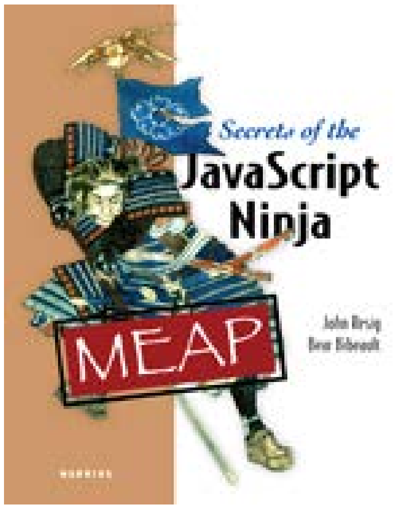
MANNING PUBLICATIONS



Introducing Ninja

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. We'll now take a look at the techniques that are used to create two of the more popular JavaScript libraries.

Our key JavaScript libraries

The techniques and practices used to create two modern JavaScript libraries 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, and so on).
- base2 (http://code.google.com/p/base2), created by Dean Edwards and released in 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 makeup 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.

Understanding the JavaScript Language

Many JavaScript coders, as they advance through their careers, get to the point at which they're actively using the vast array of language elements, 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 generally is 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.

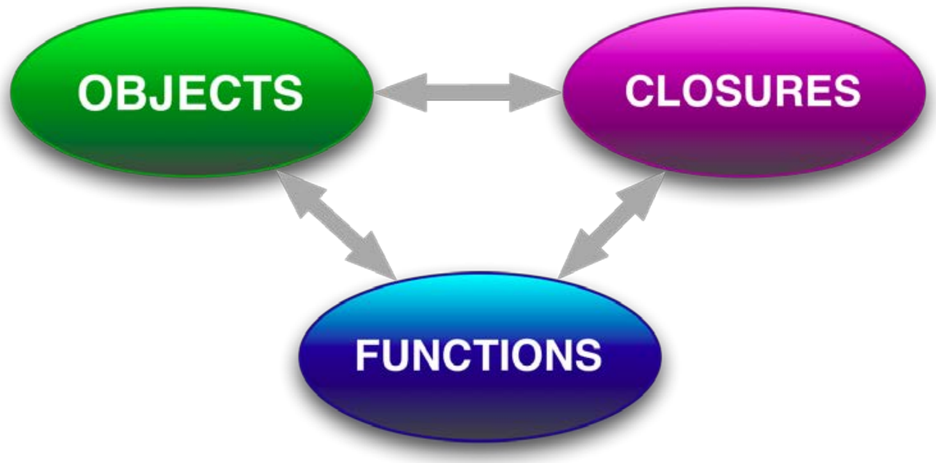


Figure 1 JavaScript consists of a close relationship between objects, functions and closures

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.

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.

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?

An approach that we we'll borrow from Yahoo! is called *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. Any ungraded browser/platform combination, or unlisted browser, is assigned a grade of X.

Table 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 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 the 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 2 shows a sample chart that represents your authors' personal choices when developing for some browsers (not all browsers included for brevity) based on March 2011 market share.

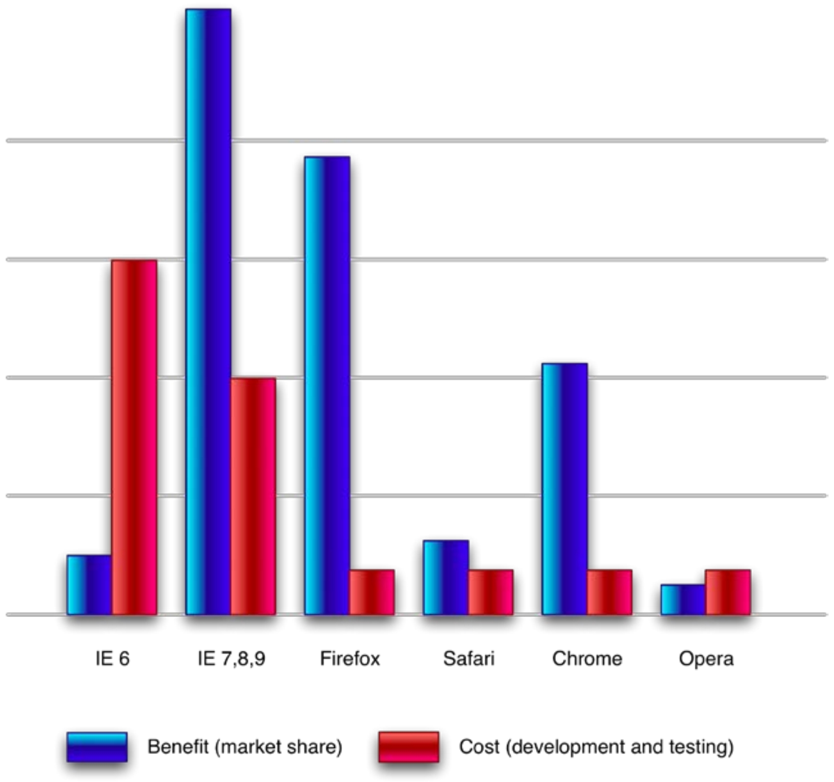


Figure 2 Analyzing the cost versus benefit of supporting various browsers tells us where to put our effort.

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 the benefit is more important than the 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.

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 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.

Best practice: testing

A number of testing techniques serve to ensure that code operates as intended. The primary tool 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).

Best practice: performance analysis

Another important practice is performance analysis. The JavaScript engine in the browsers has been making astounding strides in the performance of JavaScript itself, but that's no excuse for us to write sloppy and inefficient code. The `perf()` function collects performance information.

An example of its use would be:

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

These best-practice techniques 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.

Summary

Cross-browser web application development is hard; harder than most people would think. In order to pull it off, we need not only master the JavaScript language but have 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.

Last updated: November 27, 2011