

JavaScript: The Good Parts

Douglas Crockford

Yahoo! Inc.

douglas@crockford.com

The World's Most Misunderstood Programming Language

JavaScript is a language with more than its share of bad parts. It went from non-existence to global adoption in an alarmingly short period of time. It never had an interval in the lab when it could be tried out and polished. It went straight into Netscape Navigator 2 just as it was, and it was very rough. When Java applets failed, JavaScript became the Language of the Web by default. JavaScript's popularity is almost completely independent of its qualities as a programming language.

Fortunately, JavaScript has some extraordinarily good parts. In JavaScript there is a beautiful, elegant, highly expressive language that is buried under a steaming pile of good intentions and blunders. The best nature of JavaScript is so effectively hidden that for many years the prevailing opinion of JavaScript was that it was an unsightly, incompetent toy. My intention here is to expose the goodness in JavaScript, an outstanding dynamic programming language. JavaScript is a block of marble, and I chip away the features that are not beautiful until the language's true nature reveals itself. I believe that the elegant subset I carved out is vastly superior to the language as a whole, being more reliable, readable, and maintainable.

A Language of Many Contrasts

JavaScript has many bad features, but it also has some of the best features to ever appear in a programming language. The JavaScript community demonstrates the broadest range of programmer skills of any programming language, from non-programmer cut-and-pasters to computer scientists and everyone in between. JavaScript's amazing expressive power is convincing evidence that it got something right.

Despite its advantages, there are many common complaints about JavaScript, including

- "JavaScript is not a language I know."
- "The browser programming experience is awful."
- "It's not fast enough."
- "The language is just a pile of mistakes."

Let's look at each of these.

"JavaScript is not a language I know."

In virtually all programming environments, such as desktop applications, server applications, or embedded systems, you get to choose the programming language. But in web applications, there is only one choice, and this is highly annoying if that language is not one that you are already familiar with. It is not uncommon for programmers when first confronted with JavaScript to willfully program in ignorance. This is not a smart way to work in any language.

"The browser programming experience is awful."

This is quite true, but the blame mostly belongs to the DOM (Document Object Model), the API of the browser. The DOM is not JavaScript. If you were to take your favorite programming language, strip away the standard libraries and replace them with the DOM, you would hate it.

"It's not fast enough."

This is a fair criticism. The current JavaScript implementations were optimized for fast time to market. I think that smart implementations could go ten to one hundred times faster.

"The language is just a pile of mistakes."

JavaScript is succeeding very well in an environment where Java was a total failure. There is much goodness in JavaScript, and most of the badness is easily avoided. By employing only the good parts, you make JavaScript a better language.

Influences

The design of JavaScript was influenced by Java, which gave JavaScript its syntax, by Self, which gave JavaScript dynamic objects with prototypal inheritance, and by Scheme, which gave JavaScript its lexically scoped functions.

JavaScript's nasty regular expressions came from Perl.

Bad Parts

Before we get to the good parts, let's visit a few of the bad parts.

JavaScript does not have a linker. Instead, all compilation units are combined in a single global space, where all top level functions and variables are global. This is terrible for reliability and even worse for security. JavaScript's global variable policy is the root cause of Cross Site Scripting attacks and other security mishaps.

The `+` operator both adds and concatenates. This is a bad thing in a loosely typed language. If you intend to add, you better make sure that both operands are numbers or you will be unpleasantly surprised.

JavaScript has a semicolon insertion mechanism that makes most uses of semicolons optional. Unfortunately, it can also mask errors and even cause errors.

JavaScript has a `typeof` operator that can identify the type of a value, which is a

very useful thing. Unfortunately, it does not distinguish between objects, arrays, and `null`.

The `with` statement is troublesome. It is a huge drag on reliability and performance. Be smart and don't ever use it, even if it looks like you can do something clever with it.

The `eval` function is the single most misused feature of the language. Don't use it unless you know what you are doing. People who know what they are doing don't use it.

An array is a linear allocation of memory in which elements are accessed by integers that are used to compute offsets. Arrays can be very fast data structures. Unfortunately, JavaScript does not have anything like that kind of array.

JavaScript has two sets of equality operators: `===` and `!==`, and their evil twins `==` and `!=`. The good ones work the way you would expect. If the two operands are of the same type and have the same value, then `===` produces `true`, and `!==` produces `false`. The evil twins do the right thing when the operands are of the same type, but if they are of different types they attempt to coerce the values. The rules by which they do that are complicated and unmemorable. These are some of the interesting cases:

```
' ' == '0'           // false
0 == ' '            // true
0 == '0'           // true

false == 'false'   // false
false == '0'       // true

false == undefined // false
false == null      // false
null == undefined  // true

' \t\r\n ' == 0    // true
```

The lack of transitivity is alarming. My advice is to never use the evil twins. Instead, always use `===` and `!==`. All of the above comparisons produce `false`

with the `===` operator.

If you attempt to extract a value from an object, and if the object does not have a member with that name, it returns the `undefined` value instead. This is useful because it allows us to easily determine if an object has a particular member.

In addition to `undefined`, JavaScript has a similar value called `null`. They are so similar that `==` thinks they are equal. That confuses some programmers into thinking that they are interchangeable, leading to code like

```
value = myObject[name];
if (value == null) {
    alert(name + ' not found.');
```

It is comparing the wrong value with the wrong operator. This code works because it contains two errors that cancel each other out. That is a crazy way to program. It is better written like this:

```
value = myObject[name];
if (value === undefined) {
    alert(name + ' not found.');
```

Good features that interact badly

Objects can inherit from other objects, which is a good thing. Functions can be members of objects, which is also a good thing because that is how we get methods. The `for..in` statement iterates over the names of the members of an object, which is another good thing.

Except that the `for..in` statement mixes inherited functions with the desired data members, which is rarely a useful thing.

The language design question was: Should `for..in` do a shallow skim or a deep dredge? It does a deep dredge. The thinking was that was the more powerful operation. If you have the deep form, you can simulate the shallow form, but if you only have the shallow form, you cannot recover the deep. So the programmer must explicitly filter out the deep members. Except they didn't tell anybody!

If you do not explicitly filter your `for..in` loops, then your program will fail when run with code from other parties, even if your code never calls their code.

Bad Heritage

JavaScript inherited blockless statements from C and Java.

```
if (foo)
    bar();
```

This convenience can save the programmer from having to type two characters. But those characters are needed to make the program text more resilient and better able to withstand bug insertion when it is edited by others.

JavaScript also has expression statements. This is a valid statement:

```
foo;
```

It doesn't do anything useful. It is probably a typo. But because it is valid, it will not be detected as an error.

JavaScript has binary floating point arithmetic. It is useful in a large class of applications, but not the sorts of applications that JavaScript is used for because it cannot correctly represent decimal fractions, so surprisingly,

```
0.1 + 0.2 !== 0.3
```

People have a reasonable expectation when they count money that the results will be exact. JavaScript gets it wrong.

Good Parts

The bad parts are more than compensated for by the good parts, which include lambda, dynamic objects, and loose typing.

Lambda

Lambda is an application of Alonso Church's Lambda Calculus. It is realized in JavaScript by functions which are first-class objects. Functions can be stored in variables, passed to other functions, or returned from functions. Functions can be defined inside of other functions. Functions can be stored in objects. A function

has access to the variables of the outer functions that it is contained within, even after the outer function has returned. This is called closure. This is one of the best ideas in the history of programming languages, and JavaScript was the first popular language to have it.

Dynamic Objects

JavaScript's objects are dynamic: You can add a new member to an object at any time simply by assigning it. There is no need to make a class. There are no classes, the language is class free.

Loose Typing

JavaScript has loose typing. That means that you do not declare the type of a variable or parameter. Any variable can hold values of any type. The fashion in most programming languages today demands strong typing, in which a declaration of type must be made for every variable and parameter. The theory is that strong typing allows a compiler to detect a large class of errors at compile time. The sooner we can detect and repair errors, the less they cost us. JavaScript is a loosely typed language, so JavaScript compilers are unable to detect type errors. This can be alarming to people who are coming to JavaScript from strongly typed languages. But it turns out that strong typing does not eliminate the need for careful testing. And I have found in my work that the sorts of errors that strong type checking finds are not the errors I worry about. On the other hand, I find loose typing to be liberating. I don't need to form complex class hierarchies. And I never have to cast or wrestle with the type system to get the behavior that I want.

Inheritance

Inheritance is object-oriented code reuse. There are two schools of thought on inheritance:

- Classical
- Prototypal

Most languages today are classical, in which classes are defined, and objects are instances of classes. Classes can inherit from other classes. JavaScript, on the

other hand, is prototypal: An object can inherit directly from another object. There are no classes. This is a powerfully direct, expressive way to think about inheritance. But it is unfamiliar, so it is common to see construction of classical structures on top of JavaScript's prototypal forms. The language has enough expressive power to handle classical patterns. (Try writing prototypally in a classical language. You'll find those languages are not as expressive.)

An object contains a secret link to another object. That link is used to delegate to, or to inherit from, another object. For example, I can beget a new object that inherits from an old object.

```
newObject = Object.beget(oldObject);
```

If we attempt to access a member from the new object, and if the new object does not have such a member, then we will automatically obtain it from the old object.

We can implement the `beget` method as follows:

```
if (typeof Object.beget !== 'function') {
  Object.beget = function (old) {
    function F() {}
    F.prototype = old;
    return new F();
  };
}
```

JavaScript is itself conflicted about its prototypal nature. So instead of providing a `beget` function, it recommends the use of constructor functions, which are functions that have a `prototype` property containing the inherited material, and a `new` operator that is used to call the constructor functions.

If you forget to use the `new` prefix, a new object will not be created. Instead you will be unintentionally clobbering global variables, which can be very bad. There is no compile-time warning. There is no run-time warning. But if you avoid constructor functions, there is never a need to use `new`, and that class of mishaps can be avoided.

Closure

Suppose you want to make a function that converts a digit into a word. One way

to do that is to make a global variable that contains an array of names.

```
var names = ['zero', 'one', 'two',
            'three', 'four', 'five', 'six',
            'seven', 'eight', 'nine'];

var digit_name = function (n) {
    return names[n];
};

alert(digit_name(3));    // 'three'
```

The problem with that is that `names` is a global variable. If any other code in our application has a variable named `names`, then we can expect the program to fail. Such failures can be difficult to diagnose, and often don't show up during testing. The best course is to avoid global variables.

We could move the definition of `names` to inside of `digit_name`.

```
var digit_name = function (n) {
    var names = ['zero', 'one', 'two',
                'three', 'four', 'five', 'six',
                'seven', 'eight', 'nine'];
    return names[n];
};

alert(digit_name(3));    // 'three'
```

But that can cause the creation of the `names` array every time the function is called. We only want to build the array once. We can do this with closure.

```
var digit_name = function () {
    var names = ['zero', 'one', 'two',
                'three', 'four', 'five', 'six',
                'seven', 'eight', 'nine'];
```

```

    return function (n) {
        return names[n];
    };
}();
alert(digit_name(3));    // 'three'

```

We assign to `digit_name` the result of a function that returns a function. The outer function defines a `names` variable. The inner function uses the `names` variable. The outer function immediately returns the inner function. The inner function continues to enjoy access to the outer function's stuff as long as it survives. This is closure.

A Module Pattern

We can extend this idea to construct objects. Here we make a singleton object that contains two methods (`firstMethod` and `secondMethod`) which have access to a private variable and a private function. Notice that we are not assigning a function to `singleton`. We are assigning the result of invoking a function to `singleton`, and that result is an object.

```

var singleton = function () {
    var privateVariable;
    function privateFunction(x) {
        ...privateVariable...
    }
    return {
        firstMethod: function (a, b) {
            ...privateVariable...
        },
        secondMethod: function (c) {
            ...privateFunction()...
        }
    };
}();

```

This gives us a way to create objects that have private members. We can create

stateful objects without dependence on global variables.

The module pattern is easily transformed into a powerful constructor pattern.

Power Constructors

We can generalize the module pattern to make lots of instances. The recipe is simple.

First, make an object. You can use an object literal, as we did with the `singleton`. Or you can call a constructor function with the `new` prefix. Or you can use the `Object.get` function. Or you can call another power constructor.

Second, define some variables and functions. These are simply vars within your power constructor. These become private members.

Third, augment the object with privileged methods. Simply assign methods to your new object. Those methods will have privileged access to the variables and functions from the second step and to the power constructor's parameters.

Four and finally, return the new object.

```
function myPowerConstructor(x) {
    var that = otherMaker(x);
    var secret = f(x);
    that.priv = function () {
        ... secret x that ...
    };
    return that;
}
```

One nice thing about power constructors is that they do not require use of the `new` operator. If someone slips up and uses `new` when it isn't needed, no harm is done.

Working with the Grain

JavaScript have a lot of expressive power, so it is possible to program JavaScript in a Java style, or in a Python style, or in a Basic style. It took me a long time to

figure out how to program in a JavaScript style, taking best advantage of the good parts and completely avoiding the bad parts. After all, why would you want to build something good out of bad parts?

For years I have been observing the errors that I make, and the errors I've seen other make, looking not just to avoid them, but to mechanically detect them. The price for this automatic verification is that you must write in a professional subset of JavaScript. I think that is a good bargain.

I developed a program called JSLint. It is a JavaScript parser written in JavaScript. I recommend that you use JSLint as a code quality tool. Have it look at your code before you test it. It will save you a lot of time.

But JSLint comes with a warning: It will hurt your feelings. It is a ruthless, heartless program. But its advice is good. If you accept its advice, your programs will get stronger.

Style Isn't Subjective

There is a perpetual argument about the best way to format blocks. There is the K&R style with the `{` on the right:

```
block {  
  
    ....  
  
}
```

And the other style with the `{` on the left:

```
block  
  
{  
  
    ....  
  
}
```

Programmers can come up with endless arguments and rationalizations about which is vastly superior, which usually bottom out at what they learned at school or at their first job. From that you could conclude that style doesn't matter, but I don't think that is true. People who understand the importance of style will agree

that whichever style you use, you must use it consistently. Sloppiness makes code harder to read and can hide bugs.

I don't know which style is best for other languages, but I can prove that the right style is the one that must be used for JavaScript.

Consider this return statement:

```
return {
    ok: true
};
```

This returns an object value. This is very commonly used in JavaScript. We saw it in the power constructor above. What happens if we write it in the bad style:

```
return
{
    ok: false
};
```

The result is a silent error. Instead of returning an object with an `ok` member, it will return `undefined`. There is no compile time warning. There is no runtime warning. What's going on?

JavaScript's semicolon insertion mechanism inserts a semicolon after the `return`. Sometimes it doesn't insert semicolons where you expect. This time it inserts the semicolon some place you do not expect. Semicolon insertion was a well intentioned feature, but it must be seen as a design error in the language.

But there are other bad things going on, which is why we get no warning. The `{` operator is overloaded. Sometimes it means an object literal, sometimes it means a block. In this case, it is treated as a block.

The `ok:` is seen as a statement label. JavaScript ignore the fact that no `break` statement references it. JavaScript also ignore the fact that it is not on a `switch`, `do`, `for`, or `while` statement, the only statements that it makes sense to label.

The word `false` is seen as a useless expression statement. Semicolon insertion sticks another semicolon after the `false`.

The semicolon after the `}` is ignored because of the null statement rule. And finally, even though the code after the `return` is unreachable, it does not

produce an error either.

So the code that JavaScript saw was

```
return;  
{  
    ok: false;  
}
```

Bad style produced a bad result. Good style for this language demands that you put `{` on the right. Bad style caused JavaScript to totally misinterpret the program. Think about how painful it would be to have to debug that.

Now you might be thinking this is the stupidest thing you have ever seen. You might be wondering how did this ever become a world standard? And you might be asking yourself why am I betting my career on this piece of crap?

It is pretty shocking to see the way that several design errors can interact to produce such a result. The bad parts are truly horrifying.

But the good parts are really good. And if you stick to the good parts, JSLint will find all of those problems. With the right understanding of the language, and with the right tools, you can write excellent JavaScript.

And that will let you take advantage of one of JavaScript's best features: It can help you reach more people than any other language. There are at least a billion people now who can be reached with JavaScript enabled browsers, and the number will ultimately grow to reach everyone in the world. No other language has that reach.

It seems strange that such an odd, flawed language should become the most important programming language in the world, but it has. It is not something I would have chosen.

But I have chosen to write good programs in the language. You can too.