ECMAScript 6: what's next for JavaScript?

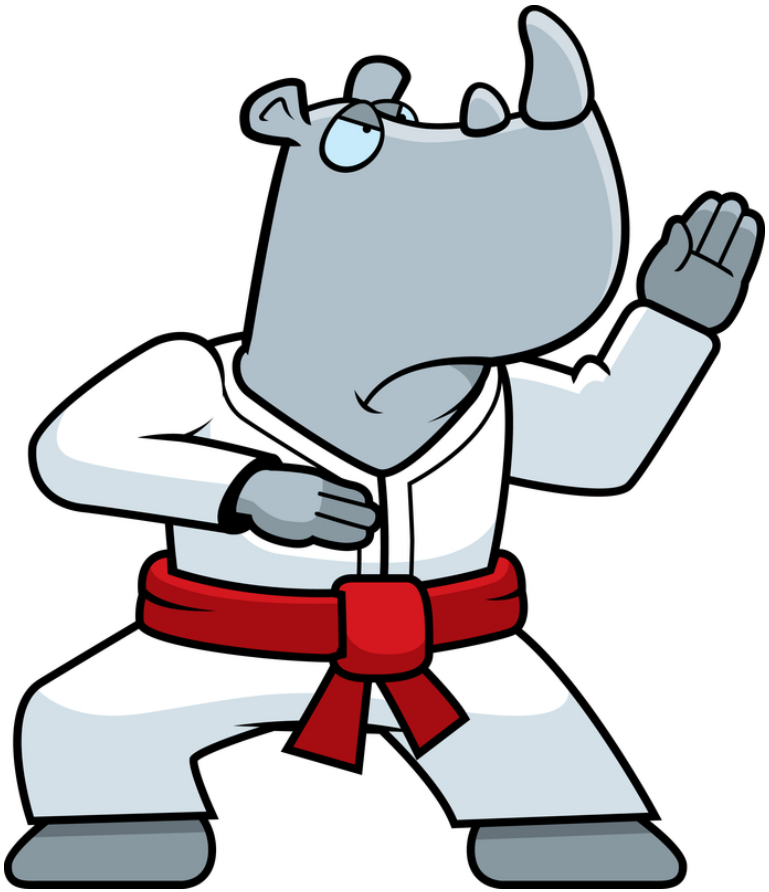
Dr. Axel Rauschmayer

rauschma.de

2014-06-13

QCon New York 2014

JavaScript has become dangerous



- Used everywhere: browsers, servers, devices, . . .
- For much more than it was created for
- Let's make it better at those tasks. . .

ECMAScript 6 (ES6): JavaScript, improved

ECMAScript 6: next version of JavaScript (current: ECMAScript 5).

This talk:

- Goals
- Design process
- Features
- When can I use it?

Background

Important ES6 terms

- **TC39 (Ecma Technical Committee 39):** the committee evolving JavaScript.
  - Members: companies (all major browser vendors etc.).
  - Meetings attended by employees and invited experts.
- **ECMAScript:** the official name of the language
  - Versions: ECMAScript 5 is short for "ECMAScript Language Specification, Edition 5"
- **JavaScript:**
  - colloquially: the language
  - formally: one implementation of ECMAScript
- **ECMAScript Harmony:** improvements after ECMAScript 5 (ECMAScript 6 and 7)

Goals for ECMAScript 6

Amongst other official goals [1]: make JavaScript better

- for complex applications
- for libraries (including the DOM)
- as a target of code generators

How ECMAScript features are designed

Avoid “design by committee”:

- Design by “champions” (1–2 experts)
- Feedback from TC39 and the web development community
- Field-testing and refining via one or more implementations
- TC39 has final word on whether/when to include

How to upgrade a web language?

Challenges w.r.t. upgrading:

1. JavaScript engines:
   - New versions = forced upgrades
   - Must run all existing code

   ⇒ ECMAScript 6 only adds features

2. JavaScript code:
   - Must run on all engines that are in use

   ⇒ wait or compile ECMAScript 6 to ES5 (details later).

Variables and scoping

Block-scoped variables

Function scope (var)

```
function order(x, y) {
    if (x > y) {
        var tmp = x;
        x = y;
        y = tmp;
    }
    console.log(tmp===x);
        // true

    return [x, y];
}
```

Block scope (let, const)

```
function order(x, y) {
    if (x > y) {
        let tmp = x;
        x = y;
        y = tmp;
    }
    console.log(tmp===x);
        // ReferenceError:
        // tmp is not defined
    return [x, y];
}
```

Destructuring: objects

Extract data (more than one value!) via patterns:

```
let obj = { first: 'Jane', last: 'Doe' };

let { first: f, last: l } = obj;
console.log(f + ' ' + l);  // Jane Doe
```

Usage:

- variable declarations
- assignments
- parameter definitions

Object literals: property value shorthand

Shorthand: `{x,y}` is the same as `{ x: x, y: y }`.

```
let obj = { first: 'Jane', last: 'Doe' };

let { first, last } = obj;
console.log(first + ' ' + last);  // Jane Doe
```

Multiple return values

```
function findElement(arr, predicate) {
    for (let index=0; index < arr.length; index++) {
        let element = arr[index];
        if (predicate(element)) {
            return { element, index };
        }
    }
    return { element: undefined, index: -1 };
}

let {element} = findElement(someArray, somePredicate);
let {index} = findElement(someArray, somePredicate);

// Order doesn't matter:
let {index, element} = findElement(...);
let {element, index} = findElement(...);
```

Destructuring: arrays

```
let [x, y] = [ 'a', 'b' ];
// x='a', y='b'

let [x, y, ...rest] = [ 'a', 'b', 'c', 'd' ];
// x='a', y='b', rest = [ 'c', 'd' ]

[x,y] = [y,x];  // swap values

let [all, year, month, day] =
    /^(\d\d\d\d)-(\d\d)-(\d\d)$/
    .exec('2999-12-31');
```

Destructuring: refutable by default

Refutable (default): exception if pattern part has no match.

```
{ a: x, b: y } = { a: 3 }; // TypeError

[x, y] = ['a']; // TypeError
[x, y] = ['a', 'b', 'c']; // OK: x='a', y='b'
```

Default value: use if no match or value is `undefined`

```
{ a: x, b: y=5 } = { a: 3 }; // x=3, y=5
{ a: x, b: y=5 } = { a: 3, b: undefined }; // x=3, y=5

[x, y='b'] = ['a']; // x='a', y='b'
[x, y='b'] = ['a', undefined]; // x='a', y='b'
```

Parameter handling

Parameter handling 1: parameter default values

Use a default value if parameter is missing.

```
function func1(x, y=3) {
    return [x,y];
}
```

Interaction:

```
# func1(1, 2)
[1, 2]
# func1(1)
[1, 3]
# func1()
[undefined, 3]
```

Parameter handling 2: rest parameters

Put trailing parameters in an array.

```
function func2(arg0, ...others) {
    return others;
}
```

Interaction:

```
# func2(0, 1, 2, 3)
[1, 2, 3]
# func2(0)
[]
# func2()
[]
```

No need for `arguments`, anymore.

Spread operator (...)

Turn an array into function/method arguments:

```
# Math.max(7, 4, 11)
11
# Math.max(...[7, 4, 11])
11
```

- The inverse of a rest parameter
- Mostly replaces `Function.prototype.apply()`
- Also works in constructors

Parameter handling 3: named parameters

Use destructuring for named parameters `opt1` and `opt2`:

```
function func3(arg0, { opt1, opt2 }) {
    return [opt1, opt2];
}
// {opt1,opt2} is same as {opt1:opt1,opt2:opt2}
```

Interaction:

```
# func3(0, { opt1: 'a', opt2: 'b' })
['a', 'b']
```

Arrow functions

Arrow functions: less to type

Compare:

```
let squares = [1, 2, 3].map(function (x) {return x * x});
```

```
let squares = [1, 2, 3].map(x => x * x);
```

Arrow functions: lexical `this`, no more that=this

```
function UiComponent {
    var that = this;
    var button = document.getElementById('#myButton');
    button.addEventListener('click', function () {
        console.log('CLICK');
        that.handleClick();
    });
}
UiComponent.prototype.handleClick = function () { ... };
```

```
function UiComponent {
    let button = document.getElementById('#myButton');
    button.addEventListener('click', () => {
        console.log('CLICK');
        this.handleClick();
    });
}
```

Arrow functions: versions

General form:

```
(arg1, arg2, ...) => expr
(arg1, arg2, ...) => { stmt1; stmt2; ... }
```

Shorter version – single parameter:

```
arg => expr
arg => { stmt1; stmt2; ... }
```

Object-orientation and modularity

Object literals

ECMAScript 6:

```
let obj = {
    __proto__: someObject,  // special property

    myMethod(arg1, arg2) {  // method definition
        ...
    }
};
```

ECMAScript 5:

```
var obj = Object.create(someObject);
obj.myMethod = function (arg1, arg2) {
    ...
};
```

Classes

```
class Point {
    constructor(x, y) {
        this.x = x;
        this.y = y;
    }
    toString() {
        return '('+this.x+', '+this.y+')';
    } }
```

```
function Point(x, y) {
    this.x = x;
    this.y = y;
}
Point.prototype.toString = function () {
    return '('+this.x+', '+this.y+')';
};
```

Classes: subclass

```
class ColorPoint extends Point {
    constructor(x, y, color) {
        super(x, y);  // same as super.constructor(x, y)
        this.color = color;
    }
    toString() {
        return this.color+' '+super();
    } }
```

```
function ColorPoint(x, y, color) {
    Point.call(this, x, y);
    this.color = color; }
ColorPoint.prototype = Object.create(Point.prototype);
ColorPoint.prototype.constructor = ColorPoint;
ColorPoint.prototype.toString = function () {
    return this.color+' '+Point.prototype.toString.call(this);
};
```

Modules: overview

```
// lib/math.js
let notExported = 'abc';
export function square(x) {
    return x * x;
}
export const MY_CONSTANT = 123;
```

```
// main.js
import {square} from 'lib/math';
console.log(square(3));
```

Modules: features

More features [3]:

- Rename imports
- Module IDs are configurable (default: paths relative to importing file)
- Programmatic (e.g. conditional) loading of modules via an API

Template strings

Template strings: string interpolation

Invocation:

```
templateHandler`Hello ${first} ${last}!`
```

Syntactic sugar for:

```
templateHandler(['Hello ', ' ', '!'], first, last)
```

Two kinds of tokens:

- Literal sections (static): `'Hello'`
- Substitutions (dynamic): `first`

Template strings: interpolation, raw strings

No handler ⇒ string interpolation.

```
if (x > MAX) {
    throw new Error(`At most ${MAX} allowed: ${x}`);
}
```

Multiple lines, no escaping:

```
var str = raw`This is a text
with multiple lines.

Escapes are not interpreted,
\n is not a newline.`;
```

Template strings: regular expressions

ECMAScript 6: XRegExp library – ignored whitespace, named groups, comments

```
let str = '/2012/10/Page.html';
let parts = str.match(XRegExp.rx`
    ^ # match at start of string only
    / (?<year> [^/]+ ) # capture top dir as year
    / (?<month> [^/]+ ) # capture subdir as month
    / (?<title> [^/]+ ) # file name base
    \.html? # file name extension: .htm or .html
    $ # end of string
`);

console.log(parts.year); // 2012
```

Advantages:

- Raw characters: no need to escape backslash and quote
- Multi-line: no need to concatenate strings with newlines at the end

Template strings: regular expressions

ECMAScript 5:

```
var str = '/2012/10/Page.html';
var parts = str.match(XRegExp(
    '^ # match at start of string only \n' +
    '/ (?<year> [^/]+ ) # capture top dir as year \n' +
    '/ (?<month> [^/]+ ) # capture subdir as month \n' +
    '/ (?<title> [^/]+ ) # file name base \n' +
    '\\.html? # file name extension: .htm or .html \n' +
    '$ # end of string',
    'x'
));
```

Template strings: other use cases

- Query languages
- Text localization
- Templating
- etc.

Standard library

Maps

Data structure mapping from arbitrary values to arbitrary values (objects: keys must be strings).

```
let map = new Map();
let obj = {};

map.set(obj, 123);
console.log(map.get(obj)); // 123
console.log(map.has(obj)); // true

map.delete(obj);
console.log(map.has(obj)); // false
```

Also: iteration (over keys, values, entries) and more.

Sets

A collection of values without duplicates.

```
let set1 = new Set();
set1.add('hello');
console.log(set1.has('hello')); // true
console.log(set1.has('world')); // false

let set2 = new Set([3,2,1,3,2,3]);
console.log(set2.values());  // 1,2,3
```

Object.assign

Merge one object into another one.

```
class Point {
    constructor(x, y) {
        Object.assign(this, { x, y });
    }
}
```

Similar to `_.extend()` from Underscore.js.

Standard library: new string methods

```
# 'abc'.repeat(3)
'abcabcabc'
# 'abc'.startsWith('ab')
true
# 'abc'.endsWith('bc')
true
# 'foobar'.contains('oo')
true
```

And more.

Standard library: new array methods

```
# [13, 7, 8].find(x => x % 2 === 0)
8
# [1, 3, 5].find(x => x % 2 === 0)
undefined

# [13, 7, 8].findIndex(x => x % 2 === 0)
2
# [1, 3, 5].findIndex(x => x % 2 === 0)
-1
```

And more.

Loops and iteration

Iterables and iterators

Iteration protocol:

- **Iterable:** a data structure whose elements can be traversed
- **Iterator:** the pointer used for traversal

Examples of iterables:

- Arrays
- Sets
- All array-like DOM objects (eventually)

for-of: a better loop

- Replaces:
  - `for-in`
  - `Array.prototype.forEach()`
- Works for: iterables
  - Convert array-like objects via `Array.from()`.

for-of loop: arrays

```
let arr = ['hello', 'world'];
for (let elem of arr) {
    console.log(elem);
}
```

Output – elements, not indices:

```
hello
world
```

for-of loop: arrays

```
let arr = ['hello', 'world'];
for (let [index, elem] of arr.entries()) {
    console.log(index, elem);
}
```

Output:

```
0 hello
1 world
```

Generators: example

Suspend via `yield` ("resumable return"):

```
function* generatorFunction() {
    yield 0;
    yield 1;
    yield 2;
}
```

Start and resume via `next()`:

```
let genObj = generatorFunction();
console.log(genObj.next());  // { value: 0, done: false }
console.log(genObj.next());  // { value: 1, done: false }
console.log(genObj.next());  // ( value: 2, done: false }
console.log(genObj.next());  // ( value: undefined, done: true }
```

Generators: implementing an iterator

```
function* iterEntries(obj) {
    let keys = Object.keys(obj);
    for (let i=0; i < keys.length; i++) {
        let key = keys[i];
        yield [key, obj[key]];
    }
}
let myObj = { foo: 3, bar: 7 };
for (let [key, value] of iterEntries(myObj)) {
    console.log(key, value);
}
```

Output:

```
foo 3
bar 7
```

Generators: asynchronous programming

Using the Q promise library:

```
Q.spawn(function* () {
    try {
        let [foo, bar] = yield Q.all(
            [ read('foo.json'), read('bar.json') ] );
        render(foo);
        render(bar);
    } catch (e) {
        console.log("read failed: " + e);
    }
});
```

Wait for asynchronous calls via `yield` (internally based on promises).

Symbols

Symbols

- Inspired by Lisp, Smalltalk etc.
- A new kind of primitive value:

```
# let sym = Symbol();
# typeof sym
'symbol'
```

- Each symbol is unique.

Symbols: enum-style values

```
const red = Symbol();
const green = Symbol();
const blue = Symbol();

function handleColor(color) {
    switch(color) {
        case red:
            ...
        case green:
            ...
        case blue:
            ...
    }
}
```

Previously:

```
var red = 'red';
var green = 'green';
var blue = 'blue';
```

Symbols: property keys

```
let specialMethod = Symbol();
let obj = {
    // computed property key
    [specialMethod]: function (arg) {
        ...
    }
};
obj[specialMethod](123);
```

Shorter – method definition syntax:

```
let obj = {
    [specialMethod](arg) {
        ...
    }
};
```

Symbols: property keys

- Advantage: No name clashes!
- Configure objects for ECMAScript and frameworks:
  - Introduce publicly known symbols.
  - Example: property key `Symbol.iterator` makes an object iterable.

When?

Various other features

Also part of ECMAScript 6:

- Promises
- Better support for Unicode (strings, regular expressions)
- Optimized tail calls
- Proxies (meta-programming)

Candidates for ECMAScript 7:

- Handling binary data
- `Object.observe()` for data binding
- Integers (64 bit, 32 bit, etc.)

Time table

ECMAScript 6 is basically done:

- Its feature set is frozen.
- It is mostly being refined now.

Time table:

- End of 2014: specification is finished (except fixing last bugs)
- March 2015: publication process starts
- June 2015: formal publication

Using ECMAScript 6 today

- Features are continually appearing in engines [4]
- TypeScript: ECMAScript 6 plus (optional) type annotations
- Traceur: ES6-to-ES5 compiler that many solutions are based on:
  - Plugins for Grunt, Gulp, Broccoli, etc.
  - es6ify: transform for Browserify
- ES6 Module Transpiler: compile ES6 modules (subset of ES6) to AMD or CJS
- ES6 Fiddle: interactively try out ES6 (based on Traceur)
- Frameworks:
  - Ember.js 1.6 is based on ECMAScript 6 modules (via ES6 Module Transpiler)
  - AngularJS 2 is based on ECMAScript 6 (via Traceur)
- es6-shim by Paul Miller: features of the ES6 standard library, backported to ES5.

More information: es6-tools by Addy Osmani.

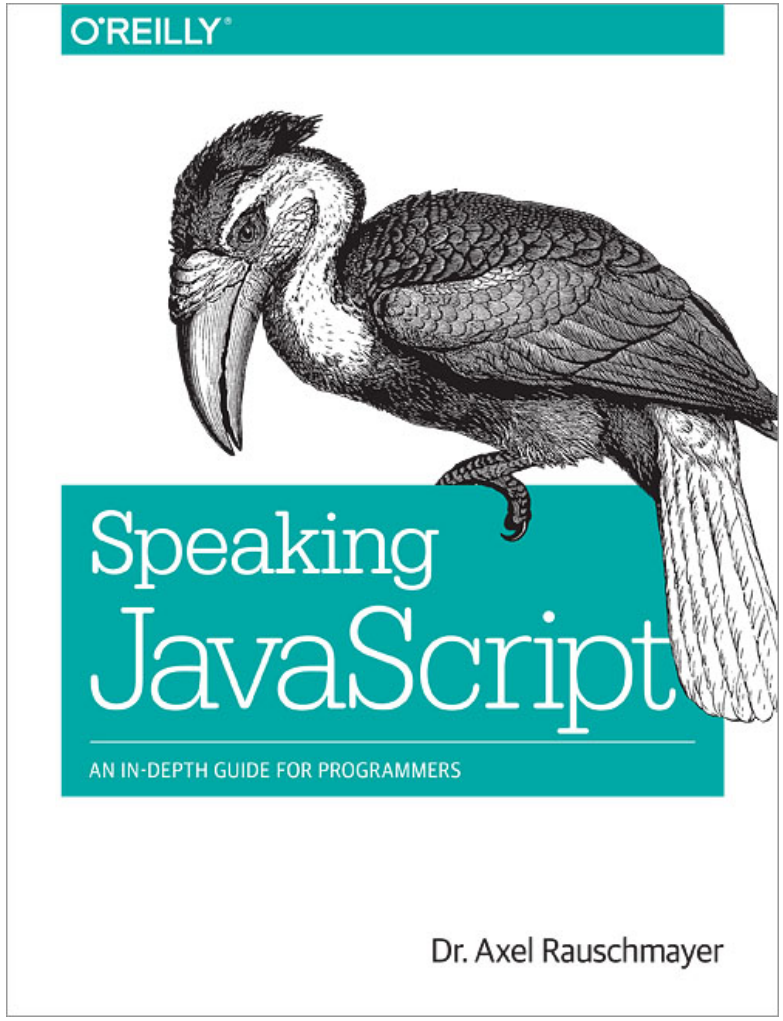
Conclusion

Take-aways: ECMAScript 6

- Some features are already in engines [4]
- Can be used today, by compiling to ECMAScript 5
- Biggest impact on community (currently: too much variety):
  - Classes
  - Modules

Thank you!

- My book (free online!): SpeakingJS.com
- Blog posts on ECMAScript 6: 2ality.com/search/label/esnext



Annex

References

1. ECMAScript Harmony wiki
2. “The Harmony Process” by David Herman
3. “ES6 Modules” by Yehuda Katz
4. “ECMAScript 6 compatibility table” by kangax [features already in JavaScript engines]

Resources

- ECMAScript 6 specification drafts by Allen Wirfs-Brock
- ECMAScript mailing list: es-discuss
- TC39 meeting notes by Rick Waldron
- "A guide to 2ality's posts on ECMAScript 6" by Axel Rauschmayer
- Continuum, an ECMAScript 6 virtual machine written in ECMAScript 3.

(Links are embedded in this slide.)