Part 1 JS

Μερικά από τα καλύτερα άρθρα για Javascript

http://talentincubator.eu

original collection of articles from Azat Mardan

JavaScript Fundamentals

The World's Most Misunderstood Programming Language

Original Article

http://javascript.crockford.com/javascript.html

Douglas Crockford, crockford.com[a]

[a] http://crockford.com

JavaScript[1], aka Mocha, aka LiveScript, aka JScript, aka ECMAScript, is one of the world's most popular programming languages. Virtually every personal computer in the world has at least one JavaScript interpreter installed on it and in active use. JavaScript's popularity is due entirely to its role as the scripting language of the WWW.

Despite its popularity, few know that JavaScript is a very nice dynamic object-oriented general-purpose programming language. How can this be a secret? Why is this language so misunderstood?

The Name

The *Java-* prefix suggests that JavaScript is somehow related to Java, that it is a subset or less capable version of Java. It seems that the name was intentionally selected to create confusion, and from confusion comes misunderstanding. JavaScript is not interpreted Java. Java is interpreted Java. JavaScript is a different language.

JavaScript has a syntactic similarity to Java, much as Java has to C. But it is no more a subset of Java than Java is a subset of C. It is better than Java in the applications that Java (fka Oak) was originally intended for.

JavaScript was not developed at Sun Microsystems, the home of Java. JavaScript was developed at Netscape. It was originally called LiveScript, but that name wasn't confusing enough.

The *-Script* suffix suggests that it is not a real programming language, that a scripting language is less than a programming language. But it is really a matter of specialization. Compared to C, JavaScript trades performance for expressive power and dynamism.

[1] http://javascript.crockford.com/

Lisp in C's Clothing

JavaScript's C-like syntax, including curly braces and the clunky for statement, makes it appear to be an ordinary procedural language. This is misleading because JavaScript has more in common with functional languages like Lisp or Scheme[2] than with C or Java. It has arrays instead of lists and objects instead of property lists. Functions are first class. It has closures. You get lambdas without having to balance all those parens.

Typecasting

JavaScript was designed to run in Netscape Navigator. Its success there led to it becoming standard equipment in virtually all web browsers. This has resulted in typecasting. JavaScript is the George Reeves[3] of programming languages. JavaScript is well suited to a large class of non-Web-related applications

Moving Target

The first versions of JavaScript were quite weak. They lacked exception handling, inner functions, and inheritance. In its present form, it is now a complete object-oriented programming language. But many opinions of the language are based on its immature forms.

The ECMA committee that has stewardship over the language is developing extensions which, while well intentioned, will aggravate one of the language's biggest problems: There are already too many versions. This creates confusion.

Design Errors

No programming language is perfect. JavaScript has its share of design errors, such as the overloading of + to mean both addition and concatenation with type coercion, and the error-prone with statement should be avoided. The reserved word policies are much too strict. Semicolon insertion was a huge mistake, as was the notation for literal regular expressions. These mistakes have led to programming errors, and called the design of the language as a whole into question. Fortunately, many of these problems can be mitigated with a good lint[4] program.

The design of the language on the whole is quite sound. Surprisingly, the ECMAScript committee does not appear to be interested in correcting these problems. Perhaps they are more interested in making new ones.

[2]http://javascript.crockford.com/little.html

[3]http://www.amazon.com/exec/obidos/ASIN/B000KWZ7JC/wrrrldwideweb

[4]http://www.jslint.com/

Lousy Implementations

Some of the earlier implementations of JavaScript were quite buggy. This reflected badly on the language. Compounding that, those implementations were embedded in horribly buggy web browsers.

Bad Books

Nearly all of the books about JavaScript are quite awful. They contain errors, poor examples, and promote bad practices. Important features of the language are often explained poorly, or left out entirely. I have reviewed dozens of JavaScript books, and**I can only recommend one**: *JavaScript: The Definitive Guide (5th Edition)*[5] by David Flanagan. (Attention authors: If you have written a good one, please send me a review copy.)

Substandard Standard

The official specification for the language[6] is published by ECMA[7]. The specification is of extremely poor quality. It is difficult to read and very difficult to understand. This has been a contributor to the Bad Book problem because authors have been unable to use the standard document to improve their own understanding of the language. ECMA and the TC39 committee should be deeply embarrassed.

Amateurs

Most of the people writing in JavaScript are not programmers. They lack the training and discipline to write good programs. JavaScript has so much expressive power that they are able to do useful things in it, anyway. This has given JavaScript a reputation of being strictly for the amateurs, that it is not suitable for professional programming. This is simply not the case.

Object-Oriented

Is JavaScript object-oriented? It has objects which can contain data and methods that act upon that data. Objects can contain other objects. It does not have classes, but it does have constructors which do what classes do, including acting as containers for class variables and methods. It does not have class-oriented inheritance, but it does have prototype-oriented inheritance.

The two main ways of building up object systems are by inheritance (is-a) and by aggregation (has-a). JavaScript does both, but its dynamic nature allows it to excel at aggregation.

Some argue that JavaScript is not truly object oriented because it does not provide information hiding. That is, objects cannot have private variables and private methods: All members are public.

[5] http://www.amazon.com/exec/obidos/ASIN/0596101996/wrrrldwideweb

[6] http://www.ecma-international.org/publications/standards/Ecma-262.htm

[7] http://www.ecma-international.org/

But it turns out that JavaScript objects *can* have private variables and private methods. (Click here now to find out how.)[8] Of course, few understand this because JavaScript is the world's most misunderstood programming language.

Some argue that JavaScript is not truly object oriented because it does not provide inheritance. But it turns out that JavaScript supports not only classical inheritance, but other code reuse patterns as well.[9]

Copyright 2001 Douglas Crockford.[10] All Rights Reserved Wrrrldwide.[11]

Understanding JavaScript Function Invocation and "this"

Original Article

http://yehudakatz.com/2011/08/11/understanding-javascript-function-invocation-and-this

Yehuda Katz, yehudakatz.com

Over the years, I've seen a lot of confusion about JavaScript function invocation. In particular, a lot of people have complained that the semantics of `this` in function invocations is confusing.

In my opinion, a lot of this confusion is cleared up by understanding the core function invocation primitive, and then looking at all other ways of invoking a function as sugar on top of that primitive. In fact, this is exactly how the ECMAScript spec thinks about it. In some areas, this post is a simplification of the spec, but the basic idea is the same.

The Core Primitive

First, let's look at the core function invocation primitive, a Function's `call` method[1]. The call method is relatively straight forward.

1. Make an argument list (`argList`) out of parameters `1` through the end
2. The first parameter is `thisValue`
3. Invoke the function with `this` set to `thisValue` and the `argList` as its argument list

For example:

[8] http://www.crockford.com/javascript/private.html
[9] http://javascript.crockford.com/inheritance.html
[10] mailto:douglas@crockford.com
[11] http://www.crockford.com/

```
function hello(thing) {
  console.log(this + " says hello " + thing);
}

hello.call("Yehuda", "world") //=> Yehuda says hello world
```

As you can see, we invoked the `hello` method with `this` set to `"Yehuda"` and a single argument `"world"`. This is the core primitive of JavaScript function invocation. You can think of all other function calls as desugaring to this primitive. (to “desugar” is to take a convenient syntax and describe it in terms of a more basic core primitive).

[1] In the ES5 spec[12], the `call` method is described in terms of another, more low level primitive, but it’s a very thin wrapper on top of that primitive, so I’m simplifying a bit here. See the end of this post for more information.

Simple Function Invocation

Obviously, invoking functions with `call` all the time would be pretty annoying. JavaScript allows us to invoke functions directly using the parens syntax (`hello("world")`. When we do that, the invocation desugars:

```
function hello(thing) {
  console.log("Hello " + thing);
}

// this:
hello("world")

// desugars to:
hello.call(window, "world");
```

This behavior has changed in ECMAScript 5 **only when using strict mode**[2]:

```
// this:
hello("world")

// desugars to:
hello.call(undefined, "world");
```

[12]http://es5.github.com/#x15.3.4.4

The short version is: **a function invocation like `fn(...args)` is the same as `fn.call(window [ES5-strict: undefined], ...args)`.**

Note that this is also true about functions declared inline: `(function() {})()` is the same as `(function() {}).call(window [ES5-strict: undefined)`.

[2] Actually, I lied a bit. The ECMAScript 5 spec says that `undefined` is (almost) always passed, but that the function being called should change its `thisValue` to the global object when not in strict mode. This allows strict mode callers to avoid breaking existing non-strict-mode libraries.

Member Functions

The next very common way to invoke a method is as a member of an object (`person.hello()`). In this case, the invocation desugars:

```
var person = {
  name: "Brendan Eich",
  hello: function(thing) {
    console.log(this + " says hello " + thing);
  }
}

// this:
person.hello("world")

// desugars to this:
person.hello.call(person, "world");
```

Note that it doesn't matter how the `hello` method becomes attached to the object in this form. Remember that we previously defined `hello` as a standalone function. Let's see what happens if we attach is to the object dynamically:

```
function hello(thing) {
  console.log(this + " says hello " + thing);
}

person = { name: "Brendan Eich" }
person.hello = hello;

person.hello("world") // still desugars to person.hello.call(person, "world")

hello("world") // "[object DOMWindow]world"
```

Notice that the function doesn't have a persistent notion of its 'this'. It is always set at call time based upon the way it was invoked by its caller.

Using `Function.prototype.bind`

Because it can sometimes be convenient to have a reference to a function with a persistent `this` value, people have historically used a simple closure trick to convert a function into one with an unchanging `this`:

```
var person = {
  name: "Brendan Eich",
  hello: function(thing) {
    console.log(this.name + " says hello " + thing);
  }
}

var boundHello = function(thing) { return person.hello.call(person, thing); }

boundHello("world");
```

Even though our `boundHello` call still desugars to `boundHello.call(window, "world")`, we turn right around and use our primitive `call` method to change the `this` value back to what we want it to be.

We can make this trick general-purpose with a few tweaks:

```
var bind = function(func, thisValue) {
  return function() {
    return func.apply(thisValue, arguments);
  }
}

var boundHello = bind(person.hello, person);
boundHello("world") // "Brendan Eich says hello world"
```

In order to understand this, you just need two more pieces of information. First, `arguments` is an Array-like object that represents all of the arguments passed into a function. Second, the `apply` method works exactly like the `call` primitive, except that it takes an Array-like object instead of listing the arguments out one at a time.

Our `bind` method simply returns a new function. When it is invoked, our new function simply invokes the original function that was passed in, setting the original value as `this`. It also passes through the arguments.

Because this was a somewhat common idiom, ES5 introduced a new method `bind` on all `Function` objects that implements this behavior:

```
var boundHello = person.hello.bind(person);
boundHello("world") // "Brendan Eich says hello world"
```

This is most useful when you need a raw function to pass as a callback:

```
var person = {
  name: "Alex Russell",
  hello: function() { console.log(this.name + " says hello world"); }
}

$("#some-div").click(person.hello.bind(person));

// when the div is clicked, "Alex Russell says hello world" is printed
```

This is, of course, somewhat clunky, and TC39 (the committee that works on the next version(s) of ECMAScript) continues to work on a more elegant, still-backwards-compatible solution.

On jQuery

Because jQuery makes such heavy use of anonymous callback functions, it uses the `call` method internally to set the `this` value of those callbacks to a more useful value. For instance, instead of receiving `window` as `this` in all event handlers (as you would without special intervention), jQuery invokes `call` on the callback with the element that set up the event handler as its first parameter.

This is extremely useful, because the default value of `this` in anonymous callbacks is not particularly useful, but it can give beginners to JavaScript the impression that `this` is, **in general** a strange, often mutated concept that is hard to reason about.

If you understand the basic rules for converting a sugary function call into a desugared `func.call(thisValue, ...args)`, you should be able to navigate the not so treacherous waters of the JavaScript `this` value.

PS: I Cheated

In several places, I simplified the reality a bit from the exact wording of the specification. Probably the most important cheat is the way I called `func.call` a "primitive". In reality, the spec has a primitive (internally referred to as `[[Call]]`) that both `func.call` and `[obj.]func()` use.

However, take a look at the definition of `func.call`:

1. If IsCallable(func) is false, then throw a TypeError exception.
2. Let argList be an empty List.
3. If this method was called with more than one argument then in left to right order starting with arg1 append each argument as the last element of argList

4. Return the result of calling the [[Call]] internal method of func, providing thisArg as the this value and argList as the list of arguments.

As you can see, this definition is essentially a very simple JavaScript language binding to the primitive `[[Call]]` operation.

If you look at the definition of invoking a function, the first seven steps set up `thisValue` and `argList`, and the last step is: "Return the result of calling the [[Call]] internal method on func, providing thisValue as the this value and providing the list argList as the argument values."

It's essentially identical wording, once the `argList` and `thisValue` have been determined.

I cheated a bit in calling `call` a primitive, but the meaning is essentially the same as had I pulled out the spec at the beginning of this article and quoted chapter and verse.

Code Conventions for the JavaScript Programming Language

Original Article

http://javascript.crockford.com/code.html

Douglas Crockford, crockford.com

This is a set of coding conventions and rules for use in JavaScript programming. It is inspired by the Sun[13] document Code Conventions for the Java Programming Language[14]. It is heavily modified of course because JavaScript is not Java[15].

The long-term value of software to an organization is in direct proportion to the quality of the codebase. Over its lifetime, a program will be handled by many pairs of hands and eyes. If a program is able to clearly communicate its structure and characteristics, it is less likely that it will break when modified in the never-too-distant future.

Code conventions can help in reducing the brittleness of programs.

All of our JavaScript code is sent directly to the public. It should always be of publication quality.

Neatness counts.

[13]http://www.sun.com/

[14]http://java.sun.com/docs/codeconv/

[15]http://javascript.crockford.com/javascript.html

JavaScript Files

JavaScript programs should be stored in and delivered as `.js` files.

JavaScript code should not be embedded in HTML files unless the code is specific to a single session. Code in HTML adds significantly to pageweight with no opportunity for mitigation by caching and compression.

`<script src=`filename`.js>` tags should be placed as late in the body as possible. This reduces the effects of delays imposed by script loading on other page components. There is no need to use the `language` or `type` attributes. It is the server, not the script tag, that determines the MIME type.

Indentation

The unit of indentation is four spaces. Use of tabs should be avoided because (as of this writing in the 21st Century) there still is not a standard for the placement of tabstops. The use of spaces can produce a larger filesize, but the size is not significant over local networks, and the difference is eliminated by minification[16].

Line Length

Avoid lines longer than 80 characters. When a statement will not fit on a single line, it may be necessary to break it. Place the break after an operator, ideally after a comma. A break after an operator decreases the likelihood that a copy-paste error will be masked by semicolon insertion. The next line should be indented 8 spaces.

Comments

Be generous with comments. It is useful to leave information that will be read at a later time by people (possibly yourself) who will need to understand what you have done. The comments should be well-written and clear, just like the code they are annotating. An occasional nugget of humor might be appreciated. Frustrations and resentments will not.

It is important that comments be kept up-to-date. Erroneous comments can make programs even harder to read and understand.

Make comments meaningful. Focus on what is not immediately visible. Don't waste the reader's time with stuff like

```
    i = 0; // Set i to zero.
```

Generally use line comments. Save block comments for formal documentation and for commenting out.

[16]http://yuiblog.com/blog/2006/03/06/minification-v-obfuscation/

Variable Declarations

All variables should be declared before used. JavaScript does not require this, but doing so makes the program easier to read and makes it easier to detect undeclared variables that may become implied globals[17]. Implied global variables should never be used.

The `var` statements should be the first statements in the function body.

It is preferred that each variable be given its own line and comment. They should be listed in alphabetical order.

```
var currentEntry; // currently selected table entry
var level;        // indentation level
var size;         // size of table
```

JavaScript does not have block scope, so defining variables in blocks can confuse programmers who are experienced with other C family languages. Define all variables at the top of the function.

Use of global variables should be minimized. Implied global variables should never be used.

Function Declarations

All functions should be declared before they are used. Inner functions should follow the `var` statement. This helps make it clear what variables are included in its scope.

There should be no space between the name of a function and the `(` (left parenthesis) of its parameter list. There should be one space between the `)` (right parenthesis) and the `{` (left curly brace) that begins the statement body. The body itself is indented four spaces. The `}` (right curly brace) is aligned with the line containing the beginning of the declaration of the function.

```
function outer(c, d) {
    var e = c * d;

    function inner(a, b) {
        return (e * a) + b;
    }

    return inner(0, 1);
}
```

This convention works well with JavaScript because in JavaScript, functions and object literals can be placed anywhere that an expression is allowed. It provides the best readability with inline functions and complex structures.

[17]http://yuiblog.com/blog/2006/06/01/global-domination/

```
    function getElementsByClassName(className) {
        var results = [];
        walkTheDOM(document.body, function (node) {
            var a;                  // array of class names
            var c = node.className; // the node's classname
            var i;                  // loop counter

// If the node has a class name, then split it into a list of simple names.
// If any of them match the requested name, then append the node to the set of r\
esults.

            if (c) {
                a = c.split(' ');
                for (i = 0; i < a.length; i += 1) {
                    if (a[i] === className) {
                        results.push(node);
                        break;
                    }
                }
            }
        });
        return results;
    }
```

If a function literal is anonymous, there should be one space between the word `function` and the `(` (left parenthesis). If the space is omited, then it can appear that the function's name is `function`, which is an incorrect reading.

```
    div.onclick = function (e) {
        return false;
    };

    that = {
        method: function () {
            return this.datum;
        },
        datum: 0
    };
```

Use of global functions should be minimized.

When a function is to be invoked immediately, the entire invocation expression should be wrapped in parens so that it is clear that the value being produced is the result of the function and not the function itself.

```
var collection = (function () {
    var keys = [], values = [];

    return {
        get: function (key) {
            var at = keys.indexOf(key);
            if (at >= 0) {
                return values[at];
            }
        },
        set: function (key, value) {
            var at = keys.indexOf(key);
            if (at < 0) {
                at = keys.length;
            }
            keys[at] = key;
            values[at] = value;
        },
        remove: function (key) {
            var at = keys.indexOf(key);
            if (at >= 0) {
                keys.splice(at, 1);
                values.splice(at, 1);
            }
        }
    };
}());
```

Names

Names should be formed from the 26 upper and lower case letters (`A` .. `Z`, `a` .. `z`), the 10 digits (`0` .. `9`), and _ (underbar). Avoid use of international characters because they may not read well or be understood everywhere. Do not use `$` (dollar sign) or `\` (backslash) in names.

Do not use _ (underbar) as the first character of a name. It is sometimes used to indicate privacy, but it does not actually provide privacy. If privacy is important, use the forms that provide private members[18]. Avoid conventions that demonstrate a lack of competence.

Most variables and functions should start with a lower case letter.

Constructor functions which must be used with the `[new](http://yuiblog.com/blog/2006/11/13/javascript-we-`
should start with a capital letter. JavaScript issues neither a compile-time warning nor a run-time

[18]http://javascript.crockford.com/private.html
[19]http://yuiblog.com/blog/2006/11/13/javascript-we-hardly-new-ya/

warning if a required `new` is omitted. Bad things can happen if `new` is not used, so the capitalization convention is the only defense we have.

Global variables should be in all caps. (JavaScript does not have macros or constants, so there isn't much point in using all caps to signify features that JavaScript doesn't have.)

Statements

Simple Statements

Each line should contain at most one statement. Put a `;` (semicolon) at the end of every simple statement. Note that an assignment statement which is assigning a function literal or object literal is still an assignment statement and must end with a semicolon.

JavaScript allows any expression to be used as a statement. This can mask some errors, particularly in the presence of semicolon insertion. The only expressions that should be used as statements are assignments and invocations.

Compound Statements

Compound statements are statements that contain lists of statements enclosed in `{ }` (curly braces).

- The enclosed statements should be indented four more spaces.
- The `{` (left curly brace) should be at the end of the line that begins the compound statement.
- The `}` (right curly brace) should begin a line and be indented to align with the beginning of the line containing the matching `{` (left curly brace).
- Braces should be used around all statements, even single statements, when they are part of a control structure, such as an `if` or `for` statement. This makes it easier to add statements without accidentally introducing bugs.

Labels

Statement labels are optional. Only these statements should be labeled: `while`, `do`, `for`, `switch`.

`return` Statement

A `return` statement with a value should not use `( )` (parentheses) around the value. The return value expression must start on the same line as the `return` keyword in order to avoid semicolon insertion.

`if` Statement

The `if` class of statements should have the following form:

`if` (condition) `{` statements `}`

if (condition) `{` statements `}` `else` `{` statements `}`

if (condition) `{` statements `}` `else if` (condition) `{` statements `}` `else` `{` statements `}`

`for` Statement

A `for` class of statements should have the following form:

`for` (initialization`;` condition`;` update) `{` statements `}`

for (variable `in` object) `{` `if` (filter) `{` statements `}` `}`

The first form should be used with arrays and with loops of a predeterminable number of iterations.

The second form should be used with objects. Be aware that members that are added to the prototype of the object ‘`will be included in the enumeration. It is wise to program defensively by using the` hasOwnProperty‘ method to distinguish the true members of the object:

`for` (variable `in` object) `{` `if` (object`.hasOwnProperty(`variable`))` `{` statements `}` `}`

`while` Statement

A `while` statement should have the following form:

`while` (condition) `{` statements `}`

`do` Statement

A `do` statement should have the following form:

`do` `{` statements `}` `while` (condition)`;`

Unlike the other compound statements, the `do` statement always ends with a `;` (semicolon).

`switch` Statement

A `switch` statement should have the following form:

`switch` (expression) `{` `case` expression`:` statements `default:` statements `}`

Each `case` is aligned with the `switch`. This avoids over-indentation.

Each group of statements (except the `default`) should end with `break`, `return`, or `throw`. Do not fall through.

`try` Statement

The `try` class of statements should have the following form:

`try {`
statements
`} catch (`variable`) {`
statements
}

`try {`
statements
`} catch (`variable`) {`
statements
‘ } finally { `statements` }‘

`continue` Statement

Avoid use of the `continue` statement. It tends to obscure the control flow of the function.

`with` Statement

The `with` statement should not be used[20].

Whitespace

Blank lines improve readability by setting off sections of code that are logically related.

Blank spaces should be used in the following circumstances:

- A keyword followed by `(` (left parenthesis) should be separated by a space.

```
    while (true) {
```

- A blank space should not be used between a function value and its `(` (left parenthesis). This helps to distinguish between keywords and function invocations.
- All binary operators except `.` (period) and `(` (left parenthesis) and `[` (left bracket) should be separated from their operands by a space.
- No space should separate a unary operator and its operand except when the operator is a word such as `typeof`.
- Each ; (semicolon) in the control part of a `for` statement should be followed with a space.
- Whitespace should follow every , (comma).

[20] http://yuiblog.com/blog/2006/04/11/with-statement-considered-harmful/

Bonus Suggestions

`{}` and `[]`

Use `{}` instead of `new Object()`. Use `[]` instead of `new Array()`.

Use arrays when the member names would be sequential integers. Use objects when the member names are arbitrary strings or names.

`,` (comma) Operator

Avoid the use of the comma operator except for very disciplined use in the control part of `for` statements. (This does not apply to the comma separator, which is used in object literals, array literals, `var` statements, and parameter lists.)

Block Scope

In JavaScript blocks do not have scope. Only functions have scope. Do not use blocks except as required by the compound statements.

Assignment Expressions

Avoid doing assignments in the condition part of `if` and `while` statements.

Is

```
    if (a = b) {
```

a correct statement? Or was

```
    if (a == b) {
```

intended? Avoid constructs that cannot easily be determined to be correct.

`===` and `!==` Operators.

It is almost always better to use the `===` and `!==` operators. The `==` and `!=` operators do type coercion. In particular, do not use `==` to compare against falsy values.

Confusing Pluses and Minuses

Be careful to not follow a `+` with `+` or `++`. This pattern can be confusing. Insert parens between them to make your intention clear.

```
    total = subtotal + +myInput.value;
```

is better written as

```
    total = subtotal + (+myInput.value);
```

so that the `+ +` is not misread as `++`.

`eval` is Evil

The `eval` function is the most misused feature of JavaScript. Avoid it.

`eval` has aliases. Do not use the `Function` constructor. Do not pass strings to `setTimeout` or `setInterval`.

Semicolons in JavaScript are optional

Original Article

http://mislav.uniqpath.com/2010/05/semicolons/

Mislav Marohnić, mislav.uniqpath.com[a]

[a]http://mislav.uniqpath.com

JavaScript is a scripting language where semicolons as statement terminators are *optional*, as in:

> **op·tion·al** (*adjective*)
> Available to be chosen but not obligatory

However, there is a lot of FUD (fear, uncertainty, and doubt) around this feature and, as a result, most developers in our community will recommend always including semicolons[21], just to be safe.

Safe from *what?* I've been searching for reasons programmers have to force semicolons on themselves. Here's what I generally found:

[21]http://stackoverflow.com/questions/444080/do-you-recommend-using-semicolons-after-every-statement-in-javascript

Spec is cryptic and JavaScript implementations differ

Rules for automatic semicolon insertion[22] are right here and are, while somewhat difficult to comprehend by a casual reader, quite explicitly laid out. As for JavaScript implementations where interpretation of these rules *differs*, well, I've yet to find one. When I ask developers about this, or find archived discussions, typically they are yeah, there's this browser where this is utterly broken, I simply forgot which. Of course, they never remember.

I write semicolon-less code and, in my experience, there isn't a JavaScript interpreter that can't handle it.

You can't minify JavaScript code without semicolons

There are 3 levels of reducing size of JavaScript source files: *compression* (e.g. gzip), *minification* (i.e. removing unnecessary whitespace and comments) and *obfuscation* (changing code, shortening variable and function names).

Compression like gzip is the easiest; it only requires one-time server configuration, doesn't need extra effort by developers and *doesn't change* your code. There was a time when IE6 couldn't handle it, but if I remember correctly it was patched years ago and pushed as a Windows update, and today nobody really cares anymore.

Minification and obfuscation *change your code*. They are tools which you run on your source code saying "here are some JavaScript files, try to make them smaller, but **don't change functionality**". I'm reluctant to use these tools because many developers report that if I don't use specific coding styles, like writing semicolons, they will break my code. I'm OK with people (community) forcing certain coding styles on me, but not tools.

Suppose I have code that works in every JavaScript implementation that I target (major browsers and some server-side implementations). If I run it through your minification tool and that tools *breaks* my code, then I'm sad to report that your tool is *broken*. If this tool edits JavaScript code, it'd better understand it as a real interpreter would.

While on the topic of minification, let's do a reality check. I took the jQuery source and removed all semicolons[23], then ran it through Google Closure Compiler[24]. Resulting size was 76,673 bytes. The size of original "jquery.min.js" was 76,674 (1 byte more). So you see, there was almost no change; and of course, its test suite passed as before.

How is that possible? Well, consider this code:

[22]http://bclary.com/2004/11/07/#a-7.9

[23]http://github.com/mislav/jquery/commit/4a2faf8987fc3fcb8aefc99def5b5ed2b4de190c

[24]http://code.google.com/closure/compiler/

```
var a=1
var b=2
var c=3
```

That's 24 bytes right there. Stamp semicolons everywhere and run it through a minifier:

```
var a=1;var b=2;var c=3;
```

Still 24 bytes. So, adding semicolons and removing newlines saved us a whopping zero bytes right there. Radical. Most size reduction after minification isn't gained by removing newline characters — it's thanks to removing code comments and leading indentation.

Update: a lot of people have pointed out that their minifiers *rewrite* this expression as `var a=1,b=2,c=3`. I know that some tools do this, but the point of this article is just to explore how semicolons relate to whitespace. If a minifier is capable of rewriting expressions (e.g. Closure Compiler) it means that it can also insert semicolons automatically.

Also, some people recommend forcing yourself do use curly braces for blocks, even if they're only one line:

```
// before
if(condition) stuff()

// after
if(condition){
  stuff()
}

// after minification
if(condition){stuff()}
```

Enforced curly braces add at least a byte to our expression, even after minification. I'm not sure what the benefit is here—it's not size and it's not readability, either.

Here are some other whitespace-sensitive languages that you might have heard about:

- Ruby — messing with spaces in expressions with operators and method calls can break the code
- Python — duh.
- HTML — see notes about Kangax's HTML minifier[25]
- Haml templates[26]

[25]http://perfectionkills.com/experimenting-with-html-minifier/
[26]http://haml-lang.com/

Of course, there's no need for minification on the server-side. I made this list for the sake of the following argument: Whitespace can, and often is, part of the (markup) language. It's not necessarily a bad thing.

It's good coding style

Also heard as:

- It's good to have them for the sake of consistency
- JSLint[27] will complain
- Douglas Crockford says so.[28]

This is another way of expressing the "everybody else is doing it" notion and is used by people during online discussion in the (rather common) case of a lack of arguments.

My advice on JSLint: don't use it. Why would you use it? If you believed that it helps you have less bugs in your code, here's a newsflash; only people can detect and solve software bugs, not tools. So instead of tools, get more people to look at your code.

Douglas Crockford also says "four spaces", and yet most popular JavaScript libraries are set in either tabs or two spaces. Communities around different projects are *different*, and that's just how it should be. As I've said before: let *people* and yourself shape your coding style, not some single person or tool.

You might notice that in this article I'm not telling you *should* be semicolon-free. I'm just laying out concrete evidence that you *can* be. The choice should always be yours.

As for *coding styles*, they exist so code is more readable and easier to understand for a group of people in charge of working on it. Think deeply if semicolons actually improve the readability of your code. What improves it the most is whitespace—indentation, empty lines to separate blocks, spaces to pad out expressions—and good variable and function naming. Look at some obfuscated code[29]; there are semicolons in there. Does it help readability? No, but what would really help is a lot of whitespace and original variable names.

Semicolon insertion bites back in return statements

When I searched for "JavaScript semicolon insertion", here is the problem most blog posts described:

[27]http://www.jslint.com/
[28]http://javascript.crockford.com/code.html
[29]http://img.skitch.com/20100509-qf8t69ad7cpmudwdksbw5hu6te.png

```
function add() {
  var a = 1, b = 2
  return
    a + b
}
```

When you're done trying to wrap your brain around why would anyone in their right mind want to write a return statement on a new line, we can continue and see how this statement is interpreted:

```
return;
  a + b;
```

Alas, the function didn't return the sum we wanted! But you know what? This problem *isn't* solved by adding a semicolon to the end of our wanted `return` expression (that is, after `a + b`). It's solved by *removing* the newline after `return`:

```
return a + b
```

Still, in an incredible display of ignorance these people actually *advise* their readers to avoid such issues by adding semicolons everywhere. Uh, alright, only it doesn't help this particular case at all. We just needed to understand better how the language is parsed.

The only real pitfall when coding without semicolons

Here is the only thing you have to be aware if you choose to code semicolon-less:

```
// careful: will break
a = b + c
(d + e).print()
```

This is actually evaluated as:

```
a = b + c(d + e).print();
```

This example is taken from an article about JavaScript 2.0 future compatibility[30], but I've ran across this in my own programs several times while using the module pattern[31].

Easy solution: when a line starts with parenthesis, prepend a semicolon to it.

[30]http://www.mozilla.org/js/language/js20-2000-07/rationale/syntax.html
[31]http://www.yuiblog.com/blog/2007/06/12/module-pattern/

```
;(d + e).print()
```

This might not be elegant, but does the job. Michaeljohn Clement elaborates on this[32] even further:

> If you choose to omit semicolons where possible, my advice is to insert them immediately before the opening parenthesis or square bracket in any statement that begins with one of those tokens, or any which begins with one of the arithmetic operator tokens `/`, `+`, or `-` if you should happen to write such a statement.

Adopt this advice as a rule and you'll be fine.

[32] http://inimino.org/~inimino/blog/javascript_semicolons