From ES5 to ES6 and ECMAScript 2016

Axel Rauschmayer

eBay Tech Talk
Berlin, 9 May 2016

Overview

- Upgrading from ES5: easy ES6 features
- Evolving JavaScript: the TC39 process
- The features of ES2016 and ES2017

© by Kenny Louie

Upgrading from ES5: easy ES6 features

From `var` to `let/const`

```
// Why?

var x = 3;
function func(randomize) {
    if (randomize) {
        var x = Math.random();
        return x;
    }
    return x;
}
func(false); // undefined
```

From var to let/const

```
// Same behavior, easier to understand:

var x = 3;
function func(randomize) {
    var x;
    if (randomize) {
        x = Math.random();
        return x;
    }
    return x;
}
func(false); // undefined
```

From var to let/const

```
// `let` instead of `var`: behavior changes

let x = 3;
function func(randomize) {
    if (randomize) {
        let x = Math.random();
        return x;
    }
    return x;
}
func(false); // 3
```

`const` vs. `let` vs. `var`

- Prefer `const`.
    - Requirement: variable always has same value.
- Otherwise, use `let`.
- Avoid `var`.

Thus: `const` ⇒ variable doesn’t change.

From IIFEs to blocks

```
(function () {  // open IIFE
    var tmp = ···;
    ···
}());  // close IIFE

console.log(tmp); // ReferenceError
```

From IIFEs to blocks

```
{  // open block
    let tmp = ···;
    ···
}  // close block

console.log(tmp); // ReferenceError
```

From concatenating strings to template literals

```
// ES5: string concatenation via +
function printCoord(x, y) {
    console.log('('+x+', '+y+')');
}

// ES6: string interpolation
function printCoord(x, y) {
    console.log(`(${x}, ${y})`);
}
```

From concatenating strings to template literals

```
// ES5: multi-line strings

var HTML5_SKELETON =
    '<!doctype html>\n' +
    '<html>\n' +
    '<head>\n' +
    '    <meta charset="UTF-8">\n' +
    '    <title></title>\n' +
    '</head>\n' +
    '<body>\n' +
    '</body>\n' +
    '</html>\n';
```

From concatenating strings to template literals

```
// ES5: multi-line strings

var HTML5_SKELETON = '\
    <!doctype html>\n\
    <html>\n\
    <head>\n\
        <meta charset="UTF-8">\n\
        <title></title>\n\
    </head>\n\
    <body>\n\
    </body>\n\
    </html>';
```

From concatenating strings to template literals

```
// ES6: multi-line strings

const HTML5_SKELETON = `
    <!doctype html>
    <html>
    <head>
        <meta charset="UTF-8">
        <title></title>
    </head>
    <body>
    </body>
    </html>`;
```

From function expressions to arrow functions

```
// ECMAScript 5

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

From function expressions to arrow functions

```
// ECMAScript 6

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

From function expressions to arrow functions

```
var squares = arr.map(
    function (x) { return x * x }); // ES5

let squares = arr.map(x => x * x); // ES6
```

Handling multiple return values

```
// ES5: multiple return values via Arrays

var matchObj =
    /^(\d\d\d\d)-(\d\d)-(\d\d)$/
    .exec('2999-12-31');
var year = matchObj[1];
var month = matchObj[2];
var day = matchObj[3];
```

Handling multiple return values

```
// ES6: access multiple return values via
// Array destructuring

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

Handling multiple return values

```
// ES5: multiple return values via objects

var obj = { foo: 123 };

var propDesc =
Object.getOwnPropertyDescriptor(obj, 'foo');

var writable     = propDesc.writable;
var configurable = propDesc.configurable;

console.log(writable, configurable); // true true
```

Handling multiple return values

```
// ES6: access multiple return values
// via object destructuring

let obj = { foo: 123 };

let {writable, configurable} =
    Object.getOwnPropertyDescriptor(obj, 'foo');

console.log(writable, configurable); // true true
```

Abbreviation:

```
{writable, configurable}
{writable: writable, configurable: configurable}
```

From `for` to `.forEach()` to `for-of`

```
// Prior to ES5: `for` loop
// Benefit: `break`

var arr = ['a', 'b', 'c'];
for (var i=0; i<arr.length; i++) {
    var elem = arr[i];
    console.log(elem);
}
```

From `for` to `.forEach()` to `for-of`

```
// ES5: Array method forEach()
// Benefit: concise

arr.forEach(function (elem) {
    console.log(elem);
});
```

From `for` to `.forEach()` to `for-of`

```
// ES6: for-of loop

let arr = ['a', 'b', 'c'];
for (let elem of arr) {
    console.log(elem);
}
```

From `for` to `.forEach()` to `for-of`

```
// ES6: for-of loop

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

Handling parameter default values

```
// ES5:
function foo(x, y) {
    x = x || 0;
    y = y || 0;
    ...
}

// ES6:
function foo(x=0, y=0) {
    ...
}
```

Only triggered by **`undefined`** (vs. any falsy value).

Handling named parameters

```
selectEntries({ start: 0, end: -1 });

// ES5:
function selectEntries(options) {
    var start = options.start || 0;
    var end = options.end || -1;
    var step = options.step || 1;
    ...
}
```

Handling named parameters

```
selectEntries({ start: 0, end: -1 });

// ES6:
function selectEntries({
    start=0, end=-1, step=1 }) {
    ...
}
```

From `arguments` to rest parameters

```
// ES5:
function format(pattern) {
    var args = [].slice.call(arguments, 1);
     ...
}

// ES6:
function format(pattern, ...args) {
     ...
}
```

From `apply()` to the spread operator (`...`)

```
// ES5
Math.max.apply(null, [-1, 5, 11, 3]); // 11

// ES6
Math.max(...[-1, 5, 11, 3]); // 11
```

From `apply()` to the spread operator (`...`)

```
// ES5
var arr1 = ['a', 'b'];
var arr2 = ['c', 'd'];
arr1.push.apply(arr1, arr2);
    // arr1 is now ['a', 'b', 'c', 'd']


// ES6
let arr1 = ['a', 'b'];
let arr2 = ['c', 'd'];
arr1.push(...arr2);
    // arr1 is now ['a', 'b', 'c', 'd']
```

From `concat()` to the spread operator (`...`)

```
// ES5
var arr1 = ['a', 'b'];
var arr2 = ['c'];
var arr3 = ['d', 'e'];

console.log(arr1.concat(arr2, arr3));
    // [ 'a', 'b', 'c', 'd', 'e' ]

// ES6
let arr1 = ['a', 'b'];
let arr2 = ['c'];
let arr3 = ['d', 'e'];

console.log([...arr1, ...arr2, ...arr3]);
    // [ 'a', 'b', 'c', 'd', 'e' ]
```

From function expressions in object literals to method definitions

```
// ES5
var obj = {
    foo: function () {
        ...
    },
    bar: function () {
        this.foo();
    }, // trailing comma is legal in ES5
};

// ES6
let obj = {
    foo() {
        ...
    },
    bar() {
        this.foo();
    },
};
```

From constructors to classes: base classes

```
// ES5
function Person(name) {
    this.name = name;
}
Person.prototype.describe = function () {
    return 'Person called '+this.name;
};
// ES6
class Person {
    constructor(name) {
        this.name = name;
    }
    describe() {
        return 'Person called '+this.name;
    }
}
```

From constructors to classes: derived classes

```
// ES5
function Employee(name, title) {
    Person.call(this, name); // super(name)
    this.title = title;
}
Employee.prototype = Object.create(Person.prototype);
Employee.prototype.constructor = Employee;
Employee.prototype.describe = function () {
    return Person.prototype.describe.call(this) // super.describe()
           + ' (' + this.title + ')';
};
// ES6
class Employee extends Person {
    constructor(name, title) {
        super(name);
        this.title = title;
    }
    describe() {
        return super.describe() + ' (' + this.title + ')';
    }
}
```

From custom error constructors to subclasses of `Error`

```
// ES5
function MyError() {
    // Use Error as a function
    var superInst = Error.apply(null, arguments);
    copyOwnPropertiesFrom(this, superInst);
}
MyError.prototype = Object.create(Error.prototype);
MyError.prototype.constructor = MyError;


// ES6
class MyError extends Error {
}
```

From objects to Maps

```
// ES5
var dict = Object.create(null);
/** Keys are words, values are counts */
function countWords(word) {
    var escapedWord = escapeKey(word);
    if (escapedWord in dict) {
        dict[escapedWord]++;
    } else {
        dict[escapedWord] = 1;
    }
}
function escapeKey(key) { ··· }

// ES6
let map = new Map();
function countWords(word) {
    let count = map.get(word) || 0;
    map.set(word, count + 1);
}
```

From CommonJS modules to ES6 modules (ES5)

```
//------ lib.js ------
var sqrt = Math.sqrt;
function square(x) {
    return x * x;
}
function diag(x, y) {
    return sqrt(square(x) + square(y));
}
module.exports = {
    sqrt: sqrt,
    square: square,
    diag: diag,
};

//------ main1.js ------
var square = require('lib').square;
var diag = require('lib').diag;

console.log(square(11)); // 121
console.log(diag(4, 3)); // 5
```

From CommonJS modules to ES6 modules (ES6)

```
//------ lib.js ------
export const sqrt = Math.sqrt;
export function square(x) {
    return x * x;
}
export function diag(x, y) {
    return sqrt(square(x) + square(y));
}

//------ main1.js ------
import { square, diag } from 'lib';
console.log(square(11)); // 121
console.log(diag(4, 3)); // 5
```

New string methods

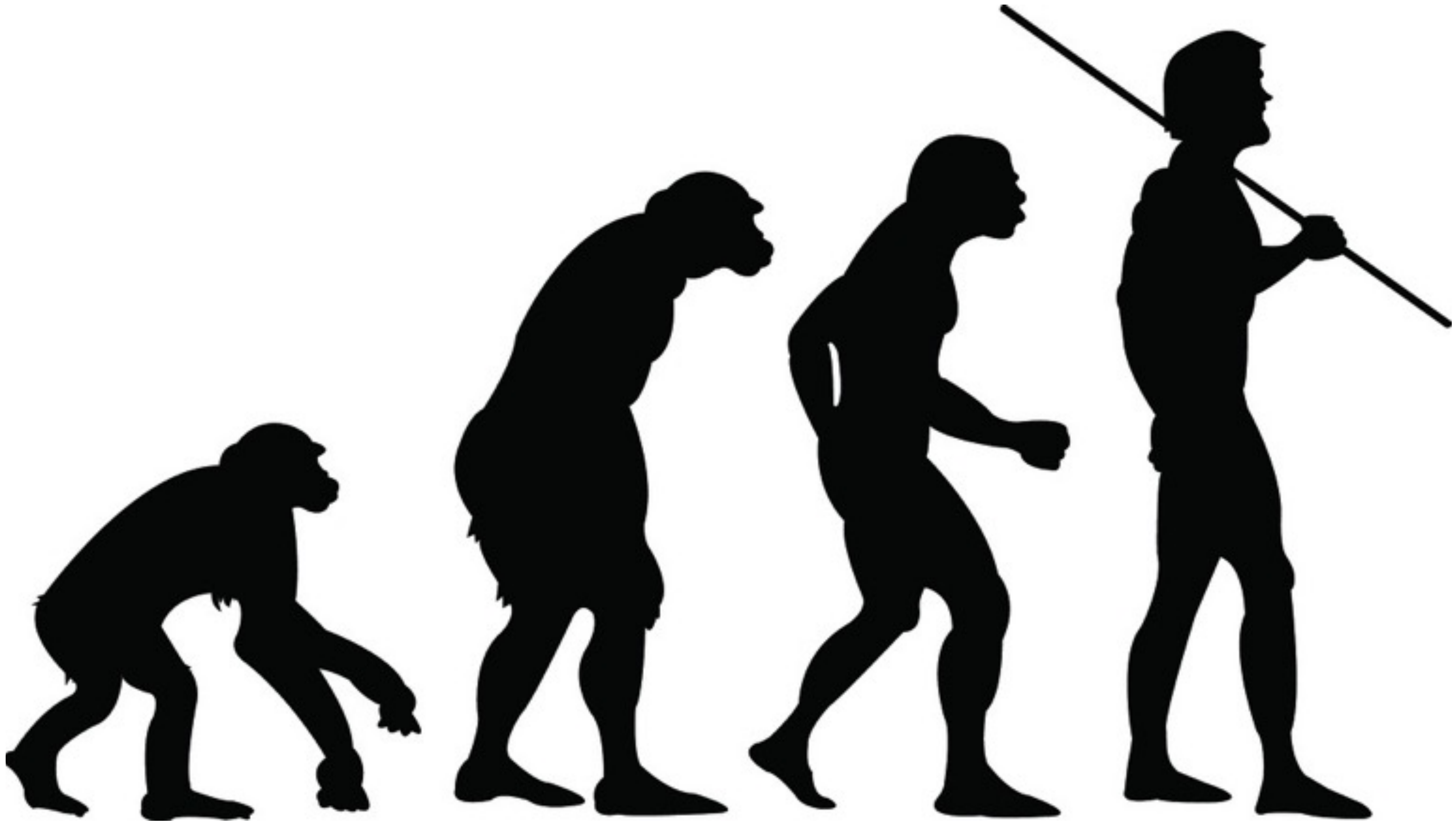
```
// From indexOf to startsWith
if (str.indexOf('x') === 0) {} // ES5
if (str.startsWith('x')) {} // ES6

// From indexOf to endsWith
function endsWith(str, suffix) { // ES5
  var index = str.indexOf(suffix);
  return index >= 0
    && index === str.length-suffix.length;
}
str.endsWith(suffix); // ES6
```

New string methods

```
// From indexOf to includes
if (str.indexOf('x') >= 0) {} // ES5
if (str.includes('x')) {} // ES6

// From join to repeat
new Array(3+1).join('#') // ES5
'#'.repeat(3) // ES6
```


© by Bryan Wright

Evolving JavaScript: the TC39 process

Ecma Technical Committee 39 (TC39)

- TC39: the committee evolving JavaScript
- Members: companies (all major browser vendors etc.)
- Bi-monthly meetings of delegates and invited experts

Ecma Technical Committee 39 (TC39)

github.com/hemanth/tc39-members

```
{
"members": {
  "Ordinary": [
    "Adobe",
    "AMD",
    "eBay",
    "Google",
    "HewlettPackard",
    "Hitachi",
    "IBM",
    "Intel",
    "KonicaMinolta"
  ],
  "Associate": [
    "Apple",
    "Canon",
    "Facebook",
    "Fujitsu",
    "JREastMechatronics",
    "Netflix",
    "NipponSignal",
    "NXP",
    "OMRONSocialSolutions",
    "Ricoh",
    "Sony",
    "Toshiba",
    "Twitter"
  ],
...
```

The TC39 process

Problems with infrequent, large releases (e.g. ES6):

- Features that are ready sooner have to wait
- Features that are not ready are under pressure to get finished, may delay release
    - Next release would be too late.

New TC39 process:

- Each proposal goes through maturity stages, numbered 0–4
- Spec is ratified once a year
    - Only features that are ready in time are added

Stage 0: strawman

What is it?

- First sketch
- Submitted by TC39 member or registered TC39 contributor

What’s required?

- Review at TC39 meeting

Stage 1: proposal

What is it?

- Formal proposal of a feature

What's required?

- Identify champion(s), one of them a TC39 member
- Describe problem: prose, examples, API, semantics and algorithms
- Identify potential obstacles (interactions with other features etc.)
- Implementation: polyfills and demos

What's next?

- TC39 is willing to help with designing the feature
- Major changes are still expected

Stage 2: draft

What is it?

- First version of what will be in the spec
- Eventual standardization is likely

What’s required?

- Formal description of syntax and semantics
- As complete as possible, gaps are OK
- Two experimental implementations (one of them can be a transpiler)

What’s next?

- Only incremental changes are expected

Stage 3: candidate

What is it?

- Proposal is mostly finished, now needs feedback from implementations

What's required?

- Spec text is complete
- Signed off by reviewers and ES spec editor
- At least two spec-compliant implementations

What's next?

- Changes only in response to critical issues.

Stage 4: finished

What is it?

- Proposal ready to be included in the ES specification

What's required?

- Test 262 acceptance tests
- Two spec-compliant shipping implementations that pass the tests
- Significant practical experience with the implementations
- ECMAScript spec editor must sign off on the spec text

What's next?

- Proposal will be added to spec as soon as possible
- When spec is next ratified, so is the proposal

Don’t call them ECMAScript 20xx features

- Before stage 4: proposals may be withdrawn.
- Stage 4: proposal will certainly become a part of ECMAScript.
    - But: can’t be sure *when*.

© by Ben Mason

ECMAScript 2016

New features in ES2016

- `Array.prototype.includes` (Domenic Denicola, Rick Waldron)
- Exponentiation Operator (Rick Waldron)

Array.prototype.includes

```
> ['a', 'b', 'c'].includes('a')
true
> ['a', 'b', 'c'].includes('d')
false

> [NaN].includes(NaN)
true
> [NaN].indexOf(NaN)
-1
```

Exponentiation operator

```
// x ** y is same as Math.pow(x, y)

let squared = 3 ** 2; // 9

let num = 3;
num **= 2; // same: num = num ** 2
console.log(num); // 9
```

© by JD Hancock

Proposals for ES2017+

Stage 4 proposals

(Probably included)

`Object.values` `Object.entries`

`Object.entries()` returns an Array of [key,value] pairs:

```
> Object.entries({ one: 1, two: 2 })
[ [ 'one', 1 ], [ 'two', 2 ] ]
```

Symbol-keyed properties are ignored:

```
> Object.entries({ [Symbol()]: 123, k: 'v' });
[ [ 'k', 'v' ] ]
```

Object.values/Object.entries

Setting up a Map:

```
let map = new Map(Object.entries({
    one: 1,
    two: 2,
}));
console.log(JSON.stringify([...map]));
    // [["one",1],["two",2]]
```

Object.values/Object.entries

`Object.values()`:

```
> Object.values({ one: 1, two: 2 })
[ 1, 2 ]
```

Stage 3 proposals

(Maybe included)

SIMD.JS

- SIMD: single instruction, multiple data
- Example: Intel’s SSE (Streaming SIMD Extensions)
- Operands – vectors of ints and floats: `float32x4, uint32x4`
- Operations, e.g.:

```
SIMD.float32x4.abs(v)
SIMD.float32x4.neg(v)
SIMD.float32x4.sqrt(v)
SIMD.float32x4.add(v, w)
SIMD.float32x4.mul(v, w)
SIMD.float32x4.equal(v, w)
```

SIMD.JS

```
var a = SIMD.float32x4(1.0, 2.0, 3.0, 4.0);
var b = SIMD.float32x4(5.0, 6.0, 7.0, 8.0);
var c = SIMD.float32x4.add(a,b);
```

Async Functions

```
// New Promise-based browser API `fetch`
function fetchJsonViaPromises(url) {
    return fetch(url)
    .then(request => request.text())
    .then(text => {
        return JSON.parse(text);
    })
    .catch(error => {
        console.log(`ERROR: ${error.stack}`);
    }); }
async function fetchJsonAsync(url) {
    try {
        let request = await fetch(url);
        let text = await request.text();
        return JSON.parse(text);
    }
    catch (error) {
        console.log(`ERROR: ${error.stack}`);
    } }
```

Async Functions

Variants:

```
// Async function declaration
async function foo() {}

// Async function expression
const foo = async function () {};

// Async arrow function
const foo = async () => {};

// Async method definition (in classes, too)
let obj = { async foo() {} };
```

String padding

```
> '1'.padStart(3, '0')
'001'
> 'x'.padStart(3)
'  x'

> '1'.padEnd(3, '0')
'100'
> 'x'.padEnd(3)
'x  '
```

String padding

Use cases:

- Displaying tabular data in a monospaced font.
- Adding a count or an ID to a file name or a URL: `'file 001.txt'`
- Aligning console output: `'Test 001: ✓'`
- Printing hexadecimal or binary numbers that have a fixed number of digits: `'0x00FF'`

Trailing commas in function parameter lists and calls

Trailing commas are legal in object literals:

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

Trailing commas in function parameter lists and calls

Trailing commas are legal in Array literals:

```
let arr = [
    'red',
    'green',
    'blue',
];
console.log(arr.length); // 3
```

Trailing commas in function parameter lists and calls

Two benefits:

- Rearranging items is simpler (no commas to add or remove)
- Version control systems track what really changed. Versus:

```
// From:
[
    'foo'
]

// To:
[
    'foo',
    'bar'
]
```

Trailing commas in function parameter lists and calls

The proposal:

```
function foo(
    param1,
    param2,
) {}

foo(
    'abc',
    'def',
);
```

Object. getOwnPropertyDescriptors

```
const obj = {
    [Symbol('foo')]: 123,
    get bar() { return 'abc' },
};
console.log(Object.getOwnPropertyDescriptors(obj));

// Output:
// { [Symbol('foo')]:
//     { value: 123,
//       writable: true,
//       enumerable: true,
//       configurable: true },
//   bar:
//     { get: [Function: bar],
//       set: undefined,
//       enumerable: true,
//       configurable: true } }
```

Object.getOwnPropertyDescriptors

```
// Copying properties
const target = {};
Object.defineProperties(target,
    Object.getOwnPropertyDescriptors(source));

// Cloning objects
const clone =
Object.create(Object.getPrototypeOf(obj),
    Object.getOwnPropertyDescriptors(obj));
```

`Function.prototype.toString` revision

- Improved spec of `toString()` for functions.

Thanks!

Twitter: @rauschma

Book by Axel (free online):
"Exploring ES6"

Blog posts: ES2016, ES2017

These slides:
speakerdeck.com/rauschma