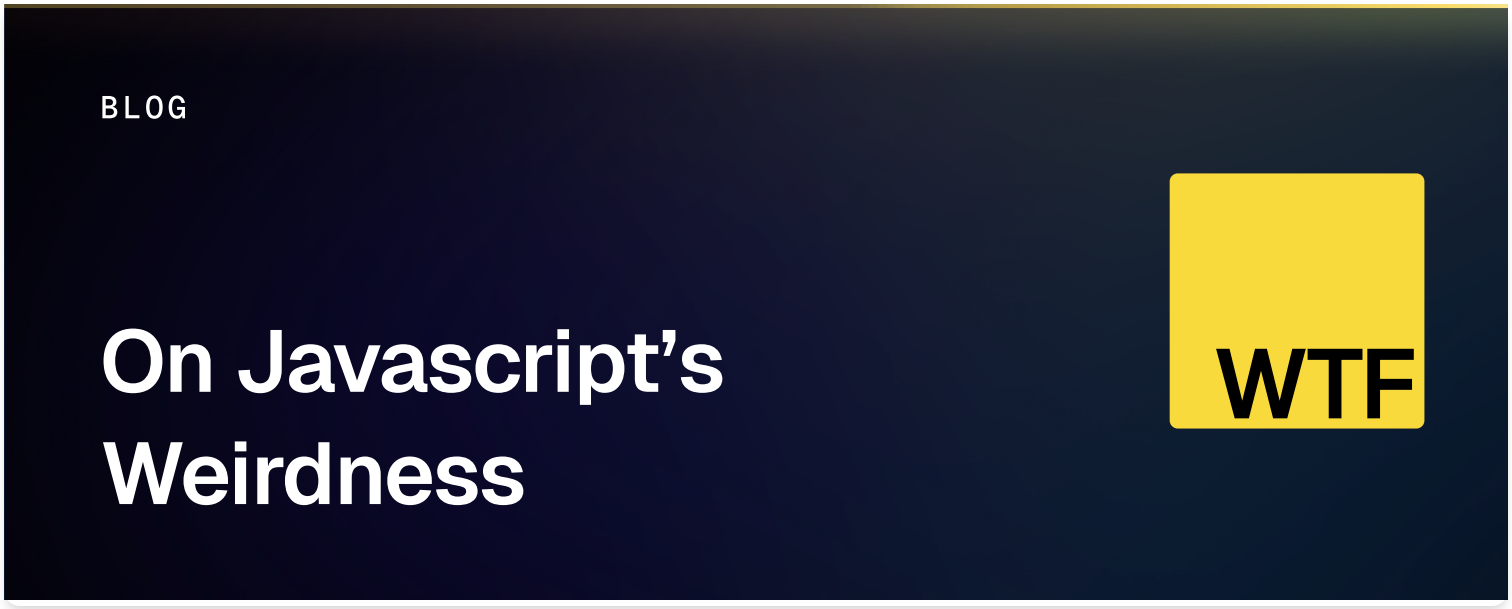


On JavaScript's Weirdness

Konstantin Wohlwend
Apr 1, 2025

> *"JavaScript sucks because* `'0' == 0` *!"*
>
> *- literally everyone ever*

Sure, that part of JavaScript sucks, but every JS setup these days contains a linter that yells at you for code like that.

Instead, I want to talk about some of the weirder quirks of JavaScript — those that are much more insidious than that — the kind of stuff you wouldn't find on r/ProgrammerHumor or a JS tutorial.

All of them can occur in any JavaScript/ECMAScript environment (so browser, Node.js, etc.), with or without `use strict` enabled. (If you're working on legacy projects without strict mode, you should run. And if you don't know where to: Stack Auth is hiring.)

#1. eval is worse than you think

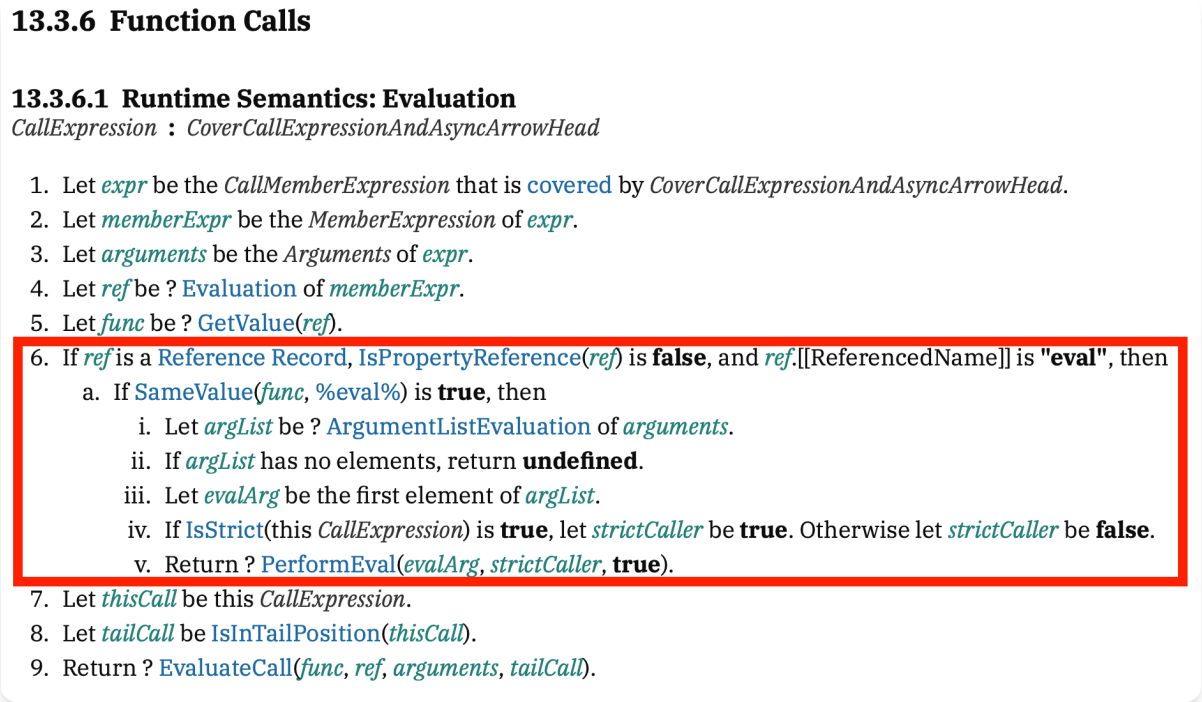
How silly it would be to think that these two are the same:


```
function a(s) {
  eval("console.log(s)");
}
a("hello");  // prints "hello"


function b(s) {
  const evalButRenamed = eval;
  evalButRenamed("console.log(s)");
}
b("hello");  // Uncaught ReferenceError: s is not defined
```

The difference is that the former has access to variables in the current scope, whereas the renamed version can only access the global scope.

Why? Turns out that ECMAScript's definition for function calls has a hardcoded special case, which runs a slightly different algorithm when the function invoked is called `eval`:



13.3.6 Function Calls

13.3.6.1 Runtime Semantics: Evaluation

CallExpression : *CoverCallExpressionAndAsyncArrowHead*

1. Let *expr* be the *CallMemberExpression* that is covered by *CoverCallExpressionAndAsyncArrowHead*.
2. Let *memberExpr* be the *MemberExpression* of *expr*.
3. Let *arguments* be the *Arguments* of *expr*.
4. Let *ref* be ? Evaluation of *memberExpr*.
5. Let *func* be ? GetValue(*ref*).
6. If *ref* is a Reference Record, IsPropertyReference(*ref*) is **false**, and *ref*.[[ReferencedName]] is **"eval"**, then
   a. If SameValue(*func*, %eval%) is **true**, then
      i. Let *argList* be ? ArgumentListEvaluation of *arguments*.
      ii. If *argList* has no elements, return **undefined**.
      iii. Let *evalArg* be the first element of *argList*.
      iv. If IsStrict(this *CallExpression*) is **true**, let *strictCaller* be **true**. Otherwise let *strictCaller* be **false**.
      v. Return ? PerformEval(*evalArg*, *strictCaller*, **true**).
7. Let *thisCall* be this *CallExpression*.
8. Let *tailCall* be IsInTailPosition(*thisCall*).
9. Return ? EvaluateCall(*func*, *ref*, *arguments*, *tailCall*).

I can't stress enough how insane it is to have this hack in the specification for *every single function call*! Although it goes without saying that any half-decent JS engine will optimize it, so while there is no direct performance penalty, it certainly makes build tools & engines more complicated. (As an example, this means that `(0, eval)(...)` differs from `eval(...)`, so minifiers must consider this when removing seemingly dead code. Scary!)

#2. JS loops pretend their variables are captured by value

Yes, the title makes no sense, but you'll see what I mean in just a second. Let's start with an example:

```
for (let i = 0; i < 3; i++) {
  setTimeout(() => console.log(i));
}
// prints "0 1 2" — as expected

let i = 0;
for (i = 0; i < 3; i++) {
  setTimeout(() => console.log(i));
}
// prints "3 3 3" — what?
```

Why does it matter where the variable is defined? It's the same variable either way, right?

In any programming language, when you capture values with a lambda/arrow function, there are two ways to pass variables: By value (copy) or by reference (passing a pointer). Some languages, like C++, let you pick:

```
// C++ code below:

// capture by value
int byValue = 0;
auto func1 = [byValue] { std::cout << byValue << std::endl; };
byValue = 1;
func1();
// prints 0, because the variable's value is copied

// capture by reference
int byReference = 0;
auto func2 = [&byReference] { std::cout << byReference << std::endl; };
byReference = 1;
func2();
// prints 1, because the variable is captured by reference
```

That said, most high-level languages (JS, Java, C#, ...) capture variables by reference:

```
let byReference = 0;
const func = () => console.log(byReference);
byReference = 1;
func();
// prints 1
```

More often than not this is what you want, but it's particularly undesirable in loops. There, it's common you need to do something with the iterator variable in a callback function:

```
// C# code below:
for (int i = 0; i < 3; i++) {
  setTimeout(() => {
    Console.WriteLine(i);
  }, 1000 * i);
}
// prints "3 3 3" — probably not what you wanted
```

As a "fix", the ECMAScript standard hacks for-loop variables to have a different behavior, but only if they're defined in the loop header:

```
for (let i = 0; i < 3; i++) {
  setTimeout(() => {
    console.log(i);
  }, 1000 * i);
}
// prints "0 1 2"

// but it doesn't work if we factor out the loop variable:
let i = 0;
for (i = 0; i < 3; i++) {
  setTimeout(() => {
    console.log(i);
  }, 1000 * i);
}
// prints "3 3 3"
```

I posted about this on Twitter, and a bunch of you told me that this "makes sense" if you understand how for-loops & closures are defined in terms of scope in the ECMAScript standard. That's true, although it's weird in the sense that it really doesn't fit most people's intuition. More precisely, if you want to unroll a for-loop in JavaScript, this would be the spec-compliant way to do it:

```
// intuitive way to unroll a for-loop (WRONG in JS)
let i = 0;
while (i < 3) {
  // ... for-loop body ...
  i++;
}

// spec-compliant way to unroll a for-loop
let _iteratorVariable = 0;
while (_iteratorVariable < 3) {
  let i = _iteratorVariable;
  // ... for-loop body ...
  i++;
  _iteratorVariable = i;
}
```

That said, the fact that nearly no one talks about it is a testament of how those "hacks" can sometimes be very useful. (TypeScript's type system has plenty of "useful" hacks like these, and I think that's part of why it's so popular despite its complexity — some day I should write a post about that.)

#3. That falsy object

Common knowledge is that there are 8 falsy values in JavaScript: `false`, `+0`, `-0`, `NaN`, `""`, `null`, `undefined`, and `0n`.

Oops, I lied. There's actually a ninth one, and it's an object:

```
console.log(document.all); // prints HTMLAllCollection [<html>, <head>, ...]
console.log(Boolean(document.all)); // prints false
```

I almost didn't include this one in this post, because it only affects browsers. But it turns out that it's actually specified in the ECMAScript standard, not in the DOM standard (where you'd usually see

browser-specific stuff), so I left it in:

> **B.3.6 The [[IsHTMLDDA]] Internal Slot**
>
> An *[[IsHTMLDDA]] internal slot* may exist on host-defined objects. Objects with an [[IsHTMLDDA]] internal slot behave like **undefined** in the ToBoolean and IsLooselyEqual abstract operations and when used as an operand for the **`typeof`** operator.
>
> NOTE Objects with an [[IsHTMLDDA]] internal slot are never created by this specification. However, the **`document.all`** object in web browsers is a host-defined exotic object with this slot that exists for web compatibility purposes. There are no other known examples of this type of object and implementations should not create any with the exception of **`document.all`**.

Why? Because on old versions of Internet Explorer, `document.getElementById` was not available and instead there was a property called `document.all`, so a lot of code was written like this:

```
if (document.all) {  // IE-specific
  // do something with document.all
} else {  // every other browser
  // do something with document.getElementById
}
```

To be compatible with IE, other browsers then went on to implement `document.all` too. However, it's much slower than `document.getElementById`, so those browsers decided that `document.all` should be falsy, in order to make code like the above take the fast path. Don't we love IE?

#4. Graphemes & string iteration

It's relatively well-known that strings in JavaScript are UTF-16 encoded, which means that there are low- and high-surrogates. Essentially, it means that some characters take up two UTF-16 code units:

```
const japanese = "𠮷";
console.log(japanese.length);  // prints 2
```

```
console.log(japanese.charCodeAt(0));  // prints 55362
console.log(japanese.charCodeAt(1));  // prints 57271
```

Surrogates always come in groups of two, never more. So, sensibly, if you have `n` characters, then `String.prototype.length` will always be between `n` and `2n`, depending on how many surrogates there are.

But then what's the output of this?

```
const family = "👨‍👩‍👧‍👦";  // two family emojis
console.log(family.length);  // prints 23
```

If you know Unicode well, you'll know that surrogates don't tell the whole story — some characters (particularly emojis) consist of multiple Unicode code points (each of which may be a single UTF-16 code unit, or a surrogate pair).

Now, what if we want to iterate over them?

```
const family = "👨‍👩‍👧‍👦";
let count = 0;
for (const char of family) {
  count++;
}
console.log(count);  // prints 15
```

A different number? Clearly something is off here.

Okay, whatever, the new `Intl` APIs exist for this purpose and they fix this mess. Right?

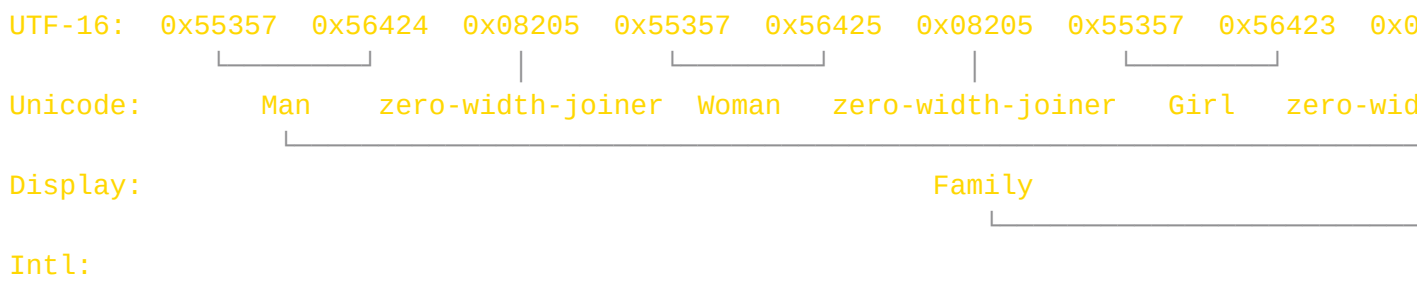
```
const family = "👨‍👩‍👧‍👦";
const chars = new Intl.Segmenter().segment(family);
console.log([...chars].length);  // prints 1
```

Still not 2!

Essentially, there are four sensible notions of "string length", and JavaScript mixes them all:

1. 23, the number of UTF-16 code units (most string functions, such as `.length`, `.split`, etc.)
2. 15, the number of Unicode code points (when iterating over strings with `for`)
3. 2, the number of *display characters* (may differ based on your browser's emoji support)
4. 1, the number of *extended grapheme clusters* (`Intl.Segmenter`)

If we paste the string above into a Unicode analyzer, it will make more sense:



Essentially, each Unicode code point is exactly one or two UTF-16 code units. Every browser/font has its own rules on how to merge them into display characters, and the extended grapheme cluster algorithm tries to approximate that, but isn't perfect.

If you're curious, Henri Sivonen wrote this excellent blog post on what other languages do, but sadly no solution is perfect because internationalization is a fundamentally hard problem. Although, I guess you can always just get rid of Unicode altogether.

#5. Sparse arrays

You can just repeat commas in arrays to make some of the elements `undefined`:

```
const sparse = [1, , , 4];
console.log(sparse[0], sparse[1], sparse[2], sparse[3]);  // prints 1 undefined unde
```

Or not?


```
const sparse = [1, , , 4];
sparse.forEach(e => console.log(e));  // prints 1 4 — doesn't print undefined
```

Let's compare it to a normal array:

```
const dense = [undefined, undefined];
const sparse = [,,];

console.log(dense.length); // prints 2
console.log(sparse.length); // prints 2

console.log(dense); // prints [undefined, undefined]
console.log(sparse); // prints [empty × 2]

console.log(dense.map(x => 123)); // prints [123, 123]
console.log(sparse.map(x => 123)); // prints [empty × 2]
```

This is called a "sparse array". The easiest way to understand what's going on is using `Object.entries`:

```
console.log(Object.entries([1, undefined, undefined, 4]));
// prints [
//   ['0', 1],
//   ['1', undefined],
//   ['2', undefined],
//   ['3', 4]
// ]

console.log(Object.entries([1, , , 4]));
// prints [
//   ['0', 1],
//   ['3', 4]
// ]
```

JavaScript arrays are really just objects, and array elements are just properties on it. If some of the properties are missing, this completely messes up a lot of the built-in array methods. We call this a sparse array.

That said, you probably shouldn't use sparse arrays at all. Unfortunately, the `Array` constructor creates sparse arrays by default, leading to very unnatural code:

```
const sparse = new Array(4);
console.log(sparse); // prints [empty × 4]

// this one doesn't work either:
const stillNotDense = new Array(4).map(x => 123);
console.log(stillNotDense); // prints [empty × 4]

// but you need to do this:
const dense = new Array(4).fill(undefined).map(x => 123);
console.log(dense); // prints [123, 123, 123, 123]

// or you could write this:
const alsoDense = Array.from({ length: 4 }, () => 123);
console.log(alsoDense); // prints [123, 123, 123, 123]
```

If that doesn't convince you, sparse arrays also have absolutely atrocious performance. Just don't use them in your code, ever, and you'll be fine.

#6. Weird ASI quirks

What will this code print? (Hint: It's not 2 1 4 3.)

```
function f1(a, b, c, d) {
  [a, b] = [b, a]
  [c, d] = [d, c]
  console.log(a, b, c, d)
}

f1(1, 2, 3, 4)
```

▶ Spoiler

The fact that I am missing semicolons is a good hint at what's going on. There's a fairly complicated algorithm called *Automatic Semicolon Insertion* (ASI) that tries to guess with a bunch of he where they're supposed to go.

```
[a, b] = [b, a]
[c, d] = [d, c]

// is interpreted by ASI as:

[a, b] = [b, a][c, d] = [d, c]
               ^  ^
               |  |
               |  comma operator
               |
               array access

// which is the same as:

[a, b] = [4, 3]
[b, a][4] = [4, 3]
```

The exact mechanics of the ASI are out of scope for this post, but in essence, it checks whether there's a syntax error, and if there is, and there's a newline right before it, it inserts a semicolon. Hence, if there's no syntax error, it usually won't insert a semicolon.

From the perspective of the ECMAScript standardization committee, this rule is quite restrictive. Adding new syntax to the language means that old syntax error may no longer be syntax errors, but because the ASI relies on syntax errors to occur at specific places, every new syntax could potentially break old code. For this purpose, there are special so-called *restricted productions* in the language, which always insert a semicolon if there's a newline, even if the code would be syntactically correct otherwise.

Et cetera

Here is a list of odd behaviors for which I didn't have enough space to write about:

- Anything that has to do with `==` and `!=`
- Anything that has to do with type coercion
- Anything that has to do with `this`
- NaN is not equal to anything
- +0 vs. -0
- Anything that has to do with floating-point precision, or otherwise stuff that's covered by IEEE 754
- `typeof null` is `"object"`
- Anything that uses non-strict mode or `var`
- Returning primitive values from constructors
- Prototype pollution
- `Array.sort` converting numbers to strings
- ...

If you know any quirks I haven't listed here, I'd love if you could let me know on Twitter or Bluesky. And if you haven't yet, check out our blog post on OAuth!