Background on ES6

From Exploring ES6 – Axel Rauschmayer – Chapters 1 through 3

'use strict';

If you put this line at the beginning of a file, all code in it is in strict mode. If you make this line the first line of a function, only that function is in strict mode. Using a directive to switch on strict mode is not very user friendly and was one of the reasons why strict mode was not nearly as popular in ES5 as it should be. However, ES6 modules and classes are implicitly in strict mode. Given that most ES6 code will live in modules, strict mode becomes the de-facto default for ES6.

TC39 (Ecma Technical Committee 39) is the committee that evolves JavaScript

<https://github.com/tc39/tc39-notes>

The ECMAScript 6 design process centers on proposals for features. Proposals are often triggered by suggestions from the developer community. To avoid design by committee, proposals are maintained by champions (1–2 committee delegates).

• Sketch (informally: “strawman proposal”): A first description of the proposed feature. • Proposal: If TC39 agrees that a feature is important, it gets promoted to official proposal status. That does not guarantee it will become a standard, but it considerably increases its chances. The deadline for ES6 proposals was May 2011. No major new proposals were considered after that. • Implementations: Proposed features must be implemented. Ideally in two JavaScript engines. Implementations and feedback from the community shape the proposal as it evolves. • Standard: If the proposal continues to prove itself and is accepted by TC39, it will eventually be included in an edition of the ECMAScript standard. At this point, it is a standard feature.

Starting with ECMAScript 2016 (ES7), TC39 will time-box releases. A new version of ECMAScript will be released every year, with whatever features are ready at that time.That means that from now on, ECMAScript versions will be relatively small upgrades.

ES6 is a superset of ES5, nothing is removed

Several goals of ES6

The goal is: Be a better language for writing:

1. complex applications;
2. libraries (possibly including the DOM) shared by those applications;
3. code generators targeting the new edition

ES6 modules were a response to breaking down complex applications and npm packages

The goal is: Improve Interoperation, adopting de facto standards where possible

Classes: are based on how constructor functions are currently used. • Modules: picked up design ideas from the CommonJS module format. • Arrow functions: have syntax that is borrowed from CoffeeScript. • Named function parameters: There is no built-in support for named parameters. Instead, the existing practice of naming parameters via object literals is supported via destructuring in parameter definitions.

Versioning

As mentioned previously, ES6 avoids versioning via “One JavaScript”: In an ES6 code base, everything is ES6, there are no parts that are ES5-specific.

ECMAScript 2015: In late 2014, TC39 decided to change the official name of ECMAScript 6 to ECMAScript 2015, in light of upcoming yearly spec releases. However, given how established the name “ECMAScript 6” already is and how late TC39 changed their minds, I expect that that’s how everybody will continue to refer to that version.

Does that mean that you shouldn’t learn ECMAScript 5, anymore? It doesn’t, for several reasons: • ECMAScript 6 is a superset of ECMAScript 5 – new JavaScript versions must never break existing code. Thus, nothing you learn about ECMAScript 5 is learned in vain. • There are several ECMAScript 6 features that kind of replace ECMAScript 5 features, but still use them as their foundations. It is important to understand those foundations. Two examples: classes are internally translated to constructors and methods are still functions (as they have always been). • As long as ECMAScript 6 is compiled to ECMAScript 5, it is useful to understand the output of the compilation process. And you’ll have to compile to ES5 for a while (probably years), until you can rely on ES6 being available in all relevant browsers. • It’s important to be able to understand legacy code.

A more fundamental issue is that allowing multiple versions per code base effectively forks a language into sub-languages that have to be maintained in parallel. This causes problems: • Engines become bloated, because they need to implement the semantics of all versions. The same applies to tools analyzing the language (e.g. style checkers such as JSLint). • Programmers need to remember how the versions differ. • Code becomes harder to refactor, because you need to take versions into consideration when you move pieces of code.

But how can we get rid of versioning? By always being backward-compatible. That means we must give up some of our ambitions w.r.t. cleaning up JavaScript: We can’t introduce breaking changes. Being backward-compatible means not removing features and not changing features. The slogan for this principle is: “don’t break the web”

The slogan for this principle is: “don’t break the web”. We can, however, add new features and make existing features more powerful

As a consequence, no versions are needed for new engines, because they can still run all old code. David Herman calls this approach to avoiding versioning One JavaScript (1JS) [1], because it avoids splitting up JavaScript into different versions or modes.

One JavaScript does not mean that you have to completely give up on cleaning up the language. Instead of cleaning up existing features, you introduce new, clean, features. One example for that is let, which declares block-scoped variables and is an improved version of var. It does not, however, replace var. It exists alongside it, as the superior option