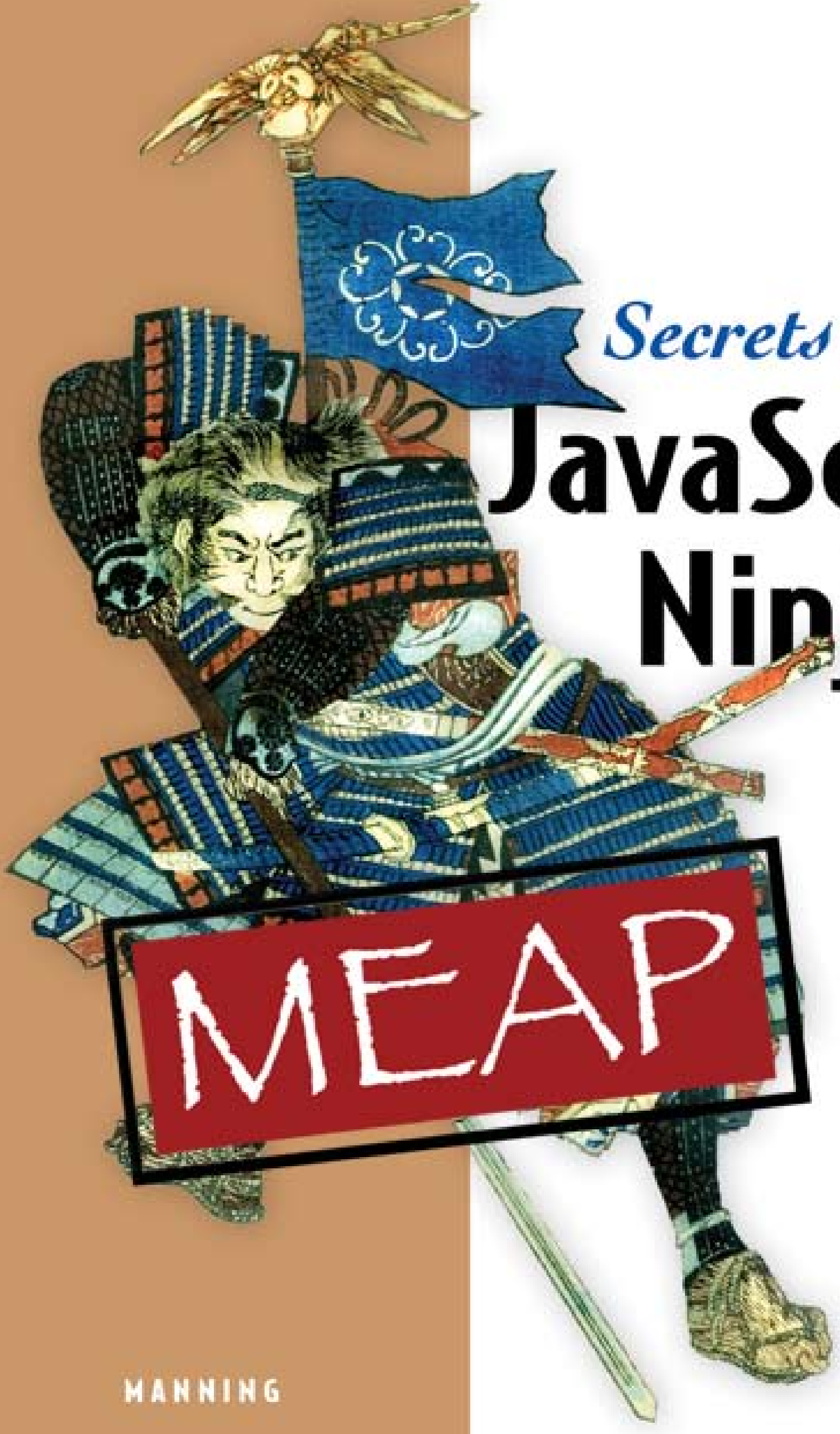


# Secrets of the JavaScript Ninja

MEAP

John Resig
Bear Bibeault

MANNING

**MEAP Edition**
**Manning Early Access Program**
**Secrets of the JavaScript Ninja version 5**

# Table of Contents

# 1 Enter the ninja

In this chapter:

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

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

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

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

## 1.1 Our key JavaScript libraries

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

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

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

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

All of these libraries are well constructed and tackle their target problem areas comprehensively. For these reasons they'll serve as a good basis for further analysis, and understanding the fundamental construction of these code bases gives us insight into the process of world-class JavaScript library construction.

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

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

## 1.2 Understanding the JavaScript Language

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

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

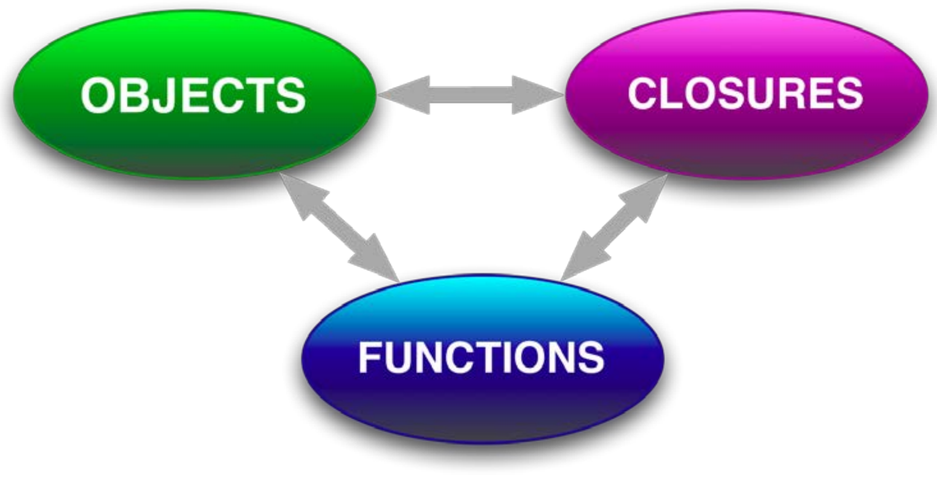


Figure 1.1 JavaScript consists of a close relationship between objects, functions and closures

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

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

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

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

All too often these two important language features are trivialized, misused, and even condemned outright by many JavaScript programmers. But by looking at the work of some of the best JavaScript coders we can see that, when used appropriately, these useful features allow for the creation of some fantastic pieces of code that wouldn't be otherwise possible. To a large degree they can also be used for some interesting meta-programming exercises, molding JavaScript into whatever we want it to be.

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

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

## 1.3 Cross-browser considerations

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

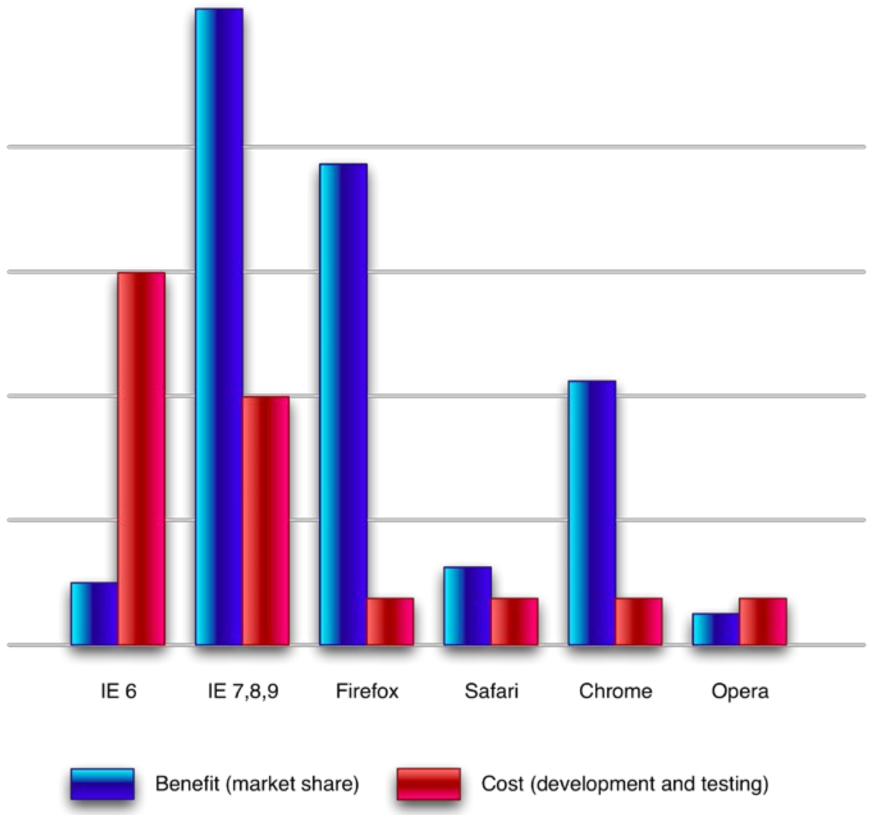


Figure 1.2 Analyzing the cost versus benefit of supporting various browsers tells us where to put our effort

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

- Even though it's relatively a lot more effort to support Internet Explorer 7 and later than the standards-compliant browsers, it's large market share makes the extra effort worthwhile.
- Supporting Firefox and Chrome is a no-brainer since that have large market share and are easy to support.
- Even though Safari has a relatively low market share, it still deserves support, as its standard-compliant nature makes its cost small.
- Opera, though not much more effort than Safari, loses out because of its minuscule

market share.

- Nothing really need be said about IE 6.

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

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

## 1.4 Best practices

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

- Testing
- Performance analysis
- Debugging skills

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

### 1.4.1 Best practice: testing

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

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

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

Consider, for example:

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

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

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

### 1.4.2 Best practice: performance analysis

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

An example of its use would be:

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

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

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

## 1.5 Summary

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

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

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

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

# 2 Testing and debugging

In this chapter:

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

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

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

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

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

Let's get started.

## 2.1 Debugging Code

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

Similar tools have been developed for all major browsers:

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

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

Let's start by looking at logging.

### 2.1.1 Logging

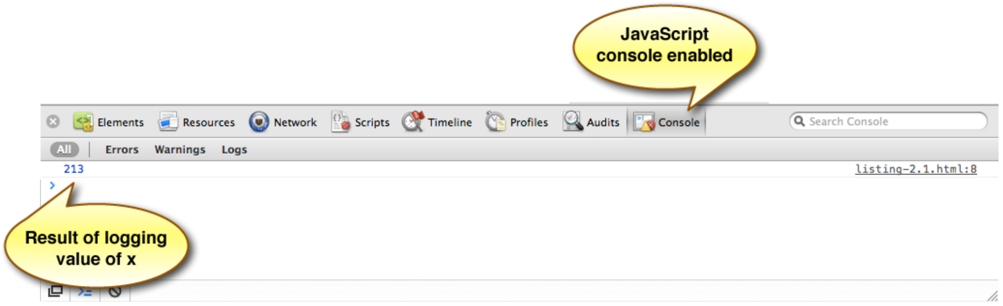
Logging statements (such as using the `console.log()` method in Firebug, Safari, Chrome and IE) are part of the code (even if perhaps temporarily) and useful in a cross-browser sense. We can write logging calls in our code, and we can benefit from seeing the messages in the console of all modern browsers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

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

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

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

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


JavaScript console enabled
Elements Resources Network Scripts Timeline Profiles Audits Console
Search Console
All Errors Warnings Logs
Result of logging value of x

```
213                                                        listing-2.1.html:8
```

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.

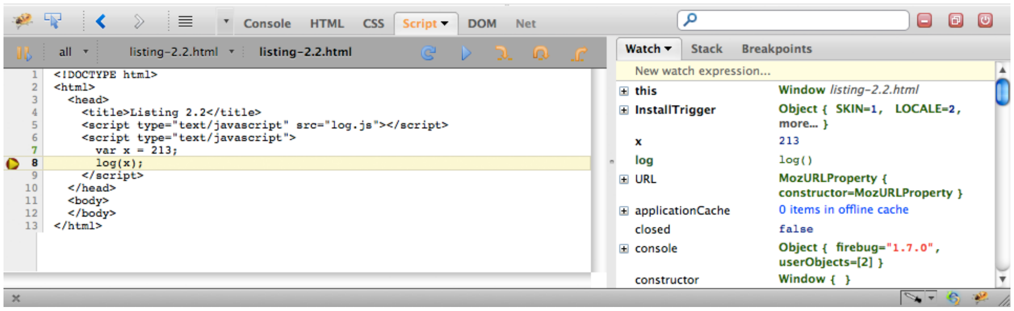
Let's say that we have a page that employs our new `log()` method as shown in listing 2.2.

**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.



```
<!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);
    </script>
  </head>
  <body>
  </body>
</html>
```

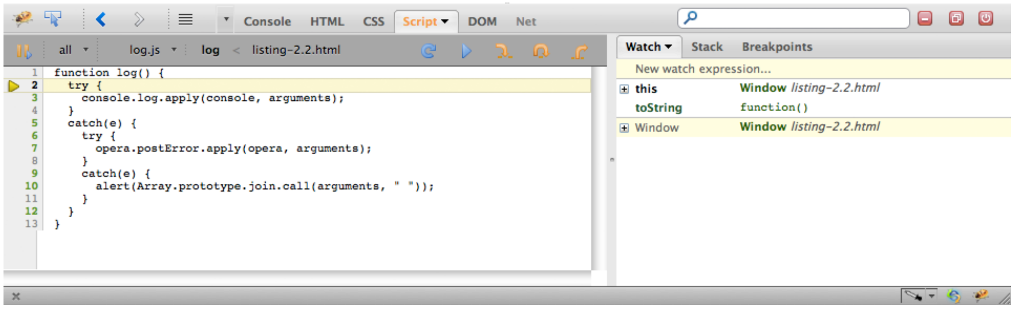
```
this               Window listing-2.2.html
InstallTrigger     Object { SKIN=1, LOCALE=2, more... }
x                  213
log                log()
URL                MozURLProperty { constructor=MozURLProperty }
applicationCache   0 items in offline cache
closed             false
console            Object { firebug="1.7.0", userObjects=[2] }
constructor        Window { }
```

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.



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

| | |
|---|---|
| this | Window *listing-2.2.html* |
| toString | function() |
| Window | Window *listing-2.2.html* |

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**

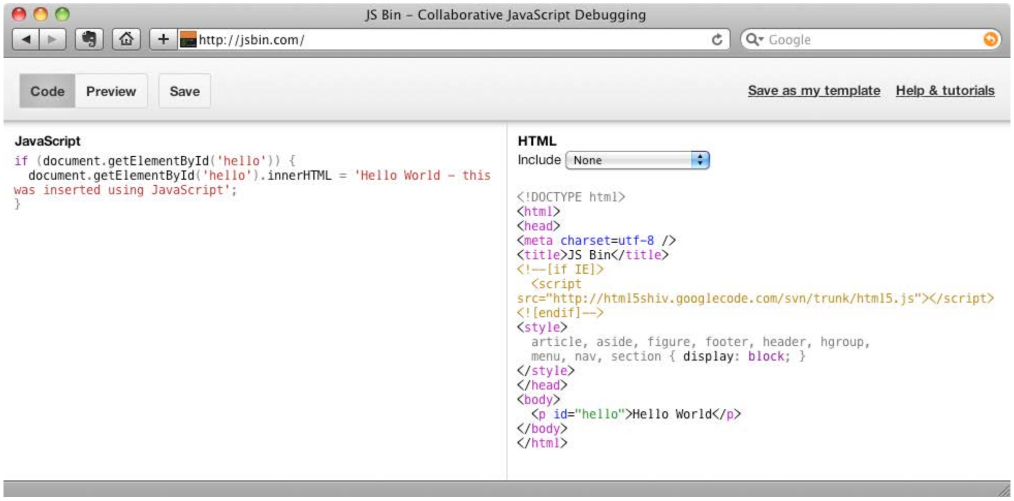
```
#!/bin/sh
# Check out a fresh copy of jQuery
git clone git://github.com/jquery/jquery.git $1
# Copy the dummy test case file in
cp $2.html $1/index.html
# Build a copy of the jQuery test suite
cd $1 && make
```

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.



JavaScript

```
if (document.getElementById('hello')) {
  document.getElementById('hello').innerHTML = 'Hello World - this
was inserted using JavaScript';
}
```

HTML

Include None

```
<!DOCTYPE html>
<html>
<head>
<meta charset=utf-8 />
<title>JS Bin</title>
<!--[if IE]>
  <script
src="http://html5shiv.googlecode.com/svn/trunk/html5.js"></script>
<![endif]-->
<style>
  article, aside, figure, footer, header, hgroup,
  menu, nav, section { display: block; }
</style>
</head>
<body>
  <p id="hello">Hello World</p>
</body>
</html>
```

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

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

## 2.3 Testing frameworks

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

A JavaScript test suite should serve a single need: display the result of the tests, making it easy to determine which tests have passed or failed. Testing frameworks can help us reach that goal without us having to worry about anything but creating the tests and organizing them into suites.

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

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

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

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

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

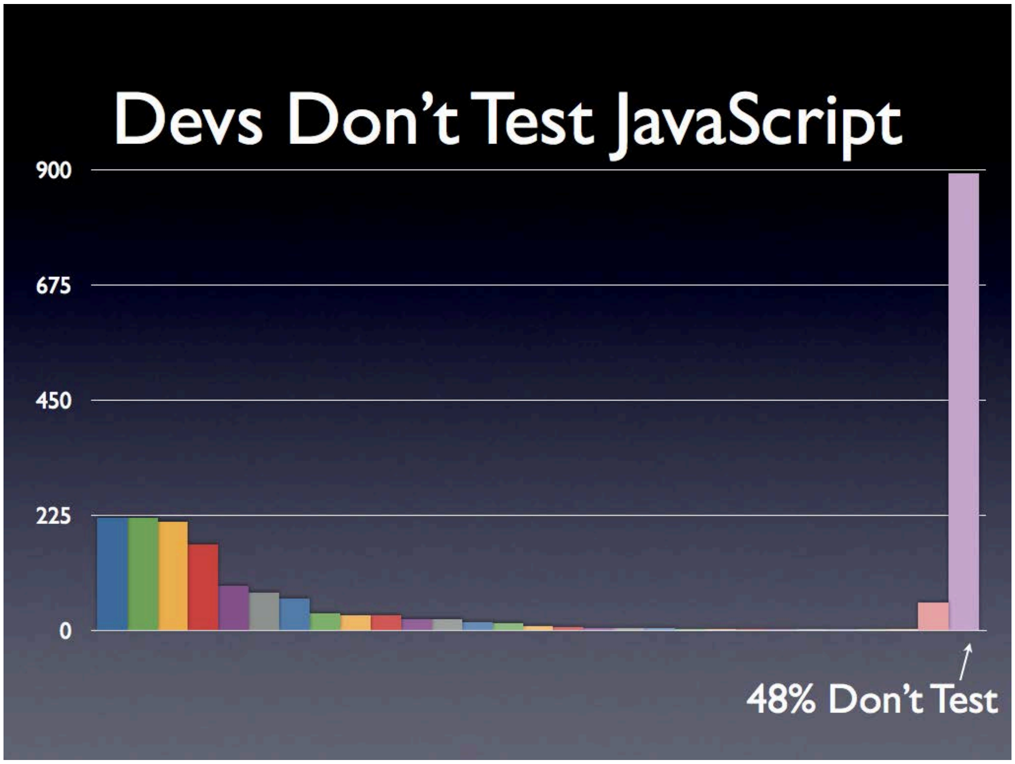


Figure 2.5 A dishearteningly large percentage of script developers don't test at all

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

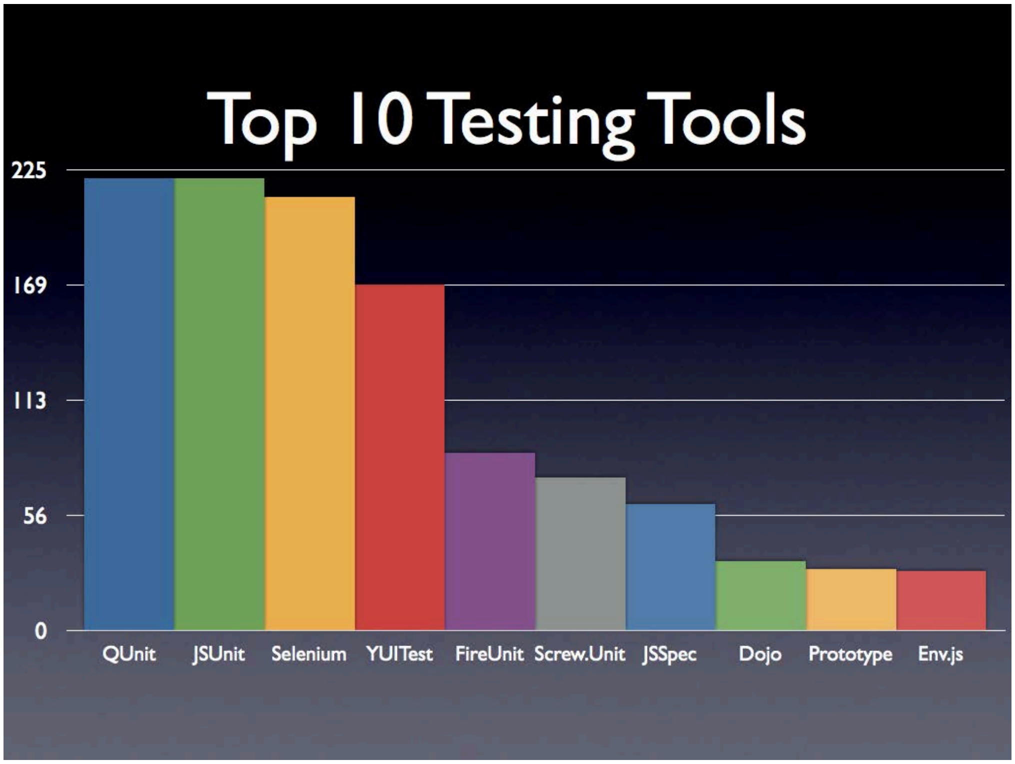


Figure 2.6 Most test-savvy developers favor a small handful of testing tools

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

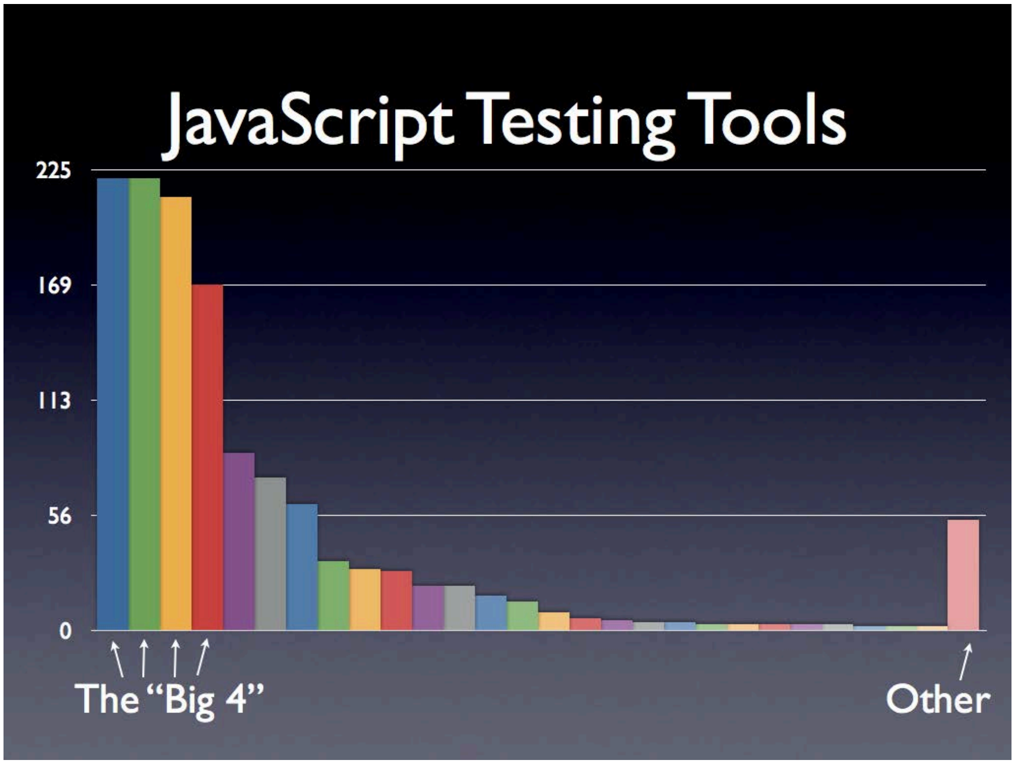


Figure 2.7 The remainder of the testing tools have few users

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

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

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

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

### 2.3.1 QUnit

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

**Distinguishing features:**

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

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

### 2.3.2 YUITest

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

**Distinguishing features:**

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

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

### 2.3.3 JSUnit

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

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

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

## 2.4 The Fundamentals of a Test Suite

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

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

One would have to ask, though: Why would I want to build a new test suite? For most cases it probably isn't necessary to write your own JavaScript test suite, there already exist a number of good-quality suites to choose from (as already shown). It can serve as a good learning experience though, especially when looking at how asynchronous testing works.

### 2.4.1 The assertion

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

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

**Listing 2.8: A simple implementation of a JavaScript assertion**

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

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

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

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

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

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

The function named `assert()` (#1) is almost surprisingly straight-forward. It creates a new `<li>` element containing the description, assigns a class name pass or fail, depending upon the value of the assertion parameter (`value`), and appends the new element to a list element in the document body (#4).

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

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

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

### 2.4.2 Test groups

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

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

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

**Listing 2.9: An implementation of test grouping**

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

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

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

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

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

### 2.4.3 Asynchronous Testing

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

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

1. Assertions that are relying upon the same asynchronous operation will need to be grouped into a unifying test group.
2. Each test group will need to be placed on a queue to be run after all the previous test groups have finished running.
3. Thus, each test group must be capable of run asynchronously.

Let’s look at the example of Listing 2.10.

**Listing 2.10: A simple asynchronous test suite**

```
<html>
  <head>
    <title>Test Suite</title>
    <script>
      (function() {
        var queue = [], paused = false, results;
        this.test = function(name, fn) {
          queue.push(function() {
            results = document.getElementById("results");
            results = assert(true, name).appendChild(
                document.createElement("ul"));
            fn();
          });
          runTest();
        };
        this.pause = function() {
          paused = true;
        };
        this.resume = function() {
          paused = false;
          setTimeout(runTest, 1);
        };
        function runTest() {
          if (!paused && queue.length) {
            queue.shift()();
            if (!paused) {
              resume();
            }
          }
        }

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

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

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

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

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

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

## 2.5 Summary

In this chapter we've looked at some of the basic technique surrounding debugging JavaScript code and constructing simple test cases based upon those results.

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

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

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

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

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

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

# 3
# Functions are fundamental

In this chapter:

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

The quality of all code that you'll ever write in JavaScript relies upon the realization that JavaScript is a functional language.

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

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

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

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

## 3.1 Declaring functions

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

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

**Listing 3.1: Three different ways to declare functions**

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

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

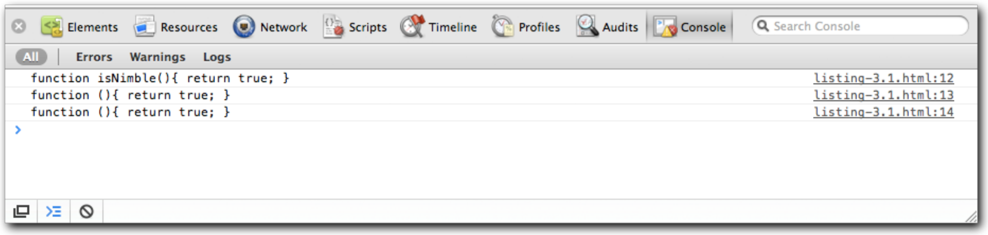
**#1 Declared as named**
**#2 Declared as a variable**
**#3 Declared as a property**
**#4 Emits window properties**

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

**NOTE** If this is the first time that you've seen a function declared in an inline and anonymous fashion as a ***function literal***, then lines (#2) and (#3) may look a bit odd to you. But the syntax is actually rather simple: the keyword `function`, followed by the parameter list for the function (empty in these cases), followed by the function body. In other words, just like a named function declaration but leaving off the name!

This literal can be used as value anywhere that a value is allowed within the language. It's no different, really, than a numeric literal such as 213 for declaring a numeric value, just a lot wordier!

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



```
function isNimble(){ return true; }          listing-3.1.html:12
function (){ return true; }                  listing-3.1.html:13
function (){ return true; }                  listing-3.1.html:14
```

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

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

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

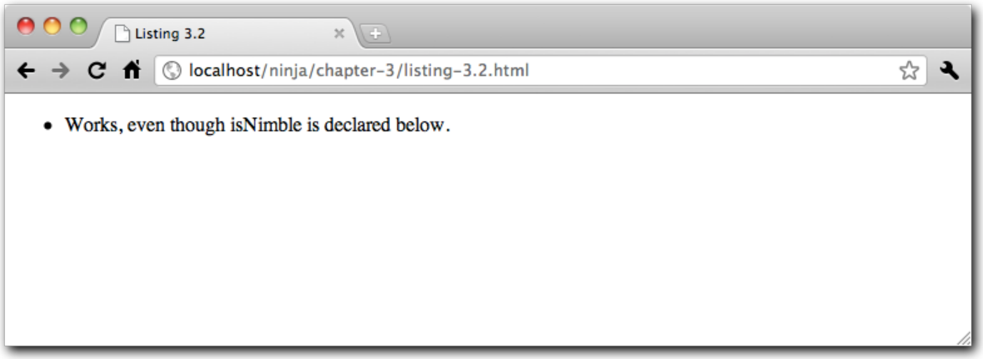
Another difference becomes noticeable when we change when we shift the order of the declarations, as shown in Listing 3.2.

**Listing 3.2: A look at the location of named functions**

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

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

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

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

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



• Works, even though isNimble is declared below.

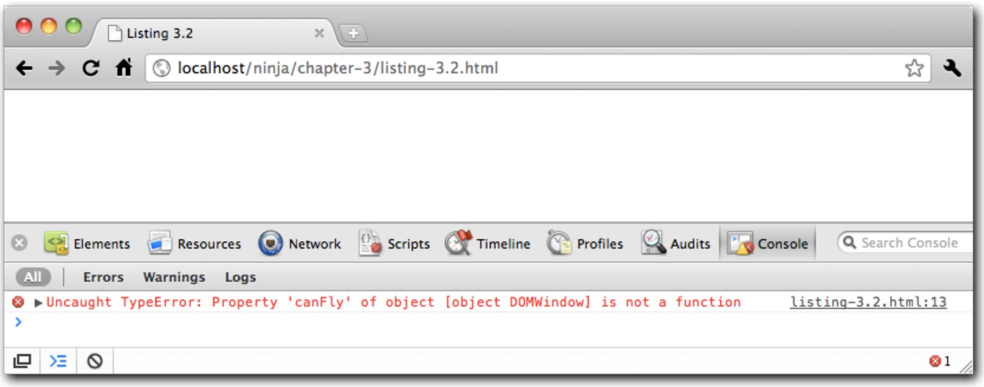
Figure 3.2 The output of the `assert()` proves that named functions can be forward referenced

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

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

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

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



```
Uncaught TypeError: Property 'canFly' of object [object DOMWindow] is not a function    listing-3.2.html:13
```

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

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

**Listing 3.3: Declaring a function below a return statement**

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

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

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

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

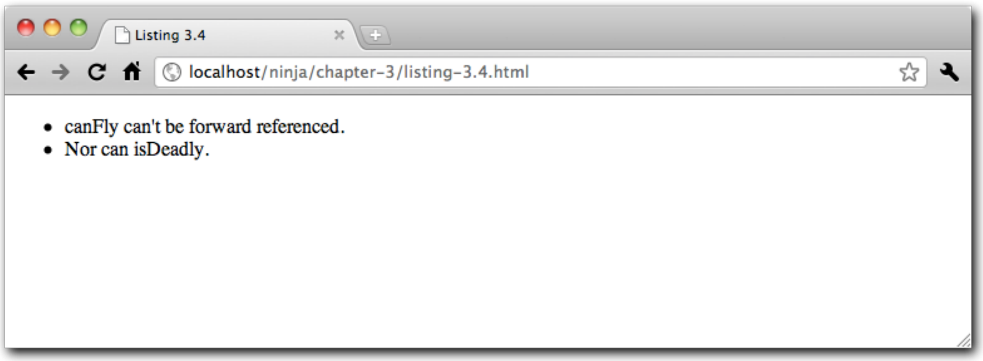
**Listing 3.4: Types of function definition that can't be placed anywhere.**

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

```
</html>
```

**#A Tests our supposition**
**#B Declares functions after references**

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



- canFly can't be forward referenced.
- Nor can isDeadly.

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

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

## 3.2 Anonymous Functions

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

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

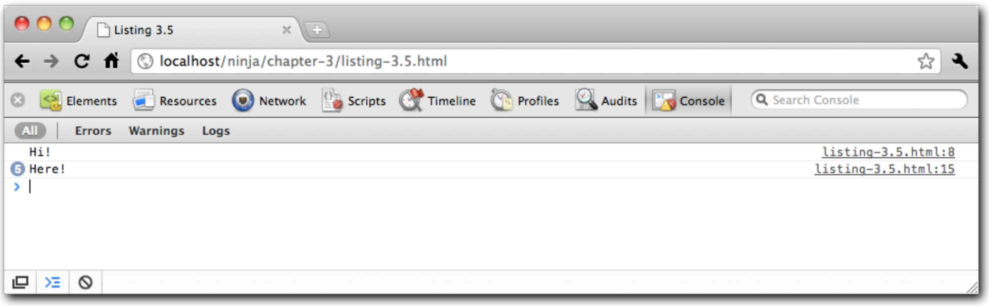
**Listing 3.5: Examples of declaring anonymous functions**

```
<!DOCTYPE html>
<html>
  <head>
    <title>Listing 3.5</title>
    <script type="text/javascript">
      var obj = {
          someMethod: function(){                          #1
            console.log('Hi!');                            #1
          }                                                #1
        };

        obj.someMethod();                                  #2

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

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

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

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

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



```
  Hi!                                                  listing-3.5.html:8
5 Here!                                               listing-3.5.html:15
>
```

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

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

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

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

### 3.2.1 Recursion

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

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

**Listing 3.6: Recursion in a named function**

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

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

</script>
```

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

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

**NOTE** This function satisfies two criteria for recursion: a reference to self, and convergence towards termination.

The function clearly calls itself, so the first criterion is satisfied. And because the value of parameter `n` decreases with each iteration, it will sooner or later reach a value less than zero and stop the recursion, satisfying the second criterion.

Note that a "recursive" function that does *not* converge towards termination is better known as an infinite loop!

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

**Listing 3.7: Function recursion within an object.**

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

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

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

**#1 Declares function as a property**

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

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

**Listing 3.8: Recursion using a missing function reference**

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

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

  ninja = {};                                                 #2

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

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

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

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

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

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

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

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

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

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

Observe the use of this technique in Listing 3.9:

**Listing 3.9: Using a named anonymous (inline) function**

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

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

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

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

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

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

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

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

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

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

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

  ninja();                                                          #3

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

</script>
```

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

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

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

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

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

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

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

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

```
</script>
```

**#1 References the argument.callee property**

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

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

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

## *3.3 Functions as objects*

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

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

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

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

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

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

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

```
var obj = {};
var fn = function(){};
obj.prop = "hitsuke (distraction)";
fn.prop = "tanuki (climbing)";
assert(obj.prop == fn.prop,
       "Both are objects, both have the property.");
```

This aspect of functions can be used in a number of different ways throughout a library or general on-page code. This is especially so when it comes to topics like event callback management. For now, let's look at a couple of the more interesting things that can be done such as storing functions and “memoizing”.

### 3.3.1 Storing Functions

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

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

**Listing 3.12: Storing a collection of unique functions**

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

  function ninja(){}

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

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

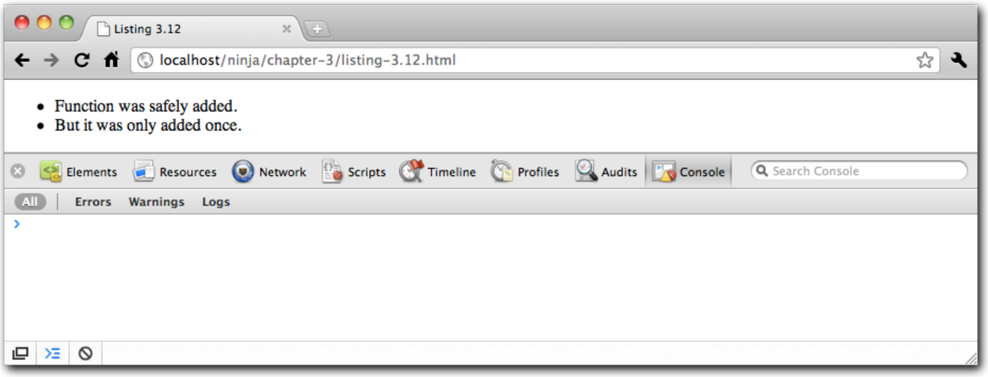
In Listing 3.12, we create an object, assigned to variable `store`, in which we will store a unique set of functions. This object has two data properties: one which stores a next available id value (#1), and one within which we will cache the stored functions (#2). Functions are added to this cache via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

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

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

**Tip** The `!!` construct is a simple way of turning any JavaScript expression into its Boolean equivalent. For example: `!!"hello" === true` and `!!0 === false`. In the above example we end up converting a function into its Boolean equivalent, which will always be `true`. (Sure we could have hard-coded true, but then we wouldn't have had a chance to introduce `!!`).

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



Listing 3.12

localhost/ninja/chapter-3/listing-3.12.html

- Function was safely added.
- But it was only added once.

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

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

## *3.3.2 Self-memoizing functions*

***Memoization*** (no, that's not a typo) is the process of building a function that is capable of remembering its previously computed answers. As a basic example, let's look at a simplistic algorithm for computing prime numbers. Note how the function appears just like a normal function but has the addition of an answer cache to which it saves, and retrieves, solved numbers, like in Listing 3.15.

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

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

  isPrime.answers = {};                                           //#3

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

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

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

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

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

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

to the beginning of the function.

This approach has two major advantages:

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

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

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

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

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

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

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

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

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

## 3.4 Context

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

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

Let’s start with some basics.

### 3.4.1 Contexts in methods

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

**Listing 3.14: Examining context within a function**

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

  katana.use();                                 #3

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

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

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

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

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

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

## 3.4.2 Contexts in standalone functions

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

**Listing 3.15: Context in a standalone function**

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

  katana();                                         #2

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

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

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

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

> **Note** In ECMAScript 3.1 strict mode, a slight change has taken place: when a function isn't defined as a property of an object no longer means that its context will be the global object. The change is that a function defined within another function will inherit the context of the outer function. What's interesting about this is that it ends up affecting very little code - having the context of inner functions default to the global object turned out to be quite useless in practice (hence the change).

Now that we understand the basics of function contexts, it's time to dig deeper and explore more complex usages.

## 3.4.3 Defining the context

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

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

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

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

Let's see how in listing 3.16.

**Listing 3.16: Choosing the context of a function call**

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

  var ronin = {};                                                 //#2

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

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

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

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

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

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

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

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

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

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

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

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

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

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

### 3.4.4 Looping and callbacks

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

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

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

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

**Listing 3.18: A looping function with a callback**

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

## 3.4.5 Faking array methods

Let's imagine that we wanted to create an object that behaved similarly to an array (say, a collection of aggregated DOM elements) but take on additional functionality not related to arrays per se.

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

**Listing 3.19: Simulating array-like methods**

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

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

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

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

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

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

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

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

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

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

## 3.5 Variable-length argument lists

JavaScript, as a whole, is very flexible in what it can do, and much of that flexibility defines the language as we know it today. One of these flexible and powerful features of JavaScript is the ability for functions to accept an arbitrary number of arguments. This flexibility offers the developer great control over how their functions, and therefore their applications, can be written.

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

### 3.5.1 Min/Max Number in an Array

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

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

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

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

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

**Listing 3.20: Generic min and max functions for arrays**

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

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

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

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

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

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

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

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

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

### *3.5.2 Function overloading*

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

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

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

#### DETECTING AND TRAVERSING ARGUMENTS

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

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

Take a look at listing 3.21.

**Listing 3-21: Traversing variable-length argument lists**

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

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

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

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

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

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

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

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

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

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

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

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

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

### Slicing and dicing an arguments list

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

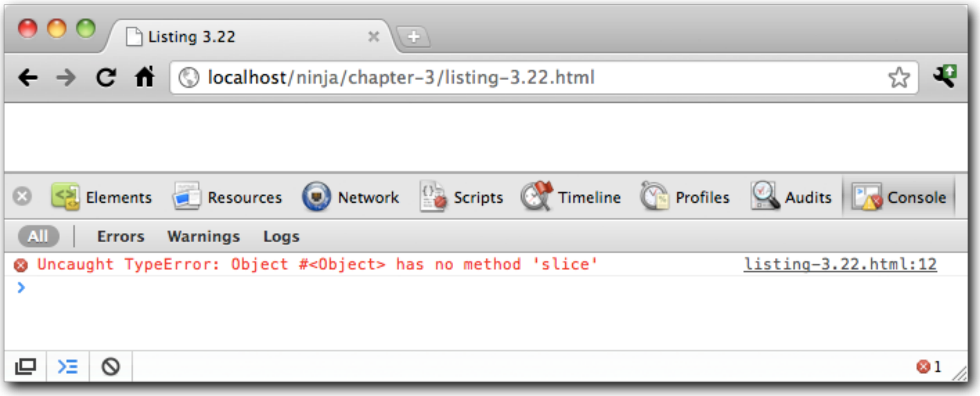
Seems simple enough – we'll grab the first argument, and multiply it by the result of using the `Math.max()` function, which we've already become familiar with, on the remainder of the argument values. Because we only want to pass the array that starts with the second element in the arguments list to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

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

**Listing 3.22: Slicing the arguments list**

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

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

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



```
Uncaught TypeError: Object #<Object> has no method 'slice'        listing-3.22.html:12
```

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

What's up with that?

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

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

**Listing 3.23: Slicing the arguments list – successfully this time**

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

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

</script>
```

**#1 Fools the `slice()` method**

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

Segue needed

### 3.5.3 Function overloading

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

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

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

#### THE FUNCTION'S LENGTH PROPERTY

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

Examine the following code:

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

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

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

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

#### OVERLOADING FUNCTIONS BY ARGUMENT COUNT

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

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

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

Let's look at Listing 3.24.

**Listing 3.24 A function overloading function**

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

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

Our `addMethod()` function accepts three arguments:

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

For example, the statement:

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

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

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

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

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

How can this be? Let's find out.

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

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

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

There's a bit of sleight of hand going on here with regard to how the inner function accesses `old` and `fn`, that involves a concept called closures, which we'll be taking a close look at in the next chapter. For now, just accept that when it executes, the inner function has access to `old` and `fn`.

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

**Listing 3.25Testing the `addMethod` function**

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

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

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

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

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

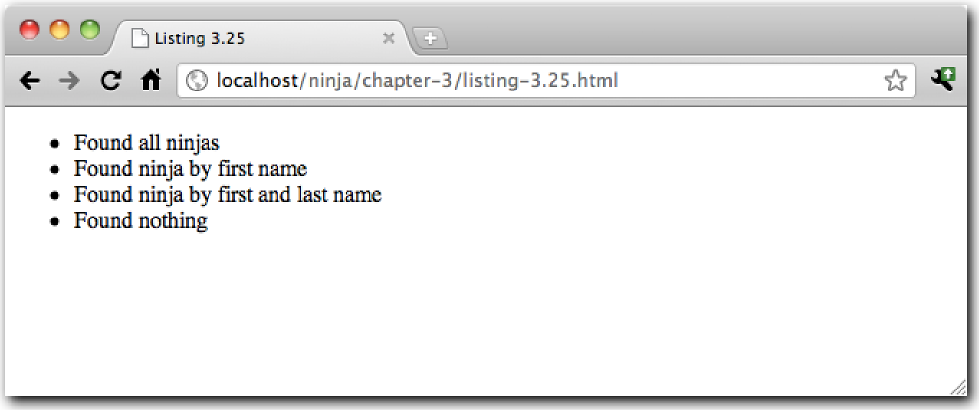
**#1 Declares base object with test data**
**#2 Binds no-argument method**
**#3 Binds single-argument method**
**#4 Binds dual-argument method**
**#5 Tests bound methods**

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

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

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

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



- Found all ninjas
- Found ninja by first name
- Found ninja by first and last name
- Found nothing

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

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

It should be noted that there are some caveats to be aware of when using this particular technique:

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

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

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

## 3.6 Checking for functions

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

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

```
function ninja(){}

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

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

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

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

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

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

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

There is one notable exception however (isn't there always?): Internet Explorer reports methods of DOM elements with a type of "object", like so: `typeof` `domNode.getAttribute == "object"` and `typeof inputElem.focus == "object"` – so this particular technique does not cover this case.

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

The reason why we don't just directly call `fn.toString()` to try and get this result is two-fold:

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

By accessing the `Object.prototype` method directly we ensure that we're not getting an overridden version of `toString()` and we end up with the exact information that we need.

This is just a quick taste of the strange world of cross-browser scripting. While it can be quite challenging the result is always rewarding: allowing us to painlessly write cross browser applications with little concern for the painful minutia. We'll explore lots more cross-browser strategies as we go along, and especially in chapter 10.

## 3.7 Summary

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

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

Then we explored the importance of treating functions like objects; attaching properties to them to store additional data, and the benefits of doing so.

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

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

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

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

# 4 Closures

In this chapter:

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

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

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

## 4.1 How closures work

Simply: A closure is a way to access and manipulate external variables from within a function. Another way of imagining it is the fact that a function is able to access all the variables, and functions, declared in the same scope as itself. The result is rather intuitive and is best explained through code, like in Listing 4.1.

**Listing 4.1: A few examples of closures.**

```
var outerValue = true;

function outerFn(arg1){
  var innerValue = true;
```

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

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

    innerFn(true);
  }

  outerFn(true);

  var otherValue = true;

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

Note that Listing 4.1 contains two closures: function `outerFn()` includes a closured reference to itself and the variable `outerValue`. function `innerFn()` includes a closured reference to the variables `outerValue`, `innerValue`, and `arg1` and references to the functions `outerFn()` and `innerFn()`.

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

### 4.1.1 Private Variables

One of the most common uses of closures is to encapsulate some information as a "private variable," of sorts. Object-oriented code, written in JavaScript, is unable to have traditional private variables (properties of the object that are hidden from outside uses). However, using the concept of a closure, we can arrive at an acceptable result, as in Listing 4.2.

**Listing 4.2: An example of keeping a variable private but accessible via a closure.**

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

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

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

In Listing 4.2 we create a variable to hold our state, `slices`. This variable is only accessible from within the `Ninja()` function (including the methods `getSlices()` and `slice()`). This means that we can maintain the state, within the function, without letting it be directly accessed by the user.

### 4.1.2 Callbacks and Timers

Another one of the most beneficial places for using closures is when you're dealing with callbacks or timers. In both cases a function is being called at a later time and within the function you have to deal with some, specific, outside data. Closures act as an intuitive way of accessing data, especially when you wish to avoid creating extra variables just to store that information. Let's look at a simple example of an Ajax request, using the jQuery JavaScript Library, in Listing 4.3.

**Listing 4.3: Using a closure from a callback in an Ajax request.**

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

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

There's a couple things occurring in Listing 4.3. To start, we're placing a loading message into the div to indicate that we're about to start an Ajax request. We have to do the query, for the div, once to edit its contents - and would typically have to do it again to inject the contents when the Ajax request completed. However, we can make good use of a closure to save a reference to the original jQuery object (containing a reference to the div element), saving us from that effort.

Listing 4.4 has a slightly more complicated example, creating a simple animation.

**Listing 4.4: Using a closure from a timer interval.**

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

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

What's especially interesting about the Listing 4.4 is that it only uses a single (anonymous) function to accomplish the animation, and three variables, accessed via a closure. This structure is particularly important as the three variables (the DOM element, the pixel counter, and the timer reference) all must be maintained across steps of the animation. This example is a particularly good one in demonstrating how the concept of closures is capable of producing some surprisingly intuitive, and concise, code.

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

## 4.2 Enforcing Function Context

Last chapter, when we discussed function context, we looked at how the `.call()` and `.apply()` methods could be used to manipulate the context of a function. While this manipulation can be incredibly useful, it can also be, potentially, harmful to object-oriented code. Observe Listing 4.5 in which an object method is bound to an element as an event listener.

**Listing 4.5: Binding a function, with a specified context, to an element.**

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

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

Now, Listing 4.5 fails because the `'this'` inside of the click function is referring to whatever context the function currently has. While we intend it to refer to the Button object, when it's re-bound to another object context the clicked property is transferred along with it. In this particular example addEventListener redefines the context to be the target element, which causes us to bind the clicked property to the wrong object.

Now, there's a way around this. We can enforce a particular function to always have our desired context, using a mix of anonymous functions, .apply(), and closures. Observe Listing 4.6 - same as Listing 4.5 - but actually working with our desired result, now.

**Listing 4.6: An alternative means of enforcing function context when binding a function.**

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

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

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

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

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

The bind() function, above, is a simple version of the function popularized by the Prototype JavaScript Library. Prototype promotes writing your code in a clean, classicial-style, object-oriented way. Thus, most object methods have this particular "incorrect context" problem when re-bound to another object. The original version of the method looks something like the code in Listing 4.7.

**Listing 4.7: An example of the function binding code used in the Prototype library.**

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

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

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

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

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

Note that this method is quite similar to the function implemented in Listing 4.7, but with a couple, notable, additions. To start, it attaches itself to all functions, rather than presenting itself as a globally-accessible function. You would, in turn, use the function like so: `myFunction.bind(myObject)`. Additionally, with this method, you are able to bind arguments to the anonymous function. This allows you to pre-specify some of the arguments, in a form of partial function application (which we'll discuss in the next section).

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

## 4.3 Partially Applying Functions

Partially applying a function is a particularly interesting technique in which you can pre-fill-in arguments, to a function, before it is ever executed. In effect, partially applying a function returns a new function which you can call. This is best understood through an example, as in Listing 4.8.

**Listing 4.8: Partially applying arguments to a native function (using the code from Listing 4.10).**

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

var results = ("John, Resig, Boston").csv();
assert( results[1] == "Resig",
  "The text values were split properly" );
```

In Listing 4.8 we've taken a common function - a String's `.split()` method - and have pre-filled-in the regular expression upon which to split. The result is a new function, `.csv()` that we can call at any point to convert a list of comma-separated values into an array.

Filling in the first couple arguments of a function (and returning a new function) is typically called currying. With that in mind, let's look at how the curry method is, roughly, implemented in the Prototype library, in Listing 4.9.

**Listing 4.9: An example of a curry function (filling in the first specified arguments).**

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

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

Now, this style of partial function application is perfectly useful, but we can do better. What if we wanted to fill in any missing argument from a given function - not just the first ones. Implementations of this style of partial function application have existed in other languages but Oliver Steele was one of the first to demonstrate it with his Functional.js (http://osteele.com/sources/javascript/functional/) library. Listing 4.10 has a possible implementation.

**Listing 4.10: An example of a more-complex partial application function (filling in the specified**

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

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

We already had the example of constructing a string splitting function, above, but let's look at some other ways in which this new functionality could be used. To start we could construct a function that's able to be easily delayed, like in Listing 4.11.

**Listing 4.11: Using partial application on a timer function.**

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

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

This means that we now have a new function, named delay, which we can pass another function in to, at any time, to have it be called asynchronously (after 10 milliseconds). We could, also create a simple function for binding events, as in Listing 4.12.

**Listing 4.12: Using partial application on event binding.**

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

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

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

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

## 4.4 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.

### 4.4.1 Memoization

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

Let's examine a method, `.memoized()`, in Listing 4.13 that we can use to remember return values from an existing function. There are no closures involved here, only functions.

**Listing 4.13: An example of function value memoization.**

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

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

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

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

Our ability to introspect into an existing function is limited. However, by adding a new `.memoized()` method we do have the ability to modify and attach properties that are associated with the function itself. This allows us to create a data store (`._values`) in which all of our pre-computed values can be saved.

This means that the first time the `.memoized()` method is called, looking for a particular value, a result will be computed and stored from the originating function. However upon subsequent calls that value will be saved and be returned immediately.

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

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

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

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

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

Now that we have a method for memoizing the values coming in-and-out of an existing function let's explore how we can use closures to produce a new function, capable of having all of its function calls be memoized, in Listing 4.14.

**Listing 4.14: A different technique for building a memoized function (using the code from Listing 4.13).**

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

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

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

Listing 4.14 builds upon our previous .memoized() method constructing a new method: `.memoize()`. This method returns a function which, when called, will always be the memoized results of the original function.

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

In Listing 4.14 we show a, comparatively, strange case of defining a new function, when we define `isPrime()`. Since we want `isPrime` to always be memoized we need to construct a temporary function whose results won't be memoized (much like what we did in the first example - but in a temporary fashion).

We take this anonymous, prime-figuring, function and memoize it immediately, giving us a new function which is assigned to the `isPrime` name. We'll discuss this construct, in depth, in the `(function(){})()` section. Note that, in this case, it is impossible to compute if a number is prime in a non-memoized fashion. Only a single isPrime function exists and it completely encapsulates the original function, hidden within a closure.

Listing 4.14 is a good demonstration of obscuring original functionality via a closure. This can be particularly useful (from a development perspective) but can also be crippling: If you obscure too much of your code then it becomes unextendable, which can be undesirable. However, hooks for later modification often counter-act this. We'll discuss this matter in depth throughout the book.

### 4.4.2 Function Wrapping

Function wrapping is a means of encapsulating the functionality of a function, and overwriting it, in a single step. It is best used when you wish to override some previous behavior of a function, while still allowing certain use-cases to still execute. The use is best demonstrated when implementing pieces of cross-browser code, like in Listing 4.15 from Prototype:

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

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

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

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

In Listing 4.15 the Prototype library is using the wrap function to implement a piece of browser-specific functionality. Specifically they're attempting to work around some bug with Opera's implementation of accessing title attributes. Their technique is interesting, as opposed to having a large if/else block within their readAttribute function (debatably messy and not a good separation of concerns) they, instead, opt to completely override the old method, simply implementing this fix, and deferring the rest of the functionality back to the original function. As with the previous example, in which we overwrote, and encapsulated, the isPrime function, so we are doing here, using a closure.

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

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

## 4.5 (function(){})()

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

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

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

### 4.5.1 Temporary Scope and Private Variables

Using the executed anonymous function we can start to build up interesting enclosures for our work. Since the function is executed immediately, and all the variables inside of it are kept inside of it, we can use this to create a temporary scope within which our state can be maintained, like in Listing 4.16.

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

**Listing 4.16: Creating a temporary enclosure for persisting a variable.**

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

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

Since the above anonymous function is executed immediately the click handler is also bound right away. Additionally, a closure is created allowing the `numClicks` variable to persist with the handler. This is the most common way in which executed anonymous functions are used, just as a simple wrapper. However it's important to remember that since they are functions they can be used in interesting ways, like in Listing 4.17.

**Listing 4.17: An alternative to the example in Listing 4.16, returning a value from the enclosure.**

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

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

This is a, debatably, more complicated version of our first example in Listing 4.16. In this case we're, again, creating an executed anonymous function but this time we return a value from it. Since this is just like any other function that value is returned and passed along to the `addEventListener` method. However, this function that we've created also gets the necessary `numClicks` variable via its closure. This technique is a very different way of looking at scope. In most languages you can scope things based upon the block which they're in. In JavaScript variables are scope based upon the function they're in.

However, with this simple construct, we can now scope variables to block, and sub-block, levels. The ability to scope some code to a unit as small as an argument within a function call is incredibly powerful and truly shows the flexibility of the language.

Listing 4.18 has a quick example from the Prototype JavaScript library:

**Listing 4.18: Using the anonymous function wrapper as a variable shortcut.**

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

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

### 4.5.2 Loops

One of the most useful applications of executed anonymous functions is the ability to solve a nasty issue with loops and closures. Listing 4.19 has a common piece of problematic code:

**Listing 4.19: A problematic piece of code in which the iterator is not maintained in the closure.**

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

In Listing 4.19 we encounter a common issue with closures and looping, namely that the variable that's being enclosed (`i`) is being updated after the function is bound. This means that every bound function handler will always alert the last value stored in i (in this case, '2'). This is due to the fact that closures only remember references to variables - not their actual values at the time at which they were called. This is an important distinction and one that trips up a lot of people.

Not to fear, though, as we can combat this closure craziness with another closure, like in Listing 4.20.

**Listing 4.20: Using an anonymous function wrapper to persist the iterator properly.**

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

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

### 4.5.3 Library Wrapping

The final concept that closures, and executed anonymous functions, encapsulate is an important one to JavaScript library development. When developing a library it's incredibly important that you don't pollute the global namespace with unnecessary variables, especially ones that are only temporarily used. This is a point at which the executed anonymous functions can be especially useful: Keep as much of the library private as possible and only selectively introduce variables into the global namespace. The jQuery library actually does a good job of this, completely enclosing all of its functionality and only introducing the variables it needs, like `jQuery`, in Listing 4.21.

**Listing 4.21: Placing a variable outside of a function wrapper.**

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

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

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

However, that isn't the only way in which that type of definition could be done, another is shown in Listing 4.22.

**Listing 4.22: An alternative means of putting a variable in the outer scope.**

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

  // ...
  return jQuery;
})();
```

The code in Listing 4.22 would have the same effect as what was shown in the Listing 4.21, but structured in a different manner. Here we define a `jQuery` function within our anonymous scope, use it and define it, then return it back out the global scope where it is assigned to a variable, also named `jQuery`. Oftentimes this particular technique is preferred, if you're only exporting a single variable, as it's clearer as to what the result of the function is.

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

## 4.6 Summary

In this chapter we dove in to how closures work in JavaScript. We started with the basics, looking at how they're implemented and then how to use them within an application. We looked at a number of cases where closures were particularly useful, including the definition of private variables and in the use of callbacks and timers. We then explored a number of advanced concepts in which closures helped to sculpt the JavaScript language including forcing function context, partially applying functions, and overriding function behavior. We then did an in-depth exploration of the `(function(){})()` construct which, as we learned, has the power to re-define how the language is used. In total, understanding closures will be an invaluable asset when developing complex JavaScript applications and will aid in solving a number of common problems that we encounter.

# 5 Function prototypes

This chapter covers:

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

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

## 5.1 Instantiation and Prototypes

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

### 5.1.1 Instantiation

Let's examine the simplest case of using the prototype property to add functionality to an instance of a function, in Listing 5.1.

**Listing 5.1: Creating an instance of an function that has a prototyped method.**

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

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

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

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

There's two points to learn from Listing 5.1: First that in order to produce an instance of a function (something that's an object, has instance properties, and has prototyped properties) you have to call the function with the `new` operator. Second we can see that the `swingSword` function has now become a property of the `ninja2` `Ninja` instance.

In some respects the base function (in this case, `Ninja`) could be considered a "constructor" (since it's called whenever the new operator is used upon it). This also means that when the function is called with the new operator it's `this` context is equal to the instance of the object itself. Let's examine this a little bit more in Listing 5.2.

**Listing 5.2: Extending an object with both a prototyped method and a method within the constructor function.**

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

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

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

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

In Listing 5.2 we're doing an operation very similar to the previous example: We add a new method by adding it to the prototype property. However, we also add a new method within the constructor function itself. The order of operations is as such:

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

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

An important thing to realize about a function's prototype is that, unlike instance properties which are static, it will continue to update, live, as it's changed - even against all existing instance of the object. For example if we were to take some of the code in Listing 5.2 and re-order it, it would still work as we would expect it to, as shown in Listing 5.3.

**Listing 5.3: The prototype continues to update instance properties live even after they've already been created.**

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

var ninja = new Ninja();

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

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

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

### *5.1.2 Object Type*

Since we now have this new instance of our function it becomes important to know which function constructed the object instance, so we can refer back to later (possibly even performing a form of type checking, if need be), as in Listing 5.4.

**Listing 5.4: Examining the type of an instance and its constructor.**

```
function Ninja(){}

var ninja = new Ninja();

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

In Listing 5.4 we start by examining the type of a function instance. Note that the `typeof` operator isn't of much use to us here; all instance will be objects, thus always returning `"object"` as its result. However, the `instanceof` operator really helps in this case, giving us a clear way to determine if an instance was created by a particular function.

On top of this we also make use of an object property: `.constructor`. This is a property that exists on all objects and offers a reference back to the original function that created it. We can use this to verify the origin of the instance (much like how we do with the `instanceof` operator). Additionally, since this is just a reference back to the original function, we can, once-again, instantiate a new Ninja object, like in Listing 5.5.

**Listing 5.5: Instantiating a new object using only a reference to its constructor.**

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

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

What's especially interesting about Listing 5.5 is that we can do this without ever having access to the original function; taking place completely behind the scenes.

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

### *5.1.3 Inheritance and the Prototype Chain*

There's an additional feature of the instanceof operator that we can use to our advantage when performing object inheritance. In order to make use of it, however, we need to understand how the prototype chain works in JavaScript, an example of which is shown in Listing 5.6.

**Listing 5.6: Different ways of copying functionality onto a function's prototype.**

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

function Ninja(){}

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

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

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

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

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

An additional side-effect of doing prototype inheritance in this manner is that all inherited function prototypes will continue to update live. The result is something akin to what's shown in Figure 5.1.

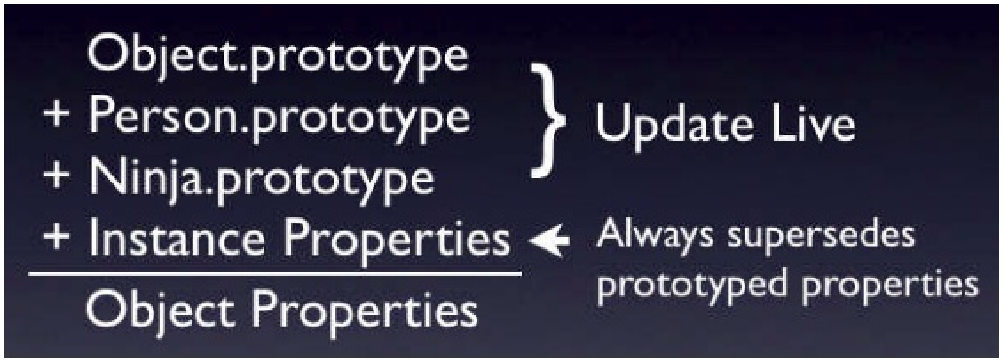


Figure 5.1: The order in which properties are bound to an instantiated object.

It's important to note from Figure 5.1 is that our object has properties that are inherited from the Object prototype. This is important to note: All native objects constructors (such as Object, Array, String, Number, RegExp, and Function) have prototype properties which can be manipulated and extended (which makes sense, since each of those object constructors are functions themselves). This proves to be an incredibly powerful feature of the language. Using this you can extend the functionality of the language itself, introducing new, or missing, pieces of the language.

For example, one such case where this becomes quite useful is with some of the features of JavaScript 1.6. This version of the JavaScript language introduces a couple helper methods, including some for Arrays. One such method is `forEach` which allows to iterate over the properties of an array, calling a function on every property. This can be especially handy for situations where we want to plug in different pieces of functionality without changing the overall looping structure. We can duplicate this functionality, right now, completely circumventing the need to wait until the next version of the language is ready, like in Listing 5.7.

**Listing 5.7: Implementing the JavaScript 1.6 array forEach method in a future-compatibile manner.**

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

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

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

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

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

## *5.1.4 HTML Prototypes*

There's a fun feature in Internet Explorer 8, Firefox, Safari, and Opera which provides base functions representing objects in the DOM. By making these accessible the browser is providing you with the ability to extend any HTML node of your choosing, like this in the Listing 5.8.

**Listing 5.8: Adding a new method to all HTML elements via the HTMLElement prototype.**

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

// Old way
```

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

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

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

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

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

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

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

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

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

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

## 5.2 Gotchas

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

### 5.2.1 Extending Object

Perhaps the most egregious mistake that someone can make with prototypes is to extend the native `Object.prototype`. This difficulty is that when you extend this prototype ALL objects receive those additional properties. This is especially problematic since when you iterate over the properties of the object these new properties appear, causing all sorts of unexpected behavior, as shown in Listing 5.9.

**Listing 5.9: The unexpected behavior of adding extra properties to the Object prototype.**

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

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

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

There is one workaround, however. Browsers provide a method called `hasOwnProperty` which can be used to detect properties which are actually on the object instance and not be imported from a prototype. The result can be seen in Listing 5.10.

**Listing 5.10: Using the hasOwnProperty method to tame Object prototype extensions.**

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

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

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

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

### *5.2.2 Extending Number*

It's safe to extend most native prototypes (save for Object, as previously mentioned). One other problematic natives is that of Number. Due to how numbers, and properties of numbers, are parsed by the JavaScript engine the result can be rather confusing, like in Listing 5.11.

**Listing 5.11: Adding a method to the Number prototype.**

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

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

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

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

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

### 5.2.3 Sub-classing Native Objects

Another tricky point is in the sub-classing of native objects. The one object that's quite simple to sub-class is Object (since it's the root of all prototype chains to begin with). However once you start wanting to sub-class other native objects the situation becomes less clear-cut. For example, with Array, everything works as you might expect it to, as shown in Listing 5.12.

**Listing 5.12: Inheriting functionality from the Array object.**

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

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

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

In general it's a better strategy to implement individual pieces of functionality from native objects, rather than attempt to sub-class them completely, such as implementing the push method directly, like in Listing 5.13.

**Listing 5.13: Simulating Array functionality but without the true sub-classing.**

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

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

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

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

In Listing 5.13 we end up calling the native Array methods and using `.apply()` to trick them into thinking that they're working on an array object. The only property that we have to implement ourselves is the `length` (since that's the one property that must remain mutable - the feature that Internet Explorer does not provide).

### 5.2.4 Instantiation

Let's start by looking at a simple case of a function which will be instantiated and populated with some values, in Listing 5.14.

**Listing 5.14: The result of leaving off the new operator from a function call.**

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

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

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

If a function is meant to be instantiated, and isn't, it can in turn end up polluting the current scope (frequently the global namespace), causing even more unexpected results. For example, observe the following code in Listing 5.15.

**Listing 5.15: An example of accidentally introducing a variable into the global namespace.**

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

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

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

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

Listing 5.16 shows a solution to all of the above problems, with a simple technique.

**Listing 5.16: Enforcing the correct context with arguments.callee and instanceof.**

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

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

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

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

This new function seems straightforward but takes a little bit of consideration. What we're doing is checking to see if we're currently inside of an instantiated version of the function, rather than just the function itself. Let's look at a simpler example, in Listing 5.17.

**Listing 5.17: Determining if we're inside of an instantiated function, or not.**

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

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

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

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

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

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

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

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

## 5.3 Class-like Code

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

Generally there are a couple features that developers crave:

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

For example, if you had a system which made this process easier, Listing 5.18 shows an example of what you can do with it:

**Listing 5.18: An example of classical-style inheritance, using the code from Listing 5.19.**

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

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

```
      return this._super();
    },
    swingSword: function(){
      return true;
    }
  });

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

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

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

A couple things to note about this implementation:

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

There are a number of different JavaScript classical-inheritance-simulating libraries that already exist. Out of all them there are two that stand up above the others: The implementations within base2 and Prototype. While they contain a number of advanced features the absolute core is the important part of the solution and is what's distilled in the implementation. The code in Listing 5.19 helps to enforce the notion of 'classes' as a structure, maintains simple inheritance, and allows for the super method calling.

**Listing 5.19: A simple class creation library.**

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

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

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

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

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

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

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

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

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

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

      // And make this class extendable
      Class.subClass = arguments.callee;

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

The two trickiest parts of Listing 5.19 are the "initializing/don't call init" and "create _super method" portions. Having a good understanding of what's being achieved in these areas will help with your understanding of the full implementation.

### Initialization

In order to simulate inheritance with a function prototype we use the traditional technique of creating an instance of the super-class function and assigning it to the prototype. Without using the above it would look something like the code in 5.20.

**Listing 5.20: Another example of maintaining the prototype chain.**

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

What's challenging about Listing 5.20, though, is that all we really want is the benefits of 'instanceof', not the whole cost of instantiating a Person object and running its constructor. To counteract this we have a variable in our code, `initializing`, that is set to true whenever we want to instantiate a class with the sole purpose of using it for a prototype.

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

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

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

### Super Method

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

For example, if you wanted to call a super-classes' constructor you could do that with the technique shown in Listing 5.21.

**Listing 5.21: An example of calling the inherited super method.**

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

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

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

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

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

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

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

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

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

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

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

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

## 5.4 Summary

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

# 6 Timers

In this chapter:

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

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

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

## 6.1 How Timers Work

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

- `var id = setTimeout(fn, delay);` - Initiates a single timer which will call the specified function after the delay. The function returns a unique ID with which the timer can be cancelled at a later time.

• `var id = setInterval(fn, delay);` - Similar to `setTimeout` but continually calls the function (with a delay every time) until it is cancelled.

• `clearInterval(id);, clearTimeout(id);` - Accepts a timer ID (returned by either of the aforementioned functions) and stops the timer callback from occurring.

In order to understand how the timers work internally there's one important concept that needs to be explored: timer delay is not guaranteed. Since all JavaScript in a browser executes on a single thread asynchronous events (such as mouse clicks and timers) are only run when there's been an opening in the execution. This is best demonstrated with a diagram, like in Figure 6.1.

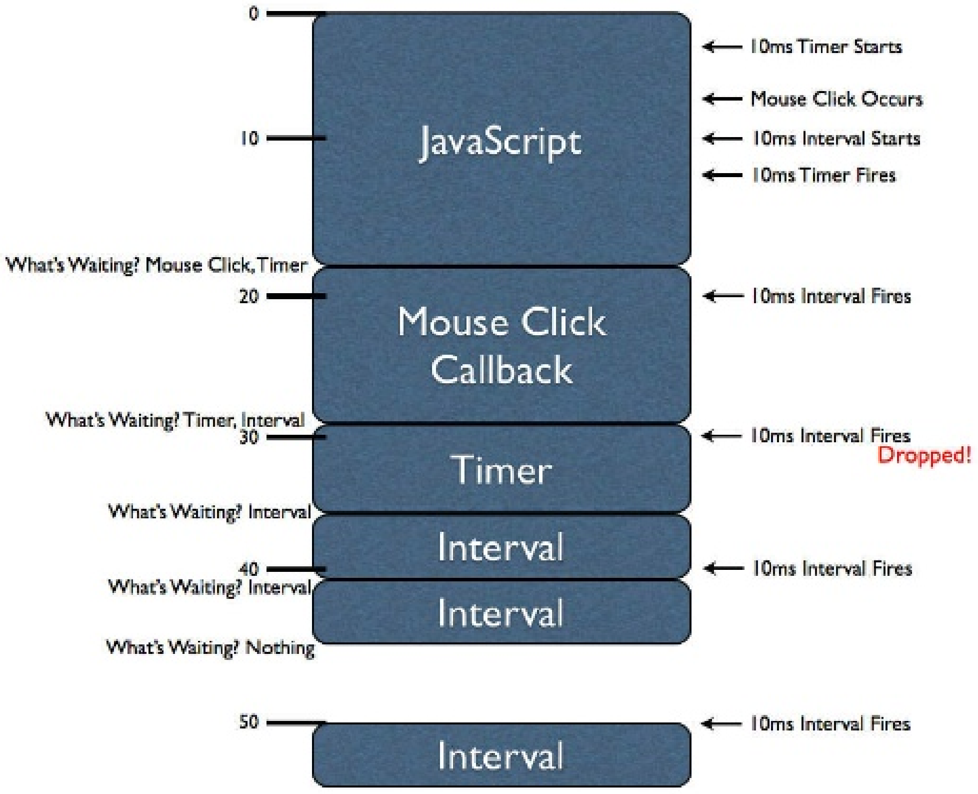


Figure 6.1: An explanation of timers.

There's a lot of information in Figure 6.1 to digest but understanding it completely will give you a better realization of how asynchronous JavaScript execution works. This diagram is one dimensional: vertically we have the (wall clock) time, in milliseconds. The blue boxes represent portions of JavaScript being executed. For example the first block of JavaScript executes for approximately 18ms, the mouse click block for approximately 11ms, and so on.

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

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

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

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

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

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

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

Let's take a look at an example to better illustrate the differences between `setTimeout` and `setInterval`, in Listing 6.1.

**Listing 6.1: Two examples of repeating timers.**

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

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

The two pieces of code in Listing 6.1 may appear to be functionally equivalent, at first glance, but they are not. Notably the `setTimeout` code will always have at least a 10ms delay after the previous callback execution (it may end up being more, but never less) whereas the `setInterval` will attempt to execute a callback every 10ms regardless of when the last callback was executed.

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

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

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

## 6.2 Minimum Timer Delay and Reliability

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

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

We can come to that conclusion by performing some simple analysis upon the presumed timer intervals across platforms. For example if we analyze a `setInterval` `delay` of `0`ms we can find what the minimum delay is for most browsers, as in Figure 6.2 and Figure 6.3.

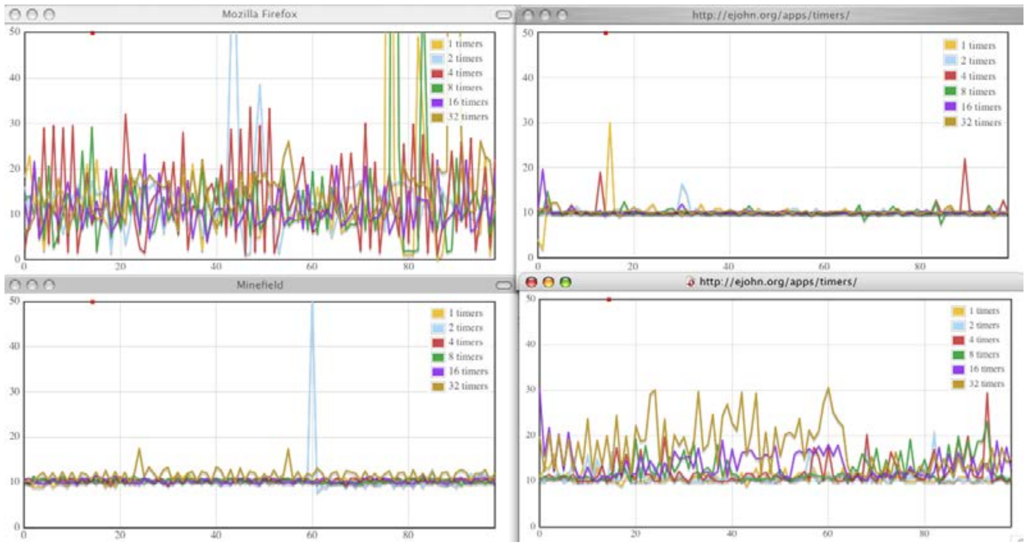


Figure 6.2: OSX Browsers, From Top Left: Firefox 2, Safari 3, Firefox 3, Opera 9.

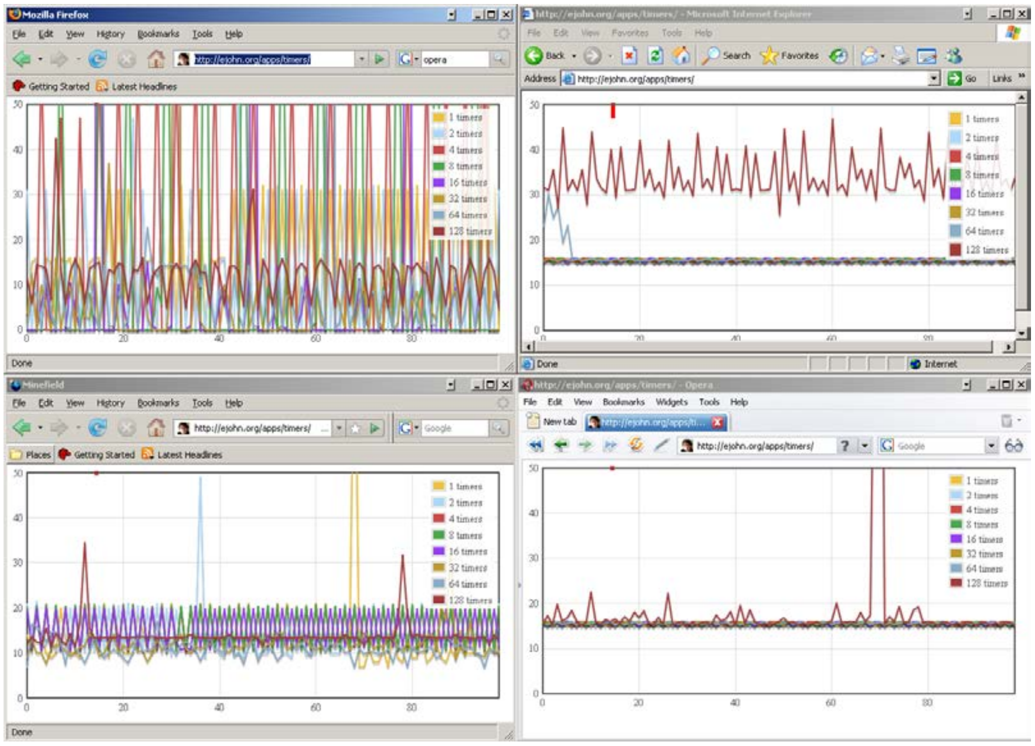

Mozilla Firefox
http://ejohn.org/apps/timers/
1 timers
2 timers
4 timers
8 timers
16 timers
32 timers
64 timers
128 timers
http://ejohn.org/apps/timers/ - Microsoft Internet Explorer
Minefield
http://ejohn.org/apps/timers/ - Opera

Figure 6.3: Windows Browsers, From Top Left: Firefox 2, Internet Explorer 6, Firefox 3, Opera 9

In Figure 6.2 and Figure 6.3 the lines and numbers indicate the number of simultaneous intervals being executed by the browser.

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

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

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

So while very small delay values can be provided by a browser the exact accuracy is not always guaranteed.

This needs to be taken into account in your applications when using timers (if the difference between 10ms and 15ms is problematic then you might have to re-think how your application code is structured).

## 6.3 Computationally-Expensive Processing

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

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

This is where timers become especially useful. Since timers are capable of, effectively, suspending execution of a piece of JavaScript until a later time it also prevents the browser from hanging (as long as the individual pieces of code aren't enough to cause the browser to hang). Taking this into account we can now convert normal, intensive, loops and operations into non-blocking operations. Let's look at Listing 6.2 where this type of operation would become necessary.

**Listing 6.2: A long-running task.**

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

In this example we're creating a total of 26,000 DOM nodes, populating a table with numbers. This is incredibly expensive and will likely hang the browser preventing the user from performing normal interactions. We can introduce timers into this situation to achieve a different, and perhaps better, result, as shown in Listing 6.3

**Listing 6.3: Using a timer to break up a long-running task.**

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

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

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

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

There's one perceptible change when using this technique, in relation to the original technique, namely that a long browser hang is now replaced with 4 visual updates of the page. While the browser will attempt to do these segments as quickly as possible it will also render the DOM changes after each step of the timer (just as it would after one large bulk update). Most of the time it's imperceptible to the user to see these types of updates occur but it's important to remember that they occur. One situation in which this technique has served me particularly well was in an application that I constructed to compute schedule permutations for college students. Originally the application was a typical CGI (communicating from the client to the server, where the schedules were computed and sent back) but I converted it to move all schedule computation to the client-side. A view of the schedule computation screen can be seen in Figure 6.4.

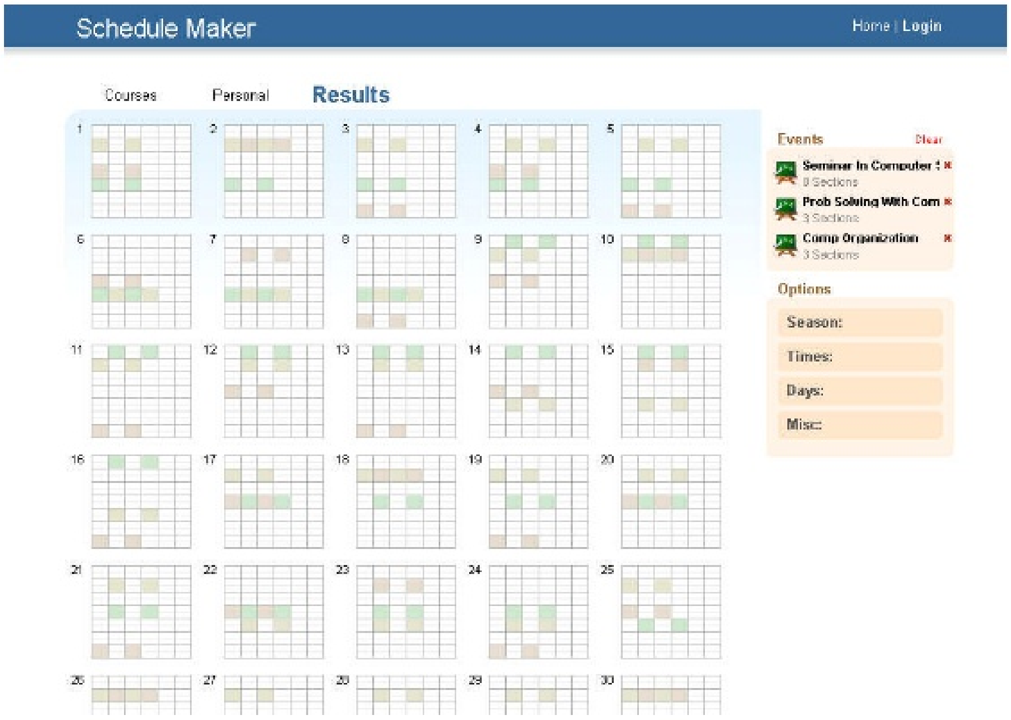


Figure 6.4: A screenshot of a web-based schedule computation application.

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

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

## 6.4 Central Timer Control

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

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

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

It's also important to realize that increasing the number of simultaneous timers is likely to increase the likelihood of a garbage collection occurring in the browser. Roughly speaking this is when the browser goes through and tries to tie up any loose ends (removing unused variables, objects, etc.). Timers are particularly problematic since they are generally managed outside of the flow of the normal JavaScript engine (through other threads). While some browsers are more capable of handling this situation others cause long garbage collection cycles to occur. You might have noticed this when you see a nice, smooth, animation in one browser - view it in another and see it stop-and-start its way to completion. Reducing the number of simultaneous timers being used will drastically help with this (and is the reason why all modern animation engines include a construct like the central timer control).

Let's take a look at an example where we have multiple functions animating separate properties but being managed by a single timer function in Listing 6.4.

**Listing 6.4: Using a central timer control to manage multiple animations.**

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

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

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

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

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

We've created a central control structure in Listing 6.4 that we can add new timer callback functions to and stop/start the execution of them. Additionally, the callback functions have the ability to remove themselves at any time by simply returning 'false' (which is much easier to do than the typical `clearTimeout` pattern). Let's step through the code to see how it works.

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

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

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

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

## 6.5 Asynchronous Testing

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

For example in a normal test suite we could easily just run the tests as we come to them however once we introduce the desire to do asynchronous testing we need to break all of those tests out and handle them separately. Listing 6.5 has some code that we can use to achieve our desired result.

**Listing 6.5: A simple asynchronous test suite.**

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

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

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

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

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

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

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

```
      resume();
    }, 200);
  });
```

The important aspect in Listing 6.5 is that each function passed to `test()` will contain, at most, one asynchronous test. It's asynchronicity is defined by the use of the `pause()` and `resume()` functions, to be called before and after the asynchronous event. Really, the above code is nothing more than a means of keeping asynchronous behavior-containing functions executing in a specific order (it doesn't, exclusively, have to be used for test cases, but that's where it's especially useful).

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

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

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

## 6.6 Summary

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

# 7 Regular Expressions

Covered in this chapter:

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

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

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

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

## 7.1 Compiling

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

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

There are two ways of defining a compiled regular expression in JavaScript, as shown in Listing 7.1.

**Listing 7.1: Two examples of creating a compiled regular expression.**

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

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

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

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

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

For example, let's say that you wanted to determine which elements, within a document, had a particular class name. Since elements are capable of having multiple class names associated with them (separated via a space) this makes for a good time to try regular expression compilation, shown in Listing 7.2.

**Listing 7.2: Compiling a regular expression for future use.**

```
<div class="foo ninja"></div>
<div class="ninja foo"></div>
<div></div>
```

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

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

  return results;
}

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

There are a number of things that we can learn from Listing 7.2. To start, note the use of new `RegExp(...)` to compile a regular expression based upon the input of the user. We construct this expression once, at the top of the function, in order to avoid frequent calls for re-compilation. Since the contents of the expression are dynamic (based upon the incoming `className` argument) we can get a major performance savings by handling the expression in this manner.

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

The ultimate example of using regular expression compilation, within a library, can be seen in the RegGrp package of the base2 library. This particular piece of functionality allows you to merge regular expressions together - allowing them to operate as a single expression (for speed) while maintaining all capturing and back-references (for simplicity). While there are too many details within the RegGrp package to go into, in particular, we can instead create a simplified version for our own purposes, like in Listing 7.3.

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

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

**Listing 7.3: Merging multiple regular expressions together.**

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

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

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

## 7.2 Capturing

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

For example in Listing 7.4 we extract the opacity value out of the filter value that Internet Explorer provides:

**Listing 7.4: A simple function for getting the current opacity of an element.**

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

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

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

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

Note that this return set is not always the case, from `match`, as shown in Listing 7.5.

**Listing 7.5: The difference between a global and local search with `match`.**

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

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

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

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

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

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

We can regain this functionality, while still maintaining a global search, by using the regular expression exec method. This method can be repeatedly called against a regular expression - causing it to return the next matched set of information every time it gets called. A typical pattern, for how it is used, looks like something seen in Listing 7.6.

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

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

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

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

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

### 7.2.1 References to Captures

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

For example, let's revisit the match in Listing 7.6 (where we match an opening, or closing, HTML tag) and modify it to, also, match the inner contents of the tag, itself, in Listing 7.7.

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

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

var match = tag.exec(html);

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

match = tag.exec(html);

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

In Listing 7.7 we use `\1` in order to refer back to the first capture within the expression (which, in this case, is the name of the tag). Using this information we can match the appropriate closing tag - referring back to the right one (assuming that there aren't any tags of the same name within the current tag, of course). These back-reference codes continue up, referring back to each, individual, capture.

Additionally, there's a way to get capture references within the replace string of a `replace` method. Instead of using the back-reference codes, like in the previous example, we use the syntax of `$1, $2, $3,` up through each capture number, show in Listing 7.8.

**Listing 7.8: Using a capture reference, within a replace.**

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

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

### 7.2.2 Non-capturing Groups

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

Probably the most compelling use case for this method is the ability to optionally match entire words (multiple-character strings) rather than being restricted to just single character optional matches, an example of which is shown in Listing 7.9.

**Listing 7.9: Matching multiple words using a non-capturing group**

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

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

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

## 7.3 Replacing with Functions

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

For example, in Listing 7.10 we use the function to provide a dynamic replacement value to adjust the case of a character.

**Listing 7.10: Converting a string to camel case with a function replace regular expression.**

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

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

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

For example, if you were looking to convert a query string like `"foo=1&foo=2&blah=a&blah=b&foo=3"` into one that looks like this: `"foo=1,2,3&blah=a,b"` a solution using regular expressions and replace could result in some, especially, terse code as shown in Listing 7.11.

**Listing 7.11: A technique for compressing a query string.**

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

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

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

  return ret.join("&");
}

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

The interesting aspect of Listing 7.11 is in using the string replace function as a means of traversing a string for values, rather than as an actual search-and-replace mechanism. The trick is two-fold: Passing in a function as the replace value argument to the `.replace()` method and, instead of returning a value, simply utilizing it as a means of searching.

Let's examine this piece of code:

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

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

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

We can see this behavior in Listing 7.12.

**Listing 7.12: Replacing a value with a empty string, in a function.**

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

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

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

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

## 7.4 Common Problems

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

### 7.4.1 Trimming a String

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

The most-commonly seen solution looks something like the code in Listing 7.13.

**Listing 7.13: A common solution to stripping whitespace from a string.**

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

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

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

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

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

Of those solutions two are particularly interesting. The first is done using regular expressions, but with no \s+ and no | or operator, shown in Listing 7.14.

**Listing 7.14: A trim method breaking down into two expressions.**

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

The second technique completely discards any attempt at stripping whitespace from the end of the string using a regular expression and does it manually, as seen in Listing 7.15.

**Listing 7.15: A trim method which slices at the rear of the string.**

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

Looking at the final breakdown in performance the difference becomes quite noticeable - and easy to see how that even when a particular method of trimming strings is particular scalable (as is seen in the third trim method) and in Table 7.1.

Table 7.1: All time in ms, for 1000 iterations.

| | Selector Trim | Document Trim |
|---|---|---|
| Listing 7.13 | 8.7 | 2075.8 |
| Listing 7.14 | 8.5 | 3706.7 |
| Listing 7.15 | 13.8 | 169.4 |

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

### 7.4.2 Matching Endlines

When performing a search it's frequently desired that the . (which normally matches any character, except for endlines) would also include endline characters. Other regular expression implementations, in other
languages, frequently include an extra flag for making this possible. In JavaScript, there are two ways, as shown in Listing 7.16.

**Listing 7.16: Matching all characters, including endlines.**

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

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

### 7.4.3 Unicode

Frequently, for an expression, you'll want to match alphanumeric characters (such as for an ID selector in a CSS selector engine implementation) for later use. However, assuming that the alpha characters will only be in English is a little short-sighted. Expanding the set to include unicode characters ends up becoming quite useful - explicitly supporting multiple languages not covered by the traditional alphanumeric character set, as seen in Listing 7.17.

**Listing 7.17: Matching Unicode characters in a CSS selector.**

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

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

### *7.4.4 Escaped Characters*

Another, common, desire for matching in an implementation of a CSS selector engine is to support escaped characters. This allows the user to specify complex names that don't conform to typical naming conventions (either specified by the HTML or CSS specification) an example of which is in Listing 7.18.

**Listing 7.18: Matching escaped characters in a CSS selector.**

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

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

## *7.5 Summary*

Regular expressions permeate modern JavaScript development. Virtually every aspect of development depends on their use in some way. With a good understanding of the advanced concepts that have been covered in this chapter, any developer should feel comfortable in tackling a challenging piece of regular expression-using JavaScript code. The techniques including compiling regular expressions, capturing values, and replacing with functions - not to mention the many minor tricky points, such as dealing with Unicode. Together these these techniques make for a formidable development armada.

# 8
# *With statements*

Covered in this chapter:

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

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

To start, let's take a look at the basics of how they work, as shown in Listing 8.1.

**Listing 8.1: Exposing the properties of the katana object using `with(){}`.**

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

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

  isSharp = false;
  use();
```

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

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

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

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

**Listing 8.2: Attempting to add a new property to a with(){}'d object.**

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

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

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

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

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

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

There's one major caveat when using `with(){}`, though: It slows down the execution performance of any JavaScript that it encompasses - and not just objects that it interacts with. Let's look at three examples, in Listing 8.3.

**Listing 8.3: Examples of interacting with a with statement.**

```
var obj = { foo: "bar" }, value;

// "No With"
for ( var i = 0; i < 1000; i++ ) {
  value = obj.foo;
}

// "Using With"
with(obj){
  for ( var i = 0; i < 1000; i++ ){
    value = foo;
  }
}

// "Using With, No Access"
with(obj){
  for ( var i = 0; i < 1000; i++ ){
    value = "no test";
  }
}
```

The results of running the statements in Listing 8.3 result in some rather dramatic performance differences, as seen in Table 8.1.

Table 8.1: All time in ms, for 1000 iterations, in a copy of Firefox 3.

| | Average | Min | Max | Deviation |
|---|---|---|---|---|
| No With | 0.14 | 0 | 1 | 0.35 |
| Using With | 1.58 | 1 | 2 | 0.50 |
| Using With, No Access | 1.38 | 1 | 2 | 0.49 |

Effectively, any variables accesses must now run through an additional with-scope check which provides an extra level of overhead (even if the result of the with statement isn't being used, as in the third case). If you are uncomfortable with this extra overhead, or if you aren't interested in any of the conveniences, then `with(){}` statements probably aren't for you.

## 8.1 Convenience

Perhaps the most common use case for using `with(){}` is as a simple means of not having to duplicate variable usage or property access. JavaScript libraries frequently use this as a means of simplification to, otherwise, complex statements.

Here are a few examples from a couple major libraries, starting with Prototype in Listing 8.4.

**Listing 8.4: A case of using `with(){}` in the Prototype JavaScript library.**

```
Object.extend(String.prototype.escapeHTML, {
  div:  document.createElement('div'),
  text: document.createTextNode('')
});

with (String.prototype.escapeHTML) div.appendChild(text);
```

Prototype uses `with(){}`, in this case, as a simple means of not having to call `String.prototype.escapeHTML` in front of both `div` and `text`. It's a simple addition and one that saves three extra object property calls.

The example in Listing 8.5 is from the base2 JavaScript library.

**Listing 8.5: A case of using `with(){}` in the base2 JavaScript library.**

```
with (document.body.style) {
  backgroundRepeat = "no-repeat";
  backgroundImage =
    "url(http://ie7-js.googlecode.com/svn/trunk/lib/blank.gif)";
  backgroundAttachment = "fixed";
}
```

Listing 8.5 uses `with(){}` as a simple means of not having to repeat `document.body.style` again, and again - allowing for some super-simple modification of a DOM element's style object. Another example from base2 is in Listing 8.6.

**Listing 8.6: A case of using `with(){}` in the base2 JavaScript library.**

```
var Rect = Base.extend({
  constructor: function(left, top, width, height) {
    this.left = left;
    this.top = top;
    this.width = width;
    this.height = height;
    this.right = left + width;
    this.bottom = top + height;
  },

  contains: function(x, y) {
      with (this)
        return x >= left && x <= right && y >= top && y <= bottom;
    },

    toString: function() {
      with (this) return [left, top, width, height].join(",");
    }
  });
```

This second case within base2 uses `with(){}` as a means of simply accessing instance properties. Normally this code would be much longer but the terseness that `with(){}` is able to provide, in this case, adds some much-needed clarity.

The final example, in Listing 8.7, is from the Firebug developer extension for Firefox.

**Listing 8.7: An example of `with(){}` use within the Firebug Firefox extension.**

```
const evalScriptPre = "with(__scope__.vars){ with(__scope__.api){" +
  " with(__scope__.userVars){ with(window){";
const evalScriptPost = "}}}}";
```

The lines in Listing 8.7, from Firebug, are especially complex - quite possibly the most complex uses of `with(){}` in a publicly-accessible piece of code. These statements are being used within the debugger portion of the extension, allowing the user to access local variables, the firebug API, and the global object all within the JavaScript console. Operations like this are generally outside the scope of most applications, but it helps to show the power of `with(){}` and in how it can make incredibly complex pieces of code possible, with JavaScript.

One interesting takeaway from the Firebug example, especially, is the dual-use of `with(){}`, bringing precedence to the window object over other, introduced, objects. Performing an action like the following:

```
with ( obj ) { with ( window ) { ... } }
```

Will allow you to have the obj object's properties be introduced by `with(){}`, while having the global variables (available by window) take precedence, exclusively.

## 8.2 Importing Namespaced Code

As shown previously one of the most common uses for the `with(){}` statement is in simplifying existing statements that have excessive object property use. You can see this most commonly in namespaced code (objects within objects, providing an organized structure to code). The side effect of this technique is that it becomes quite tedious to re-type the object namespaces again-and-again.

Note Listing 8.8, both performing the same operation using the Yahoo UI JavaScript library, but also gaining extra simplicity and clarity by using `with(){}`.

**Listing 8.8: Binding click events to two elements using Yahoo UI - done both normally and using `with(){}`.**

```
YAHOO.util.Event.on(
  [YAHOO.util.Dom.get('item'), YAHOO.util.Dom.get('otheritem')],
  'click', function(){
    YAHOO.util.Dom.setStyle(this,'color','#c00');
  }
);

with ( YAHOO.util.Dom ) {
  YAHOO.util.Event.on([get('item'), get('otheritem')], 'click',
     function(){ setStyle(this,'color','#c00'); });
}
```

The addition of this single `with(){}` statement allows for a considerable increase in code simplicity. The resulting clarity benefits are debatable (especially when , although the reduction in code size

## 8.3 Clarified Object-Oriented Code

When writing object-oriented JavaScript a couple issues come into play - especially when creating objects that include private instance data. `with(){}` statements are able to change the way this code is written in a particularly interesting way. Observe the constructor code in Listing 8.9.

**Listing 8.9: Defining an object which has both private and public data, equally exposed using `with(){}`.**

```
function Ninja(){with(this){
  // Private Information
  var cloaked = false;

  // Public property
  this.swings = 0;

  // Private Method
  function addSwing(){
    return ++swings;
  }

  // Public Methods
  this.swingSword = function(){
    cloak( false );
    return addSwing();
  };

  this.cloak = function(value){
    return value != null ?
      cloaked = value :
      cloaked;
  };
}}
```

```
var ninja = new Ninja();

assert( !ninja.cloak(), "We start out uncloaked." );
assert( ninja.swings == 0, "And with no sword swings." );

assert( ninja.cloak( true ),
  "Verify that the cloaking worked correctly." );
assert( ninja.swingSword() == 1, "Swing the sword, once." );
assert( !ninja.cloak(),
  "The ninja became uncloaked with the sword swing." );
```

A couple points to consider about the style of the code in Listing 8.9:

- There is an explicit difference between how private and public instance data are defined (as is the case with most object-oriented code).
- There is absolutely no difference in how private data is accessed, from public data. All data is accessed in the same way: by its name. This is made possible by the use of `with(this){}` at the top of the code - forcing all public properties to become local variables.
- Methods are defined similarly to variables (publicly and privately). Public methods must include the `this.` prefix when assigning - but are accessed in the same ways as private methods.

The ability to access properties and methods using a unified naming convention (as opposed to some being through direct variable access and others through the instance object) is quite valuable.

## 8.4 Testing

When testing pieces of functionality in a test suite there's a couple things that you end up having to watch out for. The primary of which is the synchronization between the assertion methods and the test case currently being run. Typically this isn't a problem but it becomes especially troublesome when you begin dealing with asynchronous tests. A common solution to counteract this is to create a central tracking object for each test run. The test runner used by the Prototype and Scriptaculous libraries follows this model - providing this central object as the context to each test run. The object, itself, contains all the needed assertion methods (easily collecting the results back to the central location). You can see an example of this in Listing 8.10.

**Listing 8.10: An example of a test from the Scriptaculous test suite.**

```
new Test.Unit.Runner({
  testSliderBasics: function(){with(this){
    var slider = new Control.Slider('handle1', 'track1');
    assertInstanceOf(Control.Slider, slider);
    assertEqual('horizontal', slider.axis);
```

```
      assertEqual(false, slider.disabled);
      assertEqual(0, slider.value);
      slider.dispose();
    }},
    // ...
  });
```

You'll note the use of `with(this){}` in the above test run. The instance variable contains all the assertion methods (`assertInstanceOf`, `assertEqual`, etc.). The above method calls could've, also, been written explicitly as `this.assertEqual` - but by using `with(this){}` to introduce the methods that we wish to use we can get an extra level of simplicity in our code.

## 8.5 Templating

The last - and likely most compelling - example of using with(){} exists within a simplified templating system. A couple goals for a templating system are usually as follows:

- There should be a way to both run embedded code and print out data.
- There should be a means of caching the compiled templates
- and, perhaps, most importantly: It should be simple to easily access mapped data.

The last point is where `with(){}` becomes especially useful. To begin to understand how this is possible, let's look at the templating code in Listing 8.11.

**Listing 8.11: A simple templating solution using with(){} for simplified templates.**

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

    this.tmpl = function tmpl(str, data){
      // Figure out if we're getting a template, or if we need to
      // load the template - and be sure to cache the result.
      var fn = !/\W/.test(str) ?
        cache[str] = cache[str] ||
          tmpl(document.getElementById(str).innerHTML) :

        // Generate a reusable function that will serve as a template
        // generator (and which will be cached).
        new Function("obj",
          "var p=[],print=function(){p.push.apply(p,arguments);};" +

          // Introduce the data as local variables using with(){}
          "with(obj){p.push('" +

          // Convert the template into pure JavaScript
          str
            .replace(/[\r\t\n]/g, " ")
            .split("<%").join("\t")
            .replace(/((^|%>)[^\t]*)'/g, "$1\r")
          .replace(/\t=(.*?)%>/g, "',$1,'")
          .split("\t").join("');")
          .split("%>").join("p.push('")
          .split("\r").join("\\'")
      + "');}return p.join('');");

    // Provide some basic currying to the user
    return data ? fn( data ) : fn;
  };
})();

assert( tmpl("Hello, <%= name %>!", {name: "world"}) ==
  "Hello, world!", "Do simple variable inclusion." );

var hello =  tmpl("Hello, <%= name %>!");
assert( hello({name: "world"}) == "Hello, world!",
  "Use a pre-compiled template." );
```

This templating system provides a quick-and dirty solution to simple variable substitution. By giving the user the ability to pass in an object (containing the names and values of template variables that they want to populate) in conjunction with an easy means of accessing the variables, the result is a simple, reusable, system. This is made possible largely in part due to the existence of the `with(){}` statement.

By introducing the data object all of its properties immediately become first-class accessible within the template - which makes for some simple, re-usable, template writing. The templating system works by converting the provided template strings into an array of values - eventually concatenating them together. The individual statements, like `<%= name %>` are then translated into the more palatable `, name,` - folding them inline into the array construction process. The result is a template construction system that is blindingly fast and efficient.

Additionally, all of these templates are generated dynamically (out of necessity, since inline code is allowed to be executed). In order to facilitate re-use of the generated templates we can place all of the constructed template code inside a new `Function(...)` - which will give us a template function that we can actively plug our needed data into.

A good question should now be: How do we use this templating system in a practical situation, especially within an HTML document? We can see this in Listing 8.12.

**Listing 8.12: Using our templating system to generate HTML.**

```
<html>
<head>
  <script type="text/tmpl" id="colors">
    <p>Here's a list of <%= items.length %> items:</p>
    <ul>
      <% for (var i=0; i < items.length; i++) { %>
        <li style='color:<%= colors[i % colors.length] %>'>
          <%= items[i] %></li>
      <% } %>
    </ul>
    and here's another...
  </script>
  <script type="text/tmpl" id="colors2">
    <p>Here's a list of <%= items.length %> items:</p>
    <ul>
      <% for (var i=0; i < items.length; i++) {
        print("<li style='color:", colors[i % colors.length], "'>",
          items[i], "</li>");
      } %>
    </ul>
  </script>
  <script src="tmpl.js"></script>
  <script>
    var colorsArray = ['red', 'green', 'blue', 'orange'];

    var items = [];
    for ( var i = 0; i < 10000; i++ )
      items.push( "test" );

    function replaceContent(name) {
      document.getElementById('content').innerHTML =
        tmpl(name, {colors: colorsArray, items: items});
    }
  </script>
</head>
<body>
  <input type="button" value="Run Colors"
    onclick="replaceContent('colors')">
  <input type="button" value="Run Colors2"
    onclick="replaceContent('colors2')">
  <p id="content">Replaced Content will go here</p>
</body>
</html>
```

Listing 8.12 shows a couple examples of functionality that's possible with this templating system, specifically the ability to run inline JavaScript in the templates, a simple print function, and the ability to embed the templates inline in HTML.

To be able to run full JavaScript code inline we break out of the array construction process and defer to the user's code. For example, the final result of the loop in the above colors template would look something like this:

```
p.push('<p>Here's a list of', items.length, ' items:</p> <ul>');
for (var i=0; i < items.length; i++) {
p.push('<li style=\'color', colors[i % colors.length], '\'>',
items[i], '</li>');
}
p.push('</ul> and here's another...');
```

The `print()` method plays into this nicely, as well, being just a simple wrapper pointing back to `p.push()`.

Finally, the full templating system is pulled together with the use of embedded templates. There's a great loophole provided by modern browsers and search engines: <script/> tags that specify a `type=""` that they don't understand are completely ignored. This means that we can specify scripts that contain our templates, given them a type of "text/tmpl", along with a unique ID, and use our system to extract the templates again, later.

The total result of this templating system is one that is easy to use (due, in large part, to the abilities of `with(){}`), fast, and cacheable: a sure win for development.

## 8.6 Summary

If anything has become obvious, over the course of this chapter, is that the primary goal of `with(){}` is to make complex code simple: simplifying access to namespaced code, improving the usability of test suites, building usable templating utilities, and even improving the use of existing JavaScript libraries. Discretion should be applied when using it, but having a good understanding of how it works can only lead to better-quality code.

# 9
# Code evaluation

Covered in this chapter:

- We examine how code evaluation works.
- Different techniques for evaluating code.
- How to be use evaluation in your applications.

JavaScript includes the ability to dynamically execute pieces of code at runtime. Code evaluation is simultaneously the most advanced and most frequently misused feature of JavaScript. Understanding the situations in which it can and should be used (along with the best techniques for using it) can give you a definite advantage when creating advanced applications.

## 9.1 Code Evaluation

Within JavaScript there are a number of different ways to evaluate code. Each have their own advantages and disadvantages and should be chosen carefully based upon the context in which they are to be used.

### 9.1.1 eval()

The `eval()` function is the most commonly used means of evaluating code. The most consistent means of accessing it is as a global function. The eval function executes all code passed in to it, within the current context, returning the result of the last evaluated expression, as in Listing 9.1.

**Listing 9.1: A basic example of the `eval` function executing and returning a result.**

```
assert( eval("5 + 5") === 10,
  "The code is evaluated and result returned " );
assert( eval("var t = 5;") === undefined, "No result returned." );
assert( t === 5, "But the t variable now exists." );
```

```
(function(){
  eval("var t = 6;");
  assert( t === 6, "eval is executed within the current scope." );
})();

assert( t === 5,
  "Verify that the scope execution was done correctly." );
```

The simplest way to determine if a result will be returned from eval is to pretend that the last expression is wrapped in parentheses and still syntactically correct. For example, the code in Listing 9.2 will return values when wrapped in parentheses.

**Listing 9.2: Examples of values returned from eval.**

```
assert( (true) === true,
  "Variables, objects, and primitives all work." );
assert( (t = 5), === 5, "Assignment returns the assigned value." );
assert( typeof (function(){}) === "function",
  "A function is returned." );
```

It should be noted that anything that isn't a simple variable, primitive, or assignment will actually need to be wrapped in a parentheses in order for the correct value to be returned, for example in Listing 9.3.

**Listing 9.3: Getting a value from an evaluated object.**

```
var obj = eval("({name:'Ninja'})");
assert( obj.name === "Ninja",
  "Objects require the extra parentheses." );

var fn = eval("(function(){return 'Ninja';})");
assert( fn() === "Ninja", "As do functions." );
```

Now the last one is interesting because is technically should work that simply (and does in Firefox, Safari, and Opera) however Internet Explorer has a problem executing the function with that particular syntax. Instead we are forced to use some boolean-expression trickery to get the function to be generated correctly.

For example Listing 9.4 shows a technique found in jQuery to create a function of that nature.

**Listing 9.4: Dynamically creating a function with `eval()`.**

```
var fn = eval("false||function(){return true;}");
assert( fn() === true,
  "Verify that the function was created correctly." );
```

Now one might wonder why we would even want to create a function in this manner in the first place. One important point is that all code executed by the eval is done so within the current scope and, thus, it has all the properties of it as well - including any enclosed variables. Thus, using the above code, we can dynamically create a function that has a closure to all the variables within the current scope and all the externally enclosed variables as well.

Of course, if we don't need that additional closure, there's a proper alternative that we can make use of.

### 9.1.2 new Function

All functions in JavaScript are an instance of `Function`. Traditionally the `function name(...){...}` syntax is used to define a new function however we can go a step further and instantiate functions directly using new Function, like in Listing 9.5.

**Listing 9.5: Dynamically create a function using new Function.**

```
var add = new Function("a", "b", "return a + b;");
assert( add(3, 4) === 7, "Function is created and works." );
```

The last argument to `new Function` is always the code that will be contained within the function. Any arguments that go before it will represent the name of the arguments passed in to the function itself.

It's important to remember that no closures are created when functions are created in this manner. This can be a good thing if you don't want to incur any of the overhead associated with possible variable lookups.

It should be noted that code can also be evaluated is asynchronously, using timers. Normally one would pass in a function reference to a timer (which is the recommended behavior) however `setTimeout` and `setInterval` also accept strings which can be evaluated. It's very rare that you would need to use this behavior (it's roughly equivalent to using `new Function`) and it's essentially unused at this point.

### 9.1.3 Evaluate in the global scope

There's one problem that is frequently encountered, when working with eval, and it's the desire to evaluate a piece of code within the global scope. For example having the following code execute and become global variables, like in Listing 9.6.

**Listing 9.6: Attempting to evaluate code into a global variable.**

```
(function(){
  eval("var test = 5;");
})();

assert( typeof test === "undefined",
  "Variable is confined to temporary enclosures." );
```

However there is one trick that we can use in modern browsers to achieve an acceptable result: Injecting a dynamic `<script/>` tag into the document with the script contents that we need to execute.

Andrea Giammarchi developed the specific technique for making this work properly cross-platform. An adaptation of his work can be found in Listing 9.7.

- Original Example: http://webreflection.blogspot.com/2007/08/global-scope-evaluation-and-dom.html

**Listing 9.7: An example implementation of evaluating code in the global scope.**

```
<script>
function globalEval( data ) {
  data = data.replace(/^\s*|\s*$/g, "");

  if ( data ) {
    var head = document.getElementsByTagName("head")[0] ||
        document.documentElement,
      script = document.createElement("script");

    script.type = "text/javascript";
    script.text = data;

    head.appendChild( script );
    head.removeChild( script );
  }
}

window.onload = function(){
  (function(){
    globalEval("var test = 5;");
  })();

  assert( test === 5, "The code is evaluated globally." );
};
</script>
```

The most common place for using this code is in dynamically executing code returned from a server. It's, almost always, a requirement that code of that nature be executed within the global scope (making the above a necessity).

### 9.1.4 Safe Code Evaluation

One question that frequently arrives is related to the safe execution of JavaScript code, namely: It is possible to safely execute untrusted JavaScript on your page without compromising the integrity of your site. Generally the answer to that question is: No. There are simply too many ways to skirt around any barriers put forth that can result in code getting access to information that it's not supposed to.

There is one project, though, produced by Google called Caja. It's an attempt to create a translator for JavaScript that converts JavaScript into a safer form (and immune to malicious attacks).

- Google Caja: http://code.google.com/p/google-caja/

For example, look at Listing 9.8 and the (relatively simple) JavaScript code and the final Caja code that it gets converted into.

**Listing 9.8: The result of a Caja conversion process.**

```
// Original Code:
var test = true;
(function(){ var foo = 5; })();
Function.prototype.toString = function(){};

// Cajoled Code:
{
___.loadModule(function (___, IMPORTS___) {
{
  var Function = ___.readImport(IMPORTS___, 'Function');
  var x0___;
  var x1___;
  var x2___;
  var test = true;
  ___.asSimpleFunc(___.primFreeze(___.simpleFunc(function () {
    var foo = 5;
  })))();
  IMPORTS___[ 'yield' ] ((x0___ = (x2___ = Function,
    x2___.prototype_canRead___?
  x2___.prototype: ___.readPub(x2___, 'prototype')),
  x1___ = ___.primFreeze(___.simpleFunc(function () {})),
  x0___.toString_canSet___? (x0___.toString = x1___):
  ___.setPub(x0___, 'toString', x1___)));
}
});
}
```

Note the extensive use of built-in methods and properties to verify the integrity of the data (most of which is verified at runtime, however a number of possible loopholes are discovered during the conversion process).

For example the following snippet throws an immediate exception during the conversion process:

```
(function(){ return this; })();
// FATAL_ERROR :4+2 - 28: Implicit xo4a only allowed in warts mode
```

The desire for secure JavaScript stems from wanting to create mashups and safe AD embedding without worrying about your user's security becoming compromised. We're certainly going to see a lot of work in this realm and Google Caja is leading the way.

## 9.2 Function Decompilation

Most JavaScript implementations also provide access to decompiling already-compiled JavaScript code. Specifically this is done with the `.toString()` method of functions. A sample result will look something like the following in Listing 9.9.

**Listing 9.9: Decompiling a function into a string.**

```
function test(a){
  return a + a;
}

assert( test.toString() ===
  "function test(a) {\n    return a + a;\n}",
  "A decompiled function." );
```

This act of decompilation has a number of potential uses (especially in the area of macros and code rewriting) however one of the most popular is one presented in the Prototype JavaScript library. In there they decompile a function in order to read out its arguments (resulting in an array of named arguments). This is frequently used to introspect into functions to determine what sort of values they are expecting to receive.

Below is a simple rewrite of the code in Prototype to become more generic in nature, in Listing 9.10.

**Listing 9.10: A function for extracting the argument names for a function.**

```
function argumentNames(fn) {
  var found = /^[\s\(]*function[^(]*\(\s*([^)]*?)\s*\)/.exec(fn);
  return found && found[1] ? found[1].split(/,\s*/) : [];
}

assert( argumentNames( argumentNames )[0] === "fn",
  "Get the first argument name." );

function temp(){}

assert( argumentNames( temp ).length === 0, "No arguments found." );
```

There is one point to take into consideration when working with functions in this manner: It's possible that a browser may not support decompilation. There aren't many that don't, but one such browser is Opera Mini.

We can do two things to counteract this: One we can write our code in a resilient manner (like was done above with the `argumentNames` function) or two we can perform a quick test to see if decompilation works in the browser, like in Listing 9.11.

**Listing 9.11: Testing to see if function decompilation works as we expect it to.**

```
assert( /abc(.|\n)*xyz/.test(function(abc){xyz;}),
     "Decompilation works as we except it to." );
```

We used this test in the chapter on Function Prototypes when we built our simple Class implementation in order to determine if we could analyze methods for if they were accessing a super method.

## 9.3 Examples

There are a number of ways in which code evaluation can be used for both interesting, and practical, purposes. Examining some examples of evaluation is used can give us a better understanding of when and where we should try to use it in our code.

### 9.3.1 JSON

Probably the most wide-spread use of eval comes in the form of converting JSON strings into their JavaScript object representations. Since JSON data is simply a subset of the JavaScript language it is perfectly capable of being evaluated.

There is one minor gotcha that we have to take into consideration, however: We need to wrap our object input in parentheses in order for it to evaluate correctly. However, the result is quite simple, as seen in Listing 9.12.

**Listing 9.12: Converting a JSON string into a JavaScript object using the eval function.**

```
var json = '{"name":"Ninja"}';
var object = eval("(" + json + ")");

assert( object.name === "Ninja",
  "Verify that the object evaluated and came out." );
```

Not only does the `eval` function win in simplicity but it also wins in performance (running drastically faster than any manual JSON parser).

However there's a major caveat to using `eval()` for JSON parsing: Most of the time JSON data is coming from a remote server and blindly executing code from a remote server is rarely a good thing.

The most popular JSON script is written by Douglas Crockford (the original specifier of the JSON markup) in it he does some initial parsing of the JSON string in an attempt to prevent any malicious information from passing through. The full code can be found here:

- http://json.org/json2.js

The important pieces of the JSON script (that pre-parses the JSON string) can be found in Listing 9.13.

**Listing 9.13: The json2.js pre-parsing code (enforcing the integrity of incoming JSON strings).**

```
var cx = /[\u0000\u00ad\u0600-\u0604\u070f\u17b4\u17b5\
  \u200c-\u200f\u2028-\u202f\u2060-\u206f\ufeff\ufff0-\uffff]/g;

cx.lastIndex = 0;
```

```
if (cx.test(text)) {
  text = text.replace(cx, function (a) {
    return '\\u' + ('0000' +
      (+(a.charCodeAt(0))).toString(16)).slice(-4);
  });
}

if (/^[\],:{}\s]*$/.test(
  text.replace(/\\(?:["\\\/bfnrt]|u[0-9a-fA-F]{4})/g, '@')
  .replace(/"[^"\\\n\r]*"|true|false|null|-?\d+\
    (?:\.\d*)?(?:[eE][+\-]?\d+)?/g, ']')
  .replace(/(?:^|:|,)(?:\s*\[)+/g, ''))) {

  j = eval('(' + text + ')');
}
```

The pre-parsing works in a couple stages, in the words on Douglas Crockford:

- First we replace certain Unicode characters with escape sequences. JavaScript handles many characters incorrectly, either silently deleting them, or treating them as line endings.
- Next we run the text against regular expressions that look for non-JSON patterns. We are especially concerned with '()' and 'new' because they can cause invocation, and '=' because it can cause mutation. But just to be safe, we want to reject all unexpected forms.
- We then split the second stage into 4 regexp operations in order to work around crippling inefficiencies in IE's and Safari's regexp engines. First we replace the JSON backslash pairs with '@' (a non-JSON character). Second, we replace all simple value tokens with ']' characters. Third, we delete all open brackets that follow a colon or comma or that begin the text. Finally, we look to see that the remaining characters are only whitespace or ']' or ',' or ':' or '{' or '}'. If that is so, then the text is safe for eval.
- Finally we use the eval function to compile the text into a JavaScript structure. The '{' operator is subject to a syntactic ambiguity in JavaScript: it can begin a block or an object literal. We wrap the text in parens to eliminate the ambiguity.

Once all the steps have been taken to protect our incoming JSON strings eval then becomes the preferred means of converting JSON data into JavaScript objects.

### *9.3.2 Importing Namespace*

Importing namespaced code into the current context can be a challenging problem - especially considering that there is no simple way to do it in JavaScript. Most of the time we have to perform an action similar to the following:

```
var DOM = base2.DOM;
var JSON = base2.JSON;
// etc.
```

The base2 library provides a very interesting solution to the problem of importing namespaces into the current context. Since there is no way to automate this problem, traditionally, we can make use of eval in order to make the above trivial to implement.

Whenever a new class or module is added to a package a string of executable code is constructed which can be evaluated to introduce the function into the context, like in Listing 9.14.

**Listing 9.14: Examining how the base2 namespace property works.**

```
base2.namespace ==
  "var Base=base2.Base;var Package=base2.Package;" +
  "var Abstract=base2.Abstract;var Module=base2.Module;" +
  "var Enumerable=base2.Enumerable;var Map=base2.Map;" +
  "var Collection=base2.Collection;var RegGrp=base2.RegGrp;" +
  "var Undefined=base2.Undefined;var Null=base2.Null;" +
  "var This=base2.This;var True=base2.True;var False=base2.False;" +
  "var assignID=base2.assignID;var detect=base2.detect;" +
  "var global=base2.global;var lang=base2.lang;" +
  "var JavaScript=base2.JavaScript;var JST=base2.JST;" +
  "var JSON=base2.JSON;var IO=base2.IO;var MiniWeb=base2.MiniWeb;" +
  "var DOM=base2.DOM;var JSB=base2.JSB;var code=base2.code;" +
  "var doc=base2.doc;"

assert( typeof DOM === "undefined", "The DOM object doesn't exist." );

eval( base2.namespace );

assert( typeof DOM === "object",
  "And now the namespace is imported." );
assert( typeof Collection === "object",
  "Verifying the namespace import." );
```

and we can see how this list is constructed in the base2 Package code:

```
this.namespace = format("var %1=%2;", this.name,
  String2.slice(this, 1, -1));
```

This is a very ingenious way of tackling this complex problem. Albeit it isn't, necessarily, done in the most graceful manner but until implementations of future versions of JavaScript exist we'll have to make do with what we have.

### 9.3.3 Compression and Obfuscation

A popular piece of JavaScript software is that of Dean Edwards' Packer. This particular script compresses JavaScript code, providing a resulting JavaScript file that is significantly smaller while still being capable of executing and self-extracting itself to run again.

- Dean Edwards' Packer: http://dean.edwards.name/packer/

The result is an encoded string which is converted into a string of JavaScript code and executed, using `eval()`. The result typically looks something like the code in Listing 9.15.

**Listing 9.15: An example of code compressed using Packer.**

```
eval(function(p,a,c,k,e,r){e=function(c){return(c<a?'':e(
     parseInt(c/a)))+((c=c%a)>35?String.fromCharCode(c+29):
     c.toString(36))};if(!''.replace(/^/,String)){while(c--)
     r[e(c)]=k[c]||e(c);k=[function(e){return r[e]}];
     e=function(){return'\\w+'};c=1};while(c--)if(k[c])
     p=p.replace(new RegExp('\\b'+e(c)+'\\b','g'),k[c]);
     return p}(' // ... long string ...
```

While this technique is quite interesting there are some fundamental flaws with it: Namely that the overhead of uncompressing the script every time it loads is quite costly. When distributing a piece of JavaScript code it's traditional to think that the smallest (byte-size) code will download and load the fastest. This is not true - and is a fascinating result of this survey. Looking at the speed of loading jQuery in three forms: normal, minified (using Yahoo! Minifier - removing whitespace and other simple tricks), and packed (using Dean Edwards' Packer - massive rewriting and uncompression using eval). By order of file size, packed is the smallest, then minifed, then normal. However, the packed version has an overhead: It must be uncompressed, on the client-side, using a JavaScript decompression algorithm. This unpacking has a tangible cost in load time. This means, in the end, that using a minified version of the code is much faster than the packed one - even though its file size is quite larger.

It breaks down to a simple formula:

```
Total_Speed = Time_to_Download + Time_to_Evaluate
```

You can see the result of a study that was done on the jQuery library, analyzing thousands of file downloads:

- http://ejohn.org/blog/library-loading-speed/

The results of which can be seen in Table 9.1.

Table 9.1: A comparison of transfer speeds, in various formats, of the jQuery JavaScript library.

| Minified | Time Avg | # of Samples |
|---|---|---|
| minified | 519.7214 | 12611 |
| packed | 591.6636 | 12606 |
| normal | 645.4818 | 12589 |

This isn't to say that using code from Packer is worthless (if you're shooting for performance then it may be) but it's still quite valuable as a means of code obfuscation.

If nothing else Packer can serve as a good example of using `eval()` gratuitously - where superior alternatives exist.

## 9.3.4 Dynamic Code Rewriting

Since we have the ability to decompile existing JavaScript functions using a function's `.toString()` method that means that we can create new functions, extracting the old function's contents, that take on new functionality.

One case where this has been done is in the unit testing library Screw.Unit.

- Screw.Unit: http://github.com/nkallen/screw-unit/tree/master

Screw.Unit takes the existing test functions and dynamically re-writes their contents to use the functions provided by the library. For example, Listing 9.16 shows what a typical Screw.Unit test looks like.

**Listing 9.16: A sample Screw.Unit test.**

```
describe("Matchers", function() {
  it("invokes the provided matcher on a call to expect", function() {
    expect(true).to(equal, true);
    expect(true).to_not(equal, false);
  });
});
```

Especially note the describe, it, and expect methods - all of which don't exist in the global scope. To counteract this Screw.Unit rewrites this code on the fly to wrap all the functions with multiple `with(){}` statements - injecting the function internals with the functions that it needs in order to execute, as seen in Listing 9.17.

**Listing 9.17: The code used by Screw.Unit to introduce new methods into an existing function.**

```
var contents = fn.toString().match(/^[^{]*{((.*\n*)*)}/m)[1];
var fn = new Function("matchers", "specifications",
  "with (specifications) { with (matchers) { " + contents + " } }"
);
```

```
fn.call(this, Screw.Matchers, Screw.Specifications);
```

This is a case of using code evaluation to provide a simpler user experience to the end-user without having to negative actions (such as introducing many variables into the global scope).

### 9.3.5 *Aspect-Oriented Script Tags*

We've previously discussed using script tags that have invalid type attributes as a means of including new pieces of data in the page that you don't want the browser to touch. We can take that concept one step further and use it to enhance existing JavaScript.

Let's say that we created a new script type called "onload" - it contained normal JavaScript code but was only executed whenever that page was already loaded (as opposed to being normally executed inline). The script in Listing 9.18 achieves that goal.

**Listing 9.18: Creating a script tag type that executes only after the page has already loaded.**

```
<script>
window.onload = function(){
  var scripts = document.getElementsByTagName("script");
  for ( var i = 0; i < scripts.length; i++ ) {
    if ( scripts[i].type == "onload" ) {
      // You may want to use a globalEval implementation here
      eval( scripts[i].innerHTML );
    }
  }
};
</script>
<script type="onload">
  ok(true, "I will only execute after the page has loaded.");
</script>
```

Obviously this technique could be extended beyond this simple example: Executing scripts on user interaction, when the DOM is ready to be manipulated, or even relatively based upon adjacent elements.

### 9.3.6 *Meta-Languages*

The most poignant example of the power of code evaluation can be seen in the implementation of other programming languages on top of the JavaScript language: dynamically converting these languages into JavaScript source and evaluating them.

There have been two such language conversions that have been especially interesting.

#### PROCESSING.JS

This was a port of the Processing Visualization Language (which is typically implemented using Java) to JavaScript, running on the HTML 5 Canvas element - by John Resig.

- Processing Visualization Language: http://processing.org/
- Processing.js: http://ejohn.org/blog/processingjs/

The result is a full programming language that you can usual to manipulate the visual display of a drawing area. Arguably Processing is particularly well suited towards it making it an effective port.

An example of Processing.js code can be seen in Listing 9.19.

**Listing 9.19: An example of a class defined using Processing.**

```
<script type="application/processing">
class SpinSpots extends Spin {
  float dim;
  SpinSpots(float x, float y, float s, float d) {
    super(x, y, s);
    dim = d;
  }
  void display() {
    noStroke();
    pushMatrix();
    translate(x, y);
    angle += speed;
    rotate(angle);
    ellipse(-dim/2, 0, dim, dim);
    ellipse(dim/2, 0, dim, dim);
    popMatrix();
  }
}
</script>
```

The above Processing code is then converted into the following JavaScript code and executed using an `eval()`, the resulting code can be seeing in Listing 9.20.

**Listing 9.20: The end JavaScript result of the above Processing code.**

```
function SpinSpots() {with(this){
  var __self=this;function superMethod(){
  extendClass(__self,arguments,Spin);
  this.dim = 0;
  extendClass(this, Spin);
  addMethod(this, 'display', function() {
    noStroke();
    pushMatrix();
    translate(x, y);
    angle += speed;
    rotate(angle);
    ellipse(-dim/2, 0, dim, dim);
    ellipse(dim/2, 0, dim, dim);
    popMatrix();
  });
  if ( arguments.length == 4 ) {
    var x = arguments[0];
    var y = arguments[1];
    var s = arguments[2];
    var d = arguments[3];
    superMethod(x, y, s);
    dim = d;
    }
  }}
```

Using the Processing language you gain a few immediate benefits over using JavaScript alone:

- You get the benefits of Processing advanced features (Classes, Inheritance)
- You get Processing's simple drawing API
- You get all of the existing documentation and demos on Processing

The important point, though: All of this is occurring in pure JavaScript - all made possible because of code evaluation.

### OBJECTIVE-J

The second major project put forth was a port of the Objective-C programming language to JavaScript, called Objective-J, by the company 280 North for the product 280 Slides (an online slideshow builder).

- 280 Slides (uses Objective-J): http://280slides.com/

The 280 North team had extensive experience developing applications for OS X (which are primarily written in Objective-C). To create a more-productive environment to work in they ported the Objective-C language to JavaScript, providing a thin layer over the JavaScript language. An important point, though: They allow JavaScript code mixed in with their Objective-C code; the result of which they dub Objective-J, an example of which is in Listing 9.21.

**Listing 9.21: An example of some Objective-J code.**

```
// DocumentController.j
// Editor
//
// Created by Francisco Tolmasky.
// Copyright 2005 - 2008, 280 North, Inc. All rights reserved.

import <AppKit/CPDocumentController.j>
import "OpenPanel.j"
import "Themes.j"
import "ThemePanel.j"
import "WelcomePanel.j"

@implementation DocumentController : CPDocumentController
{
    BOOL    _applicationHasFinishedLaunching;
}

- (void)applicationDidFinishLaunching:(CPNotification)aNotification
{
        [CPApp runModalForWindow:[[WelcomePanel alloc] init]];
        _applicationHasFinishedLaunching = YES;
    }

    - (void)newDocument:(id)aSender
    {
        if (!_applicationHasFinishedLaunching)
            return [super newDocument:aSender];

        [[ThemePanel sharedThemePanel]
            beginWithInitialSelectedSlideMaster:SaganThemeSlideMaster
              modalDelegate:self
                didEndSelector:@selector(themePanel:didEndWithReturnCode:)
                  contextInfo:YES];
    }

    - (void)themePanel:(ThemePanel)aThemePanel
      didEndWithReturnCode:(unsigned)aReturnCode
    {
        if (aReturnCode == CPCancelButton)
            return;

        var documents = [self documents],
            count = [documents count];

        while (count--)
            [self removeDocument:documents[0]];

        [super newDocument:self];
    }
```

In their parsing application (which is written in JavaScript and converts the Objective-J code on-the-fly at runtime) they use light expressions to match and handle the Objective-C syntax, without disrupting the existing JavaScript. The result is a string of JavaScript code which is evaluated.

While this implementation has less far-reaching benefits (it's a specific hybrid language that can only be used within this context). Its potential benefits to users who are already familiar with Objective-C, but wish to explore web programming, will become greatly endowed.

## 9.4 Summary

In this chapter we've looked at the fundamentals of code evaluation in JavaScript. Exploring a number of different techniques for evaluating code and looking at what makes each method well suited to its conditions. We then looked at a variety of use cases for code evaluation along with some examples of
how it has been best used. While potential for misuse will always remain, the incredible power that comes with harnessing code evaluation correctly is sure to give you an excellent tool to wield in your daily development.

# 10 Strategies for cross-browser code

In this chapter:

- A sound strategy for developing reusable JavaScript code.
- Analyzing the types of issues that need to be tackled.
- How to tackle issues in a smart way.

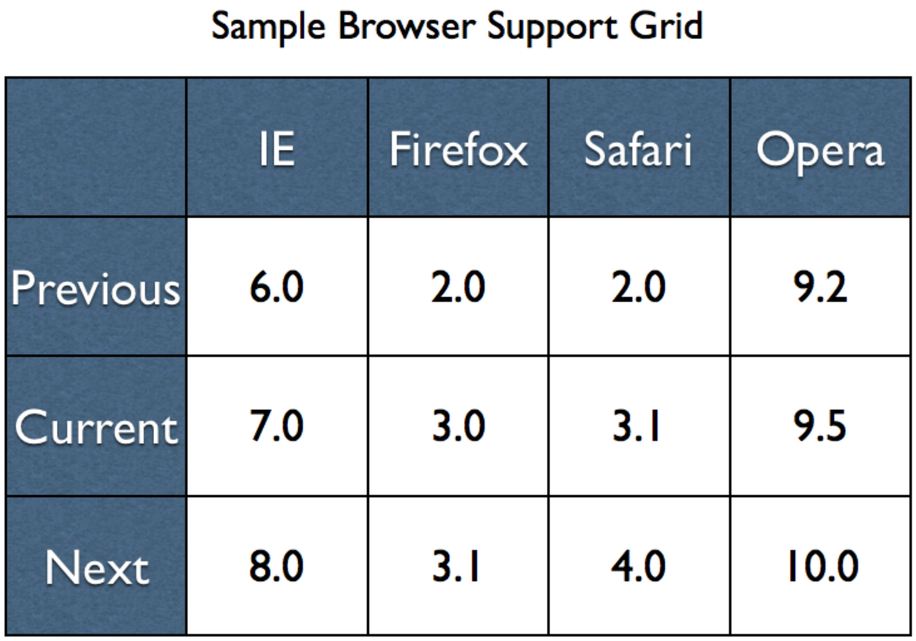
When developing capable cross-browser code there is a wide field of points, and counter-points, that need to be taken into consideration. Everything from the basic level of development to planning for future browser releases and web pages that have yet to be encountered. This is certainly a non-trivial task – the result of which must be balanced in a way that best works for your development methodologies. It doesn't matter if you wish your site to work in every browser that ever existed - and ever will exist - if you only have a finite amount of development resources. You must plan your resources appropriately and carefully; getting the maximum result from your effort.

## 10.1 Picking Your Core Browsers

Your primary concern should be taking into account which browsers will be the primary target on which your development will take place. As with virtually any aspect of web development you need to end up picking a number of browsers on which you'll want your users to have an optimal experience. When you choose to support a browser you are typically promising a couple things:

1. That you'll actively test against that browser with your test suite.
2. You'll fix bugs and regressions associated with that browser.
3. That the browser will have comparably reasonable performance.

For example, most JavaScript libraries end up supporting about 12 browser: The previous release, the current release, and the upcoming release of Internet Explorer, Firefox, Safari, and Opera. The choice of the JavaScript library is generally independent of the actual time required to support these browsers or their associated market share (as they are trying to, simultaneously, match actual market share and developer market share - which are often quite the opposite) - however this may not be the case in your development, an example of which is seen in Figure 10.1.



Sample Browser Support Grid

| | IE | Firefox | Safari | Opera |
|---|---|---|---|---|
| Previous | 6.0 | 2.0 | 2.0 | 9.2 |
| Current | 7.0 | 3.0 | 3.1 | 9.5 |
| Next | 8.0 | 3.1 | 4.0 | 10.0 |

Figure 10.1: An example break-down of browser support for a JavaScript library.

A JavaScript library (or, really, any piece of reusable JavaScript code) is in a unique situation: It must be willing to encounter a large number of situations in which their code may not work correctly. Thus it is optimal to try and work on as many platforms as possible, simultaneously, without sacrificing quality or efficiency.

In order to understand what we're up against we must break down the different types of situations that JavaScript code will encounter and then examine the best ways to write preventative code which will assuage any potential problems that might occur.

## 10.2 Development Concerns

There are about five major points on which your reusable JavaScript code will be challenged. You'll want to balance how much time you spend on each point with how much benefit you'll receive in the end result. Is an extra 40 hours of development time worth better support for Netscape Navigator 4? Ultimately it's a question that you'll have to answer yourself, but it's one that can be determined rather concretely after some simple analysis.

There's one axiom that can be used when developing: Remember the past, consider the future – test the present.

When attempting to develop reusable JavaScript code you must take all points into consideration: You have to pay primary attention to the most-popular browsers that exist right now (as stated previously), then you have to take into concern what changes are upcoming in the next versions of the browsers, and then try to keep good compatibility with old browser versions (supporting as many features as you can without being susceptible to becoming unusable), summarized in Figure 10.2.

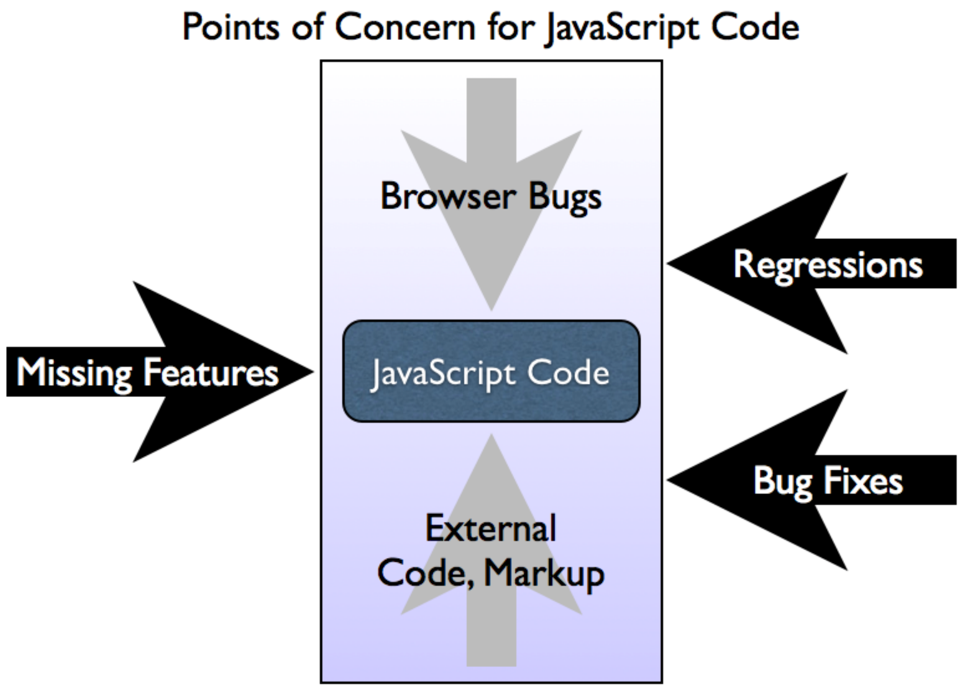


Figure 10.2: A breakdown of the different points that need to be considered when developing reusable JavaScript code.

Let's break down the various concerns so that we can have a better understanding of what we're up against.

### 10.2.1 Browser Bugs

The primary concern of reusable JavaScript code development should be in handling the browser bugs and API inequality associated with the set of browsers that you support. This means that any features that you provide in your application should be completely - and verifiably - usable in all of those browsers.

The solution to this is quite straight forward, albeit challenging: You need a comprehensive suite of tests to cover the common (and fringe) use cases of your code. With good coverage you can feel safe in knowing that the code that you develop will, both, work in the most commonly-used browsers (hopefully covering some 95-99% of your total browser market share) and in the next versions of those browsers (giving you that guaranteed percentage many years into the future).

The trick becomes, however: How can you implement fixes for current browser bugs in a way that is resistant to fixes implemented in future versions of the browser? Assuming that a browser will forever contain a bug is quite foolhardy and is a dangerous development strategy. It's best to use a form of Feature Simulation (as we discuss later on) to determine if a bug has been fixed (and that a browser feature works as you expect it to).

Where the line blurs, though, is in determining what exactly a browser bug is. If a browser implements a private API for handling a piece of functionality can you safely assume that the API will be maintained?

We'll look at a number of these issues when we examine how to handle regressions and bug fixes coming in from future browser versions.

### 10.2.2 External Code and Markup

A tricky point in developing reusable JavaScript code is in the code and markup that surrounds it. If you're expecting your code to work on any web site then you need to expect it to be able to handle the random code associated with it. This has two meanings: Your code must strive to be able to survive poorly written external code and not affect any external code.

This point of concern depends a lot upon which environments you expect your code to be used. For example, if you are using your reusable code on a limited number of web sites, but targeting a large number of browsers, it's probably best to worry less about external code (as you have the power and control to fix it yourself). However if you're trying to develop code that'll have the broadest applicability you'll need to make sure that your code is robust and effective.

To keep your code from affecting other pieces of code on the page it's best to practice encapsulation. Keep an incredibly-small footprint when introducing your code into a page (usually a single global variable). This includes modifying existing variables, function prototypes, or even DOM elements. Any place that your code modifies is a potential area for collision and confusion.

You can even extend these practices further and introduce other libraries in your test suite - making sure that their code doesn't affect how yours performs, and vice versa.

Secondly, when expecting outside code or markup you need to assume the worst (that they will be modifying function prototypes and existing DOM element methods, etc.). There aren't many steadfast rules when dealing when situations of this nature, but here are a couple tips:

hasOwnProperty: When looping through the properties of an object you'll want to verify that the properties that you're expecting are from the object itself and not introduced by an extension to Object.prototype. Thankfully the number of scripts that use this technique is very small but the harm is quite large. You can counter this by using the method `hasOwnProperty` to determine if the property is coming from this object or another one, as shown in Listing 10.1.

**Listing 10.1: Using hasOwnProperty to build resilient object looping.**

```
Object.prototype.otherKey = "otherValue";

var object = { key: "value" };
for ( var prop in object ) {
  if ( object.hasOwnProperty( prop ) ) {
    assert( prop, "key",
  "There should only be one iterated property." );
  }
}
```

Greedy IDs: Internet Explorer and Opera both have the concept of "greedy DOM IDs" - in which elements become properties of other objects based upon their ID names. Listing 10.2 shows two examples of this nasty trick in action.

**Listing 10.2: A simple example of markup that's capable of causing great confusion in development.**

```
<form id="form">
  <input type="text" id="length"/>
  <input type="submit" id="submit"/>
</form>
```

If you were to call `document.getElementsByTagName("input").length` you'd find that the length property (which should contain a number) would now contain a reference to the input element with the ID of length. Additionally if you were to call `document.getElementById("form").submit()` that would no longer work as the submit method will have be overwritten by the submit input element. This particular "feature" of the browser can cause numerous problems in your code and will have to be taken in to consideration when calling default properties in the above situations.

Order of Stylesheets: The best way to ensure that CSS rules (provided by stylesheets) are available when the JavaScript code on the page executes is to specifying the external stylesheets prior to including the external script files. Not doing so can cause unexpected results as the script attempts to access the incorrect style information. Unfortunately this isn't an issue that can be easily rectified with pure JavaScript and should be, instead, relegated to user documentation.

These are just some basic examples of how externalities can truly affect how your code works – frequently in a very unintentional manner. Most of the time these issues will pop up once users try to integrate your code into their sites - at which point you'll be able to diagnose the issue and build appropriate tests to handle them. It's unfortunate that there are no better solutions to handling these integration issues other than to take some smart first steps and to write your code in a defensive manner.

### *10.2.3 Missing Features*

For a large segment of the browsers that exist, but aren't in your list of actively supported browsers, they will most-likely be missing some key features that you need to power your code (or have enough bugs to make them un-targetable).

The fundamental problem, in this area, is that the harder you work to support more browsers the less your ultimate result will be worth. For example, if you worked incredibly hard and were able to bring Internet Explorer 4 support to your library what would the end result be? There would be so few users and the time that you spent to create that result would be so much as to be ludicrous. And that's saying nothing of determining if the CSS of your page would render in this situation, as well.

The strategy usually becomes, at this point: How can we get the most functionality delivered to the user while failing gracefully when we cannot. However there is one flaw in this, take this example: What if a browser is capable of initializing and hiding a number of pieces of navigation on a page (in hopes of creating a drop-down menu) but then the event-related code doesn't work. The result is a half-functional page, which helps no one.

The better strategy is to design your code to be as backwards-compatible as possible and then actively direct known failing browsers over to a tailored version of the page. Yahoo! adopts this strategy with their web sites, breaking down browsers with graded levels of support. After a certain amount of time they "blacklist" a browser (usually when it hits an infinitesimal market-share - like 0.05%) and direct users of that browser (based upon the detected user agent) to a pure-HTML version of the application (no CSS or JavaScript involved).

This means that their developers are able to target, and build, an optimal experience for the vast majority of their users (around 99%) while passing off antiquated browsers to a functional equivalent (albeit with a less-optimal experience).

The key points of this strategy:

- No assumptions are made about the user experience of old browsers. After a browser is no longer able to be tested (and has a negligible market-share) it is simply cut off and served with a simplified page.
- All users of current and past browsers are guaranteed to have a page that isn't broken.
- Future/unknown browsers are assumed to work.

With this strategy most of the extra development effort (beyond the currently-targeted browsers and platforms) is focused towards handling future browsers. This is a smart strategy as it will allow your applications to last longer with only minimal changes.

### *10.2.4 Bug Fixes*

When writing a piece of reusable JavaScript code you'll want to make sure that it's able to last for a long time. As with writing any aspect of a web site (CSS, HTML, etc.) it's undesirable to have to go back and change a broken web site (caused by a new browser release).

The most common form of web site breakage comes in when making assumptions about browser bugs. This can be generalized as: Specific hacks that are put in place to workaround bugs introduced by a browser, which break when the browsers fix the bugs in future releases. The issue is frequently circumvented by building pieces of feature simulation code (discussed in this chapter) instead of making assumptions about the browser.

The issue with handling browser bugs callously is two-fold:

1. First, your code will break if the browser ever resolves the issue.
2. Second, browser vendors become more inclined to not fix the bugs, for fear of causing web sites to break.

An interesting example of the above situation occurred during the development of Firefox 3. A change was introduced which forced DOM nodes to be adopted by a DOM document if they were going to be injected into them (in accordance with the DOM specification), like in Listing 10.3.

**Listing 10.3: Examples of having to use adoptNode before injecting a node into a new document.**

```
// Shouldn't work
var node = documentA.createElement("div");
documentB.documentElement.appendChild( node );

// Proper way
var node = documentA.createElement("div");
documentB.adoptNode( node );
documentB.documentElement.appendChild( node );
```

However, since there was a bug in Firefox (it allowed the first situation to work, when it shouldn't have) users wrote their code in a manner that depended on that code working. This forced Mozilla to rollback their change, for fear of breaking a number of web sites.

This brings up an important point concerning bugs, though: When determining if a piece of functionality is, potentially, a bug - go back to the specification. In the above case Internet Explorer was actually more forceful (throwing an exception if the node wasn't in the correct document - which was the correct behavior) but users just assumed that it was an error with Internet Explorer, in that case, and wrote conditional code to fall back. This caused a situation in which users were following the specification for only a subset of browsers and forcefully rejecting it in others.

A browser bug should be differentiated from an unspecified API. It's important to point back to browser specifications since those are the exact standards that browsers will be using in order to develop and improve their code. Whereas with an unspecified API the implementation could change at any point (especially if the implementation ever attempts to become specified - and changed in the process). In the case of inconsistency in unspecified APIs you should always test for your expected output, running additional cases of feature simulation. You should always be aware of future changes that could occur in these APIs as they become solidified.

Additionally, there's a distinction between bug fixes and API changes. Whereas bug fixes are easily foreseen (a browser will eventually fix the bugs with its implementation - even if it takes a long amount of time) API changes are much harder to spot. While you should always be cautious when using an unspecified API it is possible for a specified API to change. While this will rarely happen in a way that will massively break most web applications the result is effectively undetectable (unless, of course, you tested every single API that you ever touched - but the overhead incurred in such an action would be ludicrous). API changes in this manner should be handled like any other regression.

In summary: Blanketing a bug to a browser is very dangerous. Your web application will break, it's only a matter of time before it occurs. Using feature simulation to gracefully fall back to a working implementation is the only smart way to handle this issue.

### *10.2.5 Regressions*

Regressions are the hardest problem that you can encounter in reusable, sustainable, JavaScript development. These are bugs, or API changes, that browsers have introduced that caused your code to break in unpredicted ways.

There are some API changes that can be easily detected and handled. For example, if Internet Explorer introduces support for DOM event handlers (bound using addEventListener) simple object detection will be able to handle that change, as shown in Listing 10.4.

**Listing 10.4: A simple example of catching the implementation of a new API.**

```
function attachEvent( elem, type, handle ) {
  // bind event using proper DOM means
  if ( elem.addEventListener )
```

```
        elem.addEventListener(type, handle, false);

      // use the Internet Explorer API
      else if ( elem.attachEvent )
        elem.attachEvent("on" + type, handle);
    }
```

However most API changes aren't that easy to predict and neither are bugs. For this reason the best that you can hope for is to be diligent in your monitoring and testing in upcoming browser releases.

For example, in Internet Explorer 7 a basic XMLHttpRequest wrapper around the native ActiveX request object. This caused virtually all JavaScript libraries to default to using the XMLHttpRequest object to perform their Ajax requests (as it should - opting to use a standards-based API is nearly always optimal).

However, in Internet Explorer's implementation they broke the handling of requesting local files (a site loaded from the desktop could no longer request files using the XMLHttpRequest object). No one really caught this bug (or really could've pre-empted it) until it was too late, causing it to escape into the wild and breaking many pages in the process. The solution was to opt to use the ActiveX implementation primarily for local file requests.

Keeping close track of upcoming browser releases is absolutely the best way to avoid future regressions of this nature. It doesn't have to be a complete tax on your normal development cycle - it can be as simple as opening your applications in each of the upcoming browsers every couple weeks to make sure no major regressions have taken place.

You can get the upcoming browser releases from the following locations:

- Internet Explorer (their blog - but releases are announced here): http://blogs.msdn.com/ie/
- Firefox: http://ftp.mozilla.org/pub/mozilla.org/firefox/nightly/latest-trunk/
- WebKit (Safari): http://nightly.webkit.org/
- Opera: http://snapshot.opera.com/

Diligence is important in this respect. Since you can never be fully precogniscent of the bugs that will be introduced by a browser it is best to make sure that you stay on top of your code and avert any crises that may arrive.

Thankfully, browser vendors are doing a lot to make sure that regressions of this nature do not occur. Both Firefox and Opera have test suites from various JavaScript libraries integrated into their main browser test suite. This allows them to be sure that no future regressions will be introduced that affect those libraries directly. While this 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.

## 10.3 Implementing Cross-Browser Code

Knowing which issues to be aware of is only half the battle. Figuring out effective strategies for implementing cross-browser code is a whole other aspect to development. There exist a range of strategies that can be used. While not every strategy will work in every situation - the combination should provide good coverage for most concerns with a code base.

### 10.3.1 Safe Cross-Browser Fixes

The simplest, and safest, class of cross-browser fixes are those that will have no negative effects on other browsers whatsoever, while simultaneously using no forms of browser or feature detection.

These instances are, generally, rather rare - but it's a result that should always be strived for in your applications. To give an example, examine Listing 10.5.

**Listing 10.5: Preventing negative values to be set on CSS height and width properties.**

```
// ignore negative width and height values
if ( (key == 'width' || key == 'height') && parseFloat(value) < 0 )
  value = undefined;
```

This change, in jQuery, came about when working with Internet Explorer. The browser throws an exception when a negative value is set on height or width style properties. All other browsers simply ignore this input. The workaround that was introduced, shown in the above listing, was to simply ignore all negative values - in all browsers. This change prevented an exception from being thrown in Internet Explorer and had no effect on any other browser. This was a painless addition which provided a unified API to the user (throwing unexpected exceptions is never desired).

Another example of this simplification can be seen in jQuery in the attribute manipulation code is shown in Listing 10.6.

**Listing 10.6: Disallow attempts to change the type attribute on input elements in all browsers.**

```
if ( name == "type" && elem.nodeName.toLowerCase() == "input" &&
     elem.parentNode )
  throw "type attribute can't be changed";
```

Internet Explorer doesn't allow you to manipulate the type attribute of input elements that have already been inserted into the document. Attempts to do so result in an unpredictable exception being thrown. jQuery came to a middle-ground solution: It simply disallowed all attempts to manipulate the type attribute on injected input elements in all browsers equally (throwing an informational exception).

This change to the jQuery code base required no browser or feature detection - it was simply introduced as a means of unifying the API across all browsers. Certainly this particular feature addition is quite controversial - it actively limits the features of the library in browsers that are unaffected by this bug. The jQuery team weighed the decision carefully and decided that it was better to have a unified API that worked consistently than an API would break unexpectedly when developing cross-browser. It's very possible that you'll come across situations like this when developing your own reusable code bases.

The important thing to remember from these style of code changes: They provide a solution that works seamlessly across browsers with no browser or feature detection (effectively making them immune to changes going forward). One should always strive to use solutions that work in this manner - even if they are few and far-between.

### 10.3.2 Object Detection

Object detection is one of the most commonly used ways of writing cross-browser code. It's simple and generally quite effective. It works by simply determining if a certain object or object property exists (and, if so, assuming that it provides the functionality implies).
Most common object detection is used to toggle between multiple APIs that provide duplicate pieces of functionality. For example, like in the code shown in Listing 10.4 and Listing 10.7, object detection is frequently used to choose the appropriate event-binding APIs provided by the browser.

**Listing 10.7: Binding an event listener using the W3C DOM Events API or the Internet Explorer-specific API using object detection.**

```
function attachEvent( elem, type, handle ) {
  // bind event using proper DOM means
  if ( elem.addEventListener )
    elem.addEventListener(type, handle, false);

  // use the Internet Explorer API
  else if ( elem.attachEvent )
    elem.attachEvent("on" + type, handle);
}
```

In this case we look to see if a property exists named addEventListener and, if so, we assume that it's a function that we can execute and that it'll bind an event listener to that element. Note that we start with addEventListener (the method provided by the W3C DOM Events specification). This is intentional.

Whenever possible we should strive to default to the specified way of performing an action. As mentioned before this will help to make our code advance well into the future and encourage browser vendors to work towards the specified way of performing actions.

One of the most important uses of object detection is in detecting the basic features of a library, or application, and providing a fallback. As discussed previously you'll want to target your code to a specific set of browsers. The best way to do this is to figure out the APIs that the browser provides, that you need, and test for their existence.

Listing 10.8 has a basic example of detecting a number of browser features, using object detection, in order to determine if we should be providing a full application or a reduced fallback experience.

**Listing 10.8: An example of using object detection to provide a fallback experience for a browser.**

```
if ( typeof document !== "undefined" &&
     (document.addEventListener || document.attachEvent) &&
     document.getElementsByTagName && document.getElementById ) {
  // We have enough of an API to work with to build our application
} else {
  // Provide Fallback
}
```

What is done in the fallback is up to you. There are a couple of options:

1. You could do further object detection and figure out how to provide a reduce experience that still uses some JavaScript.
2. Simply opt to not execute any JavaScript (falling back to the HTML on the page).
3. Redirect a user to a plain version of the site (Google does this with GMail, for example).

Since object detection has very little overhead associated with it (it's just simple property/object lookups) and is relatively simplistic in its implementation, it makes for a good candidate to provide basic levels of fallbacks - both at the API and at the application level. It absolutely should serve as a good first line of defense in your reusable code authoring.

### 10.3.3 Feature Simulation

The final means that we have of preventing regressions - and the most effective means of detecting fixes to browser bugs - exists in the form of feature simulation. Contrary to object detection, which are just simple object/property lookups, feature simulation runs a complete run-through of a feature to make sure that it works as one would expect it to.

Typically object detection isn't enough to know if a feature will behave as is intended but if we know of specific bugs we can quickly build contrived tests for future knowledge. For example, Internet Explorer will return both elements and comments if you do `getElementsByTagName("*")`. Doing simple object detection isn't enough to determine if this will happen. Additionally this bug will, most likely, be fixed by the Internet Explorer team at some future release of the browser.

We can see an example of using feature simulation to determine if the `getElementsByTagName` method will work as we expect it to in Listing 10.9.

**Listing 10.9: Defining a global variable that captures information about how a browser feature works and using it later on.**

```
// Run once, at the beginning of the program
var ELEMENTS_ONLY = (function(){
  var div = document.createElement("div");
  div.appendChild( document.createComment("test" ) );
  return div.getElementsByTagName("*").length === 0;
})();

// Later on:
var all = document.getElementsByTagName("*");

if ( ELEMENTS_ONLY ) {
  for ( var i = 0; i < all.length; i++ ) {
    action( all[i] );
  }
} else {
  for ( var i = 0; i < all.length; i++ ) {
    if ( all[i].nodeType === 1 ) {
      action( all[i] );
    }
  }
}
```

Feature simulation works in two ways. To start a simple test is run to determine if a feature work as we expect it to. It's important to try and verify the integrity of a feature (making sure it works correctly) rather than explicitly testing for the presence of a bug. It's only a semantic distinction but it's one that is important to maintain in your coding.

Secondly we use the results of the test run later on in our program to speed up looping through an array of elements. Since a browser that returns only elements doesn't need to perform these element checks on every stage of the loop we can completely skip it and get the performance benefit in the browsers that work correctly.

The is the most common idiom used in feature simulation: Making sure a feature works as you expect it to and providing browsers, that work correctly, with optimized code - an example of which can be seen in Listing 10.10.

**Listing 10.10: Figure out the attribute name used to access textual element style information.**

```
<div id="test" style="color:red;"></div>
<div id="test2"></div>
<script>
// Perform the initial attribute check
var STYLE_NAME = (function(){
  var div = document.createElement("div");
  div.style.color = "red";

  if ( div.getAttribute("style") )
    return "style";
    if ( div.getAttribute("cssText") )
      return "cssText";
  })();

  // Later on:
  window.onload = function(){
    document.getElementsById("test2").getAttribute( STYLE_NAME ) =
        document.getElementById("test").getAttribute( STYLE_NAME );
  };
  </script>
```

In this example we, again, break down into two parts. We start by creating a dummy element, setting a style property, and then attempt to get the complete style information back out again. We start by using the standards-compliant way (accessing the style attribute) then by trying the Internet Explorer-specific way. In either case we return the name of the property that worked.

We can then use that property name at any point in our code when it comes time to get or set style attribute values, which is what we do when the page fully loads.

There's a couple points to take into consideration when examining feature simulation. Namely that it's a trade-off. For the extra performance overhead of the initial simulation and extra lines of code added to your program you get the benefit of knowing exactly how a feature works in the current browser and you become immune of future bug fixes. The immunity can be absolutely priceless when dealing with reusable code bases.

### 10.3.4 Untestable

Unfortunately there are a number of features in JavaScript and the DOM that are either impossible or prohibitively expensive to test. These features are, generally, quite rare. If you encounter an issue in your code that appears to be untestable it will always pay to investigate the matter further.

Following are some issues that are impossible to be tested using any conventional JavaScript interactions.

Determining if an event handler has been bound. Browsers do not provide a way of determining if if any functions have been bound to an event listener on an element. Because of this there is no way to remove all bound event handlers from an element unless you have foreknowledge of, and maintain references to, all bound handlers.

Determining if an event will fire. While it's possible to determine if a browser supports a means of binding an event (such as using `bindEventListener`) it's not possible to know if a browser will actually fire an event. There are a couple places where this becomes problematic.

First, if a script is loaded dynamically, after the page has already loaded, it may try to bind a listener to wait for the window to load when, in fact, that event happened some time ago. Since there's no way to determine that the event already occurred the code may wind up waiting indefinitely to execute.

Second, if a script wishes to uses specific events provided by a browser as an alternative. For example Internet Explorer provides mouseenter and mouseleave events which simplify the process of determining when a user's mouse enters and leaves an element. These are frequently used as an alternative to the mouseover and mouseout events. However since there's no way of determining if these events will fire without first binding the events and waiting for some user interaction against them it becomes prohibitive to use them for interaction.

Determining if changing certain CSS properties affects the presentation. A number of CSS properties only affect the visual representation of the display and nothing else (don't change surrounding elements or event other properties on the element) - like color, backgroundColor, or opacity. Because of this there is no way to programmatically determine if changing these style properties are actually generating the effects that are desired. The only way to verify the impact is through a visual examination of the page.

Testing script that causes the browser to crash. Code that causes a browser to crash is especially problematic since, unlike exceptions which can be easily caught and handled, these will always cause the browser to break.

For example, in older versions of Safari, creating a regular expression that used Unicode characters ranges would always cause the browser to crash, like in the following example:

```
new RegExp("[\\w\u0128-\uFFFF*_-]+");
```

The problem with this occurring is that it's not possible to test how this works using regular feature simulation since it will always produce an undesired result in that older browser. Additionally bugs that cause crashes to occur forever become embroiled in difficulty since while it may be acceptable to have JavaScript be disabled in some segment of the population using your browser, it's never acceptable to outright crash the browser of those users.

Testing bugs that cause an incongruous API. In Listing 10.6 when we looked at disallowing the ability to change the type attribute in all browsers due to a bug in Internet Explorer. We could test this feature and only disable it in Internet Explorer however that would give us a result where the API works differently from browser-to-browser. In issues like this - where a bug is so bad that it causes an API to break – the only option is to write around the affected area and provide a different solution.

The following are items that are not impossible to test but are prohibitively difficult to test effectively.

The performance of specific APIs. Sometimes specific APIs are faster or slower in different browsers. It's important to try and use the APIs that will provide the fastest results but it's not always obvious which API will yield that. In order to do effective performance analysis of a feature you'll need to make it difficult enough as to take a large amount of time in order

Determining if Ajax requests work correctly. As mentioned when we looked at regressions, Internet Explorer broke requesting local files via the XMLHttpRequest object in

Internet Explorer 7. We could test to see if this bug has been fixed but in order to do so we would have to perform an extra request on every page load that attempted to perform a request. Not only that but an extra file would have to be included with the library whose sole purpose would be to exist for these extra requests. The overhead of both these matters is quite prohibitive and would, certainly, not be worth the extra effort of simply doing object detection instead.

While untestable features are a significant hassle (limiting the effectiveness of writing reusable JavaScript) they are almost always able to be worked around. By utilizing alternative techniques, or constructing your APIs in a manner as to obviate these issues in the first place, it will most likely be possible that you'll be able to build effective code, regardless.

## 10.4 Implementation Concerns

Writing cross-browser, reusable, code is a battle of assumptions. By using better means of detection and authoring you're reducing the number of assumptions that you make in your code. When you make assumptions about the code that you write you stand to encounter problems later on. For example, if you assume that a feature or a bug will always exist in a specific browser - that's a huge assumption. Instead, testing for that functionality proves to be much more effective.

In your coding you should always be striving to reduce the number of assumptions that you make, effectively reducing the room that you have for error. The most common area of assumption-making, that is normally seen in JavaScript, is that of user agent detection. Specifically, analyzing the user agent provided by a browser (`navigator.userAgent`) and using it to make an assumption about how the browser will behave. Unfortunately, most user agent string analysis proves to be a fairly large source of future-induced errors. Assuming that a bug will always be linked to a specific browser is a recipe for disaster.

However, there is one problem when dealing with assumptions: it's virtually impossible to remove all of them. At some point you'll have to assume that a browser will do what it proposes. Figuring out the best point at which that balance can be struck is completely up to the developer.

For example, let's re-examine the event attaching code that we've been looking at, in Listing 10.11.

**Listing 10.11: A simple example of catching the implementation of a new API.**

```
function attachEvent( elem, type, handle ) {
  // bind event using proper DOM means
  if ( elem.addEventListener )
    elem.addEventListener(type, handle, false);

  // use the Internet Explorer API
  else if ( elem.attachEvent )
    elem.attachEvent("on" + type, handle);
}
```

In the above listing we make three assumptions, namely:

1. That the properties that we're checking are, in fact, callable functions.
2. That they're the right functions, performing the action that we expect.
3. That these two methods are the only possible ways of binding an event.

We could easily negate the first assumption by adding checks to see if the properties are, in fact, functions. However how we tackle the remaining two points is much more problematic.

In your code you'll need to decide how many assumptions are a correct level for you. Frequently when reducing the number of assumptions you also increase the size and complexity of your code base. It's fully possible to attempt to reduce assumptions to the point of insanity but at some point you'll have to stop and take stock of what you have and work from there. Even the least assuming code is still prone to regressions introduced by a browser.

## 10.5 Summary

Cross-browser development is a juggling act between three points:

- Code Size: Keeping the file size small.
- Performance Overhead: Keeping the performance hits to a palatable minimum.
- API Quality: Making sure that the APIs that are provided work uniformly across browsers.

There is no magic formula for determining what the correct balance of these points are. They are something that will have to be balanced by every developer in their individual development efforts. Thankfully using smart techniques like object detection and feature simulation it's possible to defend against any of the numerous directions from which reusable code will be attacked, without making any undue sacrifices.

# 11 CSS selector engine

In this chapter:

- The tools that we have for building a selector engine
- The strategies for engine construction

CSS selector engines are a relatively new development in the world of JavaScript but they've taken the world of libraries by storm. Every major JavaScript library includes some implementation of a JavaScript CSS selector engine.

The premise behind the engine is that you can feed it a CSS selector (for example "div > span") and it will return all the DOM elements that match the selector on the page (in this case all spans that are a child of a div element). This allows users to write terse statements that are incredibly expressive and powerful. By reducing the amount of time the user has to spend dealing with the intricacies of traversing the DOM it frees them to handle other tasks.

It's standard, at this point, that any selector engine should implement CSS 3 selectors, as defined by the W3C:

- http://www.w3.org/TR/css3-selectors/

There are three primary ways of implementing a CSS selector engine:

1. Using the W3C Selectors API - a new API specified by the W3C and implemented in most newer browsers.
2. XPath - A DOM querying language built into a variety of modern browsers.
3. Pure DOM - A staple of CSS selector engines, allows for graceful degradation if either of the first two don't exist.

This chapter will explore each of these strategies in depth allowing you to make some educated decisions about implementing, or at least understanding, a JavaScript CSS selector engine.

## 11.1 Selectors API

The W3C Selectors API is a, comparatively, new API that is designed to reduce much of the work that it takes to implement a full CSS selector engine in JavaScript. Browser vendors have pounced on this new API and it's implemented in all major browsers (starting in Safari 3, Firefox 3.1, Internet Explorer 8, and Opera 10). Implementations of the API generally support all selectors implemented by the browser's CSS engine. Thus if a browser has full CSS 3 support their Selectors API implementation will reflect that.

The API provides two methods:

- `querySelector`: Accepts a CSS selector string and returns the first element found (or null if no matching element is found).
- `querySelectorAll`: Accepts a CSS selector string and returns a static NodeList of all elements found by the selector.

and these two methods exist on all DOM elements, DOM documents, and DOM fragments. Listing 11.1 has a couple examples of how it could be used.

**Listing 11.1: Examples of the Selectors API in action.**

```
<div id="test">
  <b>Hello</b>, I'm a ninja!
</div>
<div id="test2"></div>
<script>
window.onload = function(){
  var divs = document.querySelectorAll("body > div");
  assert( divs.length === 2, "Two divs found using a CSS selector." );

  var b = document.getElementById("test")
    .querySelector("b:only-child");
  assert( b,
  "The bold element was found relative to another element." );
};
</script>
```

Perhaps the one gotcha that exists with the current Selectors API is that it more-closely capitulates to the existing CSS selector engine implementations rather than the implementations that were first created by JavaScript libraries. This can be seen in the matching rules of element-rooted queries, as seen in Listing 11.2.

**Listing 11.2: Element-rooted queries.**

```
<div id="test">
  <b>Hello</b>, I'm a ninja!
</div>
```

```
<script>
window.onload = function(){
  var b = document.getElementById("test").querySelector("div b");
  assert( b, "Only the last part of the selector matters." );
};
</script>
```

Note the issue here: When performing an element-rooted query (calling `querySelector` or `querySelectorAll` relative to an element) the selector only checks to see if the final portion of the selector is contained within the element. This will probably seem counter-intuitive (looking at the previous listing we can verify that there are no div elements with the element with an ID of test - even though that's what the selector looks like it's verifying).

Since this runs counter to how most users expect a CSS selector engine to work we'll have to provide a work-around. The most common solution is to add a new ID to the rooted element to enforce its context, like in Listing 11.3.

**Listing 11.3: Enforcing the element root.**

```
<div id="test">
  <b>Hello</b>, I'm a ninja!
</div>
<script>
(function(){
  var count = 1;

  this.rootedQuerySelectorAll = function(elem, query){
    var oldID = elem.id;
    elem.id = "rooted" + (count++);

    try {
      return elem.querySelectorAll( "#" + elem.id + " " + query );
    } catch(e){
      throw e;
    } finally {
      elem.id = oldID;
    }
  };
})();

window.onload = function(){
  var b = rootedQuerySelectorAll(
    document.getElementById("test"), "div b");
  assert( b.length === 0, "The selector is now rooted properly." );
};
</script>
```

Looking at the previous listing we can see a couple important points. To start we must assign a unique ID to the element and restore the old ID later. This will ensure that there are no collisions in our final result when we build the selector. We then prepend this ID (in the form of a "#id " selector) to the selector.

Normally it would be as simple as removing the ID and returning the result from the query but there's a catch: Selectors API methods can throw exceptions (most commonly seen for selector syntax issues or unsupported selectors). Because of this we'll want to wrap our selection in a try/catch block. However since we want to restore the ID we can add an extra finally block. This is an interesting feature of the language - even though we're returning a value in the try, or throwing an exception in the catch, the code in the finally will always execute after both of them are done executing (but before the value is return from the

function or the object is thrown). In this manner we can verify that the ID will always be restored properly.

The Selectors API is absolutely one of the most promising APIs to come out of the W3C in recent history. It has the potential to completely replace a large portion of most JavaScript libraries with a simple method (naturally after the supporting browsers gain a dominant market share).

## 11.2 XPath

A unified alternative to using the Selectors API (in browsers that don't support it) is the use of XPath querying. XPath is a querying language utilized for finding DOM nodes in a DOM document. It is significantly more powerful than traditional CSS selectors. Most modern browsers (Firefox, Safari 3+, Opera 9+) provide some implementation of XPath that can be used against HTML-based DOM documents. Internet Explorer 6, 7, and 8 provide XPath support for XML documents (but not against HTML documents - the most common target).

If there's one thing that can be said for utilizing XPath expressions: They're quite fast for complicated expressions. When implementing a pure-DOM implementation of a selector engine you are constantly at odds with the ability of a browser to scale all the JavaScript and DOM operations. However, XPath loses out for simple expressions.

There's a certain indeterminate threshold at which it becomes more beneficial to use XPath expressions in favor of pure DOM operations. While this might be able to be determined programatically there are a few gives: Finding elements by ID ("#id") and simple tag-based selectors ("div") will always be faster with pure-DOM code.

If you and your users are comfortable using XPath expressions (and are happy limiting yourself to the modern browsers that support it) then simply utilize the method shown in Listing 11.4 and completely ignore everything else about building a CSS selector engine.

**Listing 11.4: A method for executing an XPath expression on an HTML document, returning an array of DOM nodes, from the Prototype library.**

```
if ( typeof document.evaluate === "function" ) {
  function getElementsByXPath(expression, parentElement) {
    var results = [];
    var query = document.evaluate(expression,
      parentElement || document,
      null, XPathResult.ORDERED_NODE_SNAPSHOT_TYPE, null);
    for (var i = 0, length = query.snapshotLength; i < length; i++)
```

```
        results.push(query.snapshotItem(i));
      return results;
    }
  }
```

While it would be nice to just use XPath for everything it simply isn't feasible. XPath, while feature-packed, is designed to be used by developers and is prohibitively complex, in comparison to the expressions that CSS selectors make easy. While it simply isn't feasible to look at the entirety of XPath we can take a quick look at some of the most common expressions and how they map to CSS selectors, in Table 11.1.

Table 11.1: Map of CSS selectors to their associated XPath expressions.

| Goal | CSS 3 | XPath |
|---|---|---|
| All Elements | * | //* |
| All P Elements | p | //p |
| All Child Elements | p > * | //p/* |
| Element By ID | #foo | //*[@id='foo'] |
| Element By Class | .foo | //*[contains(concat(" ", @class, "")," foo ")] |
| Element With Attribute | *[title] | //*[@title] |
| First Child of All P | p > *:first-child | //p/*[0] |
| All P with an A descendant | Not possible | //p[a] |
| Next Element | p + * | //p/following-sibling::*[0] |

Using XPath expressions would work as if you were constructing a pure-DOM selector engine (parsing the selector using regular expressions) but with an important deviation: The resulting CSS selector portions would get mapped to their associated XPath expressions and executed.

This is especially tricky since the result is, code-wise, about as large as a normal pure-DOM CSS selector engine implementation. Many developers opt to not utilize an XPath engine simply to reduce the complexity of their resulting engines. You'll need to weigh the performance benefits of an XPath engine (especially taking into consideration the competition from the Selectors API) against the inherent code size that it will exhibit.

## 11.3 DOM

At the core of every CSS selector engine exists a pure-DOM implementation. This is simply parsing the CSS selectors and utilizing the existing DOM methods (such as getElementById or getElementsByTagName) to find the corresponding elements.

It's important to have a DOM implementation for a number of reasons:

1. Internet Explorer 6 and 7. While Internet Explorer 8 has support for querySelectorAll the lack of XPath or Selectors API support in 6 and 7 make a DOM implementation necessary.
2. Backwards compatibility. If you want your code to degrade in a graceful manner and support browsers that don't support the Selectors API or XPath (like Safari 2) you'll have to have some form of a DOM implementation.
3. For speed. There are a number of selectors that a pure DOM implementation can simply do faster (suchas finding elements by ID).

With that in mind we can take a look at the two possible CSS selector engine implementations: Top down and bottom up.

A top down engine works by parsing a CSS selector from left-to-right, matching elements in a document as it goes, working relatively for each additional selector segment. It can be found in most modern JavaScript libraries and is, generally, the preferred means of finding elements on a page.

For example, given the selector "div span" a top down-style engine will find all div elements in the page then, for each div, find all spans within the div.

There are two things to take into consideration when developing a selector engine: The results should be in document order (the order in which they've been defined) and the results should be unique (no duplicate elements returned). Because of these gotchas developing a top down engine can be quite tricky.

Take the following piece of markup, from Listing 11.5, into consideration, as if we were trying to implement our "div span" selector.

**Listing 11.5: A simple top down selector engine.**

```
<div>
  <div>
    <span>Span</span>
  </div>
</div>
<script>
window.onload = function(){
  function find(selector, root){
    root = root || document;

    var parts = selector.split(" "),
      query = parts[0],
      rest = parts.slice(1).join(" "),
      elems = root.getElementsByTagName( query ),
      results = [];

    for ( var i = 0; i < elems.length; i++ ) {
      if ( rest ) {
        results = results.concat( find(rest, elems[i]) );
      } else {
          results.push( elems[i] );
        }
      }

      return results;
    }

    var divs = find("div");
    assert( divs.length === 2, "Correct number of divs found." );

    var divs = find("div", document.body);
    assert( divs.length === 2,
    "Correct number of divs found in body." );

    var divs = find("body div");
    assert( divs.length === 2,
    "Correct number of divs found in body." );

    var spans = find("div span");
    assert( spans.length === 2, "A duplicate span was found." );
  };
  </script>
```

In the above listing we implement a simple top down selector engine (one that is only capable of finding elements by tag name). The engine breaks down into a few parts: Parsing the selector, finding the elements,

filtering, and recursing and merging the results.

### 11.3.1 Parsing the Selector

In this simple example our parsing is reduced to converting a trivial CSS selector ("div span") into an array of strings (["div", "span"]). As introduced in CSS 2 and 3 it's possible to find elements by attribute or attribute value (thus it's possible to have additional spaces in most selectors - making our splitting of the selector too simplistic. For the most part this parsing ends up being "good enough" for now.

For a full implementation we would want to have a solid series of parsing rules to handle any expressions that may be thrown at us (most likely in the form of regular expressions).

Listing 11.6 has an example of a regular expression that's capable of capturing portions of a selector and breaking it in to pieces (splitting on commas, if need be):

**Listing 11.6: A regular expression for breaking apart a CSS selector.**

```
var selector = "div.class > span:not(:first-child) a[href]"
var chunker = /((?:\([^\)]+\)|\[[^\]]+\]|[^ ,\(\[]+)+)(\s*,\s*)?/g;
var parts = [];

// Reset the position of the chunker regexp (start from beginning)
chunker.lastIndex = 0;

// Collect the pieces
while ( (m = chunker.exec(selector)) !== null ) {
  parts.push( m[1] );
```

```
    // Stop if we've countered a comma
    if ( m[2] ) {
      extra = RegExp.rightContext;
      break;
    }
  }

  assert( parts.length == 4,
    "Our selector is broken into 4 unique parts." );
  assert( parts[0] === "div.class", "div selector" );
  assert( parts[1] === ">", "child selector" );
  assert( parts[2] === "span:not(:first-child)", "span selector" );
  assert( parts[3] === "a[href]", "a selector" );
```

Obviously this chunking selector is only one piece of the puzzle - you'll need to have additional parsing rules for each type of expression that you want to support. Most selector engines end up containing a map of regular expressions to functions - when a match is made on the selector portion the associated function is executed.

### 11.3.2 Finding the Elements

Finding the correct elements on the page is one piece of the puzzle that has many solutions. Which techniques are used depends a lot on which selectors are being support and what is made available by the browser. There are a number of obvious correlations, though.

getElementById: Only available on the root node of HTML documents, finds the first element on the page that has the specified ID (useful for the ID CSS selector "#id"). Internet Explorer and Opera will also find the first element on the page that has the same specified name. If you only wish to find elements by ID you will need an extra verification step to make sure that the correct result is being found.

If you wish to find all elements that match a specific ID (as is customary in CSS selectors - even though HTML documents are generally only permitted one specific ID per page) you will need to either: Traverse all elements looking for the ones that have the correct ID or use `document.all["id"]` which returns an array of all elements that match an ID in all browsers that support it (namely Internet Explorer, Opera, and Safari).

getElementsByTagName: Tackles the obvious result: Finding elements that match a specific tag name. It has a dual purpose, though - finding all elements within a document or element (using the "*" tag name). This is especially useful for handling attribute-based selectors that don't provide a specific tag name, for example: ".class" or "[attr]".

One caveat when finding elements comments using "*" - Internet Explorer will also return comment nodes in addition to element nodes (for whatever reason, in Internet Explorer, comment nodes have a tag name of "!" and are thusly returned). A basic level of filtering will need to be done to make sure that the correct nodes are matched.

getElementsByName: This is a well-implemented method that serves a single purpose: Finding all elements that have a specific name (such as for input elements that have a name). Thus it's really only useful for implementing a single selector: "[name=NAME]".

getElementsByClassName: A relatively new method that's being implemented by browsers (most prominently by Firefox 3 and Safari 3) that finds elements based upon the contents of their class attribute. This method proves to be a tremendous speed-up to class-selection code.

While there are a variety of techniques that can be used for selection the above methods are, generally, the primary tools used to what you're looking for on a page. Using the results from these methods it will be possible to

### 11.3.3 Filtering

A CSS expression is generally made up of a number of individual pieces. For example the expression "div.class[id]" has three parts: Finding all div elements that have a class name of "class" and have an attribute named "id".

The first step is to identify a root selector to being with. For example we can see that "div" is used - thus we can immediately use `getElementsByTagName` to retrieve all div elements on the page. We must, then, filter those results down to only include those that have the specified class and the specified id attribute.

This filtering process is a common feature of most selector implementations. The contents of these filters primarily deal with either attributes or the position of the element relative to its siblings.

Attribute Filtering: Accessing the DOM attribute (generally using the getAttribute method) and verifying their values. Class filtering (".class") is a subset of this behavior (accessing the className attribute and checking its value).

Position Filtering: For selectors like ":nth-child(even)" or ":last-child" a combination of methods are used on the parent element. In browser's that support it `.children` is used (IE, Safari, Opera, and Firefox 3.1) which contains a list of all child elements. All browsers have `.childNodes` (which contains a list of child nodes - including text nodes, comments, etc.). Using these two methods it becomes possible to do all forms of element position filtering.

Constructing a filtering function serves a dual purpose: You can provide it to the user as a simple method for testing their elements, quickly checking to see if an element matches a specific selector.

### 11.3.4 Recursing and Merging

As was shown in Listing 11.1, we can see how selector engines will require the ability to recurse (finding descendant elements) and merge the results together.

However our implementation is too simple, note that we end up receiving two spans in our results instead of just one. Because of this we need to introduce an additional check to make sure the returned array of elements only contains unique results. Most top down selector implementations include some method for enforcing this uniqueness.

Unfortunately there is no simple way to determine the uniqueness of a DOM element. We're forced to go through and assign temporary IDs to the elements so that we can verify if we've already encountered them, as in Listing 11.7.

**Listing 11.7: Finding the unique elements in an array.**

```
<div id="test">
  <b>Hello</b>, I'm a ninja!
</div>
<div id="test2"></div>
<script>
(function(){
  var run = 0;

  this.unique = function( array ) {
    var ret = [];

    run++;

    for ( var i = 0, length = array.length; i < length; i++ ) {
      var elem = array[ i ];

      if ( elem.uniqueID !== run ) {
        elem.uniqueID = run;
        ret.push( array[ i ] );
      }
    }

    return ret;
  };
})();

window.onload = function(){
  var divs = unique( document.getElementsByTagName("div") );
  assert( divs.length === 2, "No duplicates removed." );

  var body = unique( [document.body, document.body] );
  assert( body.length === 1, "body duplicate removed." );
};
</script>
```

This unique method adds an extra property to all the elements in the array - marking them as having been visited. By the time a complete run through is finished only unique elements will be left in the resulting array. Variations of this technique can be found in all libraries.

A longer discussion on the intricacies of attaching properties to DOM nodes see the chapter on Events.

### *11.3.5 Bottom Up Selector Engine*

If you prefer not to have to think about uniquely identifying elements there is an alternative style of CSS selector engine that doesn't require its use.

A bottom up selector engine works in the opposite direction of a top down one. For example, given the selector "div span" it will first find all span elements then, for each element, navigate up the ancestor elements to find an ancestor div element. This style of selector engine construction matches the style found in most browser engines.

This engine style isn't as popular as the others. While it works well for simple selectors (and child selectors) the ancestor travels ends up being quite costly and doesn't scale very well. However the simplicity that this engine style provides can end up making for a nice trade-off.

The construction of the engine is simple. You start by finding the last expression in the CSS selector and retrieve the appropriate elements (just like with a top down engine, but the last expression rather than the first). From here on all operations are performed as a series of filter operations, removing elements as they go, like in Listing 11.8.

**Listing 11.8: A simple bottom up selector engine.**

```
<div>
  <div>
    <span>Span</span>
  </div>
</div>
<script>
window.onload = function(){
  function find(selector, root){
    root = root || document;

    var parts = selector.split(" "),
      query = parts[parts.length - 1],
      rest = parts.slice(0,-1).join("").toUpperCase(),
      elems = root.getElementsByTagName( query ),
      results = [];

    for ( var i = 0; i < elems.length; i++ ) {
      if ( rest ) {
        var parent = elems[i].parentNode;
        while ( parent && parent.nodeName != rest ) {
          parent = parent.parentNode;
        }

        if ( parent ) {
          results.push( elems[i] );
        }
      } else {
        results.push( elems[i] );
      }
    }

    return results;
  }

  var divs = find("div");
  assert( divs.length === 2, "Correct number of divs found." );
```

```
    var divs = find("div", document.body);
    assert( divs.length === 2,
    "Correct number of divs found in body." );

    var divs = find("body div");
    assert( divs.length === 2,
    "Correct number of divs found in body." );

    var spans = find("div span");
    assert( spans.length === 1, "No duplicate span was found." );
  };
  </script>
```

Listing 11.8 shows the construction of a simple bottom up selector engine. Note that it only works one ancestor level deep. In order to work more than one level deep the state of the current level would need to be tracked. This would result in two state arrays: The array of elements that are going to be returned (with some elements being set to undefined as they don't match the results) and an array of elements that correspond to the currently-tested ancestor element.

As mentioned before, this extra ancestor verification process does end up being slightly less scalable than the alternative top down method but it completely avoids having to utilize a unique method for producing the correct output, which some may see as an advantage.

## 11.4 Summary

JavaScript-based CSS selector engines are deceptively powerful tools. They give you the ability to easily locate virtually any DOM element on a page with a trivial amount of selector. While there are many nuances to actually implementing a full selector engine (and, certainly, no shortage of tools to help) the situation is rapidly improving.

With browsers working quickly to implement versions of the W3C Selector API having to worry about the finer points of selector implementation will soon be a thing of the past. For many developers that day cannot come soon enough.

# 12 DOM modification

In this chapter:

- Injecting HTML strings into a page
- Cloning elements
- Removing elements
- Manipulating element text

Next to traversing the DOM the most common operation required by most pieces of reusable JavaScript code is that of modifying DOM structures - in addition to modifying DOM node attributes or CSS properties (which we'll discuss later) - the brunt of the work falls back to injecting new nodes into a document, cloning nodes, and removing them again.

## 12.1 Injecting HTML

In this chapter we'll start by looking at an efficient way to insert an HTML string into a document at an arbitrary location. We're looking at this technique, in particular, since it's frequently used in a few ways: Injecting arbitrary HTML into a page, manipulating and inserting client-side templates, and retrieving and injecting HTML sent from a server. No matter the framework it'll have to deal with, at least, some of these problems.

On top of those points it's technically very challenging to implement correctly. Compare this to building an object-oriented style DOM construction API (which are certainly easier to implement but require an extra layer of abstraction from injecting the HTML).

There already exists an API for injecting arbitrary HTML strings - introduced by Internet Explorer (and in the process of being standardized in the W3C HTML 5 specification). It's a method that exists on all HTML DOM elements called `insertAdjacentHTML`. We'd spend more time looking at this API but there are a few problems.

1. It currently only exists in Internet Explorer (thus an alternative solution would have to be implemented anyway).
2. Internet Explorer's implementation is incredibly buggy (only working on a subset of all available elements).

For these reasons we're going to have to implement a clean API from scratch. An implementation is broken down into a couple of steps:

1. Converting an arbitrary, valid, HTML/XHTML string into a DOM structure.
2. Injecting that DOM structure into an arbitrary location in the DOM as efficiently as possible.
3. Executing any inline scripts that were in the string.

All together, these three steps will provide a user with a smart API for injecting HTML into a document.

### *12.1.1 Converting HTML to DOM*

Converting an HTML string to a DOM structure doesn't have a whole lot of magic to it - it uses the exact tool that you are already familiar with: innerHTML. It's a multi-step process:

- Make sure the HTML string contains valid HTML/XHTML (or, at least, tweak it so that it's closer to valid).
- Wrap the string in any enclosing markup required
- Insert the HTML string, using `innerHTML`, into a dummy DOM element.
- Extract the DOM nodes back out.

The steps aren't too complex - save for the actual insertion, which has some gotchas - but they can be easily smoothed over.

#### **PRE-PROCESS XML/HTML**

To start, we'll need to clean up the HTML to meet user expectations. This first step will certainly depend upon the context, but within the construction of jQuery it became important to be able to support XML- style elements like "`<table/>`".

The above XML-style of elements, in actuality, only works for a small subset of HTML elements - attempting to use that syntax otherwise is able to cause problems in browsers like Internet Explorer.

We can do a quick pre-parse on the HTML string to convert elements like "`<table/>`" to "`<table>></table>`" (which will be handled uniformly in all browsers).

**Listing 12.1: Make sure that XML-style HTML elements are interpreted correctly.**

```
var tags = /^(abbr|br|col|img|input|link|meta|param|hr|area|embed)$/i;

function convert(html){
  return html.replace(/(<(\w+)[^>]*?)\/>/g, function(all, front, tag){
    return tags.test(tag) ?
      all :
      front + "></" + tag + ">";
  });
}

assert( convert("<a/>") === "<a></a>", "Check anchor conversion." );
assert( convert("<hr/>") === "<hr/>", "Check hr conversion." );
```

### HTML Wrapping

We now have the start of an HTML string - but there's another step that we need to take before injecting it into the page. A number of HTML elements must be within a certain container element before they must be injected. For example an option element must be within a select. There are two options to solve this problem (both require constructing a map between problematic elements and their containers).

- The string could be injected directly into a specific parent, previously constructed using `createElement`, using `innerHTML`. While this may work in some cases, in some browsers, it is not universally guaranteed to work.
- The string could be wrapped with the appropriate markup required and then injected directly into any container element (such as a div).

The second technique is the preferred one. It involves very little browser-specific code in contrast with the other one, which would be mostly browser-specific code.
The set of elements that need to be wrapped is rather manageable (with 7 different groupings occurring).

- option and optgroup need to be contained in a `<select multiple="multiple">...</select>`
- legend need to be contained in a `<fieldset>...</fieldset>`
- thead, tbody, tfoot, colgroup, and caption need to be contained in a `<table>...</table>`
- tr need to be in a `<table><thead>...</thead></table>`, `<table><tbody>...</tbody></table>`, or a `<table><tfoot>...</tfoot></table>`
- td and th need to be in a `<table><tbody><tr>...</tr></tbody></table>`
- col in a `<table><tbody></tbody><colgroup>...</colgroup></table>`
- link and script need to be in a `div<div>...</div>`

Nearly all of the above are self-explanatory save for the multiple select, the col, and the link and script ones. A multiple select is used (instead of a regular select) because it won't automatically check any of the options that are placed inside of it (whereas a single select will auto-check the first option). The col fix includes an extra tbody, without which the colgroup won't be generated properly. The link and script fix is a weird one: Internet Explorer is unable to generate link and script elements, via innerHTML, unless they are both contained within another element and there's an adjacent node.

### Generating the DOM

Using the above map of characters we not have enough information to generate the HTML that we need to insert in to a DOM element.

**Listing 12.2: Generate a list of DOM nodes from some markup.**

```
function getNodes(htmlString){
  var map = {
    "<td>": [3, "<table><tbody><tr>", "</tr></tbody></table>"],
    "<option>": [1, "<select multiple='multiple'>", "</select>"]
    // a full list of all element fixes
  };

  var name = htmlString.match(/<\w+/),
    node = name ? map[ name[0] ] : [0, "", ""];

  var div = document.createElement("div");
  div.innerHTML = node[1] + htmlString + node[2];

  while ( node[0]-- )
    div = div.lastChild;

  return div.childNodes;
}

assert( getNodes("<td>test</td><td>test2</td>").length === 2,
  "Get two nodes back from the method." );
assert( getNodes("<td>test</td>")[0].nodeName === "TD",
  "Verify that we're getting the right node." );
```

There are two bugs that we'll need to take care of before we return our node set, though - and both are bugs in Internet Explorer. The first is that Internet Explorer adds a tbody element inside an empty table (checking to see if an empty table was intended and removing any child nodes is a sufficient fix). Second is that Internet Explorer trims all leading whitespace from the string passed to `innerHTML`. This can be remedied by checking to see if the first generated node is a text and contains leading whitespace, if not create a new text node and fill it with the whitespace explicitly.

After all of this we now have a set of DOM nodes that we can begin to insert into the document.

### 12.1.2 Inserting into the Document

Once you have the actual DOM nodes it becomes time to insert them into the document. There are a couple steps the need to take place - none of which are particularly tricky.

Since we have an array of elements that we need to insert and, potentially, into any number of locations into the document, we'll need to try and cut down on the number of operations that need to occur.

We can do this by using DOM Fragments. Fragments are part of the W3C DOM specification and are supported in all browsers. They give you a container that you can use to hold a collection of DOM nodes. This, in itself, is useful but it also has two extra advantages: The fragment can be injected and cloned in a single operation instead of having to inject and clone each individual node over and over again. This has the potential to dramatically reduce the number of operations required for a page.

In the following example, derived from the code in jQuery, a fragment is created and passed in to the clean function (which converts the incoming HTML string into a DOM). This DOM is automatically appended on to the fragment.

**Listing 12.3: Inserting a DOM fragment into multiple locations in the DOM, using the code from Listing 12.1 and Listing 12.4.**

```
<div id="test"><b>Hello</b>, I'm a ninja!</div>
<div id="test2"></div>
<script>
window.onload = function(){
  function insert(elems, args, callback){
    if ( elems.length ) {
      var doc = elems[0].ownerDocument || elems[0],
        fragment = doc.createDocumentFragment(),
        scripts = getNodes( args, doc, fragment ),
        first = fragment.firstChild;

      if ( first ) {
        for ( var i = 0; elems[i]; i++ ) {
          callback.call( root(elems[i], first),
            i > 0 ? fragment.cloneNode(true) : fragment );
        }
      }
    }
  }

  var divs = document.getElementsByTagName("div");

  insert(divs, ["<b>Name:</b>"], function(fragment){
    this.appendChild( fragment );
  });

  insert(divs, ["<span>First</span> <span>Last</span>"],
    function(fragment){
      this.parentNode.insertBefore( fragment, this );
    });
};
```

```
</script>
```

There's another important point here: If we're inserting this element into more than one location in the document we're going to need to clone this fragment again and again (if we weren't using a fragment we'd have to clone each individual node every time, instead of the whole fragment at once).

One final point that we'll need to take care of, albeit a relatively minor one. When users attempt to inject a table row directly into a table element they normally mean to insert the row directly in to the tbody that's in the table. We can write a simple mapping function to take care of that for us.

**Listing 12.4: Figure out the actual insertion point of an element.**

```
function root( elem, cur ) {
  return elem.nodeName.toLowerCase() === "table" &&
    cur.nodeName.toLowerCase() === "tr" ?
    (elem.getElementsByTagName("tbody")[0] ||
      elem.appendChild(elem.ownerDocument.createElement("tbody"))) :
    elem;
}
```

Altogether we now have a way to both generate and insert arbitrary DOM elements in an intuitive manner.

### *12.1.3 Script Execution*

Aside from the actual insertion of HTML into a document a common requirement is the execution of inline script elements. This is mostly used when a piece of HTML is coming back from a server and there's script that needs to be executed along with the HTML itself.

Usually the best way to handle inline scripts is to strip them out of the DOM structure before they're actually inserted into the document. In the function that's used to convert the HTML into a DOM node the end result would look something like the following code from jQuery.

**Listing 12.5: Collecting the scripts from a piece an array of**

```
for ( var i = 0; ret[i]; i++ ) {
  if ( jQuery.nodeName( ret[i], "script" ) &&
      (!ret[i].type ||
        ret[i].type.toLowerCase() === "text/javascript") ) {
    scripts.push( ret[i].parentNode ?
      ret[i].parentNode.removeChild( ret[i] ) :
      ret[i] );
  } else if ( ret[i].nodeType === 1 ) {
    ret.splice.apply( ret, [i + 1, 0].concat(
      jQuery.makeArray(ret[i].getElementsByTagName("script"))) );
  }
}
```

The above code is dealing with two arrays: ret (which holds all the DOM nodes that have been generated) and scripts (which becomes populated with all the scripts in this fragment, in document order). Additionally, it takes care to only remove scripts that are normally executed as JavaScript (those with no type or those with a type of 'text/javascript').

Then after the DOM structure is inserted into the document take the contents of the scripts and evaluate them. None of this is particularly difficult - just some extra code to shuffle around.

But it does lead us to the tricky part:

### GLOBAL CODE EVALUATION

When users are including inline scripts to be executed, they're expecting them to be evaluated within the global context. This means that if a variable is defined it should become a global variable (same with functions, etc.).

The built-in methods for code evaluation are spotty, at best. The one fool-proof way to execute code in the global scope, across all browsers, is to create a fresh script element, inject the code you wish to execute inside the script, and then quickly inject and remove the script from the document. This will cause the browser to execute the inner contents of the script element within the global scope. This technique was pioneered by Andrea Giammarchi and has ended up working quite well. Below is a part of the global evaluation code that's in jQuery.

**Listing 12.6: Evaluate a script within the global scope.**

```
function globalEval( data ) {
  data = data.replace(/^\s+|\s+$/g, "");

  if ( data ) {
    var head = document.getElementsByTagName("head")[0] ||
        document.documentElement,
      script = document.createElement("script");

    script.type = "text/javascript";
    script.text = data;

    head.insertBefore( script, head.firstChild );
    head.removeChild( script );
  }
}
```

Using this method it becomes easy to rig up a generic way to evaluate a script element. We can even add in some simple code for dynamically loading in a script (if it references an external URL) and evaluate that as well.

**Listing 12.7: A method for evaluating a script (even if it's remotely located).**

```
function evalScript( elem ) {
  if ( elem.src )
```

```
      jQuery.ajax({
        url: elem.src,
        async: false,
        dataType: "script"
      });

    else
      jQuery.globalEval( elem.text || "" );

    if ( elem.parentNode )
      elem.parentNode.removeChild( elem );
  }
```

Note that after we're done evaluating the script we remove it from the DOM. We did the same thing earlier when we removed the script element before it was injected into the document. We do this so that scripts won't be accidentally double-executed (appending a script to a document, which ends up recursively calling itself, for example).

## 12.2 Cloning Elements

Cloning an element (using the DOM `cloneNode` method) is straightforward in all browsers, except Internet Explorer. Internet Explorer has three behaviors that, when they occur in conjunction, result in a very frustrating scenario for handling cloning.

First, when cloning an element, Internet Explorer copies over all event handlers on to the cloned element. Additionally, any custom expandos attached to the element are also carried over. In jQuery a simple test to determine if this is the case.

**Listing 12.8: Determining if a browser copies event handlers on clone.**

```
<script>
var div = document.createElement("div");

if ( div.attachEvent && div.fireEvent ) {
  div.attachEvent("onclick", function(){
    // Cloning a node shouldn't copy over any
    // bound event handlers (IE does this)
    jQuery.support.noCloneEvent = false;
    div.detachEvent("onclick", arguments.callee);
  });
  div.cloneNode(true).fireEvent("onclick");
}
</script>
```

Second, the obvious step to prevent this would be to remove the event handler from the cloned element - but in Internet Explorer, if you remove an event handler from a cloned element it gets removed from the original element as well. Fun stuff. Naturally, any attempts to remove custom expando properties on the clone will cause them to be removed on the original cloned element, as well.

Third, the solution to all of this is to just clone the element, inject it into another element, then read the `innerHTML` of the element - and convert that back into a DOM node. It's a multi-step process but one that'll result in an untainted cloned element. Except, there's another IE bug: The `innerHTML` (or `outerHTML`, for that matter) of an element doesn't always reflect the correct state of an element's attributes. One common place where this is realized is when the name attributes of input elements are changed dynamically - the new value isn't represented in the innerHTML.

This solution has another caveat: `innerHTML` doesn't exist on XML DOM elements, so we're forced to go with the traditional `cloneNode` call (thankfully, though, event listeners on XML DOM elements are pretty rare).

The final solution for Internet Explorer ends up becoming quite circuitous. Instead of a quick call to `cloneNode` it is, instead, serialized by `innerHTML`, extracted again as a DOM node, and then monkey-patched for any particular attributes that didn't carry over. How much monkeying you want to do with the attributes is really up to you.

**Listing 12.9: A portion of the element clone code from jQuery.**

```
function clone() {
  var ret = this.map(function(){
    if ( !jQuery.support.noCloneEvent && !jQuery.isXMLDoc(this) ) {
      var clone = this.cloneNode(true),
        container = document.createElement("div");
      container.appendChild(clone);
      return jQuery.clean([container.innerHTML])[0];
    } else
      return this.cloneNode(true);
  });

  var clone = ret.find("*").andSelf().each(function(){
    if ( this[ expando ] !== undefined )
      this[ expando ] = null;
  });

  return ret;
}
```

Note that the above code uses jQuery's `jQuery.clean` method which converts an HTML string into a DOM structure (which was discussed previously).

## 12.3 Removing Elements

Removing an element from the DOM should be simple (a quick call to removeChild) - but of course it isn't. We have to do a lot of preliminary cleaning up before we can actually remove an element from the DOM.

There's usually two steps of cleaning that need to occur on an DOM element before it can be removed from the DOM. Both of them relate to events, so we'll be discussing this in more detail when we cover it in the Events chapter.

The first things to clean up are any bound event handlers from the element. If a framework is designed well it should only be binding a single handler for an element at a time so the cleanup shouldn't be any harder than just removing that one function. This step is important because Internet Explorer will leak memory should the function reference an external DOM element.

The second point of cleanup is removing any external data associated with the element. We'll discuss this more in the Events chapter but a framework needs a good way to associate pieces of data with an element (especially without directly attaching the data as an expando property). It is a good idea to clean up this data simply so that it doesn't consume any more memory.

Now both of these points need to be done on the element that is being removed - and on all descendant elements, as well (since all the descendant elements are also being removed - just in a less-obvious way).

For example, here's the relevant code from jQuery:

**Listing 12.10: The remove element function from jQuery.**

```
function remove() {
  // Go through all descendants and the element to be removed
  jQuery( "*", this ).add([this]).each(function(){
    // Remove all bound events
    jQuery.event.remove(this);

    // Remove attached data
    jQuery.removeData(this);
  });

  // Remove the element (if it's in the DOM)
  if ( this.parentNode )
    this.parentNode.removeChild( this );
}
```

The second part to consider, after all the cleaning up, is the actual removal of the element from the DOM. Most browsers are perfectly fine with the actual removal of the element from the page -- except for Internet Explorer. Every single element removed from the page fails to reclaim some portion of its used memory, until the page is finally left. This means that long-running pages, that remove a lot of elements from the page, will find themselves using considerably more memory in Internet Explorer as time goes on.

There's one partial solution that seems to work quite well. Internet Explorer has a proprietary property called `outerHTML`. This property will give you an HTML string representation of an element. For whatever reason `outerHTML` is also a setter, in addition to a getter. As it turns out, if you set `outerHTML = ""` it will wipe out the element from Internet Explorer's memory more-completely than simply doing `removeChild`. This step is done in addition to the normal `removeChild` call .

**Listing 12.11: Set outerHTML in an attempt to reclaim more memory in Internet Explorer.**

```
// Remove the element (if it's in the DOM)
if ( this.parentNode )
  this.parentNode.removeChild( this );
```

```
if ( typeof this.outerHTML !== "undefined" )
  this.outerHTML = "";
```

It should be noted that it isn't successful in reclaiming all of the memory that was used by the element, but it absolutely reclaims more of it (which is a start, at least).

It's important to remember that any time an element is removed from the page that you should go through the above three steps - at the very least. This includes emptying out the contents of an element, replacing the contents of an element (with either HTML or text), or replacing an element directly. Remember to always keep your DOM tidy and you won't have to work so much about memory issues later on.

## 12.4 Text Contents

Working with text tends to be much easier than working with HTML elements, especially since there are built-in methods, that work in all browsers, for handling this behavior. Of course, there are all sorts of bugs that we end up having to work around, making these APIs obsolete in the process.

There are typically two desired scenarios: Getting the text contents out of an element and setting the text contents of an element.

W3C-compliant browsers provide a `.textContent` property on their DOM elements. Accessing the contents of this property gives you the textual contents of the element (both its direct children and descendant nodes, as well).

Internet Explorer has its own property, `.innerText`, for performing the exact-same behavior as `.textContent`.

**Listing 12.12: Using textContent and innerText.**

```
<div id="test"><b>Hello</b>, I'm a ninja!</div>
<div id="test2"></div>
<script>
window.onload = function(){
  var b = document.getElementById("test");
  var text = b.textContent || b.innerText;

  assert( text === "Hello, I'm a ninja!",
  "Examine the text contents of an element." );
  assert( b.childNodes.length === 2,
  "An element and a text node exist." );

  if ( typeof b.textContent !== "undefined" ) {
    b.textContent = "Some new text";
  } else {
    b.innerText = "Some new text";
  }

  text = b.textContent || b.innerText;

  assert( text === "Some new text", "Set a new text value." );
  assert( b.childNodes.length === 1,
    "Only one text nodes exists now." );
  };
  </script>
```

Note that when we set the `textContent/innerText` properties the original element structure is removed. So while both of these properties are very useful there are a certain number of gotchas. First, as when we discussed removing elements from the page, not having an sort of special consideration for element memory leaks will come back to bite you. Additionally, the cross-browser handling of whitespace is absolutely abysmal in these properties. No browser appears capable of returning a consistent result.

Thus, if you don't care about preserving whitespace (especially endlines) feel free to use textContent/innerText for accessing the element's text value. For setting, though, we'll need to devise an alternative solution.

### SETTING TEXT

Setting a text value comes in two parts: Emptying out the contents of the element and inserting the new text contents in its place. Emptying out the contents is straightforward - we've already devised a solution in Listing 12.10.

To insert the new text contents we'll need to use a method that'll properly escape the string that we're about to insert. An important difference between inserting HTML and inserting text is that the inserted text will have any problematic HTML-specific characters escaped. For example '<' will appear as &lt;

We can actually use the built-in `createTextNode` method, that's available on DOM documents, to perform precisely that.

**Listing 12.13: Setting the text contents of an element.**

```
  <div id="test"><b>Hello</b>, I'm a ninja!</div>
  <div id="test2"></div>
  <script>
  window.onload = function(){
    var b = document.getElementById("test");

    // Replace with your empty() method of choice
    while ( b.firstChild )
      b.removeChild( b.firstChild );

    // Inject the escaped text node
    b.appendChild( document.createTextNode( "Some new text" ) );

    var text = b.textContent || b.innerText;

    assert( text === "Some new text", "Set a new text value." );
    assert( b.childNodes.length === 1,
    "Only one text nodes exists now." );
  };
  </script>
```

#### Getting Text

To get an accurate text value of an element we have to ignore the results from textContent and innerText. The most common problem is related to endlines being unnecessarily stripped from the return result. Instead we must collect all the text node values manually to get an accurate result.

A possible solution would look like this, making good use of recursion:

**Listing 12.14: Getting the text contents of an element.**

```
<div id="test"><b>Hello</b>, I'm a ninja!</div>
<div id="test2"></div>
<script>
window.onload = function(){
  function getText( elem ) {
    var text = "";

    for ( var i = 0, l = elem.childNodes.length; i < l; i++ ) {
      var cur = elem.childNodes[i];

      // A text node has a nodeType === 3
      if ( cur.nodeType === 3 )
        text += cur.nodeValue;

      // If it's an element we need to recurse further
      else if ( cur.nodeType === 1 )
        text += getText( cur );
    }

    return text;
  }

  var b = document.getElementById("test");
  var text = getText( b );

  assert( text === "Hello, I'm a ninja!",
  "Examine the text contents of an element." );
  assert( b.childNodes.length === 2,
  "An element and a text node exist." );
};
</script>
```

In your applications if you can get away with not worrying about whitespace definitely stick with `textContent/innerText` as it'll make your life so much simpler.

## 12.5 Summary

We've taken a comprehensive look at the best ways to tackle the difficult problems surrounding DOM manipulation. While these problems should be easier than they are - the cross-browser issues introduced make their actual implementations much more difficult. With a little bit of extra work we can have a unified solution that will work well in all major browsers - which is exactly what we should strive for.

# 13 Attributes and CSS

In this chapter:

- The difference between DOM Attributes and DOM properties.
- Dealing with cross-browser attributes and styles.
- Techniques for handling element dimensions.

DOM attributes and properties are an important piece of creating a solid piece of reusable JavaScript code. They act as the primary means through which additional information can be attached to a DOM element. There likely isn't an area in the DOM that is more riddled with bug and cross-browser issues than this.

The same goes for CSS and styling of elements. Many of the complications that arise when constructing a dynamic web application come from setting and getting element styling properly. While this book won't cover all that is known about handling element styling (that's enough for an entire other book) the core essentials will be considered.

## 13.1 DOM Attributes and Expandos

When accessing the values of element attributes you have two possible options: Using the traditional DOM methods (such as `getAttribute` and `setAttribute`) or using properties of the DOM objects themselves (frequently called an 'expando' property). For example, you can see the different styles in Listing 13.1.

**Listing 13.1: An example of using traditional DOM attribute setting and an expando.**

```
<div></div>
<script>
window.onload = function(){
  var div = document.getElementsByTagName("div")[0];
```

```
    // Set the ID
    div.setAttribute("id", "somediv");
    div.id = "somediv";
  };
  </script>
```

Besides the obvious simplicity that an expando gives you (less typing is always a plus) there are five major differences between the two that need to be understood:

First, the naming of expandos are generally more consistent across browsers. If you're able to access an expando in one browser you have a good chance of it having the the same in other browsers, as well.

Listing 13.2 shows a case where there's a naming discrepancy when using normal DOM attributes.

**Listing 13.2: A naming discrepancy that occurs between browsers when using traditional DOM attribute methods.**

```
  <div class="oldclass"></div>
  <script>
  window.onload = function(){
    var div = document.getElementsByTagName("div")[0];

    // Set the class
    div.setAttribute("class", "newclass");

    // The above doesn't work in IE, this does
    div.setAttribute("className", "newclass");
  };
  </script>
```

Obviously having naming discrepancies like this can be quite frustrating so attempting to minimize the difference is generally a good idea.

Second, expandos have some name limitations as to what names they are capable of using. The ECMAScript 3.0 specification states that certain keywords aren't able to be used (such as 'for' and 'class') so alternatives must be used ('htmlFor' and 'className'). Additionally, attribute names that are two words long (e.g. 'readonly') switch to camel case ('readOnly'). Some examples of these translations can be found in Table 13.1.

Table 13.1: Cases where the expando name of an attribute is different from the original name.

| Original Name | Expando Name |
|---|---|
| for | htmlFor |
| class | className |
| readonly | readOnly |
| maxlength | maxLength |

| | |
|---|---|
| cellspacing | cellSpacing |
| rowspan | rowSpan |
| colspan | colSpan |
| tabindex | tabIndex |

Third, expando properties don't exist on XML nodes. Expando properties are primarily only available on HTML documents, thus you'll need to use the traditional DOM attribute methods. This isn't so bad because on XML documents there really doesn't exist the normal litany of naming mistakes that you see from DOM attributes in HTML documents.

You'll probably want some form of a check in your code to determine if an element (or document) is an XML element (or document). An example function can be seen in Listing 13.3.

**Listing 13.3: A function for determining if an element (or document) is from an XML document.**

```
function isXML( elem ) {
  return (elem.ownerDocument || elem)
    .documentElement.nodeName !== "HTML";
}
```

Fourth, it should be noted that not all attributes become an expando. It's generally the case for attributes that are natively specified by the browser - but not so for custom attributes specified by the user. There's a relatively simple way around this, though: If you know that you're accessing a custom attribute, just use the normal DOM attribute technique. Otherwise, you should check to see if the expando is undefined, and if so, use the normal DOM attribute technique for getting the attribute, as in Listing 13.4.

**Listing 13.4: A way for getting the value of a custom attribute value.**

```
<div custom="value"></div>
<script>
window.onload = function(){
  var div = document.getElementsByTagName("div")[0];

  var val = div.custom;

  if ( typeof val === "undefined" ) {
    val = div.getAttribute("custom");
  }
};
</script>
```

Last, and perhaps most importantly, expandos are much, much, faster than their corresponding DOM attribute operations (especially so in Internet Explorer). You can see the stark difference in Table 13.2.

Table 13.2: The results of running the expando and attribute performance tests, results in milliseconds.

| Browser | Expando (ms) | Attribute (ms) |
|---|---|---|
| Firefox 3.5 | 55 | 71 |
| Safari 4 | 11 | 15 |
| IE 6 | 103 | 1202 |

Internet Explorer 6 ends up being almost 12 times slower at traditional DOM attribute access than through an expando.

For a simple test of the speed differences, as shown in Listing 13.5, the class and id attributes are accessed in a tight loop.

**Listing 13.5: Comparing the performance difference between access attributes values through the DOM `getAttribute` method and an expando.**

```
<html>
<head>
  <title>Attribute vs. Expando Speed</title>
</head>
<body>
<div id="test" class="test"></div>
<script>
var elem = document.getElementsByTagName("div")[0];

var start = (new Date).getTime();
expando();
var test1 = (new Date).getTime() - start;

start = (new Date).getTime();
attr();
var test2 = (new Date).getTime() - start;

alert( test1 + " " + test2 );

function expando(){
  for ( var i = 0; i < 10000; i++ ) {
    if ( elem.id === "test" ) {}
    if ( elem.className === "test" ) {}
  }
}

function attr(){
  for ( var i = 0; i < 10000; i++ ) {
    if ( elem.getAttribute("id") === "test" ) {}
    if ( elem.getAttribute("class") === "test" ) {}
  }
}
</script>
</body>
</html>
```

This difference in speed can be crippling, especially in cases where a number of queries take place (like in a CSS selector engine, for example).

The end result of having these two systems (traditional DOM attributes and expandos) is a hybrid which attempts to take the best of both worlds. An example implementation can be seen in Listing 13.6.

**Listing 13.6: A simple implementation for setting and accessing attributes (using both expandos and normal DOM attributes).**

```
(function(){

  // Map expando names
  var map = {
    "for": "htmlFor",
    "class": "className",
    readonly: "readOnly",
    maxlength: "maxLength",
    cellspacing: "cellSpacing",
    rowspan: "rowSpan",
    colspan: "colSpan",
    tabindex: "tabIndex"
  };

  this.attr = function( elem, name, value ) {
    var expando = map[name] || name,
      expandoExists = typeof elem[ expando ] !== "undefined";

    if ( typeof value !== "undefined" ) {
      if ( expandoExists ) {
        elem[ expando ] = value;
      } else {
        elem.setAttribute( name, value );
      }
    }

    return expandoExists ?
      elem[ expando ] :
      elem.getAttribute( name );
  };

})();
```

It should be noted, though, that the above implementation doesn't take into account many of the cross-browser issues that occur in attribute access, which will be covered in the next section.

It should be noted that expandos are also capable of holding non-string values. You can add an extra expando to an element, holding an object, a function, or some other value, and it'll persist along with the DOM object. This will be discussed in more depth in the chapter on Events.

## 13.2 Cross-Browser Attributes

As mentioned before, writing cross-browser attribute-handling code can be quite harrowing - the number of cross-browser issues at play is nearly limitless. A few of the major, and most commonly encountered, issues are outlined in this section.

### 13.2.1 DOM ID/Name Expansion

The nastiest bug to deal with is a mis-implmentation of the original DOM code in browsers. The premise is that browsers take inputs that have a specific name or ID and add them as expandos properties to a form element. (This problem actually occurs elsewhere, as well, but this is, by far, its most prevalent form.) Both Internet Explorer and Opera take these expandos and actively overwrite any existing expandos that might already be on the DOM object.

Additionally, Internet Explorer goes a step further and sets that element as the attribute value even when you use the traditional DOM attribute methods.

An example of both of these problems can be seen in Listing 13.7.

**Listing 13.7: A case where form elements expand on to a form overwriting the original ID and action attributes.**

```
<html>
<head>
<title>Expansion Test</title>
<script>
window.onload = function(){
  var form = document.getElementById("form");

  // Both are DOM elements in Internet Explorer and Opera
  alert( form.id );
  alert( form.action );

  // Both are DOM elements in Internet Explorer
  alert( form.getAttribute("id") );
  alert( form.getAttribute("action") );
};
</script>
</head>
<body>
<form id="form" action="/">
  <input type="text" id="id"/>
  <input type="text" name="action"/>
</form>
</body>
</html>
```

The fact that it's now impossible to get at any of the attributes of the form element in a reliable manner an alternative technique for access needs to be devised.

One technique is to gain access to the DOM node representing the element attribute itself. This node ends up remaining untainted from the inner DOM elements. An example of this technique can be seen in Listing 13.8.

**Listing 13.8: Getting access to the original values of the form element attributes.**

```
<html>
<head>
<title>Expansion Test</title>
<script>
function attr( elem, name ) {
  if ( elem.nodeName.toUpperCase() === "form" &&
       elem.getAttributeNode(name) )
    return elem.getAttributeNode(name).nodeValue;
  else
    return elem.getAttribute(name);
}

window.onload = function(){
  var form = document.getElementById("form");

  // Both are 'form' and '/' accordingly
  alert( attr(form, "id") );
  alert( attr(form, "action") );
};
</script>
</head>
<body>
<form id="form" action="/">
  <input type="text" id="id"/>
  <input type="text" name="action"/>
</form>
</body>
</html>
```

If you're interested in the sort of problems that arise from these element expansions I recommend checking out Juriy Zaytsev's DOMLint tool, which is capable of analyzing a page for potential problems and Garrett Smith's write-up the issue, as a whole.

- DOMLint: http://yura.thinkweb2.com/domlint/
- Unsafe Names for HTML Form Controls: http://jibbering.com/faq/names/

### 13.2.2 URL Normalization

There's a bug in Internet Explorer that violates the principle of least surprise: When accessing an attribute that references a URL (such as href, src, or action) that URL is automatically converted from its specified form into a full, canonical, URL. For example if `href="/test/"` was specified then the value retrieved from `.href` or `.getAttribute("href")` would be the full URL: `http://example.com/test/`.

Incidentally, Internet Explorer (the only browser to fail this measure), also provides an addition to the `getAttribute` method. If you pass in an extra attribute at the end with a value of '2' it'll leave the URL value as it was originally entered. When can use this fact to develop a method that handles this workaround, as in Listing 13.9.

**Listing 13.9: An example workaround for handling attributes that URLs in them.**

```
<html>
<head>
<title>Attribute URL Test</title>
<script>
(function(){
  var div = document.createElement("div");
  div.innerHTML = "<a href='/'></a>";

  var urlok = div.firstChild.href === "/";

  this.urlAttr = function(elem, name, value) {
    if ( typeof value !== "undefined" ) {
      elem.setAttribute(name, value);
    }

    return urlok ?
      elem[ name ] :
      elem.getAttribute(name, 2);
  };
})();

window.onload = function(){
  var a = document.getElementById("a");

  // Alerts out the full URL 'http://example.com/'
  alert( a.getAttribute("href") );

  // Alerts out the input '/'
  alert( urlAttr( "href", a ) );
};
</script>
</head>
<body>
  <a id="a" href="/"></a>
</body>
</html>
```

A brief note: Be sure that you don't use `getAttribute(..., 2)` indiscriminately. Older versions of Opera would crash whenever the extra '2' was passed in (for no apparent reason).

### *13.2.3 Style Attribute*

One attribute that's particularly challenging to set and get the value of is the style attribute of an element. HTML DOM elements have a style object property which you can access to gain information about the style information of an element (e.g. `elem.style.color`) however if you want to get the full style string that was set on the element it becomes much more challenging (`<div style='color:red;'></div>`).

To start, expandos don't work here because the 'style' expando is already taken up by the 'style' object. So while `get/setAttribute("style")` work in most browsers, it doesn't work in Internet Explorer. Instead, IE has a property of the style object which you can get/set to affect the style attribute: `elem.style.cssText`.

Listing 13.10 has an example of a method dedicated to accessing and setting the style attribute of a DOM element.

**Listing 13.10: Create a method for getting and setting the style attribute string across browsers. (References code from Listing 13.3.)**

```
<html>
<head>
<title>Style Attribute Test</title>
<script>
(function(){
  // Create a dummy div to test access the style information
  var div = document.createElement("div");
  div.innerHTML = '<b style="color:red;"></b>';

  // Get the style information from getAttribute
  // (IE uses .cssText insted)
  var hasStyle = /red/.test( div.firstChild.getAttribute("style") );

  this.styleAttr = function(elem, value) {
    var hasValue = typeof value !== "undefined";

    // Use the isXML method from Listing 13.3
    if ( !hasStyle && !isXML(elem) ) {
      if ( hasValue ) {
        elem.style.cssText = value + "";
      }

      return elem.style.cssText;
    } else {
      if ( hasValue ) {
        elem.setAttribute("style", value);
      }

      return elem.getAttribute("style");
    }
  };
})();

window.onload = function(){
  var a = document.getElementById("a");

  // Fails in Internet Explorer
  alert( a.getAttribute("style") );

  // Alerts out "color:red;"
  alert( styleAttr( a ) );
};
</script>
</head>
<body>
  <a id="a" href="/" style="color:red;"></a>
</body>
</html>
```

While directly getting and setting the style attribute is comparatively uncommon (compared to accessing/mutating the style property object directly) it is still quite useful for getting a full serialization of all the style information currently on an element.

### 13.2.4 Type Attribute

There's another Internet Explorer gotcha related to the type attribute of form inputs - to which there isn't a reasonable fix. Once an input element has been inserted into a document its type attribute can no longer be changed (and, in fact, throws an exception if you attempt to change it).

An example of the problem can be seen in Listing 13.11.

**Listing 13.11: Attempting to change an input element's type attribute after it has already been inserted into a document.**

```
<html>
<head>
<title>Type Attribute Test</title>
<script>
window.onload = function(){
  var input = document.createElement("input");
  input.type = "text";

  document.getElementById("form").appendChild( input );

  // Errors out in Internet Explorer
  input.type = "hidden";
};
</script>
</head>
<body>
<a id="a" href="/"></a>
<form id="form" action="/">
  <input type="text" id="id"/>
  <input type="text" name="action"/>
</form>
</body>
</html>
```

There are two possible stopgap solutions:

1. Create a new input and copy over any existing properties, attributes, and event handlers (assuming that you've been tracking the event handlers to begin with). Unfortunately any

references to the old element will still exist, you would need to have a way to change all of those, as well.
2. Simply reject any attempts to change the type attribute at the API level.

jQuery employs the second method (throwing an informative exception if the users attempts to change the type attribute after it has already been inserted into the document). Obviously this isn't a terribly good 'solution' but at least the user experience is consistent across all platforms.

### 13.2.5 Tab Index

Determining the tab index of an element is another weird attribute - and one where there's little consensus at to how it should work. While it's possible to get the tab index of an element using the traditional means (`.tabIndex` and `.getAttribute("tabindex")`) the result is quite inconsistent for elements that haven't had a tab index explicitly defined.

For example, a div that has no tab index specified should return undefined - but Internet Explorer 6 and 7 return 0 and Firefox 2 and 3 return -1 (when using `.tabIndex`).

There is a workaround, though. We can use the `getAttributeNode` method to get at the node representing the tabindex attribute. Once we have that we can check its 'specified' property to see if the user has ever specified a tab index (this will be false if the browser is trying to use guesswork and give you a bogus number instead of undefined).

Listing 13.12 shows how we can use this technique to get at the correct tab index value and how we can report the correct tab index for attributes that should (or shouldn't) have one.

**Listing 13.12: A method for getting the correct tab index of an element.**

```
(function(){
  var check = /^(button|input|object|select|textarea)$/i,
      a = /^(a|area)$/i;

  this.getIndex = function(elem){
    var attributeNode = elem.getAttributeNode("tabIndex");
    return attributeNode && attributeNode.specified ?
      attributeNode.value :
      check.test( elem.nodeName ) ||
          a.test( elem.nodeName ) && elem.href ?
        0 :
        undefined;
  };
})();
```

This is a complex issue and is one that is especially important in the world of usability and accessibility. The Fluid Project has been doing a lot of research into the area and has written up some comprehensive information on the subject matter.

- http://fluidproject.org/blog/2008/01/09/getting-setting-and-removing-tabindex-values-with-javascript/

### 13.2.6 Node Name

While this isn't related to an attribute, per se, determining the name of a node can be slightly tricky. Specifically, the case sensitivity of the name changes depending upon which type of document you are in. If it's a normal HTML document then the `nodeName` property will return the name of the element in all uppercase (e.g. "HTML" or "BODY"). However, if it's in an XML or XHTML document then the `nodeName` will return the name as specified by the user (meaning that it could be lowercase, uppercase, or a mix of both).

If you have a DOM node, and you know which `nodeName` you're looking for, you can something like in Listing 13.13.

**Listing 13.13: Normalizing the nodeName of an element in order to simplify node name checks.**

```
<div></div>
<ul></ul>
<script>
window.onload = function(){
  var all = document.getElementsByTagName("*")[0];

  for ( var i = 0; i < all.length; i++ ) {
    var nodeName = all[i].nodeName.toLowerCase();
    if ( nodeName === "div" || nodeName === "ul" ) {
      all[i].className = "found";
    }
  }
};
</script>
```

If you already know all the documents in which you'll be checking `nodeNames` then you probably won't have to worry about this case sensitivity. However, if you're writing reusable code that could run in any environment, it's best to be adaptive and gracefully handle the different pages.

## 13.3 CSS

As with attributes, getting and setting CSS attributes can be quite tricky. Similarly to attributes we are, again, presented with two APIs for handling style values. The most commonly used API is the `.style` property of an element (which is an object holding properties corresponding to set style properties). On top of this there is an API for accessing the current, computed, style information of an element.

Style information (located on the `.style` property of a DOM element object) comes from two places: If an inline style attribute is declare (e.g. `style="color:red;"`) then that style information will be put in the style object. Also, if a user sets a value on the style object it will affect the display of the element and store the value on the object.

No values from internal, or external, stylesheets will be available on the element's style object (computed style will be required to access that information). Listing 13.14 has an example of the limitations of the style object.

**Listing 13.14: An example of where the properties on an element's style property come from.**

```
<div style="color:red;">text</div>
<style>div { font-size: 10px; }</style>
<script>
window.onload = function(){
  var div = document.getElementsByTagName("div")[0];

  // Will alert 'red' or '#ff0000' (in Opera)
  alert( div.style.color );

  // Will be an empty alert
  alert( div.style.fontSize );

  div.style.backgroundColor = "blue";

  // Will alert 'blue' or '#0000ff' (in Opera)
  alert( div.style.backgroundColor );
};
</script>
```

It should be noted that any values in an element's style object will take precedence over anything specified by a stylesheet (even if the stylesheet rule uses `!important`). In the end it is the ultimate arbiter of what an element looks like. For this reason, in your CSS property access code, you should always check the value of the `.style` property first as it'll have the most definite state in it.

With CSS attributes there are relatively few cross-browser difficulties when it comes to accessing the values provided by the browser. CSS attributes frequently use names like "font-weight" and "background-color" - all of which need to be converted into camel case in order to be accessed correctly (two examples of this were shown in Listing 13.14).

Note browser-specific attributes. They generally start with a prefix that corresponds to the browser, for example: `"-moz-border-radius"`. When accessing the value of this attribute through JavaScript it'll need to be camel cased as such: `"MozBorderRadius"`. However, while Mozilla allows you to both read and write custom attributes WebKit only allows you to write to them (at least via `.style`, it allows you to read the computed value).

Some simple code for converting the attribute name to camel case can be see in Listing 13.15.

**Listing 13.15: A simple method for access style information of an element, handling correct camel-casing.**

```
<div style="color:red;font-size:10px;-moz-border-radius:3px;"></div>
<script>
function style(elem, name, value){
  name = name.replace(/-([a-z])/ig, function(all, letter){
    return letter.toUpperCase();
  });

  if ( typeof value !== "undefined" ) {
    elem.style[ name ] = value;
  }

  return elem.style[ name ];
}

window.onload = function(){
  var div = document.getElementsByTagName("div")[0];

  // Will alert 'red' or '#ff0000' (in Opera)
  alert( style(div, "color") );

  // Will alert '10px'
  alert( style(div, "font-size") );

  // Will alert '3px 3px 3px 3px' in Firefox
  alert( style(div, "-moz-border-radius") );
};
</script>
```

### 13.3.1 Float Property

The one, major, naming bug that exists with CSS attributes relates to how the float attribute is called. Since the float property conflicts with the built in float keyword in JavaScript the browsers have to provide an alternative name.

Nearly all browsers choose to use `'cssFloat'` as the alternative name whereas Internet Explorer uses `'styleFloat'`.

A simple function for handling these differences can be seen in Listing 13.16.

**Listing 13.16: An example of retrieving the float style attribute across browsers.**

```
<html>
<head>
<title>Float Style Test</title>
<script>
(function(){
  var div = document.createElement("div");
  div.innerHTML = "<div style='float:left;'></div>";

  var floatName = div.firstChild.style.cssFloat === "left" ?
    "cssFloat" :
    "styleFloat";

  this.floatStyle = function(elem, value) {
    if ( typeof value !== "undefined" ) {
      elem.style[ floatName ] = value;
    }

    return elem.style[ floatName ];
  };
})();

window.onload = function(){
    var div = document.getElementById("div");

    // Alerts out 'left'
    alert( floatStyle( div ) );
  };
  </script>
  </head>
  <body>
    <div style="float:left;"></div>
    <a id="a" href="/"></a>
  </body>
  </html>
```

As seen in Listing 13.16 it's pretty easy to use feature detection to make sure that the property exists before using attempting to access it. After that we can make a fairly safe assumption that if the spec-compatible `cssFloat` doesn't work then `styleFloat` will.

### 13.3.2 Raw Pixel Values

The last point to consider when setting a style value is the setting of numbers to properties that normal consume pixels. When setting a number into a style property you must specify the unit in order for it to work across all browsers.

```
// Works across browsers
elem.style.height = "10px";
elem.style.height = 10 + "px";

// Doesn't work across browsers
elem.style.height = 10;
```

We can work around this by watching for when a number comes in as a value and automatically add on a the "px" suffix. Of course, we have to be careful of cases where we actually don't want to add a suffix, for which there are a few style attributes:

- z-index
- font-weight
- opacity
- zoom
- line-height

Additionally, when attempting to read a pixel value out of a style attribute the `parseFloat` method should be used.

Altogether a unified method can be created for handling both of these cases, as in Listing 13.17.

**Listing 13.17: A method for getting and setting pixel-based style attributes using numbers.**

```
<html>
<head>
<title>Pixel Style Test</title>
<script>
(function(){
  var check = /z-?index|font-?weight|opacity|zoom|line-?height/i;

  this.pxStyle = function(elem, name, value) {
    var nopx = check.test( name );

    if ( typeof value !== "undefined" ) {
      if ( typeof value === "number" ) {
        value += nopx ? "" : "px";
      }
      elem.style[ name ] = value;
    }

    return nopx ?
      elem.style[ name ] :
      parseFloat( elem.style[ name ] );
  };
})();

window.onload = function(){
  var div = document.getElementById("div");

  pxStyle( div, "top", 5 );

  // Alerts out '5'
  alert( pxStyle( div, "left" ) );
};
</script>
</head>
<body>
  <div style="top:10px;left:5px;"></div>
</body>
</html>
```

These concerns cover most of what you have to worry about when it comes to handling the `.style` property of an element. There's still one, large, missing gap though: Retrieving the current computed style of an element.

### 13.3.3 Computed Style

The computed style of an element is a combination of the styles applied to it via stylesheets, the inline `style` attribute, and any manipulations of the `.style` property.

The primary API specified by the W3C (and implemented by all browsers but Internet Explorer) is the `getComputedStyle` method on the document's defaultView. The method takes an element and returns an interface through which property queries can be made. The interface provides the `getPropertyValue( name )` for retrieving the computed style of a specific property.

Contrary to the `.style` object the `getPropertyValue( name )` method only accepts method names that are in the original CSS property style (e.g. "font-size").

As alluded to before, Internet Explorer has a completely different technique for accessing the computed style of an element. Incidentally, it's also a dramatically simpler API. A single .currentStyle property is provided on all elements and it behaves exactly like the .style property, except providing live, computed, style information.

Listing 13.18 has an example function which is able to access the computed style information for an element using the W3C-compatible API or Internet Explorer's `currentStyle` property.

**Listing 13.18: A function for accessing computed style information using the W3C-compatible API or Internet Explorer's `currentStyle`.**

```
<div style="color:red;">text</div>
<style>div { font-size: 10px; display: inline; }</style>
<script>
function computedStyle( elem, name ) {
  var defaultView = elem.ownerDocument.defaultView;

  if ( defaultView && defaultView.getComputedStyle ) {
    var computedStyle = defaultView.getComputedStyle( elem, null );

    if ( computedStyle ) {
      name = name.replace(/([A-Z])/g, "-$1").toLowerCase();
      return computedStyle.getPropertyValue( name );
    }
  } else if ( elem.currentStyle ) {
    name = name.replace(/-([a-z])/ig, function(all, letter){
      return letter.toUpperCase();
    });

    return elem.currentStyle[ name ];
  }
}

window.onload = function(){
  var div = document.getElementsByTagName("div")[0];

  // Will alert '10px'
  alert( computedStyle( div, "font-size" ) );

  // Will alert '10px'
  alert( computedStyle( div, "fontSize" ) );

  // Will alert 'inline'
  alert( computedStyle( div, "display" ) );
};
</script>
```

Internet Explorer's `currentStyle` property has an added benefit. The W3C way of accessing a computed value will always return a numerical result in pixels. Thus if you have a font-size measured in '2em' the computed font-size that you'll get back will be something like '14.45px'. This isn't terribly useful, especially if you want to get at the original value provided by a stylesheet.

Internet Explorer's `currentStyle` has no such problem. A font size of '2em' will come out as '2em'. This ends up being simultaneously useful and frustrating. Since the API is different from the W3C-style API (that every other browser provides) we're forced to bang it into shape (forcing it to return pixel values whenever it returns numbers based in another unit).

There's one very interesting hack written by Dean Edwards that is able to handle quite a few of the common conversion cases. The code for it is shown in Listing 13.19.

- Conversion trick by Dean Edwards: http://erik.eae.net/archives/2007/07/27/18.54.15/#comment-102291

**Listing 13.19: Dean Edward's technique for getting at pixel values for different united properties.**

```
// If we're not dealing with a regular pixel number
// but a number that has a different ending, we need to convert it
if ( /^[0-9.]+(em|ex|pt|%)?$/i.test( value ) ) {
  // Remember the original values
  var left = elem.style.left, rsLeft = elem.runtimeStyle.left;

  // Put in the new values to get a computed value out
  elem.runtimeStyle.left = elem.currentStyle.left;
  elem.style.left = value || 0;
  ret = elem.style.pixelLeft + "px";

  // Revert the changed values
  elem.style.left = left;
  elem.runtimeStyle.left = rsLeft;
}
```

This technique works by using a couple Internet Explorer-specific features. We're already familiar with `.currentStyle` but there's also `.runtimeStyle` and `.style.pixelLeft`.

`.runtimeStyle` is analogous to `.style` with one exception: Any style properties set using .runtimeStyle.prop will override `.style.prop` (which still leaving the value contained in `.style.prop` intact).

`.pixelLeft` (and Top, Bottom, Right, Height, and Width) are properties for directly accessing the pixel value of the respective CSS properties. (Note: These could be used as alternatives to even using Dean Edwards' technique in the first place for these properties - Dean's technique is more useful for things like font-size, line-height, etc.).

The technique works by setting the current computed left to the runtime left ('left' is chosen arbitrarily - it could also work with top, right, etc.) - this will keep the positioning of the element intact while we compute the pixel value. The pixel value is computed by setting the `.style.left` and then reading the resulting pixel value out using `.style.pixelLeft`.

Dean's technique isn't perfect - especially when attempting to handle percentages on elements that are a child of the body element (they end up expanding to consume a large portion of the document - 10% becomes 200px, for example) but it ends up working well enough to handle most of the common use cases.

#### AUTO VALUES

One aspect that is particularly hard to solve using Internet Explorer's `currentStyle` technique is that of 'auto' values. For example if an element has no specific height or width defined then it defaults to 'auto' (automatically expanding and contracting based upon its contents). Thankfully it's possible to find the correct value of those properties in other ways, which we'll discuss later in this chapter.

However there do still exist some properties that fall prey to this problem, like margin and font-size.

If you want to figure out the current pixel value that corresponds to an auto margin, as it currently appears, you'll have to perform some extra calculations and gymnastics - many of which will have to depend upon the specific code base. For example, figuring out the pixel margin of an element that's center-aligned with a margin of auto would be a piece of work in and of itself.

Font size can actually be a little bit easier since a font-size of auto corresponds to a size of 1em. You can work your way up the tree to figure out precisely what '1em' means (in pixel values) or use Dean's trick from the previous section.

One library that obsessively tackles the problem of computed font size is called CSSStyleObject and can be found here:

- http://www.strictly-software.com/CSSStyleObject.asp

#### CATCH-ALL PROPERTIES

CSS properties that are actually amalgams of sub-properties (like margin, padding, border, and background-position) don't provide consistent results across all browsers (more than likely they simply don't provide any results).

Thus, in order to figure out what the actual border is for an element you'll need to check the border-top-width, border-bottom-width, border-left-width, and border-right-width to get a clear picture.

It's a hassle, especially when all four values have the same result (and especially when something like `border: 5px;` was used in the stylesheet directly) but that's the hand we've been dealt.

### 13.3.4 Height and Width

As mentioned before, when looking at properties that could have an auto value, it was mentioned that height and width are two properties that frequently have these values.

Because of this we'll need to utilize a different technique to get at the actual height and width of an element.

Thankfully the `offsetHeight` and `offsetWidth` properties provide just that: A fairly reliable place to access the correct height and width of an element. The numbers returned by the two properties include the padding of the element as well. This information may be desired (if you're attempting to position an element over another one, for example, you'll want to leave the padding in). In reality you'll probably want to have a few methods for accessing an elements height or width (height with no padding, height with padding, and height with padding and border).

There is one gotcha, though: If you want to find out the height and width of an element that is hidden (by its display property being set to 'none'), offsetWidth and offsetHeight will both return 0. One way to workaround this problem is to temporarily make the element visible, but in a hidden state).

We can achieve this by setting the `display` property to `block` (making it visible and able to be measured) and immediately set its `visibility` to `hidden` (making it invisible to the user) and finally set its `position` to `absolute` (taking it out of the flow of the document, stopping it from pushing any other elements out of the way).

Listing 13.20 has an example of setting these properties on an element, getting its height or width, and immediately setting the properties back.

**Listing 13.20: An example of getting the height and width of an element (both visible and invisible).**

```
<html>
<head>
  <title>Height/Width Test</title>
  <script>
  (function(){

    this.height = function( elem ) {
      return this.offsetHeight ||
        swap( elem, props, function(){
          return elem.offsetHeight;
        });
    };

    this.width = function( elem ) {
      return this.offsetWidth ||
        swap( elem, props, function(){
          return elem.offsetWidth;
        });
    };

    var props = {
      position: "absolute",
      visibility: "hidden",
      display: "block"
    };
```

```
      function swap( elem, options, callback ) {
        var old = {}, ret;
        // Remember the old values, and insert the new ones
        for ( var name in options ) {
          old[ name ] = elem.style[ name ];
          elem.style[ name ] = options[ name ];
        }

        ret = callback();

        // Revert the old values
        for ( var name in options ) {
          elem.style[ name ] = old[ name ];
        }

        return ret;
      }

    })();

    window.onload = function(){
      var block = document.getElementById("block");
      var none = document.getElementById("none");

      // Both will alert out the same values
      alert( width(block) + " " + height(block) );
      alert( width(none) + " " + height(none) );
    };
    </script>
  </head>
  <body>
    <div id="block">Test</div>
    <div id="none" style="display:none;">Test</div>
  </body>
  </html>
```

One thing to note in the code of Listing 13.20 is the use of `offsetHeight` or `offsetWidth` as a crude means of determining the visibility of an element. As it turns out this serves to be an incredibly efficient way to do just that. We can assume that any element that has both an `offsetHeight` and `offsetWidth` not equal to zero then it must be visible. Likewise if they're both zero then it's definitely not visible. Ambiguity comes in when either one or the other is not zero (but we can fall back to our `computedStyle` method from Listing 13.18).

The one gotcha that exists is that a tr element in Internet Explorer will always return zero (even if it's visible). Thus we work around this by always checking the computed style of that element.

Listing 13.21 has a sample method for determining the visibility of an element.

**Listing 13.21: An example of getting the visibility of an element (both visible and invisible). Uses code from Listing 13.18.**

```
<html>
<head>
  <title>Visibility Test</title>
  <script>
  function isVisible( elem ) {
    var isTR = elem.nodeName.toLowerCase() === "tr",
      w = elem.offsetWidth, h = elem.offsetHeight;
    return w !== 0 && h !== 0 && !isTR ?
      true :
      w === 0 && h === 0 && !isTR ?
        false :
        computedStyle( elem, "display" ) === "none";
  }

  window.onload = function(){
    var block = document.getElementById("block");
    var none = document.getElementById("none");

    // Alerts out 'true'
    alert( isVisible(block) );

    // Alerts out 'false'
    alert( isVisible(none) );
  };
  </script>
</head>
<body>
  <div id="block">Test</div>
  <div id="none" style="display:none;">Test</div>
</body>
</html>
```

While height and width do end up needing to be handled in a unique manner that means through which that occurs isn't overly difficult. Additionally the results work across all platforms without much browser-specific trickery.

## 13.3.5 Opacity

Opacity is a special case that needs to be handled differently in Internet Explorer from other browsers. All other modern browsers support the opacity CSS property whereas Internet Explorer uses its proprietary alpha filter.

An example of how the rules might be defined in a stylesheet:

```
opacity: 0.5;
filter: alpha(opacity=50);
```

The problem would be pretty simple if alpha was the only type of filter allowed, but it's not (there are many different filters, including transformations, available in Internet Explorer). Thus we'll have to do some extra parsing to arrive at our desired result.

Listing 13.21 has an example of how to handle opacity across all browsers.

**Listing 13.21: An example of retrieving the opacity style attribute across browsers.**

```
<html>
<head>
<title>Opacity Style Test</title>
<script>
(function(){
  var div = document.createElement("div");
  div.innerHTML = "<div style='opacity:.5;'></div>";

  // If .5 becomes 0.5, we assume the browser knows how
  // to handle native opacity
  var native = div.firstChild.style.opacity === "0.5";

  this.opacityStyle = function(elem, value) {
    if ( typeof value !== "undefined" ) {
      if ( native ) {
        elem.style.opacity = value;
      } else {
        // .filter may not exist
        elem.style.filter = (elem.style.filter || "")
          // Need to replace any existing alpha
          .replace( /alpha\([^)]*\)/, "" ) +
          // If we have a number, set that as the value (0-100)
          (parseFloat( value ) + '' == "NaN" ?
            "" :
            "alpha(opacity=" + value * 100 + ")");
      }

      elem.style[ floatName ] = value;
    }

    if ( native ) {
      return elem.style.opacity;
    } else {
      // Only attempt to get a value if one might exist
      var match = elem.style.filter.match(/opacity=([^)]*)/);
      return match ?
        (parseFloat( match[1] ) / 100) + "" :
        "";
    }
  };
})();

window.onload = function(){
  var div = document.getElementById("div");

  // Alerts out '0.5'
  alert( opacityStyle( div ) );
};
```

```
</script>
</head>
<body>
  <div style="opacity:0.5;filter:alpha(opacity=50);"></div>
</body>
</html>
```

Getting at the computed opacity works in a very similar manner. Instead of access the `.style.filter` property we look at `.currentStyle.filter` instead.

The only other change to accessing computed opacity is that when no value is found (in Listing 13.21 we just returned an empty string) we need to return "1" instead (since the default value for opacity is completely opaque).

### 13.3.6 Colors

Handling color values in CSS can be tricky. When accessing them via `.style` you're at the mercy of whatever names/syntaxes the user chose. When you're accessing them via the different computed style methods there is no particular census around what values the method should return.

Because of this, any attempts to gain access to the useful parts of a color (presumably its red, blue, green color channel information) will require some legwork.

In total there are five separate ways in which color information can be represented:

- `rgb(R,G,B)` - Where R, G, B are numbers from 0 to 255.
- `rgb(R%,G%,B%)` - Where R, G, B are numbers from 0% to 100%.
- `#RRGGBB` - Where RR, GG, BB are hexadecimal representations of the numbers 0 through 255.
- `#RGB` - Where R, G, B are hexadecimal representations similar to the previous one, but in shorthand (e.g. #F54 is equal to #FF5544).
- `red, blue`, etc. - The name representing a set of RGB values.

All together this can result in a lot of code dedicated to handling these particulars. There are a lot of implementations floating around that tackle the same problem, one such one is shown in Listing 13.22. This code comes from the jQuery Color plugin with code written by Blair Mitchelmore (for the highlightFade plugin) and Stefan Petre (for the Interface plugin).

- jQuery Color Plugin: http://plugins.jquery.com/project/color
- highlightFade Plugin: http://jquery.offput.ca/highlightFade/
- Interface Plugin: http://interface.eyecon.ro/

**Listing 13.22: A method for converting a string value into an array of RGB values (including the conversion of color names to values - full list is available in the source links).**

```
var num = /rgb\(\s*([0-9]+)\s*,\s*([0-9]+)\s*,\s*([0-9]+)\s*\)/,
    pc = /rgb\(\s*([0-9.]+)%\s*,\s*([0-9.]+)%\s*,\s*([0-9.]+)%\s*\)/,
    hex = /#([a-fA-F0-9]{2})([a-fA-F0-9]{2})([a-fA-F0-9]{2})/,
    hex2 = /#([a-fA-F0-9])([a-fA-F0-9])([a-fA-F0-9])/;

// Parse strings looking for color tuples [255,255,255]
function getRGB(color) {
  var result;

  // Look for rgb(num,num,num)
  if (result = num.exec(color))
    return [parseInt(result[1]), parseInt(result[2]),
      parseInt(result[3])];

  // Look for rgb(num%,num%,num%)
  if (result = pc.exec(color))
    return [parseFloat(result[1])*2.55, parseFloat(result[2])*2.55,
      parseFloat(result[3])*2.55];

  // Look for #a0b1c2
  if (result = hex.exec(color))
    return [parseInt(result[1],16), parseInt(result[2],16),
      parseInt(result[3],16)];

  // Look for #fff
  if (result = hex2.exec(color))
    return [parseInt(result[1]+result[1],16),
      parseInt(result[2]+result[2],16),
      parseInt(result[3]+result[3],16)];

  // Otherwise, we're most likely dealing with a named color
  return colors[color.replace(/\s+/g, "").toLowerCase()];
}

// Map color names to RGB values
var colors = {
  aqua:[0,255,255],
  azure:[240,255,255],
  beige:[245,245,220],
  black:[0,0,0],
  blue:[0,0,255],
  // ... snip ...
  silver:[192,192,192],
  white:[255,255,255],
  yellow:[255,255,0]
};
```

Some libraries choose to not handle this functionality natively and push it off into an add-on (as is the case in jQuery). Generally color conversion is most useful when doing animations (animating from one color to another) so its inclusion really only occurs when that functionality is desired.

## 13.4 Summary

Getting and setting DOM attributes and CSS is the area of JavaScript development most fraught with cross-browser compatibility issues. Thankfully the issues can be handled in a way that is particularly cross-browser compliant without resorting to browser-specific code.

# 14 Events

In this chapter:

- Techniques for binding events
- Custom events and event triggering
- Event bubbling and delegation

DOM event management is a problem that should be relatively simple, all browsers provide relatively stable APIs for managing events (although, with different implementations). Unfortunately the features provided by browsers are insufficient for most of the tasks that need to be handled by sufficiently-complex applications. Because of these shortcomings the end result is a near-full duplication of the existing DOM APIs in JavaScript.

## 14.1 Binding and Unbinding Event Handlers

When binding and unbinding event handlers you typically use with the browser's native methods: addEventListener and removeEventListener for DOM-compliant browsers and attachEvent and detachEvent in Internet Explorer. For the most part the two techniques behave very similarly but with one exception: Internet Explorer's event system doesn't provide a way to listen for the capturing stage of an event, only the bubbling phase (after the event occurs and begins to traverse back up the DOM tree).

Additionally, Internet Explorer's event implementation doesn't properly set a context on the bound handler (thus if you try to access the 'this' inside the handler you'll just get the global object instead of the current element). For this reason the handler provided by the user shouldn't be bound directly to the element in Internet Explorer and should, instead, bind an intermediary handler that corrects the context before executing the handler.

These techniques can be seen together in Listing 14.1.

**Listing 14.1: Simple methods for binding and unbinding an event handler while preserving the context. Also a proxy method for changing the context for any given function.**

```
<script>
(function(){
  if ( document.addEventListener ) {
    this.addEvent = function( elem, type, fn ) {
      elem.addEventListener( type, fn, false );
      return fn;
    };

    this.removeEvent = function( elem, type, fn ) {
      elem.removeEventListener( type, fn, false );
    };

  } else if ( document.attachEvent ) {
    this.addEvent = function( elem, type, fn ) {
      var bound = function() {
        return fn.apply( elem, arguments );
      };

      elem.attachEvent( "on" + type, bound );
      return bound;
    };

    this.removeEvent = function( elem, type, fn ) {
      elem.detachEvent( "on" + type, fn );
    };
  }
})();

addEvent( window, "load", function() {
  var li = document.getElementsByTagName("li");
  for ( var i = 0; i < li.length; i++ ) (function(elem){
    var handler = addEvent( elem, "click", function() {
      this.style.backgroundColor = "green";
      removeEvent( elem, "click", handler );
    });
  })(li[i]);
});
</script>
<ul>
<li>Click</li>
<li>me</li>
<li>once.</li>
</ul>
```

The code in Listing 14.1 exposes three new methods: `addEvent`, `removeEvent`, and `proxy`.`addEvent` and `removeEvent` work exactly as advertised: You provide a DOM element, document, or window object, the type of the event, and the function that you wish to bind/unbind and it'll attach an event to that object during the bubbling phase.

The one bit of trickiness is with the context-fixing function for Internet Explorer. Since we've used an intermediary function to correct the context in Internet Explorer the normal handler will no longer be able to remove the same handler function that we put in. Instead the addEvent function now returns the new function that has been bound (which we will use to unbind the handler later on). A better workaround is to use a separate object to store all the event handlers, which we will discuss in the next section.

## 14.2 The Event Object

Many browsers, especially Internet Explorer, miss out on a number of the W3C event object properties and methods. The only reasonable way to work around this is to create a new object that simulates the browser's native event object, fixing issues on it where appropriate (it's not always possible to modify the existing object, many properties can't be overwritten).

To start, it's important to note that the event object isn't passed in as the first argument to the bound handler in Internet Explorer instead it's contained within a global object named `event`. Having a single event object makes it all the more important to clone it and transfer the properties to a new object, since once a new event begins the old event object will go away.

A function for event normalization can be found in Listing 14.2.

**Listing 14.2: Implement some basic event object normalization.**

```
function fixEvent( event ) {
  if ( !event || !event.stopPropagation ) {
    var old = event || window.event;

    // Clone the old object so that we can modify the values
    event = {};

    for ( var prop in old ) {
      event[ prop ] = old[ prop ];
    }

    // The event occurred on this element
    if ( !event.target ) {
      event.target = event.srcElement || document;
    }

    // Handle which other element the event is related to
    event.relatedTarget = event.fromElement === event.target ?
      event.toElement :
      event.fromElement;

    // Stop the default browser action
    event.preventDefault = function() {
      event.returnValue = false;
      event.isDefaultPrevented = returnTrue;
    };

    event.isDefaultPrevented = returnFalse;

    // Stop the event from bubbling
    event.stopPropagation = function() {
        event.cancelBubble = true;
        event.isPropagationStopped = returnTrue;
      };

      event.isPropagationStopped = returnFalse;

      // Stop the event from bubbling and executing other handlers
      event.stopImmediatePropagation = function() {
        this.isImmediatePropagationStopped = returnTrue;
        this.stopPropagation();
      };

      event.isImmediatePropagationStopped = returnFalse;

      // Handle mouse position
      if ( event.clientX != null ) {
        var doc = document.documentElement, body = document.body;

        event.pageX = event.clientX +
          (doc && doc.scrollLeft || body && body.scrollLeft || 0) -
          (doc && doc.clientLeft || body && body.clientLeft || 0);
        event.pageY = event.clientY +
          (doc && doc.scrollTop || body && body.scrollTop || 0) -
          (doc && doc.clientTop || body && body.clientTop || 0);
      }

      // Handle key presses
      event.which = event.charCode || event.keyCode;

      // Fix button for mouse clicks:
      // 0 == left; 1 == middle; 2 == right
      if ( event.button != null ) {
        event.button = (event.button & 1 ? 0 :
          (event.button & 4 ? 1 :
            (event.button & 2 ? 2 : 0)));
      }
    }

    return event;

    function returnTrue() { return true; }
    function returnFalse() { return false; }
  }
```

The `fixEvent` function in Listing 14.2 handles many of the common discrepancies between the W3C DOM event object and the one provided by Internet Explorer. You could use it within your handlers to get a usable event object back.

A few of the concerns that are fixed in this process:

- `.target` The commonly-used property denoting the original source of the event. IE frequently stores this in `.srcElement`.
- `.relatedTarget` comes into use when it's used on an event that works in

conjunction with another element (such as "mouseover" or "mouseout"). The `.toElement` and `.fromElement` are IE's counterparts.

• `.preventDefault()` doesn't exist in the IE implementation, which would normally prevent the default browser action from occurring, instead the `.returnValue` property needs to be set to false.

• `.stopPropagation()` also doesn't exist, which would normally stop the event from bubbling further up the tree. Setting the `.cancelBubble` property to true will make this happen.

• `.pageX` and `.pageY` don't exist in IE (provide the position of the mouse relative to the whole document) but can be easily duplicated using other information (`clientX/Y` provides the position of the mouse relative to the window, `scrollTop/Left` gives the scrolled position of the document, and `clientTop/Left` gives the offset of the document itself - combining these three will give you the final `pageX/Y` values).

• `.which` is equivalent to the key code pressed during a keyboard event, this can be duplicated by accessing the `.charCode` and `.keyCode` properties.

• `.button` matches the mouse button clicked by the user on a mouse event. Internet Explorer uses a bitmask (1 for left client, 2 for right click, 4 for middle click) so it needs to be converted to their equivalent values (1, 2, and 3).

Some of the best information on the DOM event object and its cross-browser capabilities can be found on the Quirksmode compatibility tables:

• Event object compatibility: http://www.quirksmode.org/dom/w3c_events.html

• Mouse position compatibility: http://www.quirksmode.org/dom/w3c_cssom.html#mousepos

Additionally issues surrounding the nitty-gritty of keyboard and mouse event object properties can be found in the excellent JavaScript Madness guide:

- Keyboard events: http://unixpapa.com/js/key.html
- Mouse events: http://unixpapa.com/js/mouse.html

### 14.2.1 Handler Management

For a number of reasons it's not advisable to bind an event handler directly to an element, you'll want to use an intermediary event handler instead and store all the handlers in a separate object. These reasons include:

• Normalizing the context of handlers.

- Fixing the properties of event objects.
- Handling garbage collection of bound handlers.
- Triggering or removing some handlers by a filter.
- Unbinding all events of a particular type.
- Cloning of event handlers.

You need to have access to the full list of the handlers bound to an element in order to be achieve all of these points. At which time it really makes the most sense to just avoid directly binding the events.

The best way to manage the handlers associated with a DOM node is to give each node that your work a unique ID and then store all data in a centralized object. Keeping the data in a central store, in this manner, helps to avoid potential memory leaks in Internet Explorer (attaching functions to a DOM element that have a closure to a DOM node is capable of causing memory to be lost after navigating away from a page).

**Listing 14.3: An example of a central object store for DOM elements.**

```
(function(){
var cache = {}, guid = 1, expando = "data" + (new Date).getTime();

this.getData = function( elem ) {
  var id = elem[ expando ];

  if ( !id ) {
    id = elem[ expando ] = guid++;
    cache[ id ] = {};
  }

  return cache[ id ];
};

this.removeData = function( elem ) {
  var id = elem[ expando ];

  if ( !id ) {
    return;
  }

  // Remove all stored data
  delete cache[ id ];

  // Remove the expando property from the DOM node
  try {
    delete elem[ expando ];
  } catch( e ) {
    if ( elem.removeAttribute ) {
      elem.removeAttribute( expando );
    }
  }
};
})();
```

The two generic functions, `getData` and `setData`, in Listing 14.3 are actually quite useful beyond the scope of managing event handlers. Using these functions you can attach all sorts of data to an object and have it be stored in a central location.
We can then use this data store to build from and create a new set of `addEvent` and `removeEvent` methods that work closer to our desired outcome in all browsers, as seen in Listing 14.4.

**Listing 14.4: Combining the fix element method from Listing 14.2 and the data store from Listing 14.3 we can now build a new way to add and remove events.**

```
<script>
(function(){
var guid = 1;

this.addEvent = function( elem, type, fn ) {
  var data = getData( elem ), handlers;

  // We only need to generate one handler per element
  if ( !data.handler ) {
    // Our new meta-handler that fixes
    // the event object and the context
    data.handler = function( event ) {
      event = fixEvent( event );

      var handlers = getData( elem ).events[ event.type ];

      // Go through and call all the real bound handlers
      for ( var i = 0, l = handlers.length; i < l; i++ ) {
        handlers[i].call( elem, event );

        // Stop executing handlers since the user requested it
        if ( event.isImmediatePropagationStopped() ) {
          break;
        }
      }
    };
  }

  // We need a place to store all our event data
  if ( !data.events ) {
    data.events = {};
  }

  // And a place to store the handlers for this event type
  handlers = data.events[ type ];

  if ( !handlers ) {
    handlers = data.events[ type ] = [];

    // Attach our meta-handler to the element,
    // since one doesn't exist
    if ( document.addEventListener ) {
      elem.addEventListener( type, data.handler, false );

    } else if ( document.attachEvent ) {
      elem.attachEvent( "on" + type, data.handler );
    }
  }

  if ( !fn.guid ) {
    fn.guid = guid++;
  }

  handlers.push( fn );
};

this.removeEvent = function( elem, type, fn ) {
  var data = getData( elem ), handlers;

  // If no events exist, nothing to unbind
  if ( !data.events ) {
    return
  }

  // Are we removing all bound events?
  if ( !type ) {
    for ( type in data.events ) {
      cleanUpEvents( elem, type );
    }

    return;
  }

  // And a place to store the handlers for this event type
  handlers = data.events[ type ];

  // If no handlers exist, nothing to unbind
  if ( !handlers ) {
    return;
  }

  // See if we're only removing a single handler
  if ( fn && fn.guid ) {
    for ( var i = 0; i < handlers.length; i++ ) {
      // We found a match
      // (don't stop here, there could be a couple bound)
      if ( handlers[i].guid === fn.guid ) {
        // Remove the handler from the array of handlers
        handlers.splice( i--, 1 );
      }
    }
  }

  cleanUpEvents( elem, type );
};

// A simple method for changing the context of a function
```

```
this.proxy = function( context, fn ) {
  // Make sure the function has a unique ID
  if ( !fn.guid ) {
    fn.guid = guid++;
  }

  // Create the new function that changes the context
  var ret = function() {
    return fn.apply( context, arguments );
  };

  // Give the new function the same ID
  // (so that they are equivalent and can be easily removed)
  ret.guid = fn.guid;

  return ret;
};

function cleanUpEvents( elem, type ) {
  var data = getData( elem );

  // Remove the events of a particular type if there are none left
  if ( data.events[ type ].length === 0 ) {
    delete data.events[ type ];

    // Remove the meta-handler from the element
    if ( document.removeEventListener ) {
      elem.removeEventListener( type, data.handler, false );

    } else if ( document.detachEvent ) {
      elem.detachEvent( "on" + type, data.handler );
    }
  }

  // Remove the events object if there are no types left
  if ( isEmpty( data.events ) ) {
    delete data.events;
    delete data.handler;
  }

  // Finally remove the expando if there is no data left
  if ( isEmpty( data ) ) {
    removeData( elem );
  }
}

function isEmpty( object ) {
  for ( var prop in object ) {
    return false;
  }

  return true;
}
})();

addEvent( window, "load", function() {
    var li = document.getElementsByTagName("li");
    for ( var i = 0; i < li.length; i++ ) (function(elem){
      addEvent( elem, "click", function handler(e) {
    // Only do this on left click
    if ( e.button === 1 ) {
          this.style.backgroundColor = "green";
          removeEvent( elem, "click", handler );
    }
      });
    })(li[i]);
  });
  </script>
  <ul>
  <li>Click</li>
  <li>me</li>
  <li>once.</li>
  </ul>
```

There is a lot of code in Listing 14.4 but none of it is particularly challenging. Let's go through the logic function by function.

### 14.2.2 addEvent

We're now storing all the incoming handlers in the central data store (the event types are within the "events" property and each event type stores an array of all the handlers that are bound) instead of binding the handler directly to the element. We use an array because it allows us to bind multiple copies of the same handler and ensure that the handlers will be executed in the same order every time they're run. If we wish to implement a cross-browser version of the W3C DOM method `stopImmediatePropagation` we can easily do that with all these handlers stored in this way.

Since we still have to bind something to the element in order to listen for the event to occur we instead bind a single, generic, handler. We only need to generate one of these handlers per element (since we capture the element via a closure which then use to look up the event data and correct the handler contexts).

While the handler comes in we also give it a unique ID. This isn't explicitly required in order to be able to remove the handler later (we could always just compare the handler directly with what's stored in the data store) but it does give us the ability to create a nice proxy function for changing the context of the handler. The proxy function is similar to the context changing functions that we looked at in previous chapters but with an important distinction: You can bind a handler run through the proxy handler but still unbind the original handler. This can be best seen in Listing 14.5.

**Listing 14.5: How to use the proxy function to count the number of clicks.**

```
  <script>
  var obj = {
    count: 0,
    method: function(){
      this.count++;
    }
  };

  addEvent( window, "load", function() {
    // Attach the handler while enforcing the context to 'obj'
    addEvent( document.body, "click", proxy( obj, obj.method ) );

    // Remove on right click
    addEvent( document.body, "click", function(e) {
      if ( e.button === 3 ) {
        // Note that we can remove the obj.method handler no problem
        removeEvent( document.body, "click", obj.method );
      }
    });
  });
  </script>
```

### 14.2.3 removeEvent

The `removeEvent` function is much more advanced than the simple one that we built previously. Not only is it capable of removing a single event handler (even one bound using the proxy function) but it can also remove all handlers of a particular type or even all events bound to the element.

```
// Remove all bound events from the body
removeEvent( document.body );

// Remove all click events from the body
removeEvent( document.body, "click" );

// Remove a single function from the body
removeEvent( document.body, "click", someFn );
```

The important aspect behind `removeEvent` is in making sure that we clean up all our event handlers, data store, and meta-handler properly. Most of the logic for handling that is contained within the `cleanUpEvents` function. This function makes sure to systematically go through the element's data store and clean up any loose ends - eventually removing the meta-handler from the element itself if all handlers of a particular type have been removed from the element.

## 14.3 Triggering Events

Normally events occur when a user action triggers them (such as clicking or moving the mouse). However there are many cases where it's appropriate to simulate a native event occurring (even more so when working with custom events).

In a triggering function there are a few things that we want to have happen. We want to trigger the bound handlers on the element that we target, we want the event to bubble up the tree triggering any other bound handlers, and finally we want the native event to be triggered on the target element (where available). A function that handles all of these cases can be seen in Listing 14.6.

**Listing 14.6: Trigger a bubbling event on an element.**

```
function triggerEvent( elem, event ) {
  var handler = getData( elem ).handler,
    parent = elem.parentNode || elem.ownerDocument;

  if ( typeof event === "string" ) {
    event = { type: event, target: elem };
  }

  if ( handler ) {
    handler.call( elem, event );
  }

  // Bubble the event up the tree to the document,
  // Unless it's been explicitly stopped
  if ( parent && !event.isPropagationStopped() ) {
    triggerEvent( parent, event );

  // We're at the top document so trigger the default action
  } else if ( !parent && !event.isDefaultPrevented() ) {
    var targetData = getData( event.target ),
      targetHandler = targetData.handler;

    if ( event.target[ type ] ) {
      // Temporarily disable the bound handler,
      // don't want to execute it twice
      if ( targetHandler ) {
        targetData.handler = function(){};
      }

      // Trigger the native event (click, focus, blur)
      event.target[ type ]();

      // Restore the handler
      if ( targetHandler ) {
        targetData.handler = targetHandler;
      }
    }
  }
}
```

The `triggerEvent` function in Listing 14.6 takes two parameters: The element on which the event will be triggered and the event that is being executed (which can be either an event object or just an event type, from which a simple event object is generated).

To trigger the event all we have to do is walk from the initial starting DOM node, find the meta-handler bound to it (if there is one), execute it and pass in the event object. All the rest of the handler execution is taken care of by the code that was originally in `addEvent` and `fixEvent`. We continue to walk up the tree by looking for new `parentNode` properties until we reach the top of the tree and move to the document. Once the document execution has completed we can now execute the default browser action.

Note that during the event bubbling up the tree we make sure that propagation hasn't been stopped. Since we're doing our own triggering and bubbling it's important that we also simulate those actions. Additionally we make sure that we don't execute the default browser action if that has also been prevented.

To trigger the default browser action we use the appropriate method on the original target element. For example if we triggered a focus event we check to see if the original target element has a `.focus()` method and if so, execute it. We need to make sure that when that happens we don't execute the handlers again (we already did that once the first time through - all we care about this time is the execute of the default browser action).

Probably the best part of all of this code though is that it implicitly allows custom events to just work.

### 14.3.1 Custom Events

With all the work that's gone in to integrating cross-browser events, thus far, supporting custom events ends up being relatively simple. Custom events are a way of simulating the experience (to the end user) of a real event without having to use the browser's underlying event structure. Custom events can be an effective means of indirectly communicating actions to elements in a one-to-many way.

With the code that we've written for `addEvent`, `removeEvent`, and `triggerEvent` - nothing has to change in order to support custom events. Functionally there is no difference between an event that does exist and will be fired by the browser and an event that doesn't exist and will only fire when triggered manually. You can see an example of triggering a custom event in Listing 14.7.

**Listing 14.7: Triggering a custom event across a number of elements.**

```
<script>
addEvent( window, "load", function() {
  var li = document.getElementsByTagName("li");

  for ( var i = 0; i < li.length; i++ ) {
    addEvent( li[i], "update", function() {
      this.innerHTML = parseInt(this.innerHTML) + 1;
    });
  }

  var input = document.getElementsByTagName("input")[0];

  addEvent( input, "click", function() {
    var li = document.getElementsByTagName("li");

    for ( var i = 0; i < li.length; i++ ) {
      triggerEvent( li[i], "update" );
    }
  });
});
</script>
<input type="button" value="Add 1"/>
<ul>
  <li>1</li>
  <li>2</li>
  <li>3</li>
</ul>
```

The important aspect behind custom events, in contrast with just using regular functions, is that you're associating the functionality directly with the elements themselves and in a one-to-many fashion. In Listing 14.7 there is one update handler bound to each list item but that doesn't have to be the case, there could be any number of update handlers bound to the item and it wouldn't make any difference, from an execution and triggering perspective. This one-to-many relationship is the fundamental advantage behind using custom events in your code and should allow you to develop applications in a much more expressive and flexible manner.

## 14.4 Bubbling and Delegation

Event delegation is one of the best techniques available for developing high performance, scalable, web applications. Traditionally one would bind event handlers directly to the elements that they wish to watch - with event delegation you bind to an ancestor element (such as a table or the entire document) and watch for all events that come in. In this way you only have to bind one event handler for a large number of elements. This technique makes good use of the event bubbling provided by the browser. Since the events will bubble up the tree we can simply bind to that once ancestor element and wait for the events to come in.

Since event bubbling is the only technique available across all browsers (event capturing only works in W3C compatible browsers) it's important that all events consistently bubble or work in a way that the user expects. Unfortunately the `submit`, `change`, `focus`, and `blur` events all have serious problems with their bubbling implementations, in various browsers, and must be worked around.

To start, the `submit` and `change` events don't bubble at all in Internet Explorer (unlike in the W3C DOM-capable browsers where they bubble consistently). Presumably, however, they would implement support for those events at some point so we need to use a technique that is capable of gracefully determining if an event is capable of bubbling up to a parent element.

One such piece of detection code was written by Juriy Zaytsev and can be seen in Listing 14.8.

- Event detection code: http://perfectionkills.com/detecting-event-support-without-browser-sniffing/

**Listing 14.8: Event bubbling detection code originally written by Juriy Zaytsev.**

```
function isEventSupported(eventName) {
  var el = document.createElement('div'), isSupported;

  eventName = 'on' + eventName;
  isSupported = (eventName in el);

  if ( !isSupported ) {
    el.setAttribute(eventName, 'return;');
    isSupported = typeof el[eventName] == 'function';
  }

  el = null;
  return isSupported;
}
```

The event detection technique works by checking to see if an existing ontype (where type is the name of the event) property exists on a div. We check a div explicitly because divs typically have more events bubbled up to them (such as `change` and `submit`). If the ontype property doesn't exist then we create the ontype attribute, putting in a bit of code, and check to see if the element knows how to translate that into a function. If it does then it's a pretty good indicator that it knows how to interpret that particular event on bubble.

We can now use this detection code as the basis for further implementing properly-working events across all browsers.

### 14.4.1 submit

The `submit` event is one of the few that doesn't bubble in Internet Explorer. Thankfully though it is one of the easiest events to simulate. A `submit` event can be triggered in a one of two ways:

- Clicking (or focusing on and hitting a trigger key, like enter or spacebar) a submit or image submit button.
- Pressing enter while inside a text or password input.

Knowing these two cases we can bind to the two appropriate events (click and keypress - both of which bubble normally) to achieve our desired result, as seen in Listing 14.9.

**Listing 14.9: An implementation of a cross-browser, bubbling, `submit` event.**

```
(function(){
  // Check to see if the submit event works as we expect it to
  if ( !isEventSupported("submit") ) {
    this.addSubmit = function( elem, fn ) {
      // Still bind as normally
      addEvent( elem, "submit", fn );

      // But we need to add extra handlers if we're not on a form
    // Only add the handlers for the first handler bound
    if ( elem.nodeName.toLowerCase() !== "form" &&
         getData( elem ).events.submit.length === 1 ) {

      addEvent( elem, "click", submitClick );
      addEvent( elem, "keypress", submitKeypress );
    }
  };

  this.removeSubmit = function( elem, fn ) {
    removeEvent( elem, "submit", fn );

    var data = getData( elem );

    // Only remove the handlers when there's nothing left to remove
    if ( elem.nodeName.toLowerCase() !== "form" &&
         !data || !data.events || !data.events.submit ) {

      addEvent( elem, "click", submitClick );
      addEvent( elem, "keypress", submitKeypress );
    }
  };

// Otherwise the event works perfectly fine
// so we just behave normally
} else {
  this.addSubmit = function( elem, fn ) {
    addEvent( elem, "submit", fn );
  };

  this.removeSubmit = function( elem, fn ) {
    removeEvent( elem, "submit", fn );
  };
}

// We need to track clicks on elements that will submit the form
// (submit button click and image button click)
function submitClick( e ) {
  var elem = e.target, type = elem.type;

  if ( (type === "submit" || type === "image") &&
       isInForm( elem ) ) {
    return triggerEvent( this, "submit" );
  }
}

// Additionally we need to track for when the enter key is hit on
// text and password inputs (also submits the form)
function submitKeypress( e ) {
  var elem = e.target, type = elem.type;

  if ( (type === "text" || type === "password") &&
       isInForm( elem ) && e.keyCode === 13 ) {
    return triggerEvent( this, "submit" );
  }
}
```

```
    // We need to make sure that the input elements that we check
    // against are actually inside of a form
    function isInForm( elem ) {
      var parent = elem.parentNode;

      while ( parent ) {
        if ( parent.nodeName.toLowerCase() === "form" ) {
          return true;
        }

        parent = parent.parentNode;
      }

      return false;
    }
  })();
```

In Listing 14.9 we can see that we don't need to bind a `submit` event to a form element as that'll continue to work as we would expect it to. Instead we should only bind our special handlers in the case where we're not binding on a form, likely farther up the document tree. Note that we continue to bind to the submit event even though it won't have any immediate effect - we use this as a means to easily store our handlers for later triggering.

Inside the `submitClick` function we verify that the user is in fact clicking on a submit or image input (both of which can submit the form). Additionally we verify that those inputs are actually inside a form (it's theoretically possible that someone might have these inputs outside a form, in which case there obviously isn't a form submission occurring).

Inside `submitKeypress` we first verify that we're working against either a text or password input and that the input is inside a form (like with the click verification). We have the additional check to make sure that the keyCode is equal to 13 (the enter key), which would trigger a `submit` event to occur.

Note that in the `addSubmit` and `removeSubmit` functions we're careful to only bind these additional handlers once and only remove them after all of the submit events have been unbound from the element. Binding them more than once would cause too many events to fire and unbinding them too early would break future submits.

All of this logic though is rather quite generic and tends to apply well to fixing other DOM bubbling events, such as the `change` event.

### 14.4.2 change

The `change` event is another event that doesn't bubble properly in Internet Explorer. Unfortunately it's significantly harder to implement properly, compared to the `submit` event. In order to implement the bubbling change event we must bind to a number of different events.

- The `focusout` event for checking the value after moving away from the form element.

- The `click` and `keydown` event for checking the value the instant it's changed.
- The `beforeactivate` for getting the previous value before a new one is set.

Listing 14.10 has a full implementation of the change that uses all of the above events to create the final result in all browsers.

**Listing 14.10: An implementation of a cross-browser bubbling change event.**

```
(function(){
  // We want to simulate change events on these elements
  var formElems = /textarea|input|select/i;

  // Check to see if the submit event works as we expect it to
  if ( !isEventSupported("change") ) {
    this.addChange = function( elem, fn ) {
      addEvent( elem, "change", fn );

      // Only add the handlers for the first handler bound
      if ( getData( elem ).events.change.length === 1 ) {
        addEvent( elem, "focusout", testChange );
        addEvent( elem, "click", changeClick );
        addEvent( elem, "keydown", changeKeydown );
        addEvent( elem, "beforceactivate", changeBefore );
      }
    };

    this.removeChange = function( elem, fn ) {
      removeEvent( elem, "change", fn );

      var data = getData( elem );

      // Only remove the handlers when there's
      // nothing left to remove
      if ( !data || !data.events || !data.events.submit ) {
        addEvent( elem, "focusout", testChange );
        addEvent( elem, "click", changeClick );
        addEvent( elem, "keydown", changeKeydown );
        addEvent( elem, "beforceactivate", changeBefore );
      }
    };

  // The change event works just fine
  } else {
    this.addChange = function( elem, fn ) {
      addEvent( elem, "change", fn );
    };

    this.removeChange = function( elem, fn ) {
      removeEvent( elem, "change", fn );
    };
  }
```

```
// Check to see that the clickable inputs are updated
function changeClick( e ) {
  var elem = e.target, type = elem.type;

  if ( type === "radio" || type === "checkbox" ||
       elem.nodeName.toLowerCase() === "select" ) {
    return testChange.call( this, e );
  }
}

// Change has to be called before submit
// Keydown will be called before keypress,
// which is used in submit-event delegation
function changeKeydown( e ) {
  var elem = e.target, type = elem.type, key = e.keyCode;

  if ( key === 13 && elem.nodeName.toLowerCase() !== "textarea" ||
       key === 32 && (type === "checkbox" || type === "radio") ||
       type === "select-multiple" ) {
    return testChange.call( this, e );
  }
}

// Beforeactivate happens before the previous element is blurred
// Use it to store information for later checking
function changeBefore( e ) {
  var elem = e.target;
  getData( elem )._change_data = getVal( elem );
}

// Get a string value back for a form element
// that we can use to verify if a change has occurred
function getVal( elem ) {
  var type = elem.type, val = elem.value;

  // Checkboxes and radios only change the checked state
  if ( type === "radio" || type === "checkbox" ) {
    val = elem.checked;

  // Multiple selects need to check all selected options
  } else if ( type === "select-multiple" ) {
    val = "";

    if ( elem.selectedIndex > -1 ) {
      for ( var i = 0; i < elem.options.length; i++ ) {
        val += "-" + elem.options[i].selected;
      }
    }

  // Regular selects only need to check what
  // option is currently selected
  } else if ( elem.nodeName.toLowerCase() === "select" ) {
    val = elem.selectedIndex;
  }

  return val;
```

```
    }

    // Check to see if a change in the value has occurred
    function testChange( e ) {
      var elem = e.target, data, val;

      // Don't need to check on certain elements and read-only inputs
      if ( !formElems.test( elem.nodeName ) || elem.readOnly ) {
        return;
      }

      // Get the previously-set value
      data = getData( elem )._change_data;
      val = getVal( elem );

      // the current data will be also retrieved by beforeactivate
      if ( e.type !== "focusout" || elem.type !== "radio" ) {
        getData( elem )._change_data = val;
      }

      // If there's been no change then we can bail
      if ( data === undefined || val === data ) {
        return;
      }

      // Otherwise the change event should be fired
      if ( data != null || val ) {
        return triggerEvent( elem, "change" );
      }
    }
  })();
```

As you can probably tell by the code in Listing 14.10 we receive virtually no help from Internet Explorer towards having a workable `change` solution. Instead we must implement all aspects of the change events: Serializing the old values, comparing the old values against the new values, and watching for potential change triggers to occur (`click`, `keydown`, and `focusout`).

The core of the functionality is encapsulated within the `getVal` and `testChange` functions. `getVal` is designed to return a simple serialized version of the state of a particular form element. This value will be stored in the `_change_data` property within the element's data object for later use. The `testChange` function is primarily responsible for determining if an actual change has occurred between the previously-stored value and the newly-set value and triggering a real change event if the value is now different.

In addition to checking to see if a change has occurred after a `focusout` (blur) we also check to see if the enter key was hit on something that wasn't a textarea or hitting the spacebar on a checkbox or radio button. We also check to see if a click occurred on a checkbox, radio, or select as that will also trigger a change to occur.

All told there's a lot of code for something that should've been tackled natively by the browser. It'll be greatly appreciated when the day comes that this code doesn't need to exist.

### FOCUSIN AND FOCUSOUT

focusin and focusout are two custom events introduced by Internet Explorer that detect when a focus or blur has occurred on any element, or element descendant. It actually occurs before the focus or blur take place (making it equivalent to a capturing event rather than a bubbling event).

The reason for even considering these two events is that the focus and blur events do not bubble (as dictated by the W3C DOM recommendation and as implemented by all browsers). It ends up being far easier to implement focusin and focusout clones across all browsers then trying to circumvent the intentions of the browsers (whatever that may be) and getting the events to bubble.

The best way to implement the focusin and focusout events would be to modify the existing addEvent function to handle the event types inline.

```
// Attach our meta-handler to the element, since once doesn't exist
if ( document.addEventListener ) {
  elem.addEventListener(
    type === "focusin" ? "focus" :
      type === "focusout" ? "blur" : type,
    data.handler, type === "focusin" || type === "focusout" );

} else if ( document.attachEvent ) {
  elem.attachEvent( "on" + type, data.handler );
}
```

and then modify the removeEvent function to unbind the events again properly:

```
// Remove the meta-handler from the element
if ( document.removeEventListener ) {
  elem.removeEventListener(
    type === "focusin" ? "focus" :
      type === "focusout" ? "blur" : type,
    data.handler, type === "focusin" || type === "focusout" );

} else if ( document.detachEvent ) {
  elem.detachEvent( "on" + type, data.handler );
}
```

The end will result will be native support for the focusin and focusout event in all browsers. Naturally you might want to keep your event-specific logic separate from your addEvent and removeEvent internals, in that case you could implement some form of extensibility for overriding the native binding/unbinding mechanisms provided by the browser, for specific event types.

Some more information about cross-browser focus and blur events can be found here:

- http://www.quirksmode.org/blog/archives/2008/04/delegating_the.html

### 14.4.3 mouseenter and mouseleave

mouseenter and mouseleave are two custom events introduced by Internet Explorer to simplify the process of determining when the mouse is currently positioned within an element, or outside of it. Traditionally one would interact with the mouseover and mouseout events provided by the browser but they don't provide what users are typically looking for: They fire the event when you move between child elements in addition to on the element itself. This is typical of the event bubbling model but it is frequently more than what a user is looking for (they simply want to make sure that they're still within the element and only car about when the element is left).

This is where the `mouseenter` and `mouseleave` events come in handy. They will only fire on the main element on which you bind them and only inform you as to if the mouse is currently inside or outside of the element. Since Internet Explorer is the only browser that currently implments these useful events we need to simulate the full event interaction for other browsers as well.

Listing 14.11 shows some code to implement the `mouseenter` and `mouseleave` events across all browsers.

**Listing 14.11: An example of implementing a hover event using the event detection code from Listing 14.8.**

```
<div>Hover <strong>over</strong> me!</div>
<style>.over { background: yellow; }</style>
<script>
(function(){
  if ( isEventSupported( "mouseenter" ) ) {
    this.hover = function( elem, fn ) {
      addEvent( elem, "mouseenter", function(){
        fn.call( elem, "mouseenter" );
      });

      addEvent( elem, "mouseleave", function(){
        fn.call( elem, "mouseleave" );
      });
    };

  } else {
    this.hover = function( elem, fn ) {
      addEvent( elem, "mouseover", function(e){
        withinElement( this, e, "mouseenter", fn );
      });

      addEvent( elem, "mouseout", function(e){
        withinElement( this, e, "mouseleave", fn );
      });
    };
  }

  function withinElement( elem, event, type, handle ) {
    // Check if mouse(over|out) are still
    // within the same parent element
    var parent = event.relatedTarget;

    // Traverse up the tree
    while ( parent && parent != elem ) {
      // Firefox sometimes assigns relatedTarget a XUL element
      // which we cannot access the parentNode property of
      try {
        parent = parent.parentNode;

      // assuming we've left the element since
      // we most likely mousedover a xul element
      } catch(e) { break; }
    }

    if ( parent != elem ) {
      // handle event if we actually just
      // moused on to a non sub-element
      handle.call( elem, type );
    }
  }
})();

window.onload = function(){
  var div = document.getElementsByTagName("div")[0];

  hover( div, function(type){
    if ( type === "mouseenter" ) {
      this.className = "over";
    } else {
      this.className = "";
    }
  });
};
</script>
```

The majority of the logic for handling the `mouseenter` and `mouseleave` events lies inside of the `withinElement` function. This function checks the `relatedTarget` of the event and makes sure that it's not contained within the original element (if it is contained within then we're just seeing `mouseover` and `mouseout` events from internal elements)

## 14.5 Document Ready

The final important event that needs to be covered is the "ready" event (implemented as `DOMContentLoaded` in W3C DOM-capable browsers). The ready event fires as soon as the entire DOM document has been loaded and is able to be traversed and manipulated. This event has become an integral part of many modern frameworks, allowing code to be layered in an unobtrusive manner, frequently executing before the page is displayed.

An implementation of the ready event, using the above technique, can be found in Listing 14.12.

**Listing 14.12: Implement a cross-browser DOM ready event.**

```
(function(){
  var isReady = false, DOMContentLoaded;

  // Catch cases where addReady is called after the
  // browser event has already occurred.
  if ( document.readyState === "complete" ) {
    return ready();
  }

  // Mozilla, Opera and Webkit currently support this event
  if ( document.addEventListener ) {
    DOMContentLoaded = function() {
      document.removeEventListener(
        "DOMContentLoaded", DOMContentLoaded, false );
      ready();
    };

    // Use the handy event callback
    document.addEventListener(
      "DOMContentLoaded", DOMContentLoaded, false );

  // If IE event model is used
  } else if ( document.attachEvent ) {
    DOMContentLoaded = function() {
      if ( document.readyState === "complete" ) {
        document.detachEvent(
          "onreadystatechange", DOMContentLoaded );
        ready();
      }
    };

    // ensure firing before onload,
    // maybe late but safe also for iframes
    document.attachEvent(
      "onreadystatechange", DOMContentLoaded );

    // If IE and not a frame
    // continually check to see if the document is ready
    var toplevel = false;
    try {
      toplevel = window.frameElement == null;
    } catch(e) {}

    if ( document.documentElement.doScroll && toplevel ) {
      doScrollCheck();
    }
  }

  function ready() {
    if ( !isReady ) {
      triggerEvent( document, "ready" );
      isReady = true;
    }
  }

  // The DOM ready check for Internet Explorer
  function doScrollCheck() {
    if ( isReady ) {
      return;
    }

    try {
      // If IE is used, use the trick by Diego Perini
      // http://javascript.nwbox.com/IEContentLoaded/
      document.documentElement.doScroll("left");
    } catch( error ) {
      setTimeout( doScrollCheck, 1 );
      return;
    }

    // and execute any waiting functions
    ready();
  }
})();
```

The majority of the logic in Listing 14.12 is directly related to figuring out when the DOM is ready in Internet Explorer. Internet Explorer doesn't provide a native means of watching for when the document is finally ready, unlike other browsers. Instead we must resort to a technique originally developed by Diego Perini in which the `doScroll("left")` method is called on the `documentElement`. Calling this method will have effect of scrolling the viewport to the left side (which is the normal starting position anyway) - but more importantly, it will throw an exception if the document isn't ready to be manipulated yet. Thus we can use a timer to constantly loop and try to run the `doScroll` method in a try/catch block. Whenever an exception is no longer thrown then it is safe to continue. More information about this particular technique can be found on Diego's site:

- http://javascript.nwbox.com/IEContentLoaded/

It should be noted that the `doScroll` technique alone doesn't work inside iframe'd documents. As a fallback we use a secondary technique: Watching the `onreadystatechange` event on the document. This particular event is more inconsistent than the `doScroll` technique - while it'll always fire after the DOM is ready it'll sometimes fire quite a while after (but always before the final window load event). That being said it serves as a good backup for IE, making sure that the at least something will fire before the window load event.

One additional check this is performed, on load, is examining the `document.readyState` property. This property, available in all browsers, notes how fully-loaded the DOM document is at that point (note that long delays in loading, especially in Internet Explorer, may cause the `readyState` to report "complete" too early, hence why we use the `doScroll` technique instead). Checking this property on load can help to avoid unnecessary event binding if the DOM is already in a ready-to-use state (as might be the case if the script was dynamically loaded at a later time).

Note: While `document.readyState` is available in all browsers it was only added to Firefox in version 3.6. This may not matter, depending upon your application, but it's good to note.

With a complete ready event implementation we now have all the tools in place for a complete DOM event system.

## 14.6 Summary

As we can see, DOM event systems are anything but simple. Internet Explorer's divergent system ends up causing much of the overhead that we see and have to write. Even so the lack of extensibility in the native API means that we still have to circumvent, or improve upon, most of the event system in order to arrive at a solution that is more universally applicable.

In this chapter we learned the fundamentals behind adding and removing events from a DOM node and the hidden object store that's used to make it all possible. Additionally we looked at triggering events and working with custom events and how applications can benefit from their unique model. Finally we examined event bubbling and delegation and all the quirks that go along with it, working to normalize the bubbling of events across all browsers.

All told we now have the knowledge necessary to implement a complete, and useful, DOM event system that is capable of tackling even the greatest challenge presented to us by the browser.