Firefox 10.0.1	282	750	792	677
Opera 11.61	13	677	679	623
IE 9	13	173	157	139

The results are dramatic and could be rather surprising. Not only are there wildly varying times across the browsers (it's clear who's been focusing on JavaScript performance), but across the tests.

Regardless of which browser the tests were executed within, the code executed within with scopes was dramatically slower. This might not be too surprising for the tests in which the `with` scope's object was referenced or assigned, but look at the times in the right-most column whose test performed no access to the object at all. The mere fact that the code was inside a with scope dramatically slowed it down, by as much as a factor of 41, even though there was no access to the scoped object at all!

We must be sure that we are comfortable with this level of extra overhead when we decide that we want to enjoy any convenience that the with statement brings to the party. And obviously, the `with` statement is completely out of the picture for code in which performance is a major consideration.

10.2 Real-world examples

Inarguably the most common use case for using `with` is for the convenience of not having to duplicate variable references for property access. JavaScript libraries frequently use this as a means of simplifying statements that would otherwise be visually complex.

Here are a few examples from a couple of the major libraries, starting with one from Prototype.

```
Object.extend(String.prototype.escapeHTML, {
  div:  document.createElement('div'),
  text: document.createTextNode('')
});

with (String.prototype.escapeHTML) div.appendChild(text);
```

Here, Prototype uses `with` to simply avoid having to prefix references to the `div` and `text` properties of `String.prototype.escapeHTML`; which we must admit is a mouthful of a prefix.

The following example is from 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";
}
```

In this snippet, base2 again uses `with` as a simple means of not having to repeat a lengthy prefix, in this case `document.body.style`, again, and again. This allows for some super-simple modification of a DOM element's style object.

Another example from base2 is:

```
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 example from base2 uses `with` as a means of simply accessing instance properties. Normally this code would be much longe,r but the terseness that `with` is able to provide adds some much-needed clarity.

The final example is from the Firebug developer extension for Firefox.

```
const evalScriptPre = "with(__scope__.vars){ with(__scope__.api){" +
  " with(__scope__.userVars){ with(window){";
const evalScriptPost = "}}}}";
```

These lines 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 how it can make simplify complex pieces of code.

One especially interesting takeaway from the Firebug example, is the dual-use of `with`, bringing precedence to the window object over other objects.

Back in listing 10.1 we saw that normally when there is a name collision, the object of the `with` scope takes precedence over the global values. Structuring code as follows:

```
with ( x ) { with ( window ) { ... } }
```

allows us to have the `x` object's properties be introduced by `with`, while still allowing global variables to take precedence in the event of a name collision.

Now let's see another use to which we might put the `with` statement.

10.3 Importing name-spaced code

As previously shown, one of the most common uses for the `with` statement is to simplify statements that have numerous references to object properties. We can see this frequently in name-spaced code in which objects are defined within objects in order to provide an organized structure and naming to the code.

A side effect of this technique is that it can sometimes become rather tedious to re-type the object namespace names again and again.

Observe the two statements in the following snippet, which both perform the same operation using the Yahoo UI JavaScript library. The first statement is as we would write it without the use of `with`, and the second with `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 the single `with` statement allows for a considerable increase in code simplicity.

Now lets see if the `with` statement has anything to say for itself when it comes to testing.

10.4 Testing

When testing pieces of functionality within a test suite there're a couple things that we end up always having to watch out for. The primary of these is attending to the synchronization between the assertion methods and the test case currently being run. Typically this isn't much of a problem, but it can become troublesome when we begin dealing with asynchronous tests.

A common solution this issue 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 such a central object as the context to each test run. The object contains all the needed assertion methods, and easily collects the results back to s central location. We can see an example of this in the following snippet:

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

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 have 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.

Now here's an advanced use of `with` that you might not have thought to consider: templating.

10.5 Templating with `with`

The final, and likely most compelling, example of using `with` that we'll consider is within a simplified templating system. The customary goals for a templating system usually include:

- There should be a way to both run embedded code and to print out data.
- There should be a means of caching compiled templates.
- It should (perhaps, most importantly) be simple to easily access mapped data.

This last point is where `with` becomes especially useful.

Before we look at how with is used in the implementation, let's take a look at a template that uses the templating system, shown in listing 10.4.

Listing 10.4: A sample template to generate an HTML page

```
<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>
```

Within this template, the special delimiters are used to indicate what's embedded JavaScript code (`<%` and `%>`) and expressions to be evaluated (`<%=` and `%>`). The Java-savvy may recognize that these delimiters match that of old-style JSP 1 templating delimiters (JSP 2 replaced these with a more modern version in 2002).

Now let's see the implementation as shown in listing 10.5.

Listing 10.5: A templating solution using `with`

```
(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." );
```

We aren't going to go into a lot of detail the implementation details of the templating system. Even though it doesn't use any concepts that we haven't already covered, it's put together in a rather sophisticated manner and shouldn't make you feel bad if you don't completely grok at this point. The important point is how a `with` scope is used (line 32) to provide the properties of the passed data object to the template. This allows the properties of the data object to be referenced within the template as if they were top-level variables.

While complex, 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 and reusable, system. This is made possible largely in part due to the existence of the `with` statement, to allow the properties to be easily referenced within the templates.

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 data into.

The full templating system is pulled together with the use of embedded templates. There's a great loophole, that we've exploited already, provided by modern browsers and search engines: `<script>` elsements 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 later on.

The total result of this templating system is one that is easy to use due, in large part, to the abilities of the `with` statement.

10.6 Summary

The glaringly obvious point that we learned from this chapter is that the primary goal of the `with` statement is to simplify complex code by allowing properties of an object to be referenced without the need for a reference to the object holding the properties. This can make code that contains many references to the object's properties a lot terser and easily understandable.

We saw how this simplification could be applied to areas such as name-spacing, testing, and even building templating systems, and some examples of how `with` is used by some of the popular JavaScript libraries.

As with all powerful tools, discretion should be exercised when using `with`; it's just as easy to obfuscate code using this features as it would be to clarify it. But having a good understanding of how `with` scopes can be used allows us to decide when it is proper to use, and when it should be avoided.

During the course of this chapter, we didn't run across any browser incompatibilities with respect to using `with` scopes, but we certainly ran into our share in the chapters leading up to this one. In the next chapter, we're going to discuss ways for coping with cross-browser madness while retaining our own sanity.

11 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 pain points when it comes to making sure that the code works flawlessly across the set 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 even be envisioned.

Coding for multiple browsers 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, getting the biggest bang for the buck from those resources.

This starts with choosing our supported browsers carefully.

11.1 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:

- That we'll actively test against that browser with our test suite.
- That we'll fix bugs and regressions associated with that browser.
- 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 a 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 a library and aims to help you choose which browsers to support in your own code.

To help you decide upon a supported browser set, you might want to make a *browser support matrix*, as shown in figure 11.1, 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 11.1: An example "browser support" chart – fill one in with your own decisions!

Any piece of reusable JavaScript code, to include mass-consumption JavaScript libraries as well as our 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, with regards to cross-browser support, throughout this chapter, and then examine some of the best ways to write that code with the aim of alleviating 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.

11.2 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 that pose the biggest challenges to our reusable JavaScript code.

The five points of concern for JavaScript development are shown in figure 11.2.

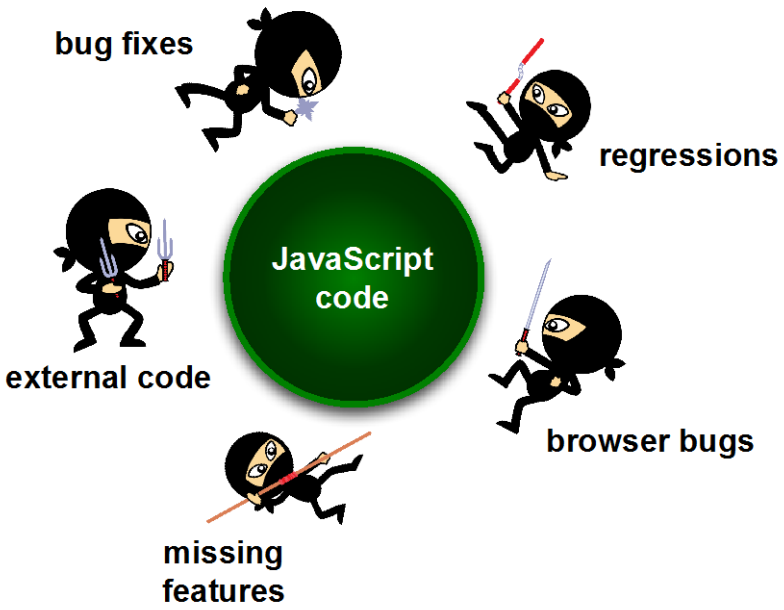

Figure 11.2 The five major points of concern for development of reusable JavaScript

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, and 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 the challenges that we're up against, and how to combat them.

11.2.1 Browser bugs and differences

A primary concern that we'll need to deal with when developing reusable JavaScript code regards 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 straightforward, having already been presented in chapter 2 and used throughout this book: 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 11.3.

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?

11.2.2 Bug fixes

Assuming that a browser will forever present a particular bug is rather foolhardy – most bugs eventually get fixed – and counting upon the presence of the bug is a dangerous development strategy. It's best to use the techniques that we'll discuss in section 11.3 to make sure that we future-proof any bug 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 11.3.3) instead of making assumptions about the browser.

The problem with handling browser bugs leisurely 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 that 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 11.3.

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 (see section 11.3.3). 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.

11.2.3 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 cohabit the page 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 – we'll pick `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 these 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 the 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 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 yourself". While this may seem a rather 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. This practice, well-meant or otherwise, can lay traps for us to step into.

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 end up not working as intended through absolutely 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 types of problems.

The next few sections will introduce these defensive steps.

AVOIDING IMPLANTED PROPERTIES

The first of these defensive steps is to learn how to avoid properties that other code may have introduced into objects behind our backs.

In order to detect such activity we'll 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 use 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 11.1.

Listing 11.1: 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 11.3.

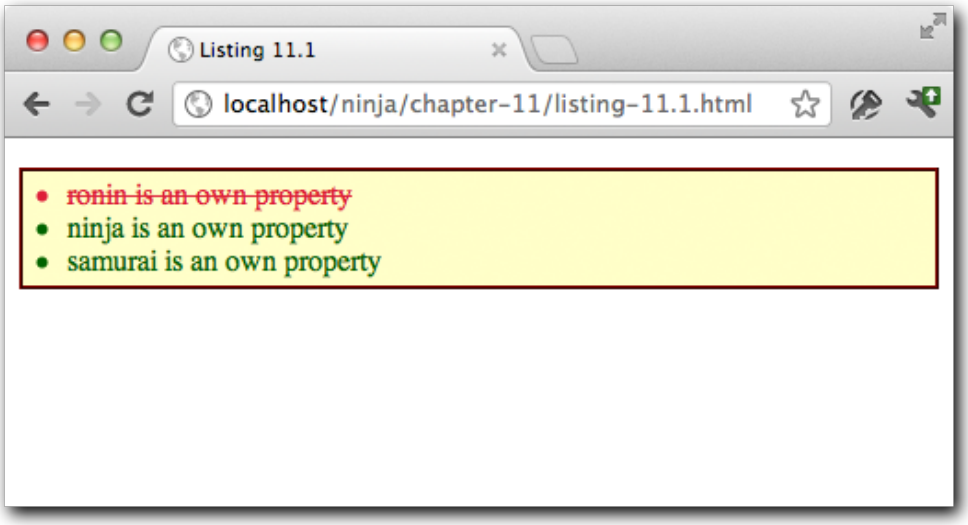

Figure 11.3 Results of tests show how we can use `hasOwnProperty()` can detect inherited properties

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 that they cause can be great if the 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, as in:

```
for (var p in someObject) {
  if (someObject.hasOwnProperty(p)) {
    // do something wonderful
  }
}
```

This snippet shows how using `hasOwnProperty()` can be used to ignore properties that have been added to the object's prototype.

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 absolutely 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 the numeric length value of the list returned by the method, instead contains a reference to the input element with the `id` of `length`.

Huh?

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`.

What's happened is that the wayward browsers have added references to the input elements within the form to the `<form>` element using the id values of the input elements as the property names, even if those names overwrite already-existing properties.

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 include 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 last few 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.

We'll next move on to the next point of concern.

11.2.4 Missing Features

For browsers that might not be 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 needed 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 older and 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.

11.2.5 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.

However, 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 11.2.

Listing 11.2: 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 use 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.

Most API future changes, alas, 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 we 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 information on 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 arise.

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.

OK, now that we know about the specific challenges that we are facing and some ways to meet them, let's explore some strategies that can help us across multiple development concerns.

11.3 Implementation strategies

Knowing which issues to be aware of is only half the battle – figuring out effective strategies for dealing with them, and using then to implement 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 robust code bases.

We'll start with something that's easy and almost trouble free.

11.3.1 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.

11.3.2 Object detection

As we've previously discussed, *object detection* is a commonly used approach when writing cross-browser code, being not only simple but also 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, the code that we saw in listing 11.2 in which 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 use of object detection is discovering 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?

11.3.3 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 doesn't guarantee that 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 8 and earlier 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, and as we hope often happens, this bug has been fixed by the Internet Explorer team in the IE 9 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 11.3 shows a use of the feature simulation snippet presented above in a useful context: working around the bug.

Listing 11.3 Putting feature simulation into practice to work around a browser bug

```
<!DOCTYPE html>
<html>
  <head>
    <title>Listing 11.3</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. Modern browsers (to include versions 8 and 9 of IE) define a set of constants on the Node object, such as Node.ELEMENT_NODE and Node.COMMENT_NODE. As our fixe will be triggered in older browsers, we cannot assume that these constants exist and we have used hard-coded values.

You can find a complete list of node type 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.

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 can we do about browser problems that stubbornly resist being tested?

11.3.4 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 for. These situations are fortunately rather rare, but when we encounter them, it will always pay to spend some time investigating the matter to see if there's something we can do about it.

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 bowsers 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 as we create them.

EVENT FIRING

Another of these aggravations is 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 already happened some time ago. As there's no way to determine that the event has 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 as 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 11.3, 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 can frequently 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.

11.4 Reducing 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 behind.

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.*)

(*Jeopardy Theme concludes.*)

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.

11.5 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 what is 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.

12

Cutting through attributes, properties and CSS

In this chapter:

- Understanding DOM attributes and DOM properties
- Dealing with cross-browser attributes and styles
- Handling element dimension properties
- Discovering computed styles

Excepting the previous chapter, a large percentage of the chapters you've read up to this point deal with JavaScript The Language. And while there are plenty of nuances to pure JavaScript as a a language, once we throw the browser DOM into the mix, things can really get confusing.

Understanding DOM concepts and how JavaScript relates to these concepts is an important part of becoming a JavaScript ninja – especially considering some of the seemingly baffling ways that some DOM concepts seem to defy logic. The area of DOM attributes and properties is one of those conceptual areas that has left many a JavaScript page author quivering with confusion.

And that's no small wonder, as not only is there some very nuanced behavior between attributes and properties, but there are few areas that are more riddled with bugs and cross-browser issues.

They're an important concept however as attributes are an important part of how the DOM gets built, and properties are the primary means by which elements holds information and by which run-time information can be retained.

Let's take a look at a quick example that demonstrates the capacity for befuddlement.

```
<img src="../images/ninja-with-nunchuks.png">
<script type="text/javascript">
```

```
    var image = document.getElementsByTagName('img')[0];
    image.src = '../images/ninja-with-pole.png';
    console.log(i.src);
    console.log(i.getAttribute('src'));
</script>
```

In this snippet, we create an image tag, then get a reference to it and change it's `src` property to a new value. That seems pretty straight-forward, but when we view the console output (this snippet assumes that we're running with a script debugger active) we see that the property isn't the value that we assigned, but rather:

```
http://localhost/ninja/images/ninja-with-pole.png
```

If we assigned a value to a property, should we not expect it to have that exact value?

Even more oddly, even though we didn't change the attribute on the element, the console out shows that the value of the `src` *attribute* has changed to:

```
../images/ninja-with-pole.png
```

What gives?

In this chapter, we'll examine all the conundrums that the browsers throw at us with respect to element properties and attributes and discover why the results weren't exactly what we might expect.

The same goes for CSS and the styling of elements. Many of the complications that we run into when constructing a dynamic web application stem from the complications of setting and getting element styling. While this book can't cover all that is known about handling element styling (that's enough to fill an entire other book), the core essentials will be discussed.

Let's start by understanding exactly what element attributes and properties are.

12.1 DOM attributes and properties

When accessing the values of element attributes we actually have two possible options: using the traditional DOM methods of `getAttribute` and `setAttribute`, or using properties of the DOM objects that correspond to the attributes.

For example, to obtain the `id` of an element whose reference is stored in variable `e`, we could use either of the following:

```
e.getAttribute('id')
e.id
```

Either way will give us the value of the `id`. Let's examine the code of listing 12.1 to better understand how attribute values and their corresponding properties behave.

Listing 12.1: Accessing attribute values via DOM methods and properties

```
<div></div>

<script type="text/javascript">

  window.onload = function(){

    var div = document.getElementsByTagName("div")[0];    #1
```

```
        div.setAttribute("id","ninja-1");                    #2
        assert(div.getAttribute('id') === "ninja-1",
               "Attribute successfully changed");

        div.id = "ninja-2";                                  #3
        assert(div.id === "ninja-2",
               "Property successfully changed");

        div.id = "ninja-3";                                  #4
        assert(div.id === "ninja-3",
               "Property successfully changed");
        assert(div.getAttribute('id') === "ninja-3",
               "Attribute successfully changed via property");

        div.setAttribute("id","ninja-4");                    #5
        assert(div.id === "ninja-4",
               "Property successfully changed via attribute");
        assert(div.getAttribute('id') === "ninja-4",
               "Attribute successfully changed");

    };

</script>
```

#1 Obtains element reference
#2 Tests DOM method
#3 Tests property value
#4 Tests property/attribute correspondence
#5 Tests more property/attribute correspondence

This example shows some interesting behavior with respect to element attributes and element properties. It starts by defining a simple `<div>` element that we'll use as a test subject.

Within the page's load handler (to ensure that the DOM is done being built) we run a few tests, after obtaining a reference to the lone `<div>` element (#1).

In our first test (#2), we set the `id` attribute to the value "ninja-1" via the `setAttribute()` method. Then we assert that the value returned from getting that attribute via `getAttribute()` returns the same value. It should be no surprise to find that this test works just fine when we load the page.

Similarly, in the next test (#3), we set the `id` property to the value "ninja-2" and then verify that the property value was indeed changed. No problem.

The next test (#4) is when things get interesting. We again set the value of `id` property to a new value, in this case "ninja-3", and again verify that the property value was changed. But then we also assert that not only should the property value have changed, but also the value of the `id` *attribute*.

Both assertions pass. From this we learn that `id` property and the `id` attribute are somehow linked together. Changing the `id` property value also changes the `id` attribute value.

The next test (#5) proves that it also works the other way around: setting an attribute value also changes the corresponding property value.

But don't let this fool you into thinking that the property and attribute are sharing the same value – they aren't. We'll see later in this chapter that the attribute and corresponding property, while linked, are not always identical.

There are five important points to consider with respect to attributes and properties:

1. Cross-browser naming
2. Naming limitations
3. HTML versus XML differences
4. Custom attribute behavior
5. Performance

Let's examine each of these points one by one.

12.1.1 Cross-browser naming

When it comes to the naming of attributes and their corresponding properties, the naming of the properties is generally more consistent across the browsers. If we're able to access a property by a certain name in one browser, there's a good chance of it being the same in other browsers as well. There are *some* differences, but there tend to be more difference with the naming of the attributes than with the naming of the properties.

For example, while the `class` attribute can be obtained as "class" in most browsers, Internet Explorer requires "className". This is likely because (as we'll see in just a bit) the name of the property is `className`, and so within IE, the name of the attribute and the property are consistent. Consistency is usually a good thing, but the naming difference across browsers can be quite frustrating.

Libraries such as jQuery help to normalize these naming discrepancies by allowing us to specify one name regardless of platform, and performing any necessary translation behind the scenes on our behalves. But without library assist, we need to be aware of the differences and write our own code accordingly.

12.1.2 Naming restrictions

Attributes, being referenced by strings passed to DOM methods, have a pretty free reign as to what they can be named. But properties, which can be represented as identifiers using the dot operator notation, are more restricted as there are some reserved words that are disallowed.

The ECMAScript Specification (found at http://www.ecma-international.org/publications/standards/Ecma-262.htm) states that certain keywords cannot be used as property names, so alternatives have been defined. As examples, the `for` attribute of `<label>` elements is represented by the `htmlFor` property as `for` is a reserved word, and the `class` attribute of all elements is represented by the `className` property as `class` is also reserved.

Additionally, attribute names that are composed of multiple words, such as `readonly` are represented by camel case property names; `readOnly` in this case.

Some examples of these differences can be found in Table 12.1.

Table 12.1: Cases where property names and attribute names differ

Attribute name	Property name
`for`	`htmlFor`
`class`	`className`
`readonly`	`readOnly`
`maxlength`	`maxLength`
`cellspacing`	`cellSpacing`
`rowspan`	`rowSpan`
`colspan`	`colSpan`
`tabindex`	`tabIndex`

Note that HTML 5 adds new elements and attributes that may need to be added to this list when the dust settles.

12.1.3 XML differences from HTML

The whole notion of properties that automatically correspond to attributes is a peculiarity of the HTML DOM. When dealing with an XML DOM, no properties are automatically created on the elements to represent attribute values.

Therefore, we'll need to use the traditional DOM attribute methods to obtain attribute values. This isn't a horrible imposition because XML documents usually don't exhibit the normal litany of naming mistakes that you see from DOM attributes in HTML documents.

It's a good idea to put some form of a check in our code to determine if an element (or document) is an XML element (or document) so we know how to proceed appropriately. An example of this type of check is shown by the following function:

```
function isXML(elem) {
  return (elem.ownerDocument || elem)
    .documentElement.nodeName !== "HTML";
}
```

This function will return `true` if the element is an XML element, and `false` otherwise.

12.1.4 Behavior of custom attributes

Not all attributes are represented by element properties. While it's generally true for attributes that are natively specified by the HTML DOM, *custom attributes* that we may place on the elements in our pages do not automatically become represented by element properties. So when accessing the value of a custom attribute, we need to use the DOM methods `getAttribute()` and `setAttribute()`.

If you're not sure if a property for an attribute exists or not, you can always test for it and fallback to the DOM methods if it does not exist.

For example:

```
var value = element.someValue ? element.someValue :
                                getAttribute('someValue');
```

TIP In HTML 5, use the prefix `data-` for all custom attributes to remain valid in the eye of the HTML 5 Specification. It's recommended you do this even if still using HTML 4 in order to future-proof your markup. Besides, it's a good convention that clearly separates custom attributes from native attributes.

12.1.5 Performance considerations

In general, property access is faster than the corresponding DOM attribute operations, especially in Internet Explorer. Let's prove that to ourselves.

Remember back in chapter 2, when we talked about performance testing? The way that we do that is to measure how long a large number of an operation takes. We can't just measure the performance of a single operation; the duration is far too short for a millisecond time (harken back to the timer discussion of chapter 8) to capture.

So if one operation is too quick to measure, what about five million of them? And that's exactly what the code of listing 12.2 measures.

Listing 12.2 Comparing the performance of DOM methods versus properties

```
<div id="testSubject"></div>

<script type="text/javascript">

  var count = 5000000;                                          #1
  var n;                                                        #1
  var begin = new Date();                                       #1
  var end;                                                      #1
  var testSubject = document.getElementById('testSubject');     #1
  var value;                                                    #1

  for (n = 0; n < count; n++) {                                 #2
    value = testSubject.getAttribute('id');
  }
  end = new Date();
  assert(true,'Time for DOM method read: ' +
      (end.getTime() - begin.getTime()));

  begin = new Date();                                           #3
  for (n = 0; n < count; n++) {
    value = testSubject.id;
  }
  end = new Date();
  assert(true,'Time for property read: ' +
         (end.getTime() - begin.getTime()));

  begin = new Date();                                           #4
```

```
  for (n = 0; n < count; n++) {
    testSubject.setAttribute('id','testSubject');
  }
  end = new Date();
  assert(true,'Time for DOM method write: ' +
         (end.getTime() - begin.getTime()));

  begin = new Date();                                    #5
  for (n = 0; n < count; n++) {
    testSubject.id = 'testSubject';
  }
  end = new Date();
  assert(true,'Time for property write: ' +
         (end.getTime() - begin.getTime()));


</script>
```

#1 Sets up variables in advance
#2 Conducts test of DOM method read
#3 conducts test of property read
#4 Conducts test of DOM method write
#5 Conducts test of property write

This code conducts a performance test of the DOM `getAttribute()` and `setAttribute()` methods, against similar operation reading and writing the corresponding property.

Running this test on multiple browsers, the results that we gathered are shown in table 12.2. All duration values are in milliseconds.

Table 12.2: Performance test results pitting DOM methods versus property access

Browser	getAttribute()	property get	setAttibute()	property set
Internet Explorer 9	4744	1119	8948	1006
Firefox 7	654	494	707	514
Safari 5	268	142	1055	627
Chrome 15	328	202	1047	714
Opera 11	2146	1960	1760	1416

NOTE These test were conducted on a 2011 MacBook Pro with a 2.2 GHz i7 processor and 8 GB of RAM running OS X Lion. The IE tests were conducted on the same system under Windows 7 running in a Parallels 7 instance.

The results of a sample run of this test are shown in figure 12.1.

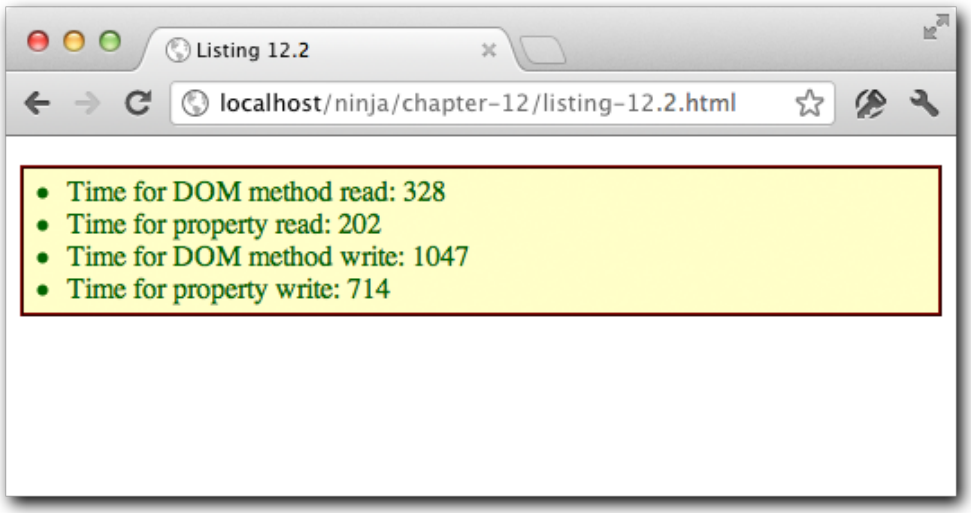

Figure 12.1 Results of running our performance test in the Chrome browser

While these differences in speed may not be crippling for individual operations, they can add up if performed many times; in a tight loop, for example. So we might want to implement a method by which we can access a value by property, if the property exists, and by DOM method as a fallback when it doesn't. Consider the code of listing 12.3.

Listing 12.3: A function for setting and getting attribute values

```
<div id="testSubject"></div>

<script type="text/javascript">

  (function(){                                                    #1

    var translations = {                                          #2
      "for": "htmlFor",
      "class": "className",
      readonly: "readOnly",
      maxlength: "maxLength",
      cellspacing: "cellSpacing",
      rowspan: "rowSpan",
      colspan: "colSpan",
      tabindex: "tabIndex"
    };

    window.attr = function(element,name,value) {                  #3
      var property = translations[name] || name,
          propertyExists = typeof element[ property ] !== "undefined";
```

```
    if (typeof value !== "undefined") {
      if (propertyExists) {
        element[property] = value;
      }
      else {
        element.setAttribute(name,value);
      }
    }

    return propertyExists ?
      element[property] :
      element.getAttribute(name);
  };

})();

var subject = document.getElementById('testSubject');      #4
assert(attr(subject,'id') === 'testSubject',
       "id value fetched");

assert(attr(subject,'id','other') === 'other',
     "new id value set");
assert(attr(subject,'id') === 'other',
     "new id value fetched");

assert(attr(subject,'data-custom','whatever') ==='whatever',
       "custom attribute set");
assert(attr(subject,'data-custom') === 'whatever',
       "custom attribute fetched");


</script>
```

#1 Creates private scope
#2 Creates translation map
#3 Defines set/get function
#4 Tests new function

This example not only establishes a setter/getter function for attribute/property values, it also shows a number of important concepts that we can use elsewhere in our code.

In our function, we need to translate between the known discrepancies between property and attribute names that we outlined in table 12.2, so we create a translation map (#2). But we don't want to pollute the global namespace with this map; we want it to be available to the function in its local scope, but no farther than that.

We accomplish that by enclosing the map definition and function declaration within an immediate function (#1), which creates a local scope. The translation map (#2) is not accessible outside the immediate function, but the set/get function that we also define (#3) within the immediate function, has access to the map via its closure. Nifty, eh?

Another important principle is exhibited by our `attr()` function itself – the function can act as both a setter and a getter simply by inspecting its own argument list. If a `value` argument is passed to the function, the function acts as a setter, setting the passed value as

the value of the attribute. If the `value` argument is omitted and only the first two arguments are passed, it acts as a getter, retrieving the value of the specified attribute.

In either case the value of the attribute is returned, which makes it easy to use the function in either of its modes in a function call chain.

It should be noted that the above implementation doesn't take into account many of the cross-browser issues that plague attribute access. Let's find out exactly what those issues are.

12.2 Cross-browser attribute issues

Cross-browser issues in general can quite harrowing, and the number of cross-browser issues at play in the area of attribute values is not trivial. Let's explore a few of the major and most commonly encountered issues, starting with DOM name expansion.

12.2.1 DOM id/name expansion

The nastiest bug to deal with is a mis-implementation of the DOM code in the browsers.

As we pointed out in the previous chapter, the problem is that all of the "Big Five" browsers take the `id` or `name` values specified on form input elements and adds references to the elements as properties on the parent `<form>` element. These generated properties actively overwrite any existing properties of the same name that might already be on the form element.

Additionally, Internet Explorer doesn't just replace the properties; it also replaces the attribute values with references to the elements.

A demonstration of these problems can be seen in listing 12.4.

Listing 12.4: Demonstrating how browsers strong-hand form elements

```
<form id="testForm" action="/">                          #1
  <input type="text" id="id"/>
  <input type="text" name="action"/>
</form>

<script>
  window.onload = function(){

    var form = document.getElementById('testForm');

    assert(form.id === 'testForm',                        #2
           "the id property is untouched");
    assert(form.action === '/',
           "the action property is untouched");

    assert(form.getAttribute('id') === 'testForm',        #3
           "the id attribute is untouched");
    assert(form.getAttribute('action') === '/',
           "the action attribute is untouched");

  };
</script>
```

#1 Creates the test subject
#2 Tests if properties have been stomped upon
#3 Tests if attributes have been mangled

This series of test shows how this unfortunate feature can cause loss of markup data. First, we define an HTML form (#1) with two input element children. One child has an id of `id`, and the other a name of `action`.

Out first test asserts (#2) that the form element's id and action properties should be as we set them in the HTML markup, and the second set of tests (#3) asserts that the attribute values reflect the markup.

But upon running the test in Chrome, we see the display of figure 12.2.

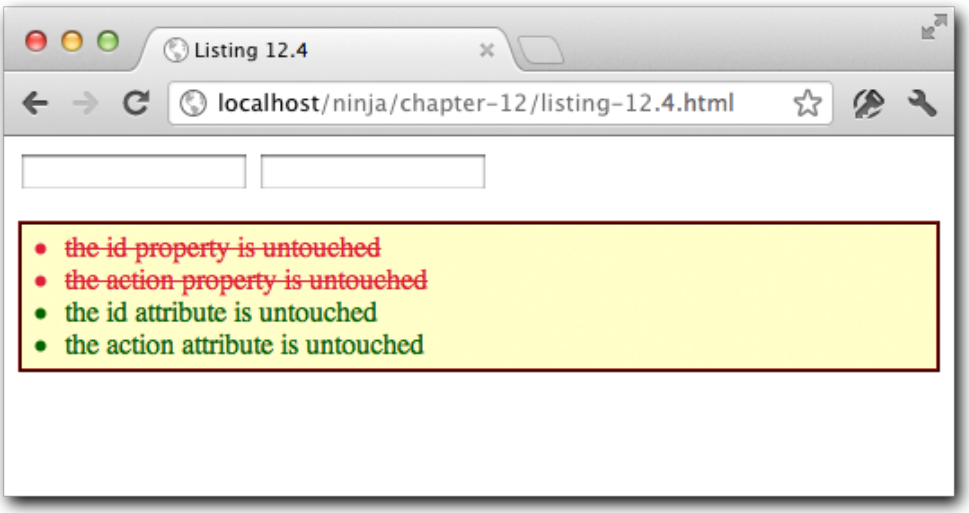

Figure 12.2 Looks as if the markup values have been stomped upon!

In all modern browsers, the `id` and `action` properties have been overwritten with references to the input elements simply because of the `id` and `name` values chosen for those elements. And the original property values are gone forever! In browsers other than IE, we can obtain the original values using the DOM attribute methods, but in IE, even those values are replaced.

But we're ninjas and will not be denied. Despite the best effort of the browsers to keep us from the values, we got a trick up our sleeves. We can gain access to the original DOM node representing the element attribute itself. This node remains untainted from the browser tinkering. To get the value from a DOM attribute node, say the one for the `action` attribute, we'd use:

```
var actionValue = element.getAttibuteNode("action").nodeValue;
```

As an exercise, see if you can use this approach to augment the `attr()` method that we developed in listing 12.3 to detect when a form element node's attribute has been replaced by an element reference, and fallback to getting the value from the DOM node when so.

> **NOTE** If you're interested in the sort of problems that arise from these "element expansions", we recommend checking out Juriy Zaytsev's DOMLint tool at http://kangax.github.com/domlint/, which is capable of analyzing a page for potential problems, and Garrett Smith's write-up of the issue at http://jibbering.com/faq/names/.

While this issue cannot be considered a bug as it's the intended behavior, it's destructive and certainly unnecessary when element references are so easy to obtain with methods such as `document.getElementById()` and other similar methods.

But this is far from the only issue with how the browsers handle attributes. Let's look at another.

12.2.2 URL normalization

There's a "feature" in all modern browsers (that we alluded to in the chapter introduction) that violates the principle of least surprise: when accessing a property that references a URL (such as `href`, `src`, or `action`) the URL value is automatically converted from its original form into a full canonical URL.

Let's write a test to demonstrate this issue in the code of listing 12.5.

Listing 12.5 Demonstrating the URL normalization issue

```
<a href="listing-12.5.html" id="testSubject">Self</a>

<script type="text/javascript">
  var link = document.getElementById('testSubject');

  var linkHref = link.getAttributeNode('href').nodeValue;      #1

  assert(linkHref === 'listing-12.5.html',                     #2
         'link node value is ok');

  assert(link.href === 'listing-12.5.html',                    #3
         'link property value is ok');

  assert(link.getAttribute('href') == linkHref,                #4
         'link attribute not modified');

</script>
```

#1 Obtains original node value
#2 Tests original node value
#3 Tests style property
#4 Tests attribute value

In this test, we establish an anchor tag with an `href` attribute that refers back to the same page. Then we obtain a reference to this element for testing.

The trick we learned in the previous section – diving down into the original nodes of the DOM to find the original value of the markup – is then employed (#1). This value is checked (#2) before we blindly assume that our trick worked.

Then we test the property to see if it matches (#4). In all browsers this test fails as the value has been normalized to a full URL.

Lastly, we test to see if the attribute value has been modified (#4). In all browsers but older versions of IE, the test passes.

Not only does this test show the nature of the issue, it also provides a work-around: we can use the DOM node trick to obtain such attributes when we want to be sure to obtain a non-modified value.

For older versions of IE (prior to IE8), another work-around is a propriety extension to the `getAttribute()` method in Internet Explorer. Passing the magic number 2 as a second parameter, will force the result to be the un-normalized value, as in:

```
var original = link.getAttribute('href',2);
```

We can feel free to use either work-around in modern browsers as the DOM node trick will work across all browsers, and modern browsers other than IE will simply ignore any second parameter passed to `getAttribute()`. Older versions of Opera would crash for no apparent reason when a second parameter is passed to `getAttribute()`, so avoid that approach if such versions of Opera are in your support matrix.

The chances that the URL normalization issue will be a problem for your code is likely small, unless your code absolutely needs to get at the un-normalize values.

Now let's examine an issue that may have more far-reaching consequences.

12.2.3 The style attribute

An important element attribute whose values are particularly challenging to set and get is the `style` attribute. HTML DOM elements are given a `style` property, which we can access to gain information about the style information of the element; for example: `element.style.color`. However, if we want to get the original `style` string that was specified on the element, it becomes more challenging. For example, consider markup of:

```
<div style='color:red;'></div>
```

What if we wanted to obtain the original "color:red" string?

The `style` property is of no help at all, as it is set to an object that contains the parsed results of the original string. And while `getAttribute("style")` works in most browsers, it doesn't work in Internet Explorer. Instead, IE stores a property on the `style` object that we can use to obtain the original style string named `cssText`; for example: `element.style.cssText`.

While directly getting the original value of `style` attribute may be a comparatively uncommon operation (as opposed accessing the resulting `style` object), there's another browser problem that's likely to affect any page that creates DOM elements at run time.

12.2.4 The type attribute

Another Internet Explorer gotcha, for IE8 and earlier versions, affects the `type` attribute of `<input>` elements, and for which there isn't any reasonable workaround. Once an `<input>` element has been inserted into a document, its `type` attribute can no longer be changed. In fact, IE throws an exception if you attempt to change it.

For example, consider the following code:

Listing 12.6: Changing an input element's type after insertion

```
<form id="testForm" action="/"></form>

<script>
  window.onload = function(){

    var input = document.createElement('input');          #1

    input.type = 'text';                                  #2
    assert(input.type == 'text',
           'Input type is text');

    document.getElementById('testForm')                   #3
            .appendChild(input);                          #3

    input.type = 'hidden';                                #4
    assert(input.type == 'hidden',
           'Input type changed to hidden');

  };
</script>
```

#1 Create new element
#2 Sets type
#3 Inserts into DOM
#4 Changes type after insertion

In this test, we create a new `<input>` element (#1), give it a type of "text" (and asserting that the assignment was successful) (#2), and insert the new element into the DOM, (#3). *After* insertion, we change the type to "hidden" and assert that the change took place (#4).

Executed in all modern browsers but IE, the tests pass without problem. In IE8 and earlier however, an exception is thrown at the assignment attempt, and the second test never executes.

While there is no easy work-around there are two stopgap measures we can take:

1. Rather than try to change the `type`, create a new <input> element, copy over all properties, and attributes, and replace the original element with the newly created element. But while this solution seems easy enough it has problems. Firstly, it's impossible to know if the element has had any event handlers established upon it using the DOM Level 2 methods unless we've been tracking them ourselves. And secondly, any references to the original element become invalid.

2. In any API you create to effect changes to properties or attributes, simply reject any attempts to change the `type` value.

Neither of these is completely satisfying.

jQuery employs the second approach, throwing an informative exception if any attempt is made to change the `type` attribute if the element has already been inserted into the document. Obviously this is a compromise "solution", but at least the user experience is consistent across all platforms.

Thankfully, this issue has been addressed in IE9.

Let's look at yet another annoyance that the browsers bedevil us with – again within the realm of form elements.

12.2.5 The tab index problem

Determining the tab index of an element is another weird problem that the browsers throw at us, and it's one where there's little consensus at to how it *should* work. While it's perfectly possible to get the tab index of an element using either the `tabIndex` property or the `"tabindex"` attribute for elements that have them explicitly defined, the browser returns a value of 0 for the `tabIndex` property, and null for the "tabindex" attribute, for elements without an explicit value.

Which means, of course, that we have no way of knowing what tab index has been assigned to the elements that we didn't explicitly set a tab index value upon.

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, which can be found at: http://fluidproject.org/blog/2008/01/09/getting-setting-and-removing-tabindex-values-with-javascript/

The last attribute-related problem we'll consider isn't really an attribute issue at all.

12.2.6 Node names

While this issue isn't directly related to attributes per se, a number of workarounds we've used in this section have relied upon finding nodes, and as it turns out determining the name of a node can be slightly tricky.

Specifically, the case sensitivity of the node name changes depending upon which type of document you are examining. 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, which means that it could be lowercase, uppercase, or any combination of either.

The conventional solution to this hindrance is to normalize the name prior to any comparison, usually to lowercase. For example, let's say we want to perform some operation on only `<div>` and `<ul>` elements. As we don't know whether the node names that we'll be getting are "div" or "DIV" or even "dIv", we'd want to take care to normalize the names as shown in this code:

```
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";
  }
}
```

When we definitively know what type of document our code will be executing within, we don't necessarily have to worry about this case sensitivity, but if we're writing reusable code that should run in any environment, it's best to be prudent and perform the normalization.

In this section, we've talked about issues with the attributes and properties of elements, and even examined a small issue with the style property. But that was just a tiny glimpse into the heartburn that the browsers have in store for us when it comes to styling. In the next section, we'll take a look at the pain points of dealing with CSS issues in the browsers.

12.3 Styling attribute headaches

As with general attributes, getting and setting styling attributes can be quite the headache. Just like the attributes and properties that we examined in the previous section, we again have two approaches for handling `style` values: the attribute value, and the element property created from it.

The most commonly used of these is the `style` element property, which is not a string but an object which holds properties that correspond to the style values specified in the element markup. In addition to this, we'll see that there is an API for accessing the computed style information of an element.

This section will outline the "things you should know" about dealing with styles in the browsers. Let's start with a look at where style information is recorded.

12.3.1 Where're my styles?

The style information located on the `style` property of a DOM element is initially set from the value specified for the `style` attribute in the element markup. For example `style="color:red;` will result in that style information being placed into the style object. During page execution, script can set or modify values in the style object, and these changes will actively affect the display of the element.

Many script authors are disappointed to find that no values from on-page `<style>` elements or external stylesheets are available in the element's `style` object. But we won't stay disappointed for long – we'll shortly see a way to obtain such information.

But for now, let's see how the `style` property gets its values. Examine the code of listing 12.7.

Listing 12.7: Examining the style property

```
<style> div { font-size: 1.8em; } </style>          #1


<div style="color:#000;" title="Ninja power!">      #2
  忍者パワー
</div>

<script>
  window.onload = function(){

    var div = document.getElementsByTagName("div")[0];

    assert(div.style.color == 'rgb(0, 0, 0)' ||        #3
           div.style.color == '#000',
           'Color was recorded');

    assert(div.style.fontSize == '1.8em',              #4
           'Font size was recorded');

    div.style.color = "#336699";                       #5

    assert(div.style.color == 'rgb(51, 102, 153)' ||   #6
           div.style.color == '#336699',
           'Color was replaced');

  };
</script>
```

#1 Declares on-page stylesheet
#2 Creates styled element
#3 Tests in-lined style
#4 Tests inherited style
#5 Replaces style
#6 Tests replaced style

In this example, we set up a `<style>` element to establish an internal stylesheet (#1) whose values will be applied to the elements on the page. The stylesheet specifies that all `<div>` elements will appear in a font size 1.8 times bigger than the default.

Then we create a `<div>` element with an in-lined style attribute that colors the text of the element black (#2).

We then begin the testing. After obtaining a reference to the `<div>` element, we test that the `style` attribute received a `color` property that represent the color assigned to the element (#3). Note that even though the `color` was specified as `#000` in the inline style, it is normalized to RGB notation when set into the `style` property in most browsers (so we check both formats).

> **WARNING** The normalization is not always consistent across browsers or even within a specific browser. Most colors will be normalized to RGB notation, but some browsers will leave colors specified as named colors (`black`, for example) as that named color.

Looking ahead to figure 12.3, we see that this test passes.

Then we naively test that the font size styling, specified in the inline stylesheet, has been set into the style object (#4). But even though we can see in figure 12.3 that the font size style has certainly been applied to the element, the test fails. This is because the style object does not reflect any style information inherited from CSS stylesheets.

Moving on, we use an assignment to change the value of the `color` property in the style object to a pleasing shade of aqua blue (#5), and test that the change was applied (#6). We can see in figure 12.3 that the test passes, and that the color change has been applied to the element (assuming you're looking at this in color – if not, display the page in a browser to see the text in all its Technicolor glory!)

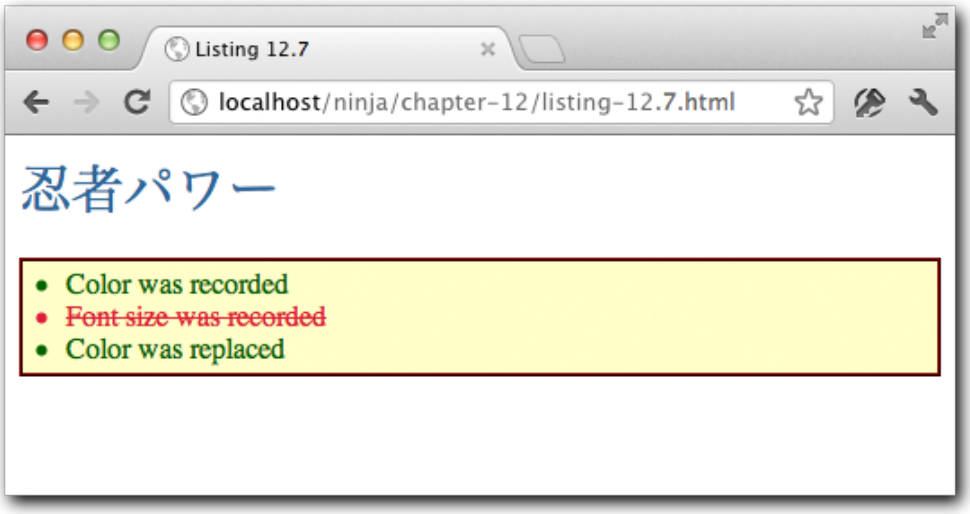

Figure 12.3 Tests show that inline and assigned styles are recorded, but inherited styles are not!

It should be noted that any values in an element's `style` property will take precedence over anything inherited by a stylesheet (even if the stylesheet rule uses the `!important` annotation).

One thing that you may have noted in listing 12.7 is that CSS specifies the font size property as `font-size`, but in script we referenced it as `fontSize`. Why is that?

12.3.2 Style property naming

With CSS attributes there are relatively few cross-browser difficulties that are thrown at us when it comes to accessing the values provided by the browser. But differences between how CSS names styles and how we access those in script do exist, and there are some style names that differ across browsers.

CSS attributes that span more than one word separate the words with a hyphen character; examples are `font-weight`, `font-size`, and `background-color`. You may recall that property names in JavaScript *can* contain a hyphen character; but doing so prevents the property from being access via the dot operator. Hypothetically consider:

```
var color = element.style['font-size'];
```

This would be perfectly valid. But the following would not:

```
var color = element.style.font-size;
```

The JavaScript parser would see the hyphen as a subtraction operator and nobody would be happy with the outcome. So rather than forcing page developers to always use the general form for property access, multi-word CSS style name are converted to camel case when used as a property name.

So `font-size` becomes `fontSize`, and `background-color` becomes `backgroundColor`.

We can either remember to do this, or we could write a simple API to set or get styles for us, that automatically handles the camel casing on our behalves. It's shown in listing 12.8.

Listing 12.8: A simple method for accessing styles

```
<div style="color:red;font-size:10px;background-color:#eee;"></div>

<script>
  function style(element,name,value){                          #1
    name = name.replace(/-([a-z])/ig,                          #2
                        function(all,letter){
                          return letter.toUpperCase();
                        });

    if (typeof value !== 'undefined') {                        #3
      element.style[name] = value;
    }

    return element.style[name];                                #4
  }

  window.onload = function(){

    var div = document.getElementsByTagName('div')[0];

    assert(true,style(div,'color'));
    assert(true,style(div,'font-size'));
    assert(true,style(div,'background-color'));

  };
</script>
```

#1 Define style function
#2 Converts name to camel case
#3 Sets value if provided
#4 Returns value

With the exception of the conversion of the `name` parameter to camel case, this function operates in a similar fashion to the `attr()` function that we developed in listing 12.3, so we won't belabor its operation.

If the regex-driven conversion operation has you scratching your head, you might want to review the material of chapter 7.

Also note that despite the inclusion of a number of `assert()` calls, we haven't really performed any testing of the function – we used the assert as a lazy way of displaying the output in the page, as shown in figure 12.4.

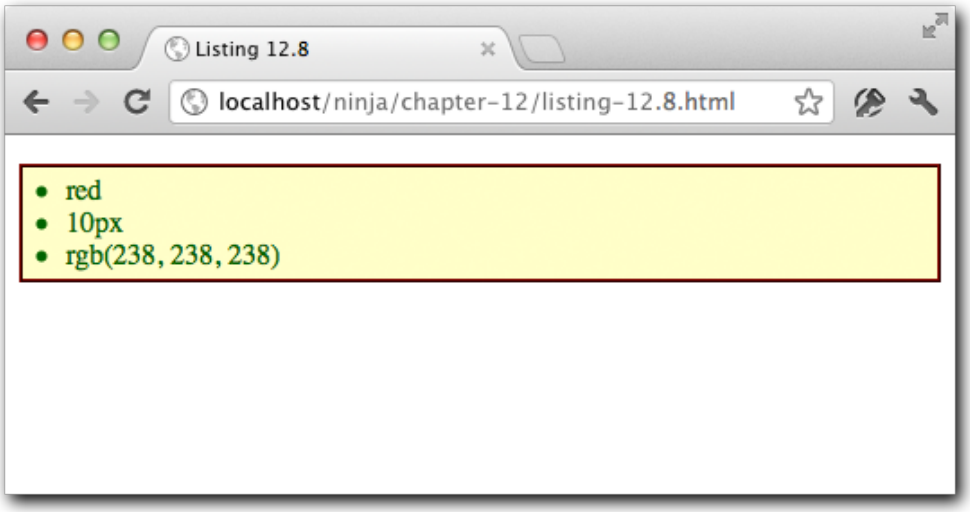

Figure 12.4 Putting out style() function to the test shows how it can automatically divine the property name gives a CSS name

As an exercise, write a series of asserts that thoroughly test our new function.

Earlier we mentioned that there were a number of "problem" style properties that are treated differently across browsers. Let's take a gander at one of them.

12.3.3 The float style property

One major naming headache that exists in the area of style attributes is the manner in which the `float` attribute is handled. By necessity, this property needs to handle specially as the name `float` is a reserved keyword in JavaScript. So the browsers need to provide an alternative name.

And as has frequently happened in such cases, the standards-compliant browsers went one way, and Internet Explorer went another. Nearly all browsers choose to use the name `cssFloat` as the alternative name, whereas Internet Explorer chose `styleFloat`.

After finishing your long sigh, and using the translation capability that we built into in listing 12.3 as inspiration, see if you can modify the `style()` function of listing 12.8 to accommodate this difference.

Earlier in this section, we saw how color values can be changed from one notation to another when added as a style property. Let's explore another such situation.

12.3.4 Conversion of pixel values

Another important point to consider when setting style values is when assigning numeric values that represent pixels. When specifying pixel values in deprecated attributes such as the `height` attribute if the `<img>` tag, we're used to just specifying a number and letting the browser deal with the units.

When assigning pixel values to style properties, that can get us into a lot of trouble. When setting a numeric value into a style property we must specify the unit in order for it to reliably work across all browsers.

For example, let's say that we want to set the `height` style value of an element to 10 pixels. Either of the following is safe way to do this across the browsers:

```
element.style.height = "10px";
element.style.height = 10 + "px";
```

However, the following *not* safe across browsers:

```
element.style.height = 10;
```

You might think it'd be easy to just add a little logic to our `style()` function of listing 12.8 to just tack a 'px' to the end of numeric value coning into the function. But not so fast! Not all numeric values represent pixels! There are a number of style properties that take numeric values that do not represent a pixel dimension. For example:

- z-index
- font-weight
- opacity
- zoom
- line-height

Knowing that, now extend the function of listing 12.8 to automatically handle pixel values.

One other point to keep in mind, when attempting to read a pixel value out of a style attribute, the `parseFloat` method should be used to make sure that you get the intended value under all circumstances.

Now let's take a look at a set of important style properties that can be tough to handle.

12.3.5 Measuring heights and widths

Style properties such as `height` and `width` pose a special problem as their values default to `auto` when not specified, and the element sizes itself according to its contents. So we can't

use the `height` and `width` style properties to get any accurate values unless explicit values were provided in the attribute string.

Thankfully, the `offsetHeight` and `offsetWidth` properties exist to provide just that: a fairly reliable means to access the actual height and width of an element. Be aware that the values assigned to these two properties include the padding of the element. This information is usually exactly what we want if we're attempting to position one element over another one. But at other times we may want to obtain information about the element's dimensions with and without borders and padding.

Something to watch out for, however, is that in highly interactive sites it's likely that elements may spend some of their time in a hidden state (as in its `display` style being set to `none`), and when an element is hidden it should be no surprise that it has no dimensions. So any attempt to fetch the `offsetWidth` or `offsetHeight` properties will result in a value of zero.

For such hidden elements, should we wish to obtain its non-hidden dimensions, we can employ a trick and momentarily unhide the element, grab the values, and hide it again. But of course, we want to do so in such a way that there's no visible clue that this is going on behind the scenes. So how can we make a hidden element not hidden, without making it visible?

Employing our ninja skills, we can do it! Here's how:

1. Change the `display` property to `block`.
2. Set `visibility` to `hidden`.
3. Set `position` to `absolute`.
4. Grab the dimension values.
5. Restore the changed properties.

Changing the `display` property to `block` allows us to grab the actual values of `offsetHeight` and `offsetWidth`, but will make the hidden element visible. So to make it invisible, we'll set the `visibility` property to `invisible`. But (isn't there always another "but") that will leave a big hole where the element is positioned, so we also set the `position` property to `absolute` to take the element out of the normal display flow.

All that actually sounds more complicated that the actual implementation of it. Let's take a look at that in listing 12.9.

Listing 12.9: Grabbing the dimensions of hidden elements

```
<div>
  Lorem ipsum dolor sit amet, consectetur adipiscing elit.
  Suspendisse congue facilisis dignissim. Fusce sodales,
  odio commodo accumsan commodo, lacus odio aliquet purus,
  <img src="../images/ninja-with-pole.png" id="withPole">
  <img src="../images/ninja-with-shuriken.png"
       id="withShuriken" style="display:none">
  vel rhoncus elit sem quis libero. Cum sociis natoque
```

```
  penatibus et magnis dis parturient montes, nascetur
  ridiculus mus. In hac habitasse platea dictumst. Donec
  adipiscing urna ut nibh vestibulum vitae mattis leo
  rutrum. Etiam a lectus ut nunc mattis laoreet at
  placerat nulla. Aenean tincidunt lorem eu dolor commodo
  ornare.
</div>

<script type="text/javascript">

  (function(){                                          #1

    var PROPERTIES = {                                  #2
      position: "absolute",
      visibility: "hidden",
      display: "block"
    };

    window.getDimensions = function(element) {          #3

      var previous = {};                                #4
      for (var key in PROPERTIES) {
        previous[key] = element.style[key];
        element.style[key] = PROPERTIES[key];           #5
      }

      var result = {                                    #6
        width: element.offsetWidth,
        height: element.offsetHeight
      };

      for (key in PROPERTIES) {                         #7
        element.style[key] = previous[key];
      }
      return result;
    };

  })();

  window.onload = function() {

    setTimeout(function(){

      var withPole = document.getElementById('withPole')
          withShuriken = document.getElementById('withShuriken');

      assert(withPole.offsetWidth == 41,                #8
             "Pole image width fetched; actual: " +
             withPole.offsetWidth + ", expected: 41");
      assert(withPole.offsetHeight == 48,
             "Pole image height fetched: actual: " +
             withPole.offsetHeight + ", expected 48");

      assert(withShuriken.offsetWidth == 36,            #9
             "Shuriken image width fetched; actual: " +
             withShuriken.offsetWidth + ", expected: 36");
```

```
      assert(withShuriken.offsetHeight == 48,
             "Shuriken image height fetched: actual: " +
             withShuriken.offsetHeight + ", expected 48");

      var dimensions = getDimensions(withShuriken);        #10

      assert(dimensions.width == 36,                       #11
             "Shuriken image width fetched; actual: " +
             dimensions.width + ", expected: 36");
      assert(dimensions.height == 48,
             "Shuriken image height fetched: actual: " +
             dimensions.height + ", expected 48");

    },3000);

  }
</script>
```

#1 Creates private scope
#2 Defines target properties
#3 Creates new function
#4 Remembers settings
#5 Replaces settings
#6 Fetches dimensions
#7 Restores settings
#8 Tests visible element
#9 Tests hidden element
#10 Uses new function
#11 Retests hidden element

Well, that's rather a long listing, but most of it is test code; the actual implementation of the new dimension-fetching function spans only about a dozen or so lines of actual code.

Let's take a look at it piece by piece. First, we set up some elements to test: a `<div>` element containing a bunch of text with two images embedded within it; left-justified by styles in an external stylesheet. These image elements will be the subjects of our tests; one is visible, and one is hidden.

Prior to running any script, the elements appear as shown in figure 12.5.

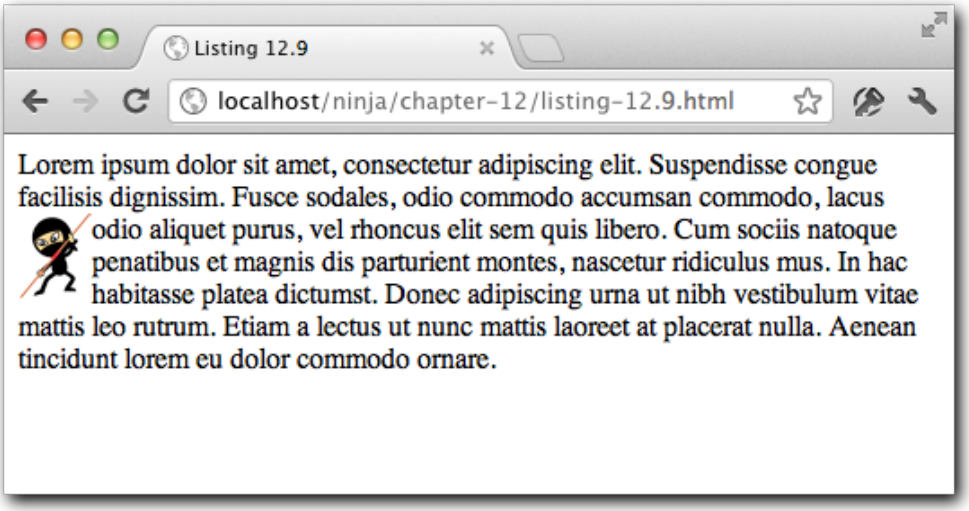

Figure 12.5 We'll use two images: one visible, one hidden, for testing fetching the dimensions of hidden elements

If the second image were not hidden, it would appear as a second ninja just to the right of the visible one.

Then we set about to define our new function. We're going to use a hash for some important information, so we repeat the trick of listing 12.3 and enclose the local variable and the function definition in an immediate function (#1) to create a local scope and closure.

The local hash to contain the properties we want to muck around with is defined (#2) and populated with the three properties and their replacement values.

Our new dimension-fetching function is then declared (#3), accepting the element that is to be measured.

Within that function we first create a hash named `previous` (#4) in which to record the previous values of the style properties that we'll be stomping on, so that we can restore them later. Looping over the replacement properties, we then record each of their previous values, and then replace those values with the new ones (#5).

That accomplished, we're ready to measure the element, which has now been made non-hidden, invisible, and absolutely positioned. The dimensions are recorded in a hash assigned to local variable `result` (#6).

Now that we have pilfered what we came for, we want to erase our tracks and restore the original values of the style properties that we had modified (#7), and return the results as a hash containing `width` and `height` properties.

All well and good; but does it work? Let's find out.

In a load handler, we perform the tests in a callback to a 30-second timer. Why, you ask? The load handler is so that we don't perform the test until we know that the DOM has been built, and the timer is so that we can watch the display while the test is running to make sure that there are no display glitches while we fiddle with the properties of the hidden element. After all, if the display is disturbed in any way when we run our function, it's a bust.

In the timer callback, we first get reference to our test subjects (the two images), and assert that we can obtain the dimensions of the visible image using the offset properties (#8). This test passes, which we can see if we peek ahead to figure 12.6.

Then we make the same test on the hidden element (#9), incorrectly assuming that the offset properties will work with a hidden image. Not surprisingly, as we've already acknowledged that this won't work, the test fails.

So, we call our new function on the hidden image (#10), and then retest with those results (#11). Success! Our test passes, as shown in figure 12.6.

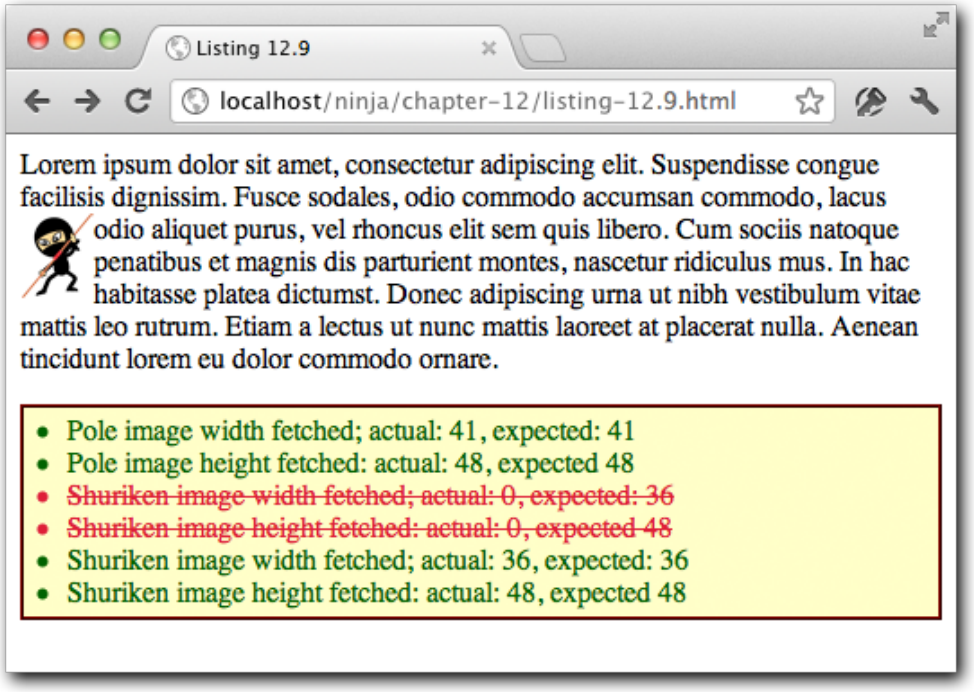

Figure 12.6 By temporarily adjusting the style properties of hidden elements, we can successfully fetch their dimensions

If we watch the display of the page while the test is running – remember we delay running the test until 3 seconds after the DOM is loaded – we can see that the display is not perturbed in any way by our behind-the-scenes adjustments of the hidden element's properties.

NOTE

Checking the `offsetWidth` and `offsetHeight` style properties for zeroes can serve as an incredibly efficient means of determine the visibility of an element.

The dimension style properties aren't the only ones that pose a challenge to us. Let's explore the nuances of dealing with the `opacity` property.

12.3.6 Seeing through opacity

The `opacity` style property is another special case that needs to be handled differently across browsers. All modern browsers, including Internet Explorer 9, natively support the `opacity` property, but versions of IE prior to IE9 use their proprietary alpha filter notation.

Because of this, we frequently see opacity styles specified as follows in a stylesheet (or directly in the `style` attribute):

```
opacity: 0.5;
filter: alpha(opacity=50);
```

The standard style uses a value from 0.0 to 1.0 to specify the opacity of an element, while the alpha filter uses an integer percentage from 0 to 100. So, the above rules both specify an opacity value of 50%.

Let's say that we have an element defined with both styles as follows:

```
<div style="opacity:0.5;filter:alpha(opacity=50);">Hello</div>
```

When trying to fetch these values, the problem we are faced with is two-fold:

- There are many different types of filters beyond `alpha`, such as transformations, so we have to deal with many filter types and can't just assume that a filter always specifies opacity.
- Even though IE 8 and lesser don't support `opacity`, the value specified for `opacity` will be returned when referencing the element's `style.opacity` property, even if it's completely ignored by the browser.

This latter point makes it hard for our code to determine if the browser has native support for `opacity` or not. But once again, we can focus our ninja powers on the problem and thumb our noses at the browsers that stubbornly try to foil us.

As it turns out, browsers that support `opacity` will always normalize an opacity value less than 1.0 with a leading zero. For example, if the opacity is specified as `opacity: .5`, browser with native opacity support will return the value as 0.5, whereas non-supporting browsers will simply leave the value in its original form of .5.

So, we can use feature simulation (remember that from chapter 11?) to determine if a browser supports opacity natively or not. Consider the code of listing 12.10:

Listing 12.10: Determining if a browser supports opacity or not

```
<img src="../images/ninja-with-nunchuks.png" style="opacity:.5;">

<script type="text/javascript">

  var div = document.createElement("div");              #1
  div. setAttribute('style','opacity:.5');
  var OPACITY_SUPPORTED = div.style.opacity === "0.5";

  assert(OPACITY_SUPPORTED,                             #2
         "Opacity is supported.");

</script>
```

#1 Checks for support
#2 Displays result

In this test, we create an image element with an `opacity` specified as `.5`. We don't use this element in the code, it's just there to provide a visual indication of whether the opacity value is honored or not.

The meat of the test follows, where we create an unattached element (#1), which we augment with a style attribute with an `opacity` value of `.5`. We then record whether opacity is natively supported by reading the value back and checking if it is fetched as the original value (not supported), or the modified value of `0.5` (supported).

Finally, we assert the support variable, which causes the test to pass on supporting browsers, and fail on non-supporting browsers. Figure 12.7 shows the result of loading this test into Chrome 17 (top) and Internet Explorer 7 (bottom).

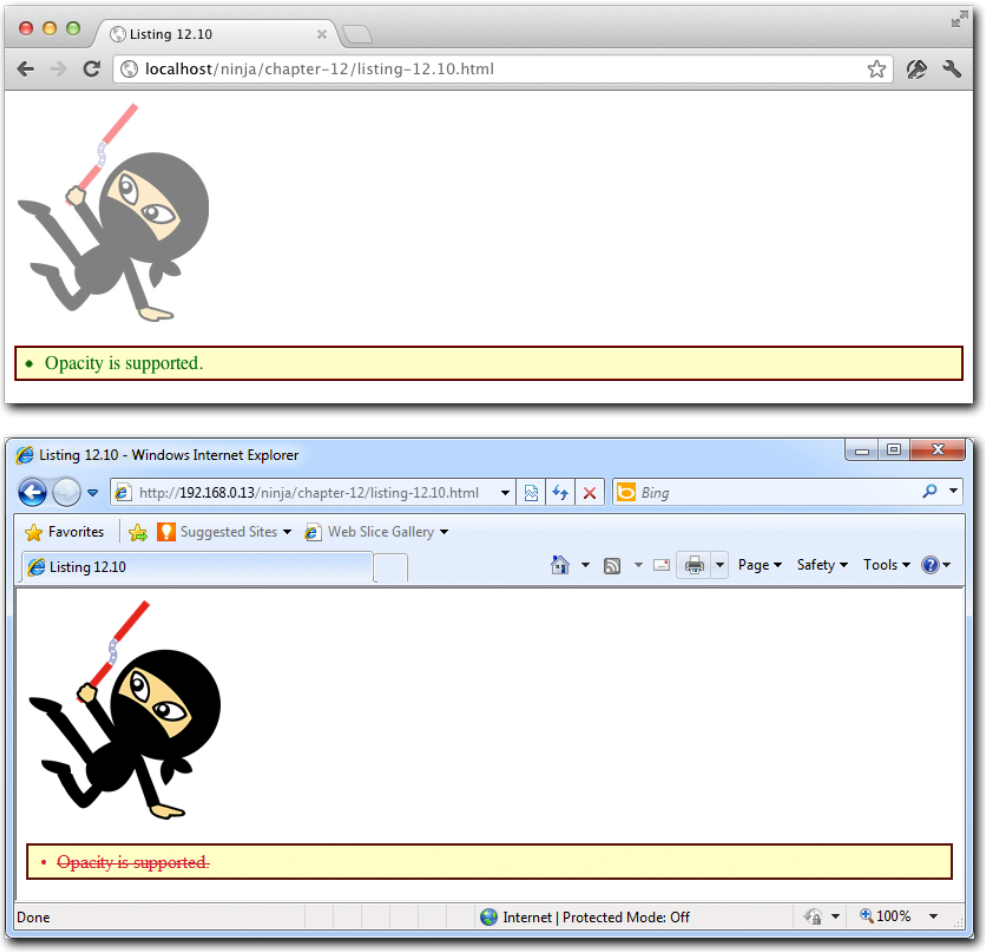

Figure 12.7: Visual clues as well as our explicit test shows that opacity is supported on Chrome but not versions of IE prior to IE9

Using this ninja knowledge, see if you can create a `getOpacity(element)` function, along the lines of the `getDimensions()` function of listing 12.9, that returns the opacity value for the passed element as a value between 0.0 and 1.0 regardless of platform.

Hint: a regular expression would be handy to find the value of the alpha opacity filter, and the `window.parseFloat()` method will be your best friend. Also, return 1.0 as a failure fallback as it's the default for opacity values.

Lets now turn our eyes to yet another set of problem style properties that causes us some pain is due to the fact that their values can take on many equivalent forms.

12.3.7 Riding the color wheel

We've already seen earlier in this chapter that color values can be expressed in a variety of formats. This makes handling color values from the `style` property somewhat tricky. We're somewhat at the mercy of whatever formats that the page author chose, and even more so on any transformations that the browsers apply to those formats.

When we're accessing them via the different computed style methods there is little consistency across the formats that the various browsers will return. Because of this, any attempts to gain access to the useful parts of a color – its red, blue, and green color channels, and as we will see, an optional alpha channel – is a great deal of legwork.

There are numerous formats in which colors can be represented in modern browsers, and they are summarized in table 12.3.

Table 12.3 CSS3 color formats

Format	Description
`keyword`	Any of the recognized HTML color keywords (`red`, `green`, `maroon`, and so on), extended SVG color keywords (`bisque`, `chocolate`, `darkred`, and so on) or the keyword `transparent` (which is equivalent to `rgba(0,0,0,0)`, see below).
`#rgb`	Short hexadecimal RGB (red, green. blue) color values, where each portion is a value from `0` to `f`.
`#rrggbb`	Long hexadecimal RGB (red, green. blue) color values, where each portion is a value from `00` to `ff`.
`rgb(r,g,b)`	RGB notation where each value is a decimal value from 0 to 255, or 0% to 100%.
`rgba(r,g,b,a)`	RGB notation with the addition of an alpha channel. The alpha value ranges from 0.0 (transparent) to 1.0 (fully opaque).
`hsl(h,s,l)`	HSL notation where each value represents Hue, Lightness and Saturation. The hue value ranges from 0 to 360 (angle on the color wheel), saturation and lightness range from 0% to 100%.
`hsla(h,s,l)`	HSL notation with the addition of the alpha channel.

As can be seen from the information in this table, the page author has a lot of flexibility with regards to expressing color information, which wouldn't be much of an issue for us if the browsers would transform the expressed color values placed into `style` into a consistent format.

They don't. So we have a problem.

Let's write a test to see what the browsers do to torment us. Examine the code of listing 12.11.

Listing 12.11: Determine how a browser formats color information

```
<div style="background-color:darkslateblue"> </div> <!--#1-->
<div style="background-color:#369"> </div>
<div style="background-color:#123456"> </div>
<div style="background-color:rgb(44,88,168)"> </div>
<div style="background-color:rgba(44,88,166,0.5)"> </div>
<div style="background-color:hsl(120,100%,25%)"> </div>
<div style="background-color:hsla(120,100%,25%,0.5)"> </div>

<script type="text/javascript">

  var divs = document.getElementsByTagName('div'); #2

  for (var n = 0; n < divs.length; n++) {          #3
    assert(true,divs[n].style.backgroundColor);
  }

</script>
```

#1 Creates colored elements
#2 Collects elements
#3 Displays color info

We start off by creating a series of `<div>` elements with background color style properties expressed in seven different formats (#1). We then collect references to those elements (#2) and iterate over the collection, displaying the value stored in the `style.backgroundColor` property (#3).

This will show us how the browser within which the test is executed formats the color info with respect to the manner in which the page developer specified it. Looking at the displays of figures 12.8 through 12.11, we can see that the stored formats are all across the board.

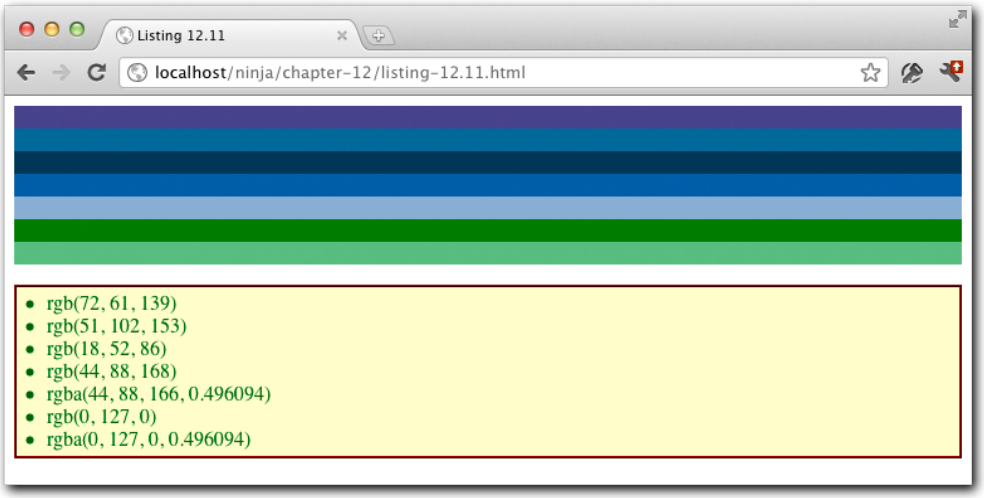

Figure 12.8 Webkit browsers (Chrome, Safari, OmniWeb) normalize colors to rgb and rgba formats

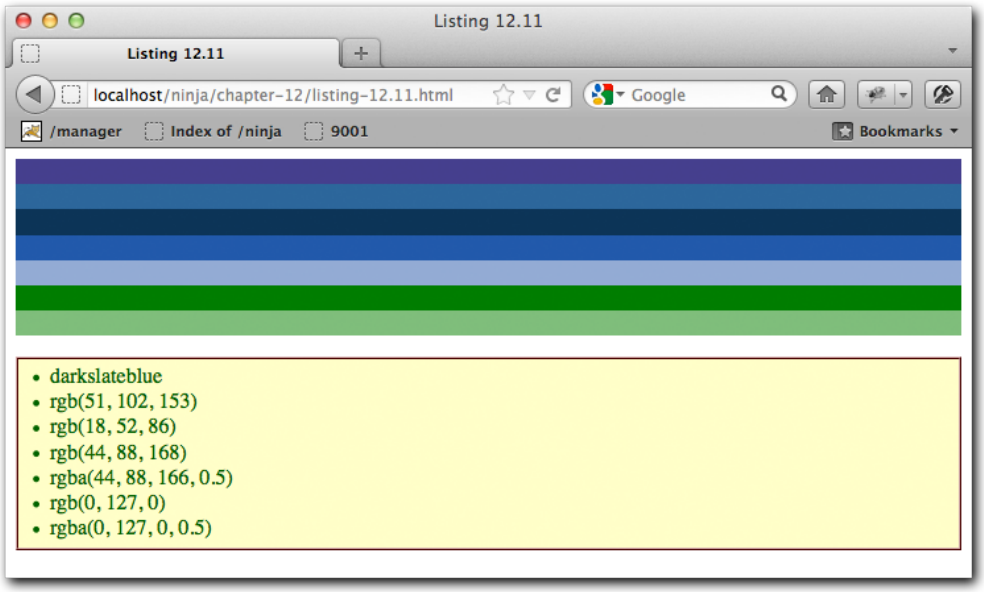

Figure 12.9 Firefox leaves color names be, but normalizes everything else to rgb and rgba formats

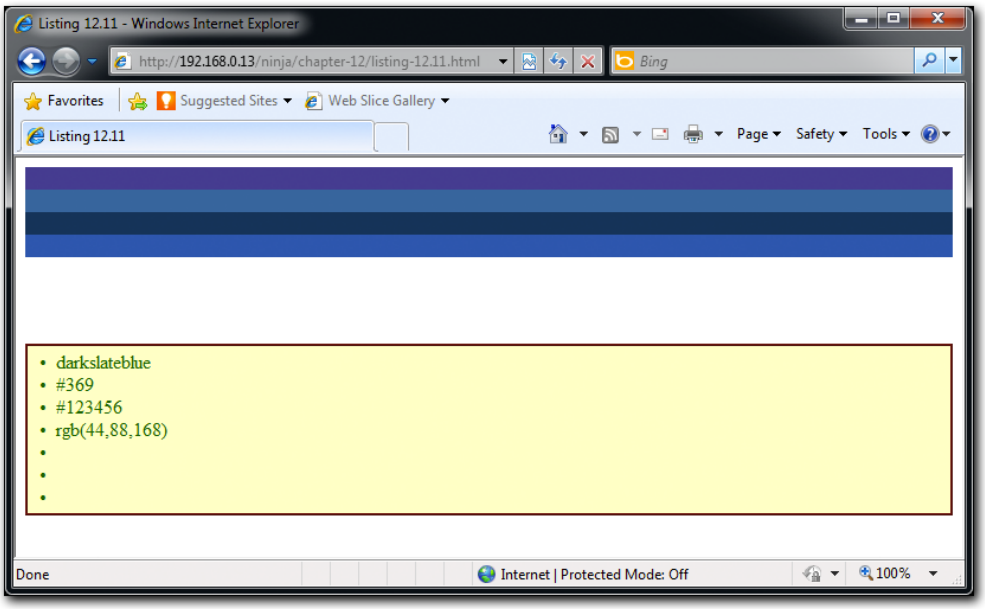

Figure 12.10 Internet Explorer 8 and earlier leave the formats as found, but don't support rgba or the HSL formats at all

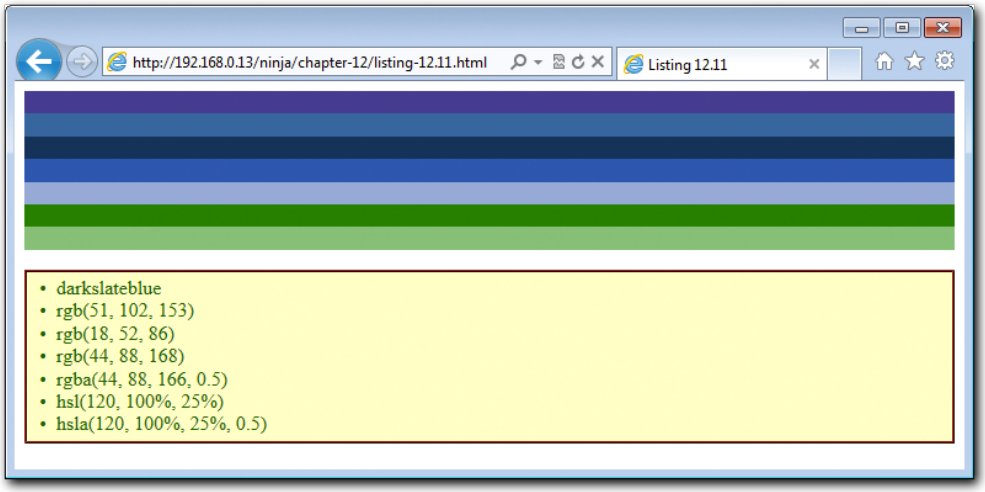

Figure 12.11 Internet Explorer 9 pretty much leaves all formats as specified by the developer

Because there are all these permutations of color information formats and differences across browsers, we're not going to take the amount of space required to develop a `getColor(element,property)` method, but leave it to you to do so. You have all the tools you need at your disposal, so it's more of a lengthy task than a difficult one.

The method should accept an element and a color property (e.g. "color", or "background-color") and return one of: a color keyword, a hash containing `red`, `green`, `blue` and `alpha` properties, or a hash containing `hue`, `lightness`, `saturation` and `alpha` properties. Given your knowledge of regular expressions from chapter 7, and the examples of the `getDimensions()` and `getOpacity()` methods that we developed earlier in this chapter, you should be well armed to tackle the task.

CHALLENGE

If you really want to accept a challenge, also convert any HSL values found to RGB using the formula found at http://en.wikipedia.org/wiki/HSL_and_HSV#Converting_to_RGB.

Obviously, handling colors is not a problem that hasn't already been tackled before. You may want to check out the jQuery Color plugin with code written by Blair Mitchelmore (for the highlightFade plugin) and Stefan Petre (for the Interface plugin). Relevant locations to visit are:

- jQuery Color Plugin: http://plugins.jquery.com/project/color
- highlightFade Plugin: http://jquery.offput.ca/highlightFade/
- Interface Plugin: http://interface.eyecon.ro/

So far, we've covered most of the issues that we need to worry about when it comes to handling the `style` property of an element. But as we pointed out, that property will not include any style information that an element inherits from stylesheets that are in scope for the element. There are many times that it'd be handy to know the full *computed style* that's been applied to an element, so let's see if there's a way that we can obtain that.

12.4 Fetching computed styles

At any point in time, the *computed style* of an element is a combination of all the styles applied to it via stylesheets, the element's `style` attribute, and any manipulations of the `style` property by script.

The standard API specified by the W3C, implemented by all modern browsers (including Internet Explorer 9 but not earlier versions), is the `window.getComputedStyle()` method.

This method accepts an element whose styles are to be computed, and returns an interface through which property queries can be made. The returned interface provides a

method named `getPropertyValue()` for retrieving the computed style of a specific style property.

Unlike the properties of an element's `style` object, the `getPropertyValue()` method accepts CSS property names (e.g. "font-size", and "background-color") rather than the camel-cased versions of those names.

Versions of Internet Explorer prior to version 9 have a proprietary technique for accessing the computed style of an element: a property named `currentStyle` is attached to all elements, and it behaves much like the `style` property except that the information provided is the live computed style information.

That gives us enough information to write a `fetchComputedStyle()` method that will get the computed value of any style property for an element.

SOMETHING TO THINK ABOUT

Why didn't we name the function `getComputedProperty()`?

Listing 12.12 has an implementation of our computed styles function that uses the standard means when available, and falls back to the proprietary method if not.

Listing 12.12 Fetfching computed style values

```
<style type="text/css">                                              #1
  div {
    background-color: #ffc; display: inline; font-size: 1.8em;
    border: 1px solid crimson; color: green;
  }
</style>

<div style="color:crimson;" id="testSubject" title="Ninja power!"> #2
  忍者パワー
</div>

<script type="text/javascript">

  function fetchComputedStyle(element,property) {                    #3

    if (window.getComputedStyle) {

      var computedStyles = window.getComputedStyle(element);         #4

      if (computedStyles) {                                          #5
        property = property.replace(/([A-Z])/g,'-$1').toLowerCase();
        return computedStyles.getPropertyValue(property);
      }
    }

    else if (element.currentStyle) {                                 #6
      property = property.replace(
        /-([a-z])/ig,
        function(all,letter){ return letter.toUpperCase(); });
```

```
      return element.currentStyle[ property ];
    }

  }

  window.onload = function(){

    var div = document.getElementsByTagName("div")[0];

    assert(true,                                                    #7
           "background-color: " +
           fetchComputedStyle(div,'background-color'));
    assert(true,
           "display: " +
           fetchComputedStyle(div,'display'));
    assert(true,
           "font-size: " +
           fetchComputedStyle(div,'fontSize'));
    assert(true,
           "color: " +
           fetchComputedStyle(div,'color'));

  };

</script>
```

#1 Defines stylesheet
#2 Creates test subject
#3 Defines new function
#4 Gets interface
#5 Fetched style value
#6 Uses proprietary means
#7 Displays results

In order to test the function that we'll be creating, we set up an element that specifies style information in its markup (#2) and a stylesheet that provides style rules that will be applied to the element (#1). It is our expectation that the computed styles will be the result of applying both the immediate and the applied styles to the element.

We then define our new function that accepts an element, and the style property that we which to find the computed value for (#3). And to be especially friendly (after all we're ninjas – making things easier for those using our code is part of the job) we'll allow multi-word property names to be specified in either format: dashed or camel-cased. In other words, we'll accept either of `backgroundColor` or `background-color`. We'll see how we can accomplish that in just a little bit.

The first thing we want to do is to check if the standard means is available – which will be true in all cases but older versions of IE – and if so, proceed to obtain the computed style interface, which we store in a variable for later reference (#4). We want to do things this way, as we don't know how expensive making this call may be, and it's likely best to avoid repeating it needlessly.

If that succeeds (and we can't think of any reason why it wouldn't, but it frequently pays to be cautious), we call the `getPropertyValue()` method of the interface to get the

computed style value (#5). But not before we adjust the name of the property to accommodate either the camel-cased or dashed version of the property name. The `getPropertyValue()` method expects the dashed version, so we use the String's `replace()` method, with a simple but clever regular expression, to insert a dash before every uppercase character, and then lowercase the whole thing. (Bet that was easier than you thought it would be.)

If we detected that the standard method was not available, we test to see if the IE-proprietary `currentStyle` property is available, and if so, we transform the property name by replacing all instances of a lowercase character preceded with a dash with the uppercase equivalent (to convert and dashed property names to camel case) and return the value of that property (#6).

Note in all cases, if anything goes awry, we simple return with no value.

To test the function, we make a number of calls to the function, passing various style names in various formats, and display the results (#7) as shown in figure 12.12.

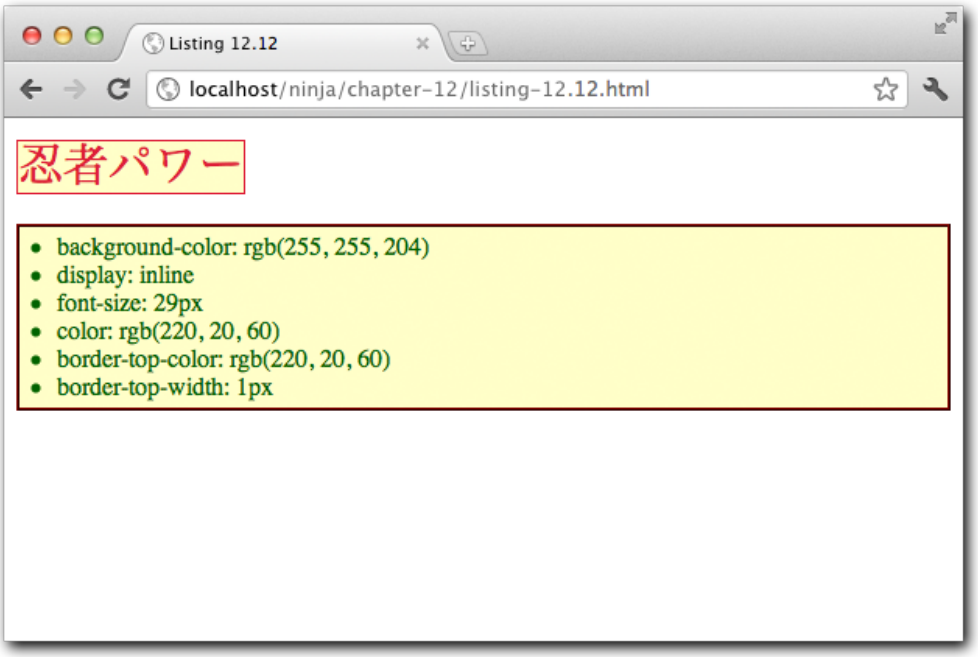

Figure 12.12 Computed styles include all styles specified on the element as well as those inherited from stylesheets

Note that the styles are fetched regardless of whether they were explicitly declared on the element, or inherited from the stylesheet.

Also note that the `color` property, specified in both the stylesheet and directly on the element, returns the explicit value. Styles specified by an element's `style` attribute always take precedence over inherited styles, even if marked `!important`.

There's one more topic that we need to be aware of when dealing with style properties; that of *amalgam* properties. CSS allows us to use a shortcut notation for the amalgam of properties such as the `border-` properties. Rather than forcing us to specify colors, widths and border styles individually and for all four borders, we can use a rule such as:

```
border: 1px solid crimson;
```

We used this exact rule in listing 12.12. This saves us a lot of typing, but we need to be aware that when we retrieve the properties, we need to fetch the low-level individual properties. So we can't fetch `border`, but we fetch styles such as `border-top-color`, and `border-top-width` just as we did in our example.

It can be a bit of a hassle, especially when all four styles were given the same values, but that's the hand we've been dealt.

12.5 Summary

When it comes to cross-browser compatibility issues, getting and setting DOM attributes, properties and styles may not be the worst area of JavaScript development for the browsers, but it's certainly has its fair share of issues. Thankfully we've learned that these issues can be handled in ways that are cross-browser compliant without resorting to browser detection.

Here are the important points to take away from this chapter:

- Attribute *values* are set from the attributes placed on the element markup.
- When retrieved, the attribute values may represent the same values, but may sometimes be formatted differently than specified in the original markup.
- Properties are created on the element that represent the attribute values.
- The keys for these properties may vary form the original attribute name, as well as across browsers, and the values may be formatted differently from either the attribute value or original markup.
- When push comes to shove, we can retrieve the original markup value by diving into the original attributes nodes in the DOM and getting the value from them.
- Dealing with the properties is usually more peformant than using the DOM attribute methods.
- Versions of IE prior to IE9 do not allow the `type` attribute of `<input>` elements to be changed once the element is part of the DOM.
- The `style` attribute poses some unique challenges and doesn't contain the computed style for the element.
- Computed styles can be fetched from the `window` using a standardized API in modern browsers, and via a proprietary property on IE8 and earlier.

13
Surviving events

In this chapter:

- Why events are such an issue
- Techniques for bind and unbinding events
- Using custom events
- Triggering events
- Event bubbling and delegation

The management of DOM events *should* be relatively simple, but, as you may have guessed by the fact that we're devoting an entire chapter to it, sadly it's not.

While all browsers provide relatively stable APIs for managing events, they do so with differing approaches and implementations. And even beyond the challenges posed by browser differences, the features that *are* provided by the browsers are insufficient for most of the tasks that need to be handled by even somewhat complex applications.

Because of these shortcomings, the end result is a near-full duplication of the existing DOM APIs in JavaScript libraries. Again, because this book assumes that you're either not using a library, or plan to write one of your own, we won't rely upon these previous implementations but, rather, will reveal the secrets that went into creating them in the first place.

Everyone who's made it this far into the book is liable to be familiar with the typical use of the DOM Level 0 Event Model, in which the event handlers are established via element properties or attributes. For example, if ignoring the principles of Unobtrusive JavaScript:

```
<body onload="doSomething()">
```

Or, if moving the behavior out of the structure:

```
window.onload = doSomething;
```

But DOM Level 0 events have severe limitations that make them unsuitable for reusable code, or for pages with any level of complexity. The DOM Level 2 Event Model provides a

more robust API, but its use is problematic as it is unavailable in legacy IE browsers prior to IE9. And, as already pointed out, lacks a number of features that we really need.

We'll be dismissing the DOM Level 0 Event Model as borderline useless to us, and concentrate on DOM Level 2.

This chapter will help us to navigate the event-handling minefield, and teach us how to survive in the somewhat hostile environment in which the browsers place us.

13.1 Binding and unbinding event handlers

Under the DOM Level 2 Event Model, we bind and unbind event handlers with the standard `addEventListener()` and `removeEventListener()` methods for modern DOM-compliant browsers, and the `attachEvent()` and `detachEvent()` methods in legacy versions of Internet Explorer (those prior to IE9).

For clarity, we'll simply refer to the DOM Level 2 Event Model as the *DOM Model*, and the proprietary legacy IE model as the *IE Model*. The former is available in all modern versions of the "Big Five" browsers, and the latter is available in all versions of IE, but is *all* that is available to IE versions prior to IE9.

For the most part, the two approaches behave in a similar manner with one glaring exception: the IE Model doesn't provide a way to listen for the capturing stage of an event. Only the bubbling phase of the event handling process is supported by the IE Model.

Additionally, the IE Model's implementation doesn't properly set a context on the bound handler, resulting in `this`, within the handler, referring to the global context (`window`) instead of the target element. Moreover, the IE Model also doesn't pass the event information to the handler; it tacks it onto the global context.

This means that there are browser-specific ways that we need to do just about anything when dealing with events:

- Binding a handler
- Unbinding a handler
- Obtaining the event information
- Obtaining the event target

It'd hardly make for robust and reusable code to have to perform browser detection and do things one way or the other at each juncture in event handling, so let's see what we can do about creating a common set of APIs that'll cut through the mayhem.

Let's start by seeing how we can address the problems of multiple APIs and the fact that the context is not set by the IE Model by inspecting the code of listing 13.1.

Listing 13.1: Providing proper context when binding event handlers

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

  if (document.addEventListener) {                          #1

    this.addEvent = function (elem, type, fn) {             #2
```

```
      elem.addEventListener(type, fn, false);
      return fn;
    };

    this.removeEvent = function (elem, type, fn) {          #3
      elem.removeEventListener(type, fn, false);
    };

  }
  else if (document.attachEvent) {                          #4

    this.addEvent = function (elem, type, fn) {             #5
      var bound = function () {
        return fn.apply(elem, arguments);
      };
      elem.attachEvent("on" + type, bound);
      return bound;
    };

    this.removeEvent = function (elem, type, fn) {          #6
      elem.detachEvent("on" + type, fn);
    };

  }

</script>
```

#1 Checks for DOM Model
#2 Creates bind function using DOM Model
#3 Creates unbind function using DOM Model
#4 Checks for IE Model
#5 Creates bind function using IE Model
#6 Creates unbind function using IE Model

The code of Listing 13.1 adds two methods to the global context: `addEvent()` and `removeEvent()`, with implementations suited to the environment in which the script is executing. If the DOM Model is present, it is used, if not, and the IE Model is present, it is used. (No methods are created is neither model is present.)

The implementation is mostly straightforward. After checking if the DOM Model is defined (#1), we define thin wrappers around the standard DOM methods: one for binding event handlers (#2) and one for unbinding handlers (#3).

Note that our functions return the established handler as their values (the significance of that will be discussed in just a few moments), and pass the value `false` as the third parameter to the DOM methods. This identifies the handlers as *bubble* handlers; being intended for cross-browser environments, our functions do not support capture phase.

If the DOM Model is not present, we then check to see if the IE Model is defined (#4), and if so we defined the two functions using that model.

The definition of the unbinding function is another straightforward wrapping of the model function (#6), but the binding function is another matter (#5).

Remember that one of the primary reasons for doing this at all, aside from defining a uniform API, was to fix the problem that the context of the handler was not set to the event

target. So in the binding function, instead of simply passing the handler function (the `fn` parameter) to the model function, we first wrap it in an anonymous function that in turn calls the handler, but uses the `apply()` method to force the context to be the target element of the event. Then we pass *that* wrapping function to the model function as the handler.

That way, when the wrapped function is triggered by the event, the handler function will be called with the proper context. And as with the other functions, we return the handler as the function value; though this time we return the wrapper, not the function that was passed in `fn`.

This is important because in order to unbind the handler later, we need to pass a reference to the function that was established as the handler according to the model function. In this case, that's the wrapping function (stored in the `bound` variable).

Let's see how that works with a quick test. As the test requires user intervention, we won't be using asserts, but simply interacting with the page as observing the results.

The test for these functions is shown in listing 13.2.

Listing 13.2 Testing the event binding API

```
addEvent(window, "load", function () {                          #1

  var elems = document.getElementsByTagName("div");             #2

  for (var i = 0; i < elems.length; i++) (function (elem) {
    var handler = addEvent(elem, "click", function () {         #3
      this.style.backgroundColor =
          this.style.backgroundColor=='' ? 'green' : '';
      removeEvent(elem, "click", handler);                      #4
    });
  })(elems[i]);

});
```

#1 Establishes load handler
#2 Fetches test elements
#3 Establishes test handlers
#4 Unbinds handler

We want to wait until the DOM is loaded before we run the test, so we use the very API that we're testing to establish the rest of the of the test as a load event handler (#1). If our binding function doesn't work, the test will never even get a chance to run!

Within the load hander, we fetch references to all `<div>` elements on the page to serve as our test subjects (#2), and iterate over the resulting collection of elements.

For each target element, we use the `addEvent()` to establish a click handler for it (#3), storing the returned function reference in a variable named `handler`. We're doing this to establish the reference in the closure for the handler as we'll be referencing the handler function within itself. Note that we can't rely upon `callee` in this case because we know that when we're operating using the IE model that the returned function will not be the same one that we passed in.

Within the click handler, we reference the target element via `this` (proving that the context has been correctly set) and determine if the background color of the element has been set, and if not, set it to green. If it *has* been set, we unset it. If we were to leave things at that, each subsequent click on the element would toggle the background of the element between green and nothing.

But we don't leave it at that. Before the handler exits, it uses our `removeEvent()` function, and the `handler` variable bound into the closure, to remove the handler. Thus, once the handler has been triggered once, it should never trigger again.

If we add the following elements to our page, and ensure that no background is applied to them via stylesheets, we'd expect that clicking on each `<div>` would turn it green, and subsequent clicks would not toggle the background.

```
<div title="Click me">私をクリック</div>
<div title="but only once">一度だけ</div>
```

Loading the page into the browser and conducting this manual test verifies that our functions work as expected. The display shown in figure 13.1 depicts the state of the page when loaded into Chrome and after the first element has been clicked on multiple times, and the second element not at all.

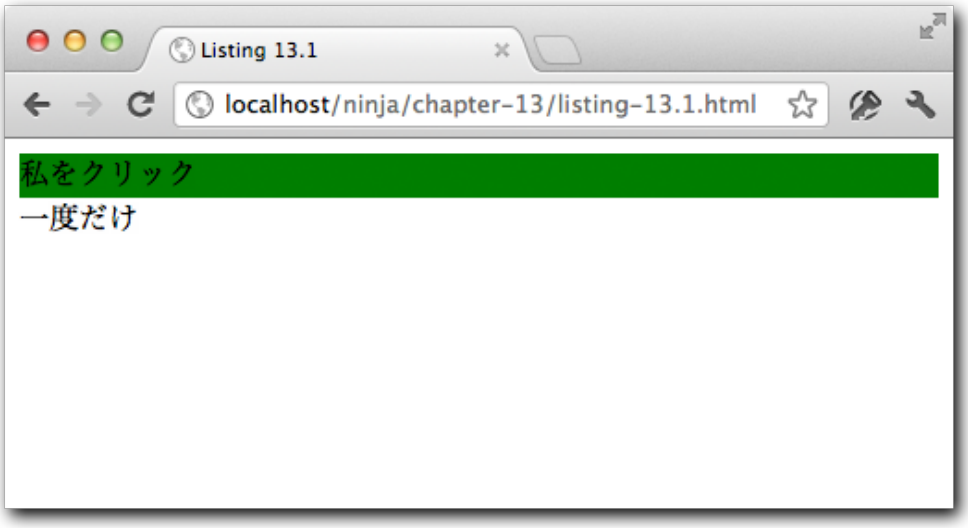

Figure 13.1 This manual tests proves that a uniform API can bind and unbind events

And figure 13.2 shows the same page, after the same actions, loaded into IE8, which does not support the DOM Model.

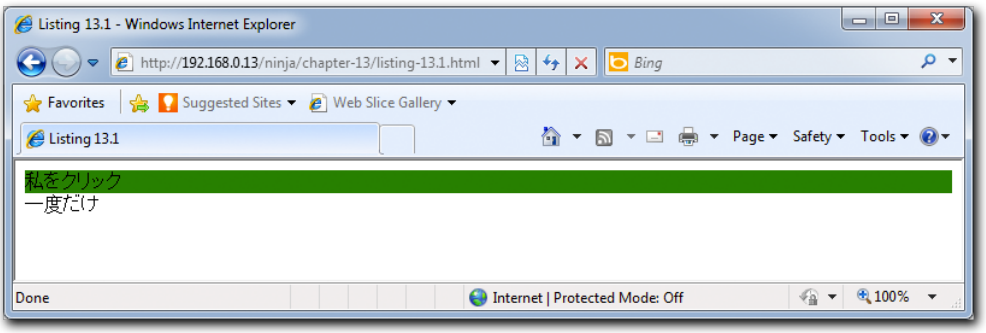

Figure 13.2 And it also works in legacy versions of IE!

That's a good start, but exhibits some weakness. The primary problem is that because we need to wrap the handler under legacy versions of IE, users of the API need to carefully record the reference to the handler as returned from the `addEvent()` function. Failing to do so will result in being unable to unbind the handler at a later point.

Another weakness is that this solution doesn't even address the problem of access to the event information.

So we've made improvements, but we're not where we want to be yet. Can we do better?

13.2 The event object

As we've already pointed out, the IE Model of event handling that we're forced to deal with in legacy browsers differs from the DOM Model in a number of ways. One of these was in the manner that an instance of the Event object is made available to the handlers. In the DOM Model, it's passed to the handler as its first parameter; in the IE Model, it is fetched from a property named `event` placed in the global context (`window.event`).

To make matters even worse, the contents of the Event instance vary across the models. What's a ninja to do?

The only reasonable way to work around this is to create a *new* object that simulates the browser's native event object, normalizing the properties within it the match the DOM Model. You might wonder why we wouldn't just modify the existing object, but that's not possible as there are many properties within it that cannot be overwritten.

Another advantage to cloning the event object is that it solves a problem caused by the fact that the IE Model stores the object in the global context. Once a new event starts, any previous event object is wiped out. Transferring the event properties to a new object whose lifetime we control solves any potential issues of this nature.

Let's try our hand at a function for event normalization, as found in Listing 13.3.

Listing 13.3: A function that normalizes the event object instance

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

  function fixEvent(event) {

    function returnTrue() { return true; }                              #1
    function returnFalse() { return false; }

    if (!event || !event.stopPropagation) {                             #2
      var old = event || window.event;

      // Clone the old object so that we can modify the values
      event = {};

      for (var prop in old) {                                           #3
        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;                                                    #4

  }
</script>
```

#1 Predefines often-used functions
#2 Test if fixing up is needed
#3 Clones existing properties
#4 Returns fixed-up instance

While this is a fairly long listing, most of what it's doing is fairly straightforward and we aren't going to exhaustively go through it line-by-line, but we will take the time to point out the most important aspects.

Essentially, the purpose of this function is to take an instance of Event and check to see if it conforms to the DOM model. If it foes not, we'll do our best to make it do so. You can read about the DOM Model's Event definition at http://www.w3.org/TR/DOM-Level-3-Events/#interface-Event.

The first thing that we do in our function is to define two functions (#1). Remember that JavaScript allows us to do this, and limits the scope of these functions to their parent function so that we don't need to worry about polluting the global namespace. We're going to need functions that return either `true` or `false` frequently throughout our fix-up code, so rather than use redundant function literals, we predefine these two functions: one that always returns `true`, and one that always returns `false`.

Then we test if we need to do anything (#2). If the instance doesn't exist (we assume that the event is defined on the global context in this case) or if it doesn't exist but the standard `stopPropogation` property is missing, we assume that we need to fix things up.

If we decide that fixing up is needed, we grab a copy of the existing event – either the one that was passed to us, or the one on the global context – and store it in a variable named `old`. Otherwise, we just fall through to the end of the function and return the existing event (#4).

If we're fixing up, we create an empty object to server as the fixed-up event, and copy all of the existing properties of the old event into this new object (#3). Then we proceed to fix things up to handle many of the common discrepancies between the W3C DOM event object and the one provided by the IE Model.

A few of the important properties that are "fixed" in this process:

- `target`: the property denoting the original source of the event. The IE Model 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` properties are IE's counterparts.
- `preventDefault`: doesn't exist in the IE Model, 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 in the IE Model, which stops 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 the IE Model to 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 us the final `pageX/Y` values).
- `which`: is equivalent to the key code pressed during a keyboard event. It can be duplicated by accessing the `charCode` and `keyCode` properties.
- `button`: matches the mouse button clicked by the user on a mouse event. The IE Model uses a bitmask (1 for left client, 2 for right click, 4 for middle click) so it needs to be converted to equivalent values (1, 2, and 3).

Another resource with great information on the DOM event object and its cross-browser capabilities is the set of 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

OK, so now we have a means to normalize the event instance. Let's see what we can do about gaining a margin of control over the binding process.

13.3 Handler management

For a number of reasons it would be advantageous to not bind event handlers directly to elements. If we use an intermediary event handler instead, and store all the handlers in a separate object, we can exert a level of control over the handling process including the ability to:

- Normalize the context of handlers
- Fix up the properties of event objects
- Handle garbage collection of bound handlers
- Trigger or remove some handlers by a filter
- Unbindi all events of a particular type
- Clone event handlers

We'll need to have access to the full list of the handlers bound to an element in order to achieve all of these benefits, so it really makes the most sense to avoid directly binding the events and to handle the binding ourselves. Let's take that on.

13.3.1 Centrally storing associated information

One of the best ways to manage the handlers associated with a DOM element is to give each element that we're working with a unique identifier (not to be confused with the DOM `id`), and then store all data associated with it in a centralized object. While it might seem more natural to store the information on each indovodual element, keeping the data in a central store will help us to avoid potential memory leaks in Internet Explorer, which is capable of losing memory under certain circumstances (namely, attaching functions to a DOM element that have a closure to a DOM node can cause memory to fail to be reclaimed after navigating away from a page).

Let's try our hand at writing a general way to centrally store information to be associated with particular DOM elements in listing 13.4.

Listing 13.4: Implementing a central object store for DOM element information

```
<div title="Ninja Power!">忍者パワー!</div>
<div title="Secrets">秘密</div>

<script type="text/javascript">
  (function () {

    var cache = {},                                    #1
        guidCounter = 1,
```

```
        expando = "data" + (new Date).getTime();

    this.getData = function (elem) {                          #2
      var guid = elem[expando];
      if (!guid) {
        guid = elem[expando] = guidCounter++;
        cache[guid] = {};
      }
      return cache[guid];
    };

    this.removeData = function (elem) {                       #3
      var guid = elem[expando];
      if (!guid) return;
      delete cache[guid];
      try {
        delete elem[expando];
      }
      catch (e) {
        if (elem.removeAttribute) {
          elem.removeAttribute(expando);
        }
      }
    };

  })();

  var elems = document.getElementsByTagName('div');           #4

  for (var n = 0; n < elems.length; n++) {                    #5
    getData(elems[n]).ninja = elems[n].title;
  }

  for (n = 0; n < elems.length; n++) {                        #6
    assert(getData(elems[n]).ninja === elems[n].title,
           "Stored data is " + getData(elems[n]).ninja);
  }

  for (n = 0; n < elems.length; n++) {                        #7
    removeData((elems[n]));
    assert(getData(elems[n]).ninja === undefined,
           "Stored data has been destroyed.")
  }

</script>
```

#1 Establishes scoped storage
#2 Defines get data function
#3 Defines remove data function
#4 Fetches test subjects
#5 Assigns associated data
#6 Tests that data was stored
#7 Tests that data is destroyed

In this example, we have set up two generic functions, `getData` and `removeData`, to respectively fetch the data block for a DOM element, and to remove it when no longer needed.

We're going to need some variables, which we don't want to contaminate the global scope with, so we do all our setup within an immediate function. This keeps any variable we declare within the scope of the immediate function, but available to out functions via their closures. (We told you back in chapter 5 that closures would play a central role in many things that we need to do.)

Within the immediate function we set up three variables (#1):

- `cache`, the object in which we'll store the data we want to associate with elements.
- `guidCounter`, a running counter that we'll use to generate element guids
- `expando`, the property name that we'll tack onto each element to store it's guid. We form this name using the current timestamp to help prevent any potential collisions with user-defined expandos.

Then we define the `getData()` method (#2). The first thing that this function does it to try and fetch any guid that's already been assigned to the element by a previous call to this method. If it's the first time that this method has been called on this element, the guid will not exist, so we create a new one (bumping the counter by one each time) and assign it to the element using the property name in `expando`, and we create a new empty object associated with the guid in the `cache`.

Regardless of whether the cache data for the element is newly created or not, it is returned as the value of the function. Callers of the function are free to add any data they would like to the cache as follows:

```
var elemData = getData(element);
elemData.someName = 213;
elemData.someOtherName = 2058;
```

Functions are data too, so we could even indirectly associate functions with the element:

```
elemData.someFunction = function(x){ /* do something */ }
```

With the `getData()` function established, we set about to create the `removeData()` function with which we can wipe out all the traces of the data in the event that it is no longer needed (#3).

In this function, we obtain the guid for the passed element, and short-circuit the function if there isn't one – meaning that the element has not been instrumented by `getData()`, or that it has already had the data removed.

Then we remove the associated data block from the cache, and we try to remove the expando. Under certain circumstances this may fail, in which case we catch the error and try to remove the attribute created on behalf of the expando.

This removes all traces of the instrumentation that `getData()` created: the cached data block, *and* the expando placed onto the elements.

That was pretty easy, let's make sure it works.

We had set up two `<div>` elements to use as test subjects, each with a unique `title` attribute. We get references to those elements (#4), and then iterate over them, creating a data element that we name `ninja` on each element that consists of the value of the `title` attribute for the element (#5).

Then we iterate over the elements again, checking that each one has an associated data value, with the name of `ninja`, that contains the same value as their title attributes (#6).

Finally, we iterate over the set once again, calling `removeData()` on each element, and verifying that the data no longer exists.

Figure 13.3 shows that all these test pass.

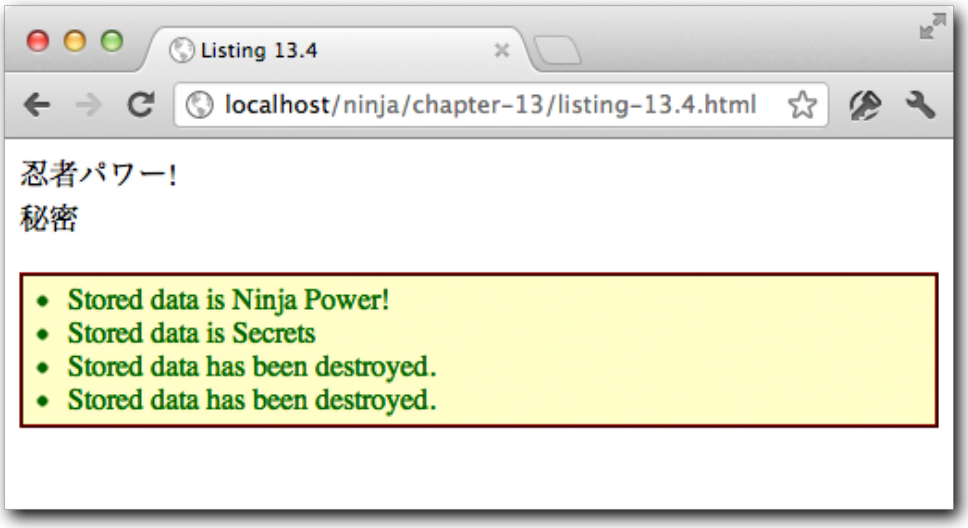

Figure 13.3 A few simple tests show that we can store data associated with an element that's not stored on the element itself

These functions can be quite useful beyond the scope of managing event handlers; using these functions we can attach any sort of data to an element. But we created these functions with the specific use case of associated event handling information with elements in mind.

Let's now use those functions to create our own set of functions to bind and unbind event handlers to elements.

13.3.2 Managing event handlers

In order to exert complete control over the event handling process, we'll need to create our own functions that wrap the binding and unbinding of events. By doing so we can present as unified an event-handling model as possible across all platforms.

Let's get to it. We'll start with binding event handlers.

BINDING EVENT HANDLERS

By writing a function to handle binding events, rather than just binding the handlers directly, we get the opportunity to keep track of the handlers and get our hooks into the process. We'll provide function to establish a function as a handler (binding), and to remove a function as a handle (unbinding). And we'll even though in a few helpful utility functions.

Let's start with binding the handlers with an `addEvent()` function as shown in listing 13.5.

Listing 13.5: A function to bind event handlers with tracking

```
(function(){

  var nextGuid = 1;

  this.addEvent = function (elem, type, fn) {

    var data = getData(elem);                              #1

    if (!data.handlers) data.handlers = {};                #2

    if (!data.handlers[type])                              #3
      data.handlers[type] = [];                            #3

    if (!fn.guid) fn.guid = nextGuid++;                    #4

    data.handlers[type].push(fn);                          #5

    if (!data.dispatcher) {                                #6
      data.disabled = false;
      data.dispatcher = function (event) {

        if (data.disabled) return;
        event = fixEvent(event);

        var handlers = data.handlers[event.type];          #7
        if (handlers) {
          for (var n = 0; n < handlers.length; n++) {      #7
            handlers[n].call(elem, event);                 #7
          }
        }
      };
    }

    if (data.handlers[type].length == 1) {                 #8
      if (document.addEventListener) {
        elem.addEventListener(type, data.dispatcher, false);
      }
      else if (document.attachEvent) {
        elem.attachEvent("on" + type, data.dispatcher);
      }
    }
```

```
    };

}();
```

#1 Gets associated data block
#2 Creates handler storage
#3 Creates array by type
#4 Marks instrumented functions
#5 Adds handler to list
#6 Creates über-handler (dispatcher)
#7 Calls registered handlers
#8 Registers dispatcher

Wow. That's seems like there's a lot going on, but each part is straightforward taken piece by piece.

First of all, because we're going to need some local storage, we use our usual trick of defining everything within an immediate function. And the storage that we need is a running counter for a guid value in variable `nextGuid`. These guid values will serve as unique markers; in a similar fashion to how we used them in the code of listing 13.4. We'll see exactly how in just a moment.

Then we define the `addEvent()` function, which accepts an element on which the handler is to be bound, the type of event, and the handler itself.

The first thing we do, upon entering the function, is to grab the data block associated with the element (#1), using the functions that we defined in listing 13.4, and store that block in variable `data`. This is done for two reasons:

1. We'll be referencing it a few times, so using a variable makes later references shorter.
2. There could be overhead in obtaining the data block, so we do it once.

Because we want to exert a high degree of control over the binding (and later, the unbinding) process, rather than add the passed handler onto the element directly, we're going to create our own über-handler that will serve as the actual event handler that we'll register with the browser, and keep track of the bound handlers so that we can execute them ourselves when appropriate.

We'll call this über-handler the *dispatcher* to distinguish it from the bound handlers that users of our function will pass into us. We'll be creating the dispatcher before the end of the function, but first we must crate the storage needed to keep track of the bound handlers.

We'll use a lot of just-in-time creation of storage, obtaining the storage as we need it, rather than pre-allocation it all up-front. After all, why create an array in which to store `mouseover` handlers if we never have any bound?

We're going to associate the handlers with their bound element via its data block (which we've conveniently obtained in variable `data`), so we test to see if the data block has a property named `handlers`, and if it does not, we create it (#2). Later invocation of the function on the same element will detect that the object exists, and will not try to create it subsequent times.

Within this object, we'll create arrays in which to store reference to handlers that should be executed, one for each event type. But, as we said earlier, we're going to smartly allocate them on an as-needed basis so we test to see if the handlers object has a property named after the passed-in `type` exists, and if not, create it (#3). This results in one array per event type, but only for the types that actually have handlers bound for them. That's just a wise use of resources.

Next we want to mark the functions that we're handling on behalf of the caller of our function (for reasons we'll see when we develop the unbinding function), so we add a `guid` property to the passed-in function, and bump the counter (#4). Note that once again we perform a check to make sure we only do this once per function (as a function can be bound as a handler multiple times if the page author wishes).

At this point, we know that we have a `handlers` object, and that it contains an array keeping track of handlers for the passed event type, so we push the passed handler onto the end of that array (#5). Note that this is pretty much the only action within this function that is guaranteed to execute whenever this function is called.

Now we're ready to deal with the dispatcher function. The first time that this function is called, no such dispatcher will exist. But we only need *one*, so we'll check to see if it exists and create it only when it doesn't (#6).

Within the dispatcher function – which will be the function that gets triggered whenever a bound event occurs – we check to see if a `disabled` flag has been set and terminate if so. We'll see in a few sections under what circumstances we might want to disable event dispatching for a time. Then we call the `fixup()` function that we created in listing 13.3, and then find and iterate through the array of handlers that were recorded for the type of event identified in the `event` instance. Each of these handlers is called, supplying the element as the function context, and the event object as its sole argument (#7).

Lastly, we check if we have just created the first handler for this type, and if so, we establish the delegate as the event handler for the event type with the browser using the means appropriate to the browser within which we are running (#8).

TIP

Note that if we moved this clause to within the conditional creation of the event handler array earlier in the function (#3) we could dispense with the check here. But we ordered the code as we did to make it easier to explain how it works (creating all of the data constructs prior to creating the delegate in which the constructs are used). In production code, it would be wise to move this clause and remove the need for the redundant check.

The final situation we end up with is that the functions passed to our routine are never established as actual event handlers – the real handler is the delegate – but rather they are stored and invoked by the delegate when an event occurs. This gives us the opportunity to make sure that the following things always happen regardless of platform:

- The `event` instance is fixed up.
- The function context is set to the target element.
- The event instance is passed to the handler as its sole argument.
- The event handlers will always be executed in the order in which they were bound.

Even Yoda would be proud of the level of control we can exert on the event handling process using this approach.

PICKING UP AFTER OURSELVES

We have a method to bind events, so we need one to unbind them. We didn't directly bind the handlers, choosing to exert control over the process with the delegate handler, so we can't rely upon the browser-supplied unbinding functions; we need to supply our own.

In addition to unbinding the bound handlers, we want to make sure that we tidy up after ourselves carefully. We took great care *not* to use up needless allocation in the binding function; it'd be silly to be remiss about reclaiming storage that becomes unused as a result of unbinding.

As it turns out, such tidying up is going need to be initiated from more than a single location, so we're going to capture it in its own function, as defined in listing 13.6.

Listing 13.6 Cleaning up the handler constructs

```
function tidyUp(elem, type) {

  function isEmpty(object) {                          #1
    for (var prop in object) {
      return false;
    }
    return true;
  }

  var data = getData(elem)

  if (data.handlers[type].length === 0) {             #2

    delete data.handlers[type];

    if (document.removeEventListener) {
      elem.removeEventListener(type, data.dispatcher, false);
    }
    else if (document.detachEvent) {
      elem.detachEvent("on" + type, data.dispatcher);
    }
  }

  if (isEmpty(data.handlers)) {                        #3
    delete data.handlers;
    delete data.dispatcher;
  }

  if (isEmpty(data)) {                                 #4
```

```
      removeData(elem);
    }
  }
```

#1 Detects empty objects
#2 Checks for types handlers
#3 Checks for any handlers
#4 Checks if data needed at all

We create a function named `tidyUp()` which accepts an element and an event type. The function will check to see if any handlers for this type are still around, and if not, clean up as much as possible, releasing any unneeded storage. This is a safe thing to do because, as we saw in the `addEvent()` function, if the storage is needed again later, that function will simply create it again as needed.

First off, we're going to need to be able to check if an object has any properties or not (is empty) in a number of locations. And since there's no "isempty" operator in JavaScript, we need to write our own check (#1). Because we're only going to use thus function within our `tidyUp()` function, we declare the `isEmpty()` function within it to keep its scope as close as possible.

We're going to be cleaning up the data block associated with the element, so we fetch it and store it in the `data` variable for later reference.

Then we start to check to see what, if anything, can be tidied away!

First, we check to see if the array of handlers associated with the passed type is empty (#2). If it is, it's no longer needed and we blow it away. Additionally, as there are no longer any handlers for this event type, we unbind the delegate that we registered with the browser as it is no longer needed.

Now that we have removed one of the arrays of handlers for an event type, there's a possibility that it may have been the only remaining such array, and its removal could leave the `handlers` object empty. So we test for that (#3), and remove the `handlers` property if its empty, and therefore useless. In such a case, the delegate is no longer needed either so it is also removed.

Finally, we test to see if all these removals have resulted in the data block associated with the element becoming pointless (#4), and if so, we jettison it as well.

And that's how we keep things spic and span.

UNBINDING EVENT HANDLERS

Now that we know that we can clean up after ourselves, pleasing Mr. Clean as well as Yoda, we're ready to tackle the function to unbind handlers that were bound with our `addEvent()` function.

To be as flexible as possible, we're going to give the callers of our functions the options of:

- Unbinding all bound events for a particular element, or
- Unbinding all events of a particular type from an element, or
- Unbinding a particular handler from an element

We'll allow these variations simply by providing a variable length argument list; the more information that the caller provides, the more specific the remove operation.

For example, to remove all bound events from an element:

```
removeEvent(element)
```

To remove all bound events of a particular type:

```
removeEvent(element,"click");
```

To remove a particular instance of a handelr:

```
removeEvent(element, "click",handler);
```

The latter assumes that we have maintained a reference to the original handler.

The unbinding function to accomplish all this is depicted in listing 13.7.

Listing 13.7 A function to unbind event handlers

```
this.removeEvent = function (elem, type, fn) {        #1

  var data = getData(elem);                           #2

  if (!data.handlers) return;                         #3

  var removeType = function(t){                       #4 utility function
    data.handlers[t] = [];
    tidyUp(elem,t);
  };

  if (!type) {                                        #5 remove all types
    for (var t in data.handlers) removeType(t);
    return;
  }

  var handlers = data.handlers[type];                  #6 get handlers for
type
  if (!handlers) return;

  if (!fn) {                                           #7 remove all of type
    removeType(type);
    return;
  }

  if (fn.guid) {                                       #8 remove one bound
function?
    for (var n = 0; n < handlers.length; n++) {
      if (handlers[n].guid === fn.guid) {
        handlers.splice(n--, 1);
      }
    }
  }
  tidyUp(elem, type);

};
```

#1 Declares the function
#2 Fetches associated element data
#3 Short-circuits if nothing to do
#4 Sets up utility function

#5 Removes all bound handlers
#6 Finds handlers for type
#7 Removes all for type
#8 Removes one bound handler

We start by defining our function signature with three parameters: the element, the event type, and the function (#1). Callers can omit trailing arguments as described earlier.

The next step is obtaining the data block associated with the passed element (#2).

TIP

As we're allowing a variable length argument list, it'd probably be a good idea to check to make sure that an element was provided, as it's not optional. How would you go about doing that?

Once we've obtained the block, we check to see if there are any bound handlers, and short-circuit the entire function if not (#3). Note that we didn't need to check inside the `handlers` object to see if it was empty, or if it contained empty lists of handlers, because of the tidying up that will happen as a result of the function that we developed in the previous section. That's going to help make this function a lot cleaner by eliminating empty data constructs and the need for complex checks for them.

If we make it through the previous check, we know that we're going to be removing bound handlers by event type; either of all types (if the `type` parameter is omitted), or of a specific type (identified by the `type` parameter). In either case we're going to be removing by type in more than one location, so to avoid needlessly repeated code, we define a utility function (#4) that, given a type `t`, removes all handlers for that type by replacing the array of handlers with an empty array, and then calls the `tidyUp()` function on that type.

With that function in place, we check to see if the `type` parameter was omitted (#5), and if so, go about removing all handlers for all types on the element. In this case, we simply return because our job is finished.

NOTE

We short-circuit this function at numerous points with `return` statements. Some developers dislike this style and prefer a single `return`, controlling flow with deeply nested conditionals. If you're one of these, you could try your hand at rejigger the function to use a single `return` (or implied return) when the explanation is complete.

If we make it to this point, we know that we've been provided with an event type for which we will be removing either all handlers (if the `fn` argument is omitted), or a specific handler for that type. So, in order to reduce code clutter, we grab the list of handlers for that type and store it in a variable named `handlers` (#6). If there aren't any, there's nothing to so, so we return.

If the `fn` argument was omitted, (#7) we call our removal utility function to blow away all of the handlers for the specified event type, and return.

Failing all the previous checks that might have caused us to remove something and then return, we know that we have a specific handler that's been passed to us for removal. But if it's not a handler that we've "touched" there's no need to bother looking for it, so we check to see if the `guid` property has been added to the function (which would have happened when the function was passed to the `addEvent()` method) and ignore it if not.

If it is a handler that we've instrumented, we look through the list of handlers for it, removing any instances that we find (there could be more than one). And, as usual, we tidy up before we return.

SMOKE-TESTING THE FUNCTIONS

Let's look at a simple smoke test for our bind and unbind functions. As before, we're going to set up small page that uses manual intervention to run a simple visual test, as shown lin listing 13.8.

NOTE

The term "smoke testing" means to make a cursory test of the major functions of whatever is being tested. It is far from a rigorous test and simply makes sure that the test subject seems to work on a gross basis. The term originates from the early world of electronics where the first test performed on a new circuit was to simply plug it in and see if anything burst into flames!

Listing 13.8 Smoke-testing the event functions

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

  addEvent(window, "load", function () {                        #1

    var subjects = document.getElementsByTagName("div");        #2

    for (var i = 0; i < subjects.length; i++) (function (elem) {

      addEvent(elem, "mouseover", function handler(e) {         #3
        this.style.backgroundColor = "red";
      });

      addEvent(elem, "click", function handler(e) {              #4
        this.style.backgroundColor = "green";
        removeEvent(elem,"click",handler);
      });

    })(subjects[i]);

  });
</script>

<div id="testSubject1" title="Click once">一度クリックします</div>
```

```
<div id="testSubject2" title="mouse over">マウス</div>
<div id="testSubject3" title="many times">何度</div>
```

#1 Binds load handler
#2 Collects test subjects
#3 Bings mouse events
#4 Binds click events

In this simple test we're going to bind three different types of events, and unbind one of them. First, we establish a handler for the page `load` event (#1), as our test subjects (three `<div>` elements) are defined after our script block. So we need to delay the execution of the rest of the script until after the DOM has been loaded.

When that event fires, our handler collects all the `<div>` elements (#2) and iterates over them. For each, we establish:

- A `mouseover` handler that turns the element red (#3).
- A `click` handler that turns the element green, then unbinds itself, such that each element will react to a click exactly *once* (#4).

Loading the page into a browser, we perform the following steps:

1. We mouse over the elements, observing that they all turn red when we do so. This verifies that the `mouseover` event was correctly bound, and activated.
2. We then click on an element, observing that it turns green. This verifies that the `click` handler was correctly bound and activated. Figure 13.4 shows the page at this stage.
3. Then we run the mouse over the clicked element, observe that it turns back to red (as expected because of the `mouseover` handler), and click on the element again.
4. If the click hander were correctly unbound, the click handler will not trigger (which would cause the element to turn to green again) and the element would remain red. Our observation verifies that this is the case.

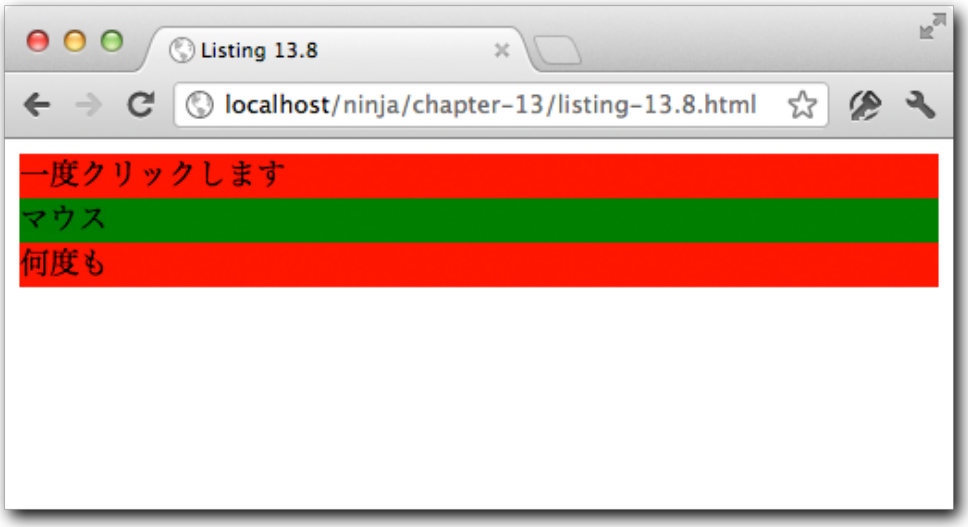

Figure 13.4 Our smoke test shows that at least some of the major features of our functions are operating correctly

This is far from a rigorous test. As an exercise, try your hand at writing a series of asserts that will automate testing of the functions, exercising all of the functions' features.

BONUS

In the events.js file included in the code examples for this book we also included a handy `proxy()` function that can be used to cajole the function context of a function to be used as a handler to be something other than the event target when triggered. This is the exact same treachery that we explored in section 4.3.

So now we can exert a great deal of control over the binding and unbind of events. Let's see what other magic wands we can wave over events.

13.4 Triggering events

Under normally circumstances events are triggered when activity such as user actions, browser actions, or network activity take place. However, it's easy to imagine circumstances under which we'd want to trigger the same activity under script control. (We'll be seeing shortly that this is not only desirable, but also necessary when working with custom events.)

For example, there may be a click handler that we not only want to trigger when the user clicks the button, but when some other activity occurs that we're executing script in response to.

We could very un-ninja-like about it, and simply repeat the code, but we know better than that. A viable approach would be to factor the common code into a named function that we could call from any location. But that solution isn't without its namespace issues, and could detract from the clarity of the code base. Besides, most of the time we'd want to do this, we're not going to merely want to call a function, but to simulate the event.

So the ability to trigger event handlers without a "real" event would be an advantage that we'd like to give ourselves.

When a triggering a handler function there are a number of things that we need to make sure happen. We want to:

- Trigger the bound handler on the element that we target
- Cause the event to bubble up the DOM triggering any other bound handlers
- Cause the default action to be triggered on the target element (when it has one)

A function that handles all of this, presupposing that we are utilizing the functions of the previous section to handle event binding, can be seen in Listing 13.9.

Listing 13.9: Trigger a bubbling event on an element

```
function triggerEvent(elem,event) {

  var elemData = getData(elem),                          #1
      parent = elem.parentNode || elem.ownerDocument;

  if (typeof event === "string") {                       #2
    event = { type:event, target:elem };
  }
  event = fixEvent(event);                               #3

  if (elemData.dispatcher) {                              #4
    elemData.dispatcher.call(elem, event);
  }

  if (parent && !event.isPropagationStopped()) {         #5
    triggerEvent(parent, event);
  }

  else if (!parent && !event.isDefaultPrevented()) {     #6

    var targetData = getData(event.target);

    if (event.target[event.type]) {                      #7

      targetData.disabled = true;                        #8

      event.target[event.type]();                        #9

      targetData.disabled = false;                       #10

    }
```

```
    }
  }
```

#1 Fetches element data and a reference to the parent (for bubbling).
#2 If an event name was passed as a string, creates an event out of it
#3 Normalizes the event properties.
#4 If the passed element has a dispatcher, executes the established handlers.
#5 Unless explicitly stopped, recursively calls this function to bubble the event up the DOM.
#6 If at the top of the DOM, triggers the default action unless disabled.
#7 Checks if the target has a default action for this event.
#8 Temporarily disables event dispatching on the target as we have already executed the handler.
#9 Executes the default action.
#10 Re-enables event dispatching.

Our `triggerEvent` function accepts two parameters:

- the element upon which the event will be triggered
- the event that is to be triggered

The latter can be either an event object or just a string containing the event type.

To trigger the event, we traverse from the initial event target all the way up to the top of the DOM, executing any handlers that find along the way (#4) . When we reach the document element, execution of bubbling is over and we can execute the default action for the event type on the target element, if it has one (#9).

Note that during the event bubbling activity, we make sure that propagation hasn't been stopped, and before executing any default action, we also check that it hasn't been disabled. Also note how we disable our event dispatcher (#8) while executing the default action as we've already triggered the handlers ourselves and don't want to risk double execution.

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 execute it.

The ability to trigger events under script control is something that's really useful in its own right, but we'll also find that it implicitly allows custom events to just work.

Custom events?

13.4.1 Custom events

Haven't you ever fervently desired the ability to trigger your own custom events?

Imagine a scenario where you want to perform an action, but it will be triggered under a variety of conditions from different pieces of code; perhaps even code that's in shared script files.

The novice would repeat the code everywhere it's needed. The intermediate would create a global function and call it from everywhere it's needed. The ninja uses custom events.

Let's chat a bit about why we'd want to consider that.

LOOSE COUPLING

Picture the scenario where we are doing operations from shared code, and want to let page code know when its time to react to some condition. If we use the global function approach,

out shared code needs to define a fixed name for the function, and all pages that use the shared code need to define such a function.

Moreover, what if there are multiple things to do when the triggering condition occurs? One of the advantages of event handlers is that we can establish as many as we want, and these handlers are completely independent.

The disadvantages that we're seeing are a result of ***close coupling***; in which the code that detects the conditions has to intimately know the details of the code that will react to that condition.

Loose coupling, on the other hand, occurs when the code that triggers the condition doesn't know anything about the code that will react to the condition, or even if there's anything that will react to it at all.

Event handling is a good example of loose coupling. When a button click event is triggered, the code triggering the event has no knowledge of what handlers we've established on the page, or even if there are any. Rather, the click event is simply pushed onto the vent queue (see chapter 3 for a refresher, if needed) and could care less what happens after that. If handlers have been established for the click event, they will eventually be invoked in a completely independent one-to-many fashion.

There's much to be said for loose coupling. In our scenario, the shared code, when it detects an interesting condition, simply triggers a signal of some sort that says "This interesting thing has happened; anyone interested can deal with it", and it could give a darn if anyone's interested or not.

And rather than invent our own signaling system, we can use the code that we've already developed in this chapter to leverage event handling as our signaling mechanism.

Let's examine a concrete example.

AN AJAX-Y EXAMPLE

Let's pretend that we've written some shared code that will be performing Ajax request on our behalves. The pages that this code will be used on want to be notified when an Ajax request begins, and when it ends; each page has its own things that it needs to do when these "events" occur.

For example, on one page using this package, we have an animated GIF of a spinning pinwheel that we want to display when an Ajax request starts, and to hide when the request completes, in order to give the user some visual feedback that a request is being processed.

If we imagine the start condition as an event named `ajax-start`, and the stop condition as `ajax-complete`, wouldn't it be grand if we could simply establish event handlers on the page for these event that show and hide the image as appropriate?

Consider:

```
var body = document.getElementsByTagName('body')[0];

addEvent(body,'ajax-start',function(e){
  document.getElementById('whirlyThing').style.display = 'inline-block';
});
```

```
addEvent(body,'ajax-complete',function(e){
  document.getElementById('whirlyThing').style.display = 'none';
});
```

Sadly these events don't really exist.

But, as we've developed the code to add event handlers, and code to mimic the triggering of handlers, we can use that code to simulate custom events that don't rely upon the browser understanding our custom event types.

TRIGGERING CUSTOM EVENTS

Custom events are a way of simulating the experience (to the user of our shared code) of a real event without having to use the browser's underlying event support. Using the work that we've already set into place to support cross-browser events thus far, supporting custom events turns out to be something that we've already implemented!

Within the code that we've already written for `addEvent()`, `removeEvent()`, and `triggerEvent()`, nothing has to change in order to support custom events. Functionally, there is no difference between a real browser event that will be fired by the browser, and an event that doesn't really exist and will only fire when triggered manually.

We can see an example of triggering a custom event in Listing 13.10.

Listing 13.10: Using custom events

```
<!DOCTYPE html>
<html>
  <head>
    <title>Listing 13.10</title>
    <meta charset="utf-8">
    <script type="text/javascript" src="data.js"></script>
    <script type="text/javascript" src="fixup.js"></script>
    <script type="text/javascript" src="events.js"></script>
    <script type="text/javascript" src="trigger.js"></script>
    <style type="text/css">
      #whirlyThing { display: none; }
    </style>

    <script type="text/javascript" src="ajaxy-operation.js"></script>

    <script type="text/javascript">

      addEvent(window,'load',function(){

        var button = document.getElementById('clickMe');          #1
        addEvent(button,'click',function(){
          performAjaxOperation(this);
        });

        var body = document.getElementsByTagName('body')[0];

        addEvent(body,'ajax-start',function(e){                    #2
          document.getElementById('whirlyThing')
            .style.display = 'inline-block';
        });
```

```
        addEvent(body,'ajax-complete',function(e){                #3
          document.getElementById('whirlyThing')
            .style.display = 'none';
        });

      });

    </script>
  </head>
  <body>

    <button type="button" id="clickMe">Start</button>             <!--4-->

    <img id="whirlyThing" src="whirly-thing.gif">                 <!--5-->

  </body>
</html>
```

#1 Adds a click handler to the button that will trigger an Ajax operation that will take about 5 seconds. Note that this handler knows nothing about the "whirly-thing" image.
#2 Establishes a handler for a custom event named `ajax-start` on the body element that will cause the image to be displayed. Note that there is no coupling with the code that reacts to the button click.
#3 Establishes a handler for a custom event named `ajax-complete` on the body element that will cause the image to be hidden. No coupling here either.
#4 Creates a button for us to click on.
#5 Defines the "whirly-thing" image that should only be shown while the Ajax operation is under way.

In this manual test, we cursorily check custom events by establishing the scenario that we described in the previous section: an animated image (#5) will be displayed while an Ajax operation is under way. The operation is triggered by the click (#1) of a button (#5).

In a completely decoupled fashion, a handler for a custom event named `ajax-start` is established (#2), as is one for the `ajax-complete` custom event (#4). The handlers for these events show and hide the "whirly-thing" image (#5) respectively.

Note how the three handlers know nothing of each other's existence. In particular the button click handler has no responsibilities with respect to showing and hiding the image.

The Ajax operation itself is simulated with the following code:

```
function performAjaxOperation(target) {

  triggerEvent(target,'ajax-start');

  window.setTimeout(function(){
    triggerEvent(target,'ajax-complete')
  },5000);

}
```

The function triggers the `ajax-start` event, pretending that an Ajax request is about to be made. The choice of the button as the initial target of the event is arbitrary. As the handlers are established in the body (a customary location), all events will eventually bubble up to the body, and the handler will be triggered.

The function then issues a 5-second timeout, simulating an Ajax request that spans five seconds. When the timer expires, we pretend that the response has been returned and trigger an `ajax-complete` event to signify that the Ajax operation has completed.

The displays are shown in figure 13.5.

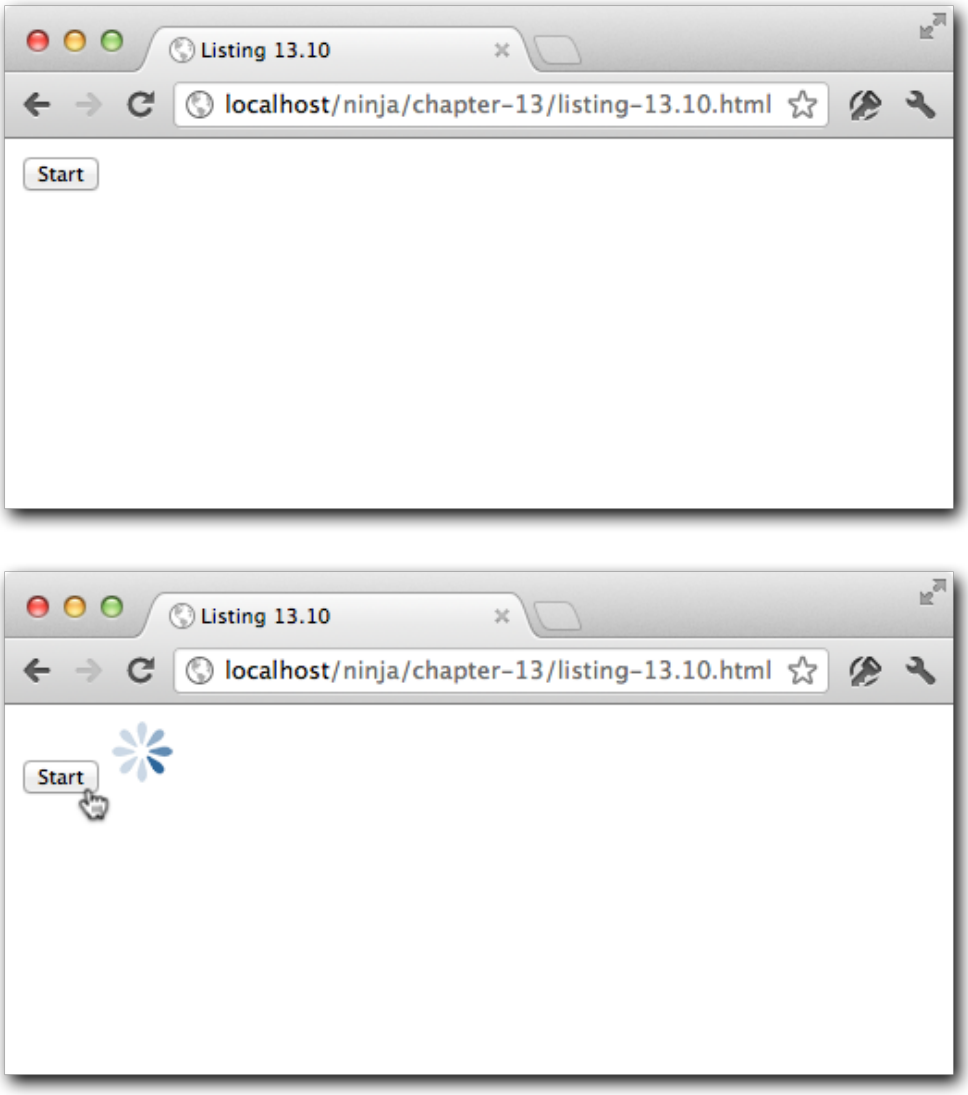

Figure 13.5 Custom events can be used to cause code to trigger in a de-coupled manner

Notice the high degree of de-coupling throughout this example. The shared Ajax operation code has no knowledge of what the page code is going to do when the events are triggered, or even if there's page code to trigger at all. The page code is modularized into small handlers that don't know about each other. And, the page code has no idea how the shared code is doing its thing, it just reacts to events that may or may not be triggered.

This level of decoupling helps to keep code modular, easier to write, and a lot easier to debug when something goes wrong. It also makes it easy to share portions of code and to move them around without fear of violating some coupled dependency between the code fragments.

Decoupling is a fundamental advantage when using custom events in our code and allows us to develop applications in a much more expressive and flexible manner.

Even though you may not yet have realized it, the code of this section was not only a good example of decoupling, it was also a good example of *delegation*.

13.5 Bubbling and delegation

Simply put, ***delegation*** is the act of establishing event handlers at higher levels in the DOM than the items of interest.

Recall how, even though the image buried within the DOM was the element that we wanted to be affected by the custom events, we established the handlers that would cause the image's visibility to be affected on the body element. This was an example of delegating authority over the image to an ancestor element, in this case, the body element.

But it's not limited to custom tags, or even the body element. Let's imagine a scenario using more mundane event types and elements.

13.5.1 Delegating events to an ancestor

Let's say that we wanted to visually indicate whether a cell within a table had been clicked on or not by the user by initially displaying a white background for each cell, and changing the background color to yellow once the cell was clicked upon. Sounds easy enough. We can just iterate through all the cells, and establish a handler on each one that changes the background color property.

```
var cells = document.getElementsByTagName('td');

for (var n = 0; n < cells.length; n++) {
  addEvent(cells[n],'click',function(){
    this.style.backgroundColor = 'yellow';
  });
}
```

Sure this works, but is it elegant? Not very.

We're establishing the exact same event handler on potentially hundreds of elements, that all do *the exact same thing*.

A much more elegant approach is to establish a single handler at a level higher than the cells that can handle all the events using the event bubbling provided by the browser. We know that all the cells will be descendants of their enclosing table, and we know that we can

get a reference to the element that was clicked upon via `event.target`. So it's much more suave to *delegate* the event handling to the table as follows:

```
var table = document.getElementById('#someTable');

addEvent(table,'click',function(event){
  if (event.target.tagName == 'td')
    event.target.style.backgroundColor = 'yellow';
});
```

Here, we establish one handler that easily handles the work of changing the background color for all cells clicked in the table. This is much more efficient and elegant.

Event delegation is one of the best techniques available for developing high performance, scalable web applications.

Because event *bubbling* is the only technique available across all browsers, event *capturing* doesn't work in IE version prior to IE9, it's important to make sure that event delegation is applied to elements that are ancestors of the elements that are the targets of the events. That way, we are sure that the events will eventually bubble up to the element to which the handler has been delegated.

That all seems logical and easy enough; but, there always seems to be a "but", doesn't there?

13.5.2 Working around browser deficiencies

Unfortunately the `submit`, `change`, `focus`, and `blur` events all have serious problems with their bubbling implementations in various browsers. If we want to employ event delegation – and we do – we must figure our how these deficiencies can be worked around.

To start, the `submit` and `change` events don't bubble at all in legacy Internet Explorer; the W3C DOM-capable browsers implement bubbling consistently. So, as we have done throughout this book, we'll use a technique that is capable of gracefully determining if the problem exists and needs to be worked around. In this case determining if an event is capable of bubbling up to a parent element.

One such piece of detection code was written by Juriy Zaytsev (as described at http://perfectionkills.com/detecting-event-support-without-browser-sniffing/) and can be seen in listing 13.11.

Listing 13.11: Event bubbling detection code originally written by Juriy Zaytsev

```
function isEventSupported(eventName) {

  var element = document.createElement('div'),          #1
      isSupported;

  eventName = 'on' + eventName;                         #2
  isSupported = (eventName in element);                 #2

  if (!isSupported) {                                   #3
    element.setAttribute(eventName, 'return;');
    isSupported = typeof element[eventName] == 'function';
  }
```

```
    element = null;                                          #4

    return isSupported;
  }
```

#1 Creates a new element that we'll perform tests upon. We'll delete it later.
#2 Tests to see if the event is supported by checking if a property supporting the event is present on the element,.
#3 If the simple approach fails, we try a more complicated one by trying to create an event handler attribute and checking if it "sticks".
#4 Regardless of result we wipe out the temporary element.

The bubbling detection technique works by checking to see if an existing on*type* (where *type* is the type of the event) property exists on a `<div>` element (#2). A `<div>` element is used because those elements typically have the most diverse types of events bubbled up to them (including `change` and `submit`).

We can't count on a element already existing in the page – and even if we could, we don't really want to start sticking our fingers in someone else's element – so a temporary element is created for use to play around with (#1).

If this quick and simple test fails, we have a more invasive one we try (#3). If the on*type* property doesn't exist, we create an `ontype` attribute, giving it 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 upon bubbling.

Now let's use this detection code as the basis for implementing properly-working event bubbling across all browsers.

BUBBLING SUBMIT EVENTS

The `submit` event is one of the few that doesn't bubble in legacy Internet Explorer, but thankfully, it is one of the easiest events to simulate.

A `submit` event can be triggered in a one of two ways:

- By triggering an input or button element with type of "submit", or an input element of type "image". Such elements can be triggered with a click, or with enter or spacebar when focused.
- By pressing "enter" while inside a text or password input.

With knowledge of these two cases, we can piggyback on the two triggering events, `click` and `keypress`, both of which bubble normally.

The approach we'll take (for now) is to create special functions to bind and unbind `submit` events. If we determine that submit events need to be handled specially because browser support is lacking, we'll establish the "piggybacking"; otherwise, we'll just bind/unbind the handler normally.

Let's see how in listing 13.12.

Listing 13.12: Piggy-backing submit bubbling on click or keypress

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

  (function(){

    function isInForm(elem) {                                    #1
      var parent = elem.parentNode;
      while (parent) {
        if (parent.nodeName.toLowerCase() === "form") {
          return true;
        }
        parent = parent.parentNode;
      }
      return false;
    }

    function triggerSubmitOnClick(e) {                           #2
      var type = e.target.type;
      if ((type === "submit" || type === "image") &&
          isInForm(e.target)) {
        return triggerEvent(this,"submit");
      }
    }

    function triggerSubmitOnKey(e) {                             #3
      var type = e.target.type;
      if ((type === "text" || type === "password") &&
          isInForm(e.target) && e.keyCode === 13) {
        return triggerEvent(this,"submit");
      }
    }

    this.addSubmit = function (elem, fn) {                       #4

      addEvent(elem, "submit", fn);                              #5
      if (isEventSupported("submit")) return;                    #5

      if (elem.nodeName.toLowerCase() !== "form" &&              #6
          getData(elem).events.submit.length === 1) {
        addEvent(elem, "click", triggerSubmitOnClick);
        addEvent(elem, "keypress", triggerSubmitOnKey);
      }

    };

    this.removeSubmit = function (elem, fn) {                     #7

      removeEvent(elem, "submit", fn);                            #8
      if (isEventSupported("submit")) return;                     #8

      var data = getData(elem);

      if (elem.nodeName.toLowerCase() !== "form" &&               #9
          !data || !data.events || !data.events.submit) {
        removeEvent(elem, "click", triggerSubmitOnClick);
```

```
            removeEvent(elem, "keypress", triggerSubmitOnKey);
          }
        };

      })();


    </script>
```

#1 Defines a utility function that we'll use to check if the passed element is within a form or not.
#2 Pre-defines a handler for clicks that will check to see if a submit event should piggyback on this event and triggers one if so.
#3 Pre-defined a handler for key presses that will check to see if a submit event should piggyback on this event and triggers one if so.
#4 Creates the special function for binding submit events.
#5 Binds the submit handler normally and short-circuits the rest of the function if browser support is adequate.
#6 If not a form and is the first submit handler, establish the handlers for click and keypress piggybacking.
#7 Creates the special function for unbinding submit events.
#8 Unbinds the handler normally, and exits if browser support is adequate.
#9 If not a form and is the last handler to be unbound, remove the piggybacking handlers.

First of all, we're using the immediate function technique – that should be familiar by now – to create a self-contained environment for our code. But before we get into the meat of adding special support for submit events, we're going to define a few things up-front that we'll need later.

First, we're going to need to determine if an element is inside a form in a couple of locations, so we defined a function named `isInForm()` (#1) that does that for us. It simply traverses the ancestor tree of the element to determine if one of its ancestors is a form.

Then we define two functions that we'll use as event handlers: one for clicks, and one for key presses.

The first such function (#2) triggers a `submit` event if the element is in a form, and if the target element has submit semantics (has a type of `submit`, or is an image input element). The second function (#3) triggers a `submit` event if the key press is the Enter key, and the target element is in a form and is a text or password input element.

With those helpers defined, we're ready to write the bind and unbind functions.

The `addSubmit()` binding function (#4) first establishes the submit handler as normal, using the `addEvent()` function (#5), and then returns if the browsers properly supports submit bubbling. If not, we check to make sure we're not bind to a form (in which case bubbling isn't a problem) and if this is the first submit handler being bound (#6). If so, we establish the piggybacking handlers for clicks and key presses.

The `removeSubmit()` unbinding function (#7) works in a similar fashion. We unbind the submit event as normal and exit if the browser adequately supports submit bubbling (#8). If it doesn't we unbind the piggybacking handers if the target is not a form and this is the last of the submit handlers being unbound (#9).

TO DO

We created this logic as separate functions that use the services of `addEvent()` to make it easier to focus on just the code necessary to handle submit events. But having separate functions is obviously not very caller friendly. What we should really do is to put this logic inside `addEvent()` so that all this would happen automatically and invisibly on behalf of the caller of our code. How would you go about merging this capability into `addEvent()`?

This approach tends to apply well to fixing other DOM bubbling events, such as the `change` event.

BUBBLING CHANGE EVENTS

The `change` event is another event that doesn't bubble properly in the IE event model of legacy Internet Explorer. Unfortunately it's significantly harder to implement properly when 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 been changed.
- The `beforeactivate` for getting the previous value before a new one is set.

Listing 13.13 shows an implementation of special functions to bind and unbind change handlers that piggybacks on all of the above events.

Listing 13.13: An implementation of a cross-browser bubbling `change` event.

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

  (function(){

    this.addChange = function (elem, fn) {                          #A

      addEvent(elem, "change", fn);                                 #B
      if (isEventSupported("change")) return;                       #B

      if (getData(elem).events.change.length === 1) {               #C
        addEvent(elem, "focusout", triggerChangeIfValueChanged);
        addEvent(elem, "click", triggerChangeOnClick);
        addEvent(elem, "keydown", triggerChangeOnKeyDown);
        addEvent(elem, "beforceactivate", triggerChangeOnBefore);
      }
    };

    this.removeChange = function (elem, fn) {                       #D

      removeEvent(elem, "change", fn);                              #E
      if (isEventSupported("change")) return;                       #E
```

```
  var data = getData(elem);
  if (!data || !data.events || !data.events.submit) {           #F
    addEvent(elem, "focusout", triggerChangeIfValueChanged);
    addEvent(elem, "click", triggerChangeOnClick);
    addEvent(elem, "keydown", triggerChangeOnKeyDown);
    addEvent(elem, "beforceactivate", triggerChangeOnBefore);
  }
};

function triggerChangeOnClick(e) {                               #G
  var type = e.target.type;
  if (type === "radio" || type === "checkbox" ||
      e.target.nodeName.toLowerCase() === "select") {
    return triggerChangeIfValueChanged.call(this, e);
  }
}

function triggerChangeOnKeyDown(e) {                             #H
  var type = e.target.type,
      key = e.keyCode;
  if (key === 13 && e.target.nodeName.toLowerCase() !== "textarea" ||
      key === 32 && (type === "checkbox" || type === "radio") ||
      type === "select-multiple") {
    return triggerChangeIfValueChanged.call(this, e);
  }
}

function triggerChangeOnBefore(e) {                              #I
  getData(e.target)._change_data = getVal(e.target);
}

function getVal(elem) {                                          #J
  var type = elem.type,
      val = elem.value;
  if (type === "radio" || type === "checkbox") {
    val = elem.checked;
  } else if (type === "select-multiple") {
    val = "";
    if (elem.selectedIndex > -1) {
      for (var i = 0; i < elem.options.length; i++) {
        val += "-" + elem.options[i].selected;
      }
    }
  } else if (elem.nodeName.toLowerCase() === "select") {
    val = elem.selectedIndex;
  }
  return val;
}

function triggerChangeIfValueChanged(e) {                        #K
  var elem = e.target, data, val;
  var formElems = /textarea|input|select/i;
  if (!formElems.test(elem.nodeName) || elem.readOnly) {
    return;
  }
  data = getData(elem)._change_data;
```

```
        val = getVal(elem);
        if (e.type !== "focusout" || elem.type !== "radio") {
          getData(elem)._change_data = val;
        }
        if (data === undefined || val === data) {
          return;
        }
        if (data != null || val) {
          return triggerEvent(elem, "change");
        }
      }

    })();

</script>
```

#A Defines special binding function for `change` events.
#B Adds handler normally and bails if browser has adequate support.
#C Piggybacks on other events on first `change` handler binding.
#D Defines special unbinding function for `change` events.
#E Removes hander normally and exits in supporting browsers.
#F Removes piggybacks if last unbinding of `change` handlers.
#G Piggyback handler for `click` events.
#H Piggyback handler for `keydown` events.
#I Piggyback handler for `beforeactivate` events. Stores the value of the element for the upcoming `focusout` event.
#J Utility function that fetches the value of the passed element.
#K Piggyback handler for the `focusout` event. Triggers if the value of the element has changed.

A lot of this code is similar in nature to the approach taken in listing 13.12 so we wont go over it in detail; there's just more of it because there are more event types to handle.

The code specific to this example is mostly found within the `getVal()` and `triggerChangeIfValueChanged()` functions.

The `getVal()` method returns a serialized version of the state of the passed form element. This value will be stored by any `beforeactivate` events in the `_change_data` property within the element's data object for later use.

The `triggerChangeIfValueChanged()` function is responsible for determining if an actual change has occurred between the previously-stored value and the newly-set value and triggering the `change` event if they differ.

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 element ,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 element 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 legacy version IE have fallen by the wayside and this code doesn't need to exist.

IMPLEMENTING FOCUSIN AND FOCUSOUT EVENTS

The `focusin` and `focusout` events are two proprietary events introduced by Internet Explorer that detect when a standard `focus` or `blur` event has occurred on any element, or

descendant of that element. These events actually occur before the focus or blur takes place, making it equivalent to a capturing event rather than a bubbling event.

The reason that these non-standard events are worthy of our consideration 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 browser standards 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, as follows:

```
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 );
}
```

The, we modify the `removeEvent()` function to unbind the events again properly:

```
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 result will be support for the non-standard `focusin` and `focusout` event in all browsers. Naturally we might want to keep our event-specific logic separate from our `addEvent` and `removeEvent` internals. In such a case we 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

There's another set if non-standard, but useful, event types we'd like to consider.

IMPLEMENTING MOUSEENTER AND MOUSELEAVE EVENTS

The `mouseenter` and `mouseleave` events are two more custom events introduced by Internet Explorer to simplify the process of determining when the mouse is currently positioned within or outside an element that are easier to use than `mouseover` and `mouseout`.

Usually we would interact with the standard `mouseover` and `mouseout` events provided by the browser but frequently they don't really provide what we're looking for. The problem is that they fire the event when you move between child elements in addition to the element parent element itself. This is typical of the event bubbling model but it is frequently a

problem when implementing things like menus and other interaction elements, when all we care about is if we're still within an element and don't want to be told we've left it just because we've entered a child element..

This situation is where the `mouseenter` and `mouseleave` events come in quite handy. They will only fire on the main element upon which we have bound them and only tell us we've left if the mouse cursor actually leaves the parent element. As Internet Explorer is the only browser that currently implements these useful events we need to simulate the full event interaction for other browsers as well.

Listing 13.14 shows the implementation of a function named `hover()` that adds support for the `mouseenter` and `mouseleave` events to all browsers.

Listing 13.14:Addind support for `mouseenter` and `mouseleave` to all browsers.

```
<script>
  (function() {

    if (isEventSupported("mouseenter")) {                    #A

      this.hover = function (elem, fn) {                     #B
        addEvent(elem, "mouseenter", function () {
          fn.call(elem, "mouseenter");
        });

        addEvent(elem, "mouseleave", function () {
          fn.call(elem, "mouseleave");
        });
      };

    }
    else {

      this.hover = function (elem, fn) {                     #C
        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) {      #D

      var parent = event.relatedTarget;                      #E

      while (parent && parent != elem) {                     #F
        try {
          parent = parent.parentNode;
        }
        catch (e) {                                          #G
          break;
```

```
          }
        }
        if (parent != elem) {                                    #H
          handle.call(elem, type);
        }
      }

  })();
</script
```

#A Tests if the browser natively supports the `mouseenter` (and hence, `mouseleave`) events.
#B Adds handlers that simply invoke the handler for browsers that support the events.
#C In non-supporting browsers, handle `mouseover` and `mouseout` using a handler that detects whether the handler should fire or not.
#D Internal handler that fires the original handler to mimic the non-standard behavior.
#E Gets the element we are entering from, or exiting to.
#F Traverses upward until we hit the top of the DOM or the hovered element.
#G In case of error assume we're done (can happen with Firefox XUL elements).
#H If not exiting or entering the hovered element, trigger the handler.

Most of the smarts for handling the `mouseenter` and `mouseleave` events lies inside of the `withinElement()` function, which we establish as the handler for `mouseover` and `mouseout` events in browsers not supporting the non-standard events. This function checks the `relatedTarget` of the event, which will be the element being entered for `mouseout` events, and the element being left for `mouseover` events. In either case, if the related element is within the hovered element, we ignore it. Otherwise, we know that it's the hovered element that's being left or entered and we trigger the handler.

Speaking of leaving, before we exit this chapter on events, there's one more event that's mighty handy to have around. Let's find out what it is.

13.6 The document ready event

The final event that we'll consider is what's called the "ready" event, which is implemented as `DOMContentLoaded` in W3C DOM-capable browsers.

This ready event fires as soon as the entire DOM document has been loaded, meaning that is ready 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. It executes before the page is displayed and without waiting for other resource to load that can delay the firing of the `load` event.

Doing this in a cross-browser fashion is once again complicated by the need to support legacy versions of IE (those prior to IE9).

The W3C browsers make it easy by triggering a `DOMContentLoaded` event when the DOM is ready. But for legacy IE we need to rely on a multi-pronged attack to try and get notified as soon as possible when the DOM is ready.

One of these techniques will use a trick developed by Diego Perini and described at http://javascript.nwbox.com/IEContentLoaded/, in which we attempt to scroll the document to the extreme left (its natural position). This attempt will fail until the document is loaded,

so if continually try to perform the operation (using a timer to make sure we don't block the event loop) we'll know when the DOM is ready when the operation stops failing.

A second prong of our attack on legacy IE is listebing for 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 afterwars (but always before the final window `load` event). Even so, it serves as a good backup for IE, making sure that the at least *something* will fire before the window `load` event.

The third prong is examining the `document.readyState` property. This property, available in all browsers, records how fully-loaded the DOM document is at that point. We want to know when it reached "complete" status. But note that long delays in loading, especially in Internet Explorer, may cause the `readyState` to report "complete" too early, which is why we aren't solely relying upon it. But checking this property on load can help to avoid unnecessary event binding if the DOM is already in a ready-to-use state.

An implementation of the ready event, using the above techniques, can be found in Listing 13.15.

Listing 13.15: Implement a cross-browser DOM ready event

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

  (function () {

    var isReady = false,                                    #A
        contentLoadedHandler;

    function ready() {                                      #B
      if (!isReady) {
        triggerEvent(document,"ready");
        isReady = true;
      }
    }

    if (document.readyState === "complete") {               #C
      ready();
    }

    if (document.addEventListener) {                        #D
      contentLoadedHandler = function () {
        document.removeEventListener(
            "DOMContentLoaded", contentLoadedHandler, false);
        ready();
      };

      document.addEventListener(                            #E
          "DOMContentLoaded", contentLoadedHandler, false);

    }

    else if (document.attachEvent) {                        #F
```

```
    contentLoadedHandler = function () {
      if (document.readyState === "complete") {
        document.detachEvent(
            "onreadystatechange", contentLoadedHandler);
        ready();
      }
    };

    document.attachEvent(                                    #G
        "onreadystatechange", contentLoadedHandler);

    var toplevel = false;
    try {
      toplevel = window.frameElement == null;
    }
    catch (e) {
    }

    if (document.documentElement.doScroll && toplevel) {     #H
      doScrollCheck();
    }
  }

  function doScrollCheck() {                                 #I
    if (isReady) return;
    try {
      document.documentElement.doScroll("left");
    }
    catch (error) {
      setTimeout(doScrollCheck, 1);
      return;
    }
    ready();
  }
})();

</script>
```

#A Starts off assuming that we're not ready.
#B Defines a function that triggers the ready handler only once; subsequent calls will do nothing.
#C If the DOM is already ready by the time we get here, just fire the handler.
#D For W3C browsers, creates a handler for the `DOMContentLoaded` event that fires off the ready handler and removes itself.
#E Establishes the just-created handler for the `DOMContentLoaded` event.
#F For IE Event Model, create a handler that removes itself and fires the ready handler if the document `readyState` is complete.
#G Establish the previous handler for the `onreadystatechange` event. Will likely fire late, but is iframe-safe.
#H If not in an iframe, perform the scroll check.
#I Defines the scroll check function, which keeps trying to scroll until success.

With a complete ready event implementation we now have all the tools in place for a complete DOM event handling system. High time to treat ourselves to a lovely beverage.

13.7 Summary

In the chapter we've more than seen that a complete DOM event handling system is anything but simple. The Internet Explorer Model in legacy versions of IE, which likely need to supported for quite a few years to come, causes a great deal of mayhem that we need to circumvent. But it's not all IE's fault, even the W3C browsers lack extensibility in the native API meaning that we still have to circumvent, and improve upon, most of the event system in order to arrive at a solution that is universally applicable.

What we learned and did in this chapter:

- There are three event handling models in the browsers that we're likely required to support:
    - DOM Level 0, which is probably the most familiar, but is unsuitable for robust event management.
    - DOM Level 2, which is the W3C standard but lacks many features that we need to create a full management suite.
    - The IE Model, which is proprietary, has fewer features than DOM Level 2, but is what we must use in legacy versions of IE.
- One of the problems with the IE Model is the lack of proper context in the handlers. We developed a handful of event binding and unbinding functions to normalize this.
- Another issue was the difference in the event information between DOM Level 2 and the IE Model, so we developed a function that "repairs" event instances to be consistent across platforms.
- We needed a means to store information regarding individual elements without resorting to global storage, so we developed a means to tack data onto elements. While we ended up using this to store event handling information, it's a general facility that could be used for many purposes.
- Using the data storage facility, we enhanced our event binding and unbinding routines to use it to keep track of handlers established for all event types for any element.
- One of the more important features that we added to our event management suite was the ability to trigger events under script control. Useful in its own right, we found that it enabled a bunch of really useful capabilities.
- One of those capabilities is the ability to create and trigger custom events. This allowed us to bring loose coupling into almost anything we want to do within a page; something that makes creating independent modular components a breeze.
- We also learned how delegating event handling to ancestors of target object can be an efficient and elegant means to minimize the amount of code we need to create and establish.
- Focusing on browser deficiencies we then developed ways to:

  - Cause `submit` events to bubble like other events
  - Cause `change` events to bubble like other events
  - Implement `focusin` and `focusout` events in all browsers
  - Implement `mouseenter` and `mouseleave` events in all browsers
- And finally, we developed a document-ready handler that fires across all browser to let us know when the DOM is ready to be manipulated in advance of the browser `load` event.

All told we now have the knowledge necessary to implement a complete and useful DOM event management system that is capable of tackling even the greatest challenge presented to us by the browsers.

14

Manipulating the DOM

In this chapter:

- Injecting HTML into a page
- Cloning elements
- Removing elements
- Manipulating element text

If we were to open up a JavaScript library we'd certainly notice (most likely with a healthy portion of surprise) the length and complexity of the code behind simple DOM operations.

Even presumably simple code, like cloning or removing a node (which both have simple DOM counterparts like `cloneNode()` and `removeChild()`) have relatively complex implementations.

This begs two questions: why is this code so complex, and why do I need to understand how it works if the library will just take care of it for me?

The most compelling reason is *performance*. Understanding how DOM modification works in libraries can allow you to write better and faster code that uses the library or, alternately, extract those techniques from the library and use them in your own code.

There are two points which will likely be surprising to most people that are using a library: not only do libraries handle more cross-browser inconsistencies than typical hand-written code, but they frequently run faster as well. The reason for the performance improvement isn't all that surprising as the library developers keep on top of the latest browser additions. Libraries are thus using the absolute best-possible techniques for creating the most performant code.

For example, when injecting HTML fragments into a page, libraries are using *document fragments* or `createContextualFragment()` to inject HTML in a way that is even faster than what would likely happen normally. Both of these techniques are not commonly used in

everyday development, and yet they both allow you to insert elements into a page in ways that are even faster than most better known methods (such as `createElement()`).

Another area of performance improvement is in the area of memory management. It's a relatively safe argument to say that most developers never think of the memory usage of their web applications. This is not the case for a JavaScript library; it must take into account memory usage and make sure that duplicate resources aren't recreated needlessly.

Looking at the examples provided in this chapter will reveal many of the techniques that you can use to help reduce memory consumption in your own applications.

This chapter will talk about all those nasty cross-browser issues prevalent in DOM modification code but also the areas in which extra performance has been squeezed out, allowing for your code to run even faster. Understanding how those performance improvements have been made will allow you to write web applications that run even faster than what you'd normally be able to create.

Here are some resources for further reading that you might enjoy:

- This is the new hotness, although it's not in jQuery yet: https://developer.mozilla.org/en/DOM/range.createContextualFragment
- An implementation that's worthy of citing: https://github.com/tomhuda/metamorph.js/blob/master/lib/metamorph.js

Enough talk, let's push up our sleeves and dive into manipulating the DOM.

14.1 Injecting HTML into the DOM

In this chapter we'll start by looking at an efficient way to insert HTML given that HTML in string form into a document at any location. We're looking at this particular technique as it's frequently used in a few ways:

- Injecting arbitrary HTML into a page, manipulating and inserting client-side templates.
- Retrieving and injecting HTML sent from a server.

It's somewhat technically challenging to implement this functionality correctly, especially when compared to building an object-oriented style DOM construction API (which are certainly easier to implement but require an extra layer of abstraction than injecting the HTML).

Be aware that there already exists an API for injecting arbitrary HTML strings that was introduced by Internet Explorer, and now part of the W3C HTML 5 specification. It's a method that exists on all HTML DOM elements and is named `insertAdjacentHTML()`. See http://www.w3.org/TR/html5/apis-in-html-documents.html#insertadjacenthtml.

This method is fairly straightforward to use and documentation on it can be found here: https://developer.mozilla.org/en/DOM/element.insertAdjacentHTML.

The problem is that we cannot rely on this API across the entire suite of browsers that we're likely to support. Even though this method is broadly available in all modern browsers, it's a recent addition to most, and it's likely that the legacy browsers in your support matrix

include browsers without support for this method. Even IE's implementation in its older versions was incredibly buggy, only working on a subset of all available elements.

And even if we had the luxury of supporting only the latest and greatest versions of the browsers, knowing how to do this sort of DOM manipulation is a skill that a JavaScript ninja should have tucked into his belt.

For these reasons we're going to have to implement a clean DOM-manipulation API from scratch.

The implementation is broken down into a number of steps:

1. Converting an arbitrary but valid HTML/XHTML string into a DOM structure.
2. Injecting that DOM structure into any location in the DOM as efficiently as possible.
3. Executing any inline scripts that were in the source string.

All together, these three steps will provide a page author with a smart API for injecting HTML into a document.

Let's get started.

14.1.1 Converting HTML to DOM

Converting an HTML string to a DOM structure doesn't have a whole lot of magic to it. In fact, it uses a tool that you're most likely already very familiar with: the `innerHTML` property of DOM elements.

Using it is a multi-step process:

- Make sure that the HTML string contains valid HTML/XHTML (or, to be friendly, tweak it so that it's closer to valid).
- Wrap the string in any enclosing markup required by browser rules.
- Insert the HTML string, using `innerHTML`, into a dummy DOM element.
- Extract the DOM nodes back out.

The steps aren't overly complex - save for the actual insertion, which has some gotchas – so let's take a look at each one.

PRE-PROCESSING THE XML/HTML SOURCE STRING

To start, we'll need to clean up the HTML to meet our needs. 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 self-closing elements such as "`<table/>`".

The above self-closing style of elements, in actuality, only works for a small subset of HTML elements; attempting to use that syntax otherwise is likely 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) as shown in listing 14.1.

Listing 14.1: Making sure that self-closing elements are interpreted correctly

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

  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.");

</script>
```

With that accomplished, we need to determine whether the new elements need to be wrapped or not.

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 certain container elements before they can be injected. For example an `<option>` element must be within a `<select>`.

There are two approaches to solving this problem, both ot which require constructing a mapping 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 required markup and then injected directly into any container element (such as a `<div>`).

The second technique is preferred; it involves very little browser-specific code in contrast with the first approach, which would require a large amount of mostly browser-specific code.

The set of problematic elements that need to be wrapped in specific container elements is fortunately a rather manageable seven, as follows (where … indicates where the elements need to be injected):

1. `<option>` and `<optgroup>` need to be contained in a `<select multiple="multiple">...</select>`.
2. `<legend>` needs to be contained in a `<fieldset>...</fieldset>`.
3. `<thead>`, `<tbody>`, `<tfoot>`, `<colgroup>`, and `<caption>` need to be contained in a `<table>...</table>`.

4. `<tr>` needs to be in a `<table><thead>...</thead></table>`, `<table><tbody>...</tbody></table>`, or a `<table><tfoot>...</tfoot></table>`.
5. `<td>` and `<th>` need to be in a `<table><tbody><tr>...</tr></tbody></table>`.
6. `<col>` must be in a `<table><tbody></tbody><colgroup>...</colgroup></table>`.
7. `<link>` and `<script>` need to be in a `<div></div><div>...</div>`.

Nearly all of the above are straightforward save for the following which require a bit of explanation:

- A `<select>` element with the `multiple` attribute is used (as opposed to a non-multiple 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.

With the elements properly wrapped let's start generating.

GENERATING THE DOM

Using the map of containers from the previous section, we now have enough information to generate the HTML that we need to insert in to a DOM element. See listing 14.2.

Listing 14.2: Generate a list of DOM nodes from some markup.

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

  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.");

</script>
```

There are two browser bugs that we'll need to work around before we return our node set, both 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). The 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.

14.1.2 Inserting into the document

Once we have the actual DOM nodes it's time to insert them into the document. There are a couple steps the need to take place that are, luckily, not particularly tricky.

As we have an array of elements that we need to insert, potentially into any number of locations into the document, we'll need to try and cut down on the number of operations that occur.

We can do this by using ***DOM fragments***. DOM fragments are part of the W3C DOM specification, and are supported in all browsers. This useful facility gives us a container that we can use to hold a collection of DOM nodes.

This in itself is quite useful, but it also has the advantage that 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 example of listing 14.3, 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 14.3: Inserting a DOM fragment into multiple locations in the DOM

```
<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.

There's one final point that we'll need to take care of; albeit a relatively minor one. When page authors 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 as outlined in listing 14.4.

Listing 14.4: Figure out the actual insertion point of an element

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

  function root(elem, cur) {
    return elem.nodeName.toLowerCase() === "table" &&
        cur.nodeName.toLowerCase() === "tr" ?
        (elem.getElementsByTagName("tbody")[0] ||
            elem.appendChild(elem.ownerDocument.createElement("tbody"))) :
        elem;
  }

</script>
```

Altogether we now have a way to both generate and insert arbitrary DOM elements in an intuitive manner.

But what about any scripting elements that are embedded in the source string?

14.1.3 Script execution

In addition to insertion of structural HTML into a document, a common requirement is the execution of inline script elements. This scenario is quite common when an HTML fragment is returned as an Ajax response 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, we could use something like the code shown in listing 14.5 from jQuery.

Listing 14.5: Collecting the scripts

```
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 code of this listing deals 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, the code takes care to only remove scripts that are normally executed as JavaScript (those without an explicit `type` or those with a `type` of 'text/javascript').

Then, after the DOM structure is inserted into the document, takes the contents of `scripts` and evaluates it. It's more about shuffling things around than intricate code, but it does lead us to a tricky part.

GLOBAL CODE EVALUATION

When inline scripts are included for execution, it's expected that they will be evaluated within the global context. This means that if a variable is defined, it should become a global variable; same with any functions, et al.

The built-in methods for code evaluation are spotty, at best. The one foolproof 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 is a technique that we discussed back in section 9.1.

This will cause the browser to execute the inner contents of the script element within the global scope.

Listing 14.6 shows a part of the global evaluation code that's in jQuery.

Listing 14.6: Evaluate a script within the global scope

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

  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);
    }
  }

</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. See listing 14.7.

Listing 14.7: A method for evaluating a script (even if it's remotely located)

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

  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);
  }

</script>
```

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 doubly executed (appending a script to a document, which ends up recursively calling itself, for example).

To our ninja toolkit we've added the ability to add new elements to the DOM. Now let's see how we can copy new elements from previously existing ones.

14.2 Cloning elements

Cloning an element (using the DOM `cloneNode` method) is straightforward in all browsers, except legacy Internet Explorer. Legacy versions of Internet Explorer have toubling behaviors that, when they occur in conjunction, result in a very frustrating scenario for handling cloning.

Firstly, when cloning an element, IE 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, shown in listing 14.8, determines if this is the case.

Listing 14.8: Determining if a browser copies event handlers on clone.

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

  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.

Finally, the solution to all of this is to just clone the element, inject it into another element, 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 (sigh), there's another IE bug: the `innerHTML` (and `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 seen is when the `name` attribute of an input element is changed dynamically. The new value isn't represented in `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. See the code of listing 14.9.

Listing 14.9: A portion of the element clone code from jQuery

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

  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;
  }

</script>
```

Note that the above code uses jQuery's `jQuery.clean` method, which converts an HTML string into a DOM structure (as was discussed previously).

OK, we've added new elements, and copied elements. How do we get rid of them?

14.3 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 are usually two steps of cleanup that need to occur on an DOM element before it can be removed from the DOM.

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 is exactly how we set up our event management framework in chapter 13.

This step is very 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. Just as we discussed in chapter 13, a framework needs a good way to associate pieces of

data with an element *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, as well as all descendant elements since all the descendant elements are also being removed.

For example, listing 14.10 shows the relevant code from jQuery:

Listing 14.10: The remove element function from jQuery

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

  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);
  }

</script>
```

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 (with the exception of Internet Explorer as described above).

In IE, 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 gives us 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 we execute the following:

```
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 as shown in listing 14.11.

Listing 14.11: 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 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 worry so much about memory issues later on.

That covers HMTL elements pretty well, but a page consists of more than just elements. We also need to consider page text.

14.4 Text contents

Working with text tends to be much easier than working with HTML elements, especially as there are built-in methods that work in all browsers for text content. But as per usual, there are all sorts of browser bugs that we end up having to work around, making these APIs not the complete solution we'd like to hope they might be.

When it comes to dealing with text, there are typically two common scenarios:

- Getting the text contents out of an element.
- Setting the text contents of an element.

W3C-compliant browsers conveniently provide a `textContent` property on their DOM elements. Accessing the contents of this property gives you the textual contents of the element, including its direct children and descendant nodes, as well.

Legacy Internet Explorer has its own property: `innerText`, for performing the exact-same behavior as `textContent`. (Just to be nice, some browsers such as WebKit based browsers also support innerText.)

Consider the code of listing 14.12.

Listing 14.12: Using textContent and innerText

```
<div id="test"><b>Hello</b>, I'm a ninja!</div>
<div id="test2"></div>

<script type="text/javascript">

  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 we discussed while removing elements from the page, not having any sort of special consideration for element memory leaks will come back to bite us later on. 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 end-lines) feel free to use `textContent`/`innerText` for accessing the element's text value. For setting though, we'll need to devise an alternative solution.

14.4.1 Setting text

Setting a text value involves two parts:

- Emptying out the contents of the element
- Inserting the new text contents in its place

Emptying out the contents is straightforward; we've already devised a solution shown in listing 14.10.

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 need to have any problematic HTML-specific characters escaped. For example < must appear as the HTML entityt `<`.

Luckily, we can actually use the built-in `createTextNode()` method, available on DOM documents, to do precisely that as shown in listing 14.13.

Listing 14.13: 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>
```

We've *set;* now let's *get*.

14.4.2 Getting text

To get the *accurate* text value of an element we have to ignore the results from `textContent` and `innerText`. The most common problem with these properties is related to end-lines being unnecessarily stripped from the returned result. Instead we must collect all the text node values manually to get an accurate result.

A possible solution would look like the code in listing 14.14, making good use of recursion.

Listing 14.14: Getting the text contents of an element

```
<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 our applications when we can get away with not worrying about whitespace, we should definitely stick with the `textContent`/`innerText` properties as they make our lives so much simpler. But it's nice to have a fallback for when those properties don't cut the mustard.

14.5 Summary

We've taken a comprehensive look at the best ways to tackle the difficult problems surrounding DOM manipulation. While modern browsers give us some newer options for DOM manipulation, knowing how to do "by hand" is important in order to provide support fo legacy browsers and for performance considerations.

These problems should be easier to overcome than they are, but cross-browser issues make their actual implementations much more difficult than they should be. 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.

What we learned in this chapter includes:

- Using regular expressions, a handy tool we mastered in chapter 7, we can cajole HTML snippets into a well-formed syntax that we can parse.
- Injecting a fragment of HTML text into a temporary element's `innerHTML` property is a quick and easy way to convert a string of HTML text into DOM elements.
- Some elements, such as table component elements, need to be wrapped with certain other container elements in order for them to be properly created.
- Script elements in HTML fragments can be executed in the global scope using the techniques that we examined in chapter 9 on code evaluation.
- Legacy versions of Internet Explorer cause headaches when cloning nodes because they actually copy too *much*; including event handlers and expandos.
- We need to be mindful of memory management needs when removing elements from the DOM, especially when creating long-lived pages.

In this chapter, we've created, cloned and removed elements. What about finding them? Let's consider the final subject for your ninja training: locating elements via CSS selectors.

15
CSS selector engines

In this chapter:

- The current status of browser selector support
- Strategies for selector engine construction
- Using the W3C API
- Some info on XPath
- Building a DOM selector engine

The good news is that we, as web development professionals, are well into the age in which the W3C Selectors API existing in all modern browsers. This API provides us with the `querySelectorAll()` and `matchesSelector()` methods, along with other goodies that we can use in our applications to write very-fast DOM traversals in ways that are relatively cross-browser. So you might ask: as the W3C Selectors API has been implemented in virtually all modern browsers, why do we really need to spend time discussing how a pure-JavaScript CSS selector engine is implemented?

While the addition of the standard API is a good thing, the implementation of most browsers' W3C Selector API (at least in their current state) is rather a shoehorning of their existing internal CSS selector engines into the standardized JavaScript/DOM realm. To make this happen, a number of niceties that one would typically associate with a good API were thrown aside. As examples, the methods don't make use of already-constructed DOM caches, don't provide good error reporting, and are unable to handle any form of extensibility.

The CSS selector engines that are in popular JavaScript libraries take into account all of these factors. They use DOM caches to provide even faster performance, they provide extra levels of error reporting, and are highly extensible.

All this being said, the question still remains: why should you understand how a pure-JavaScript CSS selector engine works? As was the case with investigating DOM modification,

understanding how a pure-JavaScript CSS Selector engine works can yield some rather astonishing performance gains.

Not only will we be able to write better traversal implementations, allowing us to search through a DOM tree even faster, but we also learn to sculpt our CSS selectors to better adapt to how the CSS selector engines work, giving us even more performant selectors.

CSS selector engines are a part of everyday development in this day and age, and understanding how they work, along with how to make them work even faster, will give us a fundamental leg-up in your development.

After all, if you think about the types of things that we need to do in on-page script, a lot of it follows the pattern of:

- Find DOM elements
- Do something to them

Finding Dom elements has never been a strong point of JavaScript, so anything we can do to make that step easier let's us focus on the more interesting "do something" part.

It's standard, at this point in time, for selector engines to implement CSS 3 selectors, as defined by the W3C as shown at http://www.w3.org/TR/css3-selectors/.

With regard to approach, there are three primary ways of implementing a CSS selector engine:

1. Using the previously mentioned W3C Selectors API as implemented in most modern browsers.
2. XPath, a DOM querying language built into a variety of modern browsers.
3. Pure DOM, a staple of CSS selector engines, allowing for graceful degradation if either of the first two mechanisms don't exist.

This chapter will explore each of these strategies in depth allowing us to make some educated decisions about implementing, or at least understanding, a JavaScript CSS selector engine.

We'll start with the W3C approach.

15.1 The W3C 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 modern browsers (starting in Safari 3, Firefox 3.1, Internet Explorer 8, Chrome 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.

This API provides a number of useful methods, two of which are implemented in modern browsers:

- `querySelector()`, which accepts a CSS selector string and returns the first element found, or `null` if no matching element is found.
- `querySelectorAll()`, which accepts a CSS selector string and returns a static NodeList of all elements found by the selector.

These two methods exist on all DOM elements, DOM documents, and DOM fragments. Listing 15.1 has a couple of examples of how it could be used.

Listing 15.1: 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 W3C Selectors API is that it is limited to supporting CSS supported by the browser rather than the wider-ranging implementations that were first created by JavaScript libraries. This can be seen in the matching rules of element-rooted queries, as seen in Listing 15.2.

Listing 15.2: 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 either of `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.

As 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, as in Listing 15.3.

Listing 15.3: 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 of 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` block will always execute after both of them are done executing (but before the value is returned from the function).

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 after the supporting browsers gain a dominant market share and support the totality (or at least majority of) CSS3.

Let's now turn our attention to a more XML-centric way of approaching the issue.

15.2 Using XPath to find elements

A unified alternative to using the Selectors API (for browsers that don't support it) is the use of XPath querying.

XPath is a querying language utilized for finding nodes in a DOM document. It is significantly more powerful than traditional CSS selectors. Most popular browsers (Firefox, Safari 3+, Opera 9+, Chrome) provide some implementation of XPath that can be used against HTML-based DOM documents. Internet Explorer 6 and onward 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 it's that they're quite fast for complicated expressions. When implementing a pure-DOM implementation of a selector engine, we are constantly at odds with the ability of a browser to scale all the JavaScript and DOM operations. On the other hand, 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 programmatically, there are a few gives: finding elements by `id` and simple tag-based selectors ("div") will always be faster with pure-DOM code.

If our intended audience is comfortable using XPath expressions (and are happy limiting themselves to the modern browsers that support it) then we can simply utilize the method shown in Listing 15.4 (from the Prototype library) and completely ignore everything else about building a CSS selector engine.

Listing 15.4: A method for executing an XPath expression on an HTML document

```
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;
  }
}
```

But 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 15.1.

Table 15.1: 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 we 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.

And now for the "rolling up our sleeves" approach.

15.3 The pure DOM implementation

At the core of every CSS selector engine exists a pure-DOM implementation. This entails 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 of a CSS Selector Engine for a number of reasons:

1. Internet Explorer 6 and 7. While Internet Explorer 8 and 9 have support for `querySelectorAll()`, the lack of XPath or Selectors API support in 6 and 7 make a DOM implementation necessary.
2. Backwards compatibility. If you want your code to degrade in a graceful manner and support browsers that don't support the Selectors API or XPath (like Safari 2) you'll have to have some form of a DOM implementation.
3. For speed. There are a number of selectors that a pure DOM implementation can simply do faster (such as finding elements by ID).

With that in mind we can take a look at the two possible CSS selector engine implementations: top down and bottom up.

A top down engine works by parsing a CSS selector from left-to-right, matching elements in a document as it goes, working relatively for each additional selector segment. It can be found in most modern JavaScript libraries and is, generally, the preferred means of finding elements on a page.

Let's take a simple example. Consider the markup:

```
<body>

  <div></div>
  <div class="ninja">
    <span>Please </span><a href="/ninja"><span>Click me!</span></a>
  </div>

</body>
```

If we wished to select the `<span>` element containing the text "Click me!", we could do so with:

```
div.ninja a span
```

The top-down approach to applying this selector to the DOM is depicted in figure 15.1.

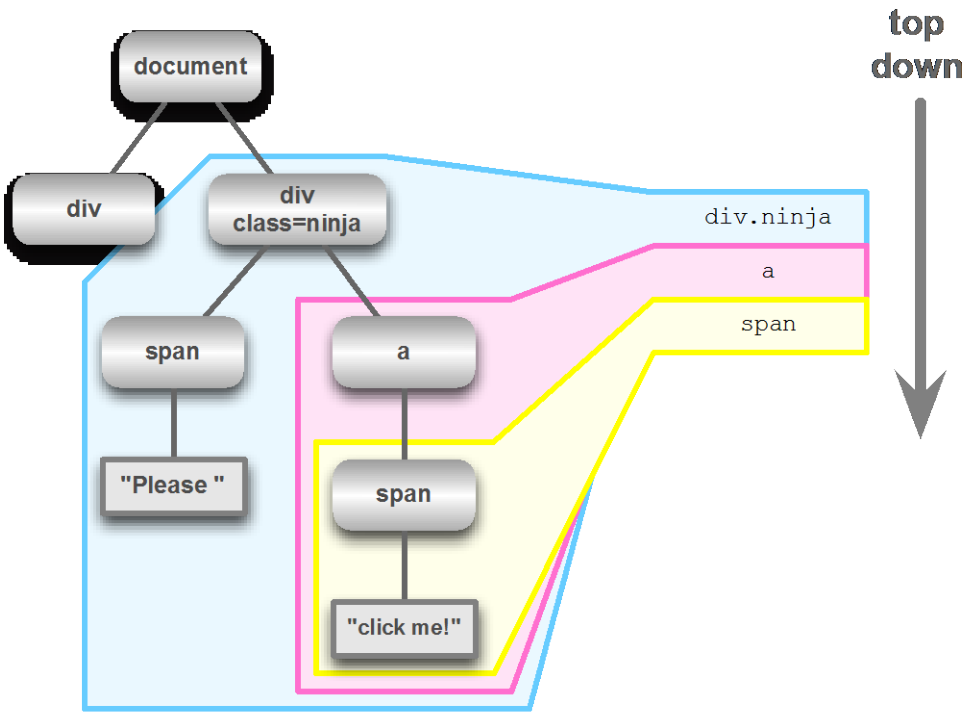

Figure 15.1 Top-down selector engines work from the top of the document, locating sub-trees matching the terms of the selector

The first term, `div.ninja`, identifies a sub-tree within the document. Within that sub0tree, the next term (`a`) is applied, identifying the sub-tree rooted at the anchor element. And finally, the `span` term identifies the target node.

Note that this is a simplified example. Multiple sub-trees can be identified at any stage.

There are two important considerations to take into account when developing a selector engine:

- The results should be in document order (the order in which they've been defined).
- The results should be unique (no duplicate elements returned).

Because of these gotchas, developing a top down engine can be rather tricky.

Let's take a look at a simplified top-down implementation, limited to finding element by their tag names in listing 15.5.

Listing 15.5: 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 implemented a simple top down selector engine 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.

We'll look at each task in turn.

15.3.1 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. But 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 15.6 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 15.6: A regular expression for breaking apart a CSS selector

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

  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");

</script>
```

Obviously this chunking selector is only one piece of the puzzle. we'll need to have additional parsing rules for each type of expression that we 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.

Next, we need to find the elements that match the parsed expression.

15.3.2 Finding the elements

Finding the correct elements on the page is a piece of the puzzle that has many solutions. Which techniques are used depends a lot on which selectors that are being supported, and what is available from the browser.

There are a number of obvious approaches though.

Consider `getElementById()`. Only available on the root node of HTML documents, this method finds the first element on the page that has the specified `id`, thereby useful for the ID CSS selector "#id"). Internet Explorer and Opera infuriatingly will also find the first element on the page that has the same specified `name`. If we only wish to find elements by `id` we will need an extra verification step to make sure to exclude elements selected by this "helpful" feature.

If we 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) we 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 the browsers that support it (namely Internet Explorer, Opera, and Safari).

The `getElementsByTagName()` method performs the obvious operation: finding elements that match a specific tag name. It has another purpose, however: finding all elements within a document or element by 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 element 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 comment nodes are excluded.

`getElementsByName()` is a well-implemented method that serves a single purpose: finding all elements that have a specific `name` (such `<input>` elements that have a `name`). Thus it's really useful for implementing the single selector "[name=*name*]".

The `getElementsByClassName(` method is a relatively new method that's being implemented by browsers (most prominently by Firefox 3, Safari 3 and Chrome) 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 find what we're looking for on a page.

Using the results from these methods it will be possible to move onto filtering.

15.3.3 Filtering the set

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 begin with. For example, we can see that "div" is used, so 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 and other relations.

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, Chrome, 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 and comments. Using these two methods it becomes possible to do all forms of element position filtering.

Constructing a filtering function serves a dual purpose: we can provide it to the user as a simple method for testing their elements, and quickly check to see if an element matches a specific selector.

Let's now focus tools to refine our results.

15.3.4 Recursing and merging

As was shown in Listing 15.1, we can see that selector engines will require the ability to recurse (finding descendant elements) and merge the results together.

However our initial implementation is too simple. Note that we end up receiving two `<span>` elements 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 identifying values to the elements so that we can verify if we've already encountered them, as shown in Listing 15.7.

Listing 15.7: 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.

For a longer discussion on the intricacies of attaching properties to DOM nodes, revisit chapter 13 on Events.

The problem we solved with this approach is a result of the top-down approach we employed. Let's consider an alternative.

15.3.5 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, not surprisingly, 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 top-down approach. 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. We start by finding the last expression in the CSS selector and retrieve the appropriate elements (just like with a top-down engine, but using 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, as shown in Listing 15.8.

Listing 15.8: 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 15.8 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 non-repetitive output, which some may see as an advantage.

15.4 Summary

JavaScript-based CSS selector engines are incredibly powerful tools. They give us the ability to easily locate virtually any DOM element on a page with a trivial amount of selector syntax. 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.

What we learned in this chapter:

- Modern browsers are implementing the W3C APIs for element selection, but they've got a long way to go.
- It still behooves us to create selector engines ourselves if for nothing other than performance.
- To create a selector engine we can:
    - Leverage the W3C APIs
    - Use Xpath
    - Traverse the DOM ourselves for optimum performance
- The top-down approach is most popular, but requires some cleanup operations; for example, to ensure uniqueness of elements.
- A bottom-up approach avoids these operations, but comes wit it's own bag of problems with respect to performance and scalability.

With modern browsers implementing the W3C Selector API, having to worry about the finer points of selector implementation may soon be a thing of the past. For many developers that day cannot come soon enough.