

# ES6 & The Future Of JS

Nenad Pečanac  
Serengeti

# Contents

- 1) History
- 2) Tooling
- 3) Syntax
- 4) Functions
- 5) Classes
- 6) Collections
- 7) Modules
- 8) Promises
- 9) The Future

# 1) History

# History (1)



JavaScript was created in 1995 by Brendan Eich for Netscape.  
It was developed as Mocha and released as LiveScript.

It was quickly renamed to JavaScript, as a marketing move due to Java's popularity.  
New name caused a significant misunderstanding of the language.

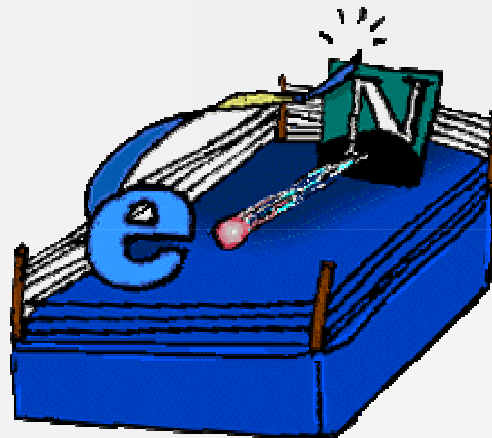


Language was created „in a week“ and released incomplete.  
Ambitious design influenced by Java, Perl, Scheme and Self.

## History (2)

Microsoft released JScript for IE 3.0 in 1996.

Browser Wars: CSS, DOM, Scripting, Dynamic HTML ...



Netscape delivered JavaScript to **Ecma International** for standardization in 1996.

**ECMAScript** was accepted as the compromised trade name.

# History (3)

Release	Year	Changes
1	1997	First edition
2	1998	Editorial changes to keep the specification fully aligned with ISO/IEC 16262 international standard
3	1999	Added regular expressions, better string handling, new control statements, try/catch exception handling, tighter definition of errors, formatting for numeric output and other enhancements
4	-	<b>Abandoned</b> due to political differences concerning language complexity.

# History (4)



Release	Year	Changes
5	2009	Adds <b>strict mode</b> , a subset intended to provide more thorough error checking, avoid error-prone constructs and clarifies many ambiguities. Adds some new features, such as getters and setters and library support for JSON.
5.1	2011	Aligned with third edition of the ISO/IEC 16262:2011.
6	2015	<b>ECMAScript Harmony</b> or ES6 Harmony.
7	...	Work in progress.

# Current browser support



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

The screenshot shows the ES6 compatibility table website. The header includes navigation links like 'ES6', 'next', 'non-standard', and 'compatibility table'. There are filters for 'Sort by: Engine types' and 'Show absolute platforms'. A legend at the top right identifies browser engines: V8, NodeMonkey, JavaScriptCore, Firefox, Chromium, Safari, and Other. Below the legend, a bar chart shows the percentage of features supported by each engine. The main table lists features under categories like 'Optimisation', 'Syntax', and 'Bindings'. Each feature has a 'Current browser' column and columns for various engines and versions, with cells containing support status (e.g., '0/2', '4/5', '5/5') and a color-coded background (green for full support, red for partial, yellow for not supported).

Feature name	Current browser	Compilers/polyfills									Desktop browsers											Servers/runtimes								
		Traceur	Babel core.js	Closure	TypeScript core.js	es6-shim	IE 11	Edge 12	Edge 13	FF53 ESR	FF41	FF43	CH 48, OP 36	CH 49, OP 38	CH 50, OP 37	SF 6.1, SF 7	SF 7.1, SF 8	SF 9	WK	KQ 4.14	PJS	Node 0.12	Node 4.0	Node 5.0	Echo JS	X56				
Optimisation																														
proper tail calls (full optimization)	0/2	0/2	0/2	0/2	0/2	0/2	0/2	0/2	0/2	0/2	0/2	0/2	0/2	0/2	0/2	0/2	0/2	0/2	0/2	0/2	0/2	0/2	0/2	0/2	0/2	0/2	0/2	2/2		
Syntax																														
default function parameters	0/2	0/2	0/2	0/2	0/2	0/2	0/2	0/2	0/2	0/2	0/2	0/2	0/2	0/2	0/2	0/2	0/2	0/2	0/2	0/2	0/2	0/2	0/2	0/2	0/2	0/2	0/2	0/2		
post-operators	5/5	4/5	3/5	2/5	3/5	0/5	0/5	5/5	5/5	4/5	5/5	5/5	5/5	5/5	5/5	5/5	5/5	5/5	5/5	5/5	5/5	5/5	5/5	5/5	5/5	5/5	5/5	5/5		
comparable supercall	15/15	15/15	13/15	12/15	0/15	0/15	0/15	12/15	15/15	15/15	15/15	15/15	15/15	15/15	15/15	15/15	15/15	15/15	15/15	15/15	15/15	15/15	15/15	15/15	15/15	15/15	15/15	15/15		
object literal extensions	6/6	6/6	6/6	6/6	6/6	6/6	6/6	6/6	6/6	6/6	6/6	6/6	6/6	6/6	6/6	6/6	6/6	6/6	6/6	6/6	6/6	6/6	6/6	6/6	6/6	6/6	6/6	6/6		
for...of loops	7/9	9/9	9/9	6/9	0/9	0/9	0/9	6/9	7/9	7/9	7/9	7/9	7/9	7/9	7/9	7/9	7/9	7/9	7/9	7/9	7/9	7/9	7/9	7/9	7/9	7/9	7/9	9/9		
octal and binary literals	4/4	7/4	4/4	7/4	4/4	7/4	0/4	4/4	4/4	4/4	4/4	4/4	4/4	4/4	4/4	4/4	4/4	4/4	4/4	4/4	4/4	4/4	4/4	4/4	4/4	4/4	4/4	4/4		
template strings	5/5	4/5	4/5	3/5	3/5	0/5	0/5	4/5	5/5	5/5	5/5	5/5	5/5	5/5	5/5	5/5	5/5	5/5	5/5	5/5	5/5	5/5	5/5	5/5	5/5	5/5	5/5	5/5		
RegExp 'v' and 'u' flags	0/2	2/4	2/4	0/4	0/4	0/4	0/4	2/4	4/4	3/4	2/4	2/4	2/4	2/4	2/4	2/4	2/4	2/4	2/4	2/4	2/4	2/4	2/4	2/4	2/4	2/4	2/4	2/4		
destructuring: declarations	0/2	20/23	21/22	18/22	11/22	0/22	0/22	0/22	0/22	19/22	19/22	19/22	0/22	21/22	21/22	0/22	9/22	19/22	22/22	0/22	0/22	0/22	0/22	0/22	0/22	0/22	12/22	21/22		
destructuring: assignments	0/24	23/24	24/24	15/24	18/24	0/24	0/24	0/24	0/24	20/24	21/24	21/24	0/24	23/24	23/24	0/24	12/24	21/24	24/24	0/24	0/24	0/24	0/24	0/24	0/24	0/24	14/24	24/24		
destructuring: parameters	0/23	19/23	20/23	17/23	11/23	0/23	0/23	0/23	0/23	18/23	18/23	18/23	0/23	22/23	22/23	0/23	10/23	18/23	22/23	0/23	0/23	0/23	0/23	0/23	0/23	0/23	12/23	23/23		
Unicode code point escapes	2/2	1/2	1/2	1/2	1/2	0/2	0/2	2/2	2/2	1/2	1/2	1/2	1/2	1/2	1/2	1/2	1/2	1/2	1/2	1/2	1/2	1/2	1/2	1/2	1/2	1/2	1/2	1/2		
arrow functions	2/2	0/2	0/2	0/2	0/2	0/2	0/2	0/2	1/2	0/2	2/2	2/2	2/2	2/2	2/2	2/2	0/2	0/2	0/2	1/2	0/2	0/2	0/2	0/2	0/2	2/2	2/2	2/2		
Bindings																														
const	5/6	6/6	6/6	6/6	6/6	0/6	6/6	6/6	6/6	6/6	6/6	6/6	6/6	6/6	6/6	6/6	6/6	6/6	6/6	6/6	6/6	6/6	6/6	6/6	6/6	6/6	6/6	6/6		
let	5/10	6/10	8/10	6/10	4/10	0/10	8/10	8/10	8/10	0/10	8/10	8/10	0/10	10/10	10/10	10/10	0/10	0/10	0/10	10/10	10/10	10/10	10/10	10/10	10/10	10/10	10/10	10/10		
block-level function declaration	Yes	Yes	Yes	Yes	No	No	Yes	Yes	Yes	No	No	No	Yes	Yes	Yes	No	No	No	No	No	No	Flags	Yes	Yes	No	Yes	Yes			



## 2) Tooling

# Tooling

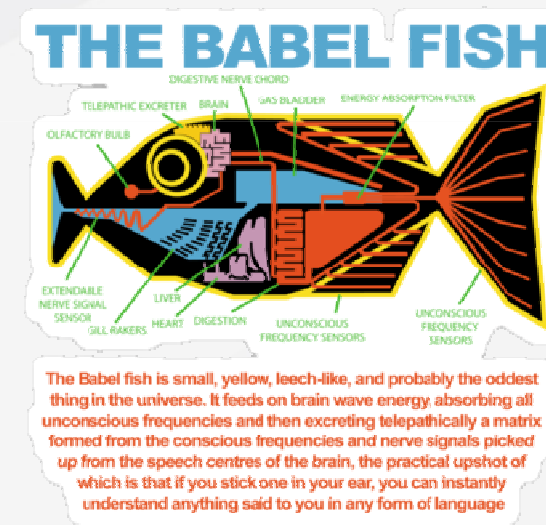


JS-to-JS **transpiler** is currently required to run ES6.

Transpiler compiles code from the latest version into older versions of the language.

As browser support gets better ES7 and ES8 will be transpiled into ES6 and beyond.

Transpilers like **Babel** also provide human-readable output.



## 3) Syntax

# Let & 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.

```
for(let i = 0, l = list.length; i < l; i++) {  
    // do something with list[i]  
}  
console.log(i); // undefined
```

WHISPER  
WORDS OF  
FREEDOM

*let it be*

# Let & Const

`Const` is also block-scoped, hoisted and must be initialized

Assigning to `const` after initialization **fails silently**

(or with an **exception** under strict mode).

```
const MY_CONSTANT = 1;
```

```
MY_CONSTANT = 2 // Error, attempt to change
```

```
const SOME_CONST; // Error, not initialized
```

Object properties can still be changed

```
const MY_OBJECT = {some: 1};
```

```
MY_OBJECT.some = 'body';
```

# Template Strings

Template strings provide syntactic sugar for constructing strings similar to string interpolation features in Perl, Python..

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



```
var name = "Bob", time = "today";  
`Hello ${name}, how are you ${time}?`
```

# Destructuring



Destructuring provides **binding** using **pattern matching**, with support for matching **arrays** and **objects**.

```
// Array matching
```

```
var list = [ 1, 2, 3 ]
```

```
var [ a, , b ] = list // a=1, b=3
```

```
[ b, a ] = [ a, b ] // a=3, b=1
```

```
// Fail-soft matching
```

```
var [missing] = [];
```

```
console.log(missing); // undefined
```

```
// Object matching
```

```
var robotA = { name: "Bender" };
```

```
var robotB = { name: "Flexo" };
```

```
var { name: nameA } = robotA;
```

```
console.log(nameA); // "Bender",
```

```
var { name: nameB } = robotB;
```

```
console.log(nameB); // "Flexo"
```

# Object Literals



Object literals are extended with several new features:

```
var obj = {  
    // __proto__  
    __proto__: theProtoObj,  
    // Shorthand for 'handler: handler'  
    handler,  
    // Methods  
    toString() {  
        // Super calls  
        return "d " + super.toString();  
    },  
    // Computed (dynamic) property names  
    [ 'prop_' + (() => 42)() ]: 42  
};
```



## 4) Functions

# Arrow Functions

Arrows are a function shorthand using the `param => return_value` syntax:

```
// Expression bodies
```

```
var odds = evens.map(v => v + 1);
```

```
var nums = evens.map((v, i) => v + i);
```

```
var pairs = evens.map(v => ({even: v, odd: v + 1}));
```

```
// Statement bodies
```

```
nums.forEach(v => {
```

```
  if (v % 5 === 0)
```

```
    fives.push(v);
```

```
});
```

# Arrow Functions

Arrow functions inherit THIS value from the enclosing scope:

```
this.nums.forEach((v) => {  
    if (v % 5 === 0)  
        this.fives.push(v)  
})
```

In ES5 we have to use self/that trick:

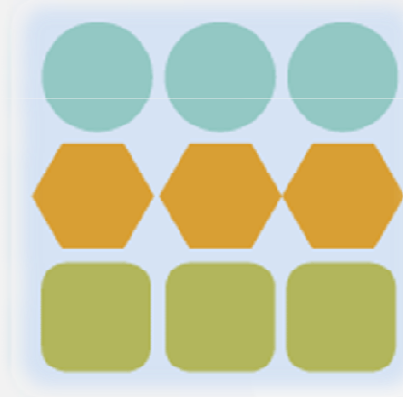
```
var self = this;  
this.nums.forEach(function (v) {  
    if (v % 5 === 0)  
        self.fives.push(v);  
})
```

## 5) Classes

# Classes (1)

ES6 classes are syntactic sugar over the **prototype-based** OO pattern.

Classes are a well-debated feature of ES6.



Some believe that they go **against the prototypal nature** of JavaScript, while others think they **lower the entry barrier** for beginners and people coming from other languages.

# Classes (2)

```
class Vehicle {  
    constructor(name) {  
        this.name = name;  
        this.kind = 'vehicle';  
    }  
    getName() {  
        return this.name;  
    }  
}  
  
// Create an instance  
let myVehicle = new Vehicle('rocky');
```

## 6) Collections

# Iterators



```
let fibonacci = {  
  [Symbol.iterator]() { // default iterator for an object.  
    let pre = 0, cur = 1;  
    return {  
      next() { // required method  
        [pre, cur] = [cur, pre + cur];  
        return { done: false, value: cur }  
      }  
    }  
  }  
}
```



# For .. of Loop

For .. of loop is new loop for all iterables.

It starts by calling the `[Symbol.iterator]()` method which returns a new iterator object.

An iterator object can be any object with a `next()` method.

```
for (var n of fibonacci) {  
    // truncate the sequence at 1000  
    if (n > 1000)  
        break;  
    console.log(n);  
}
```

# Map + Set



```
// Maps
```

```
var m = new Map();
```

```
m.set("hello", 42);
```

```
m.set(s, 34);
```

```
m.get(s) == 34;
```

```
// Sets
```

```
var s = new Set();
```

```
s.add("hello").add("goodbye").add("hello");
```

```
s.size === 2;
```

```
s.has("hello") === true;
```

# WeakMap + WeakSet



Weak collections allow GC collection of their keys.

```
// Weak Maps
```

```
var wm = new WeakMap();
```

```
wm.set(s, { extra: 42 });
```

```
wm.size === undefined
```

```
// Weak Collections are not enumerable and do not have size
```

```
// Weak Sets
```

```
var ws = new WeakSet();
```

```
ws.add({ data: 42 });
```

```
// If data has no other references, it can be GC collected
```

## 7) Modules

# Modules (1)

Language-level support for modules for component definition.

Codifies patterns from [AMD](#), [CommonJS](#) ..



Runtime behaviour defined by a host-defined default loader.

**Implicitly async model** – no code executes until requested modules are available and processed.

## Modules (2)



```
// lib/math.js
export function sum(x, y) {
  return x + y;
}
export var pi = 3.141593;
```

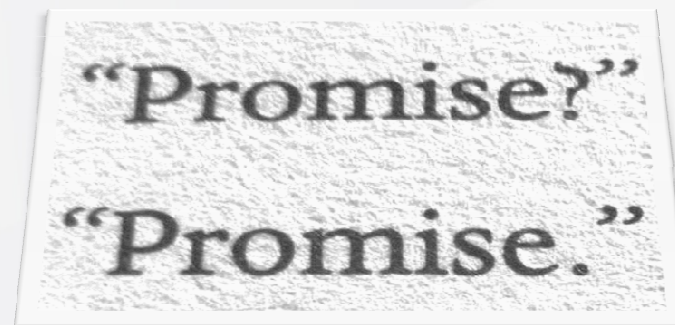
```
// app.js
import * as math from "lib/math";
alert("2π = " + math.sum(math.pi, math.pi));
```

## 8) Promises

# Promises

Promises are used for **asynchronous programming**.

Promises are first class representation of a value that may be made available in the future.



Promises are used in many existing JavaScript libraries.



# Promises



```
function resolveUnderThreeSeconds (delay) {  
  return new Promise(function (resolve, reject) {  
    setTimeout(resolve, delay);  
    // once a promise is settled, it's result can't change  
    setTimeout(reject, 3000);  
  })  
}  
  
resolveUnderThreeSeconds(2000); // resolves!  
resolveUnderThreeSeconds(7000);  
// fulfillment took so long, it was rejected.
```

## 9) The Future

# Future releases (1)

ES7 is due 12 months after ES6.

TC39 (the ECMAScript standard committee)  
is embracing **12 month release cycle**.

New, naming convention:

ES6 = ES2015

ES7 = ES2016

ES8 = ES2017 ...



Completed features will be published once a year,  
others will be scheduled for the next release.

## Future releases (2)



*„ECMAScript is a vibrant language and the evolution of the language is not complete. Significant technical enhancement will continue with future editions of this specification.”*

... ES 5 specification



## Future releases (3)

New proposed features for ES7:

- **concurrency** and atomics,
- zero-copy **binary data transfer**,
- number and math enhancements,
- observable **streams**,
- better **metaprogramming** with classes,
- class and instance **properties**,
- operator **overloading**,
- **value types** (first-class primitive-like objects),
- records, tuples, **traits** ...

The End



To be continued ...