ES6 In Depth: Symbols

By Jason Orendorff

Posted on June 11, 2015 in ES6 In Depth, Featured Article, and JavaScript

Share This

ES6 In Depth is a series on new features being added to the JavaScript programming language in the 6th Edition of the ECMAScript standard, ES6 for short.

Note: There is now a Vietnamese translation of this post, created by Julia Duong of the Coupofy team.

What are ES6 symbols?

Symbols are not logos.

They're not little pictures you can use in your code.

```
let 😻 = 🐱 × 😍;  // SyntaxError
```

They're not a literary device that stands for something else.

They're definitely not the same thing as cymbals.

(It is not a good idea to use cymbals in programming. They have a tendency to crash.)

So, what *are* symbols?

The seventh type

Since JavaScript was first standardized in 1997, there have been six *types*. Until ES6, every value in a JS program fell into one of these categories.

- Undefined
- Null
- Boolean
- Number
- String
- Object

Each type is a set of values. The first five sets are all finite. There are, of course, only two Boolean values, `true` and `false`, and they aren't making new ones. There are rather more Number and String values. The standard says there are 18,437,736,874,454,810,627 different Numbers (including `NaN`, the Number whose name is short for "Not a Number"). That's nothing compared to the number of different possible Strings, which I think is $(2^{144,115,188,075,855,872} - 1) \div 65{,}535$ ...though I may have miscounted.

The set of Object values, however, is open-ended. Each object is a unique, precious snowflake. Every time you open a Web page, a rush of new objects is created.

ES6 symbols are values, but they're not strings. They're not objects. They're something new: a seventh type of value.

Let's talk about a scenario where they might come in handy.

One simple little boolean

Sometimes it would be awfully convenient to stash some extra data on a JavaScript object that really belongs to someone else.

For example, suppose you're writing a JS library that uses CSS transitions to make DOM elements zip around on the screen. You've noticed that trying to apply multiple CSS transitions to a single `div` at the same time doesn't work. It causes ugly, discontinuous "jumps". You think you can fix this, but first you need a way to find out if a given element is already moving.

How can you solve this?

One way is to use CSS APIs to ask the browser if the element is moving. But that sounds like overkill. Your library should *already know* the element is moving; it's the code that set it moving in the first place!

What you really want is a way to *keep track* of which elements are moving. You could keep an array of all moving elements. Each time your library is called upon to animate an element, you can search the array to see if that element is already there.

Hmm. A linear search will be slow if the array is big.

What you really want to do is just set a flag on the element:

```
if (element.isMoving) {
  smoothAnimations(element);
}
element.isMoving = true;
```

There are some potential problems with this too. They all relate to the fact that your code isn't the only code using the DOM.

1. Other code using `for-in` or `Object.keys()` may stumble over the property you created.
2. Some other clever library author may have thought of this technique first, and your library would interact badly with that existing library.
3. Some other clever library author may think of it in the future, and your library would interact badly with that future library.
4. The standard committee may decide to add an `.isMoving()` method to all elements. Then you're *really* hosed!

Of course you can address the last three problems by choosing a string so tedious or so silly that nobody else would ever name anything that:

```
if (element.__$jorendorff_animation_library$PLEASE_DO_NOT_USE_THIS_PROPERTY$isMoving__) {
  smoothAnimations(element);
}
element.__$jorendorff_animation_library$PLEASE_DO_NOT_USE_THIS_PROPERTY$isMoving__ = true;
```

This seems not quite worth the eye strain.

You could generate a practically unique name for the property using cryptography:

```
// get 1024 Unicode characters of gibberish
var isMoving = SecureRandom.generateName();

...

if (element[isMoving]) {
  smoothAnimations(element);
}
element[isMoving] = true;
```

The `object[name]` syntax lets you use literally any string as a property name. So this will work: collisions are virtually impossible, and your code looks OK.

But this is going to lead to a bad debugging experience. Every time you `console.log()` an element with that property on it, you'll be looking a huge string of garbage. And what if you need more than one property like this? How do you keep them straight? They'll have different names every time you reload.

Why is this so hard? We just want one little boolean!

Symbols are the answer

Symbols are values that programs can create and use as property keys without risking name collisions.

```
var mySymbol = Symbol();
```

Calling `Symbol()` creates a new symbol, a value that's not equal to any other value.

Just like a string or number, you can use a symbol as a property key. Because it's not equal to any string, this symbol-keyed property is guaranteed not to collide with any other property.

```
obj[mySymbol] = "ok!";  // guaranteed not to collide
console.log(obj[mySymbol]);  // ok!
```

Here is how you could use a symbol in the situation discussed above:

```
// create a unique symbol
var isMoving = Symbol("isMoving");

...

if (element[isMoving]) {
  smoothAnimations(element);
}
element[isMoving] = true;
```

A few notes about this code:

- The string `"isMoving"` in `Symbol("isMoving")` is called a *description*. It's helpful for debugging. It's shown when you write the symbol to `console.log()`, when you convert it to a string using `.toString()`, and possibly in error messages. That's all.
- `element[isMoving]` is called a *symbol-keyed property*. It's simply a property whose name is a symbol rather than a string. Apart from that, it is in every way a normal property.
- Like array elements, symbol-keyed properties can't be accessed using dot syntax, as in `obj.name`. They must be accessed using square brackets.
- It's trivial to access a symbol-keyed property if you've already got the symbol. The above example shows how to get and set `element[isMoving]`, and we could also ask `if (isMoving in element)` or even `delete element[isMoving]` if we needed to.
- On the other hand, all of that is only possible as long as `isMoving` is in scope. This makes symbols a mechanism for weak encapsulation: a module that creates a few symbols for itself can use them on whatever objects it wants to, **without fear of colliding** with properties created by other code.

Because symbol keys were designed to avoid collisions, JavaScript's most common object-inspection features simply ignore symbol keys. A `for-in` loop, for instance, only loops over an object's string keys. Symbol keys are skipped. `Object.keys(obj)` and `Object.getOwnPropertyNames(obj)` do the same. But symbols are not exactly private: it is possible to use the new API `Object.getOwnPropertySymbols(obj)` to list the symbol keys of an object. Another new API, `Reflect.ownKeys(obj)`, returns both string and symbol keys. (We'll discuss the `Reflect` API in full in an upcoming post.)

Libraries and frameworks will likely find many uses for symbols, and as we'll see later, the language itself is using of them for a wide range of purposes.

But what are symbols, exactly?

```
> typeof Symbol()
"symbol"
```

Symbols aren't exactly like anything else.

They're immutable once created. You can't set properties on them (and if you try that in strict mode, you'll get a TypeError). They can be property names. These are all string-like qualities.

On the other hand, each symbol is unique, distinct from all others (even others that have the same description) and you can easily create new ones. These are object-like qualities.

ES6 symbols are similar to the more traditional symbols in languages like Lisp and Ruby, but not so closely integrated into the language. In Lisp, all identifiers are symbols. In JS, identifiers and most property keys are still considered strings. Symbols are just an extra option.

One quick caveat about symbols: unlike almost anything else in the language, they can't be automatically converted to strings. Trying to concatenate a symbol with strings will result in a TypeError.

```
> var sym = Symbol("<3");
> "your symbol is " + sym
// TypeError: can't convert symbol to string
> `your symbol is ${sym}`
// TypeError: can't convert symbol to string
```

You can avoid this by explicitly converting the symbol to a string, writing `String(sym)` or `sym.toString()`.

Three sets of symbols

There are three ways to obtain a symbol.

- **Call `Symbol()`.** As we already discussed, this returns a new unique symbol each time it's called.
- **Call `Symbol.for(string)`.** This accesses a set of existing symbols called the *symbol registry*. Unlike the unique symbols defined by `Symbol()`, symbols in the symbol registry are shared. If you call `Symbol.for("cat")` thirty times, it will return the *same* symbol each time. The registry is useful when multiple web pages, or multiple modules within the same web page, need to share a symbol.
- **Use symbols like `Symbol.iterator`, defined by the standard.** A few symbols are defined by the standard itself. Each one has its own special purpose.

If you still aren't sure if symbols will be all that useful, this last category is interesting, because they show how symbols have already proven useful in practice.

How the ES6 spec is using well-known symbols

We've already seen one way that ES6 uses a symbol to avoid conflicts with existing code. A few weeks ago, in the post on iterators, we saw that the loop `for (var item of myArray)` starts by calling `myArray[Symbol.iterator]()`. I mentioned that this method could have been called `myArray.iterator()`, but a symbol is better for backward compatibility.

Now that we know what symbols are all about, it's easy to understand why this was done and what it means.

Here are a few of the other places where ES6 uses well-known symbols. (These features are not implemented in Firefox yet.)

- **Making `instanceof` extensible.** In ES6, the expression *`object` `instanceof` `constructor`* is specified as a method of the constructor: `constructor[Symbol.hasInstance](object)`. This means it is extensible.
- **Eliminating conflicts between new features and old code.** This is seriously obscure, but we found that certain ES6 `Array` methods broke existing web sites *just by being there.* Other Web standards had similar problems: simply adding new methods in the browser would break existing sites. However, the breakage was mainly caused by something called *dynamic scoping*, so ES6 introduces a special symbol, `Symbol.unscopables`, that Web standards can use to prevent certain methods from getting involved in dynamic scoping.
- **Supporting new kinds of string-matching.** In ES5, `str.match(myObject)` tried to convert `myObject` to a `RegExp`. In ES6, it first checks to see if `myObject` has a method `myObject[Symbol.match](str)`. Now libraries can provide custom string-parsing classes that work in all the places where `RegExp` objects work.

Each of these uses is quite narrow. It's hard to see any of these features by themselves having a major impact in my day-to-day code. The long view is more interesting. Well-known symbols are JavaScript's improved version of the `__doubleUnderscores` in PHP and Python. The standard will use them in the future to add new hooks into the language with no risk to your existing code.

When can I use ES6 symbols?

Symbols are implemented in Firefox 36 and Chrome 38. I implemented them for Firefox myself, so if your symbols ever act like cymbals, you'll know who to talk to.

To support browsers that do not yet have native support for ES6 symbols, you can use a polyfill, such as core.js. Since symbols are not exactly like anything previously in the language, the polyfill isn't perfect. Read the caveats.

Next week, we'll have *two* new posts. First, we'll cover some long-awaited features that are finally coming to JavaScript in ES6—and complain about them. We'll start with two features that date back almost to the dawn of programming. We'll continue with two features that are very similar, but *powered by ephemerons.* So please join us next week as we look at ES6 collections in depth.

And, stick around for a bonus post by Gastón Silva on a topic that isn't an ES6 feature at all, but might provide the nudge you need to start using ES6 in your own projects. See you then!

About Jason Orendorff

https://blog.mozilla.org/jorendorff/

More articles by Jason Orendorff…

I'm okay with Mozilla handling my info as explained in this Privacy Policy.

Sign up now

27 comments

Christoph

Nice series of articles!
I have to admit that I giggled when I read the cymbals-in-programming pun :)

June 12th, 2015 at 02:39

David Madore

Doesn't this feature risk creating uncollectable garbage (I mean garbage that the GC will never be able to reclaim)? The idea of being able to `mykey = Symbol()` and then `anyobject[mykey] = myvalue` with no fear of key conflict is great, but if the idea of symbols being private is taken seriously, then whenever `mykey` goes out of scope and becomes garbage, *anything indexed by it should become garbage as well* because they are now unheld references (if one wants every subroutine to feel free putting temporary data on any kind of global object by using symbol keys, GC on them is essential).

But the existence of `Object.getOwnPropertySymbols()` goes against the philosophy of symbols being private, and I suppose also goes against the values they index becoming garbage (unless this method of access is treated as a weak key of sorts), so it seems like the ES6 designers had mixed feelings as to what they were trying to achieve. (Also, the name "symbols" is very confusing, and "key" might have been better. Or, even better, "symbolkey"…)

June 12th, 2015 at 03:14

> ### Samuel Reed
>
> My guess is that symbols are mostly going to be used for the purposes described in the last few paragraphs; now we have a non-hacky way to add new prototype features and expand the language without worrying about breaking old code by using string keys that might already be used by existing libraries. That alone is worth it.
>
> I don't think there is a GC problem here. If you attach something to an object, it is attached until the object goes out of scope, just like it would with any string key, because a string is a value, just like a Symbol is a value. It can't go "out of scope", and you can't lose all references to it, since it can still be found through enumeration.
>
> Is there a way to add a symbol property to an object but make it non-enumerable? Probably, using Object.defineProperty – you could attach data and make it actually hidden.
>
> June 12th, 2015 at 06:35

Jason Orendorff AUTHOR

David: You're right that GC can be a concern. Since symbol-keyed properties are exactly like normal properties in every way, they live as long as the object they're attached to. If your code attaches many properties to long-lived objects—with or without symbols—memory usage is something to keep an eye on.

Properties are like event listeners in this regard.

Weak references are not in ES6, but there is WeakMap, and it's very relevant to this use case. I'll be writing about it next week.

Anyway, as long as the number of symbols you create is small, this is unlikely to be a problem. See http://codepen.io/anon/pen/oXwQOq for an example. The amount of memory this uses is nothing extravagant, and if an element got GC'd, its symbol-keyed properties would be GC'd too.

June 12th, 2015 at 09:30

tgc

Very interesting article!

One question: isn't setting properties on DOM elements frowned upon? I see that symbols solve the name collision problem of string keys but IIRC host objects shouldn't be modified because their behavior is not defined in the spec.

June 12th, 2015 at 03:35

Flimm

I understood the rationale for symbols until I came across the symbol registry. If the whole purpose of symbols is to avoid picking the same string name that another piece of code might pick now or in the future, then surely this registry undermines it? The keys of the registry are strings, with potential for conflict.

June 12th, 2015 at 05:16

Samuel Reed

I think the only difference now is that the symbol registry is wide open, while so many string keys are already reserved on so many types of objects.

It appears the registry is "runtime-wide": as in, on any page, calling Symbol.for("cat") will always return the same symbol.

I agree with your reaction. It appears the spec writers had a moment of weakness where they had a truly unique object, then at the last moment decided to make them more like strings. Perhaps Jason could comment?

June 12th, 2015 at 06:38

William

I had this same question, and found the answer (at least according to current Firefox behaviour).

Try this out:
let a = Symbol('cat')
let b = Symbol('cat')
let c = Symbol.for('cat')
let d = Symbol.for('cat')
console.log(a === b) // false
console.log(a === c) // false
console.log(c === d) // true

Now, if your modules want to share a symbol that someone else's modules also want to share, then I think you have a problem. But if you are just using Symbol('name') then it looks like you'll be ok.

June 12th, 2015 at 09:07

CD

That's a valid point, but something I missed in the first reading of the article is that Symbols are only added to the registry if you explicitly use the "`Symbol.for(string)"` syntax.

So, that method of Symbol creation should be used sparingly, and only in situations where you want to share the Symbol externally, and are aware of the potential for conflicts.

I made a CodePen that illustrates this:
http://codepen.io/chrisdeely/pen/VLWqEO?editors=001

June 12th, 2015 at 10:51

Jason Orendorff AUTHOR

I think the design intent here is that sometimes a program will want the other qualities of symbols — in particular, the way symbol-keyed properties are not visited by for-in, and are pretty hard to access by accident — but will need to be able to access that property across iframes, or across libraries within a single window.

When that's what you need, and you aren't super concerned about uniqueness, you can use Symbol.for().

June 12th, 2015 at 15:12

realityking

Ignoring the core language hooks, in my own code, what's the advantag of adding some value to an object using a Symbol vs. using a Map or WeakMap. This would address tgc's concern about garbage collection and prevent adding new properties to objects which seems to make the JIT compilers cry.

June 12th, 2015 at 05:17

javascript coder

The introduction of Symbols into the Javascript language could have been avoided if only there was a method on Object returning a unique hash for every instance – an incrementing 64 bit value would suffice. Trying to establish object identity in javascript is pretty tedious and inefficient – you basically have to search all elements and compare them with the === operator. See Crockfords's decycle function https://github.com/douglascrockford/JSON-js/blob/master/cycle.js#L69 for an example. The addition of Symbol to ECMAScript does not help this situation.

June 12th, 2015 at 07:45

Kyle Simpson

I think the main use-case advanced here, which is to tag an object you don't own with extra meta data, is an anti-pattern.

ES6 has a feature almost entirely for this exact purpose: WeakMap. Use the third-party-object as the key in your own WM and associate it with whatever kind of data you need. O(1) lookup. Better memory management.

Mutating an object you don't own, even with an unguessable, uncollidable key, is bad practice. Use the right tool for the right job.

All that having been said, I think symbols are nice for annotating your *own* objects with metadata, especially in making it more resistant to collision or external meta programming. I'm glad this article points out the feature itself, I just wish a different use-case had been shown.

June 12th, 2015 at 09:50

> **Jason Orendorff** AUTHOR
>
> I'll be writing about WeakMaps next week, and I'll discuss the tradeoffs.
>
> It's a little hard for me to know right now what the best practice will be on these things tomorrow. Practice takes a while to converge, and there is a "good enough" threshold that it's hard for me to make predictions about.
>
> I can certainly agree that WeakMap does a great job of keeping separate things separate, and if that's what you want, it's the way to go. Plus, if an object is frozen, mutating it won't even work. So there's that.
>
> The performance picture is less clear. Lookup should be O(1) expected in either case. It's not obvious to me that either data structure should be faster than the other (though if I had to guess, I think caching favors property access). WeakMaps, unfortunately, are bad for GC performance. But we're working on it.
>
> June 12th, 2015 at 14:58

Andrea Giammarchi

The Symbol only polyfills that works also in older mobile browsers: https://github.com/WebReflection/get-own-property-symbols

Read also Caveat section because it covers also parts of the core.js shim

For instance, you cannot use Symbols with null objects. Mot even with Babel and core.js because these will fail.

typeof doesn't work in core.js, neither in my polyfill ... **etc etc**

But hey, these works like a charm.

June 12th, 2015 at 10:20

MrD

@Flimm, I just tested this, seems like the registry only works if you use Symbols.for(), so if you don't want conflicts, just use Symbol():

```
var a = Symbol("foo");
var b = Symbol.for("foo");
var c = Symbol("foo");
var d = Symbol.for("foo");
a == b; // false
a == c; // false
a == d; // false
b == d; // true
b === d; // true
```

June 12th, 2015 at 12:14

David Bonnet

Thank you for the introduction. What's with the "@@" shorthand for built-in symbols such as `@@iterator`for `Symbol.iterator`? Could you tell us more about that?

June 15th, 2015 at 04:40

> **Jason Orendorff** AUTHOR
>
> Sure! @@iterator is not something you can write in JS code. It's a sort of technical notation used inside the ES6 language specification. It means "the symbol that's the initial value of Symbol.iterator".
>
> The ES6 spec always refers to Symbol.iterator as "@@iterator", for precision. A script can `delete Symbol` or otherwise tamper with the globals, but @@iterator never changes.
>
> http://people.mozilla.org/~jorendorff/es6-draft.html#sec-well-known-symbols
>
> Similarly, the ES6 spec also says "%ArrayPrototype%" in places where ES5 would have said "the object that is the initial value of Array.prototype". Several objects get this treatment:

> http://people.mozilla.org/~jorendorff/es6-draft.html#sec-well-known-intrinsic-objects
>
> Even in discussions with other JS engine hackers, I usually avoid this notation and just write “Symbol.iterator” or “Array.prototype”. But of course you’ll need to know the notation if you’re going to look things up in the spec.
>
> June 16th, 2015 at 05:28
>
> > **David Bonnet**
> >
> > Ah, so it’s spec-specific… Thank you for clarifying this!
> >
> > June 16th, 2015 at 05:40

Brett Zamir

Your long property name made me think of this Far Side comic: http://i674.photobucket.com/albums/vv101/Konradius5/Gary%20Larson%20Comics/InsaneBranding.jpg

June 15th, 2015 at 16:32

Yanis

Thank you for great article. I’ve just updated my very old question on stackoverflow to include this article link: http://stackoverflow.com/questions/21724326/why-bring-symbols-to-javascript

June 16th, 2015 at 01:27

bent

null is of type object

June 30th, 2015 at 12:46

Michael

It seems like there should be a Symbol.for(obj,name) since I can see the usefulness in wanting to share symbols within a certain scope without being global.

July 1st, 2015 at 12:11

Max Battcher

@Michael, In cases where you want a “scoped” Symbol your current best bet is to rely on the module system and export your symbol in some lucky module and import it anytime you need that symbol. Luckily, ES6 also comes with a module system.

```
// a.js
export var mySymbol = Symbol('mySymbol');
```

// b.js
import { mySymbol } from ‘a’;

July 6th, 2015 at 11:20

roland

Just to be sure, that I don’t miss a point in your article:

> var abc=Symbol(‘123’);
abc
abc
two different ways although both engines return
>abc.toString()+”1"
<"Symbol(123)1"

so not really the way you describe above reg. conversion into a String.
Especially since the "123" is more or less a Key-value….

Last note: I've tested as well with node.js 0.12.6 and the results are similar to Chrome43.0!

July 9th, 2015 at 09:11

> **Jason Orendorff** AUTHOR
>
> A new symbol is created each time you call Symbol(). But if you only call it once, you’ll only create one symbol:
>
> > var abc = Symbol(“123”);
> > abc === abc
> true
>
> July 9th, 2015 at 10:25

Michael

It’d be useful if you could read your description string back out of your symbol – for easier use in formatting messages etc. Right now you pretty much have to do a toString() and remove the Symbol() portion which is needlessly messy.

July 10th, 2015 at 13:06

Comments are closed for this article.