ECMAScript: latest and upcoming features

Axel Rauschmayer
@rauschma

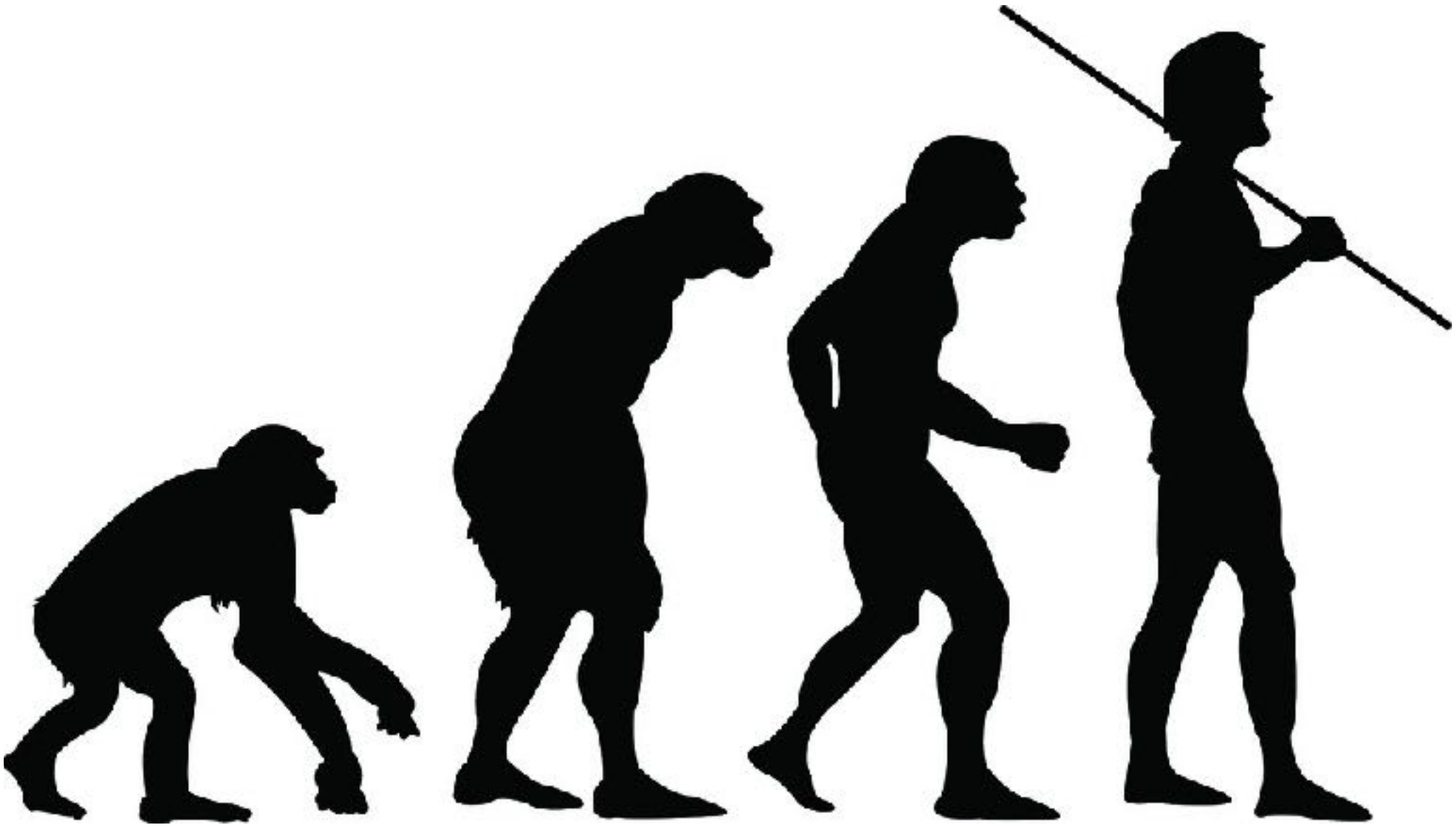
HolyJS
Moscow, 11 December 2016

Slides: speakerdeck.com/rauschma

Overview

- JavaScript:
    - What's new?
    - What does the future bring?
- Background:
    - TC39: who designs JavaScript?
    - TC39 process: how is JavaScript evolved?

You'll see lots of code!



Evolving JavaScript: TC39 and the TC39 process

JavaScript vs. ECMAScript

- Meaning:
    - JavaScript: the language
    - ECMAScript: the standard for the language
        - ECMAScript 6 etc.: language versions
- Ecma: standards organisation hosting ECMAScript

TC39

- **Ecma Technical Committee 39** (TC39): the committee evolving JavaScript

- Members – strictly speaking: 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"
  ],
...
```

Timeline of ECMAScript

- ECMAScript 1 (June 1997): first version
- ECMAScript 2 (June 1998): keep in sync with ISO standard
- **ECMAScript 3 (December 1999):** many core features – "[…] regular expressions, better string handling, new control statements [do-while, switch], try/catch exception handling, […]"
- ECMAScript 4 (abandoned in July 2008)
- ECMAScript 5 (December 2009): minor improvements (standard library and strict mode)
- ECMAScript 5.1 (June 2011): keep in sync with ISO standard
- **ECMAScript 6 (June 2015):** many new features

The TC39 process

Problems with infrequent, large releases (such as ES6):

- Features that are ready sooner have to wait.
- Features that are not ready are under pressure to get finished.
    - Next release would be a long time away.
    - They may delay the release.

The TC39 process

New TC39 process:

- Manage features individually (vs. one monolithic release).
- Per feature: proposal that goes through maturity stages, numbered 0 (strawman) – 4 (finished).
    - Introduce features gradually.
- Once a year, there is a new ECMAScript version.
    - Only features that are ready (=stage 4) 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?

- Actual proposal of a feature
- **TC39 is willing to help with designing the feature**

What's required?

- Identify champion(s), one of them a TC39 member
- Spec: prose, examples, API, semantics and algorithms
- Implementation: polyfills and demos

What's next?

- Major changes are still expected

Stage 2: draft

What is it?

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

What's required?

- Formal description of syntax and semantics (gaps are OK)
- **Two experimental implementations** (incl. one 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?

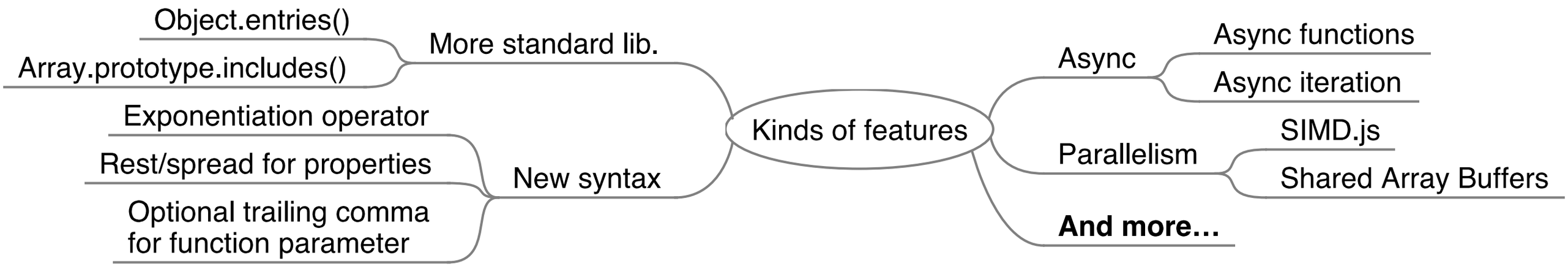
- Proposal will be added to spec as soon as possible
- When spec is next ratified, the proposal becomes a standard

Think in proposals & stages, not in ES versions

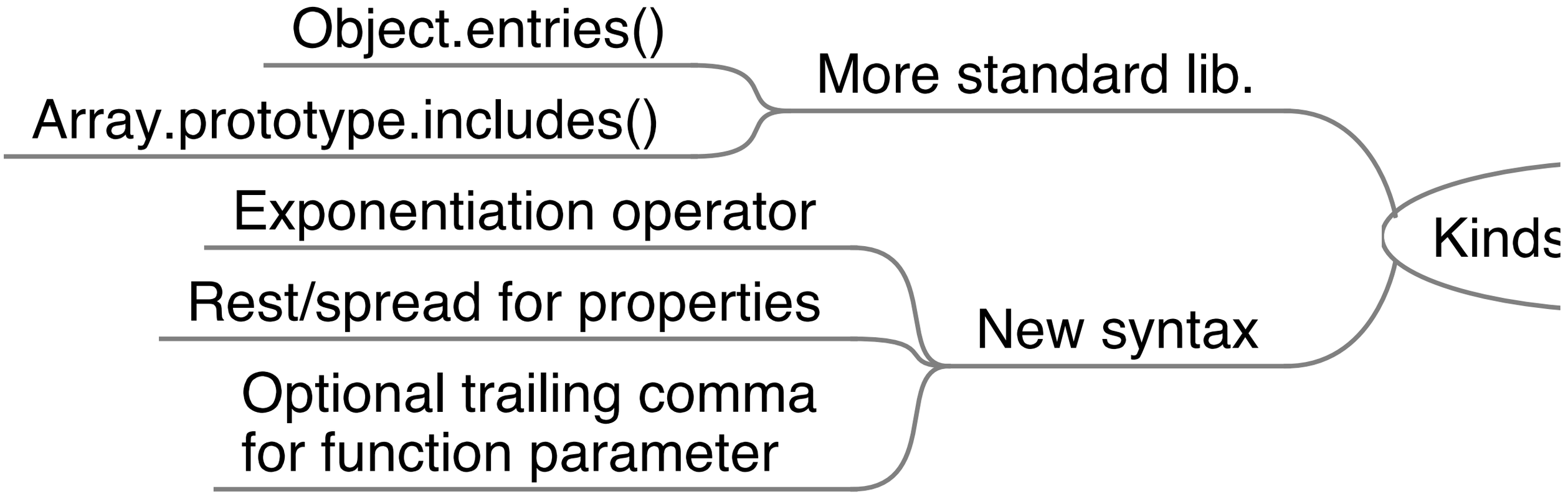
- Before stage 4: proposals may be withdrawn.
    - `Object.observe()`: withdrawn at stage 2
- Stage 4: proposal will certainly become a part of ECMAScript.
    - But: don't know when exactly.

Tip: Ignore features before stage 3.

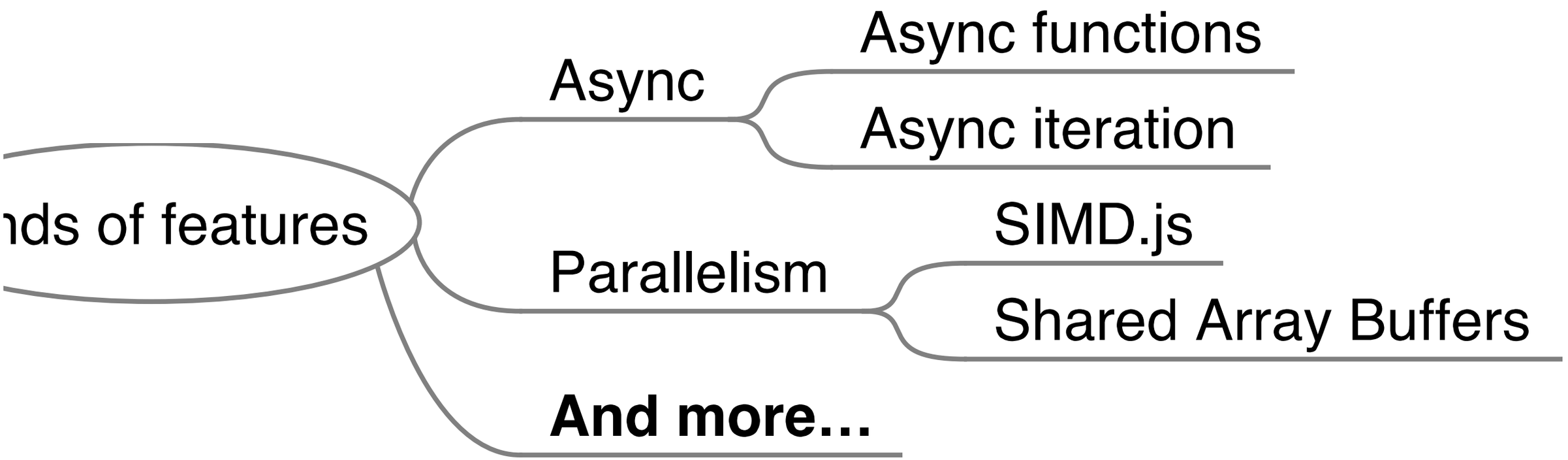
Kinds of features



Kinds of features



Kinds of features



ECMAScript 2016

Standardised in June 2016

New features in ES2016

ES2016 has only two new features:

- `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) >= 0
false
```

Exponentiation operator

```
// x ** y is same as Math.pow(x, y)
const squared = 3 ** 2; // 9

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

© by JD Hancock

ECMAScript 2017+

Features accepted for ES2017

Async Functions

```
function fetchJsonViaPromises(url) {
    return fetch(url) // browser API, returns Promise
    .then(request => request.text()) // returns Promise
    .then(text => {
        return JSON.parse(text);
    })
    .catch(error => {
        console.log(`ERROR: ${error.stack}`);
    }); }

async function fetchJsonAsync(url) {
    try {
        const request = await fetch(url);
        const 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)
const obj = { async foo() {} };
```

Fulfilling the Promise of an async function

```
async function asyncFunc() {
    return 123;
}

asyncFunc()
.then(x => console.log(x));
    // 123
```

Rejecting the Promise of an async function

```
async function asyncFunc() {
    throw new Error('Problem!');
}

asyncFunc()
.catch(err => console.log(err));
    // Error: Problem!
```

Handling rejected Promises inside async functions

```
async function asyncFunc() {
    try {
        await otherAsyncFunc();
    } catch (err) {
        console.error(err);
    }
}

// Equivalent to:
function asyncFunc() {
    return otherAsyncFunc()
    .catch(err => {
        console.error(err);
    });
}
```

Async functions and Node.js

Node.js 7.2.1:

```
$ node --harmony_async_await

> async function f() { return 123 }
> f().then(x => console.log(x))
123
```

`Object.entries()`

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

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

`Object.entries()`

Easier to iterate over properties:

```
const obj = { one: 1, two: 2 };
for (const [k, v] of Object.entries(obj)) {
    console.log(k, v);
}
// Output
// "one" 1
// "two" 2
```

`Object.values()`

Complements `Object.keys()` and `Object.entries()`:

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

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 to a file name: `'file 001.txt'`
- Aligning console output: `'Test 001: ✓'`
- Printing hexadecimal or binary numbers that have a fixed number of digits: `'0x00FF'`

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()`

`Object.assign()` is limited: can't copy getters and setters, etc.

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

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

Trailing commas in function parameter lists and calls

Trailing commas are legal in object and Array literals:

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

const 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 can track what really changed. Versus:

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

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

Trailing commas in function parameter definitions and calls

The proposal:

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

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

Stage 3 proposals

(Maybe included in ES2017)

import()

So far, ES6 modules can only be loaded *statically* (in a fixed manner, specified at compile time).

Load modules dynamically:

```
import('./dir/someModule.js')
.then(someModule => someModule.foo());
```

An operator, but used like a function.

import()

Use cases:

- Code splitting: load parts of your program on demand.
- Conditional loading of modules:
  `if (cond) { import(···).then(···) }`
- Computed module specifiers:
  `import('module'+count).then(···)`

Rest operator for properties (destructuring)

```
const obj = {foo: 1, bar: 2, baz: 3};
const {foo, ...rest} = obj;
    // Same as:
    // const foo = 1;
    // const rest = {bar: 2, baz: 3};

function f({param1, param2, ...rest}) { // rest
    console.log('All parameters: ',
        {param1, param2, ...rest}); // spread
    return param1 + param2;
}
```

Spread operator for properties (object literals)

```
> const obj = {foo: 1, bar: 2};
> {...obj, baz: 3}
{ foo: 1, bar: 2, baz: 3 }
```

Spread properties use cases

```
// Cloning objects
const clone1 = {...obj};

// Merging objects
const merged = {...obj1, ...obj2};

// Filling in defaults
const data = {...DEFAULTS, ...userData};

// Non-destructively updating property `foo`
const obj = {foo: 'a', bar: 'b'};
const obj2 = {...obj, foo: 1};
    // {foo: 1, bar: 'b'}
```

global

Accessing the global object:

- Browsers (main thread): `window`
- Browsers (main thread & workers): `self`
- Node.js: `global`

```
// In browsers
console.log(global === window); // true
```

History of concurrency in JavaScript

- Single main thread + asynchronicity via callbacks
- Web Workers
    - Originally: copy and send strings
    - Structured cloning: copy and send structured data
    - Transferables: move and send structured data
- Failed experiment: PJS / River Trail
    - High-level support for data parallelism (`map()`, `filter()`, `reduce()`)

Shared Array Buffers

New – Shared Array Buffers:

- A primitive building block for higher-level concurrency abstractions
- Share data between workers
- Consequence: “atomic” operations for synchronising between threads

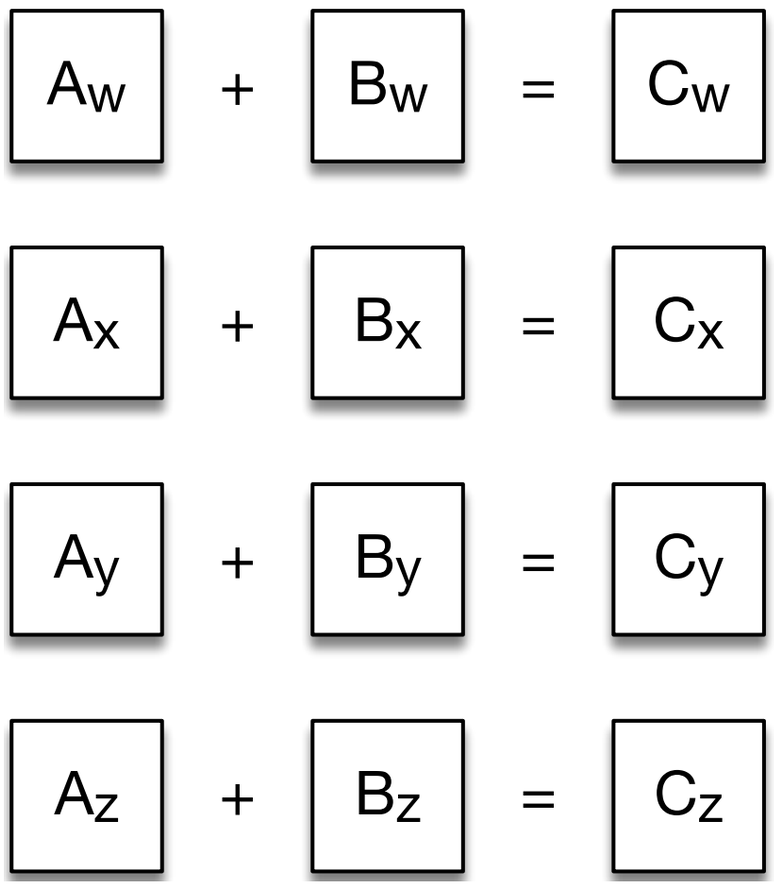
Shared Array Buffers

```
const sh = new SharedArrayBuffer(
    Uint32Array.BYTES_PER_ELEMENT * 10);
const ta = new Uint32Array(sh);
worker.postMessage(ta); // share with worker

// Writer
Atomics.store(ta, 0, 123);

// Reader
while (Atomics.load(ta, 0) !== 123)
    ;
```

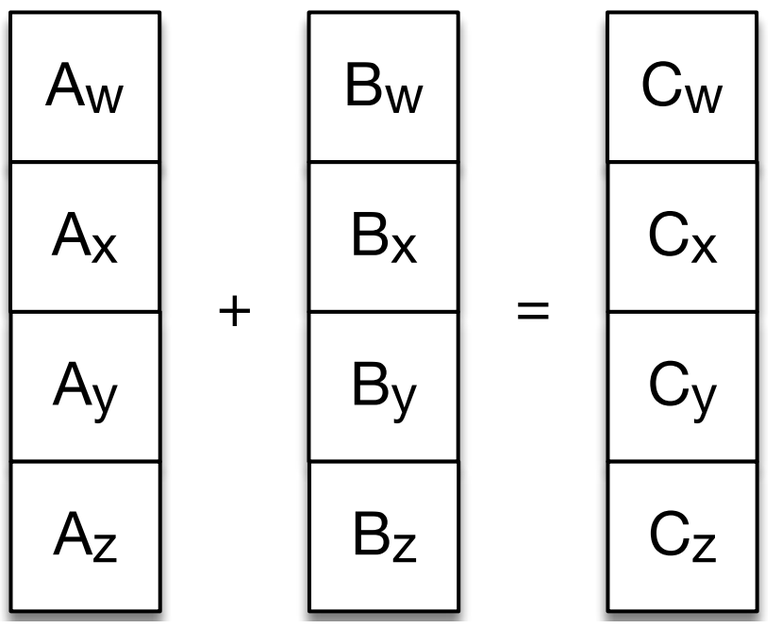
Shared Array Buffers

```
Atomics.load( typedArray, index )
Atomics.store( typedArray, index, value )
Atomics.exchange( typedArray, index, value )
Atomics.compareExchange( typedArray, index,
    expectedValue, replacementValue )

Atomics.add( typedArray, index, value )
Atomics.sub( typedArray, index, value )

Atomics.and( typedArray, index, value )
Atomics.or( typedArray, index, value )
Atomics.xor( typedArray, index, value )

Atomics.wait( typedArray, index, value, timeout )
Atomics.wake( typedArray, index, count )
Atomics.isLockFree( size )
```

SIMD: single instruction, multiple data



SISD



SIMD

SIMD.JS

- SIMD.JS: SIMD support in JavaScript
- Example target: Intel's SSE (Streaming SIMD Extensions)
- Speed-up: up to 4 times (sometimes even more)

SIMD.JS

- Operands – vectors of booleans, ints, floats: `Bool8x16, 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

```
const a = SIMD.Float32x4(1.0, 2.0, 3.0, 4.0);
const b = SIMD.Float32x4(5.0, 6.0, 7.0, 8.0);
const c = SIMD.Float32x4.add(a,b);
```

SIMD.JS: data types

Always 128 bits:

- Booleans: `SIMD.Bool8x16, SIMD.Bool16x8, SIMD.Bool32x4, SIMD.Bool64x2`
- Signed integers: `SIMD.Int8x16, SIMD.Int16x8, SIMD.Int32x4`
- Unsigned integers: `SIMD.Uint8x16, SIMD.Uint16x8, SIMD.Uint32x4`
- Floating point numbers: `SIMD.Float32x4, SIMD.Float64x2`

`Function.prototype.toString` revision

Improved spec of `toString()` for functions:

- Return source code whenever possible
    - Previously: optional
- Otherwise: standardised placeholder
    - Previously: must cause SyntaxError (hard to guarantee!)

Template literal revision

Syntax rules after backslash:

- \u starts a Unicode escape, which must look like \u{1F4A4} or \u004B
- \x starts a hex escape, which must look like \x4B.
- \ plus digit starts an octal escape (such as \141). Octal escapes are forbidden in template literals and strict mode string literals.

Illegal:

```
latex`\unicode`
windowsPath`C:\uuu\xxx\111`
```

Template literal revision

```
function tagFunc(tmplObj, substs) {
    return {
        Cooked: tmplObj,
        Raw: tmplObj.raw,
    };
}

tagFunc`\u{4B}`;
  // { Cooked: [ 'K' ], Raw: [ '\\u{4B}' ] }
```

Template literal revision

Solution:

```
tagFunc`\uu ${1} \xx`
    // { Cooked: [ undefined, undefined ],
    //   Raw:    [ '\\uu ', ' \\xx' ] }
```

Asynchronous iteration

```
// Synchronous iteration:
for (const l of readLinesSync(fileName)) {
    console.log(l);
}
```

Problem: `readLinesSync()` must be synchronous.

Asynchronous iteration

```
// Asynchronous iteration:
for await (const l of readLinesAsync(fileName)) {
    console.log(l);
}
```

Works inside:

- Async functions
- Async generators (new, part of proposal)

Asynchronous iteration

```
async function f() {
    const aI = createAsyncIterable(['a', 'b']);
    for await (const x of aI) {
        console.log(x);
    }
}
// Output:
// a
// b
```

Asynchronous generators

```
async function* createAsyncIterable(syncIterable) {
    for (const elem of syncIterable) {
        yield elem;
    }
}
```

Asynchronous generators

```
async function* id(asyncIterable) {
    for await (const elem of asyncIterable) {
        yield elem;
    }
}
```

Asynchronous iteration

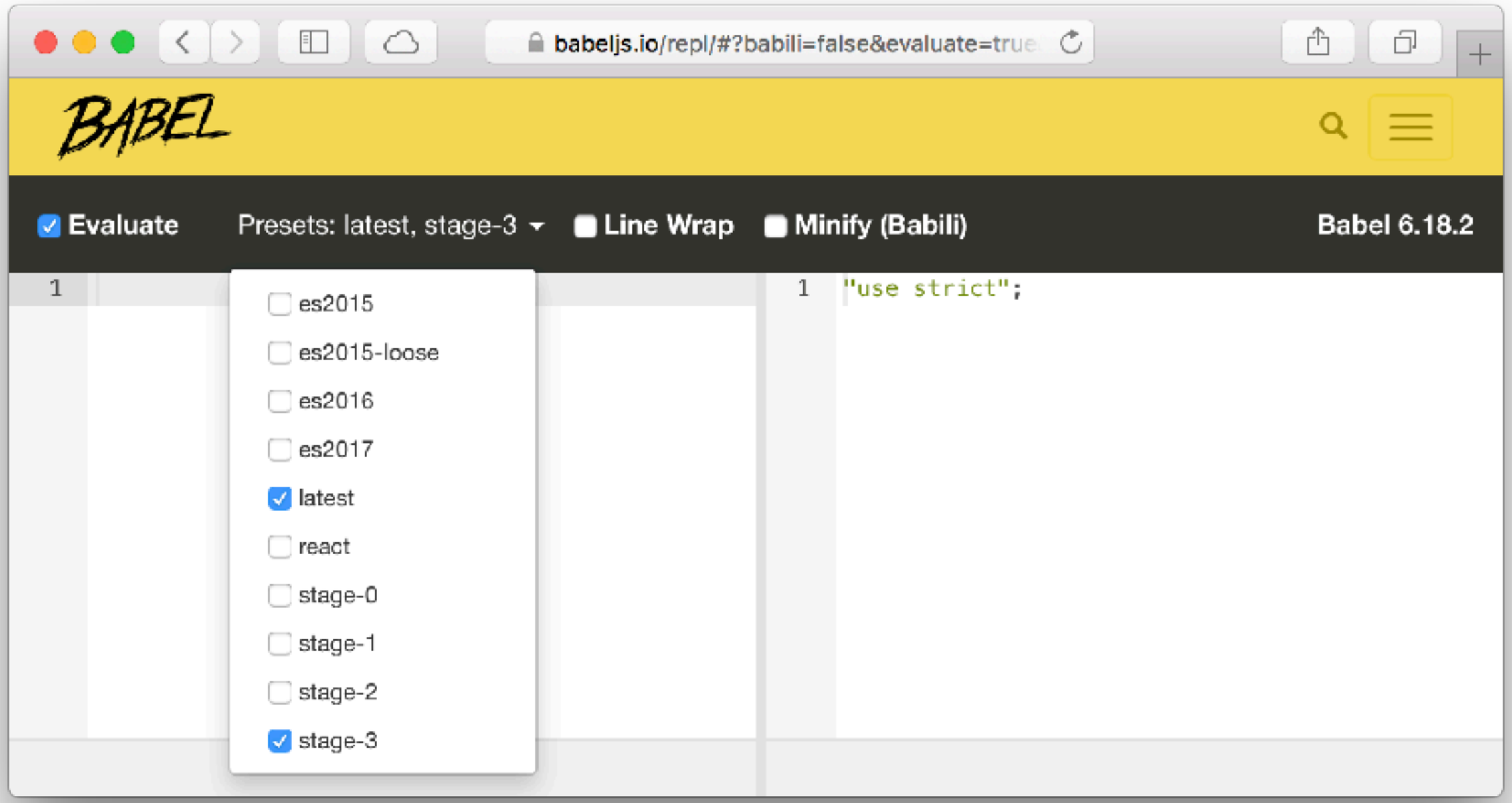
```
interface AsyncIterable {
    [Symbol.asyncIterator]() : AsyncIterator;
}
interface AsyncIterator {
    next() : Promise<IteratorResult>;
}
interface IteratorResult {
    value: any;
    done: boolean;
}
```

Asynchronous iteration

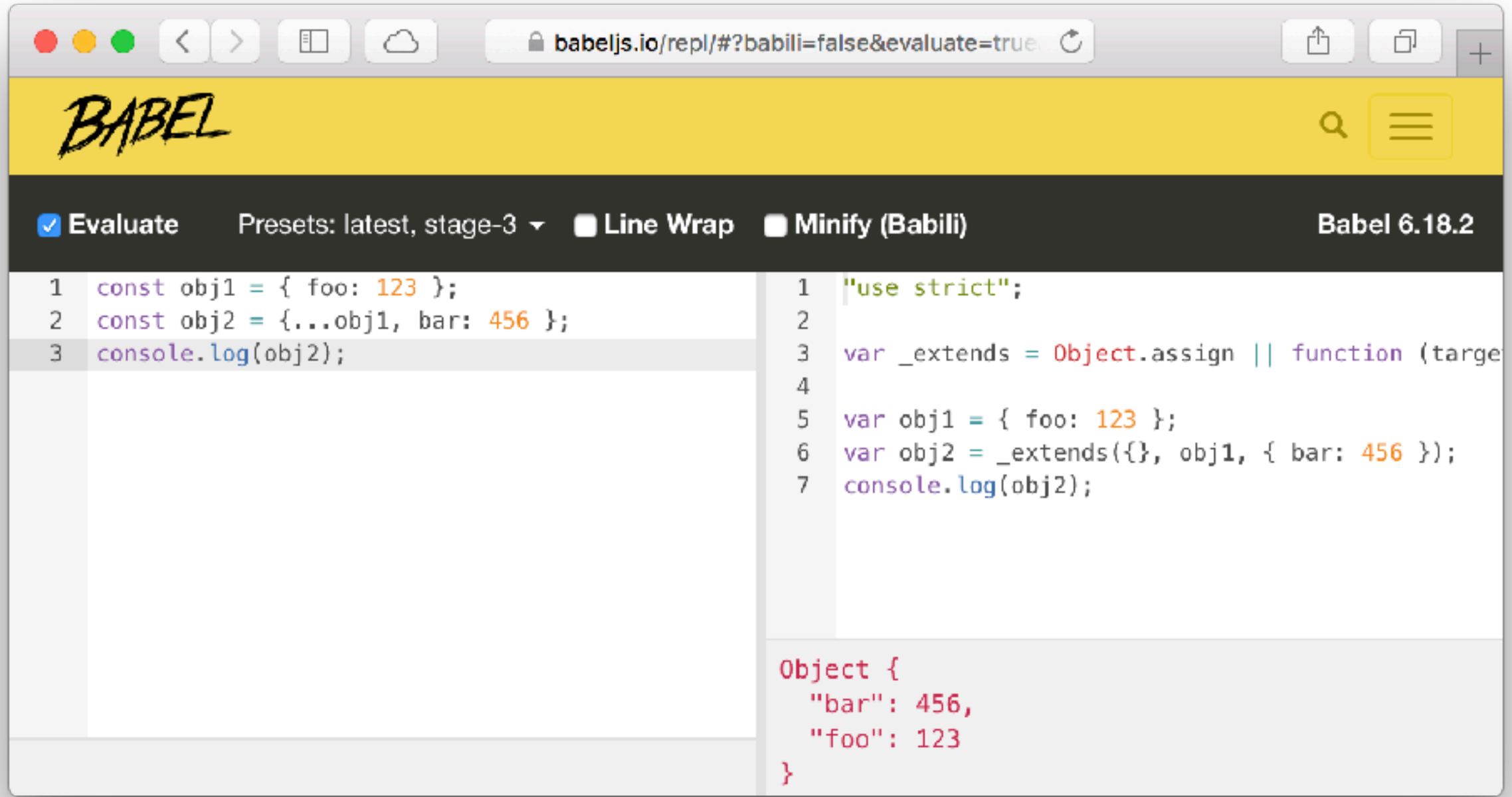
I’m skeptical:

- Relatively complex iteration protocol
- New language complexity: async generators (function declaration, function expression, method definition)
- Communicating sequential processes (CSP) may be a simpler solution: github.com/ubolonton/js-csp

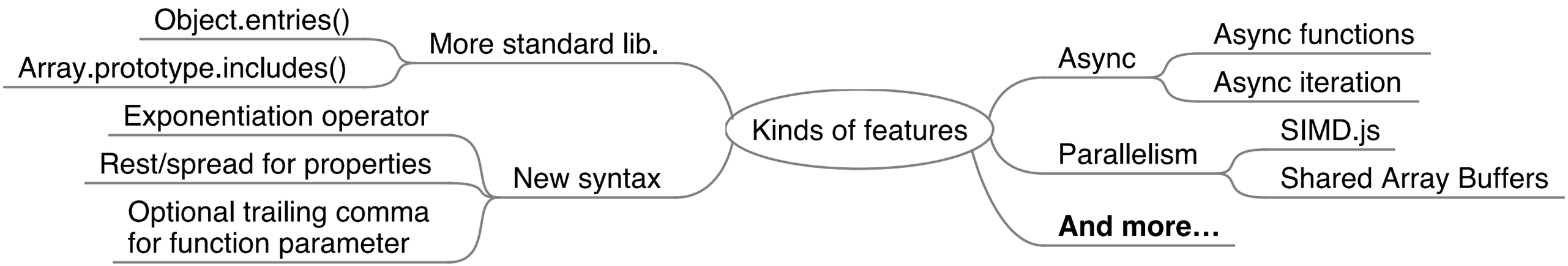
Trying out new ES features



Trying out new ES features



Summary


Kinds of features
More standard lib.
Object.entries()
Array.prototype.includes()
New syntax
Exponentiation operator
Rest/spread for properties
Optional trailing comma for function parameter
Async
Async functions
Async iteration
Parallelism
SIMD.js
Shared Array Buffers
And more…

Thanks!

Twitter: @rauschma

Books by Axel (free online):
exploringjs.com

Blog post: ES2017

These slides:
speakerdeck.com/rauschma