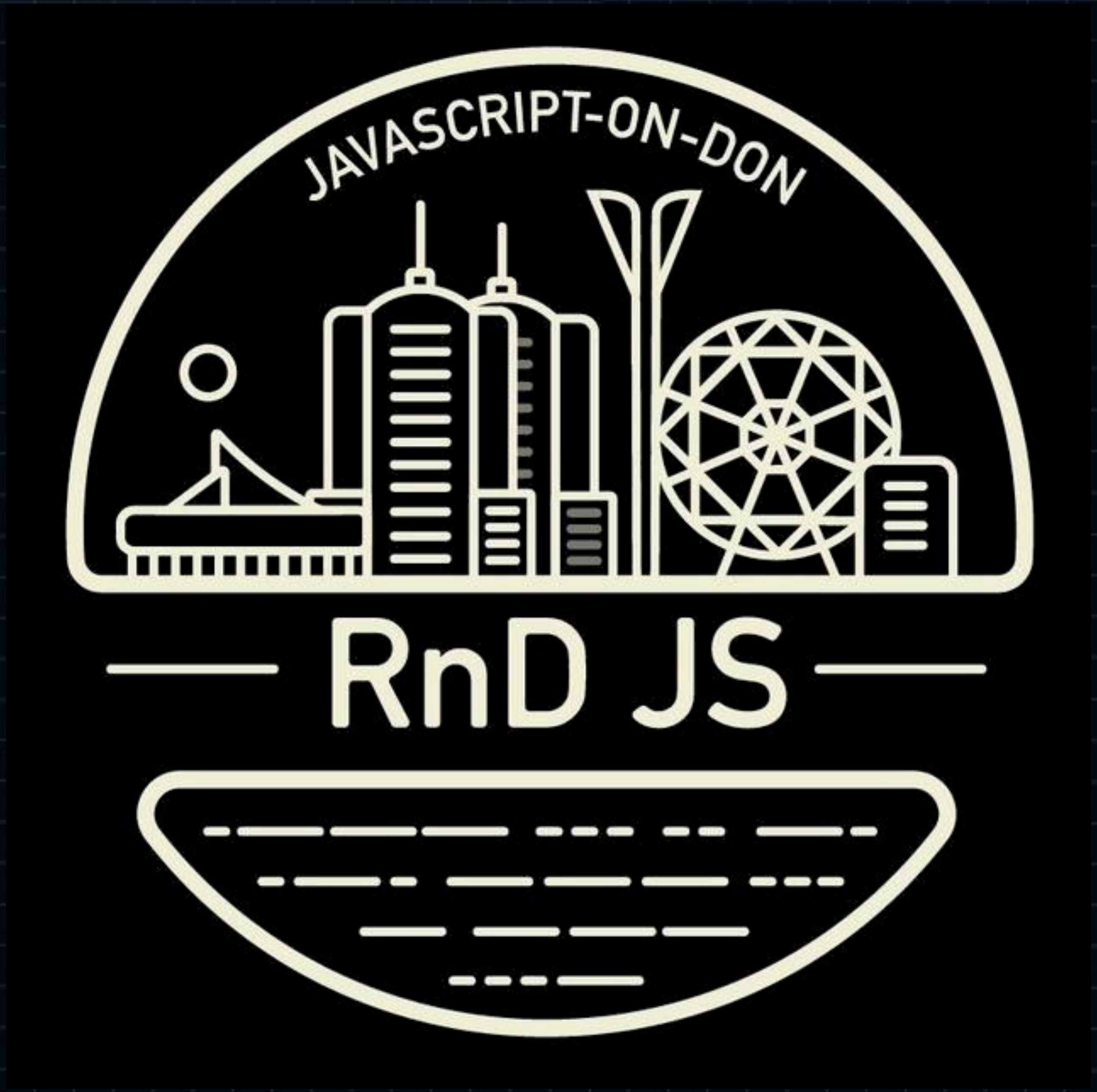
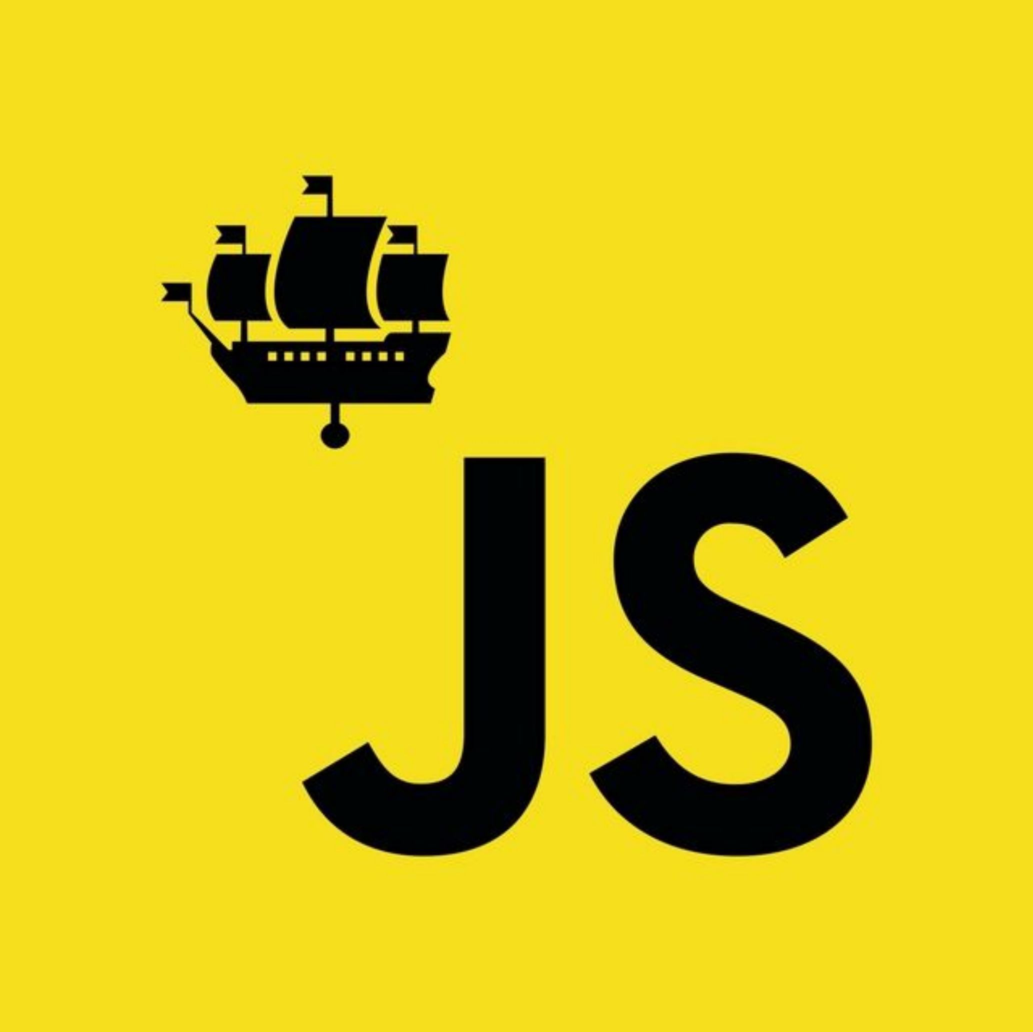
JavaScript
как конструктор
безопасного языка

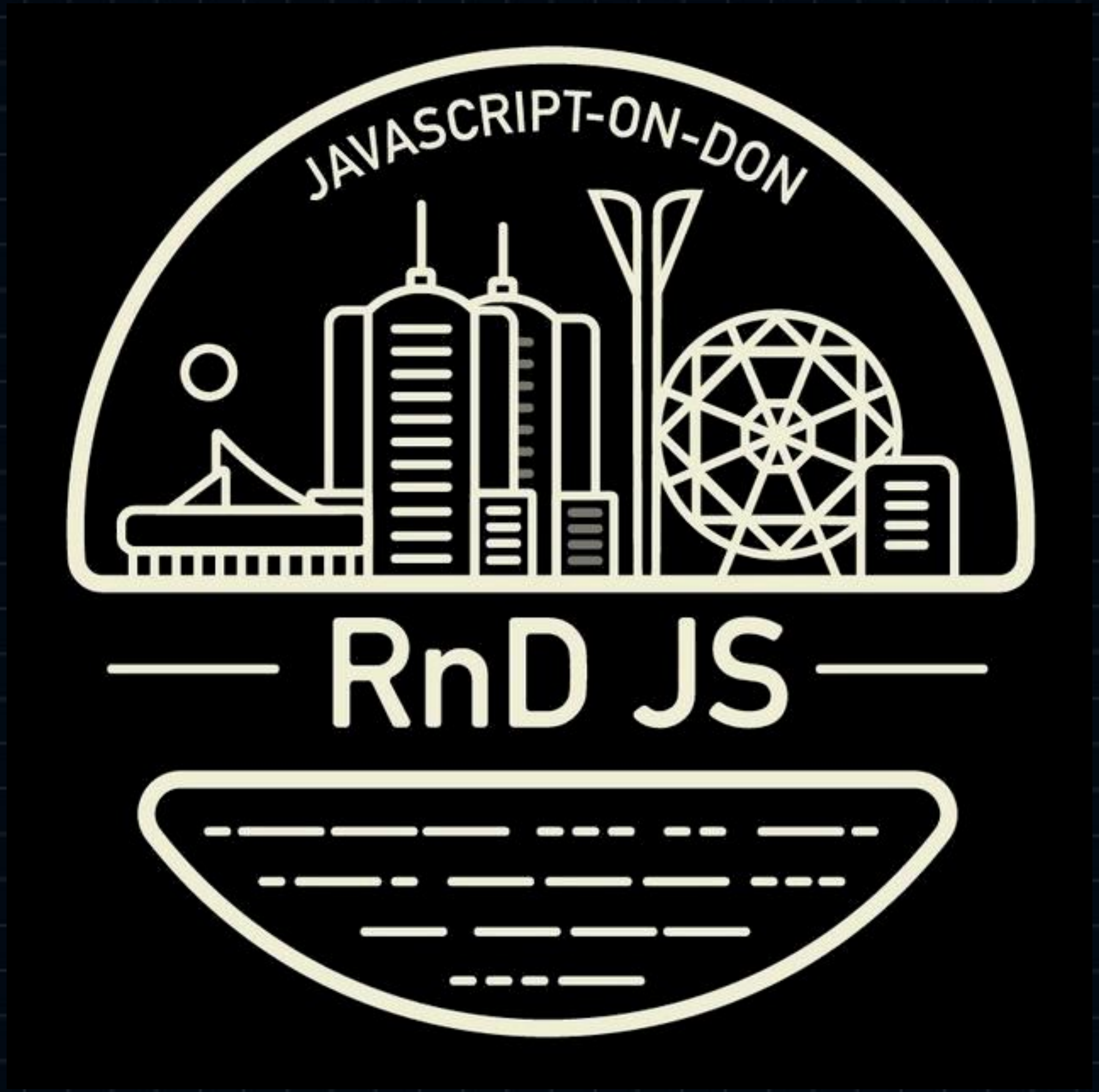
wentout

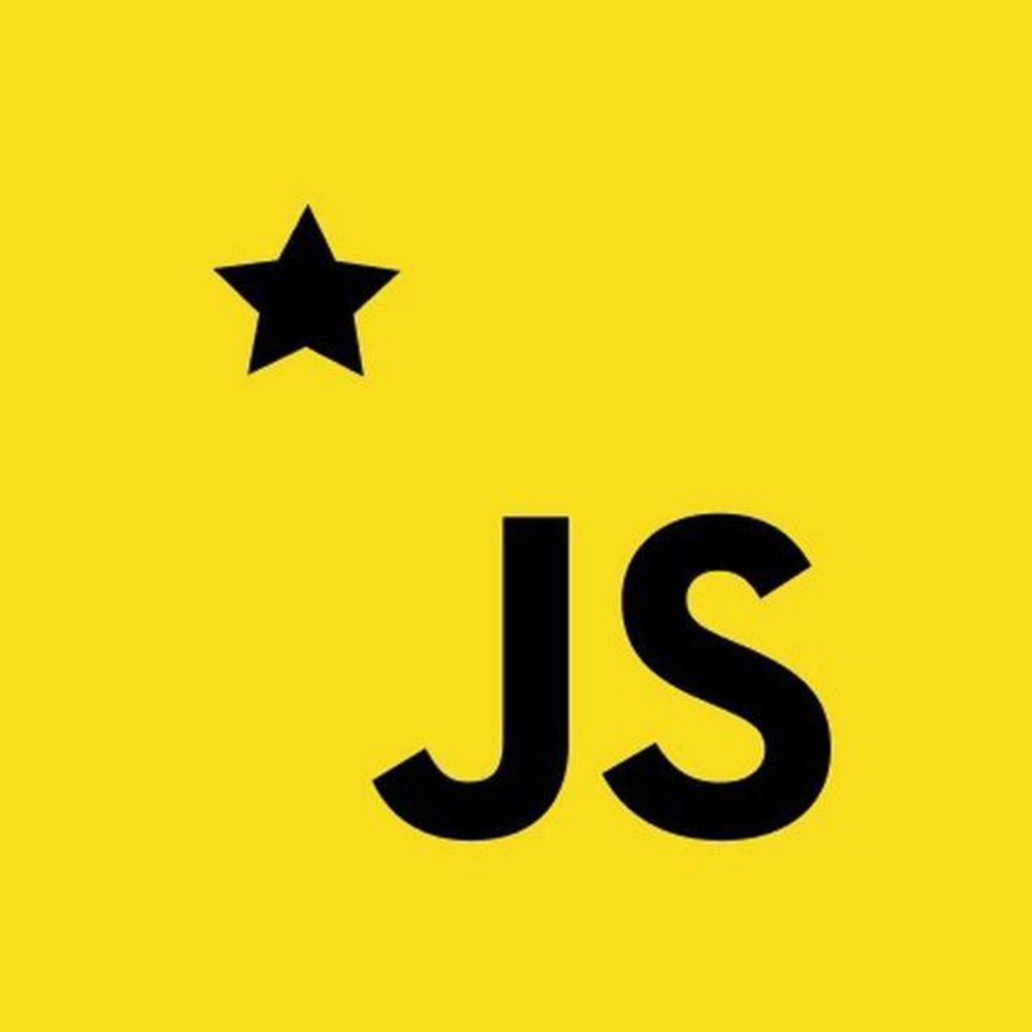
Виктор
Вершанский

SafeCode
2024


JAVASCRIPT-ON-DON
RnD JS


JS


JAVASCRIPT-ON-DON
RnD JS

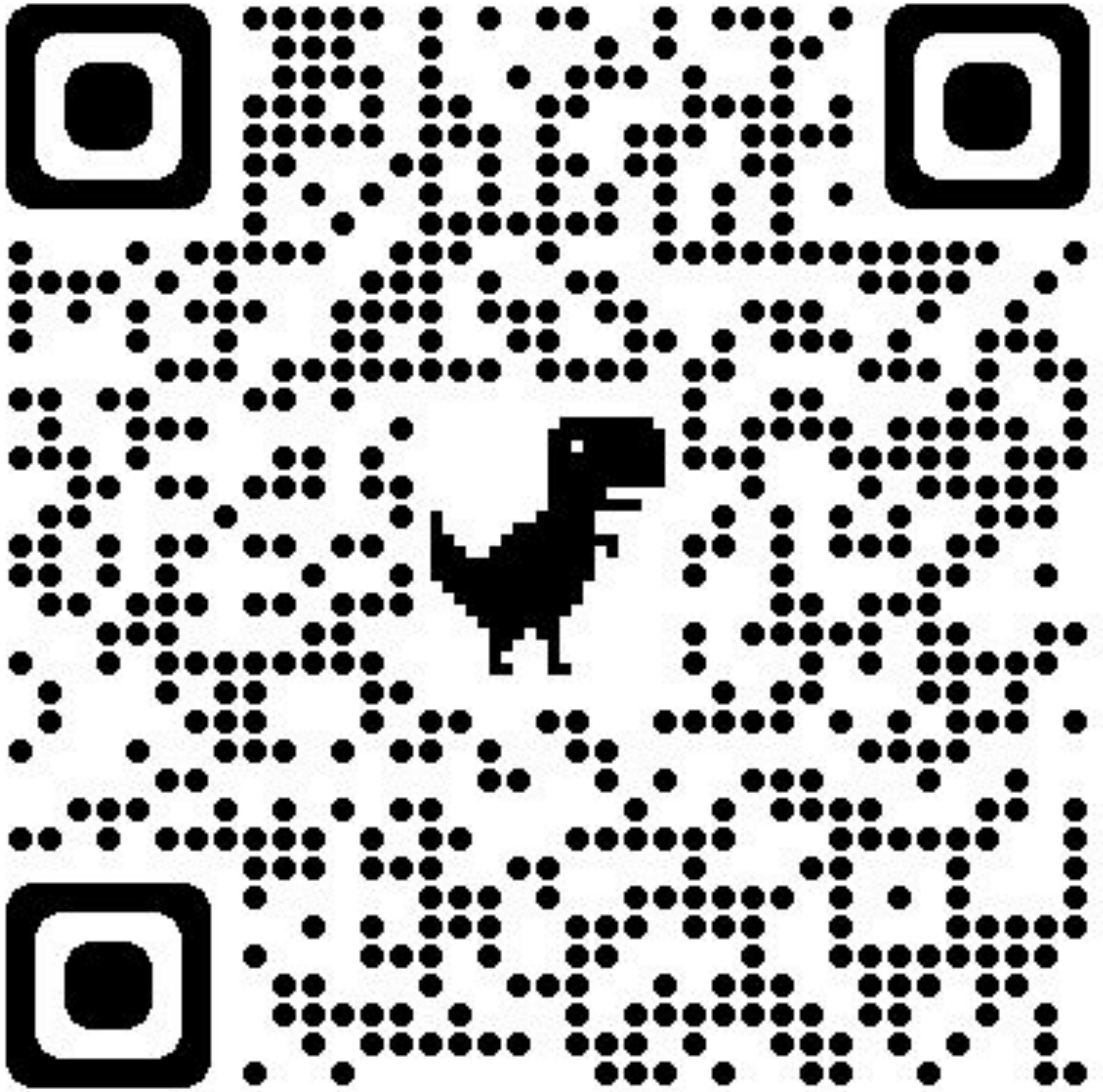

JS

Bio

Виктор

✈ wentout

- JS в продакшен 1999
- Back-End на JS в 2000
- Node.js с 2009
- Diagnostics Group
- BUGs Chrome & v8
- PhD in Economy of IT
- PMI PMBoK + Agile



о чём будет идти речь

- контекст постановки задачи
- формулировка проблематики
- аспекты, специфичные в JS
- как могут выглядеть решения
- практические примеры

контекст

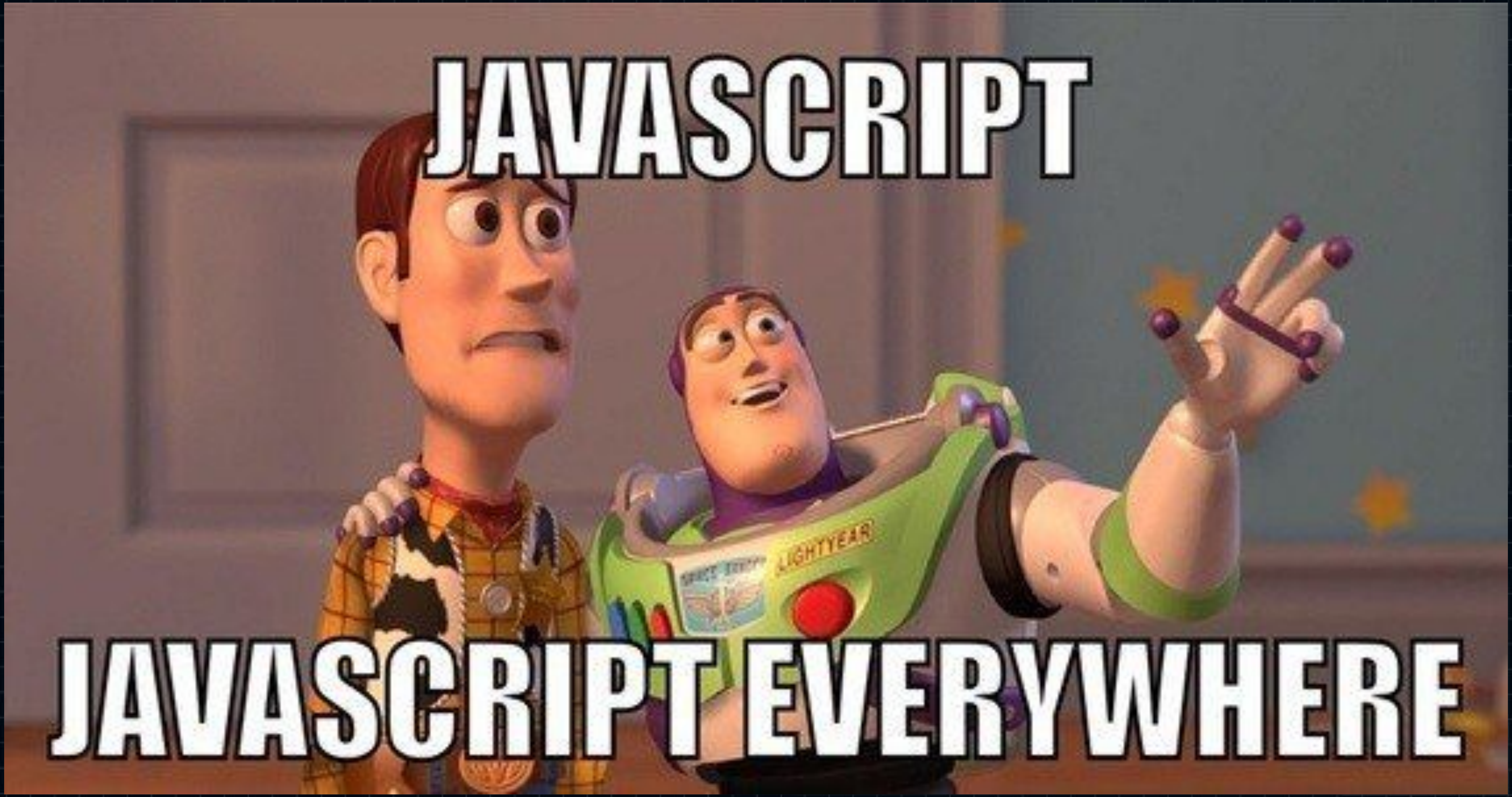
контекст

null is not a mistake

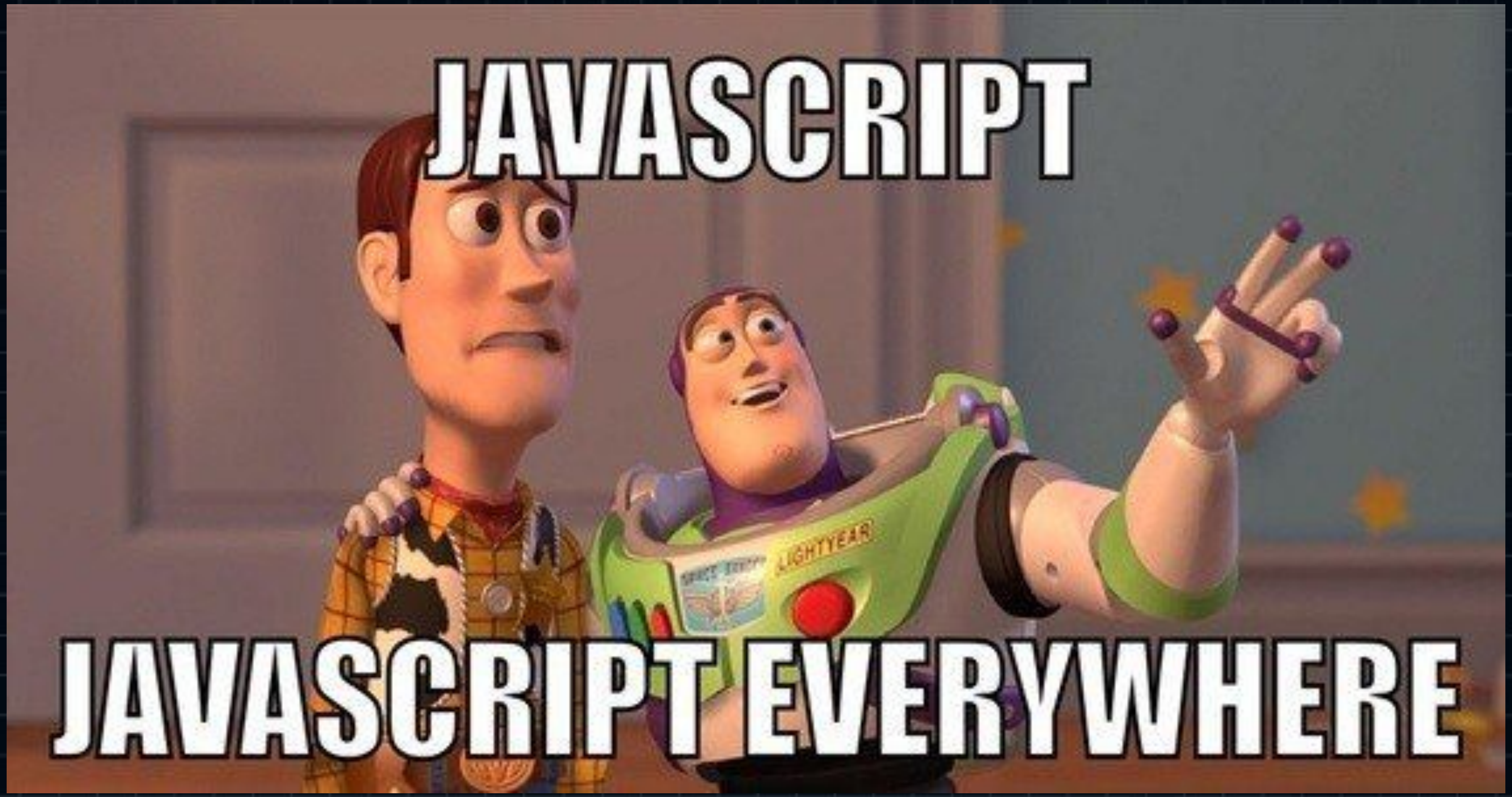
my apologies to Sir Charles Antony Richard Hoare

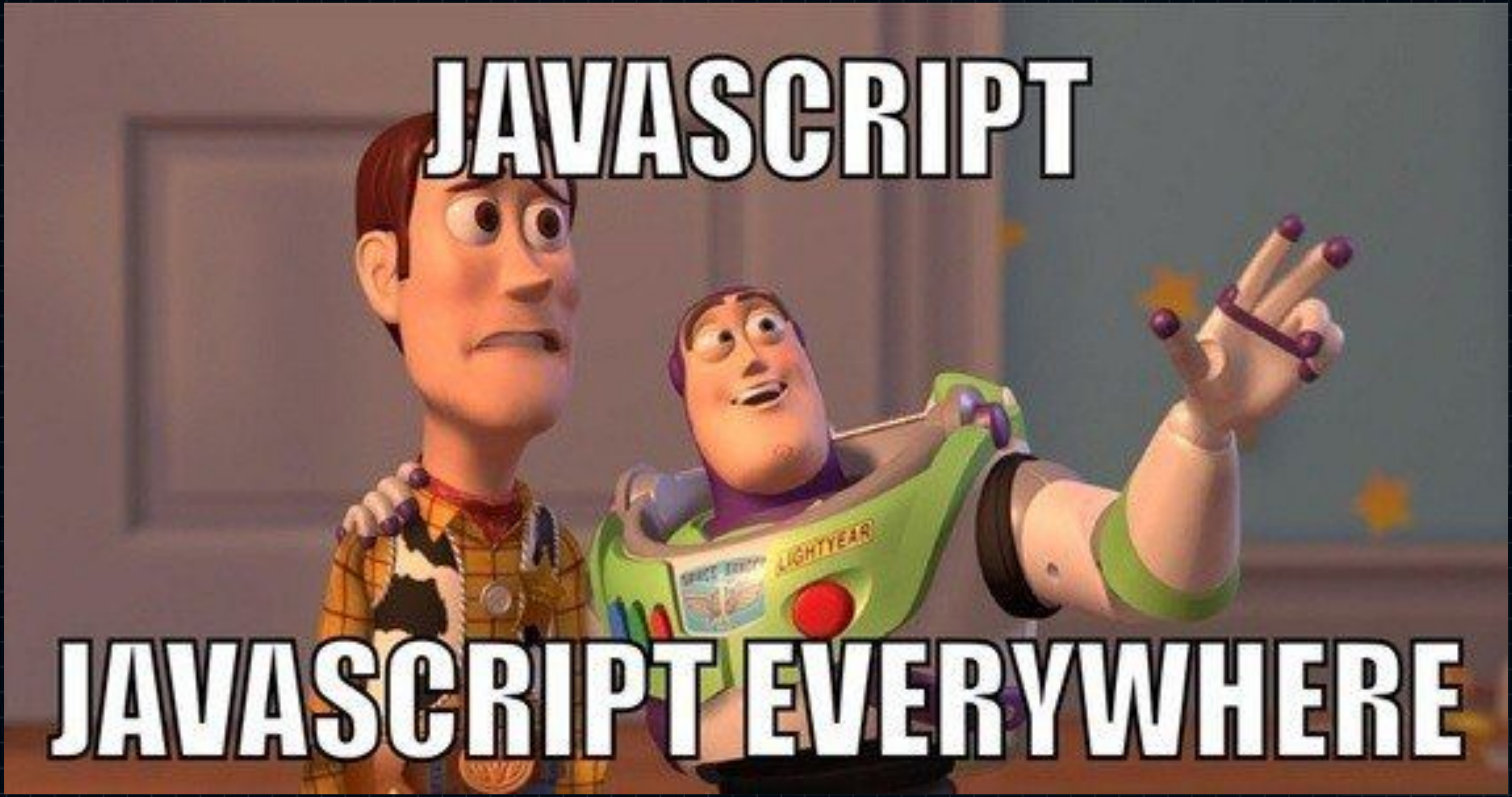
typeof null is also good

to my apologies to Brendan Eich

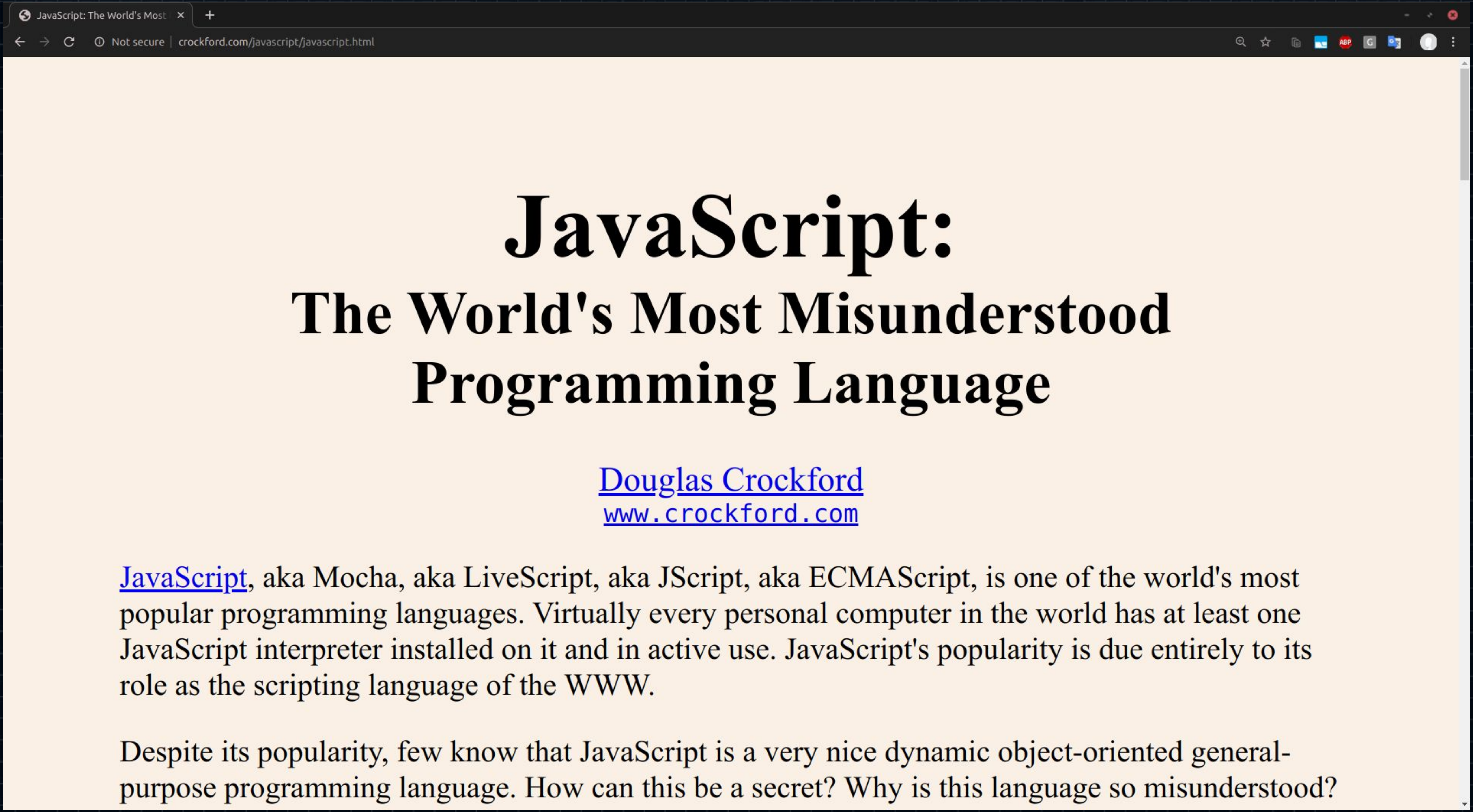

JAVASCRIPT
JAVASCRIPT EVERYWHERE
LIGHTYEAR

Disclaimer про Ожидания




JAVASCRIPT
JAVASCRIPT EVERYWHERE
LIGHTYEAR

приступим ...


JavaScript: The World's Most
Not secure
crockford.com/javascript/javascript.html

JavaScript:

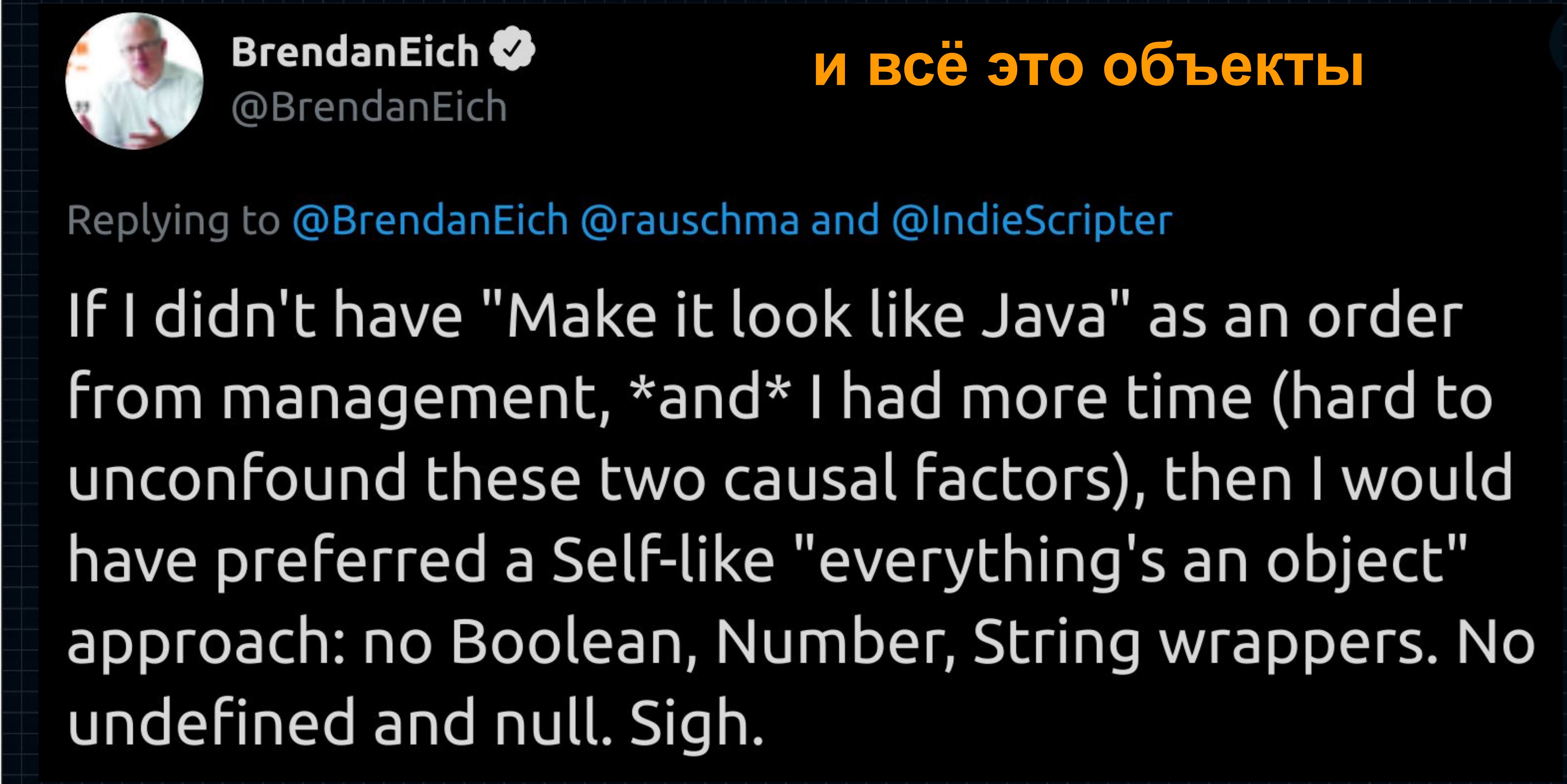
The World's Most Misunderstood Programming Language

Douglas Crockford
www.crockford.com

JavaScript, aka Mocha, aka LiveScript, aka JScript, aka ECMAScript, is one of the world's most popular programming languages. Virtually every personal computer in the world has at least one JavaScript interpreter installed on it and in active use. JavaScript's popularity is due entirely to its role as the scripting language of the WWW.

Despite its popularity, few know that JavaScript is a very nice dynamic object-oriented general-purpose programming language. How can this be a secret? Why is this language so misunderstood?

и всё это объекты

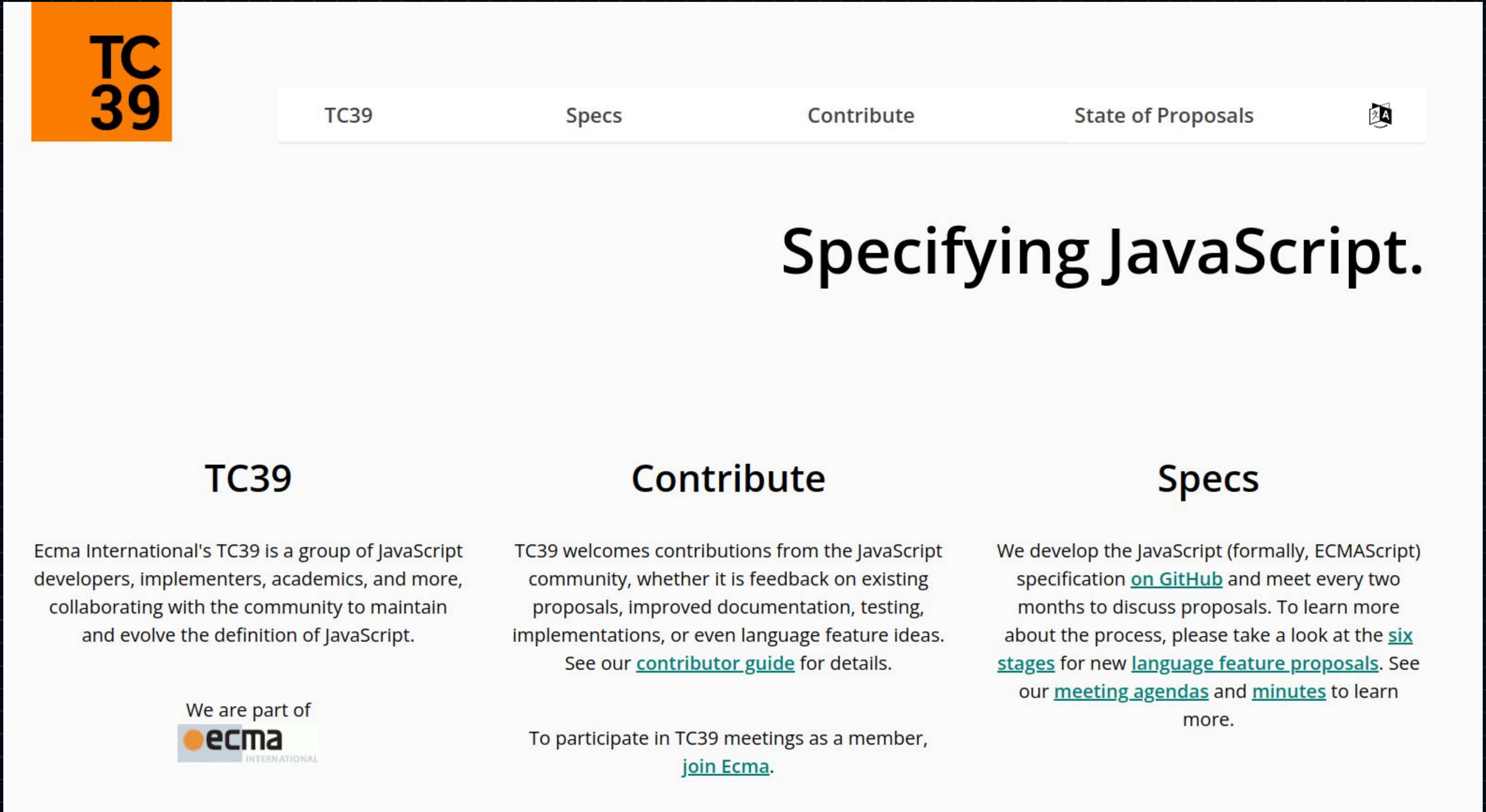


BrendanEich
@BrendanEich

Replying to @BrendanEich @rauschma and @IndieScripter

If I didn't have "Make it look like Java" as an order from management, *and* I had more time (hard to unconfound these two causal factors), then I would have preferred a Self-like "everything's an object" approach: no Boolean, Number, String wrappers. No undefined and null. Sigh.

но они всё равно её донапридумывали …



Specifying JavaScript.

TC39

Ecma International's TC39 is a group of JavaScript developers, implementers, academics, and more, collaborating with the community to maintain and evolve the definition of JavaScript.

We are part of

Contribute

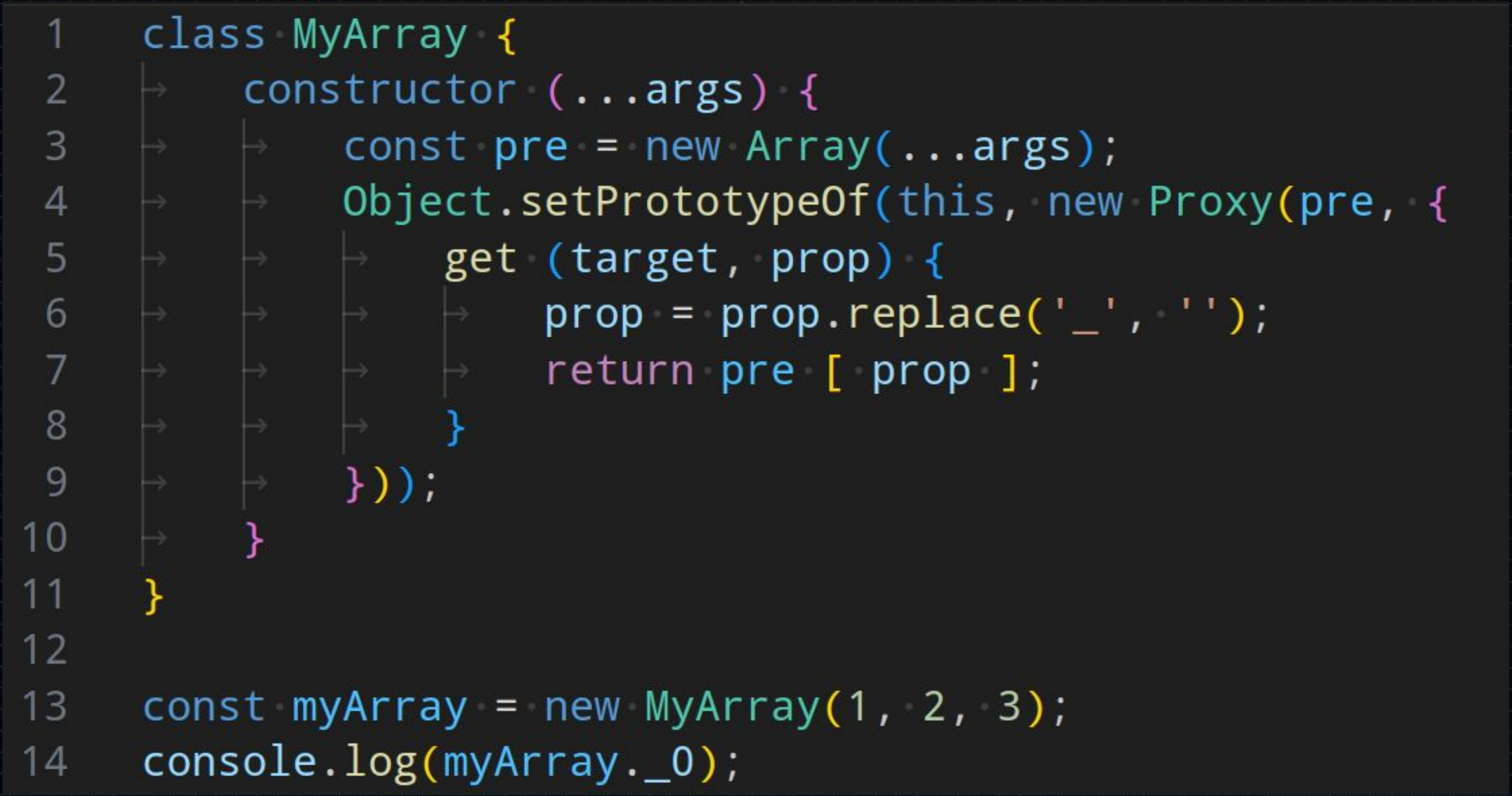
TC39 welcomes contributions from the JavaScript community, whether it is feedback on existing proposals, improved documentation, testing, implementations, or even language feature ideas. See our contributor guide for details.

To participate in TC39 meetings as a member, join Ecma.

Specs

We develop the JavaScript (formally, ECMAScript) specification on GitHub and meet every two months to discuss proposals. To learn more about the process, please take a look at the six stages for new language feature proposals. See our meeting agendas and minutes to learn more.

практический пример конструирования



```
class MyArray {
    constructor (...args) {
        const pre = new Array(...args);
        Object.setPrototypeOf(this, new Proxy(pre, {
            get (target, prop) {
                prop = prop.replace('_', '');
                return pre [ prop ];
            }
        }));
    }
}

const myArray = new MyArray(1, 2, 3);
console.log(myArray._0);
```

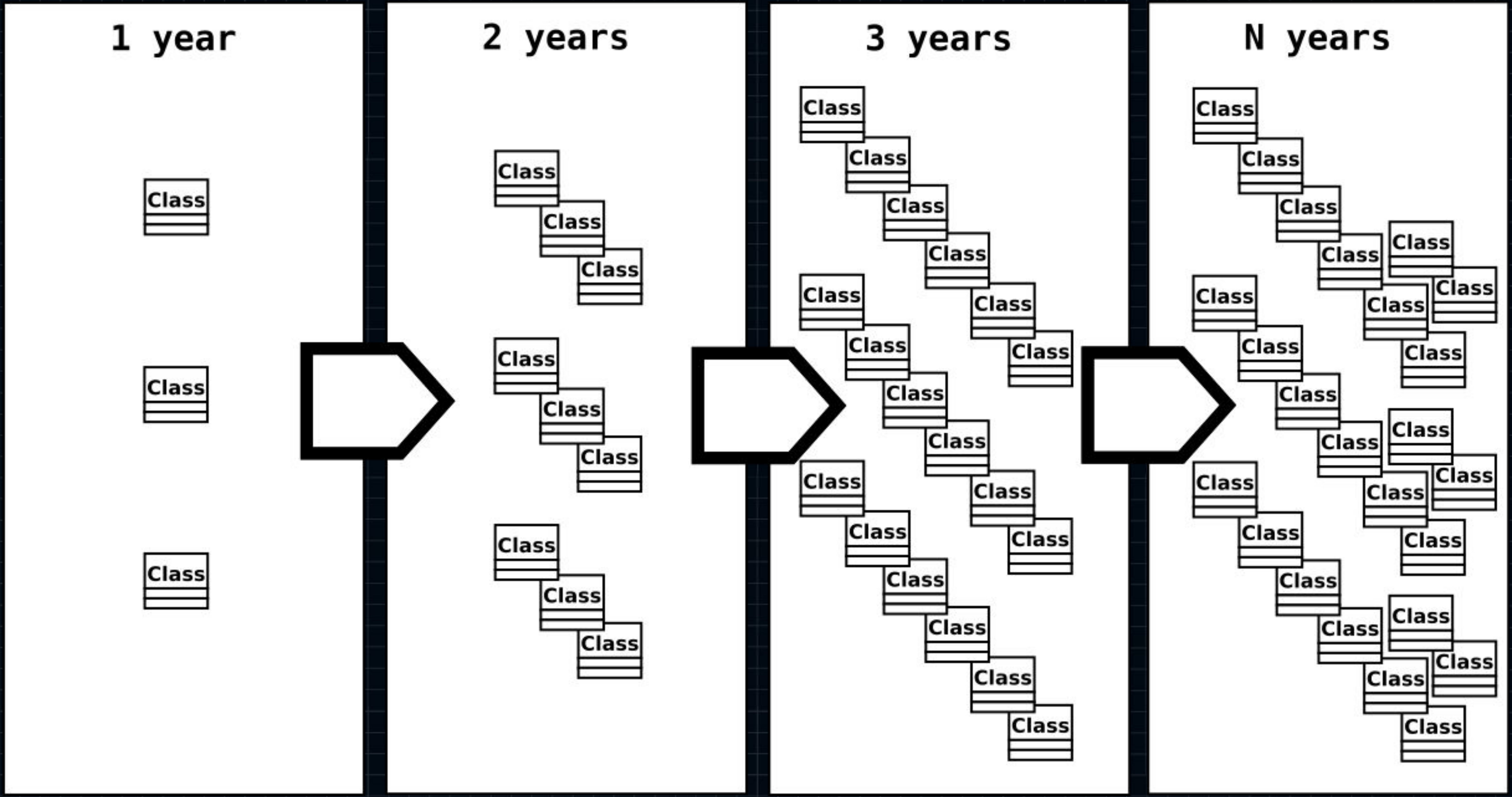
и ещё один почти убедительный рабочий пример

проблематика

проблематика

2

- контекст постановки задачи
- формулировка проблематики
- **как создаётся код для решения**

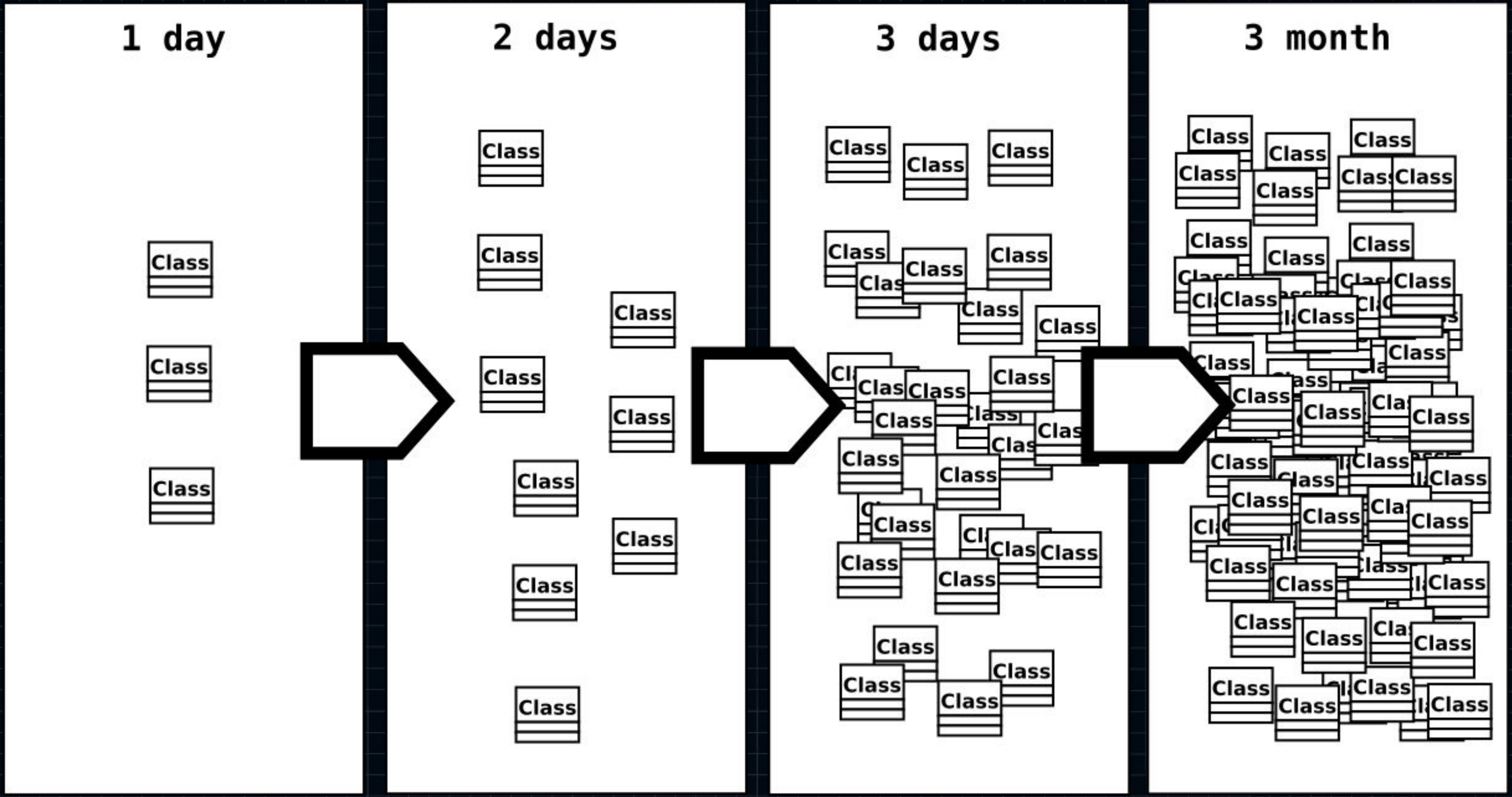


1 year

2 years

3 years

N years

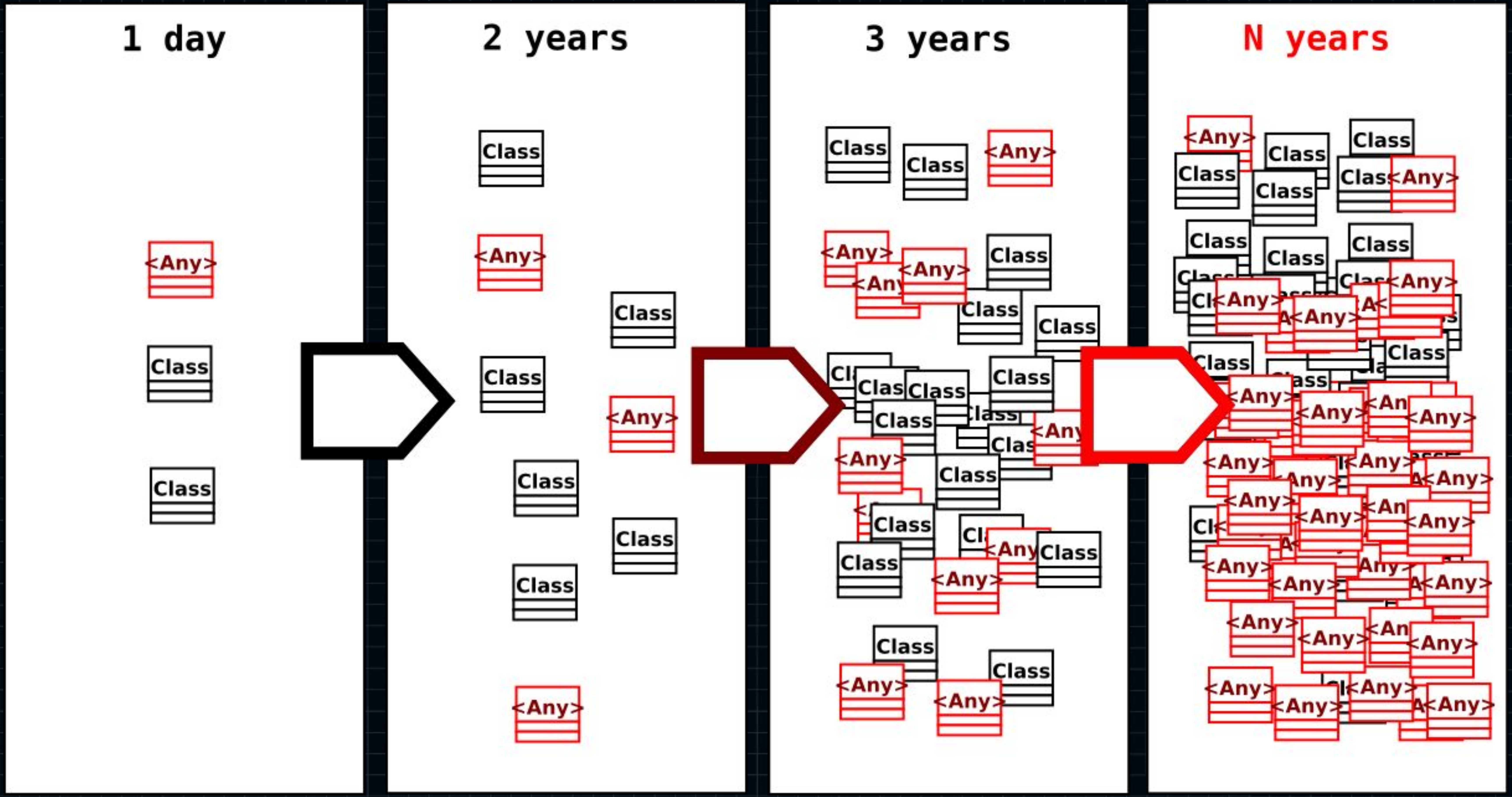


1 day

2 days

3 days

3 month

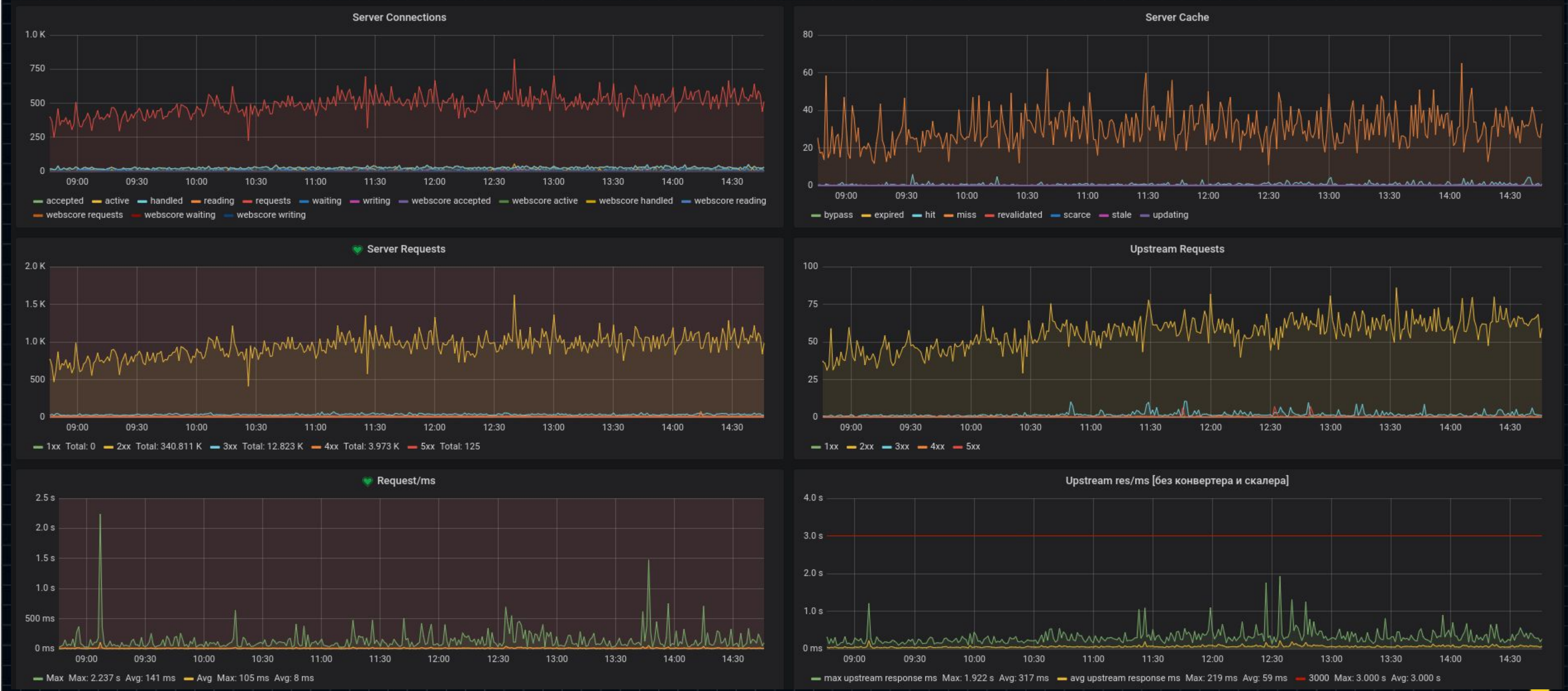


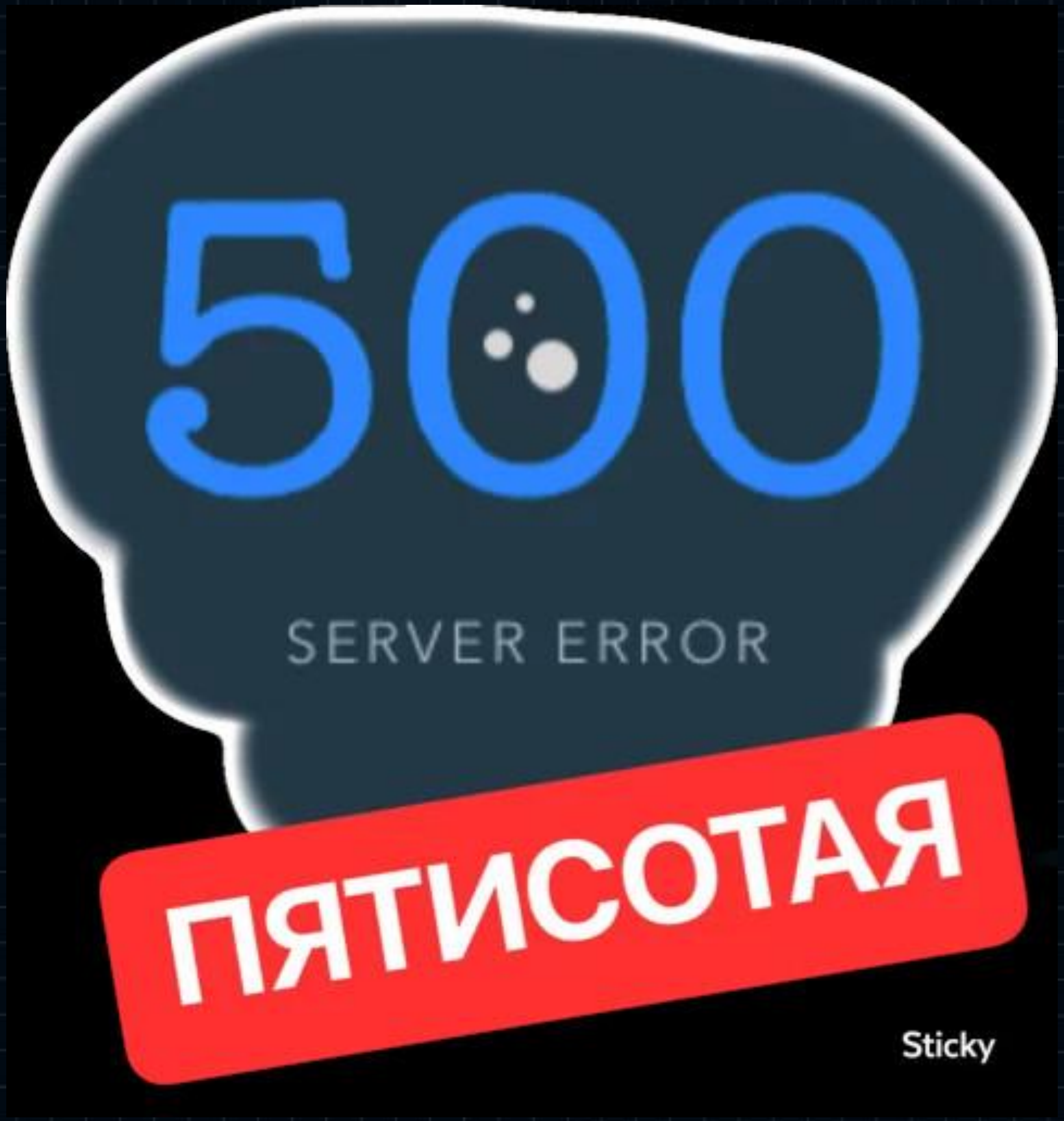
1 day

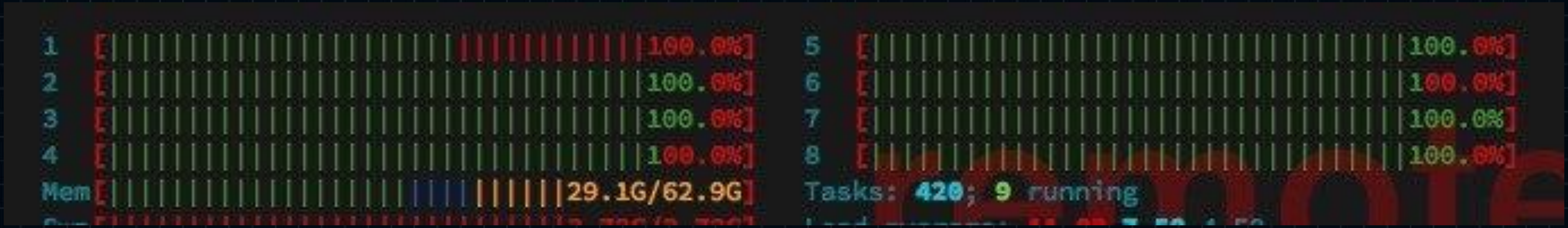
2 years

3 years

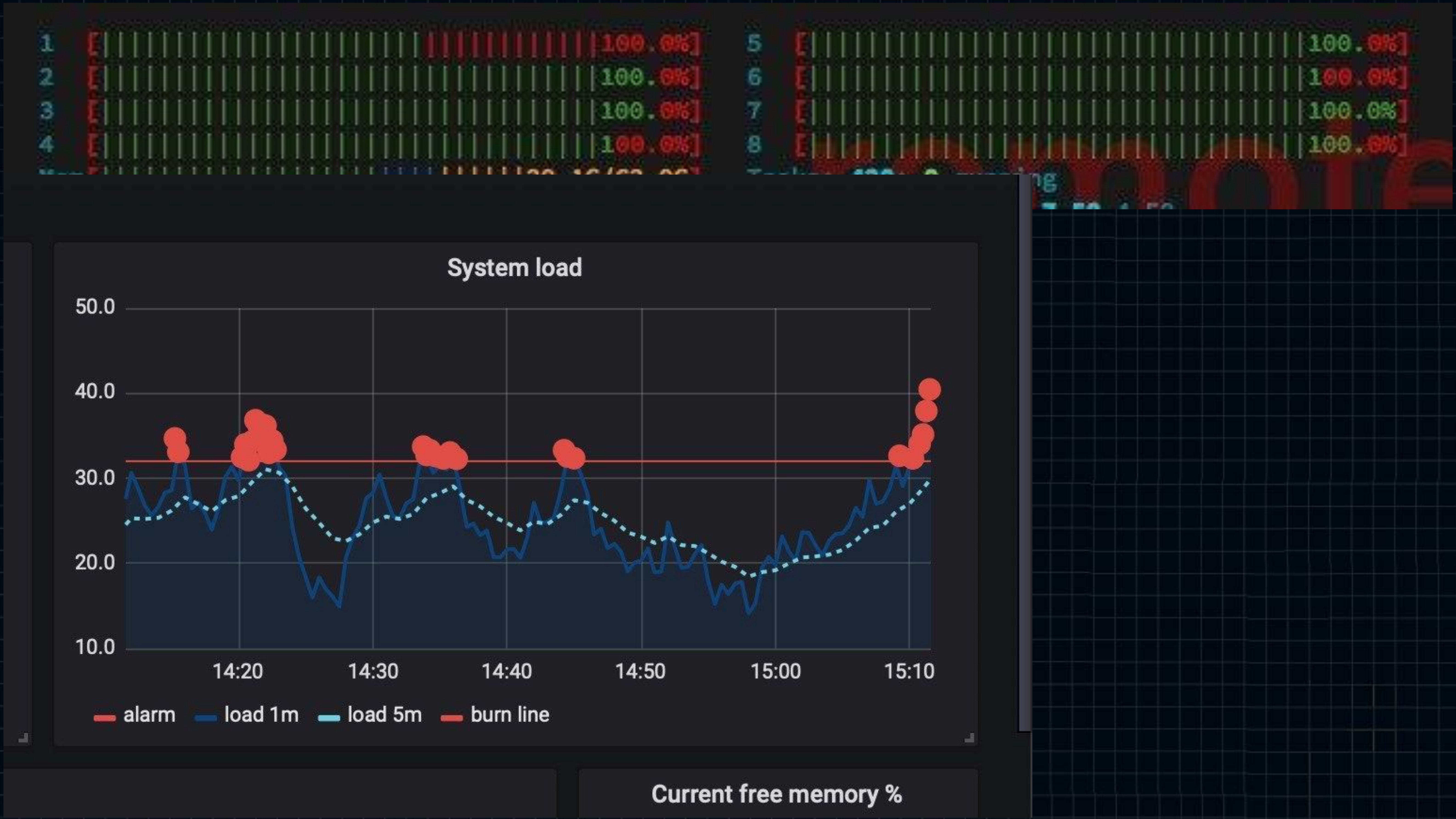
N years


Server Connections
1.0 K
750
500
250
0
09:00
09:30
10:00
10:30
11:00
11:30
12:00
12:30
13:00
13:30
14:00
14:30
accepted
active
handled
reading
requests
waiting
writing
webscore accepted
webscore active
webscore handled
webscore reading
webscore requests
webscore waiting
webscore writing
Server Cache
80
60
40
20
0
bypass
expired
hit
miss
revalidated
scarce
stale
updating
Server Requests
2.0 K
1.5 K
1.0 K
500
1xx Total: 0
2xx Total: 340.811 K
3xx Total: 12.823 K
4xx Total: 3.973 K
5xx Total: 125
Upstream Requests
100
75
50
25
1xx
2xx
3xx
4xx
5xx
Request/ms
2.5 s
2.0 s
1.5 s
1.0 s
500 ms
0 ms
Max Max: 2.237 s Avg: 141 ms
Avg Max: 105 ms Avg: 8 ms
Upstream res/ms [без конвертера и скалера]
4.0 s
3.0 s
2.0 s
1.0 s
0 ms
max upstream response ms Max: 1.922 s Avg: 317 ms
avg upstream response ms Max: 219 ms Avg: 59 ms
3000 Max: 3.000 s Avg: 3.000 s

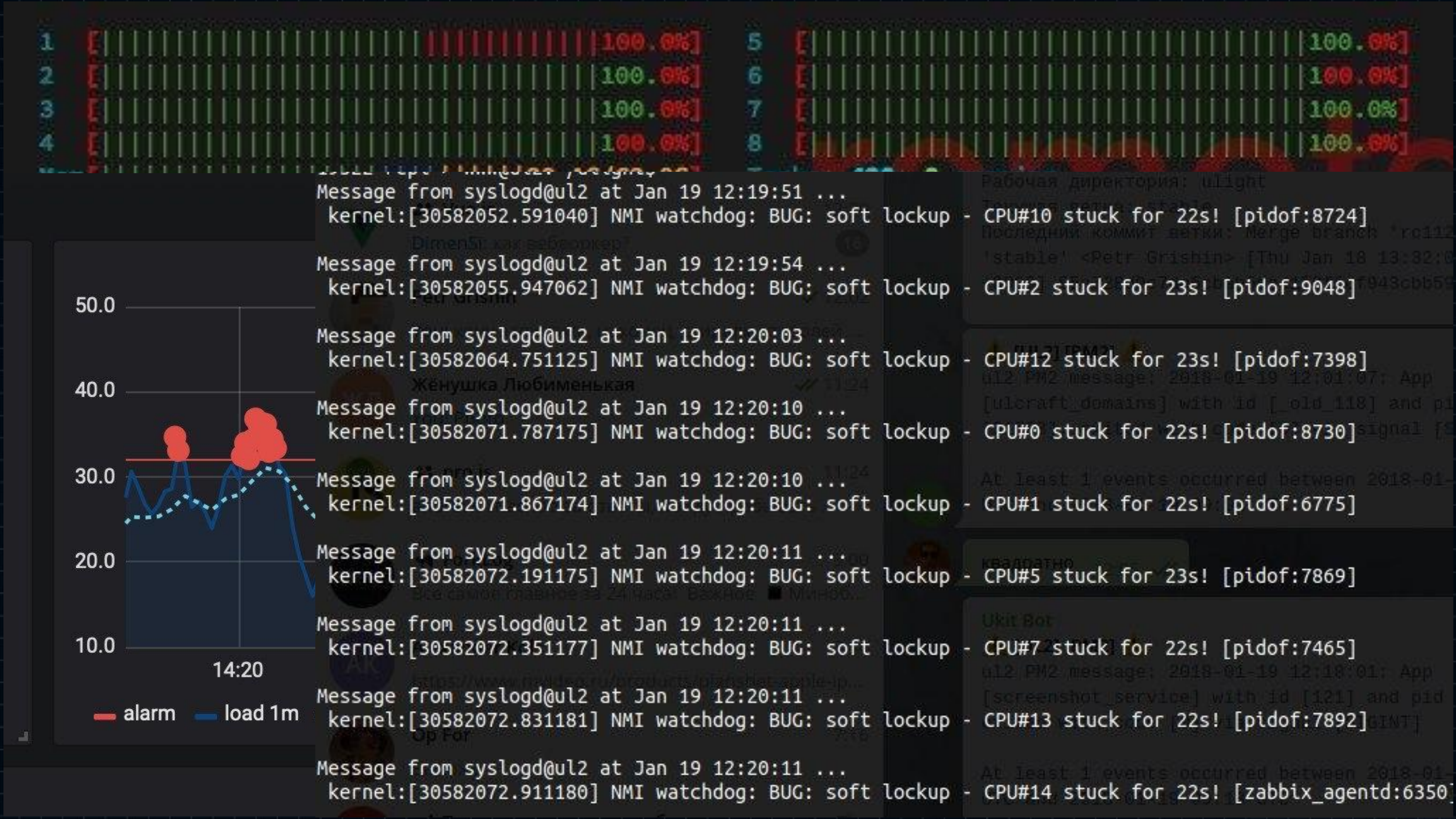

500
SERVER ERROR
ПЯТИСОТАЯ
Sticky



```
1   [|||||||||||||||||||||||||||||||||||100.0%]   5   [|||||||||||||||||||||||||||||||||||100.0%]
2   [|||||||||||||||||||||||||||||||||||100.0%]   6   [|||||||||||||||||||||||||||||||||||100.0%]
3   [|||||||||||||||||||||||||||||||||||100.0%]   7   [|||||||||||||||||||||||||||||||||||100.0%]
4   [|||||||||||||||||||||||||||||||||||100.0%]   8   [|||||||||||||||||||||||||||||||||||100.0%]
Mem[|||||||||||||||||||||||||||||29.1G/62.9G]   Tasks: 420; 9 running
```



```
1  [||||||||||||||||||||||||||||||||||||100.0%]   5  [||||||||||||||||||||||||||||||||||||100.0%]
2  [||||||||||||||||||||||||||||||||||||100.0%]   6  [||||||||||||||||||||||||||||||||||||100.0%]
3  [||||||||||||||||||||||||||||||||||||100.0%]   7  [||||||||||||||||||||||||||||||||||||100.0%]
4  [||||||||||||||||||||||||||||||||||||100.0%]   8  [||||||||||||||||||||||||||||||||||||100.0%]
```



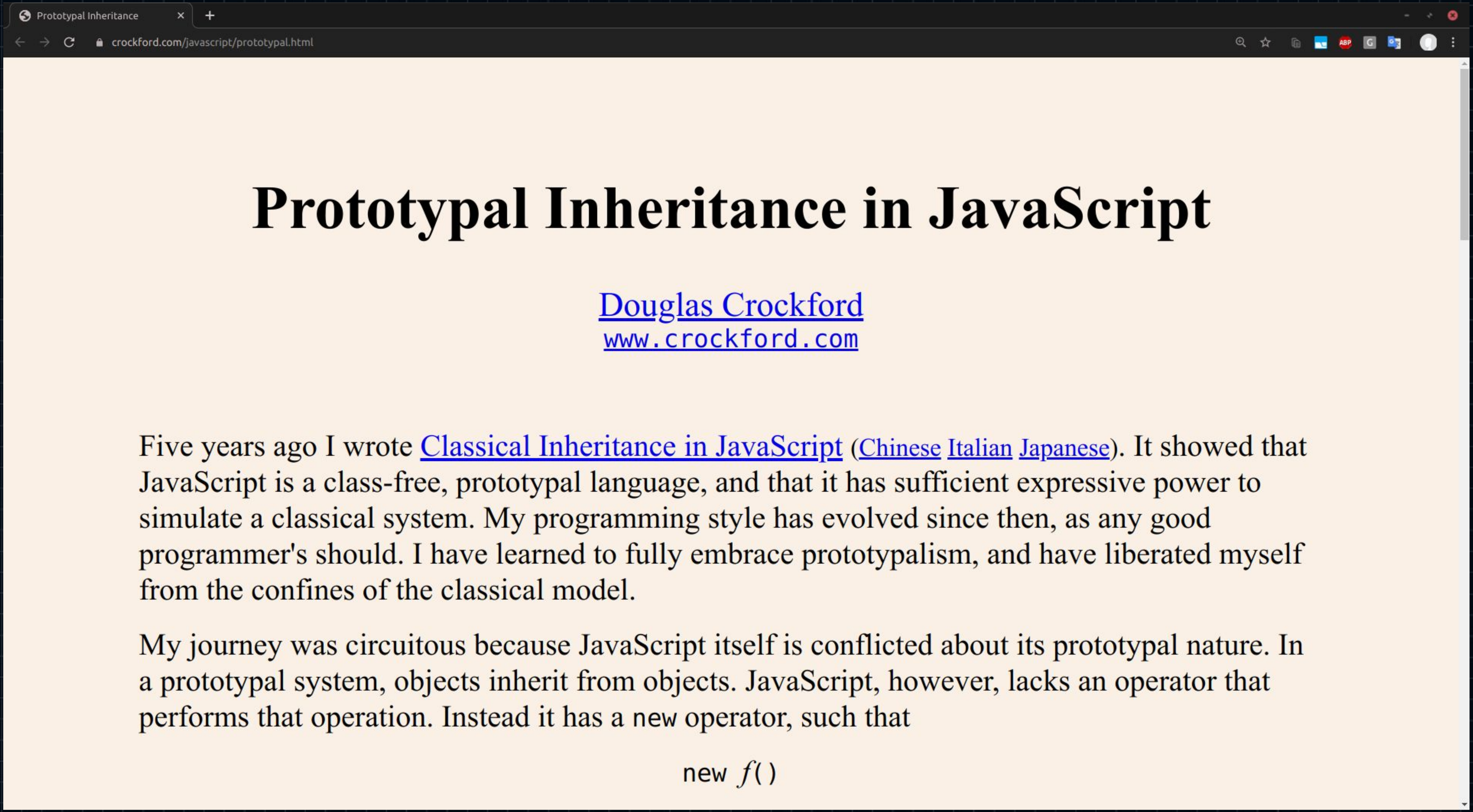
```
Message from syslogd@ul2 at Jan 19 12:19:51 ...
 kernel:[30582052.591040] NMI watchdog: BUG: soft lockup - CPU#10 stuck for 22s! [pidof:8724]

Message from syslogd@ul2 at Jan 19 12:19:54 ...
 kernel:[30582055.947062] NMI watchdog: BUG: soft lockup - CPU#2 stuck for 23s! [pidof:9048]

Message from syslogd@ul2 at Jan 19 12:20:03 ...
 kernel:[30582064.751125] NMI watchdog: BUG: soft lockup - CPU#12 stuck for 23s! [pidof:7398]

Message from syslogd@ul2 at Jan 19 12:20:10 ...
 kernel:[30582071.787175] NMI watchdog: BUG: soft lockup - CPU#0 stuck for 22s! [pidof:8730]

Message from syslogd@ul2 at Jan 19 12:20:10 ...
 kernel:[30582071.867174] NMI watchdog: BUG: soft lockup - CPU#1 stuck for 22s! [pidof:6775]

Message from syslogd@ul2 at Jan 19 12:20:11 ...
 kernel:[30582072.191175] NMI watchdog: BUG: soft lockup - CPU#5 stuck for 23s! [pidof:7869]

Message from syslogd@ul2 at Jan 19 12:20:11 ...
 kernel:[30582072.351177] NMI watchdog: BUG: soft lockup - CPU#7 stuck for 22s! [pidof:7465]

Message from syslogd@ul2 at Jan 19 12:20:11 ...
 kernel:[30582072.831181] NMI watchdog: BUG: soft lockup - CPU#13 stuck for 22s! [pidof:7892]

Message from syslogd@ul2 at Jan 19 12:20:11 ...
 kernel:[30582072.911180] NMI watchdog: BUG: soft lockup - CPU#14 stuck for 22s! [zabbix_agentd:6350
```

Безопасен ли JavaScript для Программиста ?

☺

специфика

специфика



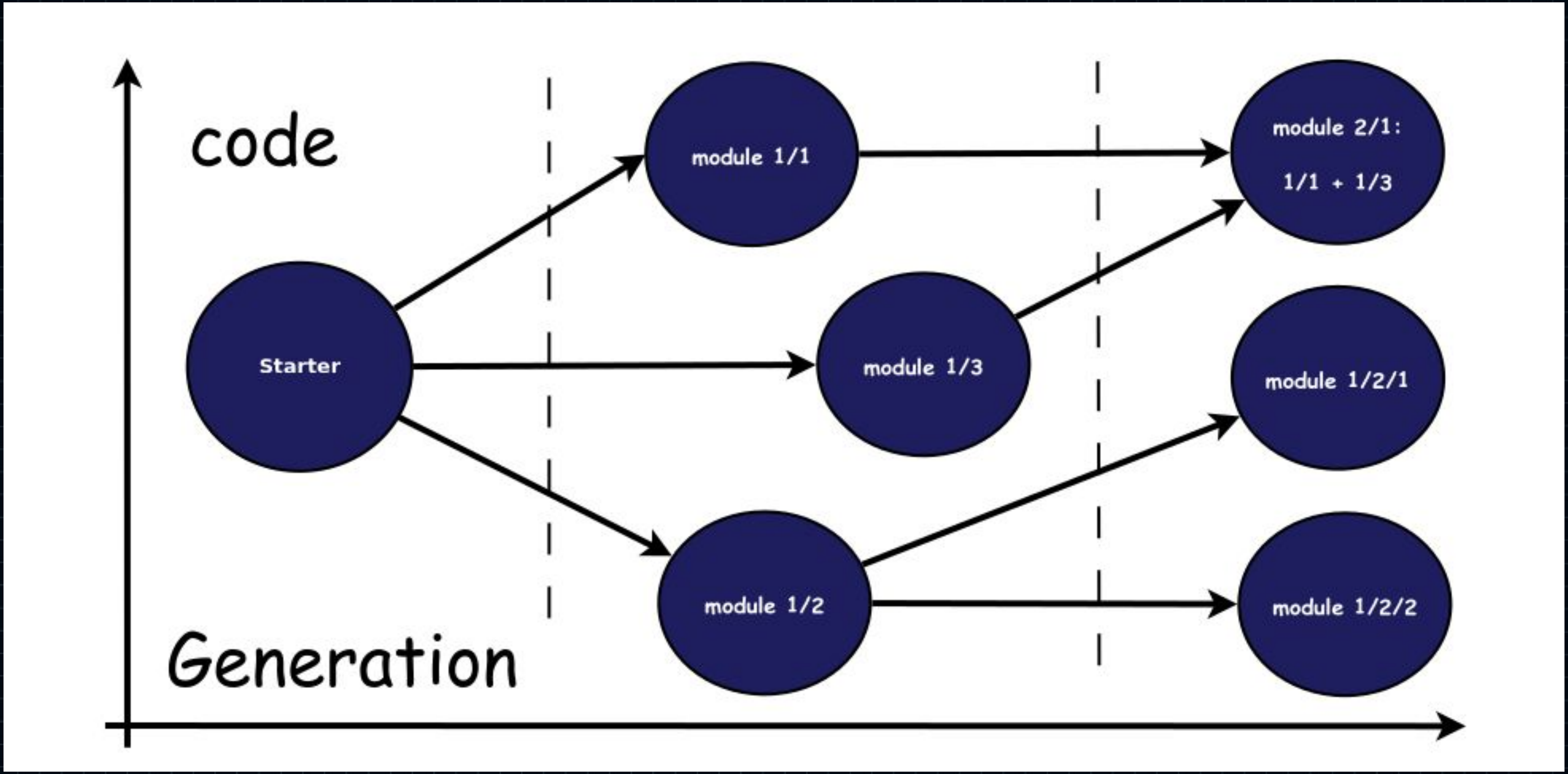
Prototypal Inheritance in JavaScript

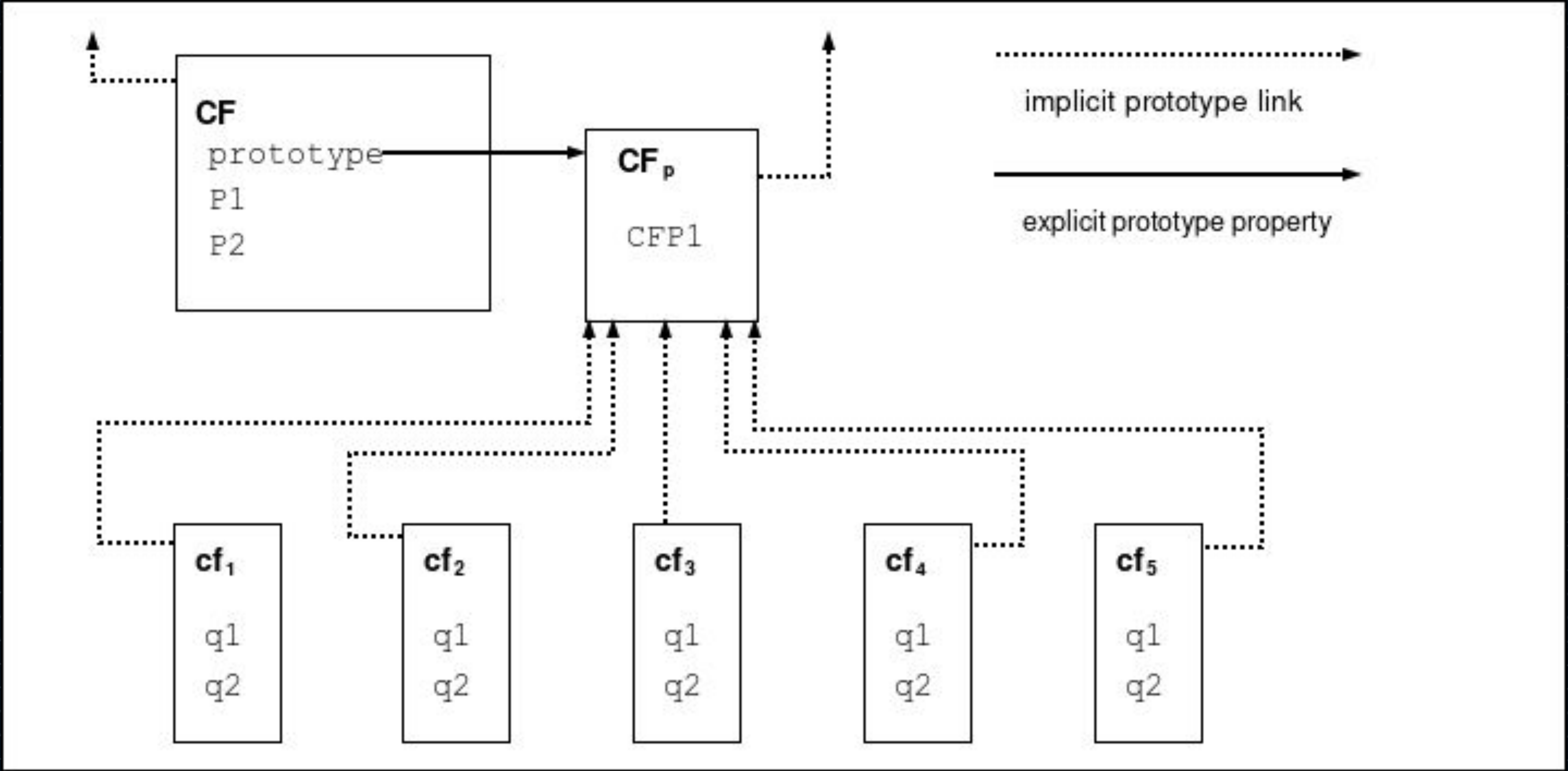
Douglas Crockford
www.crockford.com

Five years ago I wrote Classical Inheritance in JavaScript (Chinese Italian Japanese). It showed that JavaScript is a class-free, prototypal language, and that it has sufficient expressive power to simulate a classical system. My programming style has evolved since then, as any good programmer's should. I have learned to fully embrace prototypalism, and have liberated myself from the confines of the classical model.

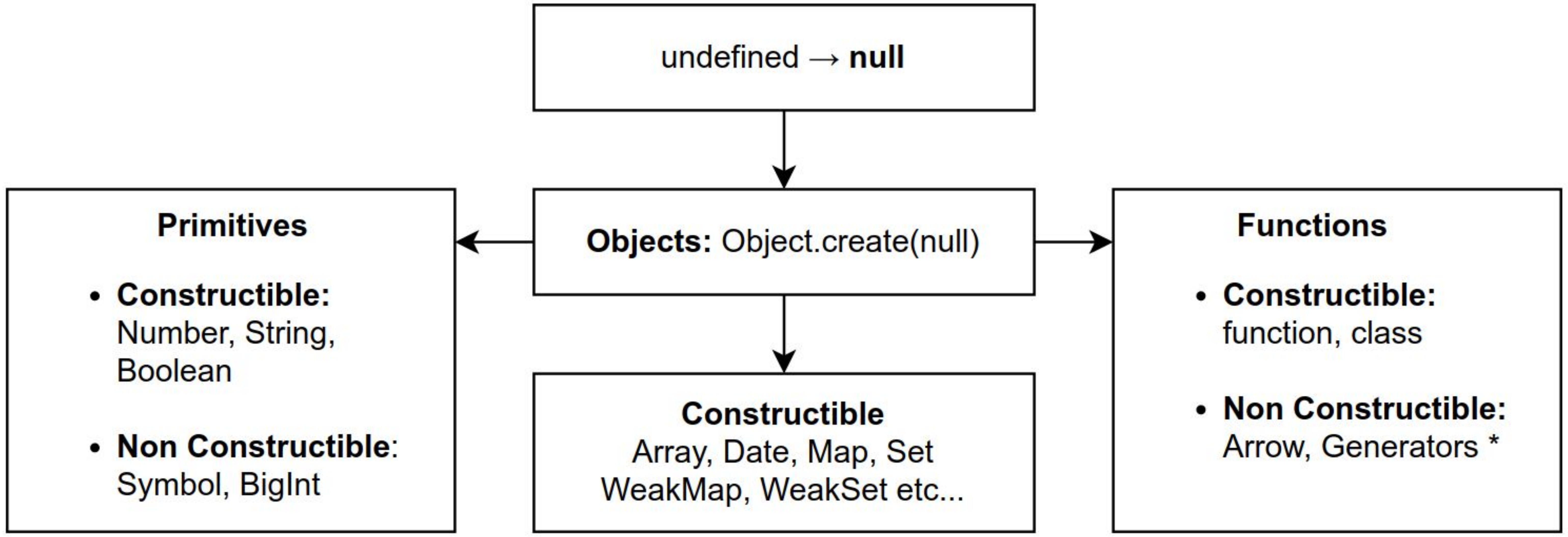
My journey was circuitous because JavaScript itself is conflicted about its prototypal nature. In a prototypal system, objects inherit from objects. JavaScript, however, lacks an operator that performs that operation. Instead it has a new operator, such that

new *f*()


code
Starter
module 1/1
module 1/3
module 1/2
module 2/1:
1/1 + 1/3
module 1/2/1
module 1/2/2
Generation


CF
prototype
P1
P2
CF$_p$
CFP1
implicit prototype link
explicit prototype property
cf$_1$
q1
q2
cf$_2$
q1
q2
cf$_3$
q1
q2
cf$_4$
q1
q2
cf$_5$
q1
q2

JavaScript Objects Topology



undefined → **null**

Objects: Object.create(null)

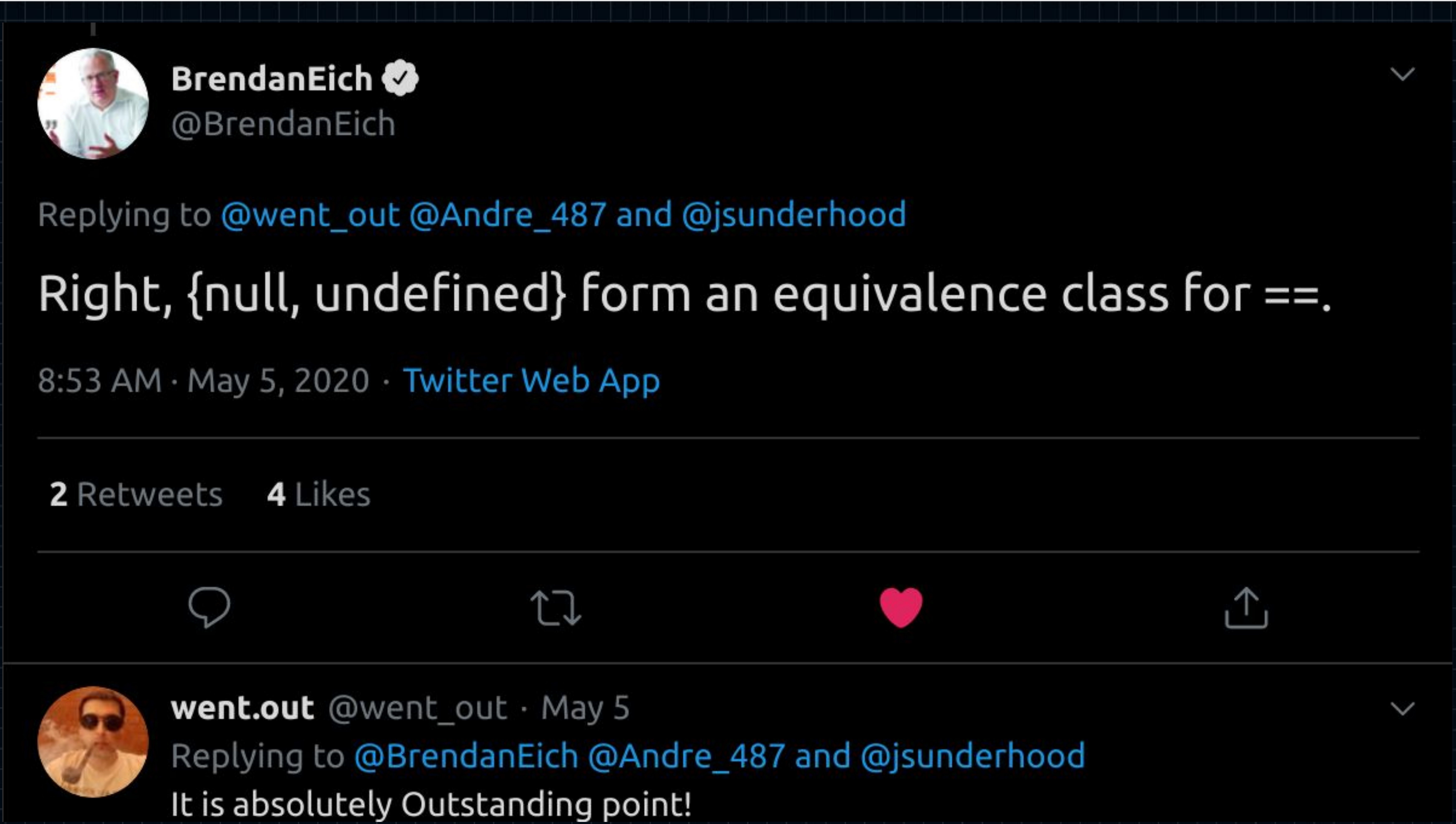
Primitives

- **Constructible:** Number, String, Boolean
- **Non Constructible**: Symbol, BigInt

Constructible
Array, Date, Map, Set
WeakMap, WeakSet etc...

Functions

- **Constructible:** function, class
- **Non Constructible:** Arrow, Generators *



BrendanEich
@BrendanEich

Replying to @went_out @Andre_487 and @jsunderhood

Right, {null, undefined} form an equivalence class for ==.

8:53 AM · May 5, 2020 · Twitter Web App

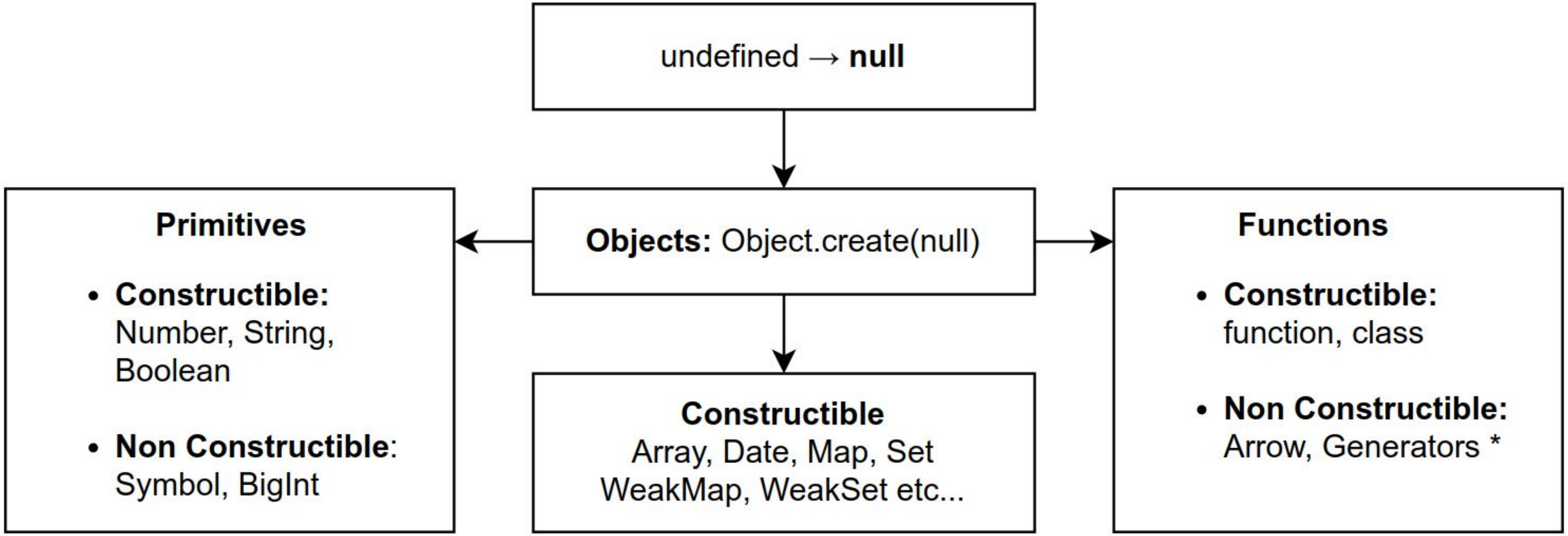
2 Retweets 4 Likes

went.out @went_out · May 5
Replying to @BrendanEich @Andre_487 and @jsunderhood
It is absolutely Outstanding point!

null is not a mistake

my apologies to Sir Charles Antony Richard Hoare

JavaScript Objects Topology



undefined → **null**

Objects: Object.create(null)

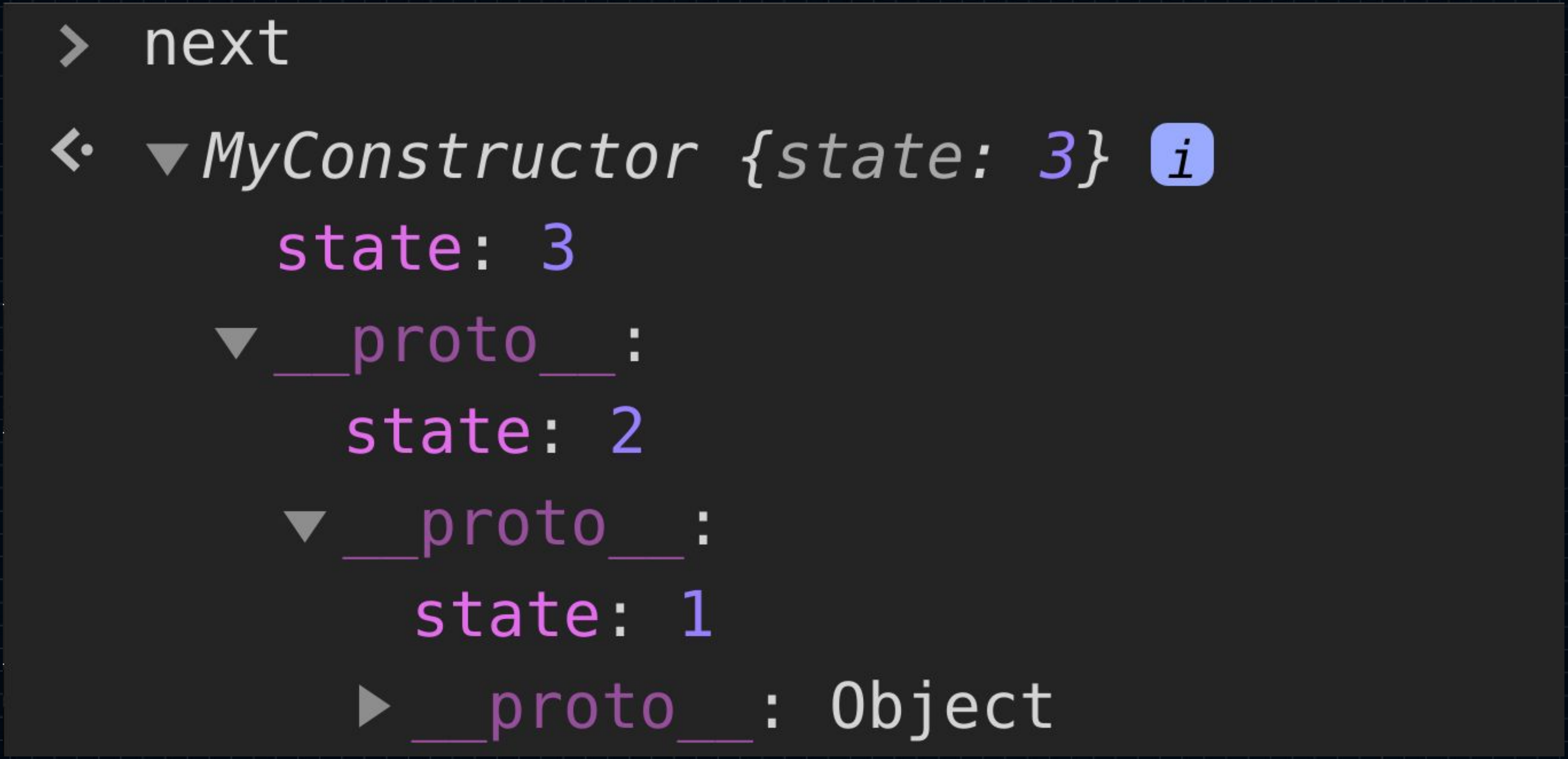
Primitives

- **Constructible:** Number, String, Boolean
- **Non Constructible**: Symbol, BigInt

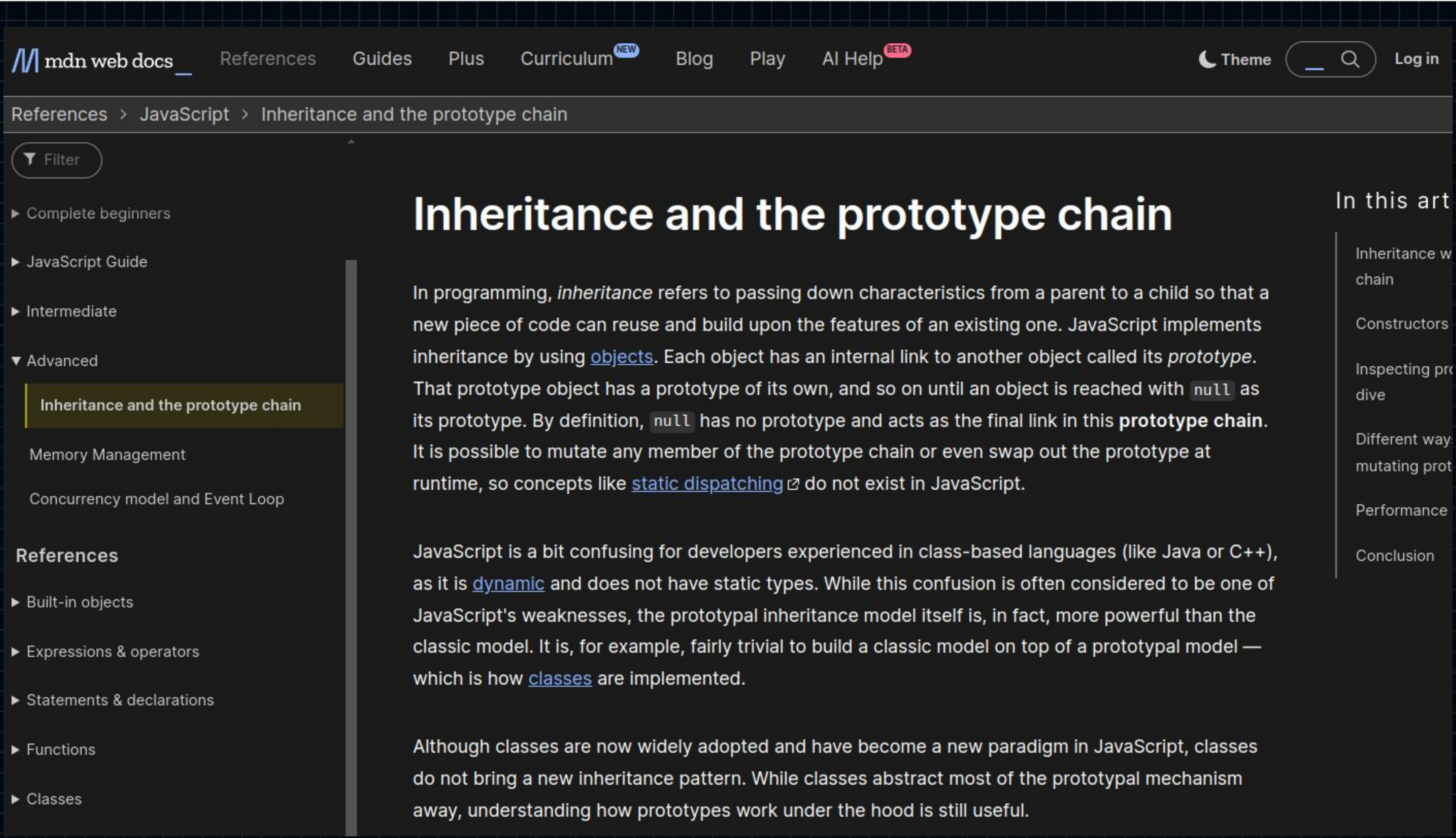
Constructible
Array, Date, Map, Set
WeakMap, WeakSet etc...

Functions

- **Constructible:** function, class
- **Non Constructible:** Arrow, Generators *



```
> next
<· ▼MyConstructor {state: 3} i
      state: 3
    ▼__proto__:
        state: 2
      ▼__proto__:
          state: 1
        ▶__proto__: Object
```

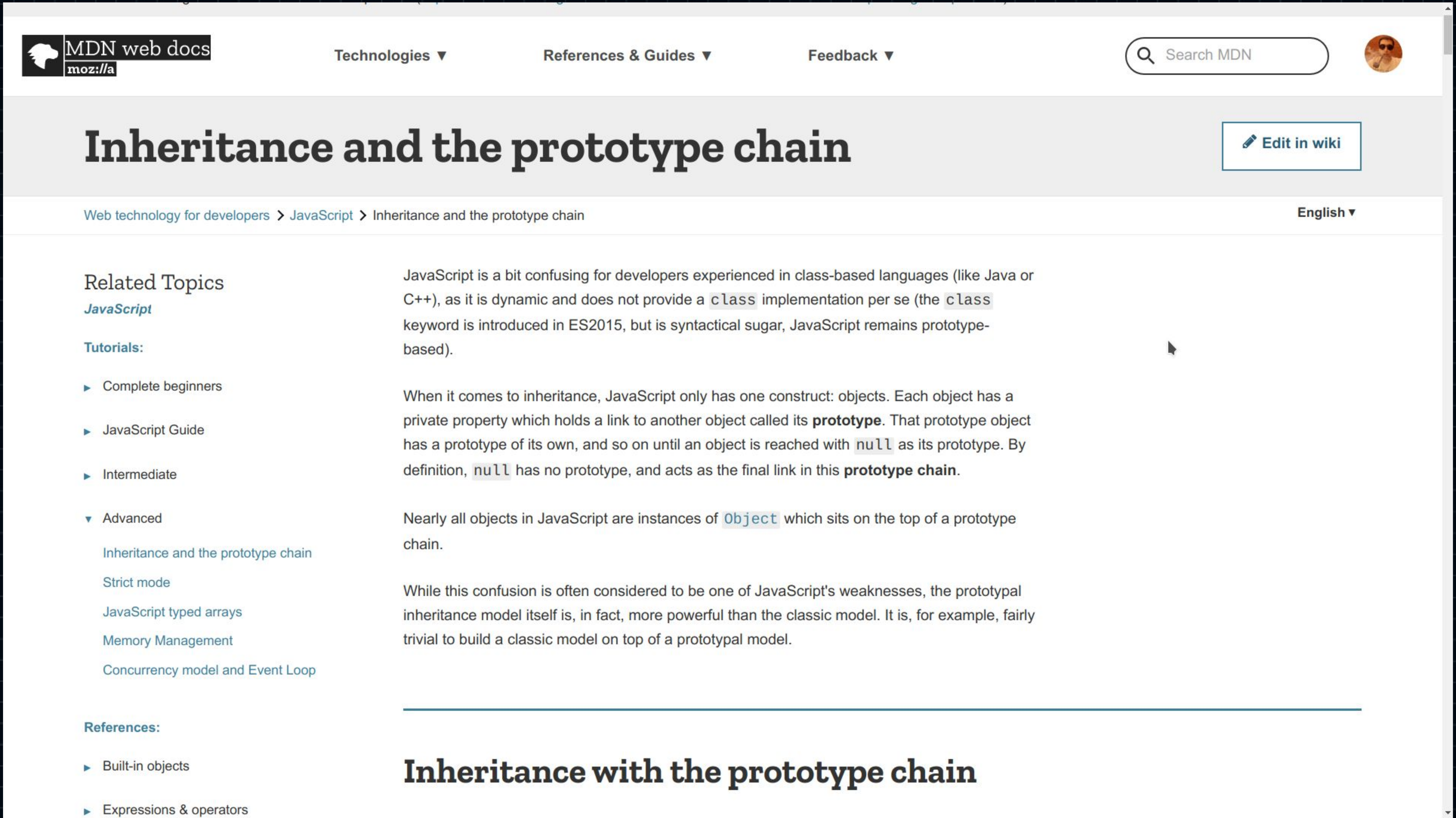


Inheritance and the prototype chain

In programming, *inheritance* refers to passing down characteristics from a parent to a child so that a new piece of code can reuse and build upon the features of an existing one. JavaScript implements inheritance by using objects. Each object has an internal link to another object called its *prototype*. That prototype object has a prototype of its own, and so on until an object is reached with `null` as its prototype. By definition, `null` has no prototype and acts as the final link in this **prototype chain**. It is possible to mutate any member of the prototype chain or even swap out the prototype at runtime, so concepts like static dispatching do not exist in JavaScript.

JavaScript is a bit confusing for developers experienced in class-based languages (like Java or C++), as it is dynamic and does not have static types. While this confusion is often considered to be one of JavaScript's weaknesses, the prototypal inheritance model itself is, in fact, more powerful than the classic model. It is, for example, fairly trivial to build a classic model on top of a prototypal model — which is how classes are implemented.

Although classes are now widely adopted and have become a new paradigm in JavaScript, classes do not bring a new inheritance pattern. While classes abstract most of the prototypal mechanism away, understanding how prototypes work under the hood is still useful.



Inheritance and the prototype chain

Web technology for developers > JavaScript > Inheritance and the prototype chain

Related Topics

JavaScript

Tutorials:

- ▸ Complete beginners
- ▸ JavaScript Guide
- ▸ Intermediate
- ▾ Advanced
  - Inheritance and the prototype chain
  - Strict mode
  - JavaScript typed arrays
  - Memory Management
  - Concurrency model and Event Loop

References:

- ▸ Built-in objects
- ▸ Expressions & operators

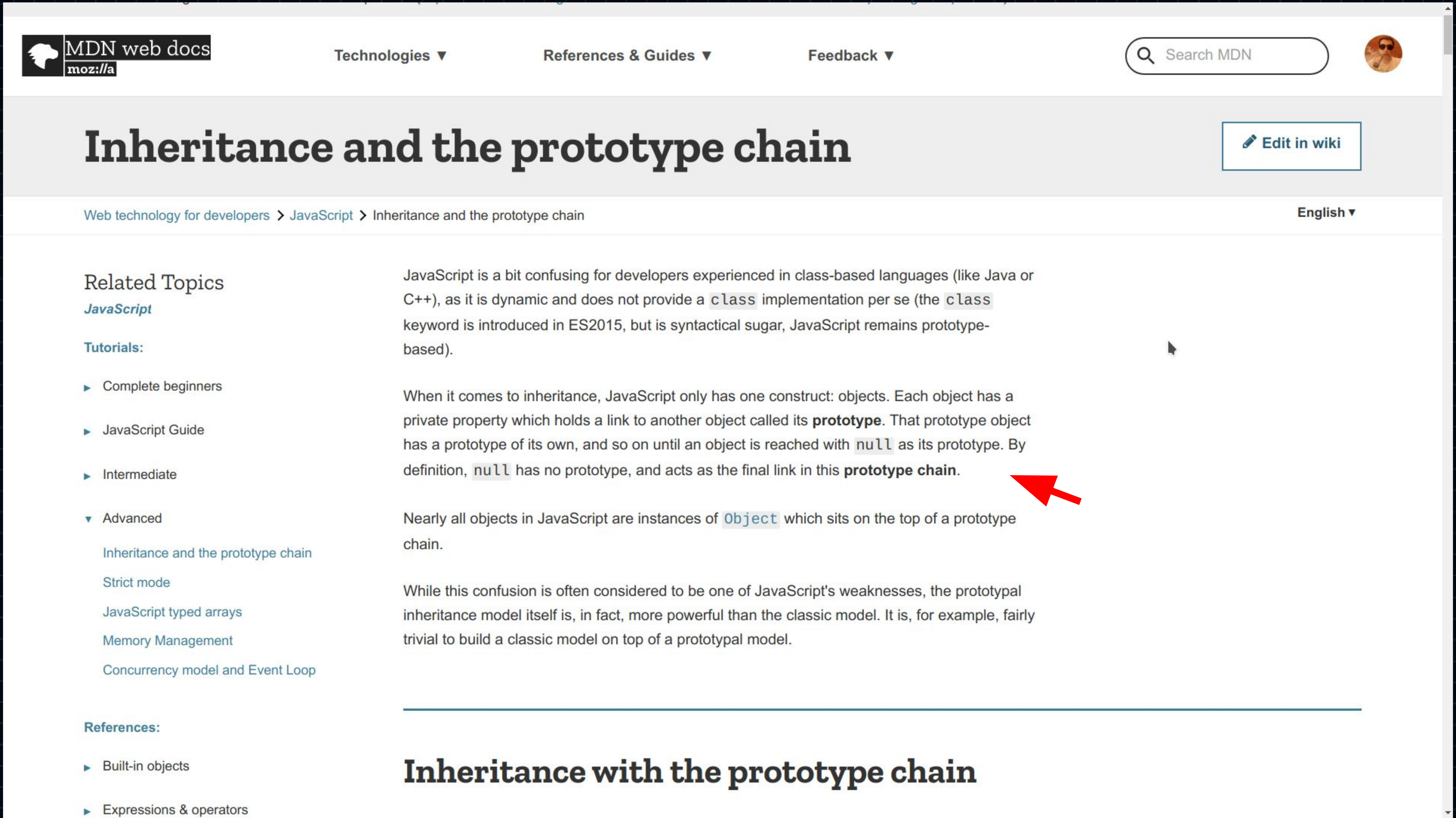
JavaScript is a bit confusing for developers experienced in class-based languages (like Java or C++), as it is dynamic and does not provide a `class` implementation per se (the `class` keyword is introduced in ES2015, but is syntactical sugar, JavaScript remains prototype-based).

When it comes to inheritance, JavaScript only has one construct: objects. Each object has a private property which holds a link to another object called its **prototype**. That prototype object has a prototype of its own, and so on until an object is reached with `null` as its prototype. By definition, `null` has no prototype, and acts as the final link in this **prototype chain**.

Nearly all objects in JavaScript are instances of `Object` which sits on the top of a prototype chain.

While this confusion is often considered to be one of JavaScript's weaknesses, the prototypal inheritance model itself is, in fact, more powerful than the classic model. It is, for example, fairly trivial to build a classic model on top of a prototypal model.

Inheritance with the prototype chain



Inheritance and the prototype chain

JavaScript is a bit confusing for developers experienced in class-based languages (like Java or C++), as it is dynamic and does not provide a `class` implementation per se (the `class` keyword is introduced in ES2015, but is syntactical sugar, JavaScript remains prototype-based).

When it comes to inheritance, JavaScript only has one construct: objects. Each object has a private property which holds a link to another object called its **prototype**. That prototype object has a prototype of its own, and so on until an object is reached with `null` as its prototype. By definition, `null` has no prototype, and acts as the final link in this **prototype chain**.

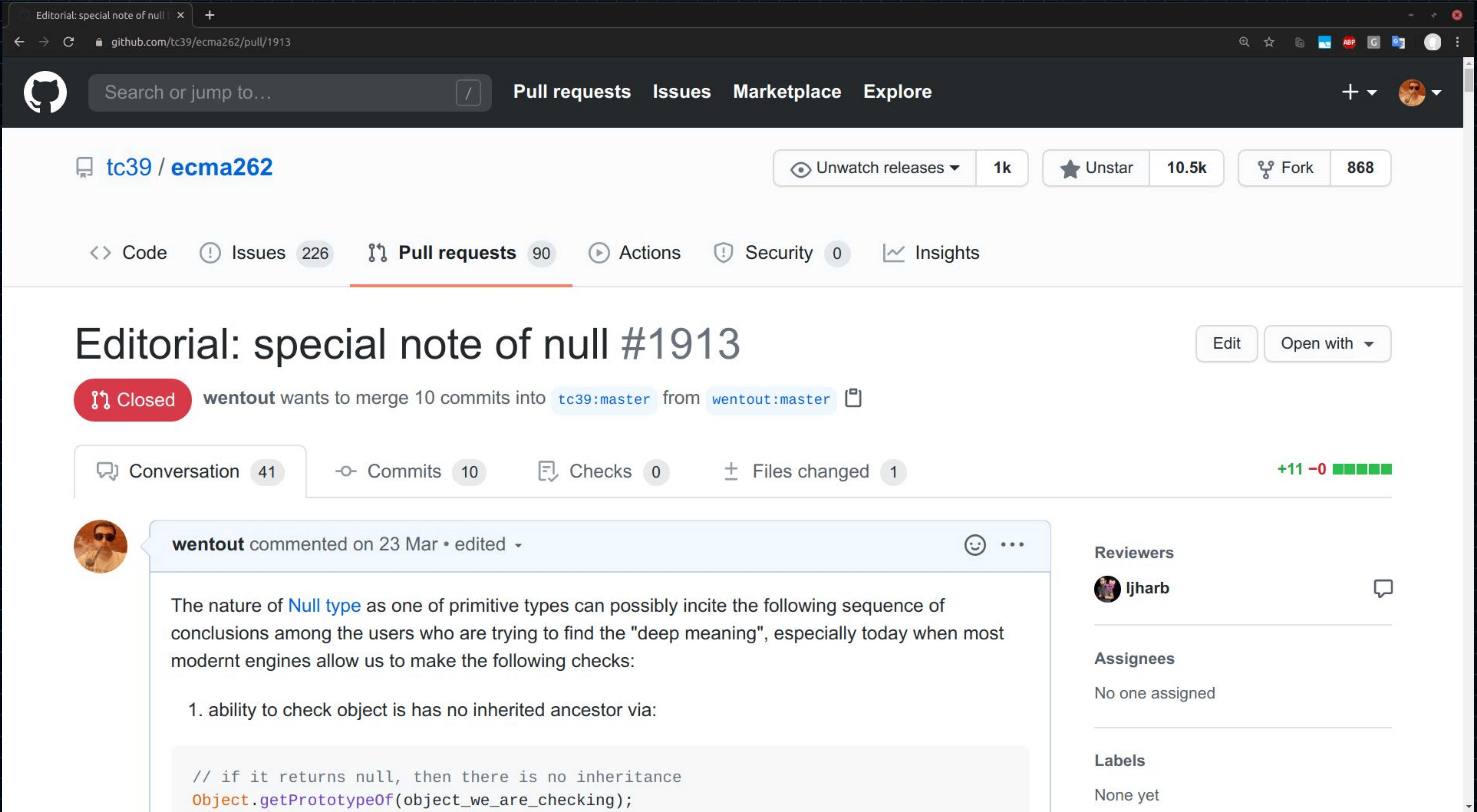
Nearly all objects in JavaScript are instances of `Object` which sits on the top of a prototype chain.

While this confusion is often considered to be one of JavaScript's weaknesses, the prototypal inheritance model itself is, in fact, more powerful than the classic model. It is, for example, fairly trivial to build a classic model on top of a prototypal model.

Inheritance with the prototype chain

typeof null is also good

to my apologies to Brendan Eich



Editorial: special note of null #1913

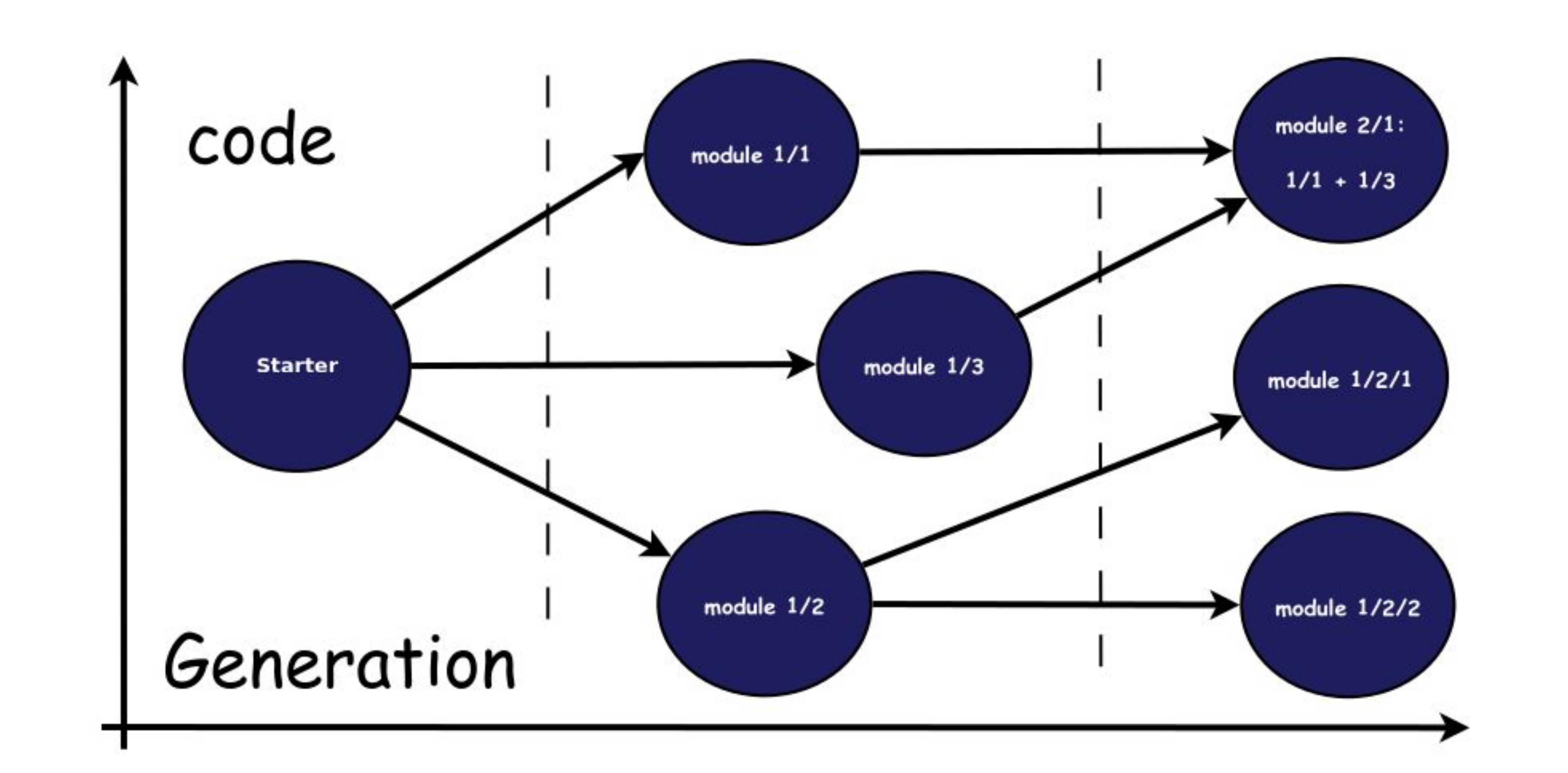
Closed **wentout** wants to merge 10 commits into `tc39:master` from `wentout:master`

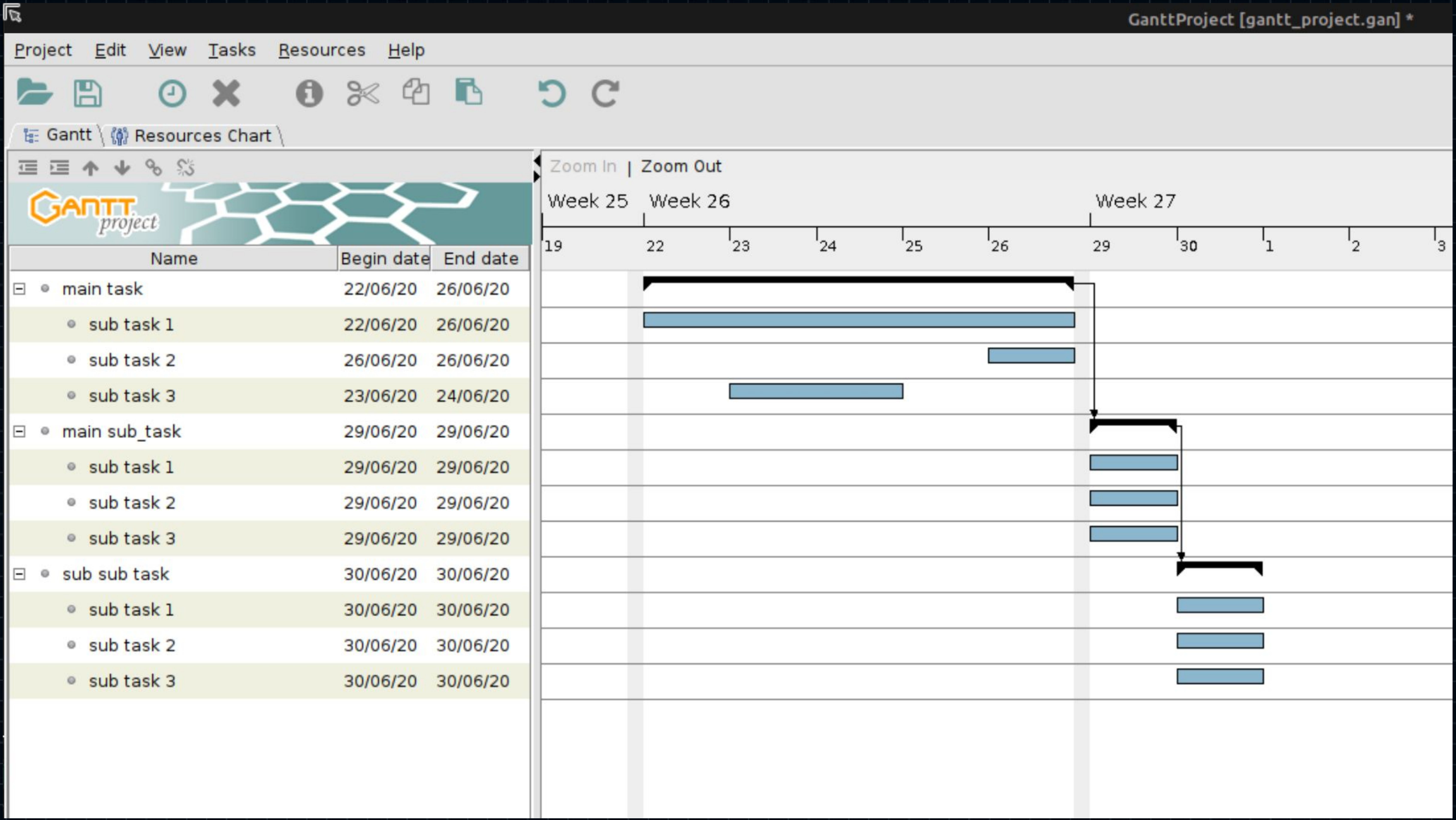
wentout commented on 23 Mar • edited

The nature of Null type as one of primitive types can possibly incite the following sequence of conclusions among the users who are trying to find the "deep meaning", especially today when most modernt engines allow us to make the following checks:

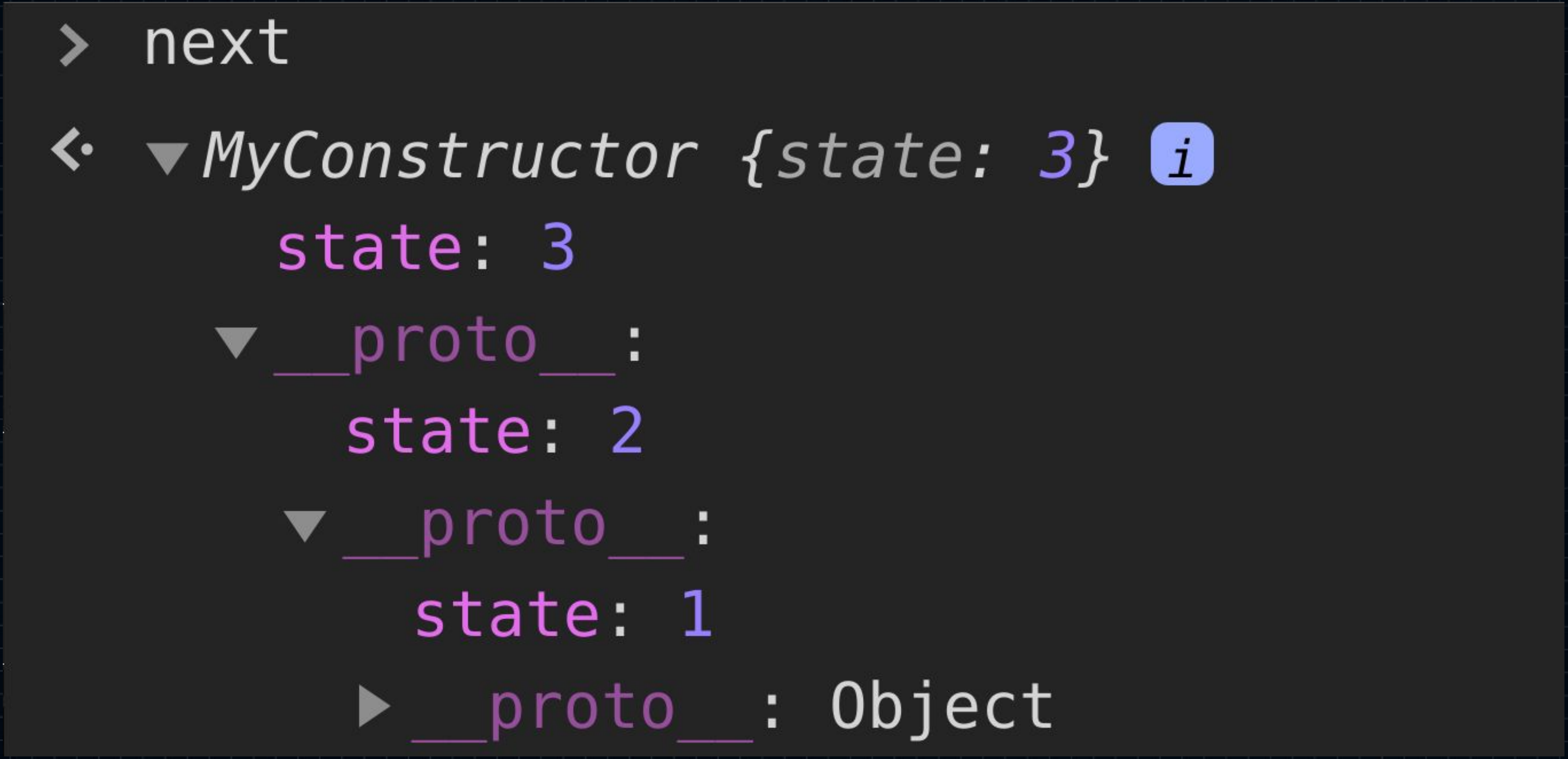
1. ability to check object is has no inherited ancestor via:

```
// if it returns null, then there is no inheritance
Object.getPrototypeOf(object_we_are_checking);
```

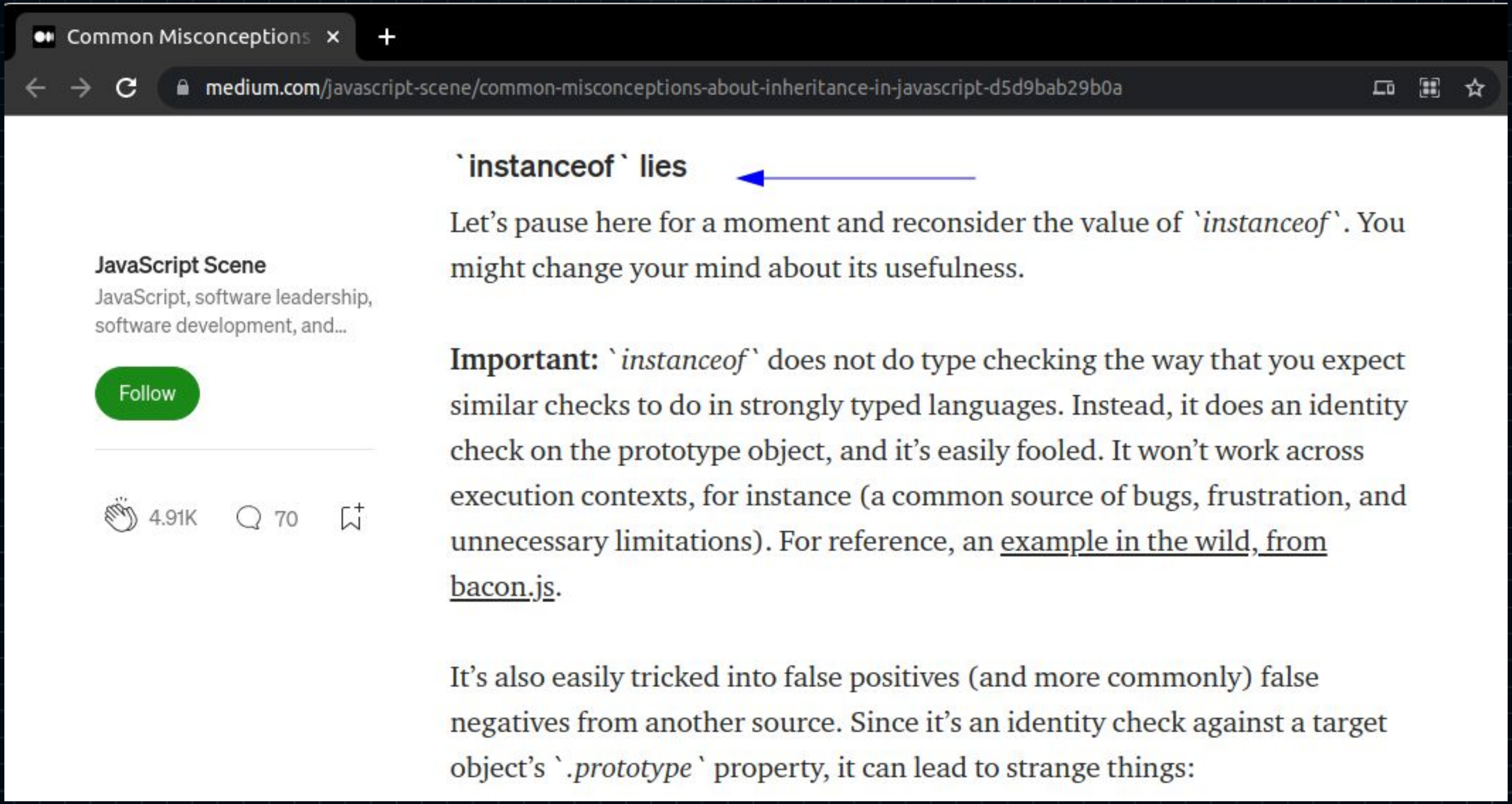

code
Generation
Starter
module 1/1
module 1/3
module 1/2
module 2/1:
1/1 + 1/3
module 1/2/1
module 1/2/2



Name	Begin date	End date
⊟ main task	22/06/20	26/06/20
sub task 1	22/06/20	26/06/20
sub task 2	26/06/20	26/06/20
sub task 3	23/06/20	24/06/20
⊟ main sub_task	29/06/20	29/06/20
sub task 1	29/06/20	29/06/20
sub task 2	29/06/20	29/06/20
sub task 3	29/06/20	29/06/20
⊟ sub sub task	30/06/20	30/06/20
sub task 1	30/06/20	30/06/20
sub task 2	30/06/20	30/06/20
sub task 3	30/06/20	30/06/20



```
> next
<· ▼MyConstructor {state: 3} i
      state: 3
    ▼__proto__:
        state: 2
      ▼__proto__:
          state: 1
        ▶__proto__: Object
```

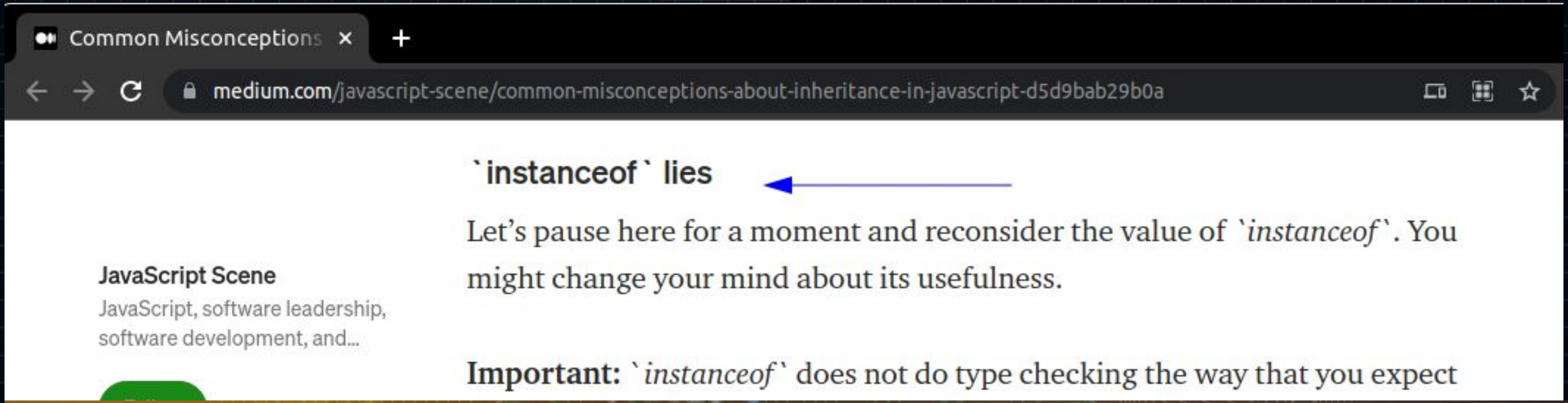

Common Misconceptions
medium.com/javascript-scene/common-misconceptions-about-inheritance-in-javascript-d5d9bab29b0a
JavaScript Scene
JavaScript, software leadership, software development, and...
Follow
4.91K
70

`instanceof` lies

Let's pause here for a moment and reconsider the value of `*instanceof*`. You might change your mind about its usefulness.

Important: `*instanceof*` does not do type checking the way that you expect similar checks to do in strongly typed languages. Instead, it does an identity check on the prototype object, and it's easily fooled. It won't work across execution contexts, for instance (a common source of bugs, frustration, and unnecessary limitations). For reference, an example in the wild, from bacon.js.

It's also easily tricked into false positives (and more commonly) false negatives from another source. Since it's an identity check against a target object's `*.prototype*` property, it can lead to strange things:



Common Misconceptions
medium.com/javascript-scene/common-misconceptions-about-inheritance-in-javascript-d5d9bab29b0a

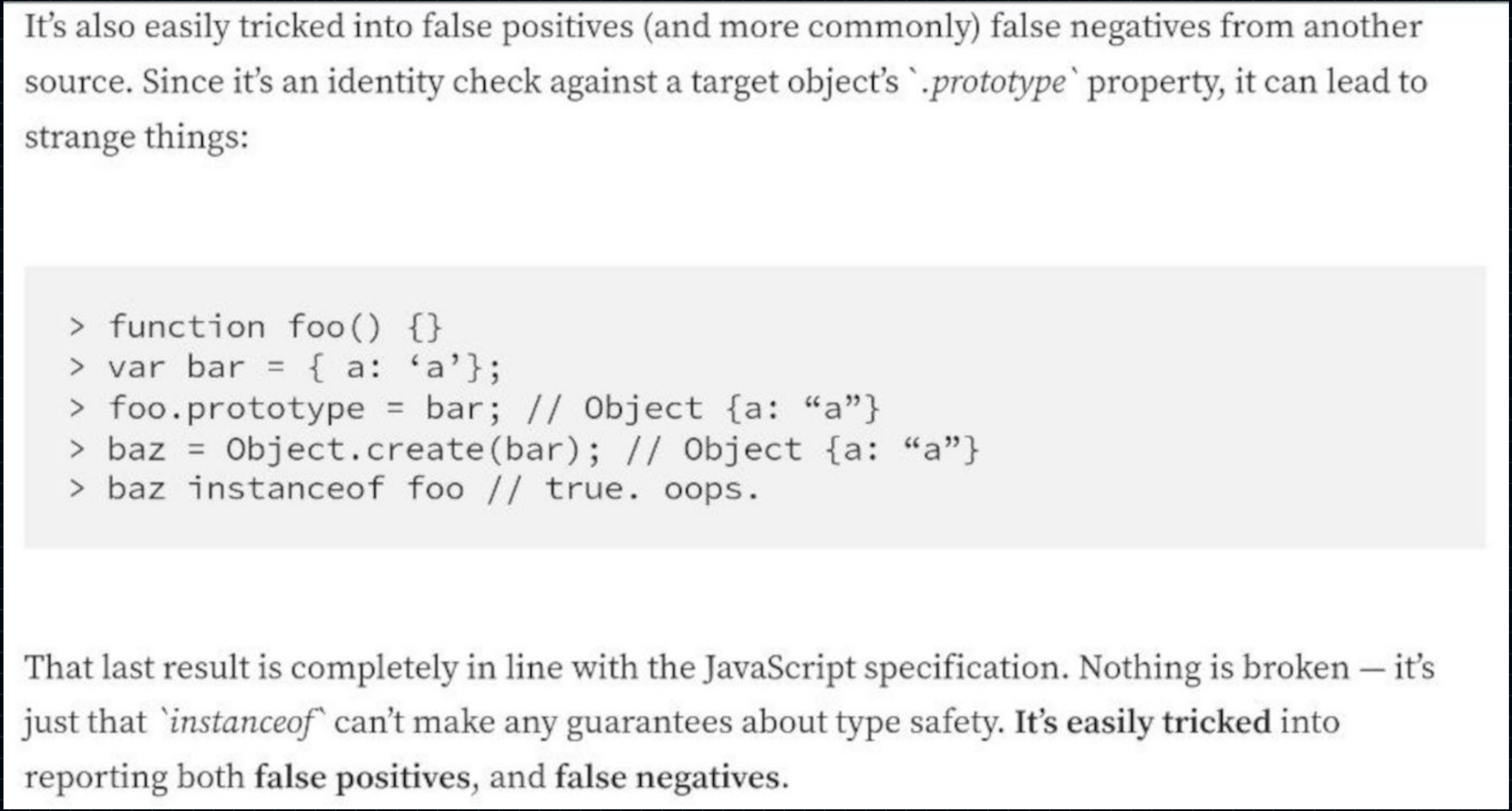
`instanceof` lies

JavaScript Scene

JavaScript, software leadership, software development, and...

Let's pause here for a moment and reconsider the value of `*instanceof*`. You might change your mind about its usefulness.

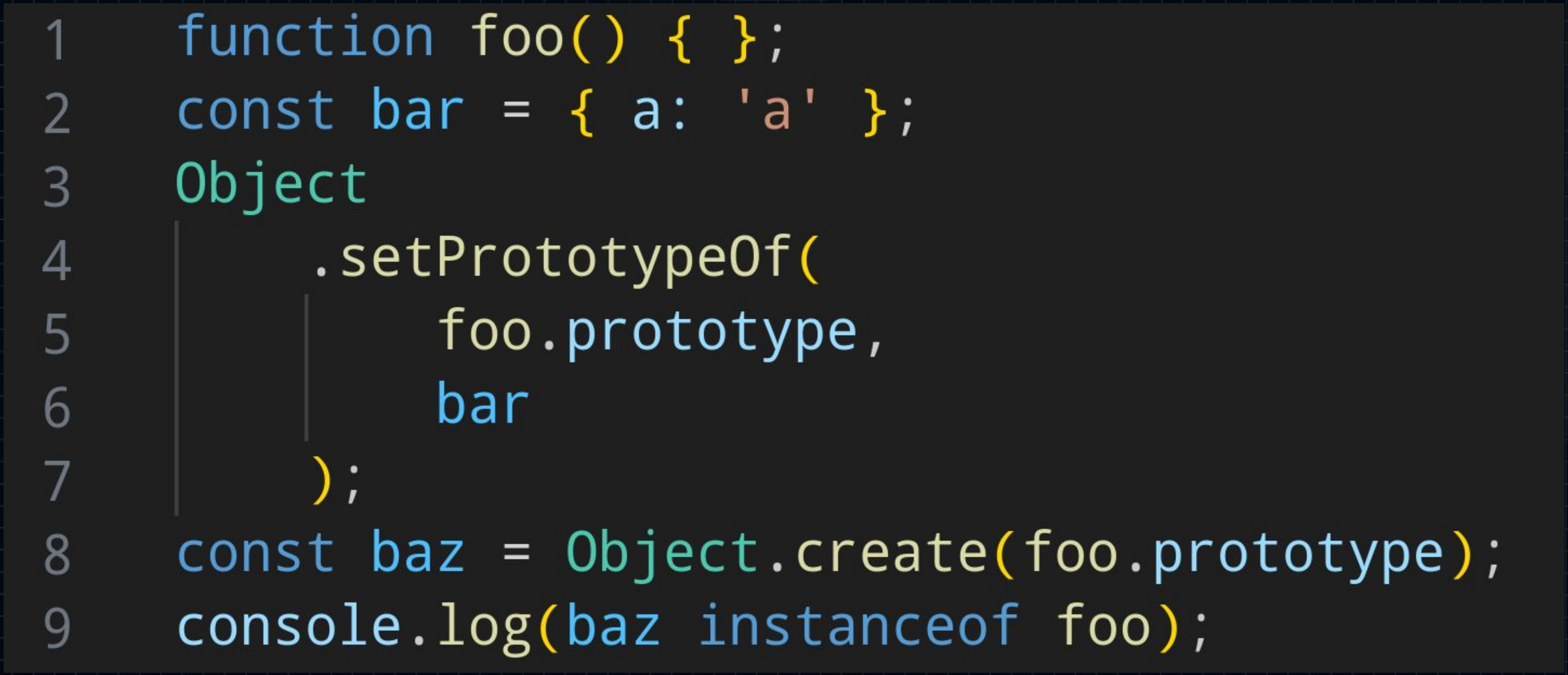
Important: `*instanceof*` does not do type checking the way that you expect



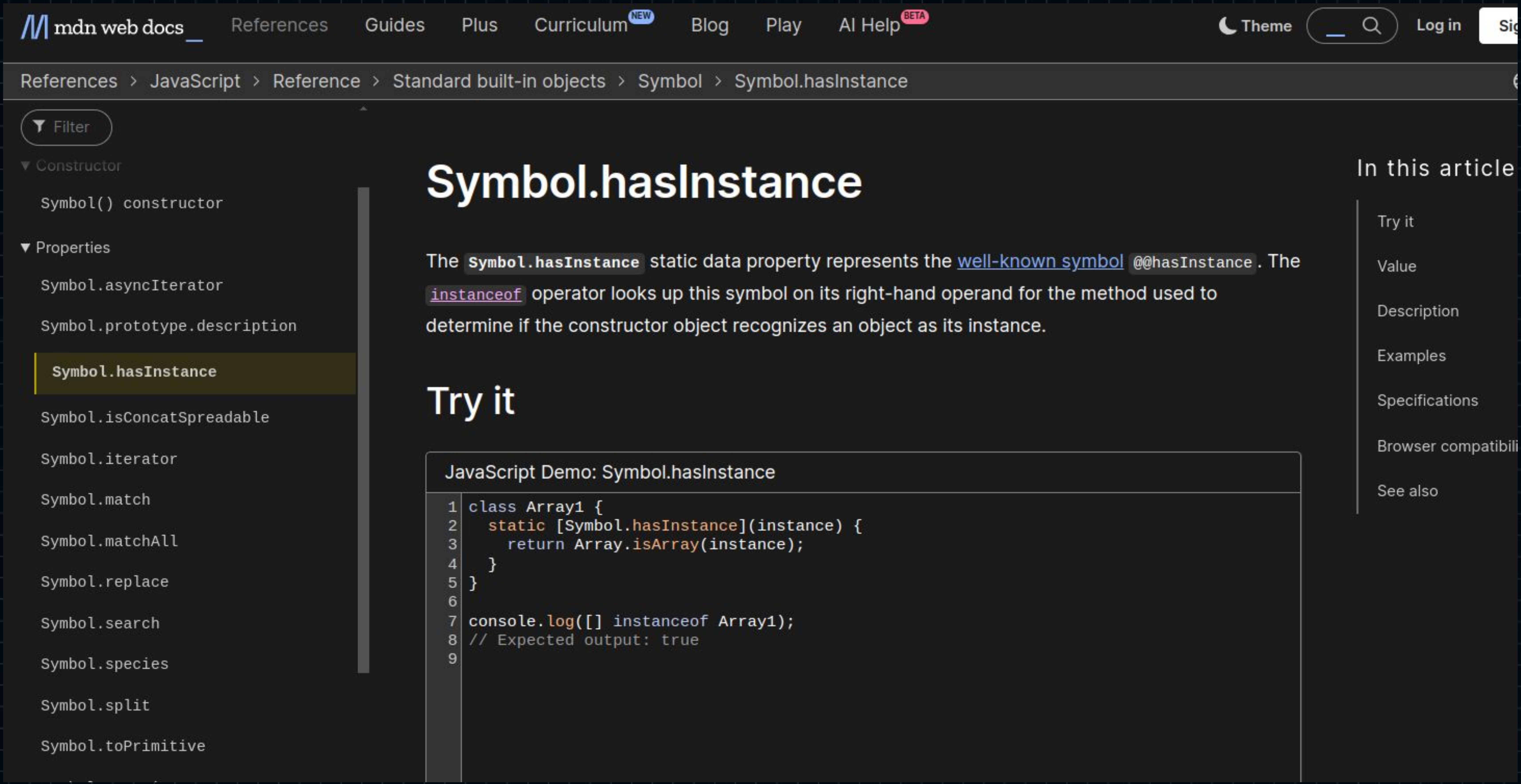
It’s also easily tricked into false positives (and more commonly) false negatives from another source. Since it’s an identity check against a target object’s `*.prototype*` property, it can lead to strange things:

```
> function foo() {}
> var bar = { a: ‘a’};
> foo.prototype = bar; // Object {a: “a”}
> baz = Object.create(bar); // Object {a: “a”}
> baz instanceof foo // true. oops.
```

That last result is completely in line with the JavaScript specification. Nothing is broken — it’s just that `*instanceof*` can’t make any guarantees about type safety. **It’s easily tricked** into reporting both **false positives**, and **false negatives.**



```javascript
function foo() { };
const bar = { a: 'a' };
Object
    .setPrototypeOf(
        foo.prototype,
        bar
    );
const baz = Object.create(foo.prototype);
console.log(baz instanceof foo);
```



Symbol.hasInstance

The `Symbol.hasInstance` static data property represents the well-known symbol `@@hasInstance`. The `instanceof` operator looks up this symbol on its right-hand operand for the method used to determine if the constructor object recognizes an object as its instance.

Try it

JavaScript Demo: Symbol.hasInstance

```
class Array1 {
  static [Symbol.hasInstance](instance) {
    return Array.isArray(instance);
  }
}

console.log([] instanceof Array1);
// Expected output: true

```

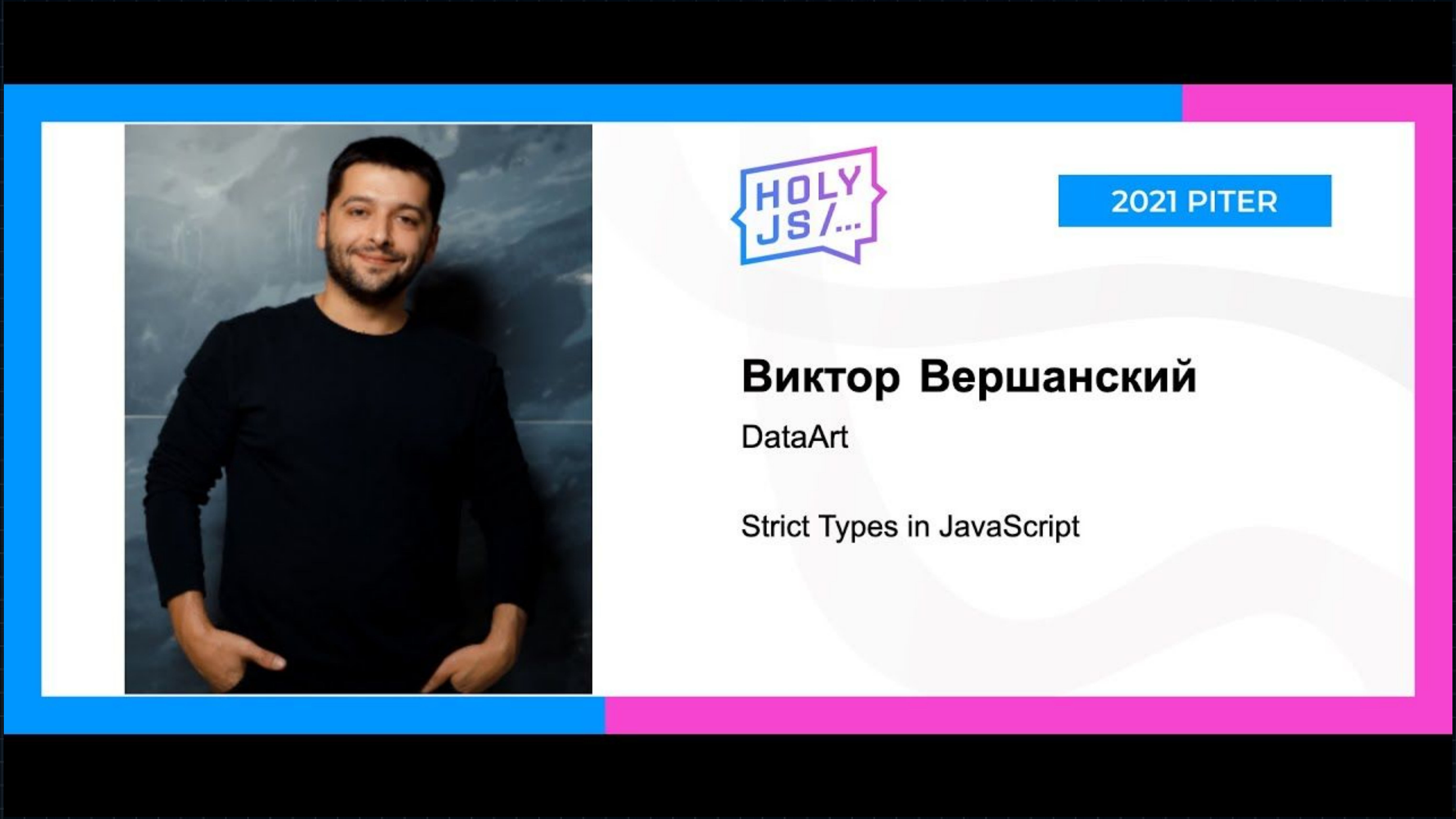
решения

решения

4

2021

Строгая типизация в JavaScript



HOLY JS/...

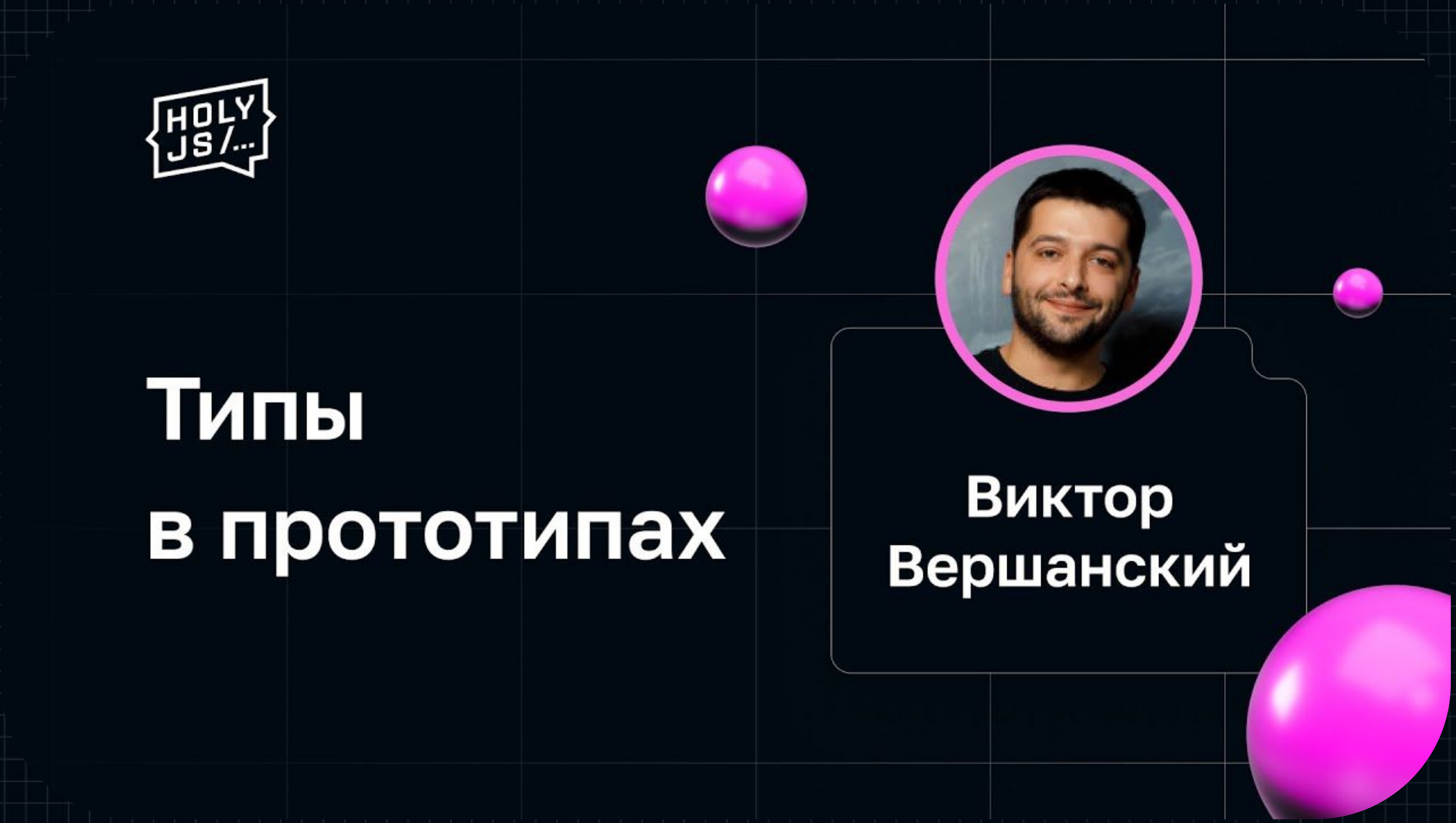
2021 PITER

Виктор Вершанский

DataArt

Strict Types in JavaScript

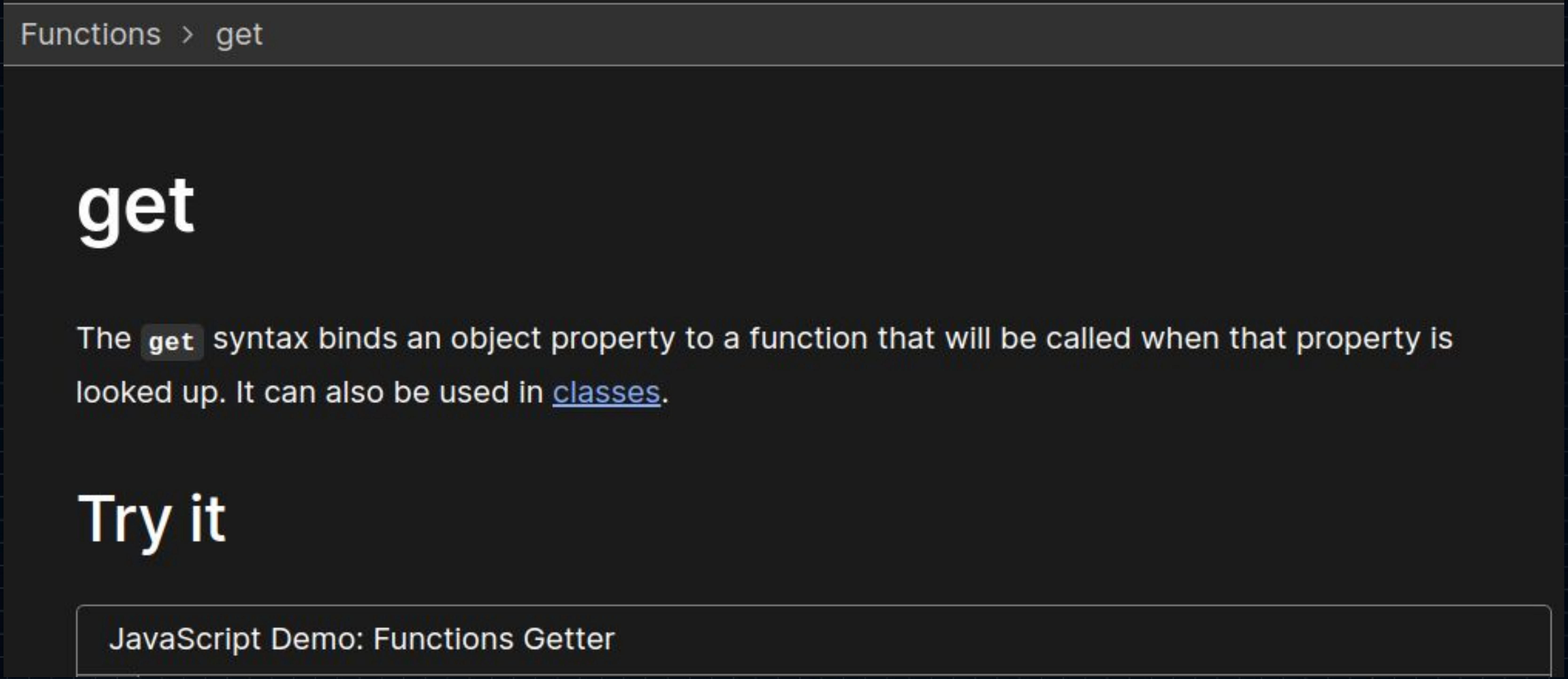
2023



Типы в прототипах

Виктор Вершанский

на чём это сделано



Functions › get

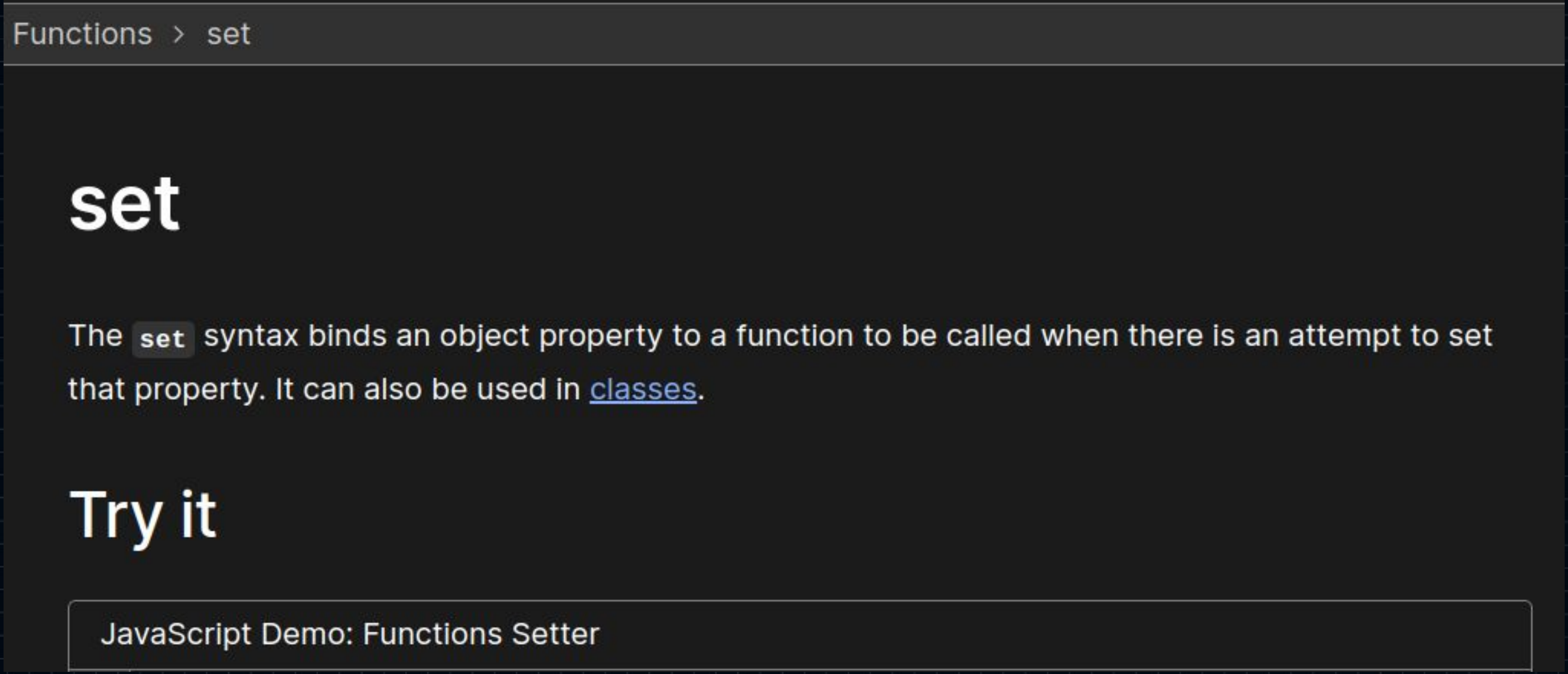
get

The `get` syntax binds an object property to a function that will be called when that property is looked up. It can also be used in classes.

Try it

JavaScript Demo: Functions Getter

на чём это сделано



Functions › set

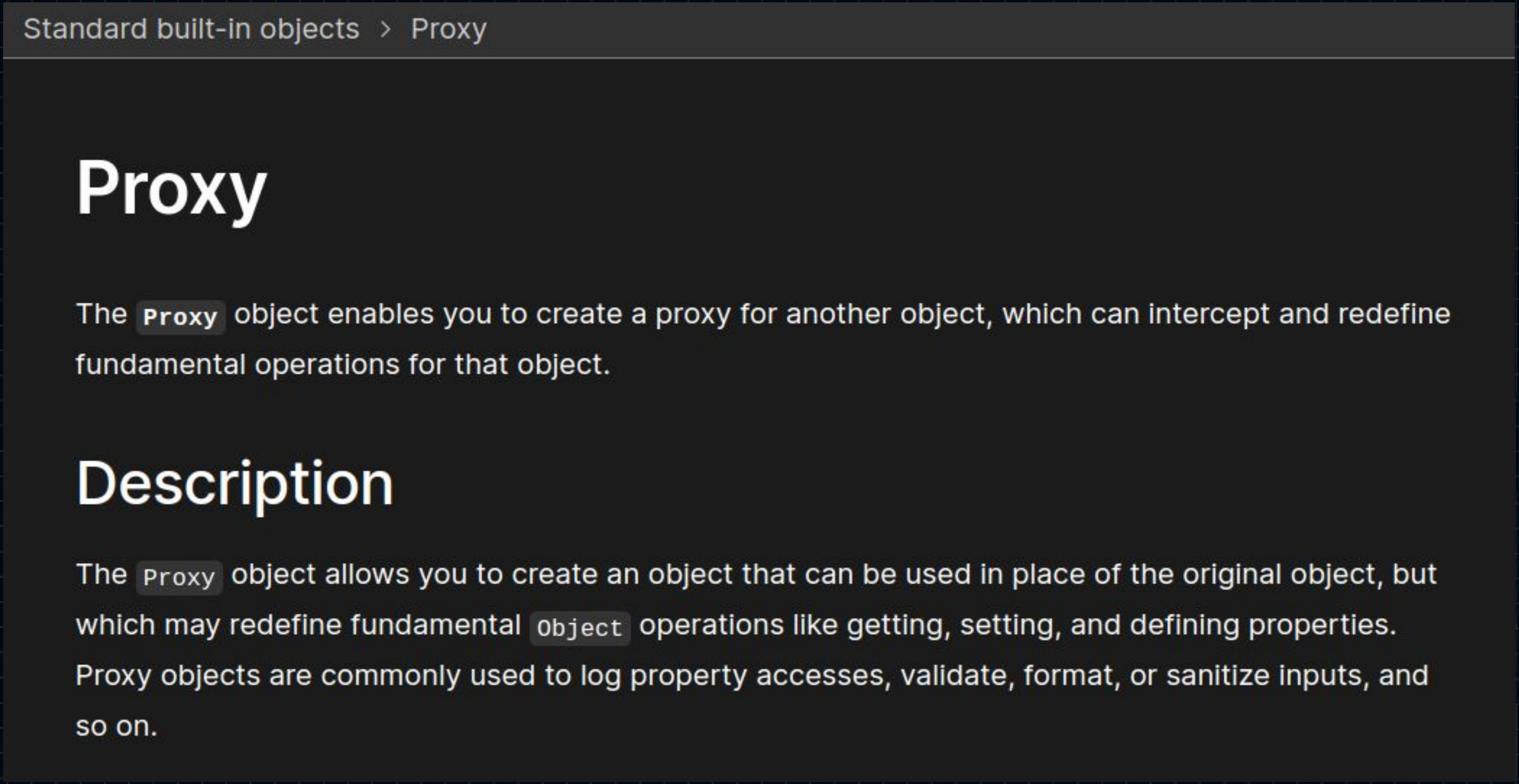
set

The `set` syntax binds an object property to a function to be called when there is an attempt to set that property. It can also be used in classes.

Try it

JavaScript Demo: Functions Setter

на чём это сделано



Standard built-in objects › Proxy

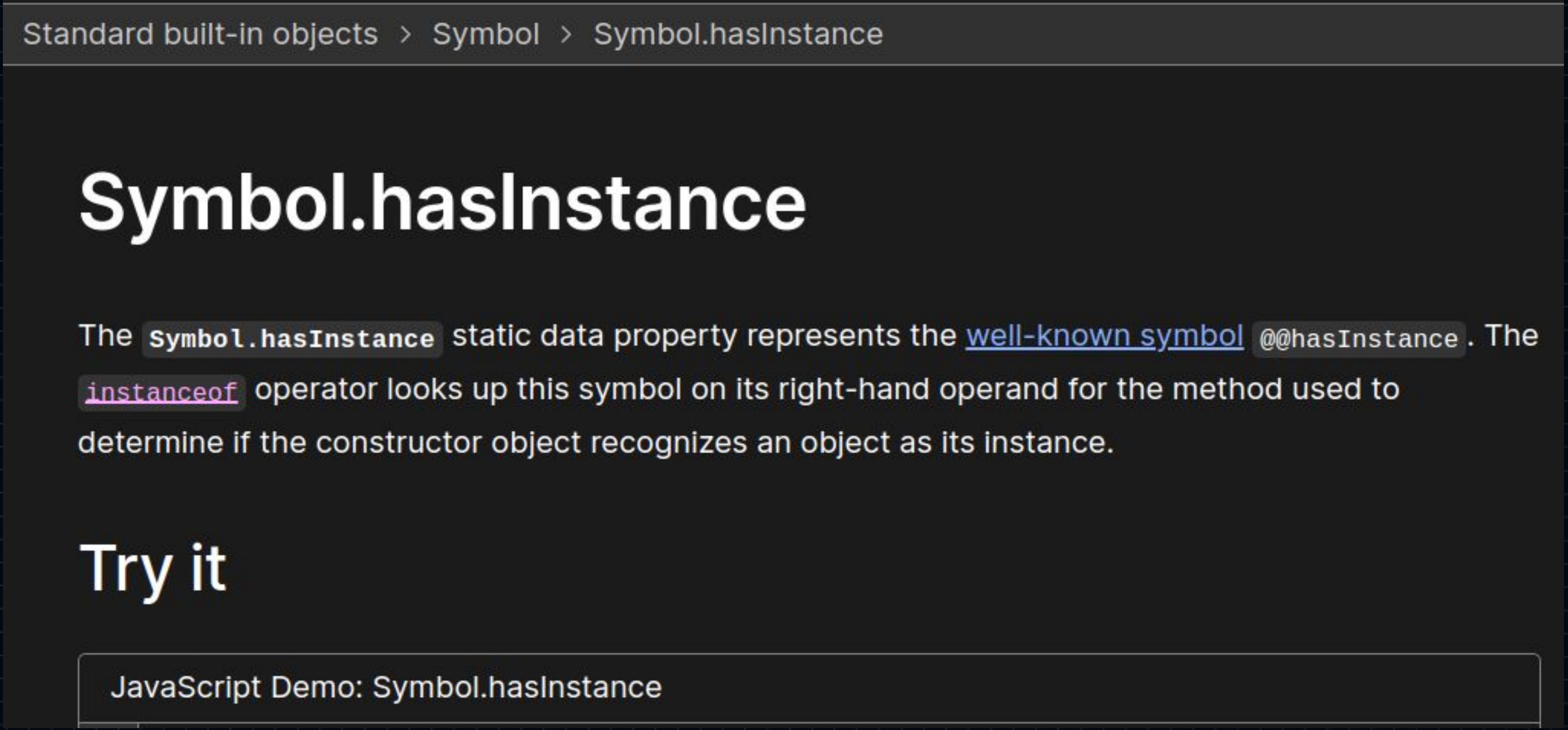
Proxy

The `Proxy` object enables you to create a proxy for another object, which can intercept and redefine fundamental operations for that object.

Description

The `Proxy` object allows you to create an object that can be used in place of the original object, but which may redefine fundamental `Object` operations like getting, setting, and defining properties. Proxy objects are commonly used to log property accesses, validate, format, or sanitize inputs, and so on.

на чём это сделано



Standard built-in objects › Symbol › Symbol.hasInstance

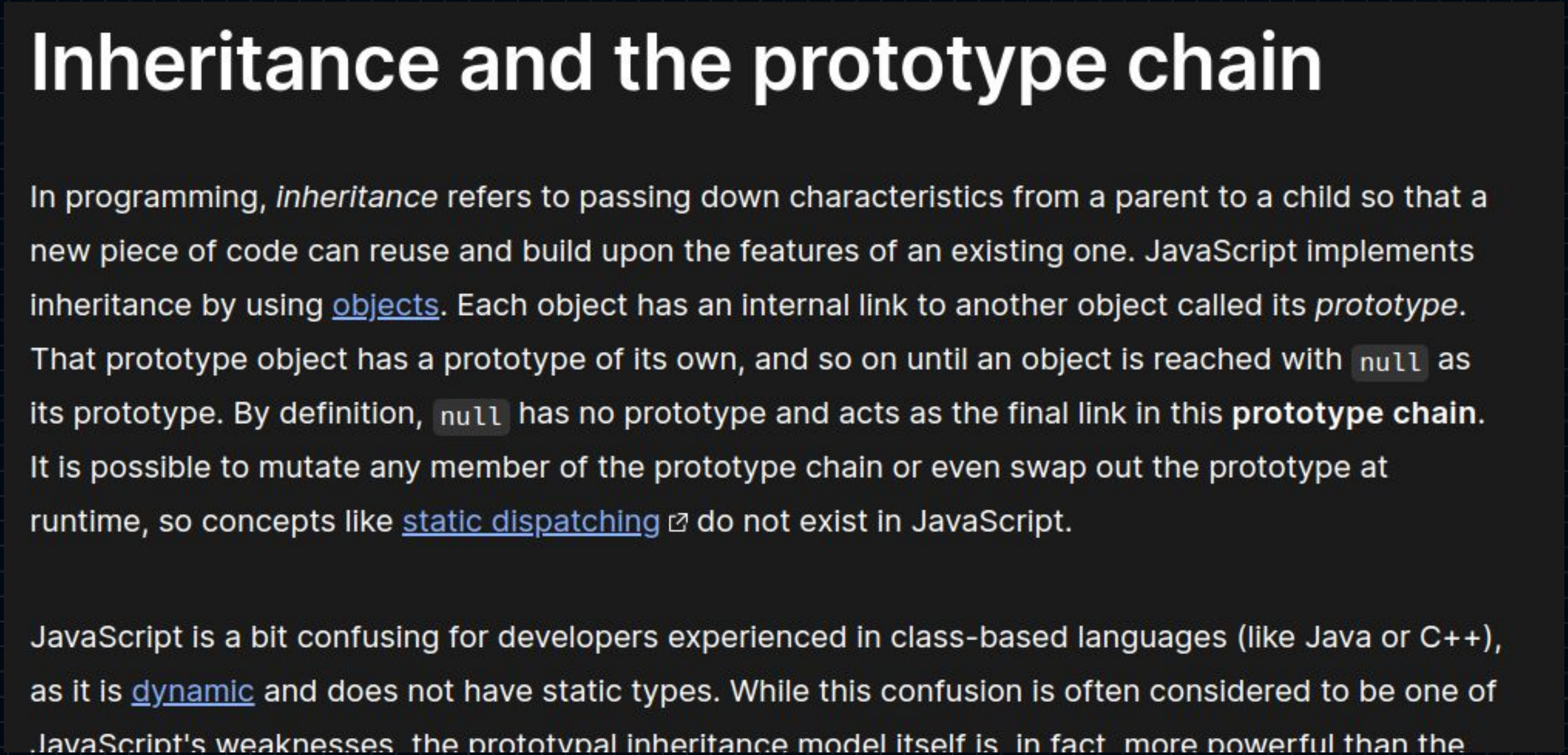
Symbol.hasInstance

The `Symbol.hasInstance` static data property represents the well-known symbol `@@hasInstance`. The `instanceof` operator looks up this symbol on its right-hand operand for the method used to determine if the constructor object recognizes an object as its instance.

Try it

JavaScript Demo: Symbol.hasInstance

на чём это сделано



Inheritance and the prototype chain

In programming, *inheritance* refers to passing down characteristics from a parent to a child so that a new piece of code can reuse and build upon the features of an existing one. JavaScript implements inheritance by using objects. Each object has an internal link to another object called its *prototype*. That prototype object has a prototype of its own, and so on until an object is reached with `null` as its prototype. By definition, `null` has no prototype and acts as the final link in this **prototype chain**. It is possible to mutate any member of the prototype chain or even swap out the prototype at runtime, so concepts like static dispatching do not exist in JavaScript.

JavaScript is a bit confusing for developers experienced in class-based languages (like Java or C++), as it is dynamic and does not have static types. While this confusion is often considered to be one of JavaScript's weaknesses, the prototypal inheritance model itself is, in fact, more powerful than the

примеры

примеры

5

04_Decorator.ts

05_NextStep.js

выводы

- Prototype Chain
- getter’ы + setter’ы
- Proxy + Symbol.hasInstance

... и немножко магии ...

Спасибо !