ECMAScript 2017: what's new for JavaScript?

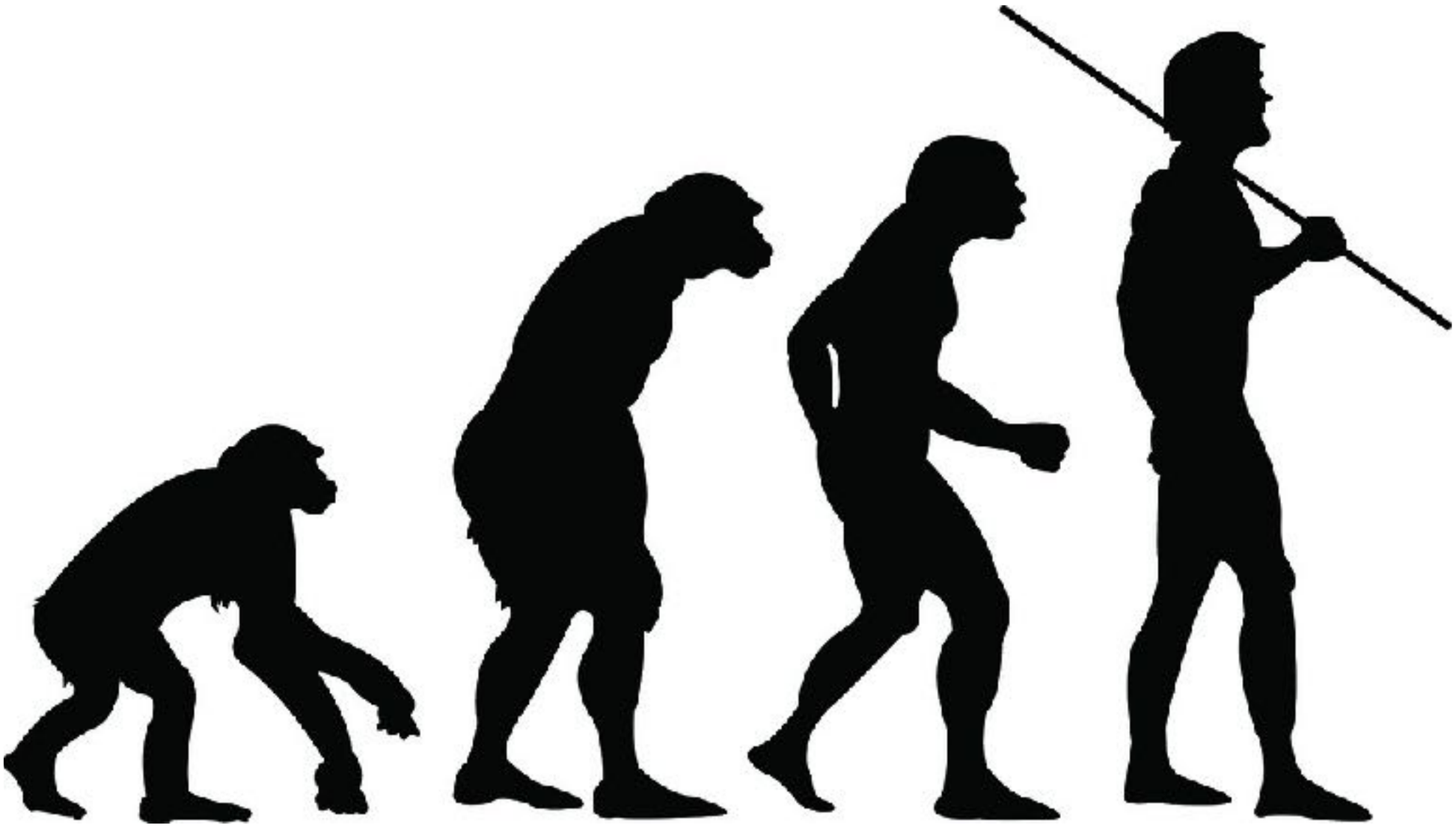
Axel Rauschmayer
@rauschma

Techorama
Antwerpen, 24 May 2017

Slides: speakerdeck.com/rauschma

Overview

- How is JavaScript being evolved?
- What are the features of ECMAScript 2017?
- What's in store for JS after ES2017?



Evolving JavaScript: TC39 and the TC39 process

JavaScript vs. ECMAScript

- **JavaScript:** the language
- **ECMAScript:** the standard for the language
    - Why a different name? Trademark for “JavaScript”!
    - Language versions: ECMAScript 6, …
- **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

TC39 members

- Adobe
- **Apple**
- Bloomberg
- Bocoup
- Dojo Foundation
- Facebook
- GoDaddy.com
- **Google**
- IBM
- Imperial College London
- Indiana University
- Inria
- Intel
- JS Foundation
- Meteor Development Group
- **Microsoft**
- **Mozilla Foundation**
- PayPal (ex eBay)
- Tilde Inc.
- Twitter
- Yahoo!

Source: http://ecma-international.org/memento/TC39-RF-TG%20-%20members.htm

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

- Creation of JavaScript: May 1995
- 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.

Additional problem: standardization before implementation

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 to community
    - Must be implemented early
- Once a year, there is a new ECMAScript version.
    - Only features that are ready (=stage 4) are added.

Stage 0: strawman

- **What?** First sketch
- **Who?** Submitted by TC39 member or registered TC39 contributor
- **Required?** Review at TC39 meeting

Stage 1: proposal

- **What?** Actual proposal. TC39 is willing to help.
- **Who?** Identify champion(s), one of them a TC39 member
- **Spec?** Informal (prose, examples, API, semantics, algorithms, …)
- **Implementations?** Polyfills and demos
- **Maturity?** Major changes still expected

Stage 2: draft

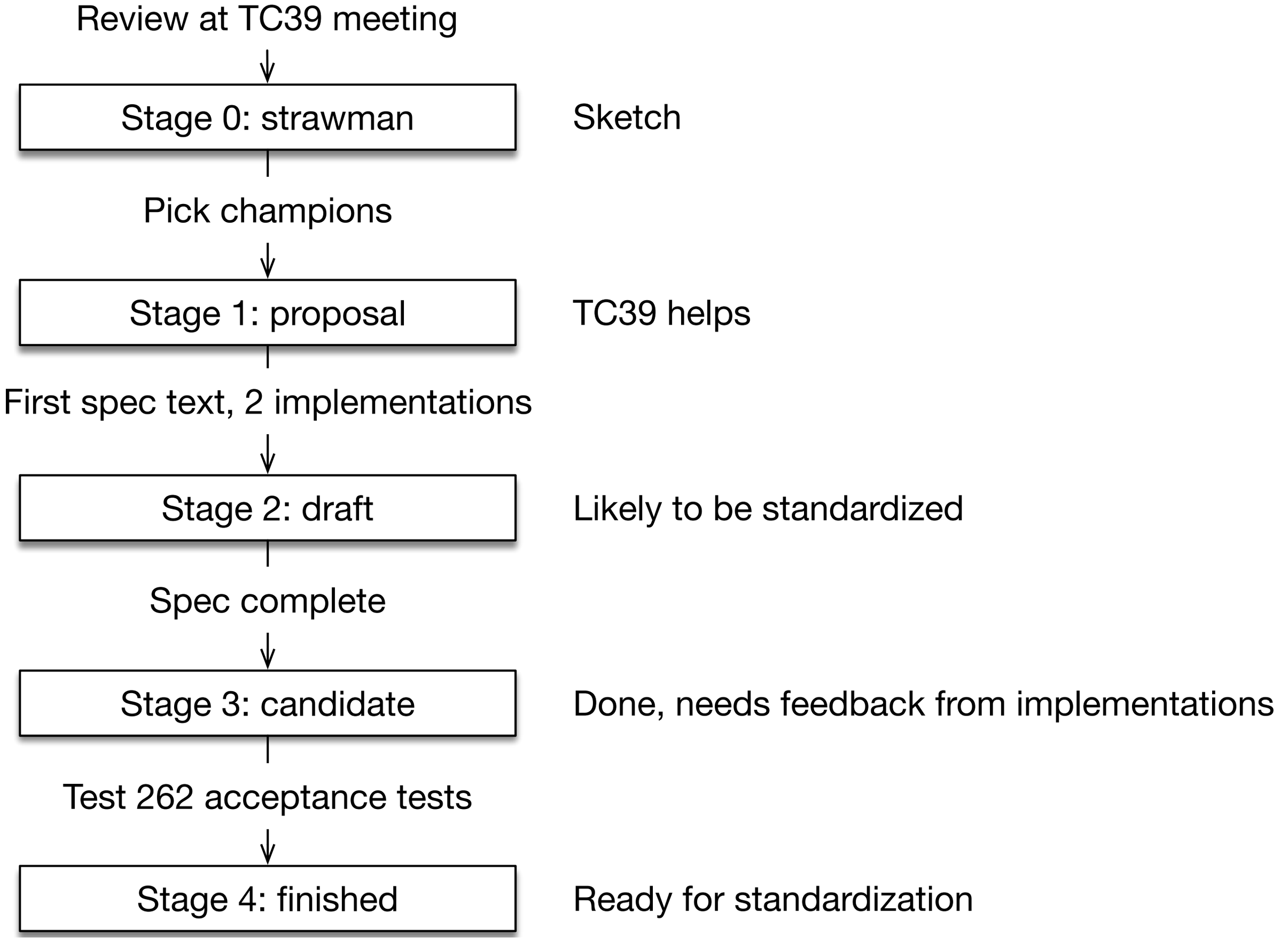
- **What?** Draft of spec text. Likely to be standardized.
- **Spec?** Formal description of syntax and semantics (gaps are OK)
- **Implementations?** Two experimental implementations (incl. one transpiler)
    - Continually kept in sync with spec
- **Maturity?** Incremental changes

Stage 3: candidate

- **What?** Proposal is finished, needs feedback from implementations
- **Spec?** Complete
- **Maturity?** Changes only in response to critical issues

Stage 4: finished

- **What?** Ready for standardization
- **Test 262 acceptance tests**
- **Implementations?**
  - Two implementations: spec-compliant, passing tests
  - Significant practical experience
- **Next?** Added to ECMAScript as soon as possible


Review at TC39 meeting
Stage 0: strawman
Sketch
Pick champions
Stage 1: proposal
TC39 helps
First spec text, 2 implementations
Stage 2: draft
Likely to be standardized
Spec complete
Stage 3: candidate
Done, needs feedback from implementations
Test 262 acceptance tests
Stage 4: finished
Ready for standardization

Think in proposals & stages, not in ES versions

Stages matter, not ECMAScript versions:

- Stage 4: will be in ECMAScript
    - No guarantees w.r.t. ES version
    - Mature, available in more and more engines

Tip: Ignore proposals before stage 3.

- Stage 3: proposal basically finished
- Before stage 4: proposals may be withdrawn.
    - `Object.observe()`: withdrawn at stage 2
    - SIMD.js: withdrawn at stage 3

ECMAScript 2017

Features of ES2017

Major new features:

- Async Functions (Brian Terlson)
- Shared memory and atomics (Lars T. Hansen)

Minor new features:

- Object.values/Object.entries (Jordan Harband)
- String padding (Jordan Harband, Rick Waldron)
- Object.getOwnPropertyDescriptors() (Jordan Harband, Andrea Giammarchi)
- Trailing commas in function parameter lists and calls (Jeff Morrison)

Async Functions

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

async function fetchJsonAsync(url) {
    try {
        const request = await fetch(url); // fulfillment
        const text = await request.text(); // fulfillment
        return JSON.parse(text);
    }
    catch (error) { // rejection
        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!
```

History of concurrency in JavaScript

- Single main thread + asynchronicity via callbacks
- **Web Workers**: heavyweight processes
    - Communication (data is never shared!):
        1. Originally: copy and send strings
        2. Structured cloning: copy and send structured data
        3. 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
    - Design principle of Extensible Web Manifesto
- Share data between workers
- Enable compilation of multi-threaded C++ code to JavaScript (later: WebAssembly)

Creating and sending a Shared Array Buffer

```
//----- main.js -----

const worker = new Worker('worker.js');

// To be shared
const sharedBuffer = new SharedArrayBuffer(
    10 * Int32Array.BYTES_PER_ELEMENT); // 10 elts

// Share sharedBuffer with the worker
worker.postMessage({sharedBuffer}); // clone

// Local only
const sharedArray = new Int32Array(sharedBuffer);
```

Receiving a Shared Array Buffer

```
//----- worker.js -----

self.addEventListener('message', event => {
  const {sharedBuffer} = event.data;
  const sharedArray = new Int32Array(sharedBuffer);
  // ...
});
```

Problem: compilers may rearrange reads

```
// Original code
while (sharedArray[0] === 1) ;
```

```
// Rearranged by compiler:
const tmp = sharedArray[0];
while (tmp === 1) ; // runs never or forever
```

Problem: writes may be reordered

```
// main.js
sharedArray[1] = 11;
sharedArray[2] = 22;

// worker.js
while (sharedArray[2] !== 22) ;
console.log(sharedArray[1]); // 0 or 11
```

Solution: Atomics

- Operations that are non-interruptible (atomic) – think transactions in DBs
- Order of reads and writes is fixed
- No reads or writes are eliminated
- Used to synchronize non-atomic reads and writes

Using Atomics

```
// main.js
Atomics.store(sharedArray, 1, 11);
Atomics.store(sharedArray, 2, 22);
```

```
// worker.js
while (Atomics.load(sharedArray, 2) !== 22) ;
console.log(Atomics.load(sharedArray, 1)); // 11
```

Atomics

Loading and storing:

- `Atomics.load(ta : TypedArray<T>, index) : T`
- `Atomics.store(ta : TypedArray<T>, index, value : T) : T`
- `Atomics.exchange(ta : TypedArray<T>, index, value : T) : T`
- `Atomics.compareExchange(ta : TypedArray<T>, index, expectedValue, replacementValue) : T`

Waiting and waking:

- `Atomics.wait(ta: Int32Array, index, value, timeout=Number.POSITIVE_INFINITY) : ('not-equal' | 'ok' | 'timed-out')`
- `Atomics.wake(ta : Int32Array, index, count)`

Atomics

Simple modifications of Typed Array elements (`ta[index] op= value`):

- `Atomics.add(ta : TypedArray<T>, index, value) : T`
- `Atomics.sub(ta : TypedArray<T>, index, value) : T`
- `Atomics.and(ta : TypedArray<T>, index, value) : T`
- `Atomics.or(ta : TypedArray<T>, index, value) : T`
- `Atomics.xor(ta : TypedArray<T>, index, value) : T`

`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 object literals and Array literals

Trailing commas are legal in object and Array literals:

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

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

Trailing commas in object literals and Array literals

Two benefits:

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

```
// Before:
[
    'foo'
]

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

Proposal: Trailing commas in function parameter definitions and calls

```
// Function definition
function foo(
    param1,
    param2,
) {}

// Function call
foo(
    'abc',
    'def',
);
```

© by JD Hancock

After ES2017

Stage 4 features

- Lifting Template Literal Restriction (Tim Disney)

Stage 3 features

- import() (Domenic Denicola)
- Rest/Spread Properties (Sebastian Markbage)
- global (Jordan Harband)
- New regular expression features
- Asynchronous Iteration (Domenic Denicola)
- Function.prototype.toString revision (Michael Ficarra)
- ~~SIMD.JS – SIMD APIs (John McCutchan, Peter Jensen, Dan Gohman, Daniel Ehrenberg)~~

import()

ES6 – load modules *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(···)`

Spread operator for properties (object literals)

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

Use cases: spreading properties

```
// 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'}
```

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;
}
```

global

Accessing the global object:

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

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

New regular expression features

- Named capture groups (Daniel Ehrenberg, Brian Terlson)
- Lookbehind assertions (Daniel Ehrenberg)
- Unicode property escapes (Brian Terlson, Daniel Ehrenberg, Mathias Bynens)
- `s` (`dotAll`) flag (Mathias Bynens, Brian Terlson)

RegExp named capture groups

Numbered capture groups:

```
const RE_DATE = /([0-9]{4})-([0-9]{2})-
([0-9]{2})/;

const matchObj = RE_DATE.exec('1999-12-31');
const year = matchObj[1]; // 1999
```

Named capture groups:

```
const RE_DATE = /(?<year>[0-9]{4})-(?
<month>[0-9]{2})-(?<day>[0-9]{2})/;

const matchObj = RE_DATE.exec('1999-12-31');
const year = matchObj.groups.year; // 1999
```

RegExp lookbehind assertions

Positive lookbehind assertion:

```
const RE_$_PREFIX = /(?<=\$)foo/g;
'$foo %foo foo'.replace(RE_$_PREFIX, 'bar');
    // '$bar %foo foo'
```

Without a lookbehind assertion:

```
const RE_$_PREFIX = /(\$)foo/g;
'$foo %foo foo'.replace(RE_$_PREFIX, '$1bar');
    // '$bar %foo foo'
```

RegExp lookbehind assertions

Negative lookbehind assertion:

```
const RE_NO_$_PREFIX = /(?<!\$)foo/g;
'$foo %foo foo'.replace(RE_NO_$_PREFIX, 'bar');
// '$foo %bar bar'
```

RegExp Unicode property escapes

Inside regular expressions:

- `\p{PropertyName=PropertyValue}`
- `\p{BinaryPropertyName}`
- Abbreviate `\p{General_Category=Letter}` as `\p{Letter}`

Enabled via `/u` flag!

RegExp Unicode property escapes: example

```
> /^\w+$/.test('äöü') // [a-zA-Z0-9_]
false
```

```
> /^[\p{Alphabetic}\p{Mark}
\p{Decimal_Number}
\p{Connector_Punctuation}
\p{Join_Control}]+$/u.test('äöü')
true
```

`s` (`dotAll`) flag for regular expressions

Old:

```
> /a.b/.test('a\nb')
false
> /a[^]b/.test('a\nb')
true
> /a[\s\S]b/.test('a\nb')
true
```

New (flag `/s`):

```
> /a.b/s.test('a\nb')
true
```

Asynchronous iteration

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

- `readLinesSync()` returns a *synchronous iterable*
- **Problem:** `readLinesSync()` must be synchronous.

Asynchronous iteration

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

- `readLinesAsync()` returns an *asynchronous iterable*
- `for-await-of` works inside:
    - async functions
    - async generators (new, part of proposal)

Asynchronous generators: `yield`

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

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

Asynchronous generators: `await`

```
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;
}
```

`Function.prototype.toString` revision

Improved spec of `toString()` for functions:

- Return source code whenever possible
    - Previously: optional
- Otherwise: standardized 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.

Therefore illegal:

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

Template literal revision

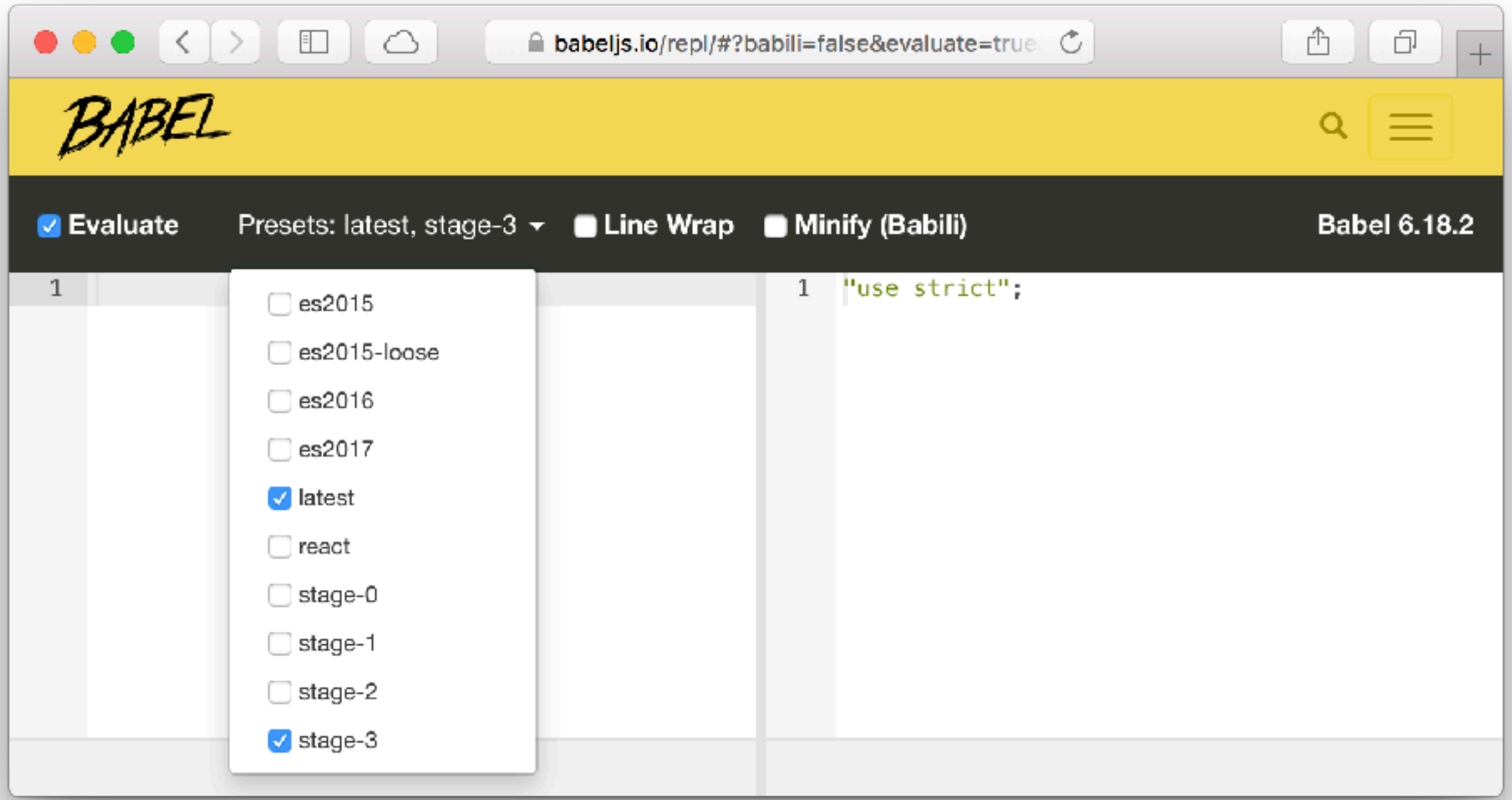
```
function tagFunc(templObj, substs) {
    return {
        Cooked: templObj,
        Raw: templObj.raw,
    };
}

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

Template literal revision

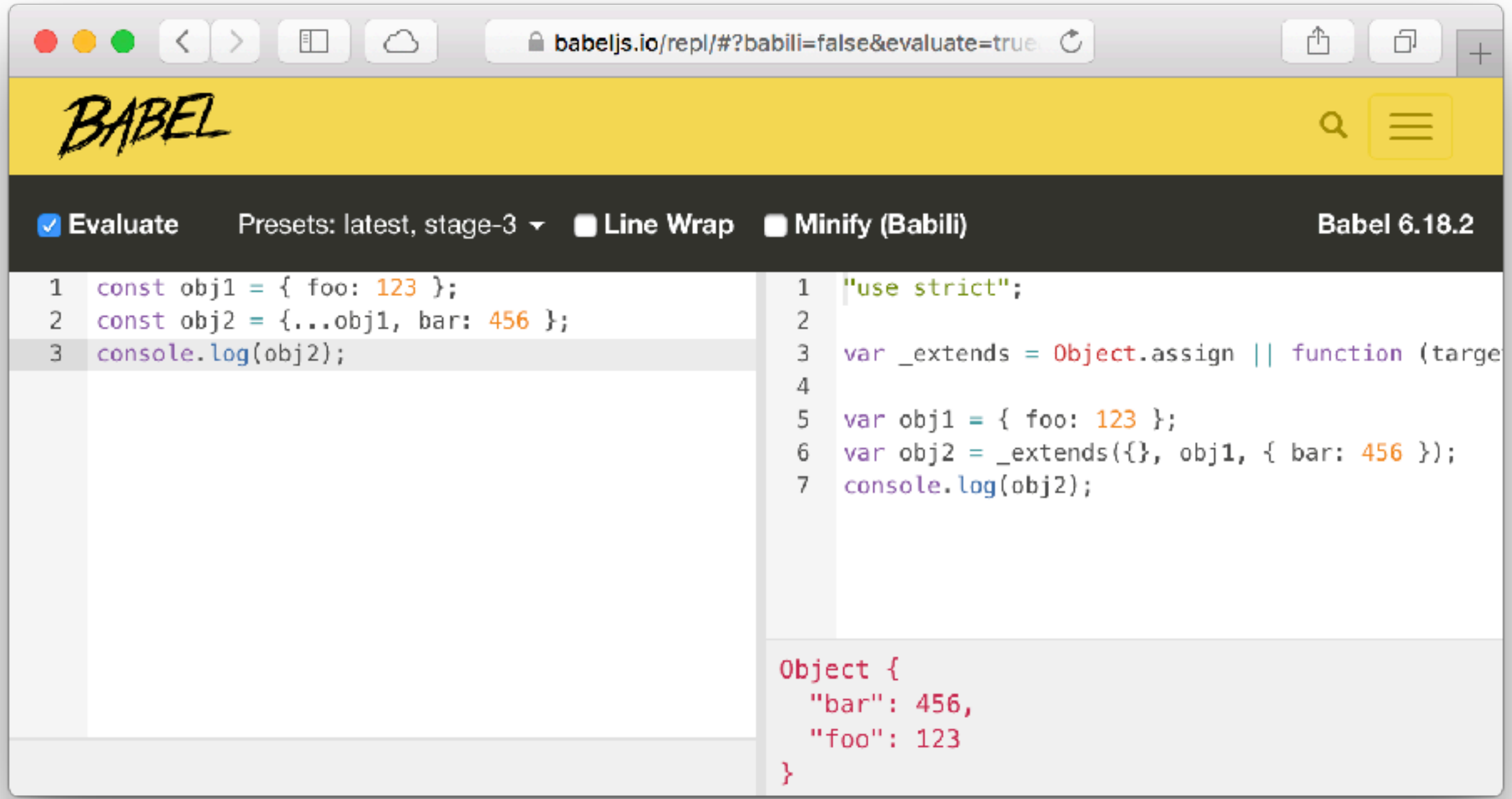
Solution:

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

Trying out new ES features



Trying out new ES features



Thanks!

Twitter: @rauschma

Books by Axel (free online):
ExploringJS.com

Upcoming JS features:
2ality.com/2017/02/ecmascript-2018.html

These slides:
speakerdeck.com/rauschma