You Don't Know JS: ES6 & Beyond

Chapter 1: ES? Now & Future

Before you dive into this book, you should have a solid working proficiency over JavaScript up to the most recent standard (at the time of this writing), which is commonly called *ES5* (technically ES 5.1). Here, we plan to talk squarely about the upcoming *ES6*, as well as cast our vision beyond to understand how JS will evolve moving forward.

If you are still looking for confidence with JavaScript, I highly recommend you read the other titles in this series first:

- *Up & Going*: Are you new to programming and JS? This is the roadmap you need to consult as you start your learning journey.
- *Scope & Closures*: Did you know that JS lexical scope is based on compiler (not interpreter!) semantics? Can you explain how closures are a direct result of lexical scope and functions as values?
- *this & Object Prototypes*: Can you recite the four simple rules for how `this` is bound? Have you been muddling through fake "classes" in JS instead of adopting the simpler "behavior delegation" design pattern? Ever heard of *objects linked to other objects* (OLOO)?
- *Types & Grammar*: Do you know the built-in types in JS, and more importantly, do you know how to properly and safely use coercion between types? How comfortable are you with the nuances of JS grammar/syntax?
- *Async & Performance*: Are you still using callbacks to manage your asynchrony? Can you explain what a promise is and why/how it solves "callback hell"? Do you know how to use generators to improve the legibility of async code? What exactly constitutes mature optimization of JS programs and individual operations?

If you've already read all those titles and you feel pretty comfortable with the topics they cover, it's time we dive into the evolution of JS to explore all the changes coming not only soon but farther over the horizon.

Unlike ES5, ES6 is not just a modest set of new APIs added to the language. It incorporates a whole slew of new syntactic forms, some of which may take quite a bit of getting used to. There's also a variety of new organization forms and new API helpers for various data types.

ES6 is a radical jump forward for the language. Even if you think you know JS in ES5, ES6 is full of new stuff you *don't know yet*, so get ready! This book explores all the major themes of ES6 that you need to get up to speed on, and even gives you a glimpse of future features coming down the track that you should be aware of.

Warning: All code in this book assumes an ES6+ environment. At the time of this writing, ES6 support varies quite a bit in browsers and JS environments (like Node.js), so your mileage may vary.

Versioning

The JavaScript standard is referred to officially as "ECMAScript" (abbreviated "ES"), and up until just recently has been versioned entirely by ordinal number (i.e., "5" for "5th edition").

The earliest versions, ES1 and ES2, were not widely known or implemented. ES3 was the first widespread baseline for JavaScript, and constitutes the JavaScript standard for browsers like IE6-8 and older Android 2.x mobile browsers. For political reasons beyond what we'll cover here, the ill-fated ES4 never came about.

In 2009, ES5 was officially finalized (later ES5.1 in 2011), and settled as the widespread standard for JS for the modern revolution and explosion of browsers, such as Firefox, Chrome, Opera, Safari, and many others.

Leading up to the expected *next* version of JS (slipped from 2013 to 2014 and then 2015), the obvious and common label in discourse has been ES6.

However, late into the ES6 specification timeline, suggestions have surfaced that versioning may in the future switch to a year-based schema, such as ES2016 (aka ES7) to refer to whatever version of the specification is finalized before the end of

2016. Some disagree, but ES6 will likely maintain its dominant mindshare over the late-change substitute ES2015. However, ES2016 may in fact signal the new year-based schema.

It has also been observed that the pace of JS evolution is much faster even than single-year versioning. As soon as an idea begins to progress through standards discussions, browsers start prototyping the feature, and early adopters start experimenting with the code.

Usually well before there's an official stamp of approval, a feature is de facto standardized by virtue of this early engine/tooling prototyping. So it's also valid to consider the future of JS versioning to be per-feature rather than per-arbitrary-collection-of-major-features (as it is now) or even per-year (as it may become).

The takeaway is that the version labels stop being as important, and JavaScript starts to be seen more as an evergreen, living standard. The best way to cope with this is to stop thinking about your code base as being "ES6-based," for instance, and instead consider it feature by feature for support.

Transpiling

Made even worse by the rapid evolution of features, a problem arises for JS developers who at once may both strongly desire to use new features while at the same time being slapped with the reality that their sites/apps may need to support older browsers without such support.

The way ES5 appears to have played out in the broader industry, the typical mindset was that code bases waited to adopt ES5 until most if not all pre-ES5 environments had fallen out of their support spectrum. As a result, many are just recently (at the time of this writing) starting to adopt things like `strict` mode, which landed in ES5 over five years ago.

It's widely considered to be a harmful approach for the future of the JS ecosystem to wait around and trail the specification by so many years. All those responsible for evolving the language desire for developers to begin basing their code on the new features and patterns as soon as they stabilize in specification form and browsers have a chance to implement them.

So how do we resolve this seeming contradiction? The answer is tooling, specifically a technique called *transpiling* (transformation + compiling). Roughly, the idea is to use a special tool to transform your ES6 code into equivalent (or close!) matches that work in ES5 environments.

For example, consider shorthand property definitions (see "Object Literal Extensions" in Chapter 2). Here's the ES6 form:

```
var foo = [1,2,3];

var obj = {
        foo             // means `foo: foo`
};

obj.foo;        // [1,2,3]
```

But (roughly) here's how that transpiles:

```
var foo = [1,2,3];

var obj = {
        foo: foo
};

obj.foo;        // [1,2,3]
```

This is a minor but pleasant transformation that lets us shorten the `foo: foo` in an object literal declaration to just `foo`, if the names are the same.

Transpilers perform these transformations for you, usually in a build workflow step similar to how you perform linting, minification, and other similar operations.

Shims/Polyfills

Not all new ES6 features need a transpiler. Polyfills (aka shims) are a pattern for defining equivalent behavior from a newer environment into an older environment, when possible. Syntax cannot be polyfilled, but APIs often can be.

For example, `Object.is(..)` is a new utility for checking strict equality of two values but without the nuanced exceptions that `===` has for `NaN` and `-0` values. The polyfill for `Object.is(..)` is pretty easy:

```
if (!Object.is) {
        Object.is = function(v1, v2) {
                // test for `-0`
                if (v1 === 0 && v2 === 0) {
                        return 1 / v1 === 1 / v2;
                }
                // test for `NaN`
                if (v1 !== v1) {
                        return v2 !== v2;
                }
                // everything else
                return v1 === v2;
        };
}
```

Tip: Pay attention to the outer `if` statement guard wrapped around the polyfill. This is an important detail, which means the snippet only defines its fallback behavior for older environments where the API in question isn't already defined; it would be very rare that you'd want to overwrite an existing API.

There's a great collection of ES6 shims called "ES6 Shim" (https://github.com/paulmillr/es6-shim/) that you should definitely adopt as a standard part of any new JS project!

It is assumed that JS will continue to evolve constantly, with browsers rolling out support for features continually rather than in large chunks. So the best strategy for keeping updated as it evolves is to just introduce polyfill shims into your code base, and a transpiler step into your build workflow, right now and get used to that new reality.

If you decide to keep the status quo and just wait around for all browsers without a feature supported to go away before you start using the feature, you're always going to be way behind. You'll sadly be missing out on all the innovations designed to make writing JavaScript more effective, efficient, and robust.

Review

ES6 (some may try to call it ES2015) is just landing as of the time of this writing, and it has lots of new stuff you need to learn!

But it's even more important to shift your mindset to align with the new way that JavaScript is going to evolve. It's not just waiting around for years for some official document to get a vote of approval, as many have done in the past.

Now, JavaScript features land in browsers as they become ready, and it's up to you whether you'll get on the train early or whether you'll be playing costly catch-up games years from now.

Whatever labels that future JavaScript adopts, it's going to move a lot quicker than it ever has before. Transpilers and shims/polyfills are important tools to keep you on the forefront of where the language is headed.

If there's any narrative important to understand about the new reality for JavaScript, it's that all JS developers are strongly implored to move from the trailing edge of the curve to the leading edge. And learning ES6 is where that all starts!