ES6 in Depth

The following 25 articles diving into ES6 (aka ES 2015) were written by the Argentinian JavaScript consultant Nicolas Bevacqua and published at his Pony Foo blog from late August through October, 2015.

Click to jump directly to the article.

A Brief History of ES6 Tooling

I wrote a few articles about React and ES6 these last few days, and today I wanted to add a bit more of context as to why I seem to be super into ES6 all of sudden. I've had an interest in ES6 for a long time, but we weren't always prepared to write code in ES6. In this post I wanted to briefly touch on the history of ES6 tooling and why I believe that today we're in a much better place to adopt ES6 than where we were half a year ago.

For the most part, we have Browserify, Babel and the spec being finalized to thank for. But we didn't always have these tools and they weren't born as mature as they are today.

Before **JavaScript-to-JavaScript** transpilers became a *(serious)* thing, there were modules that would add specific bits of ES6 functionality to your apps. There were things like `gnode`, which allows you to use generators in `node` by interpreting your code during runtime *(or turning on the Harmony flag for generators in `node >= 0.11.x`)*

> *Trivia question: how many different names has ES6 accrued over the years?*

Then we also started to see libraries that implemented ES6 module loading, such as `es6-module-loader`. These libraries helped advance the spec by giving developers something to chew on as implementations started cropping up. Of course, you could always use CoffeeScript or TypeScript back then which had implemented language features equivalent to those in ES6.

I didn't care for the syntax in CoffeeScript, nor the fact that it would've effectively reduced my ability to contribute to open-source, so that one was out. TypeScript would've been okay but it has many features on top of what's coming with ES6, and *where possible* I try to learn things that will be **useful to me for a long time**. That being said, both of these languages contributed to the shaping of ES6, so we have them to thank for that. There's also the fact that for a long time, they were as close as you could get to trying a language with anything resembling the features in ES6.

Eventually, transpilers made an appearance. The first one was Traceur, and it came out around a time where the spec wasn't locked down yet. It was constantly changing so it wasn't a very good idea to try and use it for more than a few minutes to toy around with the syntax. I got frustrated very quickly while writing example code for my application design book. Around the same time, 6to5 started making waves and there was also `esnext`, but `esnext` never implemented ES6 modules. Earlier this year those projects merged into what we know as Babel today.

> *Come June, the spec was finalized.*

Locking down the language features was crucial for adoption. It meant that compilers could now finally implement something and *not have stale syntax* within the next month. The spec being finalized and Babel becoming the de-facto *JavaScript-to-JavaScript* build tool got me interested in ES6 once again, so I started experimenting with them again.

We now have the ability to mix Browserify and Babel using `babelify`. We can use `babel-node` on the server during development – *and compile to ES5 for production because performance reasons.* We can use Webpack if we're into CSS Modules, and there's a bunch of ES6 features ready for us to use. *We need to be careful not to overplay our hand, though.* With this much going on, it's going to be hard trying to keep up while maintaining a high quality codebase that doesn't get every single new feature and shiny toy crammed into it just because we can.

> There's plenty of room in front-end tooling for feature creep, unfortunately, but **we need to battle against that** now.

> Tomorrow I'll be publishing an article about the parts of the future of JavaScript I'm most excited about and the concerns I have

about mindlessly adopting ES6 features.

↑ Back to top

ES6 JavaScript Destructuring in Depth

I've briefly mentioned a few ES6 features *(and how to get started with Babel)* in the React article series I've been writing about, and now I want to **focus on the language features** themselves. I've read a *ton* about ES6 and ES7 and it's about time we started discussing ES6 and ES7 features here in Pony Foo.

This article **warns about going overboard** with ES6 language features. Then we'll start off the series by discussing about Destructuring in ES6, and when it's most useful, as well as some of its gotchas and caveats.

A word of caution

When uncertain, chances are **you probably should default to ES5 and older syntax instead of adopting ES6 just because you can**. By this I don't mean that using ES6 syntax is a bad idea – quite the opposite, see I'm writing an article about ES6! My concern lies with the fact that when we adopt ES6 features we must do it because **they'll absolutely improve our code quality**, and not just because of the *"cool factor"* – whatever that may be.

The approach I've been taking thus far is to write things in *plain ES5*, and then adding ES6 sugar on top where it'd genuinely improve my code. I presume over time I'll be able to more quickly identify scenarios where a ES6 feature may be worth using over ES5, but when getting started it might be a good idea *not* to go overboard too soon. Instead, carefully analyze what would fit your code best first, and **be mindful of adopting ES6**.

> This way, you'll **learn to use** the new features in your favor, rather than just *learning the syntax*.

Onto the cool stuff now!

Destructuring

This is easily one of the features I've been using the most. It's also one of the simplest. It binds properties to as many variables as you need and it works with both Arrays and Objects.

```
var foo = { bar: 'pony', baz: 3 }
var { bar, baz } = foo
console.log(bar)
// <- 'pony'
console.log(baz)
// <- 3
```

It makes it very quick to pull out a specific property from an object. You're also allowed to map properties into aliases as well.

```
var foo = { bar: 'pony', baz: 3 }
var { bar: a, baz: b } = foo
console.log(a)
// <- 'pony'
console.log(b)
// <- 3
```

You can also pull properties as deep as you want, and you could also alias those deep bindings.

```
var foo = { bar: { deep: 'pony', dangerouslySetInnerHTML: 'lol' } }
var {bar: { deep, dangerouslySetInnerHTML: sure }} = foo
console.log(deep)
// <- 'pony'
console.log(sure)
// <- 'lol'
```

By default, properties that aren't found will be `undefined`, just like when accessing properties on an object with the dot or bracket notation.

```
var {foo} = {bar: 'baz'}
console.log(foo)
// <- undefined
```

If you're trying to access a deeply nested property of a parent that doesn't exist, then you'll get an exception, though.

```
var {foo:{bar}} = {baz: 'ouch'}
// <- Exception
```

That makes a lot of sense, if you think of destructuring as sugar for ES5 like the code below.

```
var _temp = { baz: 'ouch' }
var bar = _temp.foo.bar
// <- Exception
```

A cool property of destructuring is that it allows you to swap variables without the need for the infamous `aux` variable.

```
function es5 () {
  var left = 10
  var right = 20
  var aux
  if (right > left) {
    aux = right
    right = left
    left = aux
  }
}

function es6 () {
  var left = 10
  var right = 20
  if (right > left) {
    [left, right] = [right, left]
  }
}
```

Another convenient aspect of destructuring is the ability to pull keys using computed property names.

```
var key = 'such_dynamic'
var { [key]: foo } = { such_dynamic: 'bar' }
console.log(foo)
// <- 'bar'
```

In ES5, that'd take an extra statement and variable allocation on your behalf.

```
var key = 'such_dynamic'
var baz = { such_dynamic: 'bar' }
var foo = baz[key]
console.log(foo)
// <- 'bar'
```

You can also define default values, for the case where the pulled property evaluates to `undefined`.

```
var {foo=3} = { foo: 2 }
console.log(foo)
// <- 2
var {foo=3} = { foo: undefined }
console.log(foo)
// <- 3
var {foo=3} = { bar: 2 }
console.log(foo)
// <- 3
```

Destructuring works for Arrays as well, as we mentioned earlier. Note how I'm **using square brackets** in the destructuring side of the declaration now.

```
var [a] = [10]
console.log(a)
// <- 10
```

Here, again, we can use the default values and follow the same rules.

```
var [a] = []
console.log(a)
// <- undefined
var [b=10] = [undefined]
console.log(b)
// <- 10
var [c=10] = []
console.log(c)
// <- 10
```

When it comes to Arrays you can conveniently skip over elements that you don't care about.

```
var [,,a,b] = [1,2,3,4,5]
console.log(a)
// <- 3
console.log(b)
// <- 4
```

You can also use destructuring in a `function`'s parameter list.

```
function greet ({ age, name:greeting='she' }) {
  console.log(`${greeting} is ${age} years old.`)
}
greet({ name: 'nico', age: 27 })
// <- 'nico is 27 years old'
greet({ age: 24 })
// <- 'she is 24 years old'
```

That's roughly **how** you can use destructuring. What is destructuring **good** for?

Use Cases for Destructuring

There are many situations where destructuring comes in handy. Here's some of the most common ones. Whenever you have a method that returns an object, destructuring makes it much terser to interact with.

```
function getCoords () {
  return {
    x: 10,
    y: 22
  }
}
var {x, y} = getCoords()
console.log(x)
// <- 10
console.log(y)
// <- 22
```

A similar use case but in the opposite direction is being able to define default options when you have a method with a bunch of options that need default values. This is particularly interesting as an alternative to named parameters in other languages like Python and C#.

```
function random ({ min=1, max=300 }) {
  return Math.floor(Math.random() * (max - min)) + min
}
console.log(random({}))
// <- 174
console.log(random({max24}))
// <- 18
```

If you wanted to make the options object *entirely optional* you could change the syntax to the following.

```
function random ({ min=1, max=300 } = {}) {
  return Math.floor(Math.random() * (max - min)) + min
}
console.log(random())
// <- 133
```

A great fit for destructuring are things like regular expressions, where you would just love to name parameters without having to resort to index numbers. Here's an example parsing a URL with a random `RegExp` I got on StackOverflow.

```
function getUrlParts (url) {
  var magic = /^(https?):\/\/(ponyfoo\.com)(\/articles\/([a-z0-9-]+))$/
  return magic.exec(url)
```

```
}
var parts = getUrlParts('http://ponyfoo.com/articles/es6-destructuring-in-depth')
var [,protocol,host,pathname,slug] = parts
console.log(protocol)
// <- 'http'
console.log(host)
// <- 'ponyfoo.com'
console.log(pathname)
// <- '/articles/es6-destructuring-in-depth'
console.log(slug)
// <- 'es6-destructuring-in-depth'
```

Special Case: `import` Statements

Even though `import` statements don't follow destructuring rules, they behave a bit similarly. This is probably the *"destructuring-like"* use case I find myself using the most, even though it's not actually destructuring. Whenever you're writing module `import` statements, you can pull just what you need from a module's public API. An example using `contra`:

```
import {series, concurrent, map } from 'contra'
series(tasks, done)
concurrent(tasks, done)
map(items, mapper, done)
```

Note that, however, `import` statements have a different syntax. When compared against destructuring, none of the following `import` statements will work.

- Use defaults values such as `import {series = noop} from 'contra'`
- "Deep" destructuring style like `import {map: { series }} from 'contra'`
- Aliasing syntax `import {map: mapAsync} from 'contra'`

The main reason for these limitations is that the `import` statement brings in a *binding*, and not a reference or a value. This is an important differentiation that we'll explore more in depth in a future article about ES6 modules.

> I'll keep posting about ES6 & ES7 features every day, so make sure to subscribe if you want to know more!

* How about we visit string interpolation tomorrow?

** We'll leave arrow functions for Monday!

🠕 **Back to top**

ES6 Template Literals in Depth

Yesterday we covered ES6 destructuring in depth, as well as some of its most common use cases. Today we'll be moving to **template literals**. What they are, and how we can use them and what good they're for.

Template literals are a new feature in ES6 to make working with strings and string templates easier. You wrap your text in `` `backticks` `` and you'll get the features described below.

- You can interpolate variables in them
- You can actually interpolate using *any kind of expression*, not just variables
- They can be **multi-line**. *Finally!*
- You can construct *raw templates* that don't interpret backslashes

In addition, you can also define *a method* that will decide what to make of the template, instead of using the default templating behavior. There are some interesting use cases for this one.

> Let's dig into template literals and see what we can come up with.

Using Template Literals

We've already covered the basic `` `I'm just a string` ``. One aspect of template literals that may be worth mentioning is that you're now able to declare strings with both `'` and `"` quotation marks in them without having to escape anything.

```
var text = `I'm "amazed" that we have so many quotation marks to choose from!`
```

That was neat, but surely there's more useful stuff we can apply template literals to. How about some *actual interpolation*? You can use the `${expression}` notation for that.

```
var host = 'ponyfoo.com'
var text = `this blog lives at ${host}`
console.log(text)
// <- 'this blog lives at ponyfoo.com'
```

I've already mentioned you can have any kind of expressions you want in there. Think of whatever expressions you put in there as defining a variable before the template runs, and then concatenating that value with the rest of the string. That means that variables you use, methods you call, and so on, should all be available to the current scope.

The following expressions would all work just as well. It'll be up to us to decide how much logic we cram into the interpolation expressions.

```
var text = `this blog lives at ${'ponyfoo.com'}`
console.log(text)
// <- 'this blog lives at ponyfoo.com'

var today = new Date()
var text = `the time and date is ${today.toLocaleString()}`
console.log(text)
// <- 'the time and date is 8/26/2015, 3:15:20 PM'

import moment from 'moment'
var today = new Date()
var text = `today is the ${moment(today).format('Do [of] MMMM')}`
console.log(text)
// <- 'today is the 26th of August'

var text = `such ${Infinity/0}, very uncertain`
console.log(text)
// <- 'such Infinity, very uncertain'
```

Multi-line strings mean that you no longer have to use methods like these anymore.

```
var text = (
  'foo\n' +
  'bar\n' +
  'baz'
)

var text = [
  'foo',
  'bar',
  'baz'
].join('\n')
```

Instead, you can now just use backticks! Note that spacing matters, so you might still want to use parenthesis in order to keep the first line of text away from the variable declaration.

```
var text = (
`foo
bar
baz`)
```

Multi-line strings really shine when you have, *for instance*, a chunk of HTML you want to interpolate some variables to. Much like with JSX, you're perfectly able to use an expression to iterate over a collection and `return` yet another template literal to declare list items. This makes it a breeze to declare sub-components in your templates. Note also how I'm using destructuring to avoid having to prefix every expression of mine with `article.`, I like to think of it as *"a `with` block, but not as insane"*.

```
var article = {
  title: 'Hello Template Literals',
  teaser: 'String interpolation is awesome. Here are some features',
  body: 'Lots and lots of sanitized HTML',
  tags: ['es6', 'template-literals', 'es6-in-depth']
}
var {title,teaser,body,tags} = article
```

```
var html = `<article>
  <header>
    <h1>${title}</h1>
  </header>
  <section>
    <div>${teaser}</div>
    <div>${body}</div>
  </section>
  <footer>
    <ul>
      ${tags.map(tag => `<li>${tag}</li>`).join('\n      ')}
    </ul>
  </footer>
</article>`
```

The above will produce output as shown below. Note how the spacing trick was enough to properly indent the `<li>` tags.

```
<article>
  <header>
    <h1>Hello Template Literals</h1>
  </header>
  <section>
    <div>String interpolation is awesome. Here are some features</div>
    <div>Lots and lots of sanitized HTML</div>
  </section>
  <footer>
    <ul>
      <li>es6</li>
      <li>template-literals</li>
      <li>es6-in-depth</li>
    </ul>
  </footer>
</article>
```

Raw templates are the same in essence, you just have to prepend your template literal with `String.raw`. This can be very convenient in some use cases.

```
var text = String.raw`The "\n" newline won't result in a new line.
It'll be escaped.`
console.log(text)
// The "\n" newline won't result in a new line.
// It'll be escaped.
```

You might've noticed that `String.raw` seems to be a special part of the template literal syntax, and you'd be right! The method you choose will be used to parse the template. Template literal methods – called *"tagged templates"* – receive an array containing a list of the static parts of the template, as well as each expression on their own variables.

For instance a template literal like `` `hello ${name}. I am ${emotion}!` `` will pass arguments to the *"tagged template"* in a function call like the one below.

```
fn(['hello ', '. I am', '!'], 'nico', 'confused')
```

You might be confused by the seeming oddity in which the arguments are laid out, but they start to make sense when you think of it this way: for every item in the template array, there's an expression result after it.

Demystifying Tagged Templates

I wrote an example `normal` method below, and it works *exactly like the default behavior*. This might help you better understand what happens under the hood for template literals.

> If you don't know what `.reduce` does, refer to MDN or my "Fun with Native Arrays" article. Reduce is always useful when you're trying to map a collection of values into a single value that can be computed from the collection.

In this case we can reduce the `template` starting from `template[0]` and then reducing all other parts by adding the preceding `expression` and the subsequent `part`.

```
function normal (template, ...expressions) {
  return template.reduce((accumulator, part, i) => {
    return accumulator + expressions[i - 1] + part
  })
```

```
}
```

The `...expressions` syntax is new in ES6 as well. It's called the *"rest parameters syntax"*, and it'll basically place all the arguments passed to `normal` that come after `template` into a single array. You can try the tagged template as seen below, and you'll notice you get the same output as if you omitted `normal`.

```
var name = 'nico'
var outfit = 'leather jacket'
var text = normal`hello ${name}, you look lovely today in that ${outfit}`
console.log(text)
// <- 'hello nico, you look lovely today in that leather jacket'
```

Now that we've figured out how tagged templates work, what can we do with them? Well, whatever we want. One possible use case might be to make user input uppercase, turning our greeting into something that sounds more satirical — *I read the result out loud in my head with Gob's voice from Arrested Development, now I'm laughing alone. I've made a huge mistake.*

```
function upperExpr (template, ...expressions) {
  return template.reduce((accumulator, part, i) => {
    return accumulator + expressions[i - 1].toUpperCase() + part
  })
}
var name = 'nico'
var outfit = 'leather jacket'
var text = upperExpr`hello ${name}, you look lovely today in that ${outfit}`
console.log(text)
// <- 'hello NICO, you look lovely today in that LEATHER JACKET'
```

There are obviously much more useful use cases for tagged templates than laughing at yourself. In fact, you could go crazy with tagged templates. A decidedly useful use case would be to sanitize user input in your templates automatically. Given a template where all expressions are considered user-input, we could use `insane` to sanitize them out of HTML tags we dislike.

```
import insane from 'insane'
function sanitize (template, ...expressions) {
  return template.reduce((accumulator, part, i) => {
    return accumulator + insane(expressions[i - 1]) + part
  })
}
var comment = 'haha xss is so easy <iframe src="http://evil.corp"></iframe>'
var html = sanitize`<div>${comment}</div>`
console.log(html)
// <- '<div>haha xss is so easy </div>'
```

Not so easy now!

> I can definitely see a future where the only strings I use in JavaScript begin and finish with a backtick.

↑ **Back to top**

ES6 Arrow Functions in Depth

The daily saga of es6-in-depth articles continues. Today we'll be discussing Arrow Functions. In previous articles we've covered destructuring and template literals. I strive to cover *all the things* when it comes to the ES6 feature-set — and eventually we'll move onto ES7. I find that writing about these features makes it way easier for them to become **engraved in my skull** as well.

Since you're reading these articles, I suggest you set up Babel and `babel-node`, and follow along by copying the self-contained examples into a file. You can then run them using `babel-node yourfile` in the terminal. Running these examples on your own and maybe tweaking them a little bit **will help you better internalize these new features** — even if you're just adding `console.log` statements to figure out what's going on.

> Now onto the topic of the day.

We've already gone over *arrow functions* a little in previous articles, using them in passing without a lot of explaining going on. This article will focus mainly in arrow functions and keep the rest of ES6 in the back burner. I think that's the best way to write about ES6 — making a single feature “stand out” in each article, and gradually adding the others and interconnecting the different concepts so that we can understand *how they interact together*. I’ve observed **a lot of synergy** in ES6 features, which is *awesome*. It’s still important to make a gradual dive into ES6 syntax and features and not jump into the water as it’s warming up, because otherwise you’ll have a bad time adjusting to the new temperature – that was probably a bad analogy, moving on.

Using Arrow Functions in JavaScript

Arrow functions are available to many other modern languages and was one of the features I sorely missed a few years ago when I moved from C# to JavaScript. Fortunately, they’re now part of ES6 and thus available to us in JavaScript. The syntax is quite expressive. We already had anonymous functions, but sometimes it’s nice to have a terse alternative.

Here’s what the syntax looks like if we have a single argument and just want to return the results for an expression.

```
[1, 2, 3].map(num => num * 2)
// <- [2, 4, 6]
```

The ES5 equivalent would be as below.

```
[1, 2, 3].map(function (num) { return num * 2 })
// <- [2, 4, 6]
```

If we need to declare more arguments *(or no arguments)*, we’ll have to use parenthesis.

```
[1, 2, 3, 4].map((num, index) => num * 2 + index)
// <- [2, 5, 8, 11]
```

You might want to have some other statements and not just an expression to return. In this case you’ll have to use bracket notation.

```
[1, 2, 3, 4].map(num => {
  var multiplier = 2 + num
  return num * multiplier
})
// <- [3, 8, 15, 24]
```

You could also add more arguments with the parenthesis syntax here.

```
[1, 2, 3, 4].map((num, index) => {
  var multiplier = 2 + index
  return num * multiplier
})
// <- [2, 6, 12, 20]
```

At that point, however, chances are you’d be better off using a named function declaration for a number of reasons.

- `(num, index) =>` is only marginally shorter than `function (num, index)`
- The `function` form allows you to name the method, improving code quality
- When a function has multiple arguments and multiple statements, I’d say it’s improbable that six extra characters will make a difference
- However, naming the method might add just enough context into the equation that those six extra characters (plus method name) become really worthwhile

Moving on, if we need to return an object literal, we’ll have to wrap the expression in parenthesis. That way the object literal won’t be interpreted as a statement block (which would result in a silent error or worse, a **syntax error** because `number: n` isn’t a valid expression in the example below. The first example interprets `number` as a label and then figures out we have an `n` expression. Since we’re in a block and not returning anything, the mapped values will be `undefined`. In the second case, after the label and the `n` expression, `, something: 'else'` makes no sense to the compiler, and a `SyntaxError` is thrown.

```
[1, 2, 3].map(n => { number: n })
// [undefined, undefined, undefined]
[1, 2, 3].map(n => { number: n, something: 'else' })
// <- SyntaxError

[1, 2, 3].map(n => ({ number: n }))
// <- [{ number: 1 }, { number: 2 }, { number: 3 }]
[1, 2, 3].map(n => ({ number: n, something: 'else' }))
/* <- [
  { number: 1, something: 'else' },
```

```
  { number: 2, something: 'else' },
  { number: 3, something: 'else' }]
*/
```

A cool aspect of arrow functions in ES6 is that they're bound to their lexical scope. That means that you can say goodbye to `var self = this` and similar hacks – *such as using `.bind(this)`* – to preserve the context from within deeply nested methods.

```
function Timer () {
  this.seconds = 0
  setInterval(() => this.seconds++, 1000)
}
var timer = new Timer()
setTimeout(() => console.log(timer.seconds), 3100)
// <- 3
```

Keep in mind that the lexical `this` binding in ES6 arrow functions means that `.call` and `.apply` won't be able to change the context. Usually however, that's more of a feature than a bug.

Conclusions

Arrow functions are neat when it comes to defining anonymous functions that should probably be *lexically bound anyways*, and they can definitely make your code more terse in some situations.

There's no reason why you should be turning all of your function declarations into arrow functions unless their arguments and expression body are descriptive enough. I'm a big proponent of named function declarations, because they improve readability of the codebase without the need for comments – which means I'll have *"a hard time"* adopting arrow functions in most situations.

That being said, I think arrow functions are particularly useful in most functional programming situations such as when using `.map`, `.filter`, or `.reduce` on collections. Similarly, arrow functions will be really useful in asynchronous flows since those typically have a bunch of callbacks that just do argument balancing, a situation where arrow functions really shine.

↑ **Back to top**

ES6 Spread and Butter in Depth

Welcome to yet another installment of ES6 in Depth on Pony Foo. Previous ones covered destructuring, template literals, and most recently, arrow functions. Today we'll cover a few more features coming in ES6. Those features are *rest parameters, the spread operator, and default parameters*.

We've already covered some of this when we talked about destructuring, which supports default values as a nod to the **synergy in ES6 features** I've mentioned yesterday. This article might end up being a tad shorter than the rest because there's not so much to say about these rather simple features. However, and like I've mentioned in the first article of the ES6 in Depth series, the simplest features are usually the most useful as well. Let's get on with it!

Rest parameters

You know how sometimes there's a ton of arguments and you end up having to use the `arguments` magic variable to work with them? Consider the following method that joins any arguments passed to it as a string.

```
function concat () {
  return Array.prototype.slice.call(arguments).join(' ')
}
var result = concat('this', 'was', 'no', 'fun')
console.log(result)
// <- 'this was no fun'
```

The rest parameters syntax enables you to pull a real `Array` out of the `function`'s arguments by adding a parameter name prefixed by `...`. Definitely simpler, the fact that it's a real `Array` is also very convenient, and I for one am glad not to have to resort to `arguments` anymore.

```
function concat (...words) {
```

```
  return words.join(' ')
}
var result = concat('this', 'is', 'okay')
console.log(result)
// <- 'this is okay'
```

When you have more parameters in your `function` it works slightly different. Whenever I declare a method that has a rest parameter, I like to think of its behavior as follows.

- Rest parameter gets all the `arguments` passed to the function call
- Each time a parameter is added on the left, it's as if its value is assigned by calling `rest.shift()`
- Note that you can't actually place parameters to the right: rest parameters can only be the last argument

It's easier to visualize how that would behave than try to put it into words, so let's do that. The method below computes the `sum` for all `arguments` except the first one, which is then used as a `multiplier` for the `sum`. In case you don't recall, `.shift()` returns the first value in an array, and also removes it from the collection, which makes it a useful mnemonic device in my opinion.

```
function sum () {
  var numbers = Array.prototype.slice.call(arguments) // numbers gets all arguments
  var multiplier = numbers.shift()
  var base = numbers.shift()
  var sum = numbers.reduce((accumulator, num) => accumulator + num, base)
  return multiplier * sum
}
var total = sum(2, 6, 10, 8, 9)
console.log(total)
// <- 66
```

Here's how that method would look if we were to use the rest parameter to pluck the numbers. Note how we don't need to use `arguments` nor do any shifting anymore. This is great because it vastly reduces the complexity in our method – which now can focus on its functionality itself and not so much on rebalancing `arguments`.

```
function sum (multiplier, base, ...numbers) {
  var sum = numbers.reduce((accumulator, num) => accumulator + num, base)
  return multiplier * sum
}
var total = sum(2, 6, 10, 8, 9)
console.log(total)
// <- 66
```

Spread Operator

Typically you invoke a function by passing arguments into it.

```
console.log(1, 2, 3)
// <- '1 2 3'
```

Sometimes however you have those arguments in a list and just don't want to access every index just for a method call – *or you just can't because the array is formed dynamically* – so you use `.apply`. This feels kind of awkward because `.apply` also takes a context for `this`, which feels out of place when it's not relevant and you have to reiterate the host object *(or use `null`)*.

```
console.log.apply(console, [1, 2, 3])
// <- '1 2 3'
```

The spread operator can be used as *a butter knife* alternative over using `.apply`. There is no need for a context either. You just append three dots `...` to the array, just like with the rest parameter.

```
console.log(...[1, 2, 3])
// <- '1 2 3'
```

As we'll investigate more in-depth next Monday, in the article about iterators in ES6, a nice perk of the spread operator is that it can be used on anything that's an *iterable*. This encompasses even things like the results of `document.querySelectorAll('div')`.

```
[...document.querySelectorAll('div')]
// <- [<div>, <div>, <div>]
```

Another nice aspect of the *butter knife operator* is that you can **mix and match** regular arguments with it, and they'll be spread over the function call exactly how you'd expect them to. This, too, can be *very very useful* when you have a lot of argument rebalancing going on in your ES5 code.

```
console.log(1, ...[2, 3, 4], 5) // becomes `console.log(1, 2, 3, 4, 5)`
// <- '1 2 3 4 5'
```

Time for a real-world example. In Express applications, I sometimes use the method below to allow `morgan` *(the request logger in Express)* to stream its messages through `winston`, a general purpose multi-transport logger. I remove the trailing line breaks from the `message` because `winston` already takes care of those. I also place some metadata about the currently executing process like the host and the process `pid` into the arguments list, and then I `.apply` everything on the `winston` logging mechanism. If you take a close look at the code, the only line of code that's actually doing anything is the one I've highlighted; the rest is just playing around with `arguments`.

```
function createWriteStream (level) {
  return {
    write: function () {
      var bits = Array.prototype.slice.call(arguments)
      var message = bits.shift().replace(/\n+$/, '') // remove trailing breaks
      bits.unshift(message)
      bits.push({ hostname: os.hostname(), pid: process.pid })
      winston[level].apply(winston, bits)
    }
  }
}
app.use(morgan(':status :method :url', {
  stream: createWriteStream('debug')
}))
```

We can thoroughly simplify the solution with ES6. First, we can use the rest parameter instead of relying on `arguments`. The rest parameter already gives us a true array, so there's no casting involved either. We can grab the `message` directly as the first parameter, and we can then apply everything on `winston[level]` directly by combining normal arguments with the rest of the `...bits` and pieces. The code below is **in much better shape**, as now every piece of it is actually relevant to what we're trying to accomplish, which is call `winston[level]` with a few *modified arguments*. The piece of code we had earlier, in contrast, spent most time manipulating the arguments, and the focus quickly dissipated into **a battle of wits against JavaScript itself** – *the method stopped being about the code we were trying to write*.

```
function createWriteStream (level) {
  return {
    write: function (message, ...bits) {
      winston[level](message.replace(/\n+$/, ''), ...bits, {
        hostname: os.hostname(), pid: process.pid
      })
    }
  }
}
```

We could further *simplify the method by pulling* the process metadata out, since that won't change for the lifespan of the process. We could've done that in the ES5 code too, though.

```
var proc = { hostname: os.hostname(), pid: process.pid }
function createWriteStream (level) {
  return {
    write: function (message, ...bits) {
      winston[level](message.replace(/\n+$/, ''), ...bits, proc)
    }
  }
}
```

Another thing we could do to shorten that piece of code might be to use an arrow function. In this case however, it **would only complicate matters**. You'd have to shorten `message` to `msg` so that it fits in a single line, and the call to `winston[level]` with the rest and spread operators in there makes it **an incredibly complicated sight** to anyone who *hasn't* spent the last 15 minutes thinking about the method – *be it a team mate or yourself the week after you wrote this function.*

```
var proc = { hostname: os.hostname(), pid: process.pid }
function createWriteStream (level) {
  return {
    write: (msg, ...bits) => winston[level](msg.replace(/\n+$/, ''), ...bits, proc)
  }
}
```

It would be wiser to just keep our earlier version. While it's *quite self-evident* in this case that an arrow function only **piles onto the complexity**, in other cases it might not be so. It's up to you to decide, and you need to be able to distinguish between using ES6 features because they genuinely improve your codebase and its maintainability, or **whether you're actually decreasing maintainability** by translating things into ES6 just for the sake of doing so.

Some other useful uses are detailed below. You can obviously use the spread operator when creating a new array, but you can also use while destructuring, in which case it works sort of like `...rest` did, and a use case that's not going to come up often but is still worth mentioning is that you can use spread to pseudo- `.apply` when using the `new` operator as well.

Use Case	ES5	ES6
Concatenation	`[1, 2].concat(more)`	`[1, 2, ...more]`
Push onto list	`list.push.apply(list, [3, 4])`	`list.push(...[3, 4])`
Destructuring	`a = list[0], rest = list.slice(1)`	`[a, ...rest] = list`
`new` + `apply`	`new (Date.bind.apply(Date, [null,2015,31,8]))`	`new Date(...[2015,31,8])`

Default Operator

The default operator is something we've covered in the destructuring article, but only tangentially. Just like you can use default values during destructuring, you can define a default value for any parameter in a function, as shown below.

```
function sum (left=1, right=2) {
  return left + right
}
console.log(sum())
// <- 3
console.log(sum(2))
// <- 4
console.log(sum(1, 0))
// <- 1
```

Consider the code that initializes options in `dragula`.

```
function dragula (options) {
  var o = options || {};
  if (o.moves === void 0) { o.moves = always; }
  if (o.accepts === void 0) { o.accepts = always; }
  if (o.invalid === void 0) { o.invalid = invalidTarget; }
  if (o.containers === void 0) { o.containers = initialContainers || []; }
  if (o.isContainer === void 0) { o.isContainer = never; }
  if (o.copy === void 0) { o.copy = false; }
  if (o.revertOnSpill === void 0) { o.revertOnSpill = false; }
  if (o.removeOnSpill === void 0) { o.removeOnSpill = false; }
  if (o.direction === void 0) { o.direction = 'vertical'; }
  if (o.mirrorContainer === void 0) { o.mirrorContainer = body; }
}
```

> Do you think it would be useful to switch to default parameters under ES6 syntax? How would you do that?

↑ **Back to top**

ES6 Object Literal Features in Depth

Once again, this is ES6 in Depth. If you haven't set foot on this series before, you might want to learn about destructuring, template literals, arrow functions, or the spread operator and rest parameters. Today's special is *object literals in ES6*. **"Sure, I can use those today"**, you say – object literals date all the way back to ES3. This article is about new features coming in ES6 for object literals.

As I did in previous articles on the series, I would love to point out that you should probably set up Babel and follow along the examples with either a REPL or the `babel-node` CLI and a file. That'll make it so much easier for you to internalize the concepts discussed in the series. If you aren't the *"install things on my computer"* kind of human, you might prefer to hop on CodePen and then click on the gear icon for JavaScript – *they have a Babel preprocessor which makes trying out ES6 a breeze.*

Onto the new stuff!

Property Value Shorthands

Whenever you find yourself assigning a property value that matches a property name, you can omit the property value, it's implicit in ES6.

```
var foo = 'bar'
var baz = { foo }
console.log(baz.foo)
// <- 'bar'
```

In the snippet shown below I re-implemented part of `localStorage` in memory as a polyfill. It displays a pattern that I've followed countless times in my code.

```
var ms = {}

function getItem (key) {
  return key in ms ? ms[key] : null
}

function setItem (key, value) {
  ms[key] = value
}

function clear () {
  ms = {}
}

module.exports = {
  getItem: getItem,
  setItem: setItem,
  clear: clear
}
```

The reasons why – *most often* – I don't place functions directly on an object definition are *several*.

- Less indentation needed
- Public API stands out
- Harder to tightly couple methods
- Easier to reason about

With ES6, we can throw another bullet into that list, and that's that the export can be even easier using *property value shorthands*. You can omit the property value if it matches the property name. The `module.exports` from the code above thus becomes:

```
module.exports = { getItem, setItem, clear }
```

So good!

Computed Property Names

We already covered computed property names briefly in the destructuring article. This was a very common thing to do for me:

```
var foo = 'bar'
var baz = {}
baz[foo] = 'ponyfoo'
console.log(baz)
// <- { bar: 'ponyfoo' }
```

Computed property names allow you to write an *expression* wrapped in square brackets instead of the regular property name. Whatever the expression evaluates to will become the property name.

```
var foo = 'bar'
var baz = { [foo]: 'ponyfoo' }
console.log(baz)
// <- { bar: 'ponyfoo' }
```

One limitation of computed property names is that you won't be able to use the shorthand expression with it. I presume this is because shorthand expression is meant to be simple, compile-time sugar.

```
var foo = 'bar'
var bar = 'ponyfoo'
var baz = { [foo] }
console.log(baz)
// <- SyntaxError
```

That being said, I believe this to be the most common use case. Here our code is simpler because we don't have to spend three steps in allocating a `foo` variable, assigning to `foo[type]`, and returning `foo`. Instead we can do all three in a single statement.

```
function getModel (type) {
  return {
    [type]: {
      message: 'hello, this is doge',
      date: new Date()
    }
  }
}
```

Neat. What else?

Method Definitions

Typically in ES5 you declare methods on an object like so:

```
var foo = {
  bar: function (baz) {
  }
}
```

While getters and setters have a syntax like this, where there's no need for the `function` keyword. It's just inferred from context.

```
var cart = {
  _wheels: 4,
  get wheels () {
    return this._wheels
  },
  set wheels (value) {
    if (value < this._wheels) {
      throw new Error('hey, come back here!')
    }
    this._wheels = value
  }
}
```

Starting in ES6, you can declare regular methods with a similar syntax, only difference is it's not prefixed by `get` or `set`.

```
var cart = {
  _wheels: 4,
  wheels () {
    return this._wheels
  },
  wheels (value) {
    if (value < this._wheels) {
      throw new Error('hey, come back here!')
    }
    this._wheels = value
  },
  dismantle () {
    this._wheels = 0
    console.warn(`you're all going to pay for this!`)
  }
}
```

I think it's nice that methods converged together with getters and setter. I for one don't use this syntax a lot because I like to name my functions and decouple them from their host objects as I explained in the shorthand section. However, it's still useful in some situations and definitely useful when declaring *"classes"* – if you're into that sort of thing.

↑ **Back to top**

ES6 Classes in Depth

Welcome to ES6 in Depth. Are you new here? You might want to learn about destructuring, template literals, arrow functions, the spread operator and rest parameters, or object literal features in ES6. Today is going to be about *"classes"* in ES6.

> As I did in previous articles on the series, I would love to point out that you should probably set up Babel and follow along the examples with either a REPL or the `babel-node` CLI and a file. That'll make it so much easier for you to **internalize the concepts** discussed in the series. If you aren't the *"install things on my computer"* kind of human, you might prefer to hop on CodePen and then click on the gear icon for JavaScript – *they have a Babel preprocessor which makes trying out ES6 a breeze.*

Onwards!

What do you mean – classes in JavaScript?

JavaScript is a prototype-based language, so what are ES6 classes really? They're syntactic sugar on top of prototypical inheritance – a device to make the language more inviting to programmers coming from other paradigms who might not be all that familiar with prototype chains. Many features in ES6 *(such as destructuring)* are, in fact, syntactic sugar – and classes are no exception. I like to clarify this because it makes it much easier to understand the underlying technology behind ES6 classes. There is no huge restructuring of the language, they just made it easier for people used to classes to leverage prototypal inheritance.

> While I may dislike the term *"classes"* for this particular feature, I have to say that the syntax is in fact much easier to work with than regular prototypal inheritance syntax in ES5, and that's a win for everyone – regardless of them being called classes or not.

Now that that's out of the way, I'll assume you understand prototypal inheritance – just because you're reading a blog about JavaScript. Here's how you would describe a `Car` that can be instantiated, fueled up, and move.

```
function Car () {
  this.fuel = 0;
  this.distance = 0;
}

Car.prototype.move = function () {
  if (this.fuel < 1) {
    throw new RangeError('Fuel tank is depleted')
  }
  this.fuel--
  this.distance += 2
}

Car.prototype.addFuel = function () {
  if (this.fuel >= 60) {
    throw new RangeError('Fuel tank is full')
  }
  this.fuel++
}
```

To move the car, you could use the following piece of code.

```
var car = new Car()
car.addFuel()
car.move()
car.move()
// <- RangeError: 'Fuel tank is depleted'
```

Neat. What about with ES6 classes? The syntax is very similar to declaring an object, except we precede it with `class Name`, where `Name` is the name for our class. Here we are leveraging the method signature notation we covered yesterday to declare the methods using a shorter syntax. The `constructor` is just like the constructor method in ES5, so you can use that to initialize any variables your instances may have.

```
class Car {
  constructor () {
    this.fuel = 0
    this.distance = 0
  }
  move () {
    if (this.fuel < 1) {
      throw new RangeError('Fuel tank is depleted')
    }
    this.fuel--
    this.distance += 2
  }
  addFuel () {
```

```
    if (this.fuel >= 60) {
      throw new RangeError('Fuel tank is full')
    }
    this.fuel++
  }
}
```

In case you haven't noticed, and for some obscure reason that escapes me, **commas are invalid** in-between properties or methods in a class, as opposed to object literals where commas are *(still)* mandatory. That discrepancy is bound to cause headaches to people trying to decide whether they want a plain object literal or a class instead, but the code *does* look sort of cleaner without the commas here.

Many times *"classes"* have static methods. Think of your friend the `Array` for example. Arrays have instance methods like `.filter`, `.reduce`, and `.map`. The `Array` *"class"* itself has static methods as well, like `Array.isArray`. In ES5 code, it's pretty easy to add these kind of methods to our `Car` *"class"*.

```
function Car () {
  this.topSpeed = Math.random()
}
Car.isFaster = function (left, right) {
  return left.topSpeed > right.topSpeed
}
```

In ES6 `class` notation, we can use precede our method with `static`, following a similar syntax as that of `get` and `set`. Again, just sugar on top of ES5, as it's quite trivial to transpile this down into ES5 notation.

```
class Car {
  constructor () {
    this.topSpeed = Math.random()
  }
  static isFaster (left, right) {
    return left.topSpeed > right.topSpeed
  }
}
```

One sweet aspect of ES6 `class` sugar is that you also get an `extends` keyword that enables you to easily *"inherit"* from other *"classes"*. We all know Tesla cars move further while using the same amount of fuel, thus the code below shows how `Tesla extends Car` and "overrides" (a concept you might be familiar with if you've ever played around with C#) the `move` method to cover a larger distance.

```
class Tesla extends Car {
  move () {
    super.move()
    this.distance += 4
  }
}
```

The special `super` keyword identifies the `Car` class we've inherited from – and since we're speaking about C#, it's akin to `base`. It's *raison d'être* is that most of the time we *override* a method by re-implementing it in the inheriting class, – `Tesla` in our example – we're supposed to call the method on the base class as well. This way we don't have to copy logic over to the inheriting class whenever we re-implement a method. That'd be particularly lousy since whenever a base class changes we'd have to paste their logic into every inheriting class, turning our codebase into a maintainability nightmare.

If you now did the following, you'll notice the Tesla car moves two places because of `base.move()`, which is what every regular car does as well, and it moves an additional four places because `Tesla` is just that good.

```
var car = new Tesla()
car.addFuel()
car.move()
console.log(car.distance)
// <- 6
```

The most common thing you'll have to override is the `constructor` method. Here you can just call `super()`, passing any arguments that the base class needs. Tesla cars are twice as fast, so we just call the base `Car` constructor with twice the advertised `speed`.

```
class Car {
  constructor (speed) {
    this.speed = speed
  }
}
```

```
class Tesla extends Car {
  constructor (speed) {
    super(speed * 2)
  }
}
```

Tomorrow, we'll go over the syntax for `let`, `const`, and `for ... of ...`. Until then!

Back to top

ES6 Let, Const and the 'Temporal Dead Zone' (TDZ) in Depth

This is yet another edition of ES6 in Depth. First time here? Welcome! So far we covered destructuring, template literals, arrow functions, the spread operator and rest parameters, object literal features in ES6, and last but not least: what *"classes"* really mean in ES6. Today is going to be about an assortment of simple language features coming our way in ES6 – `let`, `const`, and the scary-sounding *"Temporal Dead Zone"*.

> As I did in previous articles on the series, I would love to point out that you should probably set up Babel and follow along the examples with either a REPL or the `babel-node` CLI and a file. That'll make it so much easier for you to **internalize the concepts** discussed in the series. If you aren't the *"install things on my computer"* kind of human, you might prefer to hop on CodePen and then click on the gear icon for JavaScript – *they have a Babel preprocessor which makes trying out ES6 a breeze.* Another alternative that's also quite useful is to use Babel's online REPL – *it'll show you compiled ES5 code to the right of your ES6 code for quick comparison.*

Shall we?

Let Statement

The `let` statement is one of the most well-known features in ES6, which is partly why I grouped it together with a few other new features. It works like a `var` statement, but it has different scoping rules. JavaScript has always had a complicated ruleset when it came to scoping, driving many programmers crazy when they were first trying to figure out how variables work in JavaScript.

Eventually, you discover this thing called hoisting, and things start making a bit more sense to you. Hoisting means that variables get pulled from anywhere they were declared in user code to the top of their scope. For example, see the code below.

```
function areTheyAwesome (name) {
  if (name === 'nico') {
    var awesome = true
  }
  return awesome
}
areTheyAwesome('nico')
// <- true
areTheyAwesome('christian heilmann')
// <- undefined
```

The reason why this doesn't implode into oblivion is, as we know, that `var` is function-scoped. That coupled with hoisting means that what we're really expressing is something like the piece of code below.

```
function areTheyAwesome (name) {
  var awesome
  if (name === 'nico') {
    awesome = true
  }
  return awesome
}
```

Whether we like it or not (or we're just used to it – I know I am), this is plainly more confusing than having block-scoped variables would be. Block scoping works on the bracket level, rather than the function level.

Instead of having to declare a new `function` if we want a deeper scoping level, block scoping allows you to just leverage existing code branches like those in `if`, `for`, or `while` statements; you could also create new `{}` blocks arbitrarily. As you may or may not know, the JavaScript language allows us to create an indiscriminate number of blocks, just because we want to.

```
{{{{{var insane = 'yes, you are'}}}}}
console.log(insane)
// <- 'yes, you are'
```

With `var`, though, one could still access the variable from outside those many, many, many blocks, and not get an error. Sometimes it can be very useful to get errors in these situations. Particularly if one or more of these is true.

- Accessing the inner variable breaks some sort of encapsulation principle in our code
- The inner variable doesn't belong in the outer scope at all
- The block in question has many siblings that would also want to use the same variable name
- One of the parent blocks already has a variable with the name we need, but it's still appropriate to use in the inner block

So how does this `let` thing work?

> The `let` statement is an alternative to `var`. It follows block scoping rules instead of the default function scoping rules. This means you **don't need entire functions** to get a new scope – *a simple `{}` block will do!*

```
let outer = 'I am so eccentric!'
{
  let inner = 'I play with neighbors in my block and the sewers'
  {
    let innermost = 'I only play with neighbors in my block'
  }
  // accessing innermost here would throw
}
// accessing inner here would throw
// accessing innermost here would throw
```

Here is where things got interesting. As I wrote this example I thought, *"Well, but if we now declare a function inside a block and access it from outside that block, things will **surely go awry**"*. Based on my existing knowledge of ES5 I fully expected the following snippet of code to work, and it does in fact *work in ES5* but it's broken in ES6. That would've been a problem because it'd make super easy to expose block-scoped properties through functions that become hoisted outside of the block. I didn't expect this to `throw`.

```
{
  let _nested = 'secret'
  function nested () {
    return _nested
  }
}
console.log(nested())
// nested is not defined
```

As it turns out, this wasn't a bug in Babel, but in fact a *(much welcome)* change in ES6 language semantics.

> @nzgb @rauschma @sebmck AFAIR, this is correct – ES6 finally specified functions in blocks to behave as block-scoped.
>
> — Ingvar Stepanyan (@RReverser) August 28, 2015

Note that you can still expose nested `let` things to outer scopes simply by assigning them to a variable that has more access. I wouldn't recommend you do this however, as there probably are cleaner ways to write code in these situations – such as **not using `let` when you don't want block scoping**.

```
var nested
{
  let _nested = 'secret'
  nested = function () {
    return _nested
  }
}
console.log(nested())
// <- 'secret'
```

In conclusion, block scoping can be quite useful in new codebases. Some people will tell you to drop `var` forever and just use `let` everywhere. Some will tell you to never use `let` because that's not the *One True Way of JavaScript*. My position might change over time, but this is it – for the time being:

> I plan on using `var` most of the time, and `let` in those situations where I would've otherwise hoisted a variable to the top of the scope for no reason, when they actually belonged inside a conditional or iterator code branch.

The *Temporal Dead Zone* and the Deathly Hallows

One last thing of note about `let` is a mystical concept called the "Temporal Dead Zone" *(TDZ)* – *ooh… so scary, I know.*

In so many words: if you have code such as the following, it'll throw.

```
there = 'far away'
// <- ReferenceError: there is not defined
let there = 'dragons'
```

If your code tries to access `there` in any way before the `let there` statement is reached, the program will throw. Declaring a method that references `there` before it's defined is okay, as long as the method doesn't get executed while `there` is in the TDZ, and `there` will be in the TDZ for as long as the `let there` statement isn't reached *(while the scope has been entered)*. This snippet won't throw because `return there` isn't executed until after `there` leaves the TDZ.

```
function readThere () {
  return there
}
let there = 'dragons'
console.log(readThere())
// <- 'dragons'
```

But this snippet will, because access to `there` occurs *before leaving the TDZ for* `there`.

```
function readThere () {
  return there
}
console.log(readThere())
// ReferenceError: there is not defined
let there = 'dragons'
```

Note that the semantics for these examples doesn't change when `there` isn't actually assigned a value when initially declared. The snippet below still throws, as it still tries to access `there` before leaving the TDZ.

```
function readThere () {
  return there
}
console.log(readThere())
// ReferenceError: there is not defined
let there
```

This snippet still works because it still leaves the TDZ before accessing `there` in any way.

```
function readThere () {
  return there
}
let there
console.log(readThere())
// <- undefined
```

The only tricky part is to remember that *(when it comes to the TDZ)* functions work sort of like blackboxes until they're actually executed for the first time, so it's okay to place `there` inside functions that don't get executed until we leave the TDZ.

> The whole point of the TDZ is to make it easier to catch errors where accessing a variable before it's declared in user code leads to unexpected behavior. This happened a lot with ES5 due both to hoisting and poor coding conventions. In ES6 it's easier to avoid. Keep in mind that hoisting still applies for `let` as well — this just means that the variables will be created when we enter the scope, and the TDZ will be born, but they will be inaccessible until code execution hits the place where the variable was actually declared, at which point we leave the TDZ and are cleared to use the variable.

Const Statement

Phew. I wrote more than I ever wanted to write about `let`. Fortunately for both of us, `const` is quite similar to `let`.

- `const` is also *block-scoped*
- `const` also enjoys the marvels of *TDZ semantics*

There's also a couple of major differences.

- `const` variables must be declared using an initializer
- `const` variables can only be assigned to once, in said initializer
- `const` variables **don't** make the assigned value immutable
- Assigning to `const` will fail silently
- Redeclaration of a variable by the same name *will* throw

Let's go to some examples. First, this snippet shows how it follows block-scoping rules just like `let`.

```
const cool = 'ponyfoo'
{
  const cool = 'dragons'
  console.log(cool)
  // <- 'dragons'
}
console.log(cool)
// <- 'ponyfoo'
```

Once a `const` is declared, you can't change the reference or literal that's assigned to it.

```
const cool = { people: ['you', 'me', 'tesla', 'musk'] }
cool = {}
// <- "cool" is read-only
```

You can however, change the reference itself. It does not become immutable. You'd have to use `Object.freeze` to make the value itself immutable.

```
const cool = { people: ['you', 'me', 'tesla', 'musk'] }
cool.people.push('berners-lee')
console.log(cool)
// <- { people: ['you', 'me', 'tesla', 'musk', 'berners-lee'] }
```

You can also make other references to the `const` that *can*, in fact, change.

```
const cool = { people: ['you', 'me', 'tesla', 'musk'] }
var uncool = cool
uncool = { people: ['edison'] } // so uncool he's all alone
console.log(uncool)
// <- { people: ['edison'] }
```

I think `const` is great because it allows us to mark things that we really need to preserve as such. Imagine the following piece of code, which does come up in some situations — *sorry about the extremely contrived example.*

```
function code (groceries) {
  return {eat}
  function eat () {
    if (groceries.length === 0) {
      throw new Error('All out. Please buy more groceries to feed the code.')
    }
    return groceries.shift()
  }
}
var groceries = ['carrot', 'lemon', 'potato', 'turducken']
var eater = code(groceries)
console.log(eater.eat())
// <- 'carrot'
```

I sometimes come across code where someone is trying to add more `groceries` to the list, and they figure that doing the following would *just work*. In many cases this does work. However, if we're passing a reference to groceries to something else, the re-assignment wouldn't be carried away to that other place, and hard to debug issues would ensue.

```
// a few hundred lines of code later...
groceries = ['heart of palm', 'tomato', 'corned beef']
```

If `groceries` were a constant in the piece of code above, this re-assignment would've been far easier to detect. Yay, ES6! I can definitely see myself using `const` a lot in the future, but I haven't quite internalized it yet.

> I guess more coding is in order!

Back to top

ES6 Iterators in Depth

This is yet another edition of ES6 in Depth. First time here? Welcome! So far we covered destructuring, template literals, arrow functions, the spread operator and rest parameters, improvements coming to object literals, the new *classes* sugar on top of prototypes, and an article on `let`, `const`, and the *"Temporal Dead Zone"*. The soup of the day is: **Iterators**.

> As I did in previous articles on the series, I would love to point out that you should probably set up Babel and follow along the examples with either a REPL or the `babel-node` CLI and a file. That'll make it so much easier for you to **internalize the concepts** discussed in the series. If you aren't the *"install things on my computer"* kind of human, you might prefer to hop on CodePen and then click on the gear icon for JavaScript – *they have a Babel preprocessor which makes trying out ES6 a breeze.* Another alternative that's also quite useful is to use Babel's online REPL – *it'll show you compiled ES5 code to the right of your ES6 code for quick comparison.*

Before getting into it, let me *shamelessly ask for your support* if you're enjoying my ES6 in Depth series. Your contributions will go towards helping me keep up with the schedule, server bills, keeping me fed, and maintaining **Pony Foo** as a veritable source of JavaScript goodies.

Thanks for listening to that, and without further ado … *shall we?*

Iterator Protocol and Iterable Protocol

> *There's a lot of new, intertwined terminology here. Please bear with me as I get some of these explanations out of the way!*

JavaScript gets two new protocols in ES6, *Iterators* and *Iterables*. In plain terms, you can think of protocols as *conventions*. As long as you follow a determined convention in the language, you get a side-effect. The *iterable* protocol allows you to define the behavior when JavaScript objects are being iterated. Under the hood, deep in the world of JavaScript interpreters and language specification *keyboard-smashers*, we have the `@@iterator` method. This method underlies the *iterable* protocol and, in the real world, you can assign to it using something called "the *well-known* `Symbol.iterator` Symbol".

We'll get back to what Symbols are *later in the series*. Before losing focus, you should know that the `@@iterator` method is called **once, whenever an object needs to be iterated**. For example, at the beginning of a `for..of` loop *(which we'll also get back to in a few minutes)*, the `@@iterator` will be asked for an *iterator*. The returned *iterator* will be used to obtain values out of the object.

Let's use the snippet of code found below as a crutch to understand the concepts behind iteration. The first thing you'll notice is that I'm making my object an iterable by assigning to its mystical `@@iterator` property through the `Symbol.iterator` property. I can't use the symbol as a property name directly. Instead, I have to wrap in square brackets, meaning it's a computed property name that evaluates to the `Symbol.iterator` *expression* – as you might recall from the article on object literals. The object returned by the method assigned to the `[Symbol.iterator]` property must adhere to the *iterator* protocol. The *iterator* protocol defines how to get values out of an object, and we must return an `@@iterator` that adheres to *iterator* protocol. The protocol indicates we must have an object with a `next` method. The `next` method takes no arguments and it should return an object with these two properties.

- `done` signals that the sequence has ended when `true`, and `false` means there may be more values
- `value` is the current item in the sequence

In my example, the iterator method returns an object that has a finite list of items and which emits those items until there aren't any more left. The code below is an iterable object in ES6.

```
var foo = {
  [Symbol.iterator]: () => ({
    items: ['p', 'o', 'n', 'y', 'f', 'o', 'o'],
    next: function next () {
      return {
        done: this.items.length === 0,
        value: this.items.shift()
      }
    }
  })
}
```

To actually iterate over the object, we could use `for..of`. What would that look like? See below. The `for..of` iteration method is also new in ES6, and it settles the everlasting war against looping over JavaScript collections and randomly finding things that didn't belong in the result-set you were expecting.

```
for (let pony of foo) {
  console.log(pony)
  // <- 'p'
  // <- 'o'
  // <- 'n'
  // <- 'y'
  // <- 'f'
  // <- 'o'
  // <- 'o'
}
```

You can use `for..of` to iterate over any object that adheres to the *iterable* protocol. In ES6, that includes arrays, any objects with a user-defined `[Symbol.iterator]` method, *generators*, DOM node collections from `.querySelectorAll` and friends, etc. If you just want to *"cast"* any iterable into an array, a couple of terse alternatives would be using the spread operator and `Array.from`.

```
console.log([...foo])
// <- ['p', 'o', 'n', 'y', 'f', 'o', 'o']
console.log(Array.from(foo))
// <- ['p', 'o', 'n', 'y', 'f', 'o', 'o']
```

To recap, our `foo` object adheres to the *iterable* protocol by assigning a method to `[Symbol.iterator]` – *anywhere in the prototype chain for* `foo` *would work*. This means that the object is *iterable*: it can be iterated. Said method returns an object that adheres to the *iterator* protocol. The iterator method is called once whenever we want to start iterating over the object, and the returned *iterator* is used to pull values out of `foo`. To iterate over iterables, we can use `for..of`, the spread operator, or `Array.from`.

What Does This All Mean?

In essence, the selling point about iteration protocols, `for..of`, `Array.from`, and the spread operator is that they provide expressive ways to effortlessly iterate over collections and array-likes *(such as* `arguments`*)*. Having the ability to define how any object may be iterated is huge, because it enables libraries like lo-dash to converge under a protocol the language natively understands – *iterables*. This is **huge**.

> Lodash's chaining wrapper is now an iterator and iterable: var w = _({ a: 1, b: 2 }); Array.from(w); // => [1, 2]
>
> — John-David Dalton (@jdalton) August 31, 2015

Just to give you another example, remember how I always complain about jQuery wrapper objects not being true arrays, or how `document.querySelectorAll` doesn't return a true array either? If jQuery implemented the iterator protocol on their collection's prototype, then you could do something like below.

```
for (let item of $('li')) {
  console.log(item)
  // <- the <li> wrapped in a jQuery object
}
```

Why wrapped? Because it's more expressive. You could easily iterate as deep as you need to.

```
for (let list of $('ul')) {
  for (let item of list.find('li')) {
    console.log(item)
    // <- the <li> wrapped in a jQuery object
  }
}
```

This brings me to an important aspect of iterables and iterators.

Lazy in Nature

Iterators are *lazy in nature*. This is fancy-speak for saying that the sequence is accessed one item at a time. It can even be an infinite sequence – a legitimate scenario with many use cases. Given that iterators are lazy, having jQuery wrap every result in the sequence with their wrapper object wouldn't have a big upfront cost. Instead, a wrapper is created each time a value is pulled from the *iterator*.

How would an infinite iterator look? The example below shows an iterator with a `1..Infinity` range. Note how it will never yields `done: true`, signaling that the sequence is over. Attempting to cast the iterable `foo` object into an array using either `Array.from(foo)` or `[...foo]` would crash our program, since the sequence *never ends*. We must be very careful with these types of sequences as they can crash and burn our Node process, or the human's browser tab.

```
var foo = {
  [Symbol.iterator]: () => {
    var i = 0
    return { next: () => ({ value: ++i }) }
  }
}
```

The correct way of working with such an iterator is with an escape condition that prevents the loop from going infinite. The example below loops over our infinite sequence using `for..of`, but it breaks the loop as soon as the value goes over `10`.

```
for (let pony of foo) {
  if (pony > 10) {
    break
  }
  console.log(pony)
}
```

> The iterator doesn't really *know* that the sequence is infinite. In that regard, this is similar to the halting problem – there is no way of knowing whether the sequence is infinite or not in code.
>
> 
>
>
> We **usually have a good idea** about whether a sequence is *finite or infinite*, since we construct those sequences. Whenever we have an infinite sequence it's up to us to add an escape condition that ensures our program won't crash in an attempt to loop over every single value in the sequence.

Come back tomorrow for a discussion about generators!

Back to top

ES6 Generators in Depth

This is ES6 in Depth, the longest-running article series in the history of Pony Foo! Trapped in the ES5 bubble? Welcome! Let me get you started with destructuring, template literals, arrow functions, the spread operator and rest parameters, improvements coming to object literals, the new *classes* sugar on top of prototypes, `let`, `const`, and the *"Temporal Dead Zone"*, and Iterators.

> As I did in previous articles on the series, I would love to point out that you should probably set up Babel and follow along the examples with either a REPL or the `babel-node` CLI and a file. That'll make it so much easier for you to **internalize the concepts** discussed in the series. If you aren't the *"install things on my computer"* kind of human, you might prefer to hop on CodePen and then click on the gear icon for JavaScript – *they have a Babel preprocessor which makes trying out ES6 a breeze.* Another alternative that's also quite useful is to use Babel's online REPL – *it'll show you compiled ES5 code to the right of your ES6 code for quick comparison.*

Before getting into it, let me *shamelessly ask for your support* if you're enjoying my ES6 in Depth series. Your contributions will go towards helping me keep up with the schedule, server bills, keeping me fed, and maintaining **Pony Foo** as a veritable source of JavaScript goodies.

Thanks for listening to that, and let's go into generators now! If you haven't yet, you should read yesterday's article on iterators, as this article pretty much assumes that you've read it.

Generator Functions and Generator Objects

Generators are a new feature in ES6. You declare a *generator function* which returns generator objects `g` that can then be iterated using any of `Array.from(g)`, `[...g]`, or `for value of g` loops. Generator functions allow you to declare a special kind of *iterator*. These iterators can suspend execution while retaining their context. We already examined iterators in the previous article and how their `.next()` method is called once at a time to pull values from a sequence.

Here is an example generator function. Note the `*` after `function`. That's not a typo, that's how you mark a generator function as a *generator*.

```
function* generator () {
  yield 'f'
  yield 'o'
  yield 'o'
}
```

Generator objects conform to both the *iterable* protocol and the *iterator* protocol. This means...

```
var g = generator()
// a generator object g is built using the generator function
typeof g[Symbol.iterator] === 'function'
// it's an iterable because it has an @@iterator
typeof g.next === 'function'
// it's also an iterator because it has a .next method
g[Symbol.iterator]() === g
// the iterator for a generator object is the generator object itself
console.log([...g])
// <- ['f', 'o', 'o']
console.log(Array.from(g))
// <- ['f', 'o', 'o']
```

(This article is starting to sound an awful lot like a Math course...)

When you create a generator object *(I'll just say "generator" from here on out)*, you'll get an *iterator* that uses the generator to produce its *sequence*. Whenever a `yield` expression is reached, that value is emitted by the iterator and **function execution is suspended**.

Let's use a different example, this time with some other statements mixed in between `yield` expressions. This is a simple generator but it behaves in an interesting enough way for our purposes here.

```
function* generator () {
  yield 'p'
  console.log('o')
```

```
  yield 'n'
  console.log('y')
  yield 'f'
  console.log('o')
  yield 'o'
  console.log('!')
}
```

If we use a `for..of` loop, this will print `ponyfoo!` one character at a time, as expected.

```
var foo = generator()
for (let pony of foo) {
  console.log(pony)
  // <- 'p'
  // <- 'o'
  // <- 'n'
  // <- 'y'
  // <- 'f'
  // <- 'o'
  // <- 'o'
  // <- '!'
}
```

What about using the spread `[...foo]` syntax? Things turn out a little different here. This might be a little unexpected, but that's how generators work, everything that's not yielded ends up becoming **a side effect**. As the sequence is being constructed, the `console.log` statements in between `yield` calls are executed, and they print characters to the console before `foo` is spread over an array. The previous example worked because we were printing characters as soon as they were pulled from the sequence, instead of waiting to construct a range for the entire sequence first.

```
var foo = generator()
console.log([...foo])
// <- 'o'
// <- 'y'
// <- 'o'
// <- '!'
// <- ['p', 'n', 'f', 'o']
```

A neat aspect of generator functions is that you can also use `yield*` to delegate to another generator function. Want a very contrived way to split `'ponyfoo'` into individual characters? Since strings in ES6 adhere to the *iterable* protocol, you could do the following.

```
function* generator () {
  yield* 'ponyfoo'
}
console.log([...generator()])
// <- ['p', 'o', 'n', 'y', 'f', 'o', 'o']
```

Of course, in the real world you could just do `[...'ponyfoo']`, since spread supports iterables just fine. Just like you could `yield*` a string, you can `yield*` anything that adheres to the iterable protocol. That includes other generators, arrays, and come ES6 – *just about anything.*

```
var foo = {
  [Symbol.iterator]: () => ({
    items: ['p', 'o', 'n', 'y', 'f', 'o', 'o'],
    next: function next () {
      return {
        done: this.items.length === 0,
        value: this.items.shift()
      }
    }
  })
}
function* multiplier (value) {
  yield value * 2
  yield value * 3
  yield value * 4
  yield value * 5
}
function* trailmix () {
  yield 0
  yield* [1, 2]
  yield* [...multiplier(2)]
  yield* multiplier(3)
  yield* foo
```

```
}
console.log([...trailmix()])
// <- [0, 1, 2, 4, 6, 8, 10, 6, 9, 12, 15, 'p', 'o', 'n', 'y', 'f', 'o', 'o']
```

You could also iterate the sequence by hand, calling `.next()`. This approach gives you the most control over the iteration, but it's also the most involved. There's a few features you can leverage here that give you even more control over the iteration.

Iterating Over Generators by Hand

Besides iterating over `trailmix` as we've already covered, using `[...trailmix()]`, `for value of trailmix()`, and `Array.from(trailmix())`, we could use the generator returned by `trailmix()` directly, and iterate over that. But `trailmix` was an overcomplicated showcase of `yield*`, let's go back to the *side-effects* `generator` for this one.

```
function* generator () {
  yield 'p'
  console.log('o')
  yield 'n'
  console.log('y')
  yield 'f'
  console.log('o')
  yield 'o'
  console.log('!')
}
var g = generator()
while (true) {
  let item = g.next()
  if (item.done) {
    break
  }
  console.log(item.value)
}
```

Just like we learned yesterday, any items returned by an iterator will have a `done` property that indicates whether the sequence has reached its end, and a `value` indicating the current value in the sequence.

> If you're confused as to **why the `'!'` is printed** even though there are no more `yield` expressions after it, that's because `g.next()` doesn't know that. The way it works is that each time its called, it executes the method until a `yield` expression is reached, emits its value and *suspends execution*. The next time `g.next()` is called, _execution is resumed _from where it left off *(the last `yield` expression)*, until the next `yield` expression is reached. When no `yield` expression is reached, the generator returns `{ done: true }`, signaling that the sequence has ended. At this point, the `console.log('!')` statement has been already executed, though.
>
> It's also worth noting that **context is preserved** across suspensions and resumptions. That means generators can be stateful. Generators are, in fact, the underlying implementation for `async` / `await` semantics coming in ES7.

Whenever `.next()` is called on a generator, there's four "events" that will suspend execution in the generator, returning an *`IteratorResult`* to the caller of `.next()`.

- A `yield` expression returning the *next* value in the sequence
- A `return` statement returning the *last* value in the sequence
- A `throw` statement halts execution in the generator entirely
- Reaching the end of the generator function signals `{ done: true }`

Once the `g` generator ended iterating over a sequence, subsequent calls to `g.next()` will have no effect and just return `{ done: true }`.

```
function* generator () {
  yield 'only'
}
var g = generator()
console.log(g.next())
// <- { done: false, value: 'only' }
console.log(g.next())
// <- { done: true }
console.log(g.next())
// <- { done: true }
```

Generators: The ~~Weird~~ ***Awesome*** Parts

Generator objects come with a couple more methods besides `.next`. These are `.return` and `.throw`. We've already covered `.next` extensively, but not quite. You could also use `.next(value)` to send values *into the generator*.

Let's make **a magic 8-ball generator**. First off, you'll need some answers. Wikipedia obliges, yielding 20 possible answers for our magic 8-ball.

```
var answers = [
  `It is certain`, `It is decidedly so`, `Without a doubt`,
  `Yes definitely`, `You may rely on it`, `As I see it, yes`,
  `Most likely`, `Outlook good`, `Yes`, `Signs point to yes`,
  `Reply hazy try again`, `Ask again later`, `Better not tell you now`,
  `Cannot predict now`, `Concentrate and ask again`,
  `Don't count on it`, `My reply is no`, `My sources say no`,
  `Outlook not so good`, `Very doubtful`
]
function answer () {
  return answers[Math.floor(Math.random() * answers.length)]
}
```

The following generator function can act as a *"genie"* that answers any questions you might have for them. Note how we discard the first result from `g.next()`. That's because the first call to `.next` enters the generator and there's no `yield` expression waiting to capture the `value` from `g.next(value)`.

```
function* chat () {
  while (true) {
    let question = yield '[Genie] ' + answer()
    console.log(question)
  }
}
var g = chat()
g.next()
console.log(g.next('[Me] Will ES6 die a painful death?').value)
// <- '[Me] Will ES6 die a painful death?'
// <- '[Genie] My sources say no'
console.log(g.next('[Me] How youuu doing?').value)
// <- '[Me] How youuu doing?'
// <- '[Genie] Concentrate and ask again'
```

Randomly dropping `g.next()` feels like a very dirty coding practice, though. What else could we do? We could flip responsibilities around.

Inversion of Control

We could have the Genie be in control, and have the generator ask the questions. How would that look like? At first, you might think that the code below is unconventional, but in fact, most libraries built around generators work by inverting responsibility.

```
function* chat () {
  yield '[Me] Will ES6 die a painful death?'
  yield '[Me] How youuu doing?'
}
var g = chat()
while (true) {
  let question = g.next()
  if (question.done) {
    break
  }
  console.log(question.value)
  console.log('[Genie] ' + answer())
  // <- '[Me] Will ES6 die a painful death?'
  // <- '[Genie] Very doubtful'
  // <- '[Me] How youuu doing?'
  // <- '[Genie] My reply is no'
}
```

You would expect the **generator to do the heavy lifting** of an iteration, but in fact generators make it easy to iterate over things by suspending execution of themselves — and deferring the heavy lifting. That's one of the most powerful aspects of generators. Suppose now that the iterator is a `genie` method in a library, like so:

```
function genie (questions) {
  var g = questions()
  while (true) {
    let question = g.next()
    if (question.done) {
      break
    }
    console.log(question.value)
```

```
    console.log('[Genie] ' + answer())
  }
}
```

To use it, all you'd have to do is pass in a simple generator like the one we just made.

```
genie(function* questions () {
  yield '[Me] Will ES6 die a painful death?'
  yield '[Me] How youuu doing?'
})
```

Compare that to the generator we had before, where questions were sent to the generator instead of the other way around. See how much more complicated the logic would have to be to achieve the same goal? Letting the library deal with the flow control means you can **just worry about the *thing* you want to iterate** over, and you can **delegate *how* to iterate over it**. But yes, it does mean your code now has an asterisk in it. *Weird*.

Dealing with asynchronous flows

Imagine now that the `genie` library gets its magic 8-ball answers from an API. How does that look then? Probably something like the snippet below. Assume the `xhr` pseudocode call always yields JSON responses like `{ answer: 'No' }`. Keep in mind this is a simple example that just processes each question in series. You could put together different and more complex flow control algorithms depending on what you're looking for.

This is just a demonstration of the sheer power of generators.

```
function genie (questions) {
  var g = questions()
  pull()
  function pull () {
    let question = g.next()
    if (question.done) {
      return
    }
    ask(question.value, pull)
  }
  function ask (q, next) {
    xhr('https://computer.genie/?q=' + encodeURIComponent(q), got)
    function got (err, res, body) {
      if (err) {
        // todo
      }
      console.log(q)
      console.log('[Genie] ' + body.answer)
      next()
    }
  }
}
```

See this link for a live demo on the Babel REPL

Even though we've just made our `genie` method asynchronous and are now using an API to fetch responses to the user's questions, the way the consumer uses the `genie` library by passing a `questions` generator function *remains unchanged!* That's awesome.

We haven't handled the case for an `err` coming out of the API. That's inconvenient. What can we do about that one?

Throwing *at* a Generator

Now that we've figured out that the most important aspect of generators is *actually the control flow code* that decides when to call `g.next()`, we can look at the other two methods and actually understand their purpose. Before shifting our thinking into *"the generator defines **what** to iterate over, not the **how**"*, we would've been hard pressed to find a user case for `g.throw`. Now however it seems immediately obvious. The flow control that leverages a generator needs to be able to tell the generator that's yielding the sequence to be iterated when something goes wrong processing an item in the sequence.

In the case of our `genie` flow, that is now using `xhr`, we may experience network issues and be unable to continue processing items, or we may want to warn the user about unexpected errors. Here's how, we simply add `g.throw(error)` in our control flow code.

```
function genie (questions) {
```

```
  var g = questions()
  pull()
  function pull () {
    let question = g.next()
    if (question.done) {
      return
    }
    ask(question.value, pull)
  }
  function ask (q, next) {
    xhr('https://computer.genie/?q=' + encodeURIComponent(q), got)
    function got (err, res, body) {
      if (err) {
        g.throw(err)
      }
      console.log(q)
      console.log('[Genie] ' + body.answer)
      next()
    }
  }
}
```

The *user code* is still unchanged, though. In between `yield` statements it may throw errors now. You could use `try` / `catch` blocks to address those issues. If you do this, execution will be able to resume. The good thing is that this is up to the user, it's still perfectly sequential on their end, and they can leverage `try` / `catch` semantics just like in high-school.

```
genie(function* questions () {
  try {
    yield '[Me] Will ES6 die a painful death?'
  } catch (e) {
    console.error('Error', e.message)
  }
  try {
    yield '[Me] How youuu doing?'
  } catch (e) {
    console.error('Error', e.message)
  }
})
```

Returning on Behalf of a Generator

Usually not as interesting in asynchronous control flow mechanisms in general, the `g.return()` method allows you to resume execution inside a generator function, much like `g.throw()` did moments earlier. The key difference is that `g.return()` won't result in an exception at the generator level, although **it will end the sequence.**

```
function* numbers () {
  yield 1
  yield 2
  yield 3
}
var g = numbers()
console.log(g.next())
// <- { done: false, value: 1 }
console.log(g.return())
// <- { done: true }
console.log(g.next())
// <- { done: true }, as we know
```

You could also return a `value` using `g.return(value)` , and the resulting `IteratorResult` will contain said `value` . This is equivalent to having `return value` somewhere in the generator function. You should be careful there though – as neither `for..of` , `[...generator()]` , nor `Array.from(generator())` include the `value` in the `IteratorResult` that signals `{ done: true }` .

```
function* numbers () {
  yield 1
  yield 2
  return 3
  yield 4
}
console.log([...numbers()])
// <- [1, 2]
console.log(Array.from(numbers()))
// <- [1, 2]
```

```
for (let n of numbers()) {
  console.log(n)
  // <- 1
  // <- 2
}
var g = numbers()
console.log(g.next())
// <- { done: false, value: 1 }
console.log(g.next())
// <- { done: false, value: 2 }
console.log(g.next())
// <- { done: true, value: 3 }
console.log(g.next())
// <- { done: true }
```

Using `g.return` is no different in this regard, think of it as the programmatic equivalent of what we just did.

```
function* numbers () {
  yield 1
  yield 2
  return 3
  yield 4
}
var g = numbers()
console.log(g.next())
// <- { done: false, value: 1 }
console.log(g.return(5))
// <- { done: true, value: 5 }
console.log(g.next())
// <- { done: true }
```

You can avoid the impending sequence termination, as Axel points out, if the code in the generator function when `g.return()` got called is wrapped in `try` / `finally`. Once the `yield` expressions in the `finally` block are over, the sequence *will* end with the `value` passed to `g.return(value)`

```
function* numbers () {
  yield 1
  try {
    yield 2
  } finally {
    yield 3
    yield 4
  }
  yield 5
}
var g = numbers()
console.log(g.next())
// <- { done: false, value: 1 }
console.log(g.next())
// <- { done: false, value: 2 }
console.log(g.return(6))
// <- { done: false, value: 3 }
console.log(g.next())
// <- { done: false, value: 4 }
console.log(g.next())
// <- { done: true, value: 6 }
```

That's all there is to know when it comes to generators *in terms of functionality*.

Use Cases for ES6 Generators

At this point in the article you should feel comfortable with the concepts of iterators, iterables, and generators in ES6. If you feel like reading more on the subject, I highly recommend you go over Axel's article on generators, as he put together an amazing write-up on use cases for generators just *a few months ago*.

↑ **Back to top**

ES6 Symbols in Depth

Buon giorno! Willkommen to ES6 – *"I can't believe this is yet another installment"* – in Depth. If you have no idea how you got here or what ES6 even is, I recommend reading A Brief History of ES6 Tooling. Then, make your way through destructuring, template literals, arrow functions, the spread operator and rest parameters, improvements coming to object literals, the new *classes* sugar on top of prototypes, `let`, `const`, and the *"Temporal Dead Zone"*, iterators, and generators. Today we'll be discussing *Symbols*.

> As I did in previous articles on the series, I would love to point out that you should probably set up Babel and follow along the examples with either a REPL or the `babel-node` CLI and a file. That'll make it so much easier for you to **internalize the concepts** discussed in the series. If you aren't the *"install things on my computer"* kind of human, you might prefer to hop on CodePen and then click on the gear icon for JavaScript – *they have a Babel preprocessor which makes trying out ES6 a breeze.* Another alternative that's also quite useful is to use Babel's online REPL – *it'll show you compiled ES5 code to the right of your ES6 code for quick comparison.*

Before getting into it, let me *shamelessly ask for your support* if you're enjoying my ES6 in Depth series. Your contributions will go towards helping me keep up with the schedule, server bills, keeping me fed, and maintaining **Pony Foo** as a veritable source of JavaScript goodies.

Thanks for listening to that, and let's go into symbols now! For a bit of context, you may want to check out the last two articles, – on iterators and generators – where we first talked about Symbols.

What are Symbols?

Symbols are a new primitive type in ES6. If you ask me, they're *an awful lot like strings*. Just like with numbers and strings, symbols also come with their accompanying `Symbol` wrapper object.

We can create our own Symbols.

```
var mystery = Symbol()()
```

Note that there was no `new`. The `new` operator even throws a `TypeError` when we try it on `Symbol`.

```
var oops = new Symbol()()
// <- TypeError
```

For debugging purposes, you can describe symbols.

```
var mystery = Symbol()('this is a descriptive description')
```

Symbols are *immutable*. Just like numbers or strings. Note however that symbols are *unique*, unlike primitive numbers and strings.

```
console.log(Symbol() === Symbol()())
// <- false
console.log(Symbol('foo') === Symbol()('foo'))
// <- false
```

Symbols are *symbols*.

```
console.log(typeof Symbol()())
// <- 'symbol'
console.log(typeof Symbol()('foo'))
// <- 'symbol'
```

There are three different flavors of symbols – each flavor is accessed in a different way. We'll explore each of these and slowly figure out what all of this means.

- You can access local symbols by obtaining a reference to them *directly*
- You can place symbols on the *global registry* and access them across *realms*
- "Well-known" symbols exist across *realms* – but you can't create them and they're not on the *global registry*

What the heck is a *realm*, you say? A *realm* is **spec-speak** for any execution context, such as the page your application is running in, or an `<iframe>` within your page.

The "Runtime-Wide" Symbol Registry

There's two methods you can use to add symbols to the runtime-wide symbol registry: `Symbol.for(key)` and `Symbol.keyFor(symbol)`. What do these do?

`Symbol.for(key)`

This method looks up `key` in the runtime-wide symbol registry. If a symbol with that `key` exists in the global registry, that symbol is returned. If no symbol with that `key` is found in the registry, one is created. That's to say, `Symbol.for(key)` is *idempotent*. In the snippet below, the first call to `Symbol.for('foo')` creates a symbol, adds it to the registry, and returns it. The second call returns that same symbol because the `key` is already in the registry by then — and associated to the symbol returned by the first call.

```
Symbol.for('foo') === Symbol().for('foo')
// <- true
```

That is in contrast to what we knew about symbols being unique. The global symbol registry however keeps track of symbols by a `key`. Note that your `key` will also be used as a `description` when the symbols that go into the registry are created. Also note that these symbols are **as global as globals get in JavaScript**, so play nice and use a prefix and don't just name your symbols `'user'` or some generic name like that.

`Symbol.keyFor(symbol)`

Given a symbol `symbol`, `Symbol.keyFor(symbol)` returns the `key` that was associated with `symbol` when the symbol was added to the global registry.

```
var symbol = Symbol.for('foo')
console.log(Symbol.keyFor(symbol))
// <- 'foo'
```

How Wide is Runtime-Wide?

Runtime-wide means the symbols in the global registry are *accessible across code realms*. I'll probably have more success explaining this with a piece of code. It just means the registry is shared across realms.

```
var frame = document.createElement('iframe')
document.body.appendChild(frame)
console.log(Symbol.for('foo') === frame.contentWindow.Symbol.for('foo'))
// <- true
```

The "Well-Known" Symbols

Let me put you at ease: **these aren't actually well-known at all.** Far from it. I didn't have any idea these things existed until a few months ago. Why are they *"well-known"*, then? That's because they are JavaScript *built-ins*, and they are used to control parts of the language. They weren't exposed to user code before ES6, but now you can fiddle with them.

A great example of a *"well-known"* symbol is something we've already been playing with on Pony Foo: the `Symbol.iterator` well-known symbol. We used that symbol to define the `@@iterator` method on objects that adhere to the *iterator* protocol. There's a list of well-known symbols on MDN, but few of them are documented at the time of this writing.

One of the well-known symbols that *is* documented at this time is `Symbol.match`. According to MDN, you can set the `Symbol.match` property on regular expressions to `false` and have them behave as string literals when matching *(instead of regular expressions, which don't play nice with `.startsWith`, `.endsWith`, or includes() – MDN"> `.includes`)*.

This part of the spec hasn't been implemented in Babel yet, – *I assume that's just because it's not worth the trouble* – but supposedly it goes like this.

```
var text = '/foo/'
var literal = /foo/
literal[Symbol.match] = false
console.log(text.startsWith(literal))
// <- true
```

Why you'd want to do that instead of just casting `literal` to a string *is beyond me*.

```
var text = '/foo/'
var casted = /foo/.toString()
console.log(text.startsWith(casted))
```

```
// <- true
```

I suspect the language has **legitimate performance reasons** that warrant the existence of this symbol, but I don't think it'll become a front-end development staple anytime soon.

> Regardless, `Symbol.iterator` is actually very useful, and I'm sure other well-known symbols are useful as well.

Note that well-known symbols are unique, but **shared across realms**, even when they're not accessible through the *global registry*.

```
var frame = document.createElement('iframe')
document.body.appendChild(frame)
console.log(Symbol.iterator === frame.contentWindow.Symbol.iterator)
// <- true
```

Not accessible through the *global registry*? Nope!

```
console.log(Symbol.keyFor(Symbol.iterator))
// <- undefined
```

Accessing them statically from anywhere should be more than enough, though.

Symbols and Iteration

Any consumer of the *iterable* protocol obviously ignores symbols other than the well-known `Symbol.iterator` that would define how to iterate and help identify the object as an *iterable*.

```
var foo = {
  [Symbol()]: 'foo',
  [Symbol('foo')]: 'bar',
  [Symbol.for('bar')]: 'baz',
  what: 'ever'
}
console.log([...foo])
// <- []
```

The ES5 `Object.keys` method ignores symbols.

```
console.log(Object.keys(foo))
// <- ['what']
```

Same goes for `JSON.stringify`.

```
console.log(JSON.stringify(foo))
// <- {"what":"ever"}
```

So, `for..in` then? Nope.

```
for (let key in foo) {
  console.log(key)
  // <- 'what'
}
```

I know, `Object.getOwnPropertyNames`. Nah! – *but close.*

```
console.log(Object.getOwnPropertyNames(foo))
// <- ['what']
```

You need to be explicitly looking for symbols to stumble upon them. They're like JavaScript neutrinos. You can use `Object.getOwnPropertySymbols` to detect them.

```
console.log(Object.getOwnPropertySymbols(foo))
// <- [Symbol(), Symbol('foo'), Symbol.for('bar')]
```

The magical drapes of symbols drop, and you can now iterate over the symbols with a `for..of` loop to finally figure out the treasures they were guarding. Hopefully, they won't be as disappointing as the flukes in the snippet below.

```
for (let symbol of Object.getOwnPropertySymbols(foo)) {
  console.log(foo[symbol])
  // <- 'foo'
```

```
  // <- 'bar'
  // <- 'baz'
}
```

Why Would I Want Symbols?

There are a few different uses for symbols.

Name Clashes

You can use symbols to **avoid name clashes** in property keys. This is important when following the *"objects as hash maps"* pattern, which regularly ends up failing miserably as native methods and properties are overridden unintentionally *(or maliciously)*.

"Privacy"?

Symbols are *invisible to all "reflection" methods before ES6*. This can be useful in some scenarios, but they're not private by any stretch of imagination, as we've just demonstrated with the `Object.getOwnPropertySymbols` API.

That being said, the fact that you have to actively look for symbols to find them means they're useful in situations where you want to define metadata that shouldn't be part of iterable sequences for arrays or any *iterable* objects.

Defining Protocols

I think the *biggest use case for symbols* is exactly what the ES6 implementers use them for: **defining protocols** – just like there's `Symbol.iterator` which allows you to define how an object can be iterated.

Imagine for instance a library like `dragula` defining a protocol through `Symbol.for('dragula.moves')`, where you could add a method on that `Symbol` to any DOM elements. If a DOM element follows the protocol, then `dragula` could call the `el[Symbol.for('dragula.moves')]()` user-defined method to assert whether the element can be moved.

This way, the logic about elements being draggable by `dragula` is shifted from a single place for the entire `drake` *(the `options` for an instance of `dragula`)*, to each individual DOM element. That'd make it easier to deal with complex interactions in larger implementations, as the logic would be delegated to individual DOM nodes instead of being centralized in a single `options.moves` method.

🠕 **Back to top**

ES6 Maps in Depth

Hello, this is ES6 – *"Please make them stop"* – in Depth. New here? Start with A Brief History of ES6 Tooling. Then, make your way through destructuring, template literals, arrow functions, the spread operator and rest parameters, improvements coming to object literals, the new *classes* sugar on top of prototypes, `let`, `const`, and the *"Temporal Dead Zone"*, iterators, generators, and Symbols. Today we'll be discussing a new collection data structure objects coming in ES6 – I'm talking about `Map`.

> As I did in previous articles on the series, I would love to point out that you should probably set up Babel and follow along the examples with either a REPL or the `babel-node` CLI and a file. That'll make it so much easier for you to **internalize the concepts** discussed in the series. If you aren't the *"install things on my computer"* kind of human, you might prefer to hop on CodePen and then click on the gear icon for JavaScript – *they have a Babel preprocessor which makes trying out ES6 a breeze.* Another alternative that's also quite useful is to use Babel's online REPL – *it'll show you compiled ES5 code to the right of your ES6 code for quick comparison.*

Before getting into it, let me *shamelessly ask for your support* if you're enjoying my ES6 in Depth series. Your contributions will go towards helping me keep up with the schedule, server bills, keeping me fed, and maintaining **Pony Foo** as a veritable source of JavaScript goodies.

Thanks for reading that, and let's go into collections now! For a bit of context, you may want to check out the article on iterators – *which are closely related to ES6 collections* – and the one on spread and rest parameters.

Now, let's start with `Map`. I moved the rest of the ES6 collections to tomorrow's publication in order to keep the series sane, as otherwise this would've been too long for a single article!

Before ES6, There Were Hash-Maps

A very common *ab*use case of JavaScript objects is hash-maps, where we map string keys to arbitrary values. For example, one might use an object to map `npm` package names to their metadata, like so:

```
var registry = {}
function add (name, meta) {
  registry[name] = meta
}
function get (name) {
  return registry[name]
}
add('contra', { description: 'Asynchronous flow control' })
add('dragula', { description: 'Drag and drop' })
add('woofmark', { description: 'Markdown and WYSIWYG editor' })
```

There are several issues with this approach, to wit:

- **Security issues** where user-provided keys like `__proto__`, `toString`, or anything in `Object.prototype` break expectations and make interaction with these kinds of *hash-map* data structures more cumbersome
- Iteration over list items is verbose with `Object.keys(registry).forEach` or implementing the *iterable* protocol on the `registry`
- Keys are limited to strings, making it hard to create hash-maps where you'd like to index values by DOM elements or other non-string references

The first problem could be fixed using a prefix, and being careful to always get or set values in the hash-map through methods. It would be even better to use ES6 proxies, but we *won't be covering those until tomorrow!*

```
var registry = {}
function add (name, meta) {
  registry['map:' + name] = meta
}
function get (name) {
  return registry['map:' + name]
}
add('contra', { description: 'Asynchronous flow control' })
add('dragula', { description: 'Drag and drop' })
add('woofmark', { description: 'Markdown and WYSIWYG editor' })
```

Luckily for us, though, *ES6 maps* provide us with an even better solution to the key-naming security issue. At the same time they facilitate collection behaviors out the box that may also come in handy. Let's plunge into their practical usage and inner workings.

ES6 Maps

Map is a key/value data structure in ES6. It provides a better data structure to be used for hash-maps. Here's how what we had earlier looks like with ES6 maps.

```
var map = new Map()
map.set('contra', { description: 'Asynchronous flow control' })
map.set('dragula', { description: 'Drag and drop' })
map.set('woofmark', { description: 'Markdown and WYSIWYG editor' })
```

One of the important differences is also that you're able to use anything for the keys. You're not just limited to primitive values like symbols, numbers, or strings, but you can even use functions, objects and dates — too. Keys won't be casted to strings like with regular objects, either.

```
var map = new Map()
map.set(new Date(), function today () {})
map.set(() => 'key', { pony: 'foo' })
map.set(Symbol('items'), [1, 2])
```

You can also provide `Map` objects with any object that follows the *iterable* protocol and produces a collection such as `[['key', 'value'], ['key', 'value']]`.

```
var map = new Map([
  [new Date(), function today () {}],
  [() => 'key', { pony: 'foo' }],
```

```
  [Symbol('items'), [1, 2]]
])
```

The above would be effectively the same as the following. Note how we're using destructuring in the parameters of `items.forEach` to *effortlessly* pull the `key` and `value` out of the two-dimensional `item`.

```
var items = [
  [new Date(), function today () {}],
  [() => 'key', { pony: 'foo' }],
  [Symbol('items'), [1, 2]]
]
var map = new Map()
items.forEach(([key, value]) => map.set(key, value))
```

Of course, it's kind of silly to go through the trouble of adding items one by one when you can just feed an iterable to your `Map`. Speaking of iterables – `Map` adheres to the *iterable* protocol. It's very easy to pull a key-value pair collection much like the ones you can feed to the `Map` constructor.

Naturally, we can use the spread operator to this effect.

```
var map = new Map()
map.set('p', 'o')
map.set('n', 'y')
map.set('f', 'o')
map.set('o', '!')
console.log([...map])
// <- [['p', 'o'], ['n', 'y'], ['f', 'o'], ['o', '!']]
```

You could also use a `for..of` loop, and we could combine that with destructuring to make it seriously terse. Also, remember template literals?

```
var map = new Map()
map.set('p', 'o')
map.set('n', 'y')
map.set('f', 'o')
map.set('o', '!')
for (let [key, value] of map) {
  console.log(`${key}: ${value}`)
  // <- 'p: o'
  // <- 'n: y'
  // <- 'f: o'
  // <- 'o: !'
}
```

Even though maps have a programmatic API to add items, keys are unique, just like with hash-maps. Setting a key over and over again will only overwrite its value.

```
var map = new Map()
map.set('a', 'a')
map.set('a', 'b')
map.set('a', 'c')
console.log([...map])
// <- [['a', 'c']]
```

In ES6 `Map`, `NaN` becomes a "corner-case" that gets **treated as a value that's equal to itself** even though the following expression actually evaluates to `true` – `NaN !== NaN`.

```
console.log(NaN === NaN)
// <- false
var map = new Map()
map.set(NaN, 'foo')
map.set(NaN, 'bar')
console.log([...map])
// <- [[NaN, 'bar']]
```

Hash-Maps and the DOM

In ES5, whenever we had a DOM element we wanted to associate with an API object for some library, we had to follow a verbose and slow pattern like the one below. The following piece of code just returns an API object with a bunch of methods for a given DOM element, allowing us to put and remove DOM elements from the cache, and also allowing us to retrieve the API object for a DOM element – if one already exists.

```
var cache = []
function put (el, api) {
  cache.push({ el: el, api: api })
}
function find (el) {
  for (i = 0; i < cache.length; i++) {
    if (cache[i].el === el) {
      return cache[i].api
    }
  }
}
function destroy (el) {
  for (i = 0; i < cache.length; i++) {
    if (cache[i].el === el) {
      cache.splice(i, 1)
      return
    }
  }
}
function thing (el) {
  var api = find(el)
  if (api) {
    return api
  }
  api = {
    method: method,
    method2: method2,
    method3: method3,
    destroy: destroy.bind(null, el)
  }
  put(el, api)
  return api
}
```

One of the coolest aspects of `Map`, *as I've previously mentioned*, is the ability to index by DOM elements. The fact that `Map` also has collection manipulation abilities also greatly simplifies things.

```
var cache = new Map()
function put (el, api) {
  cache.set(el, api)
}
function find (el) {
  return cache.get(el)
}
function destroy (el) {
  cache.delete(el)
}
function thing (el) {
  var api = find(el)
  if (api) {
    return api
  }
  api = {
    method: method,
    method2: method2,
    method3: method3,
    destroy: destroy.bind(null, el)
  }
  put(el, api)
  return api
}
```

The fact that these methods have now become one liners means we can just inline them, as readability is no longer an issue. We just went from *~30 LOC* to **half that amount**. Needless to say, at some point in the future this will also perform *much* faster than the haystack alternative.

```
var cache = new Map()
function thing (el) {
  var api = cache.get(el)
  if (api) {
    return api
  }
  api = {
    method: method,
    method2: method2,
    method3: method3,
```

```
    destroy: () => cache.delete(el)
  }
  cache.set(el, api)
  return api
}
```

The simplicity of `Map` is amazing. If you ask me, we desperately needed this feature in JavaScript. Being to index a collection by arbitrary objects is **super important**.

> What else can we do with `Map` ?

Collection Methods in `Map`

Maps make it very easy to probe the collection and figure out whether a `key` is defined in the `Map`. As we noted earlier, `NaN` equals `NaN` as far as `Map` is concerned. However, `Symbol` values are always different, so you'll have to use them by value!

```
var map = new Map([[NaN, 1], [Symbol(), 2], ['foo', 'bar']])
console.log(map.has(NaN))
// <- true
console.log(map.has(Symbol()))
// <- false
console.log(map.has('foo'))
// <- true
console.log(map.has('bar'))
// <- false
```

As long as you keep a `Symbol` reference around, you'll be okay. *Keep your references close, and your `Symbol`s closer?*

```
var sym = Symbol()
var map = new Map([[NaN, 1], [sym, 2], ['foo', 'bar']])
console.log(map.has(sym))
// <- true
```

Also, remember the **no key-casting** thing? *Beware!* We are so used to objects casting keys to strings that this may bite you if you're not careful.

```
var map = new Map([[1, 'a']])
console.log(map.has(1))
// <- true
console.log(map.has('1'))
// <- false
```

You can also clear a `Map` entirely of entries without losing a reference to it. This can be very handy sometimes.

```
var map = new Map([[1, 2], [3, 4], [5, 6]])
map.clear()
console.log(map.has(1))
// <- false
console.log([...map])
// <- []
```

When you use `Map` as an iterable, you are actually looping over its `.entries()`. That means that you don't need to **explicitly** iterate over `.entries()`. It'll be done on your behalf anyways. You do remember `Symbol.iterator`, right?

```
console.log(map[Symbol.iterator] === map.entries)
// <- true
```

Just like `.entries()`, `Map` has two other iterators you can leverage. These are `.keys()` and `.values()`. I'm sure you guessed what sequences of values they yield, but here's a code snippet anyways.

```
var map = new Map([[1, 2], [3, 4], [5, 6]])
console.log([...map.keys()])
// <- [1, 3, 5]
console.log([...map.values()])
// <- [2, 4, 6]
```

Maps also come with a *read-only* `.size` property that behaves sort of like `Array.prototype.length` — at any point in time it gives you the current amount of entries in the map.

```
var map = new Map([[1, 2], [3, 4], [5, 6]])
console.log(map.size)
```

```
// <- 3
map.delete(3)
console.log(map.size)
// <- 2
map.clear()
console.log(map.size)
// <- 0
```

One more aspect of `Map` that's worth mentioning is that their entries are always iterated in **insertion order**. This is in contrast with `Object.keys` loops which follow an arbitrary order.

> The `for..in` statement iterates over the enumerable properties of an object, in arbitrary order.

Maps also have a `.forEach` method that's identical in *behavior* to that in ES5 `Array` objects. Once again, keys do not get casted into strings here.

```
var map = new Map([[NaN, 1], [Symbol(), 2], ['foo', 'bar']])
map.forEach((value, key) => console.log(key, value))
// <- NaN 1
// <- Symbol() 2
// <- 'foo' 'bar'
```

> Get up early tomorrow morning, we'll be having `WeakMap`, `Set`, and `WeakSet` for breakfast :)

↑ **Back to top**

ES6 WeakMaps, Sets, and WeakSets in Depth

Welcome once again to ES6 – *"I can't take this anymore"* – in Depth. New here? Start with A Brief History of ES6 Tooling. Then, make your way through destructuring, template literals, arrow functions, the spread operator and rest parameters, improvements coming to object literals, the new *classes* sugar on top of prototypes, `let`, `const`, and the *"Temporal Dead Zone"*, iterators, generators, Symbols and Maps. This morning we'll be discussing three more collection data structures coming in ES6: `WeakMap`, `Set` and `WeakSet`.

> As I did in previous articles on the series, I would love to point out that you should probably set up Babel and follow along the examples with either a REPL or the `babel-node` CLI and a file. That'll make it so much easier for you to **internalize the concepts** discussed in the series. If you aren't the *"install things on my computer"* kind of human, you might prefer to hop on CodePen and then click on the gear icon for JavaScript – *they have a Babel preprocessor which makes trying out ES6 a breeze.* Another alternative that's also quite useful is to use Babel's online REPL – *it'll show you compiled ES5 code to the right of your ES6 code for quick comparison.*

Before getting into it, let me *shamelessly ask for your support* if you're enjoying my ES6 in Depth series. Your contributions will go towards helping me keep up with the schedule, server bills, keeping me fed, and maintaining **Pony Foo** as a veritable source of JavaScript goodies.

Thanks for reading that, and let's go into collections now! For a bit of context, you may want to check out the article on iterators – *which are closely related to ES6 collections* – and the one on spread and rest parameters.

> Now, let's pick up where we left off – it's time for `WeakMap`.

ES6 WeakMaps

You can think of `WeakMap` as a subset of `Map`. There are a few limitations on `WeakMap` that we didn't find in `Map`. The biggest limitation is that `WeakMap` is not iterable, as opposed to `Map` – that means there is no *iterable* protocol, no `.entries()`, no `.keys()`, no `.values()`, no `.forEach()` and no `.clear()`.

Another *"limitation"* found in `WeakMap` as opposed to `Map` is that every `key` must be an object, and **value types are not admitted as keys**. Note that `Symbol` is a value type as well, and they're not allowed either.

```
var map = new WeakMap()
map.set(1, 2)
// TypeError: 1 is not an object!
map.set(Symbol(), 2)
// TypeError: Invalid value used as weak map key
```

> This is more of a feature than an issue, though, as it enables map keys to be garbage collected when they're only being referenced as `WeakMap` keys. Usually you want this behavior when storing metadata related to something like a DOM node, and now you can keep that metadata in a `WeakMap`. If you want all of those you could always use a regular `Map` as we explored earlier.

You are still able to pass an iterable to populate a `WeakMap` through its constructor.

```
var map = new WeakMap([[new Date(), 'foo'], [() => 'bar', 'baz']])
```

Just like with `Map`, you can use `.has`, `.get`, and `.delete` too.

```
var date = new Date()
var map = new WeakMap([[date, 'foo'], [() => 'bar', 'baz']])
console.log(map.has(date))
// <- true
console.log(map.get(date))
// <- 'foo'
map.delete(date)
console.log(map.has(date))
// <- false
```

Is This a Strictly Worse Map?

I know! You must be wondering – why the hell would I use `WeakMap` when it has so many limitations when compared to `Map`?

The difference that may make `WeakMap` worth it is in its name. `WeakMap` holds references to its keys *weakly*, meaning that if there are no other references to one of its keys, the object is subject to **garbage collection**.

Use cases for `WeakMap` generally revolve around the need to specify metadata or extend an object while still being able to garbage collect it if nobody else cares about it. A perfect example might be the underlying implementation for `process.on('unhandledRejection')` which uses a `WeakMap` to keep track of promises that were rejected but *no error handlers dealt with the rejection* within a tick.

Keeping data about DOM elements that should be released from memory when they're no longer of interested is another very important use case, and in this regard using `WeakMap` is probably an even better solution to the DOM-related API caching solution we wrote about earlier using `Map`.

In so many words then, **no**. `WeakMap` is not strictly worse than `Map` – *they just cater to different use cases.*

ES6 Sets

Sets are *yet another* collection type in ES6. Sets are *very* similar to `Map`. To wit:

- `Set` is also *iterable*
- `Set` constructor also accepts an *iterable*
- `Set` also has a `.size` property
- Keys can also be arbitrary values
- Keys must be unique
- `NaN` equals `NaN` when it comes to `Set` too
- All of `.keys`, `.values`, `.entries`, `.forEach`, ~~`.get`~~, ~~`.set`~~, `.has`, `.delete`, and `.clear`

However, there's a few differences as well!

- Sets only have `values`
- No `set.get` – but **why** would you want `get(value) => value`?
- Having `set.set` would be weird, so we have `set.add` instead
- `set[Symbol.iterator] !== set.entries`
- `set[Symbol.iterator] === set.values`
- `set.keys === set.values`
- `set.entries()` returns an iterator on a sequence of items like `[value, value]`

In the example below you can note how it takes an iterable with duplicate values, it can be spread over an `Array` using the spread operator, and how the duplicate value *has been ignored.*

```
var set = new Set([1, 2, 3, 4, 4])
console.log([...set])
// <- [1, 2, 3, 4]
```

Sets may be a great alternative to work with DOM elements. The following piece of code creates a `Set` with all the `<div>` elements on a page and then prints how many it found. Then, we query the DOM *again* and call `set.add` again for every DOM element. Since they're all already in the `set`, the `.size` property won't change, meaning the `set` remains the same.

```
function divs () {
  return [...document.querySelectorAll('div')]
}
var set = new Set(divs())
console.log(set.size)
// <- 56
divs().forEach(div => set.add(div))
console.log(set.size)
// <- 56
// <- look at that, no duplicates!
```

ES6 WeakSets

Much like with `WeakMap` and `Map`, `WeakSet` is `Set` **plus weakness** minus the *iterability – I just made that term up, didn't I?*

That means you can't iterate over `WeakSet`. Its values must be **unique object references**. If nothing else is referencing a `value` found in a `WeakSet`, it'll be subject to garbage collection.

Much like in `WeakMap`, you can only `.add`, `.has`, and `.delete` values from a `WeakSet`. And just like in `Set`, there's no `.get`.

```
var set = new WeakSet()
set.add({})
set.add(new Date())
```

As we know, we can't use primitive values.

```
var set = new WeakSet()
set.add(Symbol())
// TypeError: invalid value used in weak set
```

Just like with `WeakMap`, passing iterators to the constructor is still allowed even though a `WeakSet` instance is not iterable itself.

```
var set = new WeakSet([new Date(), {}, () => {}, [1]])
```

Use cases for `WeakSet` vary, and here's one from a thread on *es-discuss* – the mailing list for the ECMAScript-262 specification of JavaScript.

```
const foos = new WeakSet()
class Foo {
  constructor() {
    foos.add(this)
  }
  method () {
    if (!foos.has(this)) {
      throw new TypeError('Foo.prototype.method called on incompatible object!')
    }
  }
}
```

As a general rule of thumb, you can also try and figure out whether a `WeakSet` will do when you're considering to use a `WeakMap` as some use cases may overlap. Particularly, if all you need to check for is whether a reference value is in the `WeakSet` or not.

> Next week we'll be having `Proxy` for brunch :)

Back to top

ES6 Proxies in Depth

Cheers, **please come in**. This is ES6 – *"Elaine, you gotta have a baby!"* – in Depth. What? Never heard of it? Check out A Brief History of ES6 Tooling. Then, make your way through destructuring, template literals, arrow functions, the spread operator and rest parameters, improvements coming to object literals, the new *classes* sugar on top of prototypes, `let`, `const`, and the *"Temporal Dead Zone"*, iterators, generators, Symbols, Maps, WeakMaps, Sets, and WeakSets. We'll be discussing *ES6 proxies* today.

> As I did in previous articles on the series, I would love to point out that you should probably set up Babel and follow along the examples with either a REPL or the `babel-node` CLI and a file. That'll make it so much easier for you to **internalize the concepts** discussed in the series. If you aren't the *"install things on my computer"* kind of human, you might prefer to hop on CodePen and then click on the gear icon for JavaScript – *they have a Babel preprocessor which makes trying out ES6 a breeze.* Another alternative that's also quite useful is to use Babel's online REPL – *it'll show you compiled ES5 code to the right of your ES6 code for quick comparison.*
>
> Note that `Proxy` is harder to play around with as Babel doesn't support it unless the underlying browser has support for it. You can check out the ES6 compatibility table for supporting browsers. At the time of this writing, you can use *Microsoft Edge* or *Mozilla Firefox* to try out `Proxy`. Personally, I'll be verifying my examples using *Firefox*.

Before getting into it, let me *shamelessly ask for your support* if you're enjoying my ES6 in Depth series. Your contributions will go towards helping me keep up with the schedule, server bills, keeping me fed, and maintaining **Pony Foo** as a veritable source of JavaScript goodies.

Thanks for reading that, and let's go into Proxies now!

ES6 Proxies

Proxies are a quite interesting feature coming in ES6. In a nutshell, you can use a `Proxy` to determine behavior whenever the properties of a `target` object are accessed. A `handler` object can be used to configure *traps* for your `Proxy`, as we'll see in a bit.

By default, proxies don't do much – in fact they don't do anything. If you don't set any *"options"*, your `proxy` will just work as a *pass-through* to the `target` object – MDN calls this a "no-op forwarding `Proxy`", which makes sense.

```
var target = {}
var handler = {}
var proxy = new Proxy(target, handler)
proxy.a = 'b'
console.log(target.a)
// <- 'b'
console.log(proxy.c === undefined)
// <- true
```

We can make our proxy a bit more interesting by adding traps. Traps allow you to intercept interactions with `target` in different ways, as long as those interactions happen through `proxy`. We could use a `get` *trap* to log every attempt to pull a value out of a property in `target`. Let's try that next.

`get`

The proxy below is able to track any and every **property access** event because it has a `handler.get` trap. It can also be used to *transform* the value we get out of accessing any given property. We can already imagine `Proxy` becoming a staple when it comes to developer tooling.

```
var handler = {
  get (target, key) {
    console.info(`Get on property "${key}"`)
    return target[key]
  }
}
var target = {}
var proxy = new Proxy(target, handler)
proxy.a = 'b'
proxy.a
// <- 'Get on property "a"'
proxy.b
// <- 'Get on property "b"'
```

Of course, your getter doesn't necessarily have to return the original `target[key]` value. How about finally making those `_prop` properties actually private?

`set`

Know how we usually define conventions such as Angular's *dollar signs* where properties prefixed by a single dollar sign should hardly be accessed from an application and properties prefixed by two dollar signs should **not be accessed at all**? We usually do something like that ourselves in our applications, typically in the form of underscore-prefixed variables.

The `Proxy` in the example below prevents property access for both `get` and `set` *(via a `handler.set` trap)* while accessing `target` through `proxy`. Note how `set` always returns `true` here? — this means that setting the property `key` to a given `value` should *succeed*. If the return value for the `set` trap is `false`, setting the property value will throw a `TypeError` under strict mode, and otherwise fail silently.

```
var handler = {
  get (target, key) {
    invariant(key, 'get')
    return target[key]
  },
  set (target, key, value) {
    invariant(key, 'set')
    return true
  }
}
function invariant (key, action) {
  if (key[0] === '_') {
    throw new Error(`Invalid attempt to ${action} private "${key}" property`)
  }
}
var target = {}
var proxy = new Proxy(target, handler)
proxy.a = 'b'
console.log(proxy.a)
// <- 'b'
proxy._prop
// <- Error: Invalid attempt to get private "_prop" property
proxy._prop = 'c'
// <- Error: Invalid attempt to set private "_prop" property
```

You do remember string interpolation with template literals, right?

It might be worth mentioning that the `target` object *(the object being proxied)* should often be completely hidden from accessors in proxying scenarios. Effectively **preventing direct access** to the `target` and instead forcing access to `target` exclusively through `proxy`. Consumers of `proxy` will get to access `target` through the `Proxy` object, but will have to **obey your access rules** — such as *"properties prefixed with `_` are off-limits"*.

To that end, you could simply wrap your proxied object in a method, and then return the `proxy`.

```
function proxied () {
  var target = {}
  var handler = {
    get (target, key) {
      invariant(key, 'get')
      return target[key]
    },
    set (target, key, value) {
      invariant(key, 'set')
      return true
    }
  }
  return new Proxy(target, handler)
}
function invariant (key, action) {
  if (key[0] === '_') {
    throw new Error(`Invalid attempt to ${action} private "${key}" property`)
  }
}
```

Usage stays the same, except now access to `target` is completely governed by `proxy` and its mischievous traps. At this point, any `_prop` properties in `target` are completely inaccessible through the proxy, and since `target` can't be accessed directly from outside the `proxied` method, they're sealed off from consumers for good.

You might be tempted to argue that you could achieve the same behavior in ES5 simply by using variables privately scoped to the `proxied` method, without the need for the `Proxy` itself. The big difference is that proxies allow you to “privatize” property access **on different layers**. Imagine an underlying `underling` object that already has several *“private”* properties, which you still access in some other `middletier` module that has intimate knowledge of the internals of `underling`. The `middletier` module could return a `proxied` version of `underling` without having to map the API onto an entirely new object in order to protect those internal variables. Just locking access to any of the “private” properties would suffice!

Here’s a use case on schema validation using proxies.

Schema Validation with Proxies

While, yes, *you could* set up schema validation on the `target` object itself, doing it on a `Proxy` means that you separate the validation concerns from the `target` object, which will go on to live as a **POJO** *(Plain Old JavaScript Object)* happily ever after. Similarly, you can use the proxy as an intermediary for access to many different objects that conform to a schema, without having to rely on prototypal inheritance or ES6 `class` classes.

In the example below, `person` is a plain model object, and we’ve also defined a `validator` object with a `set` trap that will be used as the `handler` for a `proxy` validator of people models. As long as the `person` properties are set through `proxy`, the model invariants will be satisfied according to our validation rules.

```
var validator = {
  set (target, key, value) {
    if (key === 'age') {
      if (typeof value !== 'number' || Number.isNaN(value)) {
        throw new TypeError('Age must be a number')
      }
      if (value <= 0) {
        throw new TypeError('Age must be a positive number')
      }
    }
    return true
  }
}
var person = { age: 27 }
var proxy = new Proxy(person, validator)
proxy.age = 'foo'
// <- TypeError: Age must be a number
proxy.age = NaN
// <- TypeError: Age must be a number
proxy.age = 0
// <- TypeError: Age must be a positive number
proxy.age = 28
console.log(person.age)
// <- 28
```

There’s also a particularly “severe” type of proxies that allows us to completely shut off access to `target` whenever we deem it necessary.

Revocable Proxies

We can use `Proxy.revocable` in a similar way to `Proxy`. The main differences are that the return value will be `{ proxy, revoke }`, and that once `revoke` is called the `proxy` **will throw** on *any operation*. Let’s go back to our pass-through `Proxy` example and make it `revocable`. Note that we’re *not using* the `new` operator here. Calling `revoke()` over and over has no effect.

```
var target = {}
var handler = {}
var {proxy, revoke} = Proxy.revocable(target, handler)
proxy.a = 'b'
console.log(proxy.a)
// <- 'b'
revoke()
revoke()
revoke()
console.log(proxy.a)
// <- TypeError: illegal operation attempted on a revoked proxy
```

This type of `Proxy` is particularly useful because you can now completely cut off access to the `proxy` granted to a consumer. You start by passing of a revocable `Proxy` and keeping around the `revoke` method *(hey, maybe you can use a `WeakMap` for that)*, and when its clear that the consumer shouldn’t have access to `target` anymore, — not even through `proxy` — you `.revoke()` the hell out of their access. *Goodbye consumer!*

Furthermore, since `revoke` is available on the same scope where your `handler` traps live, you could set up **extremely paranoid rules** such as *"if a consumer attempts to access a private property more than once, revoke their `proxy` entirely"*.

Check back tomorrow for the second part of the article about proxies, which discusses `Proxy` *traps* beyond `get` and `set`.

↑ **Back to top**

ES6 Proxy Traps in Depth

Welcome to ES6 – *"Please, not again"* – in Depth. Looking for other ES6 goodness? Refer to A Brief History of ES6 Tooling. Then, make your way through destructuring, template literals, arrow functions, the spread operator and rest parameters, improvements coming to object literals, the new *classes* sugar on top of prototypes, `let`, `const`, and the *"Temporal Dead Zone"*, iterators, generators, Symbols, Maps, WeakMaps, Sets, and WeakSets, and proxies. We'll be discussing *ES6 proxy **traps*** today.

> As I did in previous articles on the series, I would love to point out that you should probably set up Babel and follow along the examples with either a REPL or the `babel-node` CLI and a file. That'll make it so much easier for you to **internalize the concepts** discussed in the series. If you aren't the *"install things on my computer"* kind of human, you might prefer to hop on CodePen and then click on the gear icon for JavaScript – *they have a Babel preprocessor which makes trying out ES6 a breeze.* Another alternative that's also quite useful is to use Babel's online REPL – *it'll show you compiled ES5 code to the right of your ES6 code for quick comparison.*
>
> Note that `Proxy` is harder to play around with as Babel doesn't support it unless the underlying browser has support for it. You can check out the ES6 compatibility table for supporting browsers. At the time of this writing, you can use *Microsoft Edge* or *Mozilla Firefox* to try out `Proxy`. Personally, I'll be verifying my examples using *Firefox*.

Before getting into it, let me *shamelessly ask for your support* if you're enjoying my ES6 in Depth series. Your contributions will go towards helping me keep up with the schedule, server bills, keeping me fed, and maintaining **Pony Foo** as a veritable source of JavaScript goodies.

Thanks for reading that, and let's go into more `Proxy` *traps* now! If you haven't yet, I encourage you to read yesterday's article on the `Proxy` built-in for an introduction to the subject.

Proxy Trap Handlers

An interesting aspect of proxies is how you can use them to intercept just about any interaction with a `target` object – not just `get` or `set` operations. Below are some of the traps you can set up, here's a summary.

- `has` – traps `in` operator
- `deleteProperty` – traps `delete` operator
- `defineProperty` – traps `Object.defineProperty` and declarative alternatives
- `enumerate` – traps `for..in` loops
- `ownKeys` – traps `Object.keys` and related methods
- `apply` – traps *function calls*

We'll bypass `get` and `set`, because we already covered those two yesterday; and there's a few more traps that aren't listed here that will make it into an article published tomorrow. *Stay tuned!*

`has`

You can use `handler.has` to *"hide"* any property you want. It's a trap for the `in` operator. In the `set` trap example we prevented changes and even access to `_`*-prefixed* properties, but unwanted accessors could still ping our proxy to figure out whether these properties are actually there or not. *Like Goldilocks*, we have three options here.

- We can let `key in proxy` *fall through* to `key in target`
- We can `return false` *(or `true`)* – even though `key` **may or may not** actually be there
- We can `throw` an error and deem the question **invalid** in the first place

The last option is quite harsh, and I imagine it being indeed a valid choice in some situations – but you would be acknowledging that the property *(or "property space")* is, in fact, *protected*. It's often best to just smoothly indicate that the property is not `in` the object. Usually, a fall-through case where you just return the result of the `key in target` expression is a good default case to have.

In our example, we probably want to `return false` for properties in the `_` -*prefixed "property space"* and the default of `key in target` for all other properties. This will keep our inaccessible properties well hidden from unwanted visitors.

```
var handler = {
  get (target, key) {
    invariant(key, 'get')
    return target[key]
  },
  set (target, key, value) {
    invariant(key, 'set')
    return true
  },
  has (target, key) {
    if (key[0] === '_') {
      return false
    }
    return key in target
  }
}
function invariant (key, action) {
  if (key[0] === '_') {
    throw new Error(`Invalid attempt to ${action} private "${key}" property`)
  }
}
```

Note how accessing properties through the proxy will now return `false` whenever accessing one of our private properties, with the consumer being none the wiser – completely unaware that we've intentionally hid the property from them.

```
var target = { _prop: 'foo', pony: 'foo' }
var proxy = new Proxy(target, handler)
console.log('pony' in proxy)
// <- true
console.log('_prop' in proxy)
// <- false
console.log('_prop' in target)
// <- true
```

Sure, we could've thrown an exception instead. That'd be useful in situations where attempts to access properties in the private space is seen more of **as a mistake that results in broken modularity** than as a *security concern* in code that aims to be embedded into third party websites.

> It really depends on your use case!

`deleteProperty`

I use the `delete` operator a lot. Setting a property to `undefined` clears its value, but the property is still part of the object. Using the `delete` operator on a property with code like `delete foo.bar` means that the `bar` property will be forever gone from the `foo` object.

```
var foo = { bar: 'baz' }
foo.bar = 'baz'
console.log('bar' in foo)
// <- true
delete foo.bar
console.log('bar' in foo)
// <- false
```

Remember our `set` trap example where we prevented access to `_` -*prefixed* properties? That code had a problem. Even though you couldn't change the value of `_prop`, you could remove the property entirely using the `delete` operator. Even through the `proxy` object!

```
var target = { _prop: 'foo' }
var proxy = new Proxy(target, handler)
console.log('_prop' in proxy)
// <- true
delete proxy._prop
console.log('_prop' in proxy)
// <- false
```

You can use `handler.deleteProperty` to prevent a `delete` operation from working. Just like with the `get` and `set` traps, throwing in the `deleteProperty` trap will be enough to prevent the deletion of a property.

```
var handler = {
  get (target, key) {
    invariant(key, 'get')
    return target[key]
  },
  set (target, key, value) {
    invariant(key, 'set')
    return true
  },
  deleteProperty (target, key) {
    invariant(key, 'delete')
    return true
  }
}
function invariant (key, action) {
  if (key[0] === '_') {
    throw new Error(`Invalid attempt to ${action} private "${key}" property`)
  }
}
```

If we run the exact same piece of code we tried earlier, we'll run into the exception while trying to delete `_prop` from the `proxy`.

```
var target = { _prop: 'foo' }
var proxy = new Proxy(target, handler)
console.log('_prop' in proxy)
// <- true
delete proxy._prop
// <- Error: Invalid attempt to delete private "_prop" property
```

Deleting properties in your `_private` property space is no longer possible for consumers interacting with `target` through the `proxy`.

`defineProperty`

We typically use `Object.defineProperty(obj, key, descriptor)` in two types of situations.

1. When we wanted to ensure cross-browser support of *getters and setters*
2. Whenever we want to define a custom property accessor

Properties added by hand are read-write, they are deletable, and they are enumerable. Properties added through `Object.defineProperty`, *in contrast*, default to being *read-only*, *write-only*, *non-deletable*, and *non-enumerable* – in other words, the property starts off being completely **immutable**. You can customize these aspects of the property descriptor, and you can find them below – alongside with their *default values* when using `Object.defineProperty`.

- `configurable: false` disables most changes to the property descriptor and makes the property *undeletable*
- `enumerable: false` hides the property from `for..in` loops and `Object.keys`
- `value: undefined` is the initial value for the property
- `writable: false` makes the property value immutable
- `get: undefined` is a method that acts as the getter for the property
- `set: undefined` is a method that receives the new `value` and updates the property's `value`

Note that when defining a property you'll have to choose between using `value` and `writable` or `get` and `set`. When choosing the former you're configuring a *data descriptor* – this is the kind you get when declaring properties like `foo.bar = 'baz'`, it has a `value` and it *may or may not* be `writable`. When choosing the latter you're creating an *accessor descriptor*, which is entirely defined by the methods you can use to `get()` or `set(value)` the value for the property.

The code sample below shows how property descriptors are completely different depending on whether you went for the declarative option or through the programmatic API.

```
var target = {}
target.foo = 'bar'
console.log(Object.getOwnPropertyDescriptor(target, 'foo'))
// <- { value: 'bar', writable: true, enumerable: true, configurable: true }
Object.defineProperty(target, 'baz', { value: 'ponyfoo' })
console.log(Object.getOwnPropertyDescriptor(target, 'baz'))
// <- { value: 'ponyfoo', writable: false, enumerable: false, configurable: false }
```

Now that we went over a blitzkrieg overview of `Object.defineProperty`, we can move on to the trap.

The `handler.defineProperty` trap can be used to intercept calls to `Object.defineProperty`. You get the `key` and the `descriptor` being used. The example below completely prevents the addition of properties through the `proxy`. How cool is it that this intercepts the declarative `foo.bar = 'baz'` property declaration alternative as well? *Quite cool!*

```
var handler = {
  defineProperty (target, key, descriptor) {
    return false
  }
}
var target = {}
var proxy = new Proxy(target, handler)
proxy.foo = 'bar'
// <- TypeError: proxy defineProperty handler returned false for property '"foo"'
```

If we go back to our *"private properties"* example, we could use the `defineProperty` trap to prevent the creation of private properties through the proxy. We'll reuse the `invariant` method we had to `throw` on attempts to define a property in the *"private `_`-prefixed space"*, and that's it.

```
var handler = {
  defineProperty (target, key, descriptor) {
    invariant(key, 'define')
    return true
  }
}
function invariant (key, action) {
  if (key[0] === '_') {
    throw new Error(`Invalid attempt to ${action} private "${key}" property`)
  }
}
```

You could then try it out on a `target` object, setting properties with a `_` prefix will now throw an error. You could make it fail silently by returning `false` – *depends on your use case!*

```
var target = {}
var proxy = new Proxy(target, handler)
proxy._foo = 'bar'
// <- Error: Invalid attempt to define private "_foo" property
```

Your `proxy` is now safely hiding `_private` properties behind a trap that guards them from definition through either `proxy[key] = value` or `Object.defineProperty(proxy, key, { value })` – *pretty amazing!*

`enumerate`

The `handler.enumerate` method can be used to trap `for..in` statements. With `has` we could prevent `key in proxy` from returning `true` for any property in our underscored private space, but what about a `for..in` loop? Even though our `has` trap hides the property from a `key in proxy` check, the consumer will accidentally stumble upon the property when using a `for..in` loop!

```
var handler = {
  has (target, key) {
    if (key[0] === '_') {
      return false
    }
    return key in target
  }
}
var target = { _prop: 'foo' }
var proxy = new Proxy(target, handler)
for (let key in proxy) {
  console.log(key)
  // <- '_prop'
}
```

We can use the `enumerate` trap to return an *iterator* that'll be used instead of the *enumerable properties* found in `proxy` during a `for..in` loop. The returned iterator must conform to the iterator protocol, such as the iterators returned from any `Symbol.iterator` method. Here's a possible implementation of such a `proxy` that would return the output of `Object.keys` minus the properties found in our *private space*.

```
var handler = {
  has (target, key) {
```

```
    if (key[0] === '_') {
      return false
    }
    return key in target
  },
  enumerate (target) {
    return Object.keys(target).filter(key => key[0] !== '_')[Symbol.iterator]()
  }
}
var target = { pony: 'foo', _bar: 'baz', _prop: 'foo' }
var proxy = new Proxy(target, handler)
for (let key in proxy) {
  console.log(key)
  // <- 'pony'
}
```

Now your private properties are hidden from those prying `for..in` eyes!

`ownKeys`

The `handler.ownKeys` method may be used to return an `Array` of properties that will be used as a result for `Reflect.ownKeys()` – *it should include all properties of `target` (enumerable or not, and symbols too).* A default implementation, *as seen below*, could just call `Reflect.ownKeys` on the proxied `target` object. Don't worry, we'll get to the `Reflect` built-in later in the es6-in-depth series.

```
var handler = {
  ownKeys (target) {
    return Reflect.ownKeys(target)
  }
}
```

Interception wouldn't affect the output of `Object.keys` in this case.

```
var target = {
  _bar: 'foo',
  _prop: 'bar',
  [Symbol('secret')]: 'baz',
  pony: 'ponyfoo'
}
var proxy = new Proxy(target, handler)
for (let key of Object.keys(proxy)) {
  console.log(key)
  // <- '_bar'
  // <- '_prop'
  // <- 'pony'
}
```

Do note that the `ownKeys` interceptor is used during all of the following operations.

- `Object.getOwnPropertyNames()` – just non-symbol properties
- `Object.getOwnPropertySymbols()` – just symbol properties
- `Object.keys()` – just non-symbol enumerable properties
- `Reflect.ownKeys()` – we'll get to `Reflect` later in the series!

In the use case where we want to shut off access to a property space prefixed by `_`, we could take the output of `Reflect.ownKeys(target)` and filter that.

```
var handler = {
  ownKeys (target) {
    return Reflect.ownKeys(target).filter(key => key[0] !== '_')
  }
}
```

If we now used the `handler` in the snippet above to pull the object keys, we'll just find the properties in the public, non `_` -prefixed space.

```
var target = {
  _bar: 'foo',
  _prop: 'bar',
  [Symbol('secret')]: 'baz',
  pony: 'ponyfoo'
}
var proxy = new Proxy(target, handler)
for (let key of Object.keys(proxy)) {
```

```
  console.log(key)
  // <- 'pony'
}
```

Symbol iteration wouldn't be affected by this as `sym[0]` yields `undefined` – *and in any case decidedly not* `'_'`.

```
var target = {
  _bar: 'foo',
  _prop: 'bar',
  [Symbol('secret')]: 'baz',
  pony: 'ponyfoo'
}
var proxy = new Proxy(target, handler)
for (let key of Object.getOwnPropertySymbols(proxy)) {
  console.log(key)
  // <- Symbol(secret)
}
```

We were able to hide properties prefixed with `_` from key enumeration while leaving symbols and other properties unaffected.

`apply`

The `handler.apply` method is quite interesting. You can use it as a trap on any invocation of `proxy`. All of the following will go through the `apply` trap for your proxy.

```
proxy(1, 2)
proxy(...args)
proxy.call(null, 1, 2)
proxy.apply(null, [1, 2])
```

The `apply` method takes three arguments.

- `target` – the function being proxied
- `ctx` – the context passed as `this` to `target` when applying a call
- `args` – the arguments passed to `target` when applying the call

A naïve implementation might look like `target.apply(ctx, args)`, but below we'll be using `Reflect.apply(...arguments)`. We'll dig deeper into the `Reflect` built-in later in the series. For now, just think of them as equivalent, and take into account that the value returned by the `apply` trap is also going to be used as the result of a function call through `proxy`.

```
var handler = {
  apply (target, ctx, args) {
    return Reflect.apply(...arguments)
  }
}
```

Besides the obvious *"being able to log all parameters of every function call for* `proxy`", this trap can be used for parameter balancing and to tweak the results of a function call without changing the method itself – *and without changing the calling code either.*

The example below proxies a `sum` method through a `twice` trap handler that doubles the results of `sum` without affecting the code around it other than using the `proxy` instead of the `sum` method directly.

```
var twice = {
  apply (target, ctx, args) {
    return Reflect.apply(...arguments) * 2
  }
}
function sum (left, right) {
  return left + right
}
var proxy = new Proxy(sum, twice)
console.log(proxy(1, 2))
// <- 6
console.log(proxy(...[3, 4]))
// <- 14
console.log(proxy.call(null, 5, 6))
// <- 22
console.log(proxy.apply(null, [7, 8]))
// <- 30
```

Naturally, calling `Reflect.apply` on the `proxy` will be caught by the `apply` *trap* as well.

```
Reflect.apply(proxy, null, [9, 10])
// <- 38
```

> What else would you use `handler.apply` for?

Tomorrow I'll publish the last article on `Proxy` – *Promise!* – It'll include the remaining *trap* handlers, such as `construct` and `getPrototypeOf`.

↑ **Back to top**

More ES6 Proxy Traps in Depth

Hey there! This is ES6 – *"Traps? Again?"* – in Depth. Looking for other ES6 goodness? Refer to A Brief History of ES6 Tooling. Then, make your way through destructuring, template literals, arrow functions, the spread operator and rest parameters, improvements coming to object literals, the new *classes* sugar on top of prototypes, `let`, `const`, and the *"Temporal Dead Zone"*, iterators, generators, Symbols, Maps, WeakMaps, Sets, and WeakSets, proxies, and proxy traps. We'll be discussing about ***more** ES6 proxy **traps*** today.

> As I did in previous articles on the series, I would love to point out that you should probably set up Babel and follow along the examples with either a REPL or the `babel-node` CLI and a file. That'll make it so much easier for you to **internalize the concepts** discussed in the series. If you aren't the *"install things on my computer"* kind of human, you might prefer to hop on CodePen and then click on the gear icon for JavaScript – *they have a Babel preprocessor which makes trying out ES6 a breeze.* Another alternative that's also quite useful is to use Babel's online REPL – *it'll show you compiled ES5 code to the right of your ES6 code for quick comparison.*
>
> Note that `Proxy` is harder to play around with as Babel doesn't support it unless the underlying browser has support for it. You can check out the ES6 compatibility table for supporting browsers. At the time of this writing, you can use *Microsoft Edge* or *Mozilla Firefox* to try out `Proxy`. Personally, I'll be verifying my examples using *Firefox*.

Before getting into it, let me *shamelessly ask for your support* if you're enjoying my ES6 in Depth series. Your contributions will go towards helping me keep up with the schedule, server bills, keeping me fed, and maintaining **Pony Foo** as a veritable source of JavaScript goodies.

Thanks for reading that, and let's go into even more `Proxy` *traps* now! If you haven't yet, I encourage you to read the previous article on the `Proxy` built-in for an introduction to the subject and the one about the first few traps I covered.

But Wait! There's More... *(Proxy Trap Handlers)*

This article covers all the trap handlers that weren't covered by the two previous articles on proxies. For the most part, the traps that we discussed yesterday had to do with property manipulation, while the first five traps we'll dig into today have mostly to do with the object being proxied *itself*. The last two have to do with properties once again – but they're a bit more involved than yesterday's traps, which were much easier to "fall into" *(the trap – muahaha)* in your everyday code.

- `construct` – traps usage of the `new` operator
- `getPrototypeOf` – traps internal calls to `[[GetPrototypeOf]]`
- `setPrototypeOf` – traps calls to `Object.setPrototypeOf`
- `isExtensible` – traps calls to `Object.isExtensible`
- `preventExtensions` – traps calls to `Object.preventExtensions`
- `getOwnPropertyDescriptor` – traps calls to `Object.getOwnPropertyDescriptor`

`construct`

You can use the `handler.construct` method to trap usage of the `new` operator. Here's a quick *"default implementation"* that doesn't alter the behavior of `new` at all. *Remember our friend the spread operator?*

```
var handler = {
  construct (target, args) {
    return new target(...args)
```

```
  }
}
```

If you use the `handler` options above, the `new` behavior you're already used to would remain unchanged. That's great because it means whatever you're trying to accomplish you can still fall back to the **default behavior** – *and that's always important.*

```
function target (a, b, c) {
  this.a = a
  this.b = b
  this.c = c
}
var proxy = new Proxy(target, handler)
console.log(new proxy(1,2,3))
// <- { a: 1, b: 2, c: 3 }
```

Obvious use cases for `construct` traps include data massaging in the arguments, doing things that should always be done around a call to `new proxy()`, logging and tracking object creation, and swapping implementations entirely. Imagine a proxy like the following in situations where you have inheritance *"branching"*.

```
class Automobile {}
class Car extends Automobile {}
class SurveillanceVan extends Automobile {}
class SUV extends Automobile {}
class SportsCar extends Car {}
function target () {}
var handler = {
  construct (target, args) {
    var [status] = args
    if (status === 'nsa') {
      return new SurveillanceVan(...args)
    }
    if (status === 'single') {
      return new SportsCar(...args)
    }
    return new SUV(...args) // family
  }
}
```

Naturally, you could've used a regular method for the branching part, but using the `new` operator also makes sense in these types of situations, as you'll end up creating a new object in all code branches anyways.

```
console.log(new proxy('nsa').constructor.name)
// <- `SurveillanceVan`
```

The most common use case for `construct` traps yet may be something simpler, and that's extending the `target` object right after creation, – *and before anything else happens* – in such a way that it better supports the `proxy` gatekeeper. You might have to add a `proxied` flag to the `target` object, or something akin to that.

`getPrototypeOf`

You can use the `handler.getPrototypeOf` method as a trap for all of the following.

- `Object.prototype.__proto__` property
- `Object.prototype.isPrototypeOf()` method
- `Object.getPrototypeOf()` method
- `Reflect.getPrototypeOf()` method
- `instanceof` operator

You could use this *trap* to make an object pretend it's an `Array`, when accessed through the proxy. However, note that that on its own isn't sufficient for the `proxy` to be an actual `Array`.

```
var handler = {
  getPrototypeOf: target => Array.prototype
}
var target = {}
var proxy = new Proxy(target, handler)
console.log(proxy instanceof Array)
// <- true
console.log(proxy.push)
// <- undefined
```

Naturally, you could keep on patching your `proxy` until you get the behavior you want. In this case, you may want to use a `get` trap to mix the `Array.prototype` with the actual back-end `target`. Whenever a property isn't found on the `target`, we'll use reflection to look it up on `Array.prototype`. It turns out, this is **good enough** for most operations.

```
var handler = {
  getPrototypeOf: target => Array.prototype,
  get (target, key) {
    return Reflect.get(target, key) || Reflect.get(Array.prototype, key)
  }
}
var target = {}
var proxy = new Proxy(target, handler)
console.log(proxy.push)
// <- function push () { [native code] }
proxy.push('a', 'b')
console.log(proxy)
// <- { 0: 'a', 1: 'b', length: 2 }
```

I definitely see some advanced use cases for `getPrototypeOf` traps in the future, but it's too early to tell what patterns may come out of it.

`setPrototypeOf`

The `Object.setPrototypeOf` method does exactly what its name conveys: it sets the prototype of an object to a reference to another object. It's considered the proper way of setting the prototype as opposed to using `__proto__` which is a legacy feature – *and now standarized as such.*

You can use the `handler.setPrototypeOf` method to set up a *trap* for `Object.setPrototypeOf`. The snippet of code shown below doesn't alter the default behavior of changing a prototype to the value of `proto`.

```
var handler = {
  setPrototypeOf (target, proto) {
    Object.setPrototypeOf(target, proto)
  }
}
var proto = {}
var target = function () {}
var proxy = new Proxy(target, handler)
proxy.setPrototypeOf(proxy, proto)
console.log(proxy.prototype === proto)
// <- true
```

The field for `setPrototypeOf` is pretty open. You could simply not call `Object.setPrototypeOf` and the trap would sink the call into a *no-op*. You could `throw` an exception making the failure more explicit – for instance if you deem the new prototype to be invalid or you don't want consumers pulling the rug from under your feet.

This is a nice *trap* to have if you want proxies to have limited access to what they can do with your `target` object. I would definitely implement a trap like the one below if I had any security concerns at all in a proxy I'm passing away to third party code.

```
var handler = {
  setPrototypeOf (target, proto) {
    throw new Error('Changing the prototype is forbidden')
  }
}
var proto = {}
var target = function () {}
var proxy = new Proxy(target, handler)
proxy.setPrototypeOf(proxy, proto)
// <- Error: Changing the prototype is forbidden
```

Then again, you may want to fail silently with no error being thrown at all if you'd rather confuse the consumer – and that may just make them go away.

`isExtensible`

The `handler.isExtensible` method can be mostly used for logging or auditing calls to `Object.isExtensible`. This *trap* is subject to a harsh invariant that puts a hard limit to what you can do with it.

> If `Object.isExtensible(proxy) !== Object.isExtensible(target)`, then a `TypeError` is thrown.

You could use the `handler.isExtensible` trap to `throw` if you don't want consumers to know whether the original object is extensible or not, but there seem to be limited situations that would warrant such an **incarnation of evil**. For completeness' sake, the piece of code below shows a trap for `isExtensible` that throws errors every once in a while, but otherwise behaves as expected.

```
var handler = {
  isExtensible (target) {
    if (Math.random() >
0.1) {
      throw new Error('gotta love sporadic obscure errors!')
    }
    return Object.isExtensible(target)
  }
}
var target = {}
var proxy = new Proxy(target, handler)
console.log(Object.isExtensible(proxy))
// <- true
console.log(Object.isExtensible(proxy))
// <- true
console.log(Object.isExtensible(proxy))
// <- true
console.log(Object.isExtensible(proxy))
// <- Error: gotta love sporadic obscure errors!
```

While this *trap* is nearly useless other than for auditing purposes and to cover all your bases, the *hard-to-break* invariant makes sense because there's also the `preventExtensions` *trap*. **That one is a little bit more useful!**

preventExtensions

You can use `handler.preventExtensions` to *trap* the `Object.preventExtensions` method. When extensions are prevented on an object, new properties can't be added any longer – *it can't be extended*.

Imagine a scenario where you want to selectively be able to `preventExtensions` on some objects – but not all of them. In this scenario, you could use a `WeakSet` to keep track of the objects that should be extensible. If an object is in the set, then the `preventExtensions` *trap* should be able to capture those requests and discard them. The snippet below does exactly that. Note that the *trap* always returns the opposite of `Object.isExtensible(target)`, because it should report whether the object has been made non-extensible.

```
var mustExtend = new WeakSet()
var handler = {
  preventExtensions (target) {
    if (!mustExtend.has(target)) {
      Object.preventExtensions(target)
    }
    return !Object.isExtensible(target)
  }
}
```

Now that we've set up the `handler` and our `WeakSet`, we can create a back-end object, a `proxy`, and add the back-end to our set. Then, you can try `Object.preventExtensions` on the proxy and you'll notice it fails to prevent extensions to the object. This is the intended behavior as the `target` can be found in the `mustExtend` set.

```
var target = {}
var proxy = new Proxy(target, handler)
mustExtend.add(target)
Object.preventExtensions(proxy)
// <- TypeError: proxy preventExtensions handler returned false
```

If we removed the `target` from the `mustExtend` set before calling `Object.preventExtensions`, then `target` would be made non-extensible as originally intended.

```
var target = {}
var proxy = new Proxy(target, handler)
mustExtend.add(target)
mustExtend.delete(target)
Object.preventExtensions(proxy)
console.log(Object.isExtensible(proxy))
// <- false
```

Naturally, you could use this distinction to prevent `proxy` consumers from making the proxy non-extensible in cases where that could lead to undesired behavior. In most cases, you probably won't have to deal with this *trap*, though. That's because you're usually going to be working with the *back-end* `target` for the most part, and not so much with the `proxy` object itself.

`getOwnPropertyDescriptor`

You can use the `handler.getOwnPropertyDescriptor` method as a trap for `Object.getOwnPropertyDescriptor`. It may return a property descriptor, such as the result from `Object.getOwnPropertyDescriptor(target, key)`; or `undefined`, signaling that the property doesn't exist. As usual, you also have the third option of throwing an exception, aborting the operation entirely.

If we go back to our canonical *"private property space"* example, we could implement a *trap* such as the one seen below to prevent consumers from learning about property descriptors of private properties.

```
var handler = {
  getOwnPropertyDescriptor (target, key) {
    invariant(key, 'get property descriptor for')
    return Object.getOwnPropertyDescriptor(target, key)
  }
}
function invariant (key, action) {
  if (key[0] === '_') {
    throw new Error(`Invalid attempt to ${action} private "${key}" property`)
  }
}
var target = {}
var proxy = new Proxy(target, handler)
Object.getOwnPropertyDescriptor(proxy, '_foo')
// <- Error: Invalid attempt to get property descriptor for private "_foo" property
```

The problem with that approach is that you're effectively telling consumers that properties with the `_` prefix are somehow off-limits. It might be best to conceal them entirely by returning `undefined`. This way, your private properties will behave no differently than properties that are *actually absent* from the `target` object.

```
var handler = {
  getOwnPropertyDescriptor (target, key) {
    if (key[0] === '_') {
      return
    }
    return Object.getOwnPropertyDescriptor(target, key)
  }
}
var target = { _foo: 'bar', baz: 'tar' }
var proxy = new Proxy(target, handler)
console.log(Object.getOwnPropertyDescriptor(proxy, 'wat'))
// <- undefined
console.log(Object.getOwnPropertyDescriptor(proxy, '_foo'))
// <- undefined
console.log(Object.getOwnPropertyDescriptor(proxy, 'baz'))
// <- { value: 'tar', writable: true, enumerable: true, configurable: true }
```

Usually when you're trying to hide things it's best to have them try and behave as if they fell in some other category than the category they're actually in. Throwing, however, just sends the *"there's something sketchy here, but we can't quite tell you what that is..."* message – and the consumer will eventually find out why that is.

> Keep in mind that if debugging concerns outweight security concerns, you probably should go for the `throw` statement.

Conclusions

This has certainly been fun. I now have a much better understanding of what proxies can do for me, and I think they'll be an instant hit once ES6 starts gaining more traction. I for one can't be more excited about them becoming well-supported in more browsers soon. I wouldn't hold out my hopes about proxies for Babel, as many of the traps are **ridiculously hard** *(or downright impossible)* to implement in ES5.

As we've learned over the last few days, there's **a myriad use cases** for proxies. Off the top of my head, we can use `Proxy` for all of the following.

- Add validation rules – *and enforce them* – on plain old JavaScript objects
- Keep track of every interaction that goes through a proxy
- Decorate objects without changing them at all
- Make certain properties on an object completely invisible to the consumer of a proxy

- Revoke access *at will* when the consumer should no longer be able to use a proxy
- Modify the arguments passed to a proxied method
- Modify the result produced by a proxied method
- Prevent deletion of specific properties through the proxy
- Prevent new definitions from succeeding, according to the desired property descriptor
- Shuffle arguments around in a constructor
- Return a result other than the object being `new` -ed up in a constructor
- Swap out the prototype of an object for something else

I can say without a shadow of a doubt that there's **hundreds more of use cases** for proxies. I'm sure many libraries will adopt a pattern we've discussed here in the series where a *"back-end"* `target` object is created and used for storage purposes but the consumer is only provided with a *"front-end"* `proxy` object with *limited and audited interaction* with the back-end.

> What would *you* use `Proxy` for?

Meet me tomorrow at… say – *same time?* We can talk about `Reflect` then.

Back to top

ES6 Reflection in Depth

Oh hey – I was just casually getting ready, didn't see you there! Welcome to another edition of ES6 – *"Oh. Good. We survived traps"* – in Depth. Never heard of it? Refer to A Brief History of ES6 Tooling. Then, make your way through destructuring, template literals, arrow functions, the spread operator and rest parameters, improvements coming to object literals, the new *classes* sugar on top of prototypes, `let`, `const`, and the *"Temporal Dead Zone"*, iterators, generators, Symbols, Maps, WeakMaps, Sets, and WeakSets, proxies, proxy traps, and more proxy traps. We'll be touching on the `Reflect` API today.

> As I did in previous articles on the series, I would love to point out that you should probably set up Babel and follow along the examples with either a REPL or the `babel-node` CLI and a file. That'll make it so much easier for you to **internalize the concepts** discussed in the series. If you aren't the *"install things on my computer"* kind of human, you might prefer to hop on CodePen and then click on the gear icon for JavaScript – *they have a Babel preprocessor which makes trying out ES6 a breeze.* Another alternative that's also quite useful is to use Babel's online REPL – *it'll show you compiled ES5 code to the right of your ES6 code for quick comparison.*

Before getting into it, let me *shamelessly ask for your support* if you're enjoying my ES6 in Depth series. Your contributions will go towards helping me keep up with the schedule, server bills, keeping me fed, and maintaining **Pony Foo** as a veritable source of JavaScript goodies.

Thanks for reading that, and let's go into `Reflect`. I suggest you read the articles on proxies: `Proxy` built-in, traps, and more traps. These will help you wrap your head around some of the content we'll go over today.

Why Reflection?

Many statically typed languages have long offered a reflection API *(such as Python or C#)*, whereas JavaScript hardly has a need for a reflection API – it *already being* a dynamic language. The introduction of ES6 features a few **new extensibility points** where the developer gets access to *previously internal aspects* of the language – yes, I'm talking about `Proxy`.

You could argue that **JavaScript already has reflection features in ES5**, even though they weren't ever called that by either the specification or the community. Methods like `Array.isArray`, `Object.getOwnPropertyDescriptor`, and even `Object.keys` are classical examples of what you'd find **categorized as reflection** in other languages. The `Reflect` built-in is, going forward, is going to house future methods in the category. That makes a lot of sense, right? Why would you have **super reflectiony static methods** like `getOwnPropertyDescriptor` *(or even `create`)* in `Object`? After all, `Object` is meant to be a base prototype, and not so much a repository of reflection methods. Having a dedicated interface that exposes most reflection methods makes more sense.

`Reflect`

We've mentioned the `Reflect` object in passing the past few days. Much like `Math`, `Reflect` is a static object you can't `new` up nor *call*, and all of its methods are static. The _traps in ES6 proxies *(covered here and here)* are **mapped one-to-one** to the `Reflect` API. For every *trap*, there's a matching reflection method in `Reflect`.

The reflection API in JavaScript has *a number of benefits* that are worth examining.

Return Values in `Reflect` vs Reflection Through `Object`

The `Reflect` equivalents to reflection methods on `Object` also provide more **meaningful** *return values*. For instance, the `Reflect.defineProperty` method returns a boolean value indicating whether the property was successfully defined. Meanwhile, its `Object.defineProperty` counterpart returns the object it got as its first argument — *not very **useful***.

To illustrate, below is a code snippet showing how to verify `Object.defineProperty` worked.

```
try {
  Object.defineProperty(target, 'foo', { value: 'bar' })
  // yay!
} catch (e) {
  // oops.
}
```

As opposed to a *much more natural* `Reflect.defineProperty` experience.

```
var yay = Reflect.defineProperty(target, 'foo', { value: 'bar' })
if (yay) {
  // yay!
} else {
  // oops.
}
```

This way we avoided a `try` / `catch` block and made our code a little more maintainable in the process.

Keyword Operators as First Class Citizens

Some of these reflection methods provide programmatic alternatives of doing things that were previously only possible through keywords. For example, `Reflect.deleteProperty(target, key)` is equivalent to the `delete target[key]` expression. Before ES6, if you wanted a method call to result in a `delete` call, you'd have to create a dedicated utility method that wrapped `delete` on your behalf.

```
var target = { foo: 'bar', baz: 'wat' }
delete target.foo
console.log(target)
// <- { baz: 'wat' }
```

Today, with ES6, you already have such a method in `Reflect.deleteProperty`.

```
var target = { foo: 'bar', baz: 'wat' }
Reflect.deleteProperty(target, 'foo')
console.log(target)
// <- { baz: 'wat' }
```

Just like `deleteProperty`, there's a few other methods that make it easy to do other things too.

Easier to mix `new` with Arbitrary Argument Lists

In ES5, this is a hard problem: How do you create a `new Foo` passing an arbitrary number of arguments? You can't do it directly, and it's super verbose if you need to do it anyways. You have to create an intermediary object that gets passed the arguments as an `Array`. Then you have *that* object's constructor return the result of applying the constructor of the object you originally intended to `.apply`. Straightforward, right? — *What do you mean **no**?*

```
var proto = Dominus.prototype
Applied.prototype = proto
function Applied (args) {
  return Dominus.apply(this, args)
}
function apply (a) {
  return new Applied(a)
}
```

Using `apply` is actually easy, thankfully.

```
apply(['.foo', '.bar'])
apply.call(null, '.foo', '.bar')
```

But that was *insane*, right? **Who does that?** Well, in ES5, everyone who has a valid reason to do it! Luckily ES6 has less insane approaches to this problem. One of them is simply to use the spread operator.

```
new Dominus(...args)
```

Another alternative is to go the `Reflect` route.

```
Reflect.construct(Dominus, args)
```

Both of these are tremendously simpler than what I had to do in the `dominus` codebase.

Function Application, The Right Way

In ES5 if we want to call a method with an arbitrary number of arguments, we can use `.apply` passing a `this` context and our arguments.

```
fn.apply(ctx, [1, 2, 3])
```

If we fear `fn` might shadow `apply` with a property of their own, we can rely on a safer but way more verbose alternative.

```
Function.prototype.apply.call(fn, ctx, [1, 2, 3])
```

In ES6, you can use spread as an alternative to `.apply` for an arbitrary number of arguments.

```
fn(...[1, 2, 3])
```

That doesn't solve your problems when you need to define a `this` context, though. You could go back to the `Function.prototype` way but that's *way* too verbose. Here's how `Reflect` can help.

```
Reflect.apply(fn, ctx, args)
```

Naturally, one of the most fitting use cases for `Reflect` API methods is default behavior in `Proxy` traps.

Default Behavior in `Proxy` Traps

We've already talked about how *traps* are mapped one-to-one to `Reflect` methods. We haven't yet touched on the fact that their interfaces match as well. That is to say, *both their arguments and their return values match.* In code, this means you could do something like this to get the default `get` *trap* behavior in your proxy handlers.

```
var handler = {
  get () {
    return Reflect.get(...arguments)
  }
}
var target = { a: 'b' }
var proxy = new Proxy(target, handler)
console.log(proxy.a)
// <- 'b'
```

There is, in fact, nothing stopping you from making that `handler` even simpler. Of course, at this point you'd be better off leaving the *trap* out entirely.

```
var handler = {
  get: Reflect.get
}
```

The important take-away here is that you could set up a trap in your proxy handlers, wire up some custom functionality that ends up throwing or logging a console statement, and then in the default case you could just use the one-liner recipe found below.

```
return Reflect[trapName](...arguments)
```

Certainly puts me at ease when it comes to demystifying `Proxy`.

Lastly, There's `__proto__`

Yesterday we talked about how the legacy `__proto__` is part of the ES6 specification but still strongly advised against and how you should use `Object.setPrototypeOf` and `Object.getPrototypeOf` instead. Turns out, there's also `Reflect` counterparts to those methods you could use. Think of these methods as *getter and setters* for `__proto__` but without the cross-browser discrepancies.

> I wouldn't just hop onto the "`setPrototypeOf` all the things" bandwagon just yet. In fact, I hope there never is a train pulling that wagon to begin with.

↑ **Back to top**

ES6 Number Improvements in Depth

Hey there! Glad you're here in time for ES6 – *"Back to School"* – in Depth. Never heard of it? Refer to A Brief History of ES6 Tooling. Then, make your way through destructuring, template literals, arrow functions, the spread operator and rest parameters, improvements coming to object literals, the new *classes* sugar on top of prototypes, `let`, `const`, and the *"Temporal Dead Zone"*, iterators, generators, Symbols, Maps, WeakMaps, Sets, and WeakSets, proxies, proxy traps, more proxy traps, and reflection. Today we'll learn about `Number` improvements.

> As I did in previous articles on the series, I would love to point out that you should probably set up Babel and follow along the examples with either a REPL or the `babel-node` CLI and a file. That'll make it so much easier for you to **internalize the concepts** discussed in the series. If you aren't the *"install things on my computer"* kind of human, you might prefer to hop on CodePen and then click on the gear icon for JavaScript – *they have a Babel preprocessor which makes trying out ES6 a breeze.* Another alternative that's also quite useful is to use Babel's online REPL – *it'll show you compiled ES5 code to the right of your ES6 code for quick comparison.*

Before getting into it, let me *shamelessly ask for your support* if you're enjoying my ES6 in Depth series. Your contributions will go towards helping me keep up with the schedule, server bills, keeping me fed, and maintaining **Pony Foo** as a veritable source of JavaScript goodies.

Thanks for reading that, and let's go into `Number` improvements. These changes don't really depend on anything we've covered so far – *although I would strongly recommend you skim over articles in the series if you haven't done so yet.* Time to dig into `Number`.

`Number` Improvements in ES6

There's a number of changes coming to `Number` in ES6 – *see what I did there?* First off, let's raise the curtain with a summary of the features we'll be talking about. We'll go over all of the following changes to `Number` today.

- Binary and Octal Literals – *using `0b` and `0o`*
- `Number.isNaN`
- `Number.isFinite`
- `Number.parseInt`
- `Number.parseFloat`
- `Number.isInteger`
- `Number.EPSILON`
- `Number.MAX_SAFE_INTEGER`
- `Number.MIN_SAFE_INTEGER`
- `Number.isSafeInteger`

Binary and Octal Literals

Before ES6, your best bet when it comes to binary representation of integers was to just pass them to `parseInt` with a radix of `2`.

```
parseInt('101', 2)
// <- 5
```

In ES6 you could also use the `0b` prefix to represent binary integer literals. You could also use `0B` but I suggest you stick with the lower-case option.

```
console.log(0b001)
// <- 1
console.log(0b010)
// <- 2
console.log(0b011)
// <- 3
console.log(0b100)
// <- 4
```

Same goes for octal literals. In ES3, `parseInt` interpreted strings of digits starting with a `0` as an octal value. That meant things got weird quickly when you forgot to specify a radix of `10` – *and that soon became a best practice.*

```
parseInt('01')
// <- 1
parseInt('08')
// <- 0
parseInt('8')
// <- 8
```

When ES5 came around, it got rid of the octal interpretation in `parseInt` – although **it's still recommended** you specify a `radix` for backwards compatibility purposes. If you actually wanted octal, you could get those using a radix of `8`, anyways.

```
parseInt('100', '8')
// <- 64
```

When it comes to ES6, you can now use the `0o` prefix for octal literals. You could also use `0O`, but that's going to look very confusing in some typefaces, so I suggest you stick with the `0o` notation.

```
console.log(0o010)
// <- 8
console.log(0o100)
// <- 64
```

Keep in mind that octal literals aren't actually going to crop up everywhere in your front-end applications anytime soon, so you shouldn't worry too much about the seemingly odd choice *(font clarity wise)* of a `0o` prefix. Besides, most of us use editors that have no trouble at all differentiating between `0o`, `0O`, `00`, `OO`, and `oo`.

```
`0o`, `0O`, `00`, `OO`, and `oo`
```

If you're now perplexed and left wondering *"what about hexadecimal?"*, don't you worry, those were already part of the language in ES5, and you can still use them. The prefix for literal *hexadecimal* notation is either `0x`, or `0X`.

```
console.log(0x0ff)
// <- 255
console.log(0xf00)
// <- 3840
```

Enough with number literals, let's talk about something else. The first four additions to `Numer` that we'll be discussing – `Number.isNaN`, `Number.isFinite`, `Number.parseInt`, and `Number.parseFloat` – already existed in ES5, but in the global namespace. In addition, the methods in `Number` are slightly different in that they don't coerce non-numeric values into numbers before producing a result.

`Number.isNaN`

This method is almost identical to ES5 global `isNaN` method. `Number.isNaN` returns whether the provided `value` equals `NaN`. This is a very different question from *"is this not a number?"*.

The snippet shown below quickly shows that anything that's not `NaN` when passed to `Number.isNaN` will return `false`, while passing `NaN` into it will yield `true`.

```
Number.isNaN(123)
// <- false, integers are not NaN
Number.isNaN(Infinity)
// <- false, Infinity is not NaN
Number.isNaN('ponyfoo')
// <- false, 'ponyfoo' is not NaN
Number.isNaN(NaN)
// <- true, NaN is NaN
Number.isNaN('pony'/'foo')
// <- true, 'pony'/'foo' is NaN, NaN is NaN
```

The ES5 `global.isNaN` method, in contrast, casts non-numeric values passed to it *before evaluating them against `NaN`*. That produces significantly different results. The example below produces incosistent results because, unlike `Number.isNaN`, `isNaN` casts the `value` passed to it through `Number` first.

```
isNaN('ponyfoo')
// <- true, because Number('ponyfoo') is NaN
isNaN(new Date())
// <- true
```

While `Number.isNaN` is more precise than its global `isNaN` counterpart because it doesn't incur in casting, it's still going to confuse people **because reasons.**

1. `global.isNaN` casts input through `Number(value)` before comparison
2. `Number.isNaN` *doesn't*
3. Neither `Number.isNaN` nor `global.isNaN` answer the *"is this not a number?"* question
4. They answer whether `value` – *or `Number(value)`* – is `NaN`

In most cases, what you actually want is to know whether a value identifies as a number – *`typeof NaN === 'number'`* – and *is* a number. The method below does just that. Note that it'd work with both `global.isNaN` and `Number.isNaN` due to type checking. Everything that reports a `typeof` value of `'number'` is a number, except `NaN`, so we *weed those out* to avoid false positives!

```
function isNumber (value) {
  return typeof value === 'number' && !Number.isNaN(value)
}
```

You can use that method to figure out whether anything is **an actual number** or not. Here's some examples of what constitutes actual JavaScript numbers or not.

```
isNumber(1)
// <- true
isNumber(Infinity)
```

```
// <- true
isNumber(NaN)
// <- false
isNumber('ponyfoo')
// <- false
isNumber(new Date())
// <- false
```

Speaking of `isNumber`, isn't there something like that in the language already? *Sort of.*

`Number.isFinite`

The *rarely-advertised* `isFinite` method has been available since ES3 and it returns whether the provided `value` **matches none of**: `Infinity`, `-Infinity`, and `NaN`.

Want to take a guess about the difference between `global.isFinite` and `Number.isFinite`?

> Correct! the `global.isFinite` method coerces values through `Number(value)`, while `Number.isFinite` doesn't. Here are a few examples using `global.isFinite`. This means that values that can be coerced into *non-* `NaN` numbers will be considered finite numbers by `global.isNumber` – *even though they're aren't actually numbers!*

In most cases `isFinite` will be good enough, just like `isNaN`, but when it comes to non-numeric values it'll start acting up and producing unexpected results due to its `value` coercion into numbers.

```
isFinite(NaN)
// <- false
isFinite(Infinity)
// <- false
isFinite(-Infinity)
// <- false
isFinite(null)
// <- true, because Number(null) is 0
isFinite('10')
// <- true, because Number('10') is 10
```

Using `Number.isFinite` is just an all-around safer bet as it doesn't incur in unwanted casting. You could always do `Number.isFinite(Number(value))` if you did want the `value` to be casted into its numeric representation.

```
Number.isFinite(NaN)
// <- false
Number.isFinite(Infinity)
// <- false
Number.isFinite(-Infinity)
// <- false
Number.isFinite(null)
// <- false
Number.isFinite(0)
// <- true
```

Once again, the discrepancy doesn't do any good to the language, but `Number.isFinite` is consistently more useful than `isFinite`. Creating a polyfill for the `Number.isFinite` version is mostly a matter of type-checking.

```
Number.isFinite = function (value) {
  return typeof value === 'number' && isFinite(value)
}
```

`Number.parseInt`

This method works the same as `parseInt`. In fact, it is the same.

```
console.log(Number.parseInt === parseInt)
// <- true
```

The `parseInt` method keeps producing inconsistencies, though – even if it didn't even change, **that's the problem**. Before ES6, `parseInt` had support for hexadecimal literal notation in strings. Specifying the `radix` is not even necessary, `parseInt` infers that based on the `0x` prefix.

```
parseInt('0xf00')
// <- 3840
parseInt('0xf00', 16)
// <- 3840
```

If you hardcoded another `radix`, – *and this is* ***yet another reason*** *for doing so* – `parseInt` would bail after the first non-digit character.

```
parseInt('0xf00', 10)
// <- 0
parseInt('5xf00', 10)
// <- 5, illustrating there's no special treatment here
```

So far, it's all good. Why wouldn't I want `parseInt` to drop `0x` from hexadecimal strings? It sounds good, although you may argue that **that's doing too much**, and you'd be *probably right*.

The aggravating issue, however, is that `parseInt` hasn't changed at all. Therefore, binary and octal literal notation in strings won't work.

```
parseInt('0b011')
// <- 0
parseInt('0b011', 2)
// <- 0
parseInt('0o800')
// <- 0
parseInt('0o800', 8)
// <- 0
```

It'll be up to you to get rid of the prefix before `parseInt`. Remember to *hard-code* the `radix`, though!

```
parseInt('0b011'.slice(2), 2)
// <- 3
parseInt('0o110'.slice(2), 8)
// <- 72
```

What's *even weirder* is that the `Number` method is **perfectly able to cast** these strings into the correct numbers.

```
Number('0b011')
// <- 3
Number('0o110')
// <- 72
```

I'm not sure what drove them to keep `Number.parseInt` identical to `parseInt`. If it were up to me, I would've made it different so that it worked just like `Number` – which *is* able to **coerce octal and binary** number literal strings into the appropriate *base ten* numbers.

It might be that this was a more involved *"fork"* of `parseInt` than just *"not coercing `input` into a numeric representation"* as we observed in `Number.isNaN` and `Number.isFinite`, but I'm just guessing here.

`Number.parseFloat`

Just like `parseInt`, `parseFloat` was just added to `Number` without any modifications whatsoever.

```
Number.parseFloat === parseFloat
// <- true
```

In this case, however, `parseFloat` already didn't have any special behavior with regard to hexadecimal literal strings, meaning that this is in fact the only method that won't introduce any confusion, other than it being ported over to `Number` for *completeness' sake*.

`Number.isInteger`

This is a new method coming in ES6. It returns `true` if the provided `value` is **a finite number** that *doesn't have a decimal part*.

```
console.log(Number.isInteger(Infinity))
// <- false
console.log(Number.isInteger(-Infinity))
// <- false
console.log(Number.isInteger(NaN))
// <- false
console.log(Number.isInteger(null))
// <- false
console.log(Number.isInteger(0))
// <- true
console.log(Number.isInteger(-10))
// <- true
console.log(Number.isInteger(10.3))
// <- false
```

If you want to look at a a polyfill for `isInteger`, you might want to consider the following code snippet. The modulus operator returns the remainder of dividing the same operands — *effectively: the decimal part*. If that's `0`, that means the number is an integer.

```
Number.isInteger = function (value) {
  return Number.isFinite(value) && value % 1 === 0
}
```

Floating point arithmetic is well-documented as being kind of ridiculous. What is this `Number.EPSILON` thing?

Number.EPSILON

Let me answer that question with a piece of code.

```
Number.EPSILON
// <- 2.220446049250313e-16, wait what?
Number.EPSILON.toFixed(20)
// <- '0.00000000000000022204', got it
```

Ok, so `Number.EPSILON` is **a terribly small number**. What good is it for? Remember that thing about how floating point sum makes no sense? Here's the canonical example, I'm sure you remember it — *Yeah, I know.*

```
0.1 + 0.2
// <- 0.30000000000000004
0.1 + 0.2 === 0.3
// <- false
```

Let's try that one more time.

```
0.1 + 0.2 - 0.3
// <- 5.551115123125783e-17, what the hell?
5.551115123125783e-17.toFixed(20)
// <- '0.00000000000000005551', got it
```

So what? You can use `Number.EPSILON` to figure out whether the difference is small enough to fall under the *"floating point arithmetic is ridiculous and the difference is negligible"* category.

```
5.551115123125783e-17 < Number.EPSILON
// <- true
```

Can we trust that? Well, `0.00000000000000005551` is indeed smaller than `0.00000000000000022204`. What do you mean you don't trust me? Here they are side by side.

```
0.00000000000000005551
0.00000000000000022204
```

See? `Number.EPSILON` is *larger* than the difference. We can use `Number.EPSILON` as an acceptable margin of error due to floating point arithmetic rounding operations.

Thus, the following piece of code figures out whether the result of a floating point operation is within the expected margin of error. We use `Math.abs` because that way the order of `left` and `right` won't matter. In other words, `withinErrorMargin(left, right)` will produce the same result as `withinErrorMargin(right, left)`.

```
function withinErrorMargin (left, right) {
  return Math.abs(left - right) < Number.EPSILON
}
withinErrorMargin(0.1 + 0.2, 0.3)
// <- true
withinErrorMargin(0.2 + 0.2, 0.3)
// <- false
```

> While, yes, you **could** do this, it's probably unnecessarily complicated unless you have to deal with very low-level mathematics. You'll be better of pulling a library like `mathjs` into your project.

Last but not least, there's the other weird aspect of number representation in JavaScript. **Not every integer** can be represented *precisely*, either.

Number.MAX_SAFE_INTEGER

This is the largest integer that can be safely and precisely represented in JavaScript, or any language that represents integers using *floating point* as specified by IEEE-754 for that matter. The code below show just how large that number is. If we need to be able to deal with numbers larger than that, then I would once again point you to `mathjs`, or maybe **try another language** for your computationally intensive services.

```
Number.MAX_SAFE_INTEGER === Math.pow(2, 53) - 1
// <- true
Number.MAX_SAFE_INTEGER === 9007199254740991
// <- true
```

And you know what they say – *If there's a maximum…*

`Number.MIN_SAFE_INTEGER`

Right, nobody says that. However, there's a `Number.MIN_SAFE_INTEGER` regardless, and it's the negative value of `Number.MAX_SAFE_INTEGER`.

```
Number.MIN_SAFE_INTEGER === -Number.MAX_SAFE_INTEGER
// <- true
Number.MIN_SAFE_INTEGER === -9007199254740991
// <- true
```

How exactly can you leverage these two constants, I hear you say? In the case of the overflow problem, you don't have to implement your own `withinErrorMargin` method like you had to do for floating point precision. Instead, a `Number.isSafeInteger` is provided to you.

`Number.isSafeInteger`

This method returns `true` for any integer in the `[MIN_SAFE_INTEGER, MAX_SAFE_INTEGER]` range. There's no type coercion here either. The input must be numeric, an integer, and within the aforementioned bounds in order for the method to return `true`. Here's a quite comprehensive set of examples for you to stare at.

```
Number.isSafeInteger('a')
// <- false
Number.isSafeInteger(null)
// <- false
Number.isSafeInteger(NaN)
// <- false
Number.isSafeInteger(Infinity)
// <- false
Number.isSafeInteger(-Infinity)
// <- false
Number.isSafeInteger(Number.MIN_SAFE_INTEGER - 1)
// <- false
Number.isSafeInteger(Number.MIN_SAFE_INTEGER)
// <- true
Number.isSafeInteger(1)
// <- true
Number.isSafeInteger(1.2)
// <- false
Number.isSafeInteger(Number.MAX_SAFE_INTEGER)
// <- true
Number.isSafeInteger(Number.MAX_SAFE_INTEGER + 1)
// <- false
```

As Dr. Axel Rauschmayer points out in his article about ES6 numbers, when we want to verify if the result of an operation is within bounds, we must verify not only the result but also both operands. The reason for that is one *(or both)* of the operands may be out of bounds, while the result is *"safe"* **(but incorrect)**. Similarly, the result may be out of bounds itself, so checking all of `left`, `right`, and the result of `left op right` is necessary to verify that we can indeed trust the result.

In all of the examples below, **the result is incorrect**. Here's the first example, where both operands are safe even though the result is not.

```
Number.isSafeInteger(9007199254740000)
// <- true
Number.isSafeInteger(993)
// <- true
Number.isSafeInteger(9007199254740000 + 993)
// <- false
9007199254740000 + 993
// <- 9007199254740992, should be 9007199254740993
```

In this example one of the operands wasn't within range, so we can't trust the result to be accurate.

```
Number.isSafeInteger(9007199254740993)
```

```
// <- false
Number.isSafeInteger(990)
// <- true
Number.isSafeInteger(9007199254740993 + 990)
// <- false
9007199254740993 + 990
// <-  9007199254741982, should be 9007199254741983
```

Note that in the example above, a subtraction would produce a result within bounds, and that result would *also* be inaccurate.

```
Number.isSafeInteger(9007199254740993)
// <- false
Number.isSafeInteger(990)
// <- true
Number.isSafeInteger(9007199254740993 - 990)
// <- true
9007199254740993 - 990
// <-  9007199254740002, should be 9007199254740003
```

It doesn't take a genius to figure out the case where both operands are out of bounds but the result is *deemed "safe"*, even though the result is incorrect.

```
Number.isSafeInteger(9007199254740993)
// <- false
Number.isSafeInteger(9007199254740995)
// <- false
Number.isSafeInteger(9007199254740993 - 9007199254740995)
// <- true
9007199254740993 - 9007199254740995
// <- -4, should be -2
```

Thus, as you can see, we can conclude that the only safe way to assert whether an operation is correct is with a method like the one below. If we can't ascertain that the operation and both its operands are within bounds, then the result may be inaccurate, and that's a problem. It's best to `throw` in those situations and have a way to error-correct, but that's specific to your programs. The important part is to actually catch these kinds of difficult bugs to deal with.

```
function trusty (left, right, result) {
  if (
    Number.isSafeInteger(left) &&
    Number.isSafeInteger(right) &&
    Number.isSafeInteger(result)
  ) {
    return result
  }
  throw new RangeError('Operation cannot be trusted!')
}
```

You could then use that every step of the way to ensure all operands remain safely within bounds. I've highlighted the unsafe values in the examples below. Note that even though none of the operations in my examples return accurate results, certain operations and numbers *may do so* even when operands are out of bounds. The problem is that that **can't be guaranteed** – *therefore the operation can't be trusted.*

```
trusty(9007199254740000, 993, 9007199254740000 + 993)
// <- RangeError: Operation cannot be trusted!
trusty(9007199254740993, 990, 9007199254740993 + 990)
// <- RangeError: Operation cannot be trusted!
trusty(9007199254740993, 990, 9007199254740993 - 990)
// <- RangeError: Operation cannot be trusted!
trusty(9007199254740993, 9007199254740995, 9007199254740993 - 9007199254740995)
// <- RangeError: Operation cannot be trusted!
trusty(1, 2, 3)
// <- 3
```

I don't think I want to write about floating point again for a while. *Time to scrub myself up.*

Conclusions

While some of the hacks to guard against rounding errors and overflow safety are *nice to have*, they don't attack the heart of the problem: *math with the IEEE-754 standard is hard.*

These days JavaScript runs on all the things, so it'd be nice if a better standard were to be *implemented alongside IEEE-754*. Roughly a year ago, *Douglas Crockford* came up with DEC64, but opinions on its merits range from *"this is genius!"* to *"this is the work of a madman"* – I guess that's the norm when it comes to most of the stuff Crockford publishes, though.

It'd be nice, to eventually see the day where JavaScript is able to *precisely* compute decimal arithmetic as well as able to represent large integers safely. That day we'll probably have something **alongside** *floating point*.

🠕 **Back to top**

ES6 Math Additions in Depth

You've made it! Here's another article in the ES6 – *"What? I'd rather develop for IE6"* – in Depth series. If you've never been around here before, start with A Brief History of ES6 Tooling. Then, make your way through destructuring, template literals, arrow functions, the spread operator and rest parameters, improvements coming to object literals, the new *classes* sugar on top of prototypes, `let`, `const`, and the *"Temporal Dead Zone"*, iterators, generators, Symbols, Maps, WeakMaps, Sets, and WeakSets, proxies, proxy traps, more proxy traps, reflection, and `Number`. Today we'll learn about new `Math` methods.

> As I did in previous articles on the series, I would love to point out that you should probably set up Babel and follow along the examples with either a REPL or the `babel-node` CLI and a file. That'll make it so much easier for you to **internalize the concepts** discussed in the series. If you aren't the *"install things on my computer"* kind of human, you might prefer to hop on CodePen and then click on the gear icon for JavaScript – *they have a Babel preprocessor which makes trying out ES6 a breeze.* Another alternative that's also quite useful is to use Babel's online REPL – *it'll show you compiled ES5 code to the right of your ES6 code for quick comparison.*

Before getting into it, let me *shamelessly ask for your support* if you're enjoying my ES6 in Depth series. Your contributions will go towards helping me keep up with the schedule, server bills, keeping me fed, and maintaining **Pony Foo** as a veritable source of JavaScript goodies.

Thanks for reading that, and let's go into `Math` improvements. For a bit of context you may want to look at the extensive article on `Number` improvements from last week. Time to dig into `Math`.

`Math` Additions in ES6

There's *heaps* of additions to `Math` in ES6. Just like you're used to, these are **static** methods on the `Math` built-in. Some of these methods were specifically engineered towards making it easier to compile C into JavaScript, and you may never come across a need for them in day-to-day development – particularly not when it comes to front-end development. Other methods are complements to the existing rounding, exponentiation, and trigonometry API surface.

Below is a full list of methods added to `Math`. They are grouped by functionality and sorted by relevance.

- **Utility**
- `Math.sign` – sign function of a number
- `Math.trunc` – integer part of a number
- **Exponentiation and Logarithmic**
- `Math.cbrt` – cubic root of value, or `¯value`
- `Math.expm1` – `e` to the `value` minus `1`, or $e^{value} - 1$
- `Math.log1p` – natural logarithm of `value + 1`, or $ln(value + 1)$
- `Math.log10` – base 10 logarithm of `value`, or $log_{10}(value)$
- `Math.log2` – base 2 logarithm of `value`, or $log_2(value)$
- **Trigonometry**
- `Math.sinh` – hyperbolic sine of a number
- `Math.cosh` – hyperbolic cosine of a number
- `Math.tanh` – hyperbolic tangent of a number
- `Math.asinh` – hyperbolic arc-sine of a number
- `Math.acosh` – hyperbolic arc-cosine of a number
- `Math.atanh` – hyperbolic arc-tangent of a number
- `Math.hypot` – square root of the sum of squares

Bitwise

- `Math.clz32` — leading zero bits in the 32-bit representation of a number
- **Compile-to-JavaScript**
- `Math.imul` — *C-like* 32-bit multiplication
- `Math.fround` — nearest single-precision float representation of a number

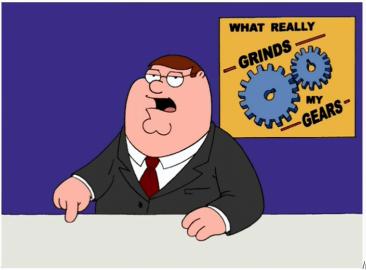
Let's get right into it.

`Math.sign`

Many languages have a `Math.Sign` method *(or equivalent)* that returns a vector like `-1`, `0`, or `1`, depending on the sign of the provided input. Surely then, you would think JavaScript's `Math.sign` method does the same. Well, sort of. The JavaScript flavor of this method has two more alternatives: `-0`, and `NaN`.

```
Math.sign(1)
// <- 1
Math.sign(0)
// <- 0
Math.sign(-0)
// <- -0
Math.sign(-30)
// <- -1
Math.sign(NaN)
// <- NaN
Math.sign('foo')
// <- NaN, because Number('foo') is NaN
Math.sign('0')
// <- 0, because Number('0') is 0
Math.sign('-1')
// <- -1, because Number('-1') is -1
```

This is just one of those methods. It **grinds my gears.** After all the trouble we went through to document how methods ported over to `Number`, such as `Number.isNaN`, don't indulge in unnecessary type coercion, why is it that `Math.sign` *does* coerce its input? I have no idea. Most of the methods in `Math` share this trait, though. The methods that were added to `Number` don't.



/p>

Sure, we're not a statically typed language, we dislike throwing exceptions, and we're fault tolerant — after all, this is one of the founding languages of the web. But was not coercing everything into a `Number` too much to ask? Couldn't we just return `NaN` for *non-numeric* values?

I'd love for us to get over implicit casting, but it seems we're *not quite there yet* for the time being.

`Math.trunc`

One of the oddities in `Math` methods is how abruptly they were named. It's like they were trying to save keystrokes or something. After all, it's not like we stopped adding *super-precise* method names like `Object.getOwnPropertySymbols()`. Why `trunc` instead of `truncate`, then? Who knows.

Anyways, `Math.trunc` is a simple alternative to `Math.floor` and `Math.ceil` where we simply discard the decimal part of a number. Once again, the input is coerced into a numeric value through `Number(value)`.

```
Math.trunc(12.34567)
// <- 12
Math.trunc(-13.58)
// <- -13
Math.trunc(-0.1234)
// <- -0
Math.trunc(NaN)
// <- NaN
Math.trunc('foo')
// <- NaN, because Number('foo') is NaN
Math.trunc('123.456')
// <- 123, because Number('123.456') is 123.456
```

While it still coerces any values into numbers, at least it stayed consistent with `Math.floor` and `Math.ceil`, enough that you could use them to create a simple polyfill for `Math.trunc`.

```
Math.trunc = function truncate (value) {
  return value >
0 ? Math.floor(value) : Math.ceil(value)
}
```

Another apt example of how *"succintly"* `Math` methods have been named in ES6 can be found in `Math.cbrt` — *although this one matches the pre-existing* `Math.sqrt` *method, to be fair.*

Math.cbrt

As hinted above, `Math.cbrt` is short for *"cubic root"*. The examples below show the sorts of output it produces.

```
Math.cbrt(-1)
// <- -1
Math.cbrt(3)
// <- 1.4422495703074083
Math.cbrt(8)
// <- 2
Math.cbrt(27)
// <- 3
```

Not much explaining to do here. Note that this method *also* coerces non-numerical values into numbers.

```
Math.cbrt('8')
// <- 2, because Number('8') is 8
Math.cbrt('ponyfoo')
// <- NaN, because Number('ponyfoo') is NaN
```

Let's move onto something else.

Math.expm1

This operation is the result of computing `e` to the `value` minus `1`. In JavaScript, the `e` constant is defined as `Math.E`. The method below is a rough equivalent of `Math.expm1`.

```
function expm1 (value) {
  return Math.pow(Math.E, value) - 1
}
```

The e^{value} operation can be expressed as `Math.exp(value)` as well.

```
function expm1 (value) {
  return Math.exp(value) - 1
}
```

Note that this method has higher precision than merely doing `Math.exp(value) - 1`, and should be the preferred alternative.

```
expm1(1e-20)
// <- 0
Math.expm1(1e-20)
```

```
// <- 1e-20
expm1(1e-10)
// <- 1.000000082740371e-10
Math.expm1(1e-10)
// <- 1.00000000005e-10
```

The inverse function of `Math.expm1` is `Math.log1p`.

`Math.log1p`

This is the natural logarithm of `value` plus `1`, — *ln*(value + 1) — and the inverse function of `Math.expm1`. The base `e` logarithm of a number can be expressed as `Math.log` in JavaScript.

```
function log1p (value) {
 return Math.log(value + 1)
}
```

This method is **more precise** than executing the `Math.log(value + 1)` operation by hand, just like the `Math.expm1` case.

```
log1p(1.00000000005e-10)
// <- 1.000000082690371e-10
Math.log1p(1.00000000005e-10)
// <- 1e-10, exactly the inverse of Math.expm1(1e-10)
```

Next up is `Math.log10`.

`Math.log10`

Base ten logarithm of a number — log_{10}(value).

```
Math.log10(1000)
// <- 3
```

You could polyfill `Math.log10` using the `Math.LN10` constant.

```
function log10 (value) {
  return Math.log(x) / Math.LN10
}
```

And then there's `Math.log2`.

`Math.log2`

Base two logarithm of a number — log_{2}(value).

```
Math.log2(1024)
// <- 10
```

You could polyfill `Math.log2` using the `Math.LN2` constant.

```
function log2 (value) {
  return Math.log(x) / Math.LN2
}
```

Note that the polyfilled version won't be as precise as `Math.log2` in some cases. Remember that the `<<` operator means *"bitwise left shift"*.

```
Math.log2(1 << 29)
// <- 29
log2(1 << 29)
// <- 29.000000000000004
```

Naturally, you could use `Math.round` or `Number.EPSILON` to get around rounding issues.

`Math.sinh`

Returns the hyperbolic sine of `value`.

Math.cosh

Returns the hyperbolic cosine of `value`.

Math.tanh

Returns the hyperbolic tangent of `value`.

Math.asinh

Returns the hyperbolic arc-sine of `value`.

Math.acosh

Returns the hyperbolic arc-cosine of `value`.

Math.atanh

Returns the hyperbolic arc-tangent of `value`.

Math.hypot

Returns the square root of the sum of the squares of the arguments.

```
Math.hypot(1, 2, 3)
// <- 3.741657386773941
```

We could polyfill `Math.hypot` by doing the operations manually. We can use `Math.sqrt` to compute the square root and `Array.prototype.reduce` combined with the spread operator to sum the squares. I'll throw in an arrow function for good measure!

```
function hypot (...values) {
  return Math.sqrt(values.reduce((sum, value) => sum + value * value, 0))
}
```

Surprisingly, our handmade method is more precise than the native one *(at least on Chrome 45)* for **this case in particular**.

```
Math.hypot(1, 2, 3)
// <- 3.741657386773941
hypot(1, 2, 3)
// <- 3.7416573867739413
```

And now for the **"*really* fun"** methods!

Math.clz32

Definitely not immediately obvious, but the name for this method is an acronym for *"count leading zero bits in 32-bit binary representations of a number"*. Remember that the `<<` operator means *"bitwise left shift"*, and thus...

```
Math.clz32(0)
// <- 32
Math.clz32(1)
// <- 31
Math.clz32(1 << 1)
// <- 30
Math.clz32(1 << 2)
// <- 29
Math.clz32(1 << 29)
// <- 2
```

Cool, and also probably the last time you're going to see that method in use for the foreseeable future. For completeness' sake, I'll add a sentence about the pair of methods that were added mostly to aid with `asm.js` compilation of C programs. *I doubt you'll be using these directly, ever.*

Math.imul

Returns the result of a C-like 32-bit multiplication.

`Math.fround`

Rounds `value` to the nearest 32-bit float representation of a number.

Conclusions

Some nice methods rounding out the `Math` API. It would've been nice to see additions to the tune of more flavors of `Math.random` and similar utilities that end up being implemented by libraries in almost every large-enough application, such as Lodash's `_.random` and `_.shuffle`.

That being said, any help towards making `asm.js` faster and more of a reality are desperately welcome additions to the language.

↑ **Back to top**

ES6 Array Extensions in Depth

Hello, traveler! This is ES6 – *"Oh cool, I like `Array`"* – in Depth series. If you've never been around here before, start with A Brief History of ES6 Tooling. Then, make your way through destructuring, template literals, arrow functions, the spread operator and rest parameters, improvements coming to object literals, the new *classes* sugar on top of prototypes, `let`, `const`, and the *"Temporal Dead Zone"*, iterators, generators, Symbols, Maps, WeakMaps, Sets, and WeakSets, proxies, proxy traps, more proxy traps, reflection, `Number`, and `Math`. Today we'll learn about new `Array` extensions.

> As I did in previous articles on the series, I would love to point out that you should probably set up Babel and follow along the examples with either a REPL or the `babel-node` CLI and a file. That'll make it so much easier for you to **internalize the concepts** discussed in the series. If you aren't the *"install things on my computer"* kind of human, you might prefer to hop on CodePen and then click on the gear icon for JavaScript – *they have a Babel preprocessor which makes trying out ES6 a breeze.* Another alternative that's also quite useful is to use Babel's online REPL – *it'll show you compiled ES5 code to the right of your ES6 code for quick comparison.*

Before getting into it, let me *shamelessly ask for your support* if you're enjoying my ES6 in Depth series. Your contributions will go towards helping me keep up with the schedule, server bills, keeping me fed, and maintaining **Pony Foo** as a veritable source of JavaScript goodies.

Thanks for reading that, and let's go into `Array` extensions. For a bit of context you may want to look at the articles on iterators, generators, arrow functions and collections.

Upcoming `Array` Methods

There's plenty to choose from. Over the years, libraries like Underscore and Lodash spoke loudly about features we were missing in the language, and now we have a ton more tools in the functional array arsenal at our disposal.

First off, there's a couple of static methods being added.

- `Array.from` – create `Array` instances from arraylike objects like `arguments` or iterables
- `Array.of`

Then there's a few methods that help you manipulate, fill, and filter arrays.

- `Array.prototype.copyWithin`
- `Array.prototype.fill`
- `Array.prototype.find`
- `Array.prototype.findIndex`

There's also the methods related to the iterator protocol.

- `Array.prototype.keys`
- `Array.prototype.values`

- `Array.prototype.entries`
- `Array.prototype[Symbol.iterator]`

There's a few more methods coming in ES2016 *(ES7)* as well, but we won't be covering those today.

- `Array.prototype.includes`
- `Array.observe`
- `Array.unobserve`

Let's get to work!

`Array.from`

This method has been long overdue. Remember the quintessential example of converting an arraylike into an actual array?

```
function cast ()
  return Array.prototype.slice.call(arguments)
}
cast('a', 'b')
// <- ['a', 'b']
```

Or a shorter form perhaps?

```
function cast ()
  return [].slice.call(arguments)
}
```

To be fair, we've already explored even more terse ways of doing this at some point during the ES6 in depth series. For instance you could use the spread operator. As you probably remember, the spread operator leverages the iterator protocol to produce a sequence of values in arbitrary objects. The downside is that the objects we want to cast with spread **must implement** `@@iterator` through `Symbol.iterator`. Luckily for us, `arguments` does implement the iterator protocol in ES6.

```
function cast ()
  return [...arguments]
}
```

Another thing you could be casting through the spread operator is DOM element collections like those returned from `document.querySelectorAll`. Once again, this is made possible thanks to ES6 adding conformance to the iterator protocol to `NodeList`.

```
[...document.querySelectorAll('div')]
// <- [<div>, <div>, <div>, ...]
```

What happens when we try to cast a jQuery collection through the spread operator? Actually, you'll **get an exception** because they haven't implemented `Symbol.iterator` quite yet. You can try this one on jquery.com in Firefox.

```
[...$('div')]
// <- TypeError: $(...)[Symbol.iterator] is not a function
```

The new `Array.from` method is different, though. It doesn't *only* rely on iterator protocol to figure out how to pull values from an object. It also has support for arraylikes out the box.

```
Array.from($('div'))
// <- [<div>, <div>, <div>, ...]
```

The one thing you cannot do with either `Array.from` nor the spread operator is to pick a start index. Suppose you wanted to pull every `<div>` after the first one. With `.slice.call`, you could do it like so:

```
[].slice.call(document.querySelectorAll('div'), 1)
```

Of course, there's nothing stopping you from using `.slice` *after* casting. This is probably way easier to read, and looks more like functional programming, so there's that.

```
Array.from(document.querySelectorAll('div')).slice(1)
```

`Array.from` actually has three arguments, *but only the* `input` *is required*. To wit:

- `input` — the arraylike or iterable object you want to cast
- `map` — a mapping function that's executed on every item of `input`
- `context` — the `this` binding to use when calling `map`

With `Array.from` we cannot slice, but we can dice!

```
function typesOf () {
  return Array.from(arguments, value => typeof value)
}
typesOf(null, [], NaN)
// <- ['object', 'object', 'number']
```

Do note that you could also just combine rest parameters and `.map` if you were just dealing with `arguments`. In this case in particular, we may be better off just doing something like the snippet of code found below.

```
function typesOf (...all) {
  return all.map(value => typeof value)
}
typesOf(null, [], NaN)
// <- ['object', 'object', 'number']
```

In some cases, like the case of jQuery we saw earlier, it makes sense to use `Array.from`.

```
Array.from($('div'))
// <- [<div>, <div>, <div>, ...]
Array.from($('div'), el => el.id)
// <- ['', 'container', 'logo-events', 'broadcast', ...]
```

I guess you get the idea.

`Array.of`

This method is exactly like the first incarnation of `cast` we played with in our analysis of `Array.from`.

```
Array.of = function of () {
  return Array.prototype.slice.call(arguments)
}
```

You can't just replace `Array.prototype.slice.call` with `Array.of`. They're different animals.

```
Array.prototype.slice.call([1, 2, 3])
// <- [1, 2, 3]
Array.of(1, 2, 3)
// <- [1, 2, 3]
```

You can think of `Array.of` as an alternative for `new Array` that doesn't have the `new Array(length)` overload. Below you'll find some of the strange ways in which `new Array` behaves thanks to its single-argument `length` overloaded constructor. If you're confused about the `undefined x ${number}` notation in the browser console, that's indicating there are array holes in those positions.

```
new Array()
// <- []
new Array(undefined)
// <- [undefined]
new Array(1)
// <- [undefined x 1]
new Array(3)
// <- [undefined x 3]
new Array(1, 2)
// <- [1, 2]
new Array(-1)
// <- RangeError: Invalid array length
```

In contrast, `Array.of` has more consistent behavior because it doesn't have the special `length` case.

```
Array.of()
// <- []
Array.of(undefined)
// <- [undefined]
Array.of(1)
// <- [1]
```

```
Array.of(3)
// <- [3]
Array.of(1, 2)
// <- [1, 2]
Array.of(-1)
// <- [-1]
```

There's not a lot to add here – let's move on.

`Array.prototype.copyWithin`

This is the most obscure method that got added to `Array.prototype`. I suspect use cases lie around buffers and typed arrays – *which we'll cover at some point, later in the series*. The method copies a sequence of array elements *within the array* to the *"paste position"* starting at `target`. The elements that should be copied are taken from the `[start, end)` range.

Here's the signature of the `copyWithin` method. The `target` *"paste position"* **is required**. The `start` index where to take elements from defaults to `0`. The `end` position defaults to the length of the array.

```
Array.prototype.copyWithin(target, start = 0, end = this.length)
```

Let's start with a simple example. Consider the `items` array in the snippet below.

```
var items = [1, 2, 3, ,,,,,,,]
// <- [1, 2, 3, undefined x 7]
```

The method below takes the `items` array and determines that it'll start *"pasting"* items in the **sixth position**. It further determines that the items to be copied will be taken starting in the **second position** *(zero-based)*, until the **third position** *(also zero-based)*.

```
items.copyWithin(6, 1, 3)
// <- [1, 2, 3, undefined × 3, 2, 3, undefined × 2]
```

> Reasoning about this method can be pretty hard. *Let's break it down.*

If we consider that the items to be copied were taken from the `[start, end)` range, then we can express that using the `.slice` operation. These are the items that were *"pasted"* at the `target` position. We can use `.slice` to *"copy"* them.

```
items.slice(1, 3)
// <- [2, 3]
```

We could then consider the "pasting" part of the operation as an advanced usage of `.splice` – one of those lovely methods that can do just about anything. The method below does just that, and then returns `items`, because `.splice` returns the items that were spliced from an Array, and in our case this is no good. Note that we also had to use the spread operator so that elements are inserted individually through `.splice`, and not as an array.

```
function copyWithin (items, target, start = 0, end = items.length) {
  items.splice(target, end - start, ...items.slice(start, end))
  return items
}
```

Our example would still work the same with this method.

```
copyWithin([1, 2, 3, ,,,,,,,], 6, 1, 3)
// <- [1, 2, 3, undefined × 3, 2, 3, undefined × 2]
```

The `copyWithin` method accepts negative `start` indices, negative `end` indices, and negative `target` indices. Let's try something using that.

```
[1, 2, 3, ,,,,,,,].copyWithin(-3)
// <- [1, 2, 3, undefined x 4, 1, 2, 3]
copyWithin([1, 2, 3, ,,,,,,,], -3)
// <- [1, 2, 3, undefined x 4, 1, 2, 3, undefined x 7]
```

Turns out, that thought exercise was useful for understanding `Array.prototype.copyWithin`, but it wasn't actually correct. Why are we seeing `undefined x 7` at the end? Why the discrepancy? The problem is that we are seeing the array holes at the end of `items` when we do `...items.slice(start, end)`.

```
[1, 2, 3, ,,,,,,,]
```

```
// <- [1, 2, 3, undefined x 7]
[1, 2, 3, ,,,,,,,].slice(0, 10)
// <- [1, 2, 3, undefined x 7]
console.log(...[1, 2, 3, ,,,,,,,].slice(0, 10))
// <- 1, 2, 3, undefined, undefined, undefined, undefined, undefined, undefined, undefined
```

Thus, we *end up splicing the holes* onto `items`, while the original solution is not. We could get rid of the holes using `.filter`, which conveniently discards array holes.

```
[1, 2, 3, ,,,,,,,].slice(0, 10)
// <- [1, 2, 3, undefined x 7]
[1, 2, 3, ,,,,,,,].slice(0, 10).filter(el => true)
// <- [1, 2, 3]
```

With that, we can update our `copyWithin` method. We'll stop using `end - start` as the splice position and instead use the amount of `replacements` that we have, as those numbers may be different now that we're discarding array holes.

```
function copyWithin (items, target, start = 0, end = items.length) {
  var replacements = items.slice(start, end).filter(el => true)
  items.splice(target, replacements.length, ...replacements)
  return items
}
```

The case where we previously added extra holes now works as expected. Woo!

```
[1, 2, 3, ,,,,,,,].copyWithin(-3)
// <- [1, 2, 3, undefined x 4, 1, 2, 3]
copyWithin([1, 2, 3, ,,,,,,,], -3)
// <- [1, 2, 3, undefined x 4, 1, 2, 3]
```

Furthermore, our polyfill seems to work correctly *across the board* now. I wouldn't rely on it for anything other than educational purposes, though.

```
[1, 2, 3, ,,,,,,,].copyWithin(-3, 1)
// <- [1, 2, 3, undefined x 4, 2, 3, undefined x 1]
copyWithin([1, 2, 3, ,,,,,,,], -3, 1)
// <- [1, 2, 3, undefined x 4, 2, 3, undefined x 1]
[1, 2, 3, ,,,,,,,].copyWithin(-6, -8)
// <- [1, 2, 3, undefined x 1, 3, undefined x 5]
copyWithin([1, 2, 3, ,,,,,,,], -6, -8)
// <- [1, 2, 3, undefined x 1, 3, undefined x 5]
[1, 2, 3, ,,,,,,,].copyWithin(-3, 1, 2)
// <- [1, 2, 3, undefined x 4, 2, undefined x 2]
copyWithin([1, 2, 3, ,,,,,,,], -3, 1, 2)
// <- [1, 2, 3, undefined x 4, 2, undefined x 2]
```

It's decidedly better to just use the actual implementation, but at least now we have **a better idea of how the hell it works!**

Array.prototype.fill

Convenient utility method to fill all places in an `Array` with the provided `value`. Note that array holes will be filled as well.

```
['a', 'b', 'c'].fill(0)
// <- [0, 0, 0]
new Array(3).fill(0)
// <- [0, 0, 0]
```

You could also determine a start index and an end index in the second and third parameters respectively.

```
['a', 'b', 'c',,,].fill(0, 2)
// <- ['a', 'b', 0, 0, 0]
new Array(5).fill(0, 0, 3)
// <- [0, 0, 0, undefined x 2]
```

The provided value can be arbitrary, and not necessarily a number or even a primitive type.

```
new Array(3).fill({})
// <- [{}, {}, {}]
```

Unfortunately, you can't fill arrays using a mapping method that takes an `index` parameter or anything like that.

```
new Array(3).fill(function foo () {})
// <- [function foo () {}, function foo () {}, function foo () {}]
```

Moving along…

Array.prototype.find

Ah. One of those methods that JavaScript desperately wanted but didn't get in ES5. The `.find` method returns the *first* `item` that matches `callback(item, i, array)` for an `array` Array. You can also optionally pass in a `context` binding for `this`. You can think of it as an equivalent of `.some` that returns the matching element *(or `undefined`)* instead of merely `true` or `false`.

```
[1, 2, 3, 4, 5].find(item => item > 2)
// <- 3
[1, 2, 3, 4, 5].find((item, i) => i === 3)
// <- 4
[1, 2, 3, 4, 5].find(item => item === Infinity)
// <- undefined
```

There's really not much else to say about this method. It's just that simple! We did want this method a lot, as evidenced in libraries like Lodash and Underscore. Speaking of those libraries… – `.findIndex` was also born there.

Array.prototype.findIndex

This method is also an equivalent of `.some` and `.find`. Instead of returning `true`, like `.some`; or `item`, like `.find`; this method returns the `index` position so that `array[index] === item`. If none of the elements in the collection match the `callback(item, i, array)` criteria, the return value is `-1`.

```
[1, 2, 3, 4, 5].findIndex(item => item > 2)
// <- 2
[1, 2, 3, 4, 5].findIndex((item, i) => i === 3)
// <- 3
[1, 2, 3, 4, 5].findIndex(item => item === Infinity)
// <- -1
```

Again, quite straightforward.

Array.prototype.keys

Returns an iterator that yields a sequence holding the keys for the array. The returned value is an iterator, meaning you can use it with all of the usual suspects like `for..of`, the spread operator, or by hand by manually calling `.next()`.

```
[1, 2, 3].keys()
// <- ArrayIterator {}
```

Here's an example using `for..of`.

```
for (let key of [1, 2, 3].keys()) {
  console.log(key)
  // <- 0
  // <- 1
  // <- 2
}
```

Unlike `Object.keys` and most methods that iterate over arrays, this sequence doesn't ignore holes.

```
[...new Array(3).keys()]
// <- [0, 1, 2]
Object.keys(new Array(3))
// <- []
```

Now onto values.

Array.prototype.values

Same thing as `.keys()`, but the returned iterator is a sequence of values instead of indices. In practice, you'll probably just iterate over the array itself, but sometimes getting an iterator can come in handy.

```
[1, 2, 3].values()
// <- ArrayIterator {}
```

Then you can use `for..of` or any other methods like a spread operator to pull out the sequence. The example below shows how using the spread operator on an array's `.values()` doesn't really make a lot of sense — *you already had that collection to begin with!*

```
[...[1, 2, 3].values()]
// <- [1, 2, 3]
```

Do note that the returned array in the example above is *a different array* and not a reference to the original one.

> Time for `.entries`.

`Array.prototype.entries`

Similar to both preceding methods, but this one returns an iterator with a sequence of key-value pairs.

```
['a', 'b', 'c'].entries()
// <- ArrayIterator {}
```

Each entry contains a two dimensional array element with the key and the value for an item in the array.

```
[...['a', 'b', 'c'].entries()]
// <- [[0, 'a'], [1, 'b'], [2, 'c']]
```

Great, last one to go!

`Array.prototype[Symbol.iterator]`

This is ~~basically~~ **exactly** the same as the `.values` method. The example below combines a spread operator, an array, and `Symbol.iterator` to iterate over its values.

```
[...['a', 'b', 'c'][Symbol.iterator]()]
// <- ['a', 'b', 'c']
```

Of course, you should probably just omit the spread operator and the `[Symbol.iterator]` part in most use cases. Same time tomorrow? We'll cover changes to the `Object` API.

↑ **Back to top**

ES6 Object Changes in Depth

Howdy. You're reading ES6 – *"I vehemently `Object` to come up with a better tagline"* – in Depth series. If you've never been around here before, start with A Brief History of ES6 Tooling. Then, make your way through destructuring, template literals, arrow functions, the spread operator and rest parameters, improvements coming to object literals, the new *classes* sugar on top of prototypes, `let`, `const`, and the *"Temporal Dead Zone"*, iterators, generators, Symbols, Maps, WeakMaps, Sets, and WeakSets, proxies, proxy traps, more proxy traps, reflection, `Number`, `Math`, and `Array`. Today we'll learn about changes to `Object`.

> As I did in previous articles on the series, I would love to point out that you should probably set up Babel and follow along the examples with either a REPL or the `babel-node` CLI and a file. That'll make it so much easier for you to **internalize the concepts** discussed in the series. If you aren't the *"install things on my computer"* kind of human, you might prefer to hop on CodePen and then click on the gear icon for JavaScript – *they have a Babel preprocessor which makes trying out ES6 a breeze.* Another alternative that's also quite useful is to use Babel's online REPL – *it'll show you compiled ES5 code to the right of your ES6 code for quick comparison.*

Before getting into it, let me *shamelessly ask for your support* if you're enjoying my ES6 in Depth series. Your contributions will go towards helping me keep up with the schedule, server bills, keeping me fed, and maintaining **Pony Foo** as a veritable source of JavaScript goodies.

Thanks for reading that, and let's go into changes to `Object`. Make sure to read some of the articles from earlier in the series to get comfortable with ES6 syntax changes.

Upcoming `Object` Changes

Objects didn't get as many new methods in ES6 as arrays did. In the case of objects, we get four new static methods, and no new instance methods or properties.

- `Object.assign`
- `Object.is`
- `Object.getOwnPropertySymbols`
- `Object.setPrototypeOf`

And just like arrays, objects are slated to get a few more static methods in ES2016 *(ES7)*. We're not going to cover these today.

- `Object.observe`
- `Object.unobserve`

Shall we?

`Object.assign`

This is another example of the kind of helper method that has been beaten to death by libraries like Underscore and Lodash. I even wrote my own implementation that's around 20 lines of code. You can use `Object.assign` to recursively overwrite properties on an object with properties from other objects. The first argument passed to `Object.assign`, `target`, will be *used as the return value as well.* Subsequent values are *"applied"* onto that object.

```
Object.assign({}, { a: 1 })
// <- { a: 1 }
```

If you already had a property, it's overwritten.

```
Object.assign({ a: 1 }, { a: 2 })
// <- { a: 2 }
```

Properties that aren't present in the object being assigned are left untouched.

```
Object.assign({ a: 1, b: 2 }, { a: 3 })
// <- { a: 3, b: 2 }
```

You can assign as many objects as you want. You can think of `Object.assign(a, b, c)` as the equivalent of doing `Object.assign(Object.assign(a, b), c)`, if that makes it easier for you to reason about it. I like to reason about it as a reduce operation.

```
Object.assign({ a: 1, b: 2 }, { a: 3 }, { c: 4 })
// <- { a: 3, b: 2, c: 4 }
```

Note that only enumerable own properties are copied over – *think* *`Object.keys`* *plus* *`Object.getOwnPropertySymbols`*. The example below shows an `invisible` property that didn't get copied over. Properties from the prototype chain aren't taken into account either.

```
var a = { b: 'c' }
Object.defineProperty(a, 'invisible', { enumerable: false, value: 'boo! ahhh!' })
Object.assign({}, a)
// <- { b: 'c' }
```

You can use this API against arrays as well.

```
Object.assign([1, 2, 3], [4, 5])
// <- [4, 5, 3]
```

Properties using symbols as their keys are also copied over.

```
Object.assign({ a: 'b' }, { [Symbol('c')]: 'd' })
// <- { a: 'b', Symbol(c): 'd' }
```

As long as they're enumerable and found directly on the object, that is.

```
var a = {}
Object.defineProperty(a, Symbol('b'), { enumerable: false, value: 'c' })
Object.assign({}, a)
// <- {}
```

There's a problem with `Object.assign`. It doesn't allow you to control how deep you want to go. You may be hoping for a way to do the following while preserving the `target.a.d` property, but `Object.assign` replaces `target.a` entirely with `source.a`.

```
var target = { a: { b: 'c', d: 'e' } }
var source = { a: { b: 'ahh!' } }
Object.assign(target, source)
// <- { a: { b: 'ahh!' } }
```

Most implementations in the wild work differently, at least giving you *the option* to make a *"deep assign"*. Take `assignment` for instance. If it finds an object reference in `target` for a given property, it has two options.

- If the value in `source[key]` is an object, it goes recursive with an `assignment(target[key], source[key])` call
- If the value is not an object, it just replaces it: `target[key] = source[key]`

This means that the last example we saw would work differently with `assignment`.

```
var target = { a: { b: 'c', d: 'e' } }
var source = { a: { b: 'ahh!' } }
assignment(target, source)
// <- { a: { b: 'ahh!', d: 'e' } }
```

This is **usually preferred** when it comes to the most common use case of this type of method: providing sensible defaults that can be overwritten by the user. Consider the following example. It uses the well-known pattern of providing your *"assign"* method with an empty object, that's then filled with default values, and then poured user preferences for good measure. Note that it doesn't change the defaults object directly because those are supposed to stay the same across invocations.

```
function markdownEditor (user) {
  var defaults = {
    height: 400,
    markdown: {
      githubFlavored: true,
      tables: false
    }
  }
  var options = Object.assign({}, defaults, user)
  console.log(options)
}
```

The problem with `Object.assign` is that if the `markdownEditor` consumer wants to change `markdown.tables` to `true`, all of the other defaults in `markdown` will be lost!

```
markdownEditor({ markdown: { tables: true } })
// <- {
//      height: 400,
//      markdown: {
//        tables: true
//      }
//    }
```

From both the library author's perspective and the library's user perspective, this is just unacceptable and weird. If we were to use `assignment` we wouldn't be having those issues, because `assignment` is built with this particular use case in mind. Libraries like Lodash usually provide many different flavors of this method.

Note that when it comes to nested arrays, **replacement** *probably is* the behavior you want most of the time. Given defaults like `{ extensions: ['css', 'js', 'html'] }`, the following would be quite weird.

```
markdownEditor({ extensions: ['js'] })
// <- { extensions: ['js', 'js', 'html'] }
```

For that reason, `assignment` replaces arrays entirely, just like `Object.assign` would. This difference **doesn't** make `Object.assign` useless, but it's still necessary to know about the difference between shallow and deep assignment.

`Object.is`

This method is pretty much a programmatic way to use the `===` operator. You pass in two arguments and it tells you whether they're the same reference or the same primitive value.

```
Object.is('foo', 'foo')
// <- true
Object.is({}, {})
// <- false
```

There are **two important differences**, however. First off, `-0` and `+0` are considered unequal by this method, even though `===` returns `true`.

```
-0 === +0
// <- true
Object.is(-0, +0)
// <- false
```

The other difference is when it comes to `NaN`. The `Object.is` method treats `NaN` as equal to `NaN`. This is a behavior we've already observed in maps and sets, which also treats `NaN` as being the same value as `NaN`.

```
NaN === NaN
// <- false
Object.is(NaN, NaN)
// <- true
```

While this may be convenient in some cases, I'd probably go for the more explicit `Number.isNaN` most of the time.

`Object.getOwnPropertySymbols`

This method returns all own property symbols found on an object.

```
var a = {
  [Symbol('b')]: 'c',
  [Symbol('d')]: 'e',
  'f': 'g',
  'h': 'i'
}
Object.getOwnPropertySymbols(a)
// <- [Symbol(b), Symbol(d)]
```

We've already covered `Object.getOwnPropertySymbols` in depth in the symbols dossier. If I were you, I'd read it!

`Object.setPrototypeOf`

Again, something we've covered earlier in the series. One of the articles about proxy traps covers this method tangentially. You can use `Object.setPrototypeOf` to change the prototype of an object.

It is, in fact, the equivalent of setting `__proto__` on runtimes that have that property.

↑ **Back to top**

ES6 Strings (and Unicode,) in Depth

Yo. Here's another edition of ES6 — *"I can't believe they killed off Stringer Bell"* — in Depth series. If you've never been around here before, start with A Brief History of ES6 Tooling. Then, make your way through destructuring, template literals, arrow functions, the spread operator and rest parameters, improvements coming to object literals, the new *classes* sugar on top of prototypes, `let`, `const`, and the *"Temporal Dead Zone"*, iterators, generators, Symbols, Maps, WeakMaps, Sets, and WeakSets, proxies, proxy traps, more proxy traps, reflection, `Number`, `Math`, `Array`, and `Object`. Today we'll be serving updates to the `String` object coming in ES6.

As I did in previous articles on the series, I would love to point out that you should probably set up Babel and follow along the examples with either a REPL or the `babel-node` CLI and a file. That'll make it so much easier for you to **internalize the concepts** discussed in the series. If you aren't the *"install things on my computer"* kind of human, you might prefer to hop on CodePen and then click on the gear icon for JavaScript – *they have a Babel preprocessor which makes trying out ES6 a breeze.* Another alternative that's also quite useful is to use Babel's online REPL – *it'll show you compiled ES5 code to the right of your ES6 code for quick comparison.*

Before getting into it, let me *shamelessly ask for your support* if you're enjoying my ES6 in Depth series. Your contributions will go towards helping me keep up with the schedule, server bills, keeping me fed, and maintaining **Pony Foo** as a veritable source of JavaScript goodies.

Thanks for reading that, and let's go into updates to the `String` object.

Updates to `String`

We've already covered template literals earlier in the series, and you may recall that those can be used to mix strings and variables to produce string output.

```
function greet (name) {
  return `hello ${name}!`
}
greet('ponyfoo')
// <- 'hello ponyfoo!'
```

Besides template literals, strings are getting a numbre of new methods come ES6. These can be categorized as string manipulation methods and unicode related methods.

- String Manipulation
- `String.prototype.startsWith`
- `String.prototype.endsWith`
- `String.prototype.includes`
- `String.prototype.repeat`
- `String.prototype[Symbol.iterator]`
- Unicode
- `String.prototype.codePointAt`
- `String.fromCodePoint`
- `String.prototype.normalize`

We'll begin with the string manipulation methods and then we'll take a look at the unicode related ones.

`String.prototype.startsWith`

A very common question in our code is *"does this string start with this other string?"*. In ES5 we'd ask that question using the `.indexOf` method.

```
'ponyfoo'.indexOf('foo')
// <- 4
'ponyfoo'.indexOf('pony')
// <- 0
'ponyfoo'.indexOf('horse')
// <- -1
```

If you wanted to check if a string started with another one, you'd compare them with `.indexOf` and check whether the *"needle"* starts at the `0` position – the beginning of the string.

```
'ponyfoo'.indexOf('pony') === 0
// <- true
'ponyfoo'.indexOf('foo') === 0
// <- false
'ponyfoo'.indexOf('horse') === 0
// <- false
```

You can now use the more descriptive and terse `.startsWith` method instead.

```
'ponyfoo'.startsWith('pony')
// <- true
'ponyfoo'.startsWith('foo')
// <- false
```

```
'ponyfoo'.startsWith('horse')
// <- false
```

If you wanted to figure out whether a string contains another one starting in a specific location, it would get quite verbose, as you'd need to grab a slice of that string first.

```
'ponyfoo'.slice(4).indexOf('foo') === 0
// <- true
```

The reason why you can't just ask `=== 4` is that this would give you false negatives when the query is found before reaching that index.

```
'foo,foo'.indexOf('foo') === 4
// <- false, because result was 0
```

Of course, you could use the `startIndex` parameter for `indexOf` to get around that. Note that we're still comparing against `4` in this case, because the string wasn't split into smaller parts.

```
'foo,foo'.indexOf('foo', 4) === 4
// <- true
```

Instead of keeping all of these string searching implementation details in your head and writing code that worries too much about the how and not so much about the what, you could just use `startsWith` passing in the optional `startIndex` parameter as well.

```
'foo,foo'.startsWith('foo', 4)
// <- true
```

Then again, it's kind of confusing that the method is called `.startsWith` but we're starting at a non-zero index – that being said it sure beats using `.indexOf` when we actually want a boolean result.

String.prototype.endsWith

This method mirrors `.startsWith` in the same way that `.lastIndexOf` mirrors `.indexOf`. It tells us whether a string ends with another string.

```
'ponyfoo'.endsWith('foo')
// <- true
'ponyfoo'.endsWith('pony')
// <- false
```

Just like `.startsWith`, we have a position index that indicates where the lookup should end. It defaults to the length of the string.

```
'ponyfoo'.endsWith('foo', 7)
// <- true
'ponyfoo'.endsWith('pony', 0)
// <- false
'ponyfoo'.endsWith('pony', 4)
// <- true
```

Yet another method that simplifies a specific use case for `.indexOf` is `.includes`.

String.prototype.includes

You can use `.includes` to figure out whether a string contains another one.

```
'ponyfoo'.includes('ny')
// <- true
'ponyfoo'.includes('sf')
// <- false
```

This is equivalent to the ES5 use case of `.indexOf` where we'd compare its results with `-1` to see if the search string was anywhere to be found.

```
'ponyfoo'.indexOf('ny') !== -1
// <- true
'ponyfoo'.indexOf('sf') !== -1
// <- false
```

Naturally you can also pass in a start index where the search should begin.

```js
'ponyfoo'.includes('ny', 3)
// <- false
'ponyfoo'.includes('ny', 2)
// <- true
```

Let's move onto something that's not an `.indexOf` replacement.

`String.prototype.repeat`

This handy method allows you to repeat a string `count` times.

```js
'na'.repeat(0)
// <- ''
'na'.repeat(1)
// <- 'na'
'na'.repeat(2)
// <- 'nana'
'na'.repeat(5)
// <- 'nananananana'
```

The provided `count` should be a positive finite number.

```js
'na'.repeat(Infinity)
// <- RangeError
'na'.repeat(-1)
// <- RangeError
```

Non-numeric values are coerced into numbers.

```js
'na'.repeat('na')
// <- ''
'na'.repeat('3')
// <- 'nanana'
```

Using `NaN` is as good as `0`.

```js
'na'.repeat(NaN)
// <- ''
```

Decimal values are floored.

```js
'na'.repeat(3.9)
// <- 'nanana', count was floored to 3
```

Values in the `(-1, 0)` range are rounded to `-0` becase `count` is passed through `ToInteger`, as documented by the specification. That step in the specification dictates that `count` be casted with a formula like the one below.

```js
function ToInteger (number) {
  return Math.floor(Math.abs(number)) * Math.sign(number)
}
```

The above translates to `-0` for any values in the `(-1, 0)` range. Numbers below that will throw, and numbers above that won't behave surprisingly, as you can only take `Math.floor` into account for positive values.

```js
'na'.repeat(-0.1)
// <- '', count was rounded to -0
```

A good example use case for `.repeat` may be your typical *"padding"* method. The method shown below takes a multiline string and pads every line with as many `spaces` as desired.

```js
function pad (text, spaces) {
  return text.split('\n').map(line => ' '.repeat(spaces) + line).join('\n')
}
pad('a\nb\nc', 2)
// <- '  a\n  b\n  c'
```

In ES6, strings adhere to the iterable protocol.

`String.prototype[Symbol.iterator]`

Before ES6, you could access each **code unit** *(we'll define these in a second)* in a string via indices — kind of like with arrays. That meant you could loop over **code units** in a string with a `for` loop.

```
var text = 'foo'
for (let i = 0; i < text.length; i++) {
  console.log(text[i])
  // <- 'f'
  // <- 'o'
  // <- 'o'
}
```

In ES6, you could loop over the **code points** *(not the same as **code units**)* of a string using a `for..of` loop, because strings are *iterable*.

```
for (let codePoint of 'foo') {
  console.log(codePoint)
  // <- 'f'
  // <- 'o'
  // <- 'o'
}
```

What is this `codePoint` variable? There is *a not-so-subtle distinction* between **code units** and **code points**. Let's switch protocols and talk about *Unicode*.

Unicode

JavaScript strings are represented using *UTF-16 code units*. Each code unit can be used to represent a code point in the `[U+0000, U+FFFF]` range — also known as the *"basic multilingual plane"* (BMP). You can represent individual code points in the BMP plane using the `'\u3456'` syntax. You could also represent code units in the `[U+0000, U+0255]` using the `\x00..\xff` notation. For instance, `'\xbb'` represents `'»'`, the `187` character, as you can verify by doing `parseInt('bb', 16)` — or `String.fromCharCode(187)`.

For code points beyond `U+FFFF`, you'd represent them as a surrogate pair. That is to say, two contiguous code units. For instance, the horse emoji `''` code point is represented with the `'\ud83d\udc0e'` contiguous code units. In ES6 notation you can also represent code points using the `'\u{1f40e}'` notation *(that example is also the horse emoji)*. Note that the internal representation hasn't changed, so there's **still two code units** behind that code point. In fact, `'\u{1f40e}'.length` evaluates to `2`.

The `'\ud83d\udc0e\ud83d\udc71\u2764'` string found below evaluates to a few emoji.

```
'\ud83d\udc0e\ud83d\udc71\u2764'
// <- ''
```

While that string consists of 5 code units, we know that the length should really be three — as there's only three emoji.

```
'\ud83d\udc0e\ud83d\udc71\u2764'.length
// <- 5
''.length
// <- 5, still
```

Before ES6, JavaScript didn't make any effort to figure out unicode quirks on your behalf — you were pretty much on your own when it came to counting cards *(err, code points)*. Take for instance `Object.keys`, still five code units long.

```
console.log(Object.keys(''))
// <- ['0', '1', '2', '3', '4']
```

If we now go back to our `for` loop, we can observe how this is a problem. We actually wanted `''`, `''`, `' '`, but we didn't get that.

```
var text = ''
for (let i = 0; i < text.length; i++) {
  console.log(text[i])
  // <- '?'
  // <- '?'
  // <- '?'
  // <- '?'
  // <- ' '
}
```

Instead, we got some weird unicode boxes — and that's if we were lucky and looking at Firefox.

```
var text = '🐎👱❤'
for (let i = 0; i < text.length; i++) {
  console.log(text[i])
}
undefined
```

That didn't turn out okay. In ES6 we can use the string iterator to go over the code points instead. The iterators produced by the string iterable are aware of this limitation of looping by code units, and so they *yield code points* instead.

```
for (let codePoint of '') {
  console.log(codePoint)
  // <- ''
  // <- ''
  // <- ' '
}
```

If we want to measure the length, we'd have trouble with the `.length` property, as we saw earlier. We can use the iterator to **split the string into its code points** – *as seen in the `for..of` example we just went over.* That means the unicode-aware length of a string equals the length of the array that contains the sequence of code points produced by an iterator. We could use the spread operator to place the code points in an array, and then pull that array's `.length`.

```
[...''].length
// <- 3
```

Keep in mind that splitting strings into code points isn't enough if you want to be *100% precise* about string length. Take for instance the *"combining overline"* `\u0305` unicode code unit. On its own, this code unit is just an overline.

```
'\u0305'
// <- ''
```

When preceded by another code unit, however, they are **combined together** into a single glyph.

```
'_\u0305'
// <- '_'
'foo\u0305'
// <- 'foo'
```

Attempts to figure out the actual length by counting code points prove **insufficient** – just like using `.length`.

```
'foo\u0305'.length
// <- 4
'foo'.length
// <- 4
[...'foo'].length
// <- 4
```

I was confused about this one as I'm no expert when it comes to unicode. So I went to someone who *is* an expert – Mathias Bynens. He promptly pointed out that – *indeed* – splitting by code points isn't enough. Unlike surrogate pairs like the emojis we've used in our earlier examples, other *grapheme clusters* aren't taken into account by the string iterator.

> @nzgb Exactly. The string iterator iterates over code points, but not grapheme clusters. https://t.co/fTGQUOvMj8

— Mathias Bynens (@mathias) September 13, 2015

In these cases we're out of luck, and we simply have to fall back to regular expressions to correctly calculate the string length. For a comprehensive discussion of the subject I suggest you read his excellent "JavaScript has a Unicode problem" piece.

Let's look at the other methods.

String.prototype.codePointAt

You can use `.codePointAt` to get the base-10 numeric representation of a code point at a given position in a string. Note that the position is indexed by code unit, not by code point. In the example below we print the code points for each of the three emoji in our demo `' '` string.

```
'\ud83d\udc0e\ud83d\udc71\u2764'.codePointAt(0)
// <- 128014
'\ud83d\udc0e\ud83d\udc71\u2764'.codePointAt(2)
// <- 128113
'\ud83d\udc0e\ud83d\udc71\u2764'.codePointAt(4)
// <- 10084
```

Figuring out the indices on your own may prove cumbersome, which is why you should just loop through the string iterator so that figures them out on your behalf. You can then just call `.codePointAt(0)` for each code point in the sequence.

```
for (let codePoint of '\ud83d\udc0e\ud83d\udc71\u2764') {
  console.log(codePoint.codePointAt(0))
  // <- 128014
  // <- 128113
  // <- 10084
}
```

Or maybe just use a combination of spread and `.map`.

```
[...'\ud83d\udc0e\ud83d\udc71\u2764'].map(cp => cp.codePointAt(0))
// <- [128014, 128113, 10084]
```

You could then take the hexadecimal *(base-16)* representation of those base-10 integers and render them on a string using the new unicode code point escape syntax of `\u{codePoint}`. This syntax allows you to represent unicode code points that are beyond the *"basic multilingual plane"* (BMP) — i.e, code points outside the `[U+0000, U+FFFF]` range that are typically represented using the `\u1234` syntax.

Let's start by updating our example to print the hexadecimal version of our code points.

```
for (let codePoint of '\ud83d\udc0e\ud83d\udc71\u2764') {
  console.log(codePoint.codePointAt(0).toString(16))
  // <- '1f40e'
  // <- '1f471'
  // <- '2764'
}
```

You can wrap those in `'\u{codePoint}'` and voilá — *you'll get the emoji out of the string once again.*

```
'\u{1f40e}'
// <- ''
'\u{1f471}'
// <- ''
'\u{2764}'
// <- ' '
```

Yay!

String.fromCodePoint

This method takes in a number and returns a code point. Note how I can use the `0x` prefix with the terse base-16 code points we got from `.codePointAt` moments ago.

```
String.fromCodePoint(0x1f40e)
// <- ''
String.fromCodePoint(0x1f471)
// <- ''
String.fromCodePoint(0x2764)
// <- ' '
```

Obviously, you can just as well use their base-10 counterparts to achieve the same results.

```
String.fromCodePoint(128014)
// <- ''
String.fromCodePoint(128113)
// <- ''
String.fromCodePoint(10084)
// <- ' '
```

You can pass in as many code points as you'd like.

```
String.fromCodePoint(128014, 128113, 10084)
// <- ''
```

As an exercise in futility, we could map a string to their numeric representation of code points, and back to the code points themselves.

```
[...'\ud83d\udc0e\ud83d\udc71\u2764']
  .map(cp => cp.codePointAt(0))
  .map(cp => String.fromCodePoint(cp))
  .join('')
// <- ''
```

Maybe you're feeling like playing a joke on your fellow inmates – *I mean, coworkers*. You can now stab them to death with this piece of code that doesn't really do anything other than converting the string into code points and then spreading those code points as parameters to `String.fromCodePoint`, which in turn restores the original string. *As amusing as it is useless!*

```
String.fromCodePoint(...[
  ...'\ud83d\udc0e\ud83d\udc71\u2764'
].map(cp => cp.codePointAt(0)))
// <- ''
```

Since we're on it, you may've noticed that reversing the string itself would cause issues.

```
'\ud83d\udc0e\ud83d\udc71\u2764'.split('').reverse().join('')
```

The problem is that you're reversing individual code units as opposed to code points.

```
'\ud83d\udc0e\ud83d\udc71\u2764'.split('').reverse().join('')
```

If we were to use the spread operator to split the string by its code points, and then reverse that, the code points would be preserved and the string would be properly reversed.

```
[...'\ud83d\udc0e\ud83d\udc71\u2764'].reverse().join('')
// <- ''
```

This way we avoid breaking code points, but once again keep in mind that this won't work for *all* grapheme clusters, as Mathias pointed out in his tweet.

```
[...'foo\u0305'].reverse().join('')
// <- ' oof'
```

The last method we'll cover today is `.normalize`.

`String.prototype.normalize`

There are different ways to represent strings that look identical to humans, even though their code points differ. Mathias gives an example as follows.

```
'mañana' === 'manana' // E.g., entered as 'man ana'
// <- false
```

What's going on here? We have a combining tilde `` and an `n` on the right, while the left just has an `ñ`. These look alike, but if you look at the code points you'll notice they're different.

```
[...'mañana'].map(cp => cp.codePointAt(0).toString(16))
// <- ['6d', '61', 'f1', '61', '6e', '61']
[...'manana'].map(cp => cp.codePointAt(0).toString(16)) // E.g., 'man ana'
// <- ['6d', '61', '6e', '303', '61', '6e', '61']
```

Just like with the `'foo'` example, the second string has a length of `7`, even though it is `6` glyphs long.

```
'mañana'.length
// <- 6
'manana'.length // E.g., 'man ana'
// <- 7
```

If we normalize the second version, we'll get back the same code points we had in the first version.

```
var normalized = 'manana'.normalize() // E.g., 'man ana'
[...normalized].map(cp => cp.codePointAt(0).toString(16))
// <- ['6d', '61', 'f1', '61', '6e', '61']
normalized.length
// <- 6
```

Just for completeness' sake, note that you can represent these code points using the `'\x6d'` syntax.

```
'\\x' + [...'mañana'].map(cp => cp.codePointAt(0).toString(16)).join('\\x')
// <- '\\x6d\\x61\\xf1\\x61\\x6e\\x61'
'\x6d\x61\xf1\x61\x6e\x61'
// <- 'mañana'
```

We could use `.normalize` on both strings to see if they're really equal.

```
function compare (left, right) {
  return left.normalize() === right.normalize()
}
compare('mañana', 'manana') // E.g., compare 'mañana' with 'man ana'
// <- true
```

Or, to prove the point in something that's a bit more visible to human eyes, let's use the `\x` syntax. Note that you can only use `\x` to represent code units with codes below 256 *(* `\xff` *is* `255` *)*. For anything larger than that we should use the `\u` escape. Such is the case of the `U+0303` combining tilde.

```
compare(
  '\x6d\x61\xf1\x61\x6e\x61',
  '\x6d\x61\x6e\u0303\x61\x6e\x61'
)
// <- true
```

See you tomorrow? *Modules are coming!*

> *Many thanks to Mathias for reviewing drafts of this article,* !

Back to top

ES6 Modules in Depth

Welcome back to ES6 – *"Oh, good. It's not another article about Unicode"* – in Depth series. If you've never been around here before, start with A Brief History of ES6 Tooling. Then, make your way through destructuring, template literals, arrow functions, the spread operator and rest parameters, improvements coming to object literals, the new *classes* sugar on top of prototypes, `let`, `const`, and the *"Temporal Dead Zone"*, iterators, generators, Symbols, Maps, WeakMaps, Sets, and WeakSets, proxies, proxy traps, more proxy traps, reflection, `Number`, `Math`, `Array`, `Object`, and `String`. This morning is about the module system in ES6.

> As I did in previous articles on the series, I would love to point out that you should probably set up Babel and follow along the examples with either a REPL or the `babel-node` CLI and a file. That'll make it so much easier for you to **internalize the concepts** discussed in the series. If you aren't the *"install things on my computer"* kind of human, you might prefer to hop on

> CodePen and then click on the gear icon for JavaScript — *they have a Babel preprocessor which makes trying out ES6 a breeze.* Another alternative that's also quite useful is to use Babel's online REPL — *it'll show you compiled ES5 code to the right of your ES6 code for quick comparison.*

Before getting into it, let me *shamelessly ask for your support* if you're enjoying my ES6 in Depth series. Your contributions will go towards helping me keep up with the schedule, server bills, keeping me fed, and maintaining **Pony Foo** as a veritable source of JavaScript goodies.

Thanks for reading that, and let's dive deep into the ES6 module system.

The ES6 Module System

Before ES6 we really went out of our ways to obtain modules in JavaScript. Systems like RequireJS, Angular's dependency injection mechanism, and CommonJS have been catering to our modular needs for a long time now — *alongside with helpful tools such as Browserify and Webpack*. Still, the year is 2015 and a standard module system was long overdue. As we'll see in a minute, you'll quickly notice that ES6 modules have been heavily influenced by CommonJS. We'll look at `export` and `import` statements, and see how ES6 modules are compatible with CommonJS, as we'll go over throughout this article.

Today we are going to cover a few areas of the ES6 module system.

- Strict Mode
- `export`
- Exporting a Default Binding
- Named Exports
- Bindings, Not Values
- Exporting Lists
- Best Practices and `export`
- `import`
- Importing Default Exports
- Importing Named Exports
- `import` All The Things

Strict Mode

In the ES6 module system, strict mode is turned on by default. In case you don't know what strict mode is, it's just a stricter version of the language that disallows lots of bad parts of the language. It enables compilers to perform better by disallowing non-sensical behavior in user code, too. The following is a summary extracted from changes documented in the strict mode article on MDN.

- Variables can't be left undeclared
- Function parameters must have unique names *(or are considered syntax errors)*
- `with` is forbidden
- Errors are thrown on assignment to *read-only* properties
- Octal numbers like `00840` are syntax errors
- Attempts to `delete` undeletable properties `throw` an error
- `delete prop` is a syntax error, instead of assuming `delete global[prop]`
- `eval` doesn't introduce *new* variables into its surrounding scope
- `eval` and `arguments` can't be bound or assigned to
- `arguments` doesn't magically track changes to method parameters
- `arguments.callee` throws a `TypeError`, no longer supported
- `arguments.caller` throws a `TypeError`, no longer supported
- Context passed as `this` in method invocations is not *"boxed" (forced)* into becoming an `Object`
- No longer able to use `fn.caller` and `fn.arguments` to access the JavaScript stack
- Reserved words *(e.g `protected`, `static`, `interface`, etc)* cannot be bound

In case it isn't immediately obvious — you should `'use strict'` in all the places. Even though it's becoming de-facto in ES6, it's still a good practice to use `'use strict'` everywhere in ES6. I've been doing it for a long time and never looked back!

Let's now get into `export`, our first ES6 modules keyword of the day!

`export`

In CommonJS, you export values by exposing them on `module.exports`. As seen in the snippet below, you could expose anything from a value type to an object, an array, or a function.

```
module.exports = 1
module.exports = NaN
module.exports = 'foo'
module.exports = { foo: 'bar' }
module.exports = ['foo', 'bar']
module.exports = function foo () {}
```

ES6 modules are files that `export` an API — just like CommonJS modules. Declarations in ES6 modules are scoped to that module, just like with CommonJS. That means that any variables declared inside a module aren't available to other modules unless they're *explicitly exported* as part of the module's API *(and then imported in the module that wants to access them)*.

Exporting a Default Binding

You can mimic the CommonJS code we just saw by changing `module.exports =` into `export default`.

```
export default 1
export default NaN
export default 'foo'
export default { foo: 'bar' }
export default ['foo', 'bar']
export default function foo () {}
```

Contrary to CommonJS, `export` statements can only be placed at the top level in ES6 modules — even if the method they're in would be immediately invoked when loading the module. Presumably, this limitation exists to make it easier for compilers to interpret ES6 modules, but it's also a good limitation to have as there aren't that many good reasons to dynamically define and expose an API based on method calls.

```
function foo () {
  export default 'bar' // SyntaxError
}
foo()
```

There isn't just `export default`, you can also use *named exports*.

Named Exports

In CommonJS you don't even have to assign an object to `module.exports` first. You could just tack properties onto it. It's still a single binding being exported — *whatever properties the* `module.exports` *object ends up holding*.

```
module.exports.foo = 'bar'
module.exports.baz = 'ponyfoo'
```

We can replicate the above in ES6 modules by using the named exports syntax. Instead of assigning to `module.exports` like with CommonJS, in ES6 you can declare bindings you want to `export`. Note that the code below cannot be refactored to extract the variable declarations into standalone statements and then just `export foo`, as that'd be a syntax error. Here again, we see how ES6 modules favor static analysis by being rigid in how the declarative module system API works.

```
export var foo = 'bar'
export var baz = 'ponyfoo'
```

It's important to keep in mind that we are exporting *bindings*.

Bindings, Not Values

An important point to make is that ES6 modules export bindings, not values or references. That means that a `foo` variable you export would be bound into the `foo` variable on the module, and its value would be subject to changes made to `foo`. I'd advise against changing the public interface of a module after it has initially loaded, though.

If you had an `./a` module like the one found below, the `foo` export would be bound to `'bar'` for 500ms and then change into `'baz'`.

```
export var foo = 'bar'
setTimeout(() => foo = 'baz', 500)
```

Besides a "default" binding and individual bindings, you could also export lists of bindings.

Exporting Lists

As seen in the snippet below, ES6 modules let you `export` *lists* of named top-level members.

```
var foo = 'ponyfoo'
var bar = 'baz'
export { foo, bar }
```

If you'd like to export something with a different name, you can use the `export { foo as bar }` syntax, as shown below.

```
export { foo as ponyfoo }
```

You could also specify `as default` when using the named member list `export` declaration flavor. The code below is the same as doing `export default foo` and `export bar` afterwards — but in a single statement.

```
export { foo as default, bar }
```

There's many benefits to using only `export default`, and only at the bottom of your module files.

Best Practices and `export`

Having the ability to define named exports, exporting a list with aliases and whatnot, and also exposing a a *"default"* `export` will mostly introduce confusion, and *for the most part* I'd encourage you to use `export default` — and to do that at the end of your module files. You could just call your API object `api` or name it after the module itself.

```
var api = {
  foo: 'bar',
  baz: 'ponyfoo'
}
export default api
```

One, the exported interface of a module becomes immediately obvious. Instead of having to crawl around the module and put the pieces together to figure out the API, you just scroll to the end. Having a clearly defined place where your API is exported also makes it easier to reason about the methods and properties your modules export.

Two, you don't introduce confusion as to whether `export default` or a named export — or a list of named exports *(or a list of named exports with aliases...)* — should be used in any given module. There's a guideline now — just use `export default` everywhere and be done with it.

Three, consistency. In the CommonJS world *it is usual* for us to export a single method from a module, and that's it. Doing so with named exports is impossible as you'd effectively be exposing an object with the method in it, unless you were using the `as default` decorator in the `export` list flavor. The `export default` approach is more versatile because it allows you to `export` just one thing.

Four, — and this is really a reduction of points made earlier — the `export default` statement at the bottom of a module makes it immediately obvious what the exported API is, what its methods are, and generally easy for the module's consumer to `import` its API. When paired with the convention of *always using `export default` and always doing it at the end of your modules*, you'll note using the ES6 module system to be painless.

Now that we've covered the `export` API and its caveats, let's jump over to `import` statements.

`import`

These statements are the counterpart of `export`, and they can be used to load a module from another one — first and foremost. The way modules are loaded is *implementation-specific*, and at the moment no browsers implement module loading. This way you can write spec-compliant ES6 code today while smart people figure out how to deal with module loading in browsers. Transpilers like Babel are able to concatenate modules with the aid of a module system like CommonJS. That means `import` statements in Babel follow *mostly* the same semantics as `require` statements in CommonJS.

Let's take `lodash` as an example for a minute. The following statement simply loads the Lodash module from our module. It doesn't create any variables, though. It **will execute** any code in the top level of the `lodash` module, though.

```
import 'lodash'
```

Before going into importing bindings, let's also make a note of the fact that `import` statements, — much like with `export` — are only allowed in the top level of your module definitions. This can help transpilers implement their module loading capabilities, as well as help other static analysis tools parse your codebase.

Importing Default Exports

In CommonJS you'd `import` something using a `require` statement, like so.

```
var _ = require('lodash')
```

To import the default exported binding from an ES6 module, you just have to pick a name for it. The syntax is a bit different than declaring a variable because you're importing a *binding*, and also to make it easier on static analysis tools.

```
import _ from 'lodash'
```

You could also import named exports and alias them.

Importing Named Exports

The syntax here is very similar to the one we just used for default exports, you just add some braces and pick any named exports you want. Note that this syntax is similar to the destructuring assignment syntax, but also bit different.

```
import {map, reduce} from 'lodash'
```

Another way in which it differs from destructuring is that you could use aliases to rename imported bindings. You can mix and match aliased and non-aliased named exports as you see fit.

```
import {cloneDeep as clone, map} from 'lodash'
```

You can also mix and match named exports and the default export. If you want it inside the brackets you'll have to use the `default` name, which you can alias; or you could also just mix the default import side-by-side with the named imports list.

```
import {default, map} from 'lodash'
import {default as _, map} from 'lodash'
import _, {map} from 'lodash'
```

Lastly, there's the `import *` flavor.

`import` All The Things

You could also import the namespace object for a module. Instead of importing the named exports or the default value, it imports all the things. Note that the `import *` syntax must be followed by an alias where all the bindings will be placed. If there was a default export, it'll be placed in `alias.default`.

```
import * as _ from 'lodash'
```

That's about it!

Conclusions

Note that you can use ES6 modules today through the Babel compiler while leveraging CommonJS modules. What's great about that is that you can actually interoperate between CommonJS and ES6 modules. That means that even if you `import` a module that's written in CommonJS it'll actually work.

The ES6 module system looks great, and it's one of the most important things that had been missing from JavaScript. I'm hoping they come up with a finalized module loading API and browser implementations soon. The many ways you can `export` or `import` bindings from a module don't introduce as much versatility as they do added complexity for little gain, but time will tell whether all the extra API surface is as convenient as it is large.

↑ **Back to top**

ES6 Promises in Depth

Welcome back to ES6 – *"Dude, we already had those!"* – in Depth series. If you've never been around here before, start with A Brief History of ES6 Tooling. Then, make your way through destructuring, template literals, arrow functions, the spread operator and rest parameters, improvements coming to object literals, the new *classes* sugar on top of prototypes, `let`, `const`, and the *"Temporal Dead Zone"*, iterators, generators, Symbols, Maps, WeakMaps, Sets, and WeakSets, proxies, proxy traps, more proxy traps, reflection, `Number`, `Math`, `Array`, `Object`, `String`, and the module system. This morning is about the `Promise` API in ES6.

PDF Note: Some of the graphics on the website are multi-part GIFs; each segment is included here for clarity, clumsy as it may appear in print.

> As I did in previous articles on the series, I would love to point out that you should probably set up Babel and follow along the examples with either a REPL or the `babel-node` CLI and a file. That'll make it so much easier for you to **internalize the concepts** discussed in the series. If you aren't the *"install things on my computer"* kind of human, you might prefer to hop on CodePen and then click on the gear icon for JavaScript – *they have a Babel preprocessor which makes trying out ES6 a breeze.* Another alternative that's also quite useful is to use Babel's online REPL – *it'll show you compiled ES5 code to the right of your ES6 code for quick comparison.*

Before getting into it, let me *shamelessly ask for your support* if you're enjoying my ES6 in Depth series. Your contributions will go towards helping me keep up with the schedule, server bills, keeping me fed, and maintaining **Pony Foo** as a veritable source of JavaScript goodies.

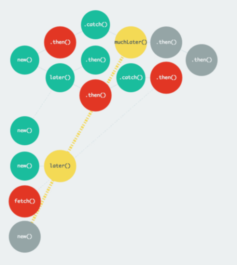
Thanks for reading that, and let's go into *Promises* in ES6. Before reading this article you might want to read about arrow functions, as they're heavily used throughout the article; and generators, as they're somewhat related to the concepts discussed here.

I also wanted to mention *Promisees* – a promise visualization playground I made last week. It offers in-browser visualizations of how promises unfold. You can run those visualizations step by step, displaying how promises in any piece of code work. You can also record a gif of those visualizations, a few of which I'll be displaying here in the article. Hope it helps!


.catch()
.then()
muchLater()
.then()
new()
.then()
.then()
later()
.catch()
.then()
.then()
new()


.catch()
.then()
muchLater()
.then()
new()
.then()
.then()
later()
.catch()
.then()
.then()
new()
new()
later()


.catch()
.then()
muchLater()
.then()
new()
.then()
.then()
later()
.catch()
.then()
.then()
new()
new()
later()



If that animation looks insanely complicated to you, read on!

Promises are a very involved paradigm, so we'll take it slow.

Here's a table of contents with the topics we'll cover in this article. Feel free to skip topics you're comfortable about.

Shall we?

What is a `Promise`?

Promises are usually vaguely defined as *"a proxy for a value that will eventually become available"*. They can be used for both synchronous and asynchronous code flows, although they make asynchronous flows easier to reason about – once you've mastered promises, that is.

Consider as an example the *upcoming* `fetch` API. This API is a simplification of `XMLHttpRequest`. It aims to be super simple to use for the most basic use cases: making a `GET` request against a resource relative to the current page over `http(s)` – it also provides a comprehensive API that caters to advanced use cases as well, but that's not our focus for now. In its most basic incarnation, you can make a request for `GET foo` like so.

```
fetch('foo')
```

The `fetch('foo')` statement doesn't seem all that exciting. It makes a *"fire-and-forget"* `GET` request against `foo` relative to the resource we're currently on. The `fetch` method returns a `Promise`. You can chain a `.then` callback that will be executed once the `foo` resource finishes loading.

```
fetch('foo').then(response => /* do something */)
```

Promises offer an alternative to callbacks and events.

Callbacks and Events

If the `fetch` API used callbacks, you'd get one last parameter that then gets executed whenever fetching ends. Typical asynchronous code flow conventions dictate that we allocate the first parameter for errors *(that may or may not occur)* during the *fetching process*. The rest of the parameters can be used to pass in resulting data. Often, a single parameter is used.

```
fetch('foo', (err, res) => {
  if (err) {
    // handle error
  }
  // handle response
})
```

The callback wouldn't be invoked until the `foo` resource has been fetched, so its execution remains asynchronous and non-blocking. Note that in this model you could only specify **a single callback**, and that callback would be responsible for *all functionality* derived from the response.

Another option might have been to use an *event-driven* API model. In this model the object returned by `fetch` would be able to listen `.on` events, binding as many event handlers as needed for any events. Typically there's an `error` event for when things go awry and a `data` event that's called when the operation completes successfully.

```
fetch('foo')
  .on('error', err => {
    // handle error
  })
  .on('data', res => {
    // handle response
  })
```

In this case, errors usually end up in hard exceptions if no event listener is attached – but that depends on what event emitter implementation is used. Promises are a bit different.

Gist of a `Promise`

Instead of binding event listeners through `.on`, promises offer a slightly different API. The snippet of code shown below displays the actual API of the `fetch` method, which returns a `Promise` object. Much like with events, you can bind as many listeners as you'd like with both `.catch` and `.then`. Note how there's no need for an event type anymore with the declarative methods used by promises.

```
var p = fetch('foo')
p.then(res => {
  // handle response
})
p.catch(error => {
  // handle error
})
```

See [this example][(http://buff.ly/1KtWGUD)] on Promisees

Also note that `.then` is able to register a reaction to rejections as its second argument. The above could be expressed as the following piece of code.

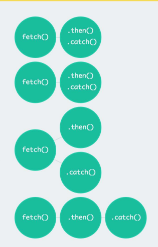
```
fetch('foo')
  .then(
    res => {
```

```
    // handle response
  },
  err => {
    // handle error
  }
)
```

See [this example][(http://buff.ly/1V8xpHI)] on Promisees

Just like you can omit the error reaction in `.then(fulfillment)`, you can also omit the reaction to *fulfillment*. Using `.then(null, rejection)` is equivalent to `.catch(rejection)`. Note that `.then` and `.catch` return **a new promise every time**. That's important because chaining can have wildly different results depending on where you append a `.then` or a `.catch` call onto. See the following example to understand the difference.

```
// here both callbacks are chained onto `fetch('foo')`
fetch('foo').then(res => {}, error => {})

// this example is identical to the previous one
var p = fetch('foo')
p.then(res => {}, error => {})

// even though semantics are different, this one is also the same
var p2 = fetch('foo')
p2.then(res => {})
p2.catch(error => {})

// here, though, `.catch` is chained onto `.then`
// and not onto the original promise
fetch('foo').then(res => {}).catch(error => {})
```



We'll get more in depth into these two methods in a bit. Let's look at a brief history of promises before doing that.

Promises in Time

Promises aren't all that new. Like *most things in computer science*, the earliest mention of Promises can be traced all the way back to the late seventies. According to the *Internet*, they made their first appearance in JavaScript in 2007 – in a library called `MochiKit`. Then `Dojo` adopted it, and `jQuery` followed shortly after that.

Then the Promises/A+ specification came out from the CommonJS group *(now famous for their CommonJS module specification)*. In its earliest incarnations, Node.js shipped with promises. Some time later, they were removed from core and everyone switched over to callbacks. Now, promises ship with the ES6 standard and V8 has already implemented them a while back.

> The ES6 standard implements **Promises/A+** natively. In the latest versions of Node.js you can use promises without any libraries. They're also available on Chrome 32+, Firefox 29+, and Safari 7.1+.

Shall we go back to the `Promise` API?

Then, Again

Going back to our example – here's some of the code we had. In the simplest use case, this is all we wanted.

```
fetch('foo').then(res => {
  // handle response
})
```

What if an error happens in one of the reactions passed to `.then`? You can catch those with `.catch`. The example in the snippet below logs the error caught when trying to access `prop` from the *undefined* `a` *property* in `res`.

```
fetch('foo')
  .then(res => res.a.prop.that.does.not.exist)
  .catch(err => console.error(err.message))
// <- 'Cannot read property "prop" of undefined'
```

Note that *where* you tack your reactions onto matters. The following example **won't** print the `err.message` twice – only once. That's because no errors happened in the first `.catch`, so the rejection branch for that promise wasn't executed. Check out the Promisee for a visual explanation of the code below.

```
fetch('foo')
  .then(res => res.a.prop.that.does.not.exist)
  .catch(err => console.error(err.message))
  .catch(err => console.error(err.message))
// <- 'Cannot read property "prop" of undefined'
```

In contrast, the snippet found below *will* print the `err.message` twice. It works by saving a reference to the promise returned by `.then`, and then tacking two `.catch` reactions onto it. The second `.catch` in the previous example was capturing errors produced in the promise returned from the first `.catch`, while in this case both `.catch` branch off of `p`.

```
var p = fetch('foo').then(res => res.a.prop.that.does.not.exist)
p.catch(err => console.error(err.message))
p.catch(err => console.error(err.message))
// <- 'Cannot read property "prop" of undefined'
// <- 'Cannot read property "prop" of undefined'
```

Here's another example that puts that difference the spotlight. The second catch is triggered this time because it's bound to the rejection branch on the first `.catch`.

```
fetch('foo')
  .then(res => res.a.prop.that.does.not.exist)
  .catch(err => { throw new Error(err.message) })
  .catch(err => console.error(err.message))
// <- 'Cannot read property "prop" of undefined'
```

If the first `.catch` call didn't return anything, then nothing would be printed.

```
fetch('foo')
  .then(res => res.a.prop.that.does.not.exist)
  .catch(err => {})
  .catch(err => console.error(err.message))
// nothing happens
```

We should observe, then, that promises can be chained "arbitrarily", that is to say: as we just saw, you can save a reference to any point in the promise chain and then tack more promises on top of it. This is one of the fundamental points to understanding promises.

> You can save a reference to any point in the promise chain.

In fact, the last example can be represented as shown below. This snippet makes it much easier to understand what we've discussed so far. Glance over it and then I'll give you some bullet points.

```
var p1 = fetch('foo')
var p2 = p1.then(res => res.a.prop.that.does.not.exist)
var p3 = p2.catch(err => {})
var p4 = p3.catch(err => console.error(err.message))
```

Good boy! Have some bullet points. Or you could just look at the Promisees visualization.

1. `fetch` returns a **brand new** `p1` promise
2. `p1.then` returns a **brand new** `p2` promise
3. `p2.catch` returns a **brand new** `p3` promise
4. `p3.catch` returns a **brand new** `p4` promise
5. When `p1` is settled *(fulfilled)*, the `p1.then` reaction is executed
6. After that `p2`, which is awaiting the pending result of `p1.then` is settled
7. Since `p2` was *rejected*, `p2.catch` reactions are executed *(instead of the* `p2.then` *branch)*
8. The `p3` promise from `p2.catch` is *fulfilled*, even though it doesn't produce any value nor an error
9. Because `p3` succeeded, `p3.catch` is never executed – the `p3.then` branch would've been used instead

You should think of promises as **a tree structure**. It all starts with a single promise, which we'll later see how to construct. You then add a branch with `.then` or `.catch`. You can tack as many `.then` or `.catch` calls as you want onto each branch, creating new branches, and so on.

Creating a Promise From Scratch

You should now understand how promises work like a tree where you can add branches where you need them, as you need them. But how do you create a promise from scratch? Writing these kinds of `Promise` tutorials is hard because its a chicken and egg situation. People hardly have a need to create a promise from scratch, since libraries usually take care of that. In this article, for instance, I purposely started explaining things using `fetch`, which internally creates a new promise object. Then, each call to `.then` or `.catch` on the promise created by fetch also creates a promise internally, and those promises depend on their parent when it comes to deciding whether the fulfillment branch or the rejection branch should be executed.

Promises can be created from scratch by using `new Promise(resolver)`. The `resolver` parameter is a method that will be used to resolve the promise. It takes two arguments, a `resolve` method and a `reject` method. These promises are fulfilled and rejected, respectively, on the next tick – as seen on Promisees.

```
new Promise(resolve => resolve()) // promise is fulfilled
new Promise((resolve, reject) => reject()) // promise is rejected
```

Resolving and rejecting promises without a value isn't that useful, though. Usually promises will resolve to some `result`, like the response from an AJAX call as we saw with `fetch`. Similarly, you'll probably want to state the `reason` for your rejections – typically using an `Error` object. The code below codifies what you've just read (see the visualization, too).

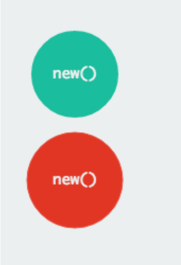
```
new Promise(resolve => resolve({ foo: 'bar' }))
  .then(result => console.log(result))
  // <- { foo: 'bar' }

new Promise((resolve, reject) =>
  reject(new Error('failed to deliver on my promise to you')))
  .catch(reason => console.log(reason))
  // <- Error: failed to deliver on my promise to you
```

As you may have guessed, there's nothing inherently synchronous about promises. Fulfillment and rejection can both be completely asynchronous. That's the whole point of promises! The promise below is fulfilled after two seconds elapse.

```
new Promise(resolve => setTimeout(resolve, 2000))
```

It's important to note that only the first call made to either of these methods will have an impact – once a promise is settled, it's result can't change. The example below creates a promise that's fulfilled in the alloted time or rejected after a generous timeout (visualization).

```
function resolveUnderThreeSeconds (delay) {
  return new Promise(function (resolve, reject) {
    setTimeout(resolve, delay)
    setTimeout(reject, 3000)
  })
}
resolveUnderThreeSeconds(2000) // resolves!
resolveUnderThreeSeconds(7000) // fulfillment took so long, it was rejected.
```

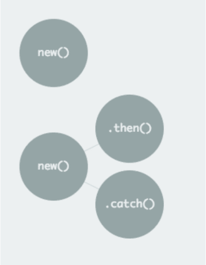


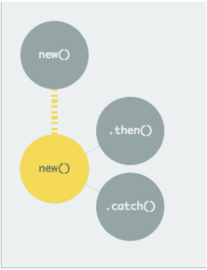
See the promises unfold with this animation

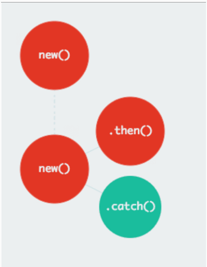
Besides returning resolution values, you could also resolve with *another promise*. What happens in those cases? In the following snippet we create a promise `p` that will be rejected in three seconds. We also create a promise `p2` that will be resolved with `p` in a second. Since `p` is still two seconds out, resolving `p2` won't have an immediate effect. Two seconds later, when `p` is rejected, `p2` will be rejected as well, with the same rejection reason that was provided to `p`.

```
var p = new Promise(function (resolve, reject) {
  setTimeout(() => reject(new Error('fail')), 3000)
})
var p2 = new Promise(function (resolve, reject) {
  setTimeout(() => resolve(p), 1000)
})
p2.then(result => console.log(result))
p2.catch(error => console.log(error))
// <- Error: fail
```

In the animation shown below we can observe how `p2` becomes blocked – *marked in yellow* – waiting for a settlement in `p`.







Note that you this behavior is only possible for fulfillment branches using `resolve`. If you try to replicate the same behavior with `reject` you'll find that the `p2` promise is just rejected with the `p` promise as the rejection `reason`.

Using `Promise.resolve` and `Promise.reject`

Sometimes you want to create a Promise but you don't want to go through the trouble of using the constructor. The following statement creates a promise that's fulfilled with a result of `'foo'`.

```
new Promise(resolve => resolve('foo'))
```

If you already know the value a promise should be fulfilled with, you can use `Promise.resolve` instead. The following statement is equivalent to the previous one.

```
Promise.resolve('foo')
```

Similarly, if you already know the rejection reason, you can use `Promise.reject`. The next statement creates a promise that's going to settle into a rejection, with `reason`.

```
Promise.reject(reason)
```

What else should we know about settling a promise?

Settling a Promise

Promises can exist in three states: pending, fulfilled, and rejected. Pending is the default state. From there, a promise can be *"settled"* into either fulfillment or rejection. Once a promise is settled, all reactions that are waiting on it are evaluated. Those on the correct branch – *`.then` for fulfillment and `.catch` for rejections* – are executed.

From this point on, the promise is *settled*. If at a later point in time another reaction is chained onto the settled promise, the appropriate branch for that reaction is executed in the next tick of the program. In the example below, `p` is resolved with a value of `100` after two seconds. Then, `100` is printed onto the screen. Two seconds later, another `.then` branch is added onto `p`, but since `p` has already fulfilled, the new branch gets executed right away.

```
var p = new Promise(function (resolve, reject) {
  setTimeout(() => resolve(100), 2000)
})
p.then(result => console.log(result))
// <- 100

setTimeout(() => p.then(result => console.log(result * 20)), 4000)
// <- 2000
```

A promise can return another promise – this is what enables and powers most of their asynchronous behavior. In the previous section, when creating a promise from scratch, we saw that we can `resolve` with another promise. We can also return promises when calling `.then`.

Paying a Promise with another Promise

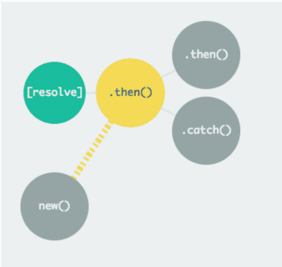
The example below shows how we use a promise and `.then` another promise that will only be settled once the returned promise also settles. Once that happens, we get back the response from the wrapped promise, and we use the `res.url` to figure out what random article we were graced with.

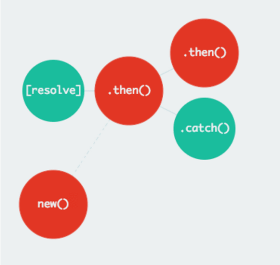
```
fetch('foo')
  .then(response => fetch('/articles/random'))
  .then(response => console.log(response.url))
// <- 'http://ponyfoo.com/articles/es6-symbols-in-depth'
```

Obviously, in the real world, your second `fetch` would probably depend on the response from the first one. Here's another example of returning a promise, where we randomly `fulfill` or `reject` after a second.

```
var p = Promise.resolve()
  .then(data => new Promise(function (resolve, reject) {
    setTimeout(Math.random() >
0.5 ? resolve : reject, 1000)
  }))

p.then(data => console.log('okay!'))
p.catch(data => console.log('boo!'))
```

The animation for this one is super fun!





Okay it's **not** *that fun*. I did have fun making the Promisees tool itself!

Transforming Values in Promises

You're not just limited to returning other promises from your `.then` and `.catch` callbacks. You could also return values, transforming what you had. The example below first creates a promise fulfilled with `[1, 2, 3]` and then has a fulfillment branch on top of that which maps thoes values into `[2, 4, 6]`. Calling `.then` on that branch of the promise will produce the doubled values.

```
Promise.resolve([1, 2, 3])
  .then(values => values.map(value => value * 2))
  .then(values => console.log(values))
  // <- [2, 4, 6]
```

Note that you can do the same thing in rejection branches. An interesting fact that may catch your eye is that if a `.catch` branch goes smoothly without errors, then it will be fulfilled with the returned value. That means that if you still want to have an error for that branch, you should `throw` again. The following piece of code takes an internal error and **masks it** behind a generic *"Internal Server Error"* message as to not leak off potentially dangerous information to its clients (visualization).

```
Promise.reject(new Error('Database ds.214.53.4.12 connection timeout!'))
  .catch(error => { throw new Error('Internal Server Error') })
  .catch(error => console.info(error))
  // <- Error: Internal Server Error
```

Mapping promise results is particularly useful when dealing with multiple concurrent promises. Let's see how that looks like.

Leveraging `Promise.all` and `Promise.race`

A tremendously common scenario – even more so for those used to Node.js – is to have a dependency on things A and B before being able to do thing C. I'll proceed that lousy description of the scenario with multiple code snippets. Suppose you wanted to pull the homepage for both Google and Twitter, and then print out the length of each of their responses. Here's how that looks in the most näive approach possible, with a hypothetical `request(url, done)` method.

```
request('https://google.com', function (err, goog) {
  request('https://twitter.com', function (err, twit) {
    console.log(goog.length, twit.length)
  })
})
```

Of course, that's going to run in series you say! Why would we wait on Google's response before pulling Twitter's? The following piece fixes the problem. It's also ridiculously long, though, right?

```
var results = {}
request('https://google.com', function (err, goog) {
  results.goog = goog
  done()
```

```
})
request('https://twitter.com', function (err, twit) {
  results.twit = twit
  done()
})
function done () {
  if (Object.keys(results).length < 2) {
    return
  }
  console.log(results.goog.length, results.twit.length)
}
```

Since nobody wants to be writing code like that, utility libraries like `async` and `contra` make this much shorter for you. You can use `contra.concurrent` to run these methods at the same time and execute a callback once they all ended. Here's how that'd look like.

```
contra.concurrent({
  goog: function (next) {
    request('https://google.com', next)
  }
  twit: function (next) {
    request('https://twitter.com', next)
  }
}, function (err, results) {
  console.log(results.goog.length, results.twit.length)
})
```

For the very common *"I just want a method that appends that magical `next` parameter at the end"* use case, there's also `contra.curry` *(equivalent of `async.apply`)* to make the code even shorter.

```
contra.concurrent({
  goog: contra.curry(request, 'https://google.com'),
  twit: contra.curry(request, 'https://twitter.com')
}, function (err, results) {
  console.log(results.goog.length, results.twit.length)
})
```

Promises already make the "run this after this other thing in series" use case very easy, using `.then` as we saw in several examples earlier. For the *"run these things concurrently"* use case, we can use `Promise.all` (visualization here).

```
Promise.all([
  fetch('/'),
  fetch('foo')
])
  .then(responses => responses.map(response => response.statusText))
  .then(status => console.log(status.join(', ')))
  // <- 'OK, Not Found'
```

Note that even if a single dependency is rejected, the `Promise.all` method will be rejected entirely as well.

```
Promise.all([
  Promise.reject(),
  fetch('/'),
  fetch('foo')
])
  .then(responses => responses.map(response => response.statusText))
  .then(status => console.log(status.join(', ')))
  // nothing happens
```

In summary, `Promise.all` has two possible outcomes.

- Settle with *a single* rejection `reason` as soon as one of its dependencies is rejected
- Settle with *all* fulfillment `results` as soon as all of its dependencies are fulfilled

Then there's `Promise.race`. This is a similar method to `Promise.all`, except the first promise to settle will "win" the race, and its value will be passed along to branches of the race. If you run the visualization for the following piece of code a few times, you'll notice that this race doesn't have a clear winner. It depends on the server and the network!

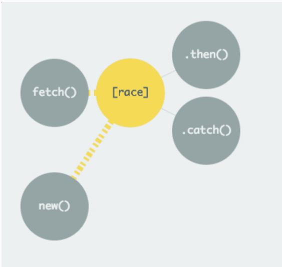
```
Promise.race([
  fetch('/'),
  fetch('foo')
])
  .then(response => console.log(response.statusText))
```

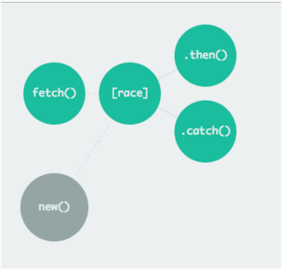
```
// <- 'OK', or maybe 'Not Found'.
```

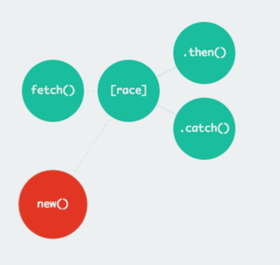
Rejections will also finish the race, and the race promise will be rejected. As a closing note we may indicate that this could be useful for scenarios where we want to time out a promise we otherwise have no control over. For instance, the following race does make sense.

```
var p = Promise.race([
  fetch('/resource-that-may-take-a-while'),
  new Promise(function (resolve, reject) {
    setTimeout(() => reject(new Error('request timeout')), 5000)
  })
])
p.then(response => console.log(response))
p.catch(error => console.log(error))
```

To close this article, I'll leave you with a visualization. It shows the race between a resource and a timeout as shown in the code above.







Here's hoping I didn't make promises even harder to understand for you!

↑ Back to top

ES6 Overview in 350 Bullet Points

My ES6 in Depth series consists of 24 articles covering most syntax changes and features coming in ES6. This article aims to summarize all of those, providing you with practical insight into most of ES6, so that you can quickly get started. Links to the articleshere in ES6 in Depth are included so that you can easily go deeper on any topic you're interested in.

I heard you like bullet points, so I made an article containing hundreds of those bad boys. To kick things off, here's a table of contents with all the topics covered. It has bullet points in it – **obviously**. Note that if you want these concepts to permeate your brain, you'll have a much better time learning the subject by going through the in-depth series and playing around, experimenting with ES6 code yourself.

Table of Contents for the Bullet Points

Apologies about that long table of contents, and here we go.

Introduction

- ES6 — also known as Harmony, `es-next`, ES2015 — is the latest finalized specification of the language
- The ES6 specification was finalized in **June 2015**, *(hence ES2015)*
- Future versions of the specification will follow the `ES[YYYY]` pattern, e.g ES2016 for ES7
- **Yearly release schedule**, features that don't make the cut take the next train
- Since ES6 pre-dates that decision, most of us still call it ES6
- Starting with ES2016 (ES7), we should start using the `ES[YYYY]` pattern to refer to newer versions
- Top reason for naming scheme is to pressure browser vendors into quickly implementing newest features

(back to table of contents)

Tooling

- To get ES6 working today, you need a **JavaScript-to-JavaScript** *transpiler*
- Transpilers are here to stay
- They allow you to compile code in the latest version into older versions of the language
- As browser support gets better, we'll transpile ES2016 and ES2017 into ES6 and beyond
- We'll need better source mapping functionality
- They're the most reliable way to run ES6 source code in production today *(although browsers get ES5)*
- Babel *(a transpiler)* has a killer feature: **human-readable output**
- Use `babel` to transpile ES6 into ES5 for static builds
- Use `babelify` to incorporate `babel` into your Gulp, Grunt, or `npm run` build process
- Use Node.js `v4.x.x` or greater as they have decent ES6 support baked in, thanks to `v8`
- Use `babel-node` with any version of `node`, as it transpiles modules into ES5
- Babel has a thriving ecosystem that already supports some of ES2016 and has plugin support
- Read A Brief History of ES6 Tooling

(back to table of contents)

Assignment Destructuring

- `var {foo} = pony` is equivalent to `var foo = pony.foo`
- `var {foo: baz} = pony` is equivalent to `var baz = pony.foo`
- You can provide default values, `var {foo='bar'} = baz` yields `foo: 'bar'` if `baz.foo` is `undefined`
- You can pull as many properties as you like, aliased or not
- `var {foo, bar: baz} = {foo: 0, bar: 1}` gets you `foo: 0` and `baz: 1`
- You can go deeper. `var {foo: {bar}} = { foo: { bar: 'baz' } }` gets you `bar: 'baz'`
- You can alias that too. `var {foo: {bar: deep}} = { foo: { bar: 'baz' } }` gets you `deep: 'baz'`
- Properties that aren't found yield `undefined` as usual, e.g: `var {foo} = {}`
- Deeply nested properties that aren't found yield an error, e.g: `var {foo: {bar}} = {}`
- It also works for arrays, `[a, b] = [0, 1]` yields `a: 0` and `b: 1`
- You can skip items in an array, `[a, , b] = [0, 1, 2]`, getting `a: 0` and `b: 2`
- You can swap without an *"aux"* variable, `[a, b] = [b, a]`
- You can also use destructuring in function parameters
- Assign default values like `function foo (bar=2) {}`
- Those defaults can be objects, too `function foo (bar={ a: 1, b: 2 }) {}`
- Destructure `bar` completely, like `function foo ({ a=1, b=2 }) {}`
- Default to an empty object if nothing is provided, like `function foo ({ a=1, b=2 } [redacted]) {}`
- Read ES6 JavaScript Destructuring in Depth

(back to table of contents)

Spread Operator and Rest Parameters

- Rest parameters is a better `arguments`
- You declare it in the method signature like `function foo (...everything) {}`
- `everything` is an array with all parameters passed to `foo`

- You can name a few parameters before `...everything`, like `function foo (bar, ...rest) {}`
- Named parameters are excluded from `...rest`
- `...rest` must be the last parameter in the list
- Spread operator is better than magic, also denoted with `...` syntax
- Avoids `.apply` when calling methods, `fn(...[1, 2, 3])` is equivalent to `fn(1, 2, 3)`
- Easier concatenation `[1, 2, ...[3, 4, 5], 6, 7]`
- Casts array-likes or iterables into an array, e.g `[...document.querySelectorAll('img')]`
- Useful when destructuring too, `[a, , ...rest] = [1, 2, 3, 4, 5]` yields `a: 1` and `rest: [3, 4, 5]`
- Makes `new` + `.apply` effortless, `new Date(...[2015, 31, 8])`
- Read ES6 Spread and Butter in Depth

(back to table of contents)

Arrow Functions

- Terse way to declare a function like `param => returnValue`
- Useful when doing functional stuff like `[1, 2].map(x => x * 2)`
- Several flavors are available, might take you some getting used to
- `p1 => expr` is okay for a single parameter
- `p1 => expr` has an implicit `return` statement for the provided `expr` expression
- To return an object implicitly, wrap it in parenthesis `() => ({ foo: 'bar' })` or you'll get **an error**
- Parenthesis are demanded when you have zero, two, or more parameters, `() => expr` or `(p1, p2) => expr`
- Brackets in the right-hand side represent a code block that can have multiple statements, `() => {}`
- When using a code block, there's no implicit `return`, you'll have to provide it – `() => { return 'foo' }`
- You can't name arrow functions statically, but runtimes are now much better at inferring names for most methods
- Arrow functions are bound to their lexical scope
- `this` is the same `this` context as in the parent scope
- `this` can't be modified with `.call`, `.apply`, or similar *"reflection"-type* methods
- Read ES6 Arrow Functions in Depth

(back to table of contents)

Template Literals

- You can declare strings with `` ` `` (backticks), in addition to `"` and `'`
- Strings wrapped in backticks are *template literals*
- Template literals can be multiline
- Template literals allow interpolation like `` `ponyfoo.com is ${rating}` `` where `rating` is a variable
- You can use any valid JavaScript expressions in the interpolation, such as `` `${2 * 3}` `` or `` `${foo()}` ``
- You can use tagged templates to change how expressions are interpolated
- Add a `fn` prefix to `` fn`foo, ${bar} and ${baz}` ``
- `fn` is called once with `template, ...expressions`
- `template` is `['foo, ', ' and ', '']` and `expressions` is `[bar, baz]`
- The result of `fn` becomes the value of the template literal
- Possible use cases include input sanitization of expressions, parameter parsing, etc.
- Template literals are almost strictly better than strings wrapped in single or double quotes
- Read ES6 Template Literals in Depth

(back to table of contents)

Object Literals

- Instead of `{ foo: foo }`, you can just do `{ foo }` – known as a *property value shorthand*
- Computed property names, `{` [redacted] `: 'bar' }`, where `prefix: 'moz'`, yields `{ mozFoo: 'bar' }`
- You can't combine computed property names and property value shorthands, `{ [foo] }` is invalid
- Method definitions in an object literal can be declared using an alternative, more terse syntax, `{ foo () {} }`
- See also `Object` section
- Read ES6 Object Literal Features in Depth

(back to table of contents)

Classes

- Not *"traditional"* classes, syntax sugar on top of prototypal inheritance
- Syntax similar to declaring objects, `class Foo {}`
- Instance methods – *`new Foo().bar`* – are declared using the short object literal syntax, `class Foo { bar () {} }`
- Static methods – *`Foo.isPonyFoo()`* – need a `static` keyword prefix, `class Foo { [redacted] isPonyFoo () {} }`
- Constructor method `class Foo { constructor () { /* initialize instance */ } }`
- Prototypal inheritance with a simple syntax `class PonyFoo [redacted] Foo {}`
- Read ES6 Classes in Depth

(back to table of contents)

Let and Const

- `let` and `const` are alternatives to `var` when declaring variables
- `let` is block-scoped instead of lexically scoped to a `function`
- `let` is hoisted to the top of the block, while `var` declarations are hoisted to top of the function
- "Temporal Dead Zone" – TDZ for short
- Starts at the beginning of the block where `let foo` was declared
- Ends where the `let foo` statement was placed in user code *(hoisiting is irrelevant here)*
- Attempts to access or assign to `foo` within the TDZ *(before the `let foo` statement is reached)* result in an error
- Helps prevent mysterious bugs when a variable is manipulated before its declaration is reached
- `const` is also block-scoped, hoisted, and constrained by TDZ semantics
- `const` variables must be declared using an initializer, `const foo = 'bar'`
- Assigning to `const` after initialization fails silently (or **loudly** – *with an exception* – under strict mode)
- `const` variables don't make the assigned value immutable
- `const foo = { bar: 'baz' }` means `foo` will always reference the right-hand side object
- `const foo = { bar: 'baz' }; foo.bar = 'boo'` won't throw
- Declaration of a variable by the same name will throw
- Meant to fix mistakes where you reassign a variable and lose a reference that was passed along somewhere else
- In ES6, **functions are block scoped**
- Prevents leaking block-scoped secrets through hoisting, `{ [redacted] _foo = 'secret', bar = [redacted]; }`
- Doesn't break user code in most situations, and typically what you wanted anyways
- Read ES6 Let, Const and the "Temporal Dead Zone" (TDZ) in Depth

(back to table of contents)

Symbols

- A new primitive type in ES6
- You can create your own symbols using `var symbol = Symbol()`
- You can add a description for debugging purposes, like `Symbol([redacted])`
- Symbols are immutable and unique. `Symbol()`, `Symbol()`, `Symbol('foo')` and `Symbol('foo')` are all different
- Symbols are of type `symbol`, thus: `typeof Symbol() === 'symbol'`
- You can also create global symbols with `Symbol.for(key)`
- If a symbol with the provided `key` already existed, you get that one back
- Otherwise, a new symbol is created, using `key` as its description as well
- `Symbol.keyFor(symbol)` is the inverse function, taking a `symbol` and returning its `key`
- Global symbols are **as global as it gets**, or *cross-realm*. Single registry used to look up these symbols across the runtime
- `window` context
- `eval` context
- `<iframe>` context, `Symbol.for('foo') === iframe.contentWindow.Symbol.for('foo')`
- There's also "well-known" symbols
- Not on the global registry, accessible through `Symbol[name]`, e.g: `Symbol.iterator`
- Cross-realm, meaning `Symbol.iterator === iframe.contentWindow.Symbol.iterator`
- Used by specification to define protocols, such as the *iterable* protocol over `Symbol.iterator`
- They're not **actually well-known** – in colloquial terms
- Iterating over symbol properties is hard, but not impossible and definitely not private
- Symbols are hidden to all pre-ES6 "reflection" methods
- Symbols are accessible through `Object.getOwnPropertySymbols`
- You won't stumble upon them but you **will** find them if *actively looking*
- Read ES6 Symbols in Depth

(back to table of contents)

Iterators

- Iterator and iterable protocol define how to iterate over any object, not just arrays and array-likes
- A well-known `Symbol` is used to assign an iterator to any object
- `var foo = { [`[redacted]`]: iterable}` , or `foo[`[redacted]`] = iterable`
- The `iterable` is a method that returns an `iterator` object that has a `next` method
- The `next` method returns objects with two properties, `value` and `done`
- The `value` property indicates the current value in the sequence being iterated
- The `done` property indicates whether there are any more items to iterate
- Objects that have a `[Symbol.iterator]` value are *iterable*, because they subscribe to the iterable protocol
- Some built-ins like `Array` , `String` , or `arguments` — and `NodeList` in browsers — are iterable by default in ES6
- Iterable objects can be looped over with `for..of` , such as `for (let el of document.querySelectorAll('a'))`
- Iterable objects can be synthesized using the spread operator, like `[...document.querySelectorAll('a')]`
- You can also use `Array.from(document.querySelectorAll('a'))` to synthesize an iterable sequence into an array
- Iterators are *lazy*, and those that produce an infinite sequence still can lead to valid programs
- Be careful not to attempt to synthesize an infinite sequence with `...` or `Array.from` as that **will** cause an infinite loop
- Read ES6 Iterators in Depth

(back to table of contents)

Generators

- Generator functions are a special kind of *iterator* that can be declared using the [redacted]`generator () {}` syntax
- Generator functions use `yield` to emit an element sequence
- Generator functions can also use `yield*` to delegate to another generator function — *or any iterable object*
- Generator functions return a generator object that's adheres to both the *iterable* and *iterator* protocols
- Given `g = generator()` , `g` adheres to the iterable protocol because `g[Symbol.iterator]` is a method
- Given `g = generator()` , `g` adheres to the iterator protocol because `g.next` is a method
- The iterator for a generator object `g` is the generator itself: `g[Symbol.iterator]() === g`
- Pull values using `Array.from(g)` , `[...g]` , `for (let item of g)` , or just calling `g.next()`
- Generator function execution is suspended, remembering the last position, in four different cases
- A `yield` expression returning the next value in the sequence
- A `return` statement returning the last value in the sequence
- A `throw` statement halts execution in the generator entirely
- Reaching the end of the generator function signals `{ done: true }`
- Once the `g` sequence has ended, `g.next()` simply returns `{ done: true }` and has no effect
- It's easy to make asynchronous flows feel synchronous
- Take user-provided generator function
- User code is suspended while asynchronous operations take place
- Call `g.next()` , unsuspending execution in user code
- Read ES6 Generators in Depth

(back to table of contents)

Promises

- Follows the `Promises/A+` specification, was widely implemented in the wild before ES6 was standarized *(e.g* `bluebird` *)*
- Promises behave like a tree. Add branches with `p.then(handler)` and `p.catch(handler)`
- Create new `p` promises with `new Promise(`[redacted]`)`
- The `resolve(value)` callback will fulfill the promise with the provided `value`
- The `reject(reason)` callback will reject `p` with a `reason` error
- You can call those methods asynchronously, blocking deeper branches of the promise tree
- Each call to `p.then` and `p.catch` creates another promise that's blocked on `p` being settled
- Promises start out in *pending* state and are **settled** when they're either *fulfilled* or *rejected*
- Promises can only be settled once, and then they're settled. Settled promises unblock deeper branches
- You can tack as many promises as you want onto as many branches as you need
- Each branch will execute either `.then` handlers or `.catch` handlers, never both
- A `.then` callback can transform the result of the previous branch by returning a value
- A `.then` callback can block on another promise by returning it
- `p.catch(fn).catch(fn)` won't do what you want — unless what you wanted is to catch errors in the error handler
- `Promise.resolve(value)` creates a promise that's fulfilled with the provided `value`
- `Promise.reject(reason)` creates a promise that's rejected with the provided `reason`
- `Promise.all(...promises)` creates a promise that settles when all `...promises` are fulfilled or 1 of them is rejected

- `Promise.race(...promises)` creates a promise that settles as soon as 1 of `...promises` is settled
- Use Promisees — the promise visualization playground — to better understand promises
- Read ES6 Promises in Depth

(back to table of contents)

Maps

- A replacement to the common pattern of creating a hash-map using plain JavaScript objects
- Avoids security issues with user-provided keys
- Allows keys to be arbitrary values, you can even use DOM elements or functions as the `key` to an entry
- `Map` adheres to *iterable* protocol
- Create a `map` using `new Map()`
- Initialize a map with an `iterable` like `[[key1, value1], [key2, value2]]` in `new Map(iterable)`
- Use `map.set(key, value)` to add entries
- Use `map.get(key)` to get an entry
- Check for a `key` using `map.has(key)`
- Remove entries with `map.delete(key)`
- Iterate over `map` with `for (let [key, value] of map)`, the spread operator, `Array.from`, etc
- Read ES6 Maps in Depth

(back to table of contents)

WeakMaps

- Similar to `Map`, but not quite the same
- `WeakMap` isn't iterable, so you don't get enumeration methods like `.forEach`, `.clear`, and others you had in `Map`
- `WeakMap` keys must be reference types. You can't use value types like symbols, numbers, or strings as keys
- `WeakMap` entries with a `key` that's the only reference to the referenced variable are subject to garbage collection
- That last point means `WeakMap` is great at keeping around metadata for objects, while those objects are still in use
- You avoid memory leaks, without manual reference counting — think of `WeakMap` as `IDisposable` in .NET
- Read ES6 WeakMaps in Depth

(back to table of contents)

Sets

- Similar to `Map`, but not quite the same
- `Set` doesn't have keys, there's only values
- `set.set(value)` doesn't look right, so we have `set.add(value)` instead
- Sets can't have duplicate values because the values are also used as keys
- Read ES6 Sets in Depth

(back to table of contents)

WeakSets

- `WeakSet` is sort of a cross-breed between `Set` and `WeakMap`
- A `WeakSet` is a set that can't be iterated and doesn't have enumeration methods
- `WeakSet` values must be reference types
- `WeakSet` may be useful for a metadata table indicating whether a reference is actively in use or not
- Read ES6 WeakSets in Depth

(back to table of contents)

Proxies

- Proxies are created with `new Proxy(target, handler)`, where `target` is any object and `handler` is configuration
- The default behavior of a `proxy` acts as a passthrough to the underlying `target` object
- Handlers determine how the underlying `target` object is accessed on top of regular object property access semantics
- You pass off references to `proxy` and retain strict control over how `target` can be interacted with
- Handlers are also known as traps, these terms are used interchangeably
- You can create **revocable** proxies with `Proxy.revocable(target, handler)`
- That method returns an object with `proxy` and `revoke` properties

- You could destructure `var [redacted] = Proxy.revocable(target, handler)` for convenience
- You can configure the `proxy` all the same as with `new Proxy(target, handler)`
- After `revoke()` is called, the `proxy` will **throw** on *any operation*, making it convenient when you can't trust consumers
- `get` – traps `proxy.prop` and `proxy['prop']`
- `set` – traps `proxy.prop = value` and `proxy['prop'] = value`
- `has` – traps `in` operator
- `deleteProperty` – traps `delete` operator
- `defineProperty` – traps `Object.defineProperty` and declarative alternatives
- `enumerate` – traps `for..in` loops
- `ownKeys` – traps `Object.keys` and related methods
- `apply` – traps *function calls*
- `construct` – traps usage of the `new` operator
- `getPrototypeOf` – traps internal calls to `[[GetPrototypeOf]]`
- `setPrototypeOf` – traps calls to `Object.setPrototypeOf`
- `isExtensible` – traps calls to `Object.isExtensible`
- `preventExtensions` – traps calls to `Object.preventExtensions`
- `getOwnPropertyDescriptor` – traps calls to `Object.getOwnPropertyDescriptor`
- Read ES6 Proxies in Depth
- Read ES6 Proxy Traps in Depth
- Read More ES6 Proxy Traps in Depth

(back to table of contents)

Reflection

- `Reflection` is a new static built-in (think of `Math`) in ES6
- `Reflection` methods have sensible internals, e.g `Reflect.defineProperty` returns a boolean instead of throwing
- There's a `Reflection` method for each proxy trap handler, and they represent the default behavior of each trap
- Going forward, new reflection methods in the same vein as `Object.keys` will be placed in the `Reflection` namespace
- Read ES6 Reflection in Depth

(back to table of contents)

`Number`

- Use `0b` prefix for binary, and `0o` prefix for octal integer literals
- `Number.isNaN` and `Number.isFinite` are like their global namesakes, except that they *don't* coerce input to `Number`
- `Number.parseInt` and `Number.parseFloat` are exactly the same as their global namesakes
- `Number.isInteger` checks if input is a `Number` value that doesn't have a decimal part
- `Number.EPSILON` helps figure out negligible differences between two numbers – e.g. `0.1 + 0.2` and `0.3`
- `Number.MAX_SAFE_INTEGER` is the largest integer that can be safely and precisely represented in JavaScript
- `Number.MIN_SAFE_INTEGER` is the smallest integer that can be safely and precisely represented in JavaScript
- `Number.isSafeInteger` checks whether an integer is within those bounds, able to be represented safely and precisely
- Read ES6 `Number` Improvements in Depth

(back to table of contents)

`Math`

- `Math.sign` – sign function of a number
- `Math.trunc` – integer part of a number
- `Math.cbrt` – cubic root of value, or $\sqrt[3]{value}$
- `Math.expm1` – `e` to the `value` minus `1`, or $e^{value} - 1$
- `Math.log1p` – natural logarithm of `value + 1`, or $ln(value + 1)$
- `Math.log10` – base 10 logarithm of `value`, or $log_{10}(value)$
- `Math.log2` – base 2 logarithm of `value`, or $log_2(value)$
- `Math.sinh` – hyperbolic sine of a number
- `Math.cosh` – hyperbolic cosine of a number
- `Math.tanh` – hyperbolic tangent of a number
- `Math.asinh` – hyperbolic arc-sine of a number
- `Math.acosh` – hyperbolic arc-cosine of a number
- `Math.atanh` – hyperbolic arc-tangent of a number
- `Math.hypot` – square root of the sum of squares

- `Math.clz32` – leading zero bits in the 32-bit representation of a number
- `Math.imul` – *C-like* 32-bit multiplication
- `Math.fround` – nearest single-precision float representation of a number
- Read ES6 `Math` Additions in Depth

(back to table of contents)

`Array`

- `Array.from` – create `Array` instances from arraylike objects like `arguments` or iterables
- `Array.of` – similar to `new Array(...items)`, but without special cases
- `Array.prototype.copyWithin` – copies a sequence of array elements into somewhere else in the array
- `Array.prototype.fill` – fills all elements of an existing array with the provided value
- `Array.prototype.find` – returns the first item to satisfy a callback
- `Array.prototype.findIndex` – returns the index of the first item to satisfy a callback
- `Array.prototype.keys` – returns an iterator that yields a sequence holding the keys for the array
- `Array.prototype.values` – returns an iterator that yields a sequence holding the values for the array
- `Array.prototype.entries` – returns an iterator that yields a sequence holding key value pairs for the array
- `Array.prototype[Symbol.iterator]` – exactly the same as the `Array.prototype.values` method
- Read ES6 `Array` Extensions in Depth

(back to table of contents)

`Object`

- `Object.assign` – recursive shallow overwrite for properties from `target, ...objects`
- `Object.is` – like using the `===` operator programmatically, but also `true` for `NaN` vs `NaN` and `+0` vs `-0`
- `Object.getOwnPropertySymbols` – returns all own property symbols found on an object
- `Object.setPrototypeOf` – changes prototype. Equivalent to `target.__proto__` setter
- See also Object Literals section
- Read ES6 `Object` Changes in Depth

(back to table of contents)

Strings and Unicode

- String Manipulation
- `String.prototype.startsWith` – whether the string starts with `value`
- `String.prototype.endsWith` – whether the string ends in `value`
- `String.prototype.includes` – whether the string contains `value` anywhere
- `String.prototype.repeat` – returns the string repeated `amount` times
- `String.prototype[Symbol.iterator]` – lets you iterate over a sequence of unicode code points *(not characters)*
- Unicode
- `String.prototype.codePointAt` – base-10 numeric representation of a code point at a given position in string
- `String.fromCodePoint` – given `...codepoints`, returns a string made of their unicode representations
- `String.prototype.normalize` – returns a normalized version of the string's unicode representation
- Read ES6 Strings and Unicode Additions in Depth

(back to table of contents)

Modules

- Strict Mode is turned on by default in the ES6 module system
- ES6 modules are files that `export` an API
- `export default value` exports a default binding
- `export var foo = 'bar'` exports a named binding
- Named exports are bindings that can be changed at any time from the module that's exporting them
- `export { foo, bar }` exports a list of named exports
- `export { foo` [redacted] `}` aliases the export to be referenced as `ponyfoo` instead
- `export { foo` [redacted] `}` marks the named export as the default export
- As a best practice, `export default api` at the end of all your modules, where `api` is an object, avoids confusion
- Module loading is implementation-specific, allows interoperation with CommonJS
- `import 'foo'` loads the `foo` module into the current module

[redacted]

- `import [redacted] 'ponyfoo'` assigns the default export of `ponyfoo` to a local `foo` variable
- `import {foo, bar} from 'baz'` imports named exports `foo` and `bar` from the `baz` module
- `import {foo [redacted]} from 'baz'` imports named export `foo` but aliased as a `bar` variable
- `import {default} from 'foo'` also imports the default export
- `import {default [redacted]} from 'foo'` imports the default export aliased as `bar`
- `import foo, {bar, baz} from 'foo'` mixes default `foo` with named exports `bar` and `baz` in one declaration
- `import * as foo from 'foo'` imports the namespace object
- Contains all named exports in `foo[name]`
- Contains the default export in `foo.default`, if a default export was declared in the module
- Read ES6 Modules Additions in Depth

(back to table of contents)

Time for a bullet point detox. Then again, I *did warn you* to read the article series instead. Also, did you try the Konami code just yet?

🠉 **Back to top**

About the Author

Nicolas Bevacqua

JavaScript and Web Performance Consultant

I the web. I am a consultant, a conference speaker, the author of JavaScript Application Design, an opinionated blogger, and an open-source evangelist. I participate actively in the online JavaScript community — as well as *offline* in beautiful Buenos Aires.

I like writing about the current state of the web, new features coming our way in ES6, leveraging web performance optimization to make our sites much faster, the importance of progressive enhancement, sane build processes and improving quality in your applications with modular design. I used to spend a lot of my time answering questions on Stack Overflow, but now I spend most that time doing open-source work instead.

I really enjoy developing small open-source modules that I publish to `npm` and GitHub. Some of these are small utilities that work well in both Node.js and the browser, and some others are front-end components that make it easier to use certain parts of the web. My favorite approach to open-source is developing small modules because that way you can *compose* them in interesting ways and it also fosters *reusability*. Learning how to write modular code is one of the most valuable things you can do to improve your skills as a JavaScript developer.

I've used a variety of tools when it comes to development. Trying out many different tools, *creating some of your own*, and experimenting are the best ways to really understand how they work and the tradeoffs between all the different tools and frameworks out there. If it's up to me, I like simple solutions. That's why I prefer to use `npm run` and Bash in my builds. I also like React and Taunus when it comes to view rendering, because they're **simpler and more performant** than anything else in the JavaScript framework landscape. I use AWS for deployments because I like having fine-grained control, but I've also experimented with other providers like Heroku and Digital Ocean.

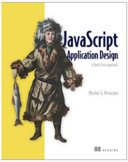
Re-inventing the wheel is **a necessary evil** if we want to learn from mistakes made in the past *(regardless of who made them)*.

Disclaimer: it might be possible that I don't look this good anymore.

On the Pony Foo home page, this note appears:

You can support my writing and open-source work through Patreon or PayPal.

Looking for a consultant? Have a conference talk opportunity? *Contact me!*



- @nzgb on Twitter
- @bevacqua on GitHub
- @bevacqua on Patreon
- @bevacqua on Lanyrd
- bevacqua.io

Pony Foo is on Twitter and Facebook.

NOTE: *You can comment on any of these articles at Pony Foo, and that's where any factual errors should be noted.*

This PDF was created (and styled) by Roger Sperberg. Any typographical or grammar errors or any issues with the PDF itself can be posted on the issues page of the ES6-info repository on GitHub.

The latest version of this PDF can be downloaded from the ES6-info repository.

NOTE: *This PDF was last revised on 2015-11-12*

Back to top