You Don't Know JS: ES6 & Beyond

Chapter 8: Beyond ES6

At the time of this writing, the final draft of ES6 (*ECMAScript 2015*) is shortly headed toward its final official vote of approval by ECMA. But even as ES6 is being finalized, the TC39 committee is already hard at work on features for ES7/2016 and beyond.

As we discussed in Chapter 1, it's expected that the cadence of progress for JS is going to accelerate from updating once every several years to having an official version update once per year (hence the year-based naming). That alone is going to radically change how JS developers learn about and keep up with the language.

But even more importantly, the committee is actually going to work feature by feature. As soon as a feature is spec-complete and has its kinks worked out through implementation experiments in a few browsers, that feature will be considered stable enough to start using. We're all strongly encouraged to adopt features once they're ready instead of waiting for some official standards vote. If you haven't already learned ES6, the time is *past due* to get on board!

As the time of this writing, a list of future proposals and their status can be seen here (https://github.com/tc39/ecma262#current-proposals).

Transpilers and polyfills are how we'll bridge to these new features even before all browsers we support have implemented them. Babel, Traceur, and several other major transpilers already have support for some of the post-ES6 features that are most likely to stabilize.

With that in mind, it's already time for us to look at some of them. Let's jump in!

Warning: These features are all in various stages of development. While they're likely to land, and probably will look similar, take the contents of this chapter with more than a few grains of salt. This chapter will evolve in future editions of this title as these (and other!) features finalize.

`async function`s

In "Generators + Promises" in Chapter 4, we mentioned that there's a proposal for direct syntactic support for the pattern of generators `yield`ing promises to a runner-like utility

that will resume it on promise completion. Let's take a brief look at that proposed feature, called `async function`.
Recall this generator example from Chapter 4:

```
run( function *main() {
        var ret = yield step1();

        try {
                ret = yield step2( ret );
        }
        catch (err) {
                ret = yield step2Failed( err );
        }

        ret = yield Promise.all([
                step3a( ret ),
                step3b( ret ),
                step3c( ret )
        ]);

        yield step4( ret );
} )
.then(
        function fulfilled(){
                // `*main()` completed successfully
        },
        function rejected(reason){
                // Oops, something went wrong
        }
);
```

The proposed `async function` syntax can express this same flow control logic without needing the `run(..)` utility, because JS will automatically know how to look for promises to wait and resume. Consider:

```
async function main() {
        var ret = await step1();

        try {
                ret = await step2( ret );
        }
        catch (err) {
                ret = await step2Failed( err );
        }

        ret = await Promise.all( [
                step3a( ret ),
                step3b( ret ),
                step3c( ret )
        ] );

        await step4( ret );
}
```

```
main()
.then(
        function fulfilled(){
                // `main()` completed successfully
        },
        function rejected(reason){
                // Oops, something went wrong
        }
);
```

Instead of the `function *main() { ..` declaration, we declare with the `async function main() { ..` form. And instead of `yield`ing a promise, we `await` the promise. The call to run the function `main()` actually returns a promise that we can directly observe. That's the equivalent to the promise that we get back from a `run(main)` call.

Do you see the symmetry? `async function` is essentially syntactic sugar for the generators + promises + `run(..)` pattern; under the covers, it operates the same!

If you're a C# developer and this `async/await` looks familiar, it's because this feature is directly inspired by C#'s feature. It's nice to see language precedence informing convergence!

Babel, Traceur and other transpilers already have early support for the current status of `async function`s, so you can start using them already. However, in the next section "Caveats", we'll see why you perhaps shouldn't jump on that ship quite yet.

Note: There's also a proposal for `async function*`, which would be called an "async generator." You can both `yield` and `await` in the same code, and even combine those operations in the same statement: `x = await yield y`. The "async generator" proposal seems to be more in flux -- namely, its return value is not fully worked out yet. Some feel it should be an *observable*, which is kind of like the combination of an iterator and a promise. For now, we won't go further into that topic, but stay tuned as it evolves.

Caveats

One unresolved point of contention with `async function` is that because it only returns a promise, there's no way from the outside to *cancel* an `async function` instance that's currently running. This can be a problem if the async operation is resource intensive, and you want to free up the resources as soon as you're sure the result won't be needed.

For example:

```
async function request(url) {
        var resp = await (
                new Promise( function(resolve,reject){
                        var xhr = new XMLHttpRequest();
                        xhr.open( "GET", url );
                        xhr.onreadystatechange = function(){
                                if (xhr.readyState == 4) {
                                        if (xhr.status == 200) {
                                                resolve( xhr );
```

```
                                        }
                                        else {
                                                reject( xhr.statusText );
                                        }
                                }
                        };
                        xhr.send();
                } )
        );

        return resp.responseText;
}

var pr = request( "http://some.url.1" );

pr.then(
        function fulfilled(responseText){
                // ajax success
        },
        function rejected(reason){
                // Oops, something went wrong
        }
);
```

This `request(..)` that I've conceived is somewhat like the `fetch(..)` utility that's recently been proposed for inclusion into the web platform. So the concern is, what happens if you want to use the `pr` value to somehow indicate that you want to cancel a long-running Ajax request, for example?

Promises are not cancelable (at the time of writing, anyway). In my opinion, as well as many others, they never should be (see the *Async & Performance* title of this series). And even if a promise did have a `cancel()` method on it, does that necessarily mean that calling `pr.cancel()` should actually propagate a cancelation signal all the way back up the promise chain to the `async function`?

Several possible resolutions to this debate have surfaced:

- `async function`s won't be cancelable at all (status quo)
- A "cancel token" can be passed to an async function at call time
- Return value changes to a cancelable-promise type that's added
- Return value changes to something else non-promise (e.g., observable, or control token with promise and cancel capabilities)

At the time of this writing, `async function`s return regular promises, so it's less likely that the return value will entirely change. But it's too early to tell where things will land. Keep an eye on this discussion.

`Object.observe(..)`

One of the holy grails of front-end web development is data binding -- listening for updates to a data object and syncing the DOM representation of that data. Most JS frameworks provide some mechanism for these sorts of operations.

It appears likely that post ES6, we'll see support added directly to the language, via a utility called `Object.observe(..)`. Essentially, the idea is that you can set up a listener to observe an object's changes, and have a callback called any time a change occurs. You can then update the DOM accordingly, for instance.
There are six types of changes that you can observe:

- add
- update
- delete
- reconfigure
- setPrototype
- preventExtensions

By default, you'll be notified of all these change types, but you can filter down to only the ones you care about.

Consider:

```
var obj = { a: 1, b: 2 };

Object.observe(
        obj,
        function(changes){
                for (var change of changes) {
                        console.log( change );
                }
        },
        [ "add", "update", "delete" ]
);

obj.c = 3;
// { name: "c", object: obj, type: "add" }

obj.a = 42;
// { name: "a", object: obj, type: "update", oldValue: 1 }

delete obj.b;
// { name: "b", object: obj, type: "delete", oldValue: 2 }
```

In addition to the main `"add"`, `"update"`, and `"delete"` change types:

- The `"reconfigure"` change event is fired if one of the object's properties is reconfigured with `Object.defineProperty(..)`, such as changing

its `writable` attribute. See the *this & Object Prototypes* title of this series for more information.

- The `"preventExtensions"` change event is fired if the object is made non-extensible via `Object.preventExtensions(..)`.
  Because both `Object.seal(..)` and `Object.freeze(..)` also imply `Object.preventExtensions(..)`, they'll also fire its corresponding change event. In addition, `"reconfigure"` change events will also be fired for each property on the object.
- The `"setPrototype"` change event is fired if the `[[Prototype]]` of an object is changed, either by setting it with the `__proto__` setter, or using `Object.setPrototypeOf(..)`.

Notice that these change events are notified immediately after said change. Don't confuse this with proxies (see Chapter 7) where you can intercept the actions before they occur. Object observation lets you respond after a change (or set of changes) occurs.

Custom Change Events

In addition to the six built-in change event types, you can also listen for and fire custom change events.

Consider:

```
function observer(changes){
        for (var change of changes) {
                if (change.type == "recalc") {
                        change.object.c =
                                change.object.oldValue +
                                change.object.a +
                                change.object.b;
                }
        }
}

function changeObj(a,b) {
        var notifier = Object.getNotifier( obj );

        obj.a = a * 2;
        obj.b = b * 3;

        // queue up change events into a set
        notifier.notify( {
                type: "recalc",
                name: "c",
                oldValue: obj.c
        } );
}
```

```
var obj = { a: 1, b: 2, c: 3 };

Object.observe(
        obj,
        observer,
        ["recalc"]
);

changeObj( 3, 11 );

obj.a;                  // 12
obj.b;                  // 30
obj.c;                  // 3
```

The change set (`"recalc"` custom event) has been queued for delivery to the observer, but not delivered yet, which is why `obj.c` is still `3`.
The changes are by default delivered at the end of the current event loop (see the *Async & Performance* title of this series). If you want to deliver them immediately, use `Object.deliverChangeRecords(observer)`. Once the change events are delivered, you can observe `obj.c` updated as expected:

```
obj.c;                  // 42
```

In the previous example, we called `notifier.notify(..)` with the complete change event record. An alternative form for queuing change records is to use `performChange(..)`, which separates specifying the type of the event from the rest of event record's properties (via a function callback). Consider:

```
notifier.performChange( "recalc", function(){
        return {
                name: "c",
                // `this` is the object under observation
                oldValue: this.c
        };
} );
```

In certain circumstances, this separation of concerns may map more cleanly to your usage pattern.

Ending Observation

Just like with normal event listeners, you may wish to stop observing an object's change events. For that, you use `Object.unobserve(..)`.
For example:

```
var obj = { a: 1, b: 2 };

Object.observe( obj, function observer(changes) {
        for (var change of changes) {
                if (change.type == "setPrototype") {
```

```
                        Object.unobserve(
                                change.object, observer
                        );
                        break;
                }
        }
} );
```

In this trivial example, we listen for change events until we see the `"setPrototype"` event come through, at which time we stop observing any more change events.

Exponentiation Operator

An operator has been proposed for JavaScript to perform exponentiation in the same way that `Math.pow(..)` does. Consider:

```
var a = 2;

a ** 4;                 // Math.pow( a, 4 ) == 16

a **= 3;         // a = Math.pow( a, 3 )
a;                              // 8
```

Note: `**` is essentially the same as it appears in Python, Ruby, Perl, and others.

Objects Properties and `...`

As we saw in the "Too Many, Too Few, Just Enough" section of Chapter 2, the `...` operator is pretty obvious in how it relates to spreading or gathering arrays. But what about objects?

Such a feature was considered for ES6, but was deferred to be considered after ES6 (aka "ES7" or "ES2016" or ...). Here's how it might work in that "beyond ES6" timeframe:

```
var o1 = { a: 1, b: 2 },
        o2 = { c: 3 },
        o3 = { ...o1, ...o2, d: 4 };

console.log( o3.a, o3.b, o3.c, o3.d );
// 1 2 3 4
```

The `...` operator might also be used to gather an object's destructured properties back into an object:

```
var o1 = { b: 2, c: 3, d: 4 };
var { b, ...o2 } = o1;

console.log( b, o2.c, o2.d );             // 2 3 4
```

Here, the `...o2` re-gathers the destructured `c` and `d` properties back into an `o2` object (`o2` does not have a `b` property like `o1` does).

Again, these are just proposals under consideration beyond ES6. But it'll be cool if they do land.

`Array#includes(..)`

One extremely common task JS developers need to perform is searching for a value inside an array of values. The way this has always been done is:

```
var vals = [ "foo", "bar", 42, "baz" ];

if (vals.indexOf( 42 ) >= 0) {
        // found it!
}
```

The reason for the `>= 0` check is because `indexOf(..)` returns a numeric value of `0` or greater if found, or `-1` if not found. In other words, we're using an index-returning function in a boolean context. But because `-1` is truthy instead of falsy, we have to be more manual with our checks.

In the *Types & Grammar* title of this series, I explored another pattern that I slightly prefer:

```
var vals = [ "foo", "bar", 42, "baz" ];

if (~vals.indexOf( 42 )) {
        // found it!
}
```

The `~` operator here conforms the return value of `indexOf(..)` to a value range that is suitably boolean coercible. That is, `-1` produces `0` (falsy), and anything else produces a non-zero (truthy) value, which is what we for deciding if we found the value or not.

While I think that's an improvement, others strongly disagree. However, no one can argue that `indexOf(..)`'s searching logic is perfect. It fails to find `NaN` values in the array, for example.

So a proposal has surfaced and gained a lot of support for adding a real boolean-returning array search method, called `includes(..)`:

```
var vals = [ "foo", "bar", 42, "baz" ];

if (vals.includes( 42 )) {
        // found it!
}
```

Note: `Array#includes(..)` uses matching logic that will find `NaN` values, but will not distinguish between `-0` and `0` (see the *Types & Grammar* title of this series). If you don't care about `-0` values in your programs, this will likely be exactly what you're hoping for. If you *do* care about `-0`, you'll need to do your own searching logic, likely using the `Object.is(..)` utility (see Chapter 6).

SIMD

We cover Single Instruction, Multiple Data (SIMD) in more detail in the *Async & Performance* title of this series, but it bears a brief mention here, as it's one of the next likely features to land in a future JS.

The SIMD API exposes various low-level (CPU) instructions that can operate on more than a single number value at a time. For example, you'll be able to specify two *vectors* of 4 or 8 numbers each, and multiply the respective elements all at once (data parallelism!).

Consider:

```
var v1 = SIMD.float32x4( 3.14159, 21.0, 32.3, 55.55 );
var v2 = SIMD.float32x4( 2.1, 3.2, 4.3, 5.4 );

SIMD.float32x4.mul( v1, v2 );
// [ 6.597339, 67.2, 138.89, 299.97 ]
```

SIMD will include several other operations besides `mul(..)` (multiplication), such as `sub()`, `div()`, `abs()`, `neg()`, `sqrt()`, and many more.
Parallel math operations are critical for the next generations of high performance JS applications.

WebAssembly (WASM)

Brendan Eich made a late breaking announcement near the completion of the first edition of this title that has the potential to significantly impact the future path of JavaScript: WebAssembly (WASM). We will not be able to cover WASM in detail here, as it's extremely early at the time of this writing. But this title would be incomplete without at least a brief mention of it.

One of the strongest pressures on the recent (and near future) design changes of the JS language has been the desire that it become a more suitable target for transpilation/cross-compilation from other languages (like C/C++, ClojureScript, etc.). Obviously, performance of code running as JavaScript has been a primary concern.

As discussed in the *Async & Performance* title of this series, a few years ago a group of developers at Mozilla introduced an idea to JavaScript called ASM.js. ASM.js is a subset of valid JS that most significantly restricts certain actions that make code hard for the JS engine to optimize. The result is that ASM.js compatible code running in an ASM-aware engine can run remarkably faster, nearly on par with native optimized C equivalents. Many viewed ASM.js as the most likely backbone on which performance-hungry applications would ride in JavaScript.

In other words, all roads to running code in the browser *lead through JavaScript*.

That is, until the WASM announcement. WASM provides an alternate path for other languages to target the browser's runtime environment without having to first pass through JavaScript. Essentially, if WASM takes off, JS engines will grow an extra capability to execute a binary format of code that can be seen as somewhat similar to a bytecode (like that which runs on the JVM).

WASM proposes a format for a binary representation of a highly compressed AST (syntax tree) of code, which can then give instructions directly to the JS engine and its underpinnings, without having to be parsed by JS, or even behave by the rules of JS. Languages like C or C++ can be compiled directly to the WASM format instead of ASM.js, and gain an extra speed advantage by skipping the JS parsing.

The near term for WASM is to have parity with ASM.js and indeed JS. But eventually, it's expected that WASM would grow new capabilities that surpass anything JS could do. For example, the pressure for JS to evolve radical features like threads -- a change that would certainly send major shockwaves through the JS ecosystem -- has a more hopeful future as a future WASM extension, relieving the pressure to change JS.

In fact, this new roadmap opens up many new roads for many languages to target the web runtime. That's an exciting new future path for the web platform!

What does it mean for JS? Will JS become irrelevant or "die"? Absolutely not. ASM.js will likely not see much of a future beyond the next couple of years, but the majority of JS is quite safely anchored in the web platform story.

Proponents of WASM suggest its success will mean that the design of JS will be protected from pressures that would have eventually stretched it beyond assumed breaking points of reasonability. It is projected that WASM will become the preferred target for high-performance parts of applications, as authored in any of a myriad of different languages.

Interestingly, JavaScript is one of the lesser likely languages to target WASM in the future. There may be future changes that carve out subsets of JS that might be tenable for such targeting, but that path doesn't seem high on the priority list.

While JS likely won't be much of a WASM funnel, JS code and WASM code will be able to interoperate in the most significant ways, just as naturally as current module interactions. You can imagine calling a JS function like `foo()` and having that actually invoke a WASM function of that name with the power to run well outside the constraints of the rest of your JS.

Things which are currently written in JS will probably continue to always be written in JS, at least for the foreseeable future. Things which are transpiled to JS will probably eventually at least consider targeting WASM instead. For things which need the utmost in performance with minimal tolerance for layers of abstraction, the likely choice will be to find a suitable non-JS language to author in, then targeting WASM.

There's a good chance this shift will be slow, and will be years in the making. WASM landing in all the major browser platforms is probably a few years out at best. In the meantime, the WASM project (https://github.com/WebAssembly) has an early polyfill to demonstrate proof-of-concept for its basic tenets.

But as time goes on, and as WASM learns new non-JS tricks, it's not too much a stretch of imagination to see some currently-JS things being refactored to a WASM-targetable language. For example, the performance sensitive parts of frameworks, game engines, and other heavily used tools might very well benefit from such a shift. Developers using these tools in their web applications likely won't notice much difference in usage or integration, but will just automatically take advantage of the performance and capabilities.

What's certain is that the more real WASM becomes over time, the more it means to the trajectory and design of JavaScript. It's perhaps one of the most important "beyond ES6" topics developers should keep an eye on.

Review

If all the other books in this series essentially propose this challenge, "you (may) not know JS (as much as you thought)," this book has instead suggested, "you don't know JS anymore." The book has covered a ton of new stuff added to the language in ES6. It's an exciting collection of new language features and paradigms that will forever improve our JS programs.

But JS is not done with ES6! Not even close. There's already quite a few features in various stages of development for the "beyond ES6" timeframe. In this chapter, we briefly looked at some of the most likely candidates to land in JS very soon.

`async functions` are powerful syntactic sugar on top of the generators + promises pattern (see Chapter 4). `Object.observe(..)` adds direct native support for observing object change events, which is critical for implementing data binding.
The `**` exponentiation operator, `...` for object properties, and `Array#includes(..)` are all simple but helpful improvements to existing mechanisms. Finally, SIMD ushers in a new era in the evolution of high performance JS.

Cliché as it sounds, the future of JS is really bright! The challenge of this series, and indeed of this book, is incumbent on every reader now. What are you waiting for? It's time to get learning and exploring!