ES6: The Road to the Future

Contents

[ES6 2](#_Toc450288395)

[Show Compatibility Table 2](#_Toc450288396)

[Complete Feature List 2](#_Toc450288397)

[Transpilers 2](#_Toc450288398)

[New Syntax 3](#_Toc450288399)

[Var Keyword 3](#_Toc450288400)

[Let Keyword 3](#_Toc450288401)

[Arrow Functions 4](#_Toc450288402)

[Destructuring 4](#_Toc450288403)

[Strings 4](#_Toc450288404)

[Module systems for current JavaScript 5](#_Toc450288405)

[ECMAScript 6 modules 5](#_Toc450288406)

[Named exports (several per module) 6](#_Toc450288407)

[Default exports (one per module) 6](#_Toc450288408)

[Design Goals 6](#_Toc450288409)

[ES6 Classes 7](#_Toc450288410)

[Class declarations 7](#_Toc450288411)

[Hoisting 7](#_Toc450288412)

[Class expressions 7](#_Toc450288413)

[Class body and method definitions 7](#_Toc450288414)

[Strict mode 7](#_Toc450288415)

[Abstract Classes 7](#_Toc450288416)

[Constructors 8](#_Toc450288417)

[Inheritance 8](#_Toc450288418)

[Sub classing with extends 8](#_Toc450288419)

[Super class calls with super 8](#_Toc450288420)

[Instance Properties & Methods 8](#_Toc450288421)

[Static Properties & Methods 8](#_Toc450288422)

[Static methods 8](#_Toc450288423)

[Conclusion 9](#_Toc450288424)

# ES6

Today we are going to discuss ES6.

Before I continue, I would like to ask how many people know what ES6 is.

How about ECMAScript? Anyone know what that is?

Simply put, ES6 or ECMAScript 6 is the sixth official version of JavaScript. It is the first update to the language since ES5, standardized in 2009. Implementation of the features in major JavaScript engines is underway now.

## Show Compatibility Table

<https://kangax.github.io/compat-table/es6/>

## Complete Feature List

<http://es6-features.org/#Constants>

# Transpilers

Anyone who's had to deal with browser compatibility issues before knows it’s not as simple as writing JavaScript that runs everywhere. That's because every browser uses a different JavaScript engine: Chrome runs V8, Firefox runs SpiderMonkey, and Internet Explorer, Chakra. Each has different performance characteristics, each implements a different subset of ES2015 features, and each is approaching full compliance with the spec at different rates.

Transpilers to the rescue!

A transcompiler or transpiler is a type of compiler that takes the source code of a program written in one programming language as its input and produces the equivalent source code in another programming language. Essentially, a transpiler is a source-to-source compiler.

If your workflow doesn't already include a transpiler, you might wonder why you'd even bother. Why learn new syntax and pick up new tools if all we get at the end of the day is the JavaScript we could have written in the first place?

In the case of languages that target JavaScript, it's largely a matter of preference or background. Writing in a language that "thinks" the way you do makes you more productive. People with backgrounds in OOP often like TypeScript because it's familiar territory. Pythonistas like CoffeeScript. Clojurists write ClojureScript. You get the idea.

Popular JavaScript transpires include Babel, Traceur, TypeScript, and many orthers.

# New Syntax

## Var Keyword

Let key word, works a bit differently than var

Before diving in to the characteristics or let, we need to briefly review a few other concepts

- Hoisting & Function Scope

- any variables set using var key word within a function are Hoisted to the top of the function.

- subsequent assignments using var simply change the value of the variable

- any var sets outside function are hoisted to the global scope (bad)

- this is why you see many people using closures?

# Let Keyword

- Variables are not hoisted

- Variables are BLOCK scoped

- should be familiar to .net devs

- provides more predictable behaviors.

# Arrow Functions

Arrow functions – also called “fat arrow” functions, from CoffeeScript (another transpiled language) are a more concise syntax for writing function expressions. They utilize a new token, =>, that looks like a fat arrow. Arrow functions are anonymous and change the way this binds in functions.

Arrow functions make our code more concise, and simplify function scoping. They are one-line mini functions which work much like Lambdas in other languages like C# or Python. By using arrow function we avoid having to type the function keyword, return keyword (it’s implicit in arrow functions), and curly brackets.

Another great thing about arrow functions is that they lexicaly bind to this. Those of you who do JavaScript may already be used to adding something like this to you functions:

var vm = this;

This is a work around the problem. Using arrow functions that is not really needed.

# Destructuring

-Break-up an object or array into component variables using a very turse syntax

- Sometime looks almost like magic

Destructuring assignment allows you to assign the properties of an array or object to variables using syntax that looks similar to array or object literals. This syntax can be extremely terse, while still exhibiting more clarity than the traditional property access.

# Strings

<https://developers.google.com/web/updates/2015/01/ES6-Template-Strings?hl=en>

Multiline strings in JavaScript have required hacky workarounds for some time. Current solutions for them require that strings either exist on a single line or be split into multiline strings using a \ (blackslash) before each newline.

## Module systems for current JavaScript

JavaScript does not have built-in support for modules, but the community has created impressive work-arounds. The two most important (and unfortunately incompatible) standards are:

* **CommonJS Modules:** The dominant implementation of this standard is [in Node.js](http://nodejs.org/api/modules.html) (Node.js modules have a few features that go beyond CommonJS). Characteristics:
  + Compact syntax
  + Designed for synchronous loading
  + Main use: server
* **Asynchronous Module Definition (AMD):** The most popular implementation of this standard is [RequireJS](http://requirejs.org/). Characteristics:
  + Slightly more complicated syntax, enabling AMD to work without eval() (or a compilation step).
  + Designed for asynchronous loading
  + Main use: browsers

# ECMAScript 6 modules

The goal for ECMAScript 6 modules was to create a format that both users of CommonJS and of AMD are happy with:

* Similar to CommonJS, they have a compact syntax, a preference for single exports and support for cyclic dependencies.
* Similar to AMD, they have direct support for asynchronous loading and configurable module loading.

Being built into the language allows ES6 modules to go beyond CommonJS and AMD (details are explained later):

* Their syntax is even more compact than CommonJS’s.
* Their structure can be statically analyzed (for static checking, optimization, etc.).
* Their support for cyclic dependencies is better than CommonJS’s.

The ES6 module standard has two parts:

* Declarative syntax (for importing and exporting)
* Programmatic loader API: to configure how modules are loaded and to conditionally load modules

- There are two kinds of exports: named exports (several per module) and default exports (one per module).

# Named exports (several per module)

A module can export multiple things by prefixing their declarations with the keyword export. These exports are distinguished by their names and are called *named exports*.

# Default exports (one per module)

Modules that only export single values are very popular in the Node.js community. But they are also common in front-end development where you often have constructors/classes for models, with one model per module. An ECMAScript 6 module can pick a *default export*, the most important exported value. Default exports are especially easy to import.

# Design Goals

Default exports are favored

The module syntax suggesting that the default export “is” the module may seem a bit strange, but it makes sense if you consider that one major design goal was to make default exports as convenient as possible. Quoting David Herman:

ECMAScript 6 favors the single/default export style, and gives the sweetest syntax to importing the default. Importing named exports can and even should be slightly less concise.

Static module structure

In current JavaScript module systems, you have to execute the code in order to find out what the imports and exports are. That is the main reason why ECMAScript 6 breaks with those systems: by building the module system into the language, you can syntactically enforce a static module structure. Let’s first examine what that means and then what benefits it brings.

A module’s structure being static means that you can determine imports and exports at compile time (statically) – you only have to look at the source code, you don’t have to execute it. The following are two examples of how CommonJS modules can make that impossible. In the first example, you have to run the code to find out what it imports:

Support for both synchronous and asynchronous loading

ECMAScript 6 modules must work independently of whether the engine loads modules synchronously (e.g. on servers) or asynchronously (e.g. in browsers). Its syntax is well suited for synchronous loading, asynchronous loading is enabled by its static structure: Because you can statically determine all imports, you can load them before evaluating the body of the module (in a manner reminiscent of AMD modules).

# ES6 Classes

## Class declarations

One way to define a class is using a **class declaration**. To declare a class, you use the class keyword with the name of the class ("Polygon" here).

## Hoisting

An important difference between **function declarations** and **class declarations** is that function declarations are hoisted and class declarations are not. You first need to declare your class and then access it, otherwise code like the following will throw a ReferenceError:

### Class expressions

A **class expression** is another way to define a class. Class expressions can be named or unnamed. The name given to a named class expression is local to the class's body.

### Class body and method definitions

The body of a class is the part that is in curly brackets {}. This is where you define class members, such as methods or constructors.

## Strict mode

The bodies of *class declarations* and *class expressions* are executed in strict mode.

## Abstract Classes

Abstract subclasses or *mix-ins* are templates for classes. An ECMAScript class can only have a single superclass, so multiple inheritance from tooling classes, for example, is not possible. The functionality must be provided by the superclass.

A function with a superclass as input and a subclass extending that superclass as output can be used to implement mix-ins in ECMAScript:

## Constructors

The constructor method is a special method for creating and initializing an object created with a class. There can only be one special method with the name "constructor" in a class. A SyntaxError will be thrown if the class contains more than one occurrence of a constructor method.

A constructor can use the super keyword to call the constructor of a parent class.

## Inheritance

## Sub classing with extends

The extends keyword is used in *class declarations* or *class expressions* to create a class as a child of another class.

## Super class calls with super

The super keyword is used to call functions on an object's parent.

## Instance Properties & Methods

## Static Properties & Methods

## Static methods

The static keyword defines a static method for a class. Static methods are called without instantiating their class and are also **not**callable when the class is instantiated. Static methods are often used to create utility functions for an application.

<https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Classes>

# Conclusion

Need to re-state that transpilation to es5 is only till most browsers support ES6 natively.

Links:

http://www.2ality.com/2014/09/es6-modules-final.html ES6 Modules

http://www.lennyreed.com

http://es6-features.org

https://kangax.github.io/compat-table/es6/

http://ilikekillnerds.com/2015/01/transpiling-wars-6to5-vs-traceur/