

JavaScript как конструктор безопасного языка



**Виктор
Вершанский**

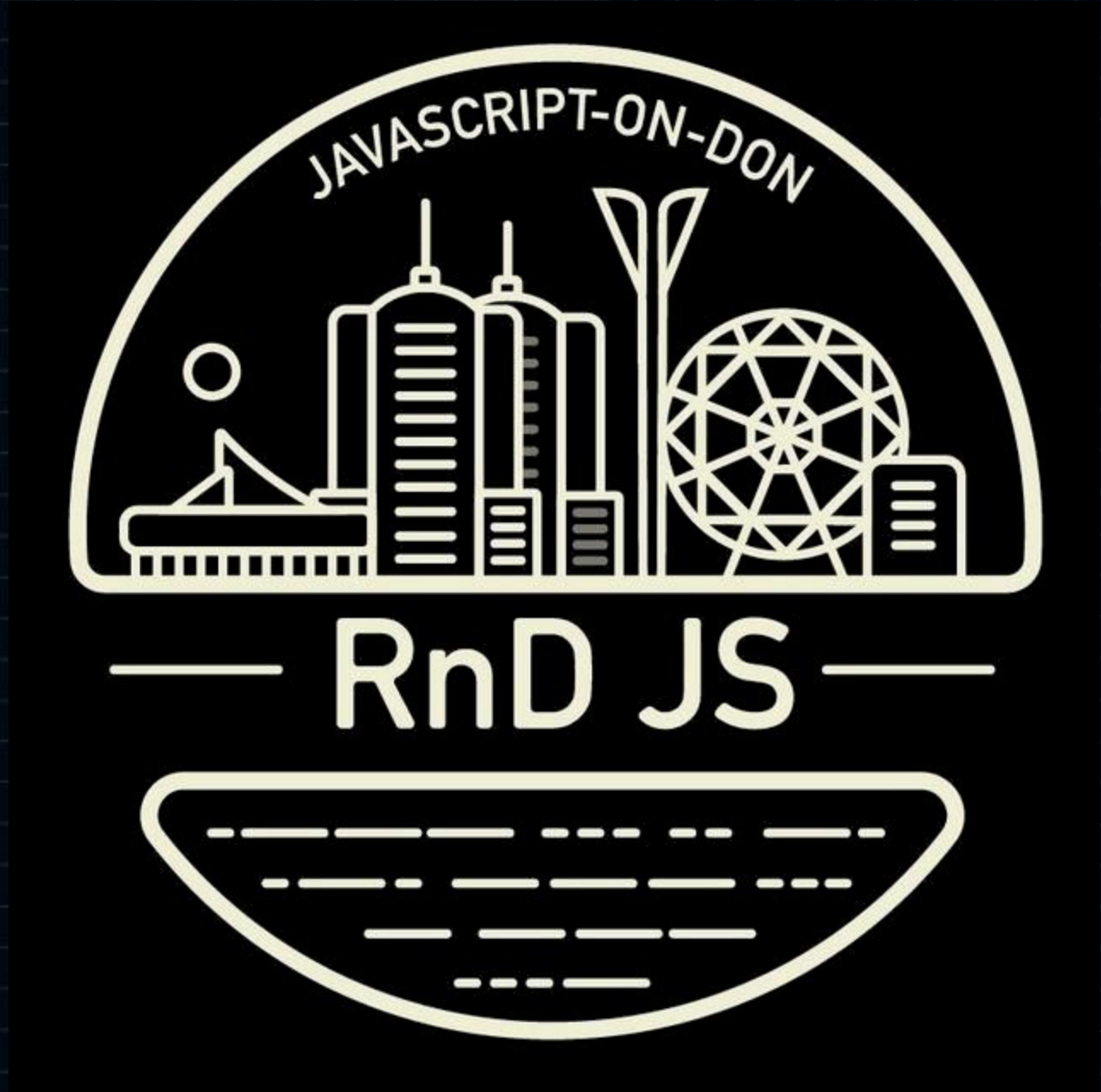
 wentout

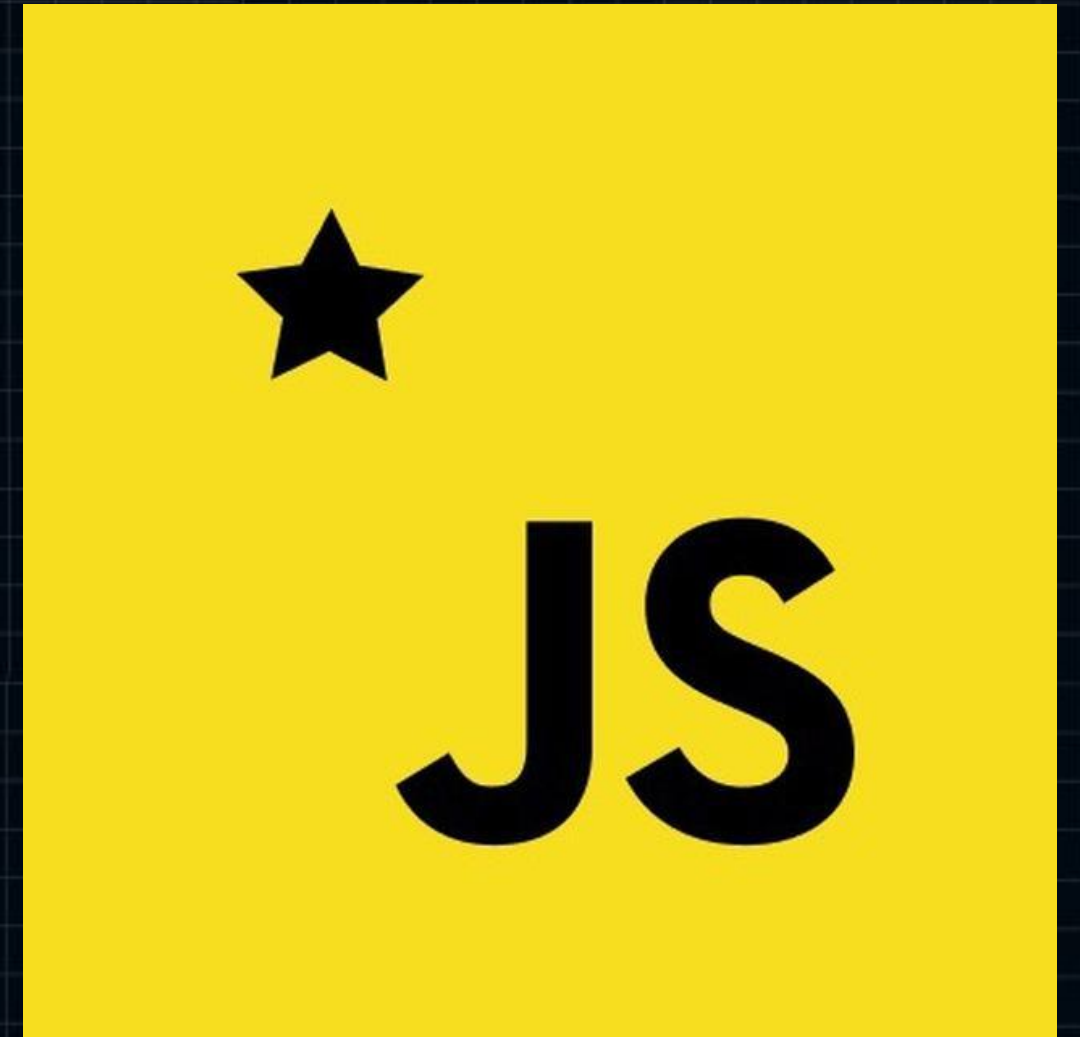
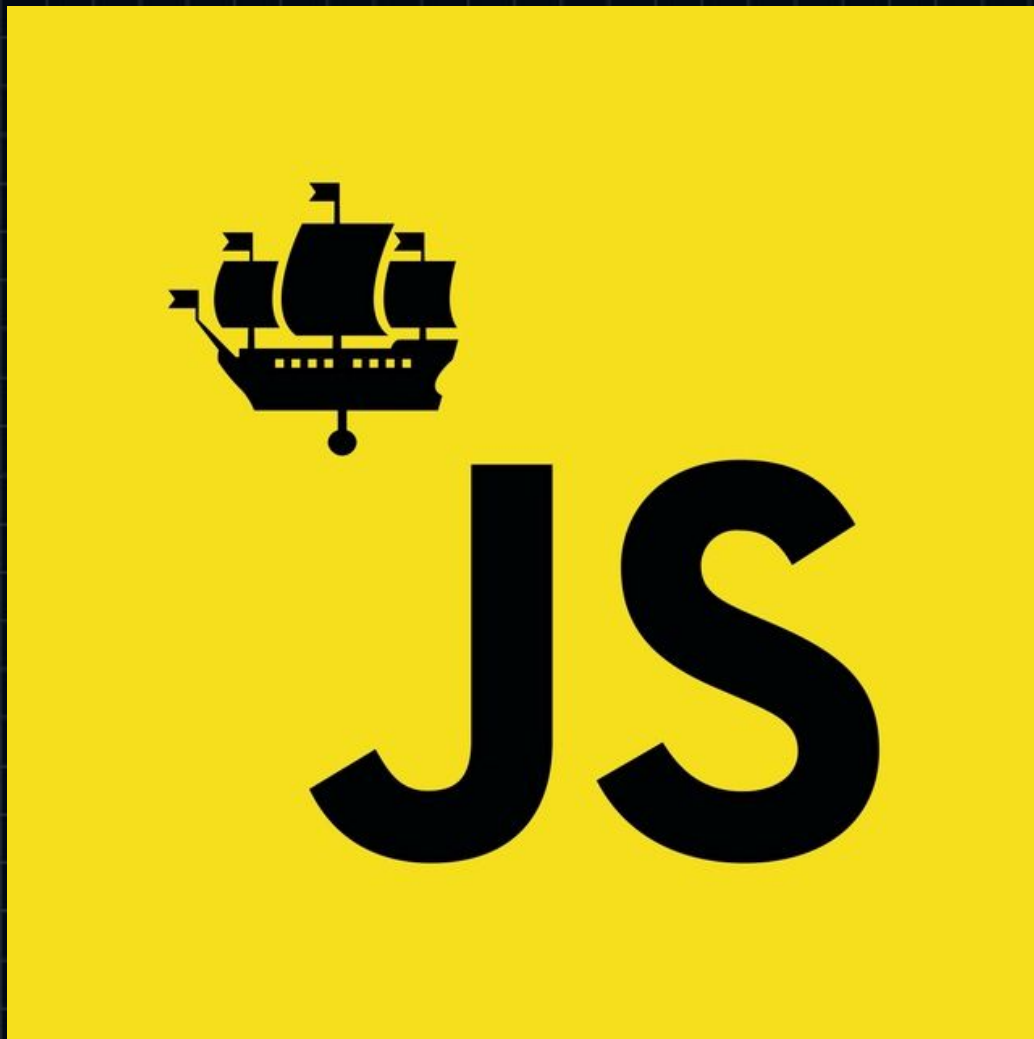


SafeCode

2024









Виктор

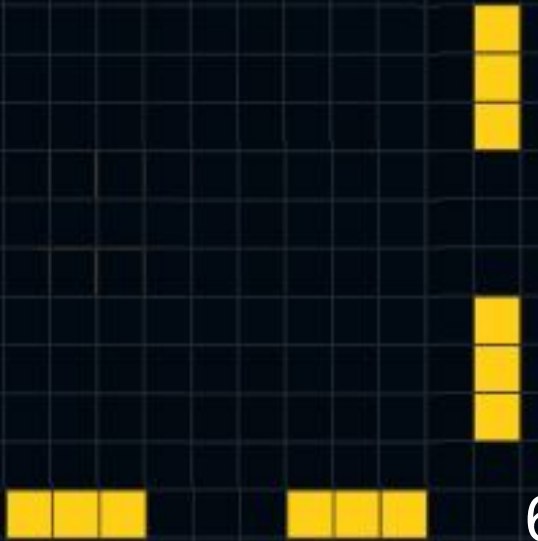
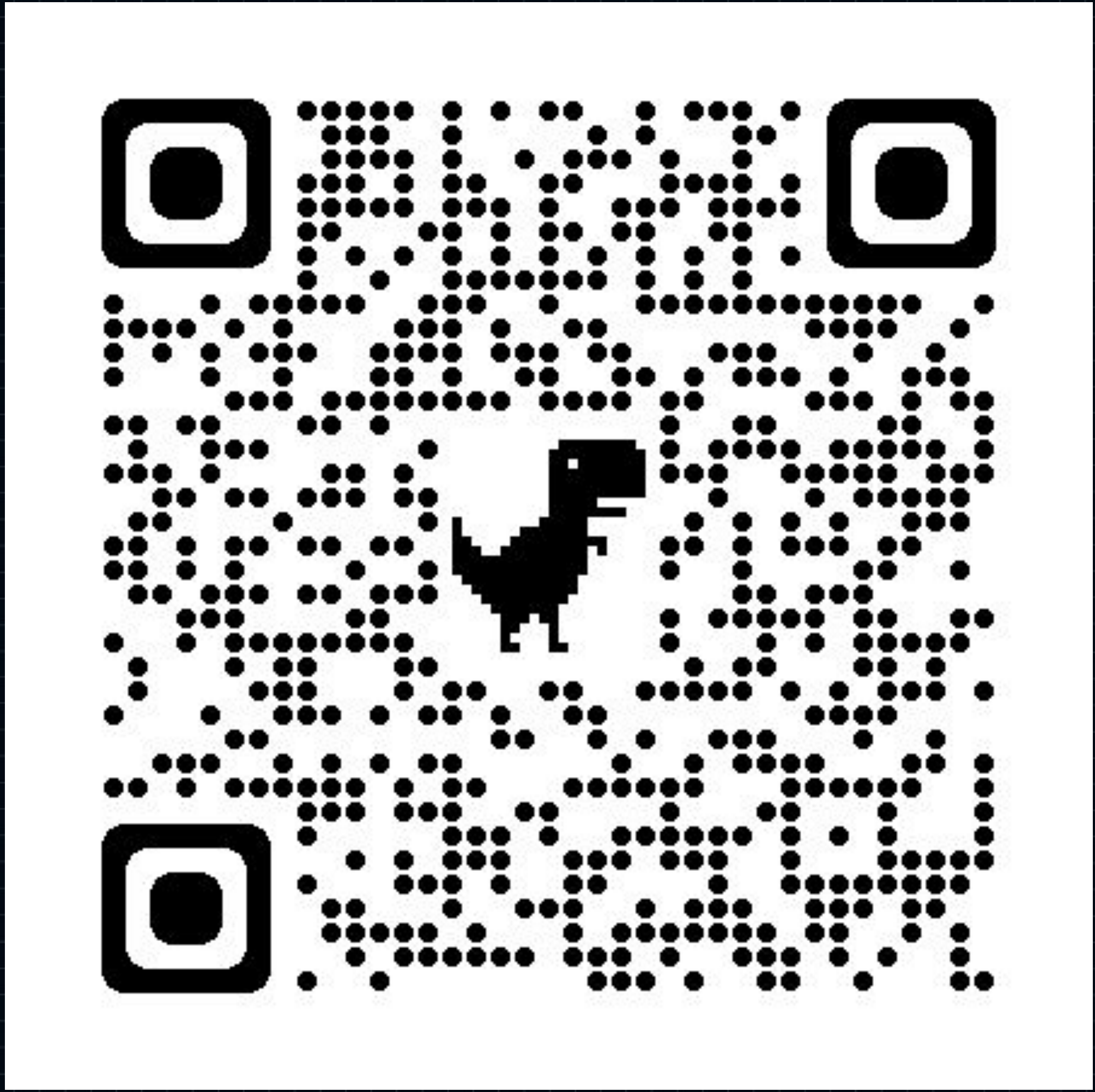


wentout

Bio

- 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

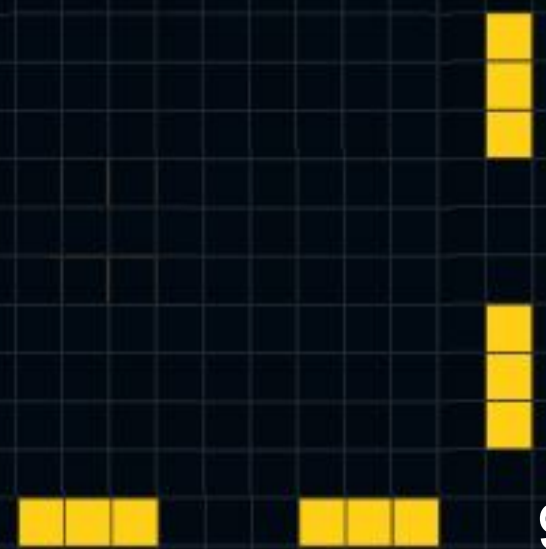
- контекст постановки задачи
- формулировка проблематики
- аспекты, специфичные в JS
- как могут выглядеть решения
- практические примеры



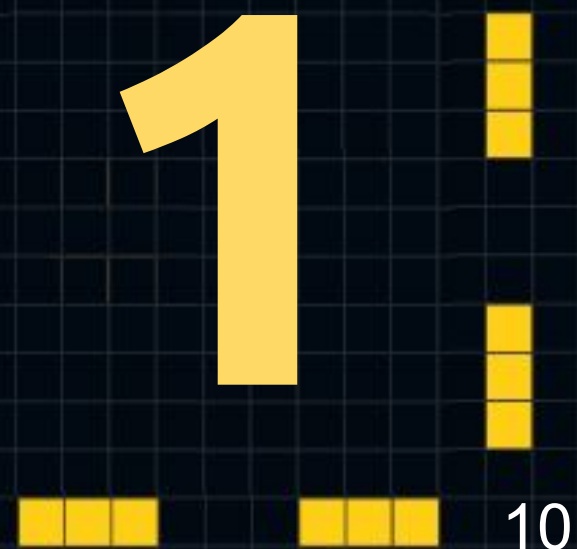
|||

|||

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

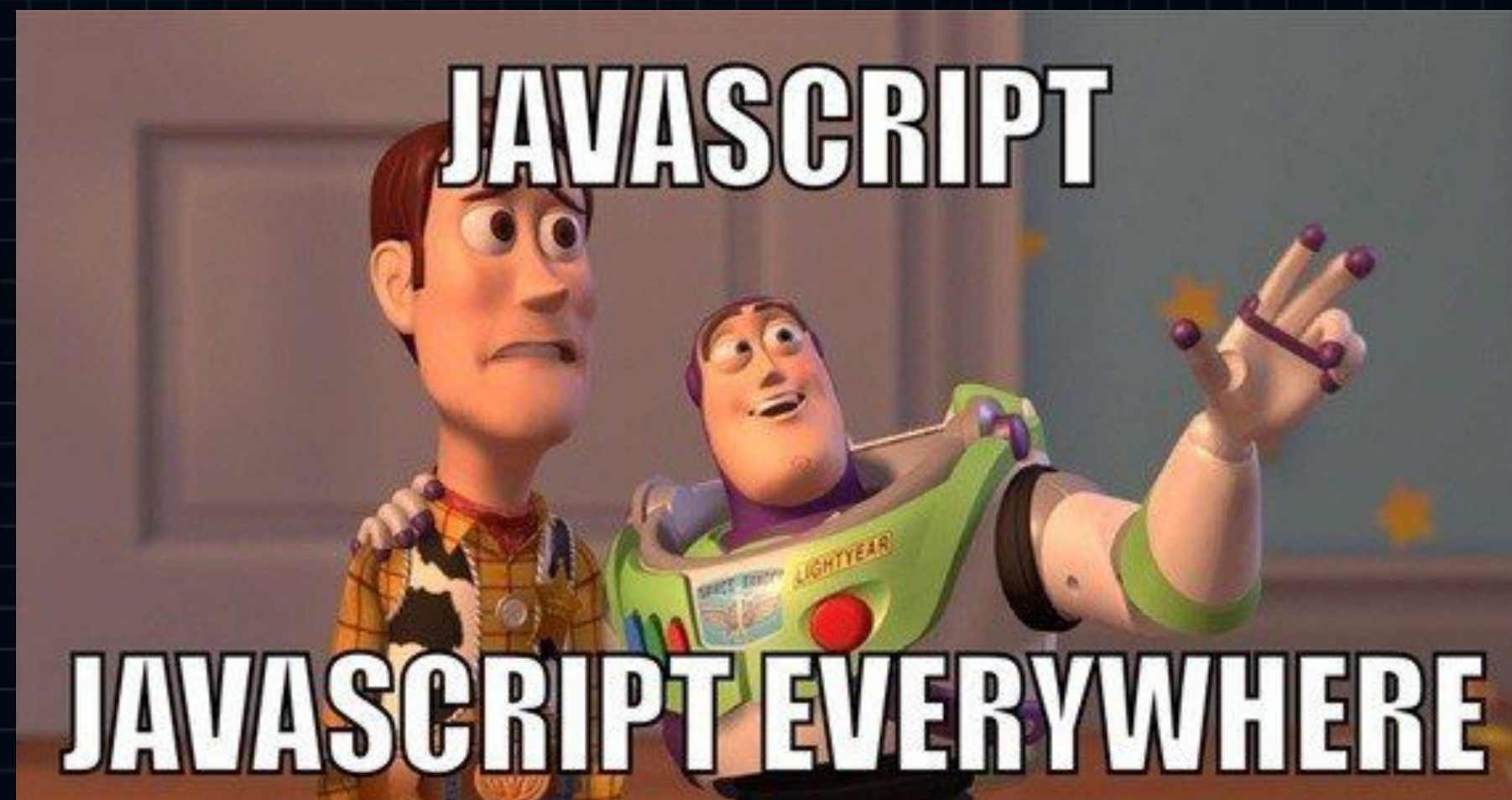


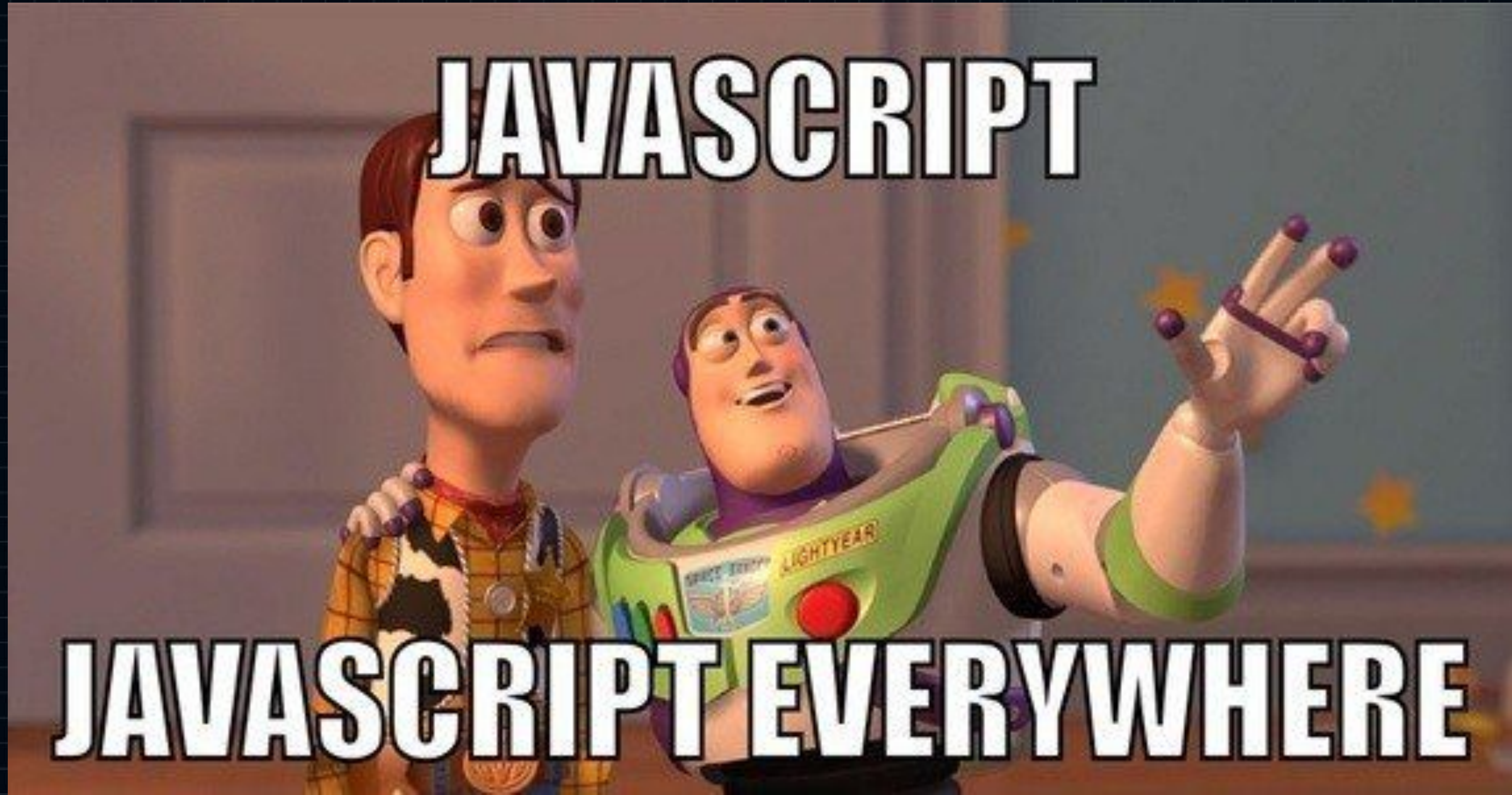


JAVASCRIPT

JAVASCRIPT EVERYWHERE

Disclaimer про Ожидания





приступим ...



11 11

11 11

17

11 11

JavaScript:

The World's Most Misunderstood Programming Language

[Douglas Crockford](http://www.crockford.com)
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?



BrendanEich ✓

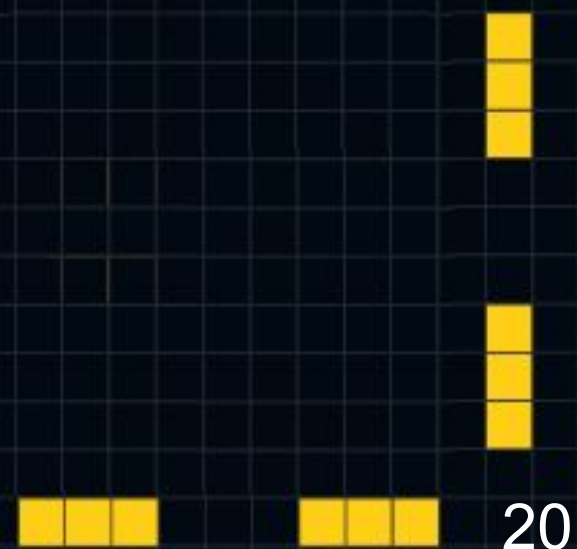
@BrendanEich

И ВСЁ ЭТО ОБЪЕКТЫ

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 ДО-
НАПРИДУМЫВАЛИ ...**



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.



Contribute

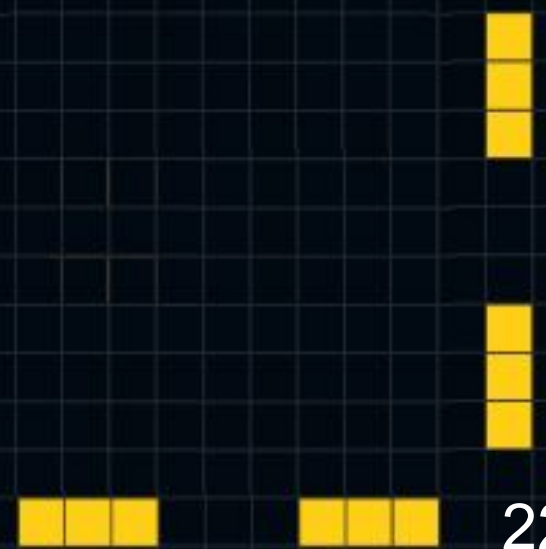
TC39 welcomes contributions from the JavaScript community, whether it is feedback on existing proposals, improved documentation, testing, implementations, or even language feature ideas. See our [contributor guide](#) for details.

To participate in TC39 meetings as a member, [join Ecma](#).

Specs

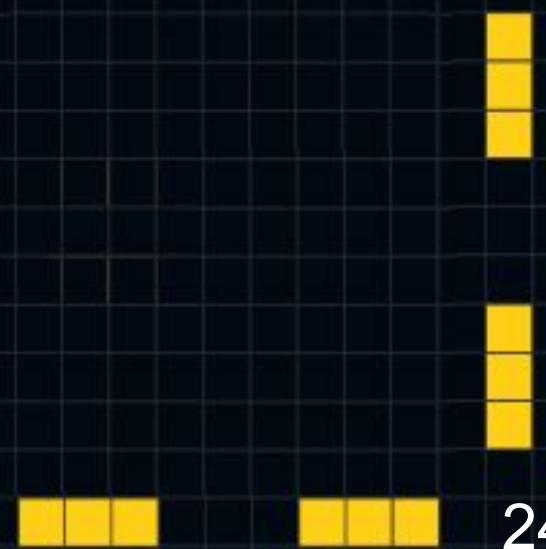
We develop the JavaScript (formally, ECMAScript) specification [on GitHub](#) and meet every two months to discuss proposals. To learn more about the process, please take a look at the [six stages](#) for new [language feature proposals](#). See our [meeting agendas](#) and [minutes](#) to learn more.

практический пример конструирования



```
1 class MyArray {
2   → constructor (...args) {
3     → → const pre = new Array(...args);
4     → → Object.setPrototypeOf(this, new Proxy(pre, {
5     → → → get(target, prop) {
6     → → → → prop = prop.replace('_', '');
7     → → → → return pre[prop];
8     → → → }
9     → → }));
10  → }
11 }
12
13 const myArray = new MyArray(1, 2, 3);
14 console.log(myArray._0);
```

**и ещё один почти
убедительный
рабочий пример**

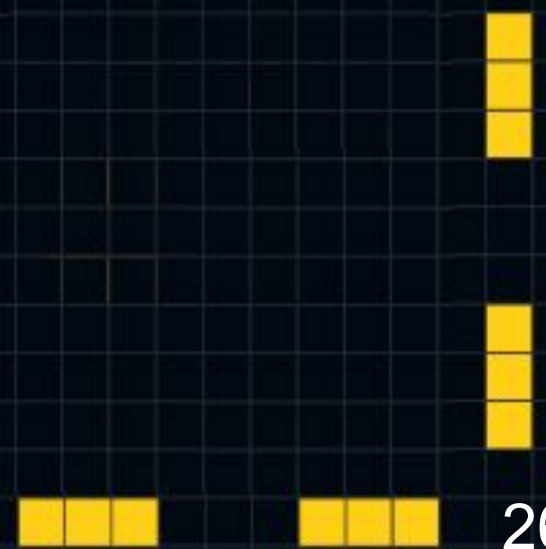


□□□

□□□

□□□
□□□
□□□

проблематика



проблематика

2

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



1 year

Class

Class

Class

2 years

Class
Class
Class

Class
Class
Class

Class
Class
Class

3 years

Class
Class
Class
Class
Class
Class
Class
Class

Class
Class
Class
Class
Class
Class
Class
Class

Class
Class
Class
Class
Class
Class
Class
Class

N years

Class
Class
Class
Class
Class
Class
Class
Class
Class
Class

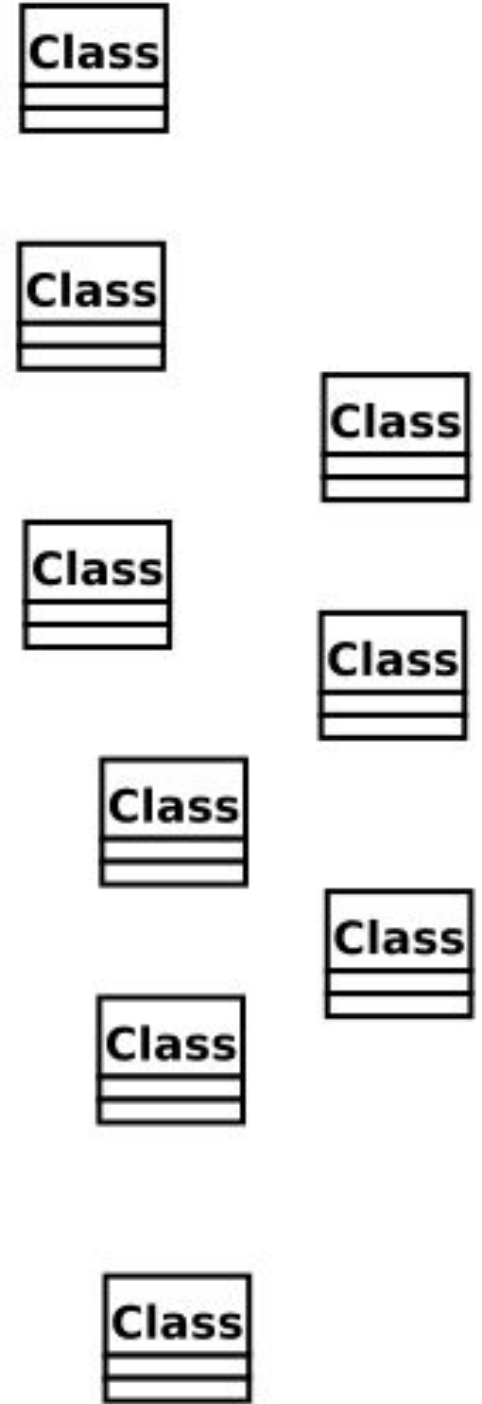
Class
Class
Class
Class
Class
Class
Class
Class
Class
Class

Class
Class
Class
Class
Class
Class
Class
Class
Class
Class

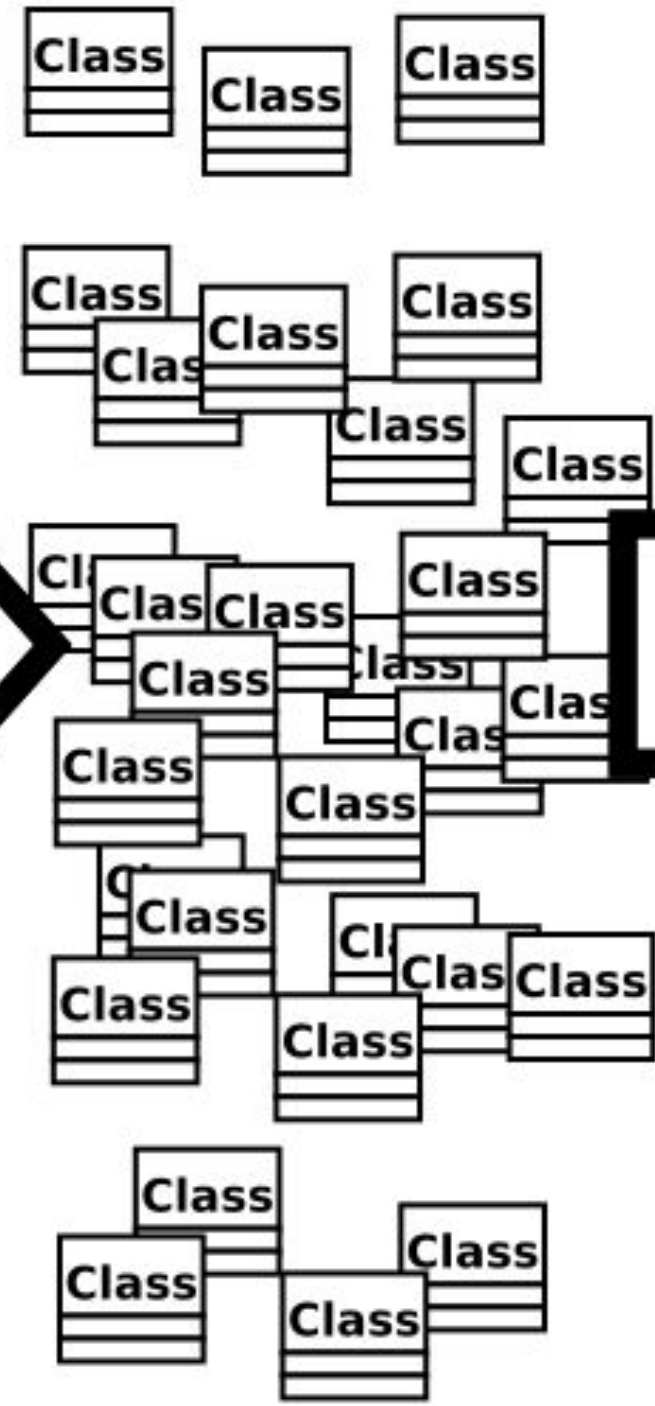
1 day



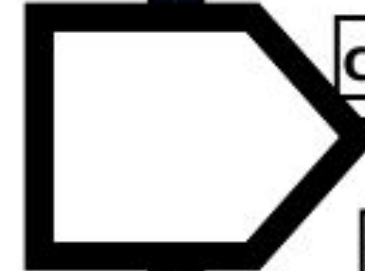
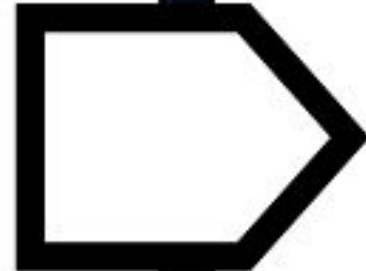
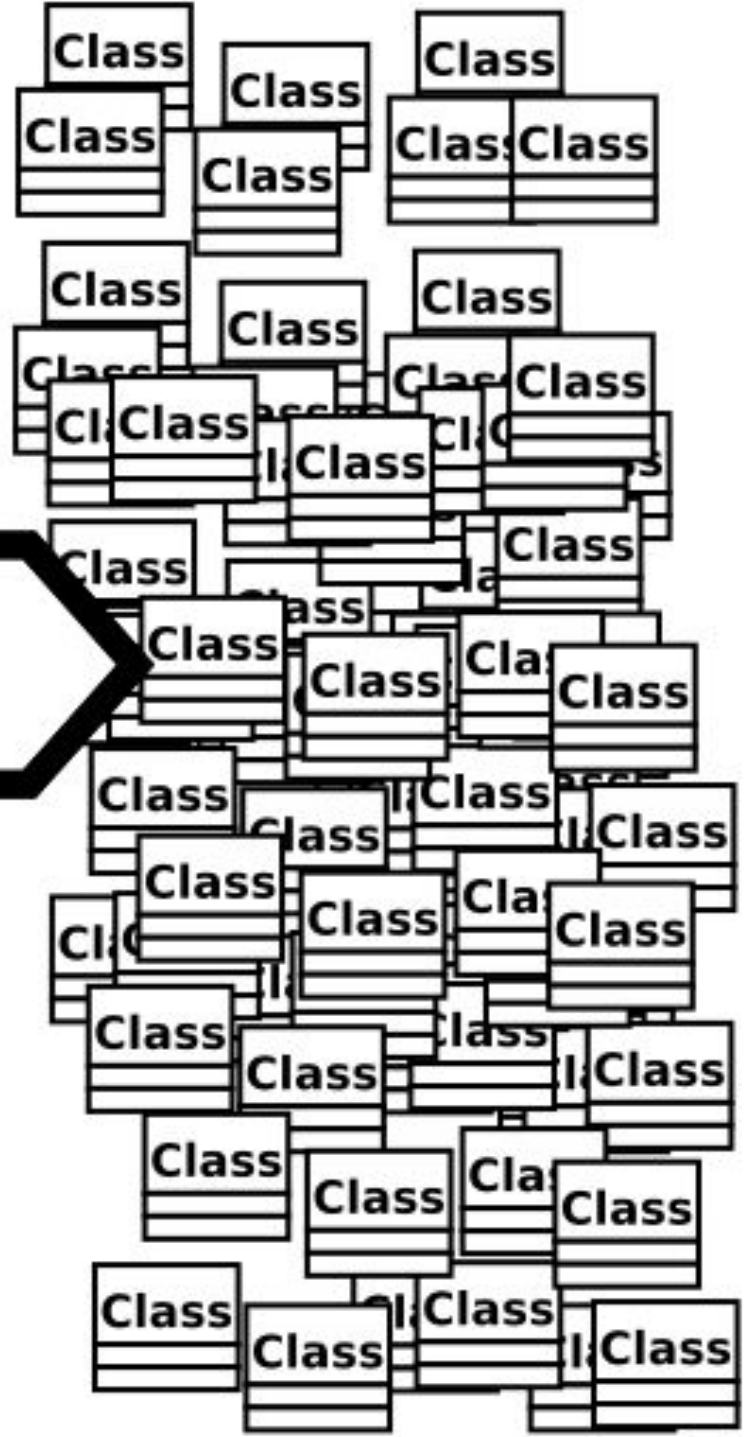
2 days



3 days



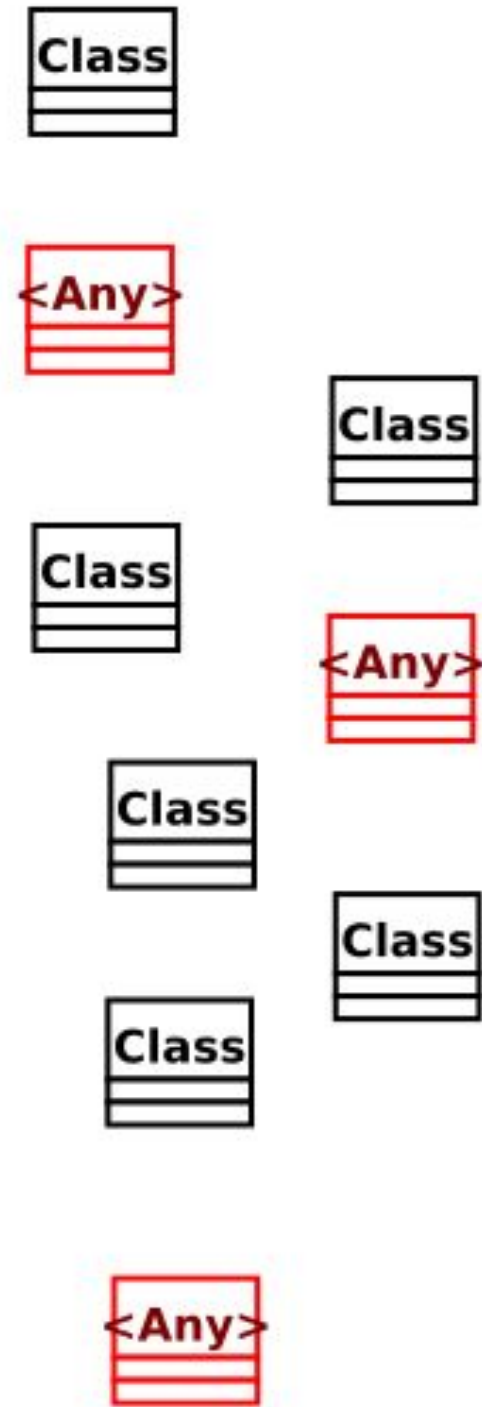
3 month



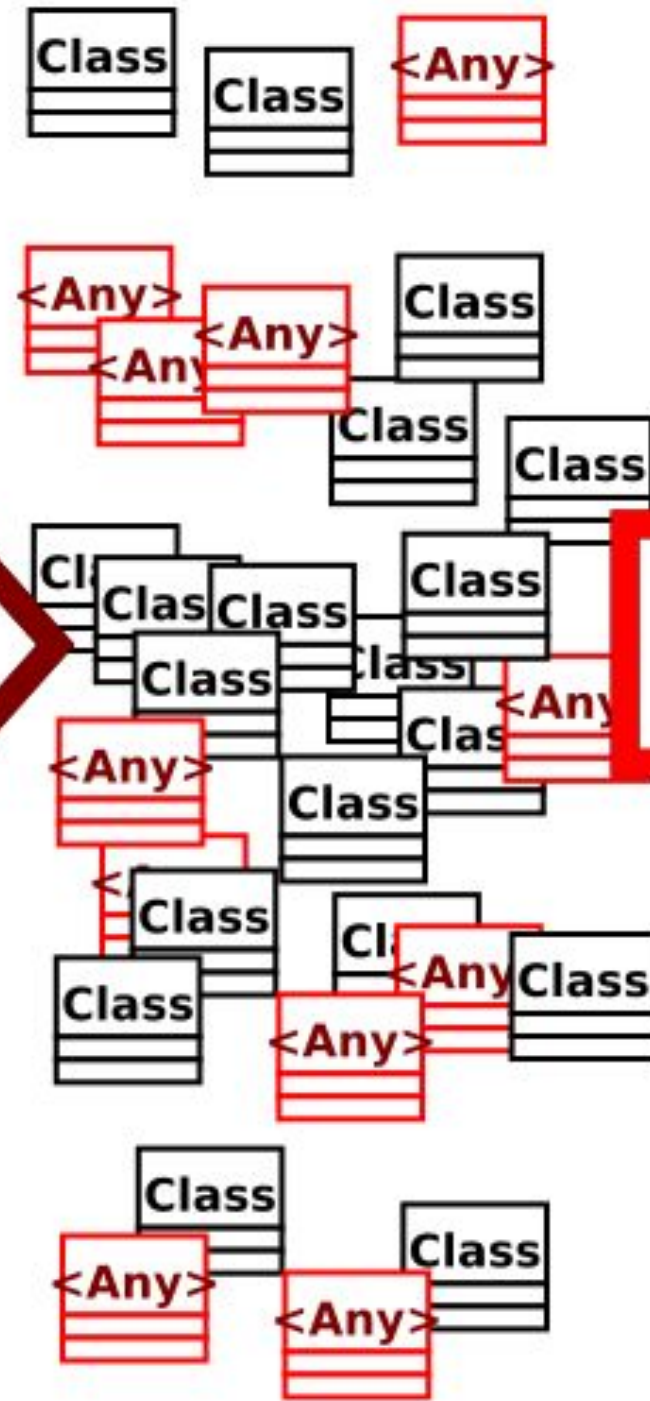
1 day



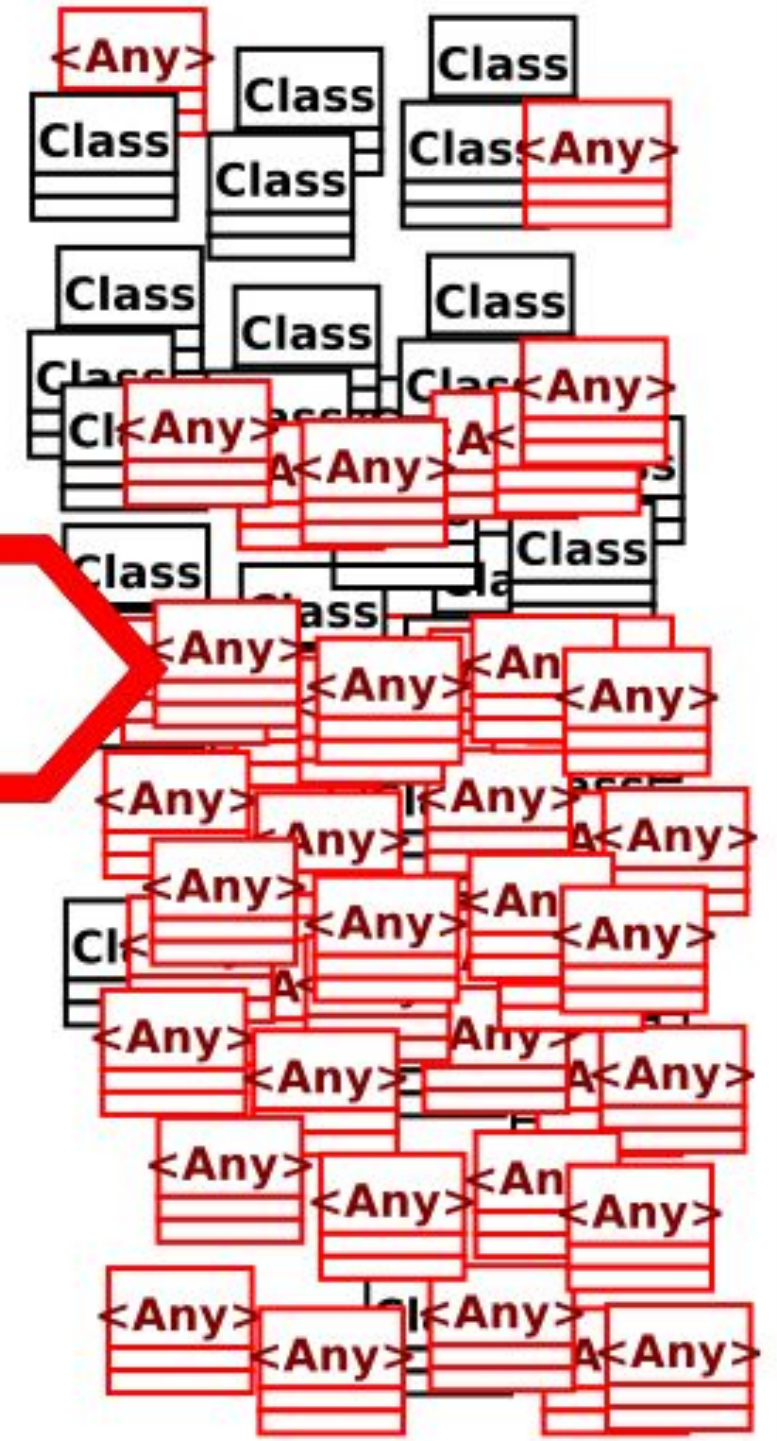
2 years

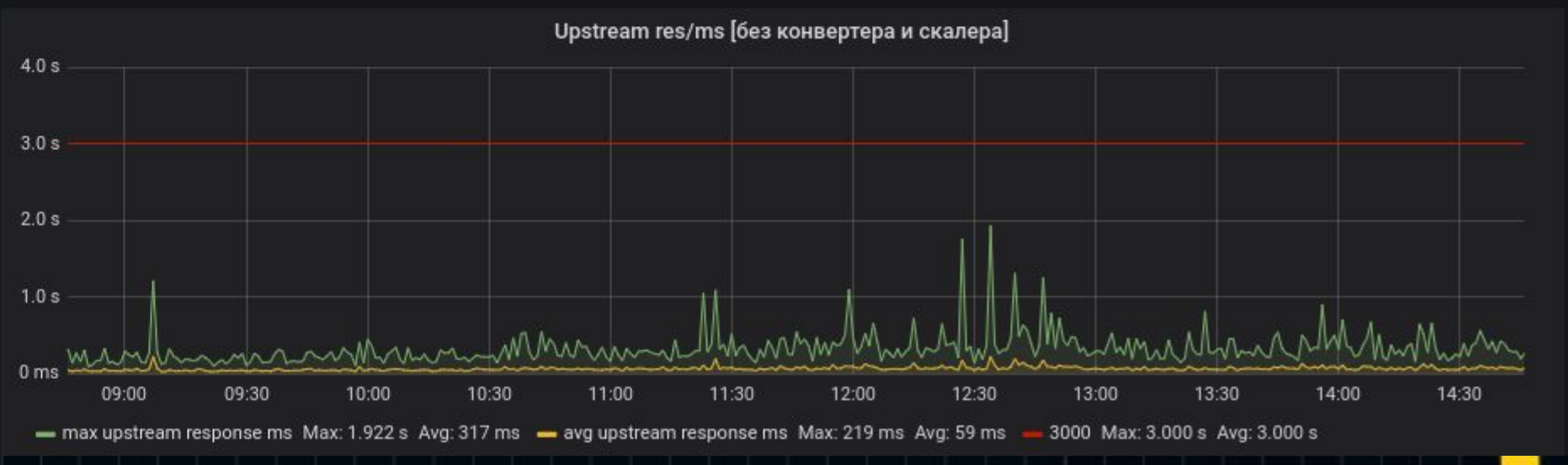
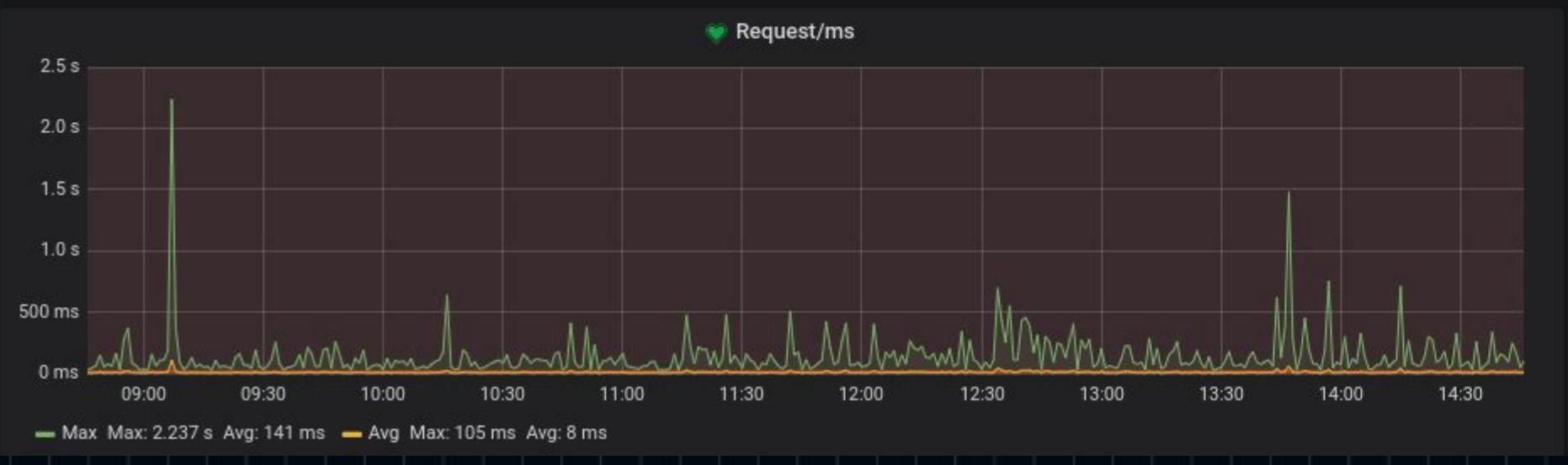
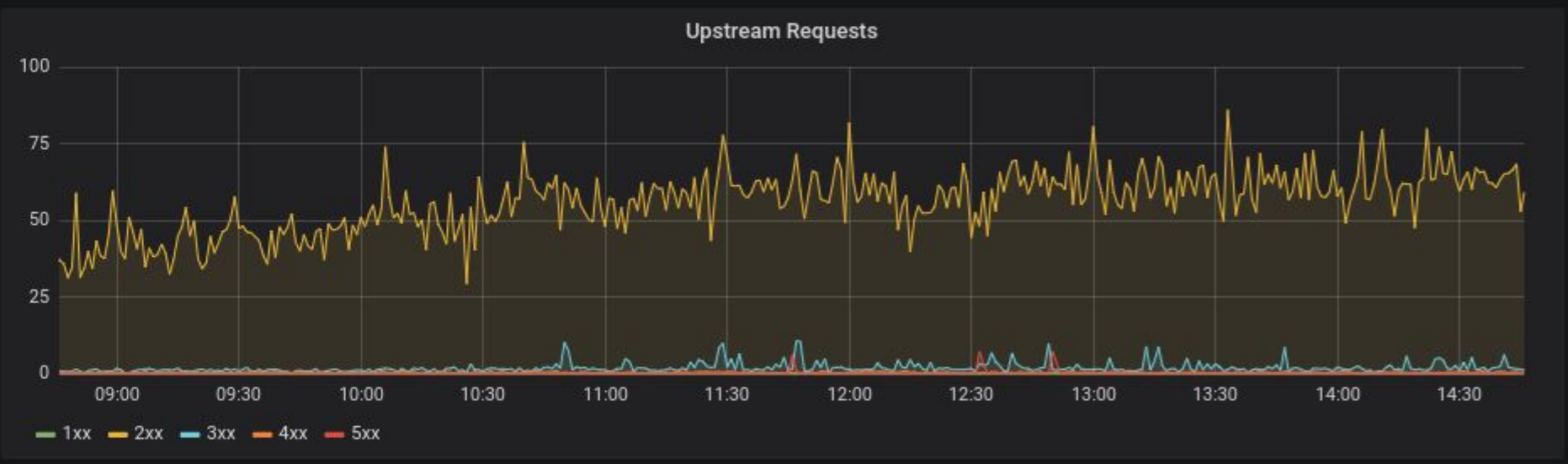
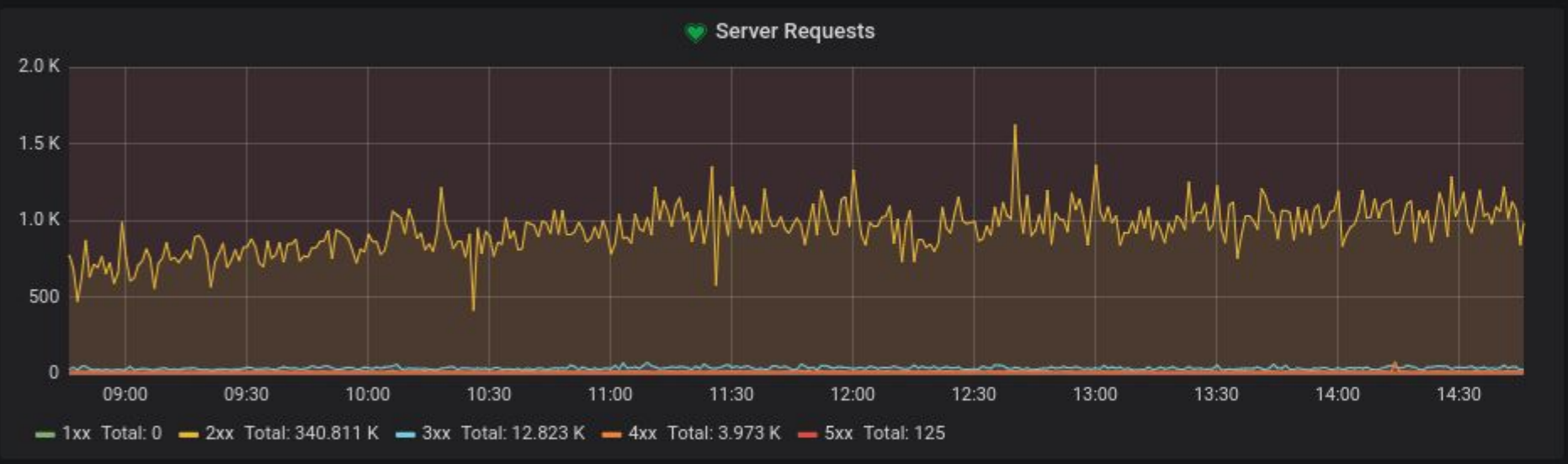
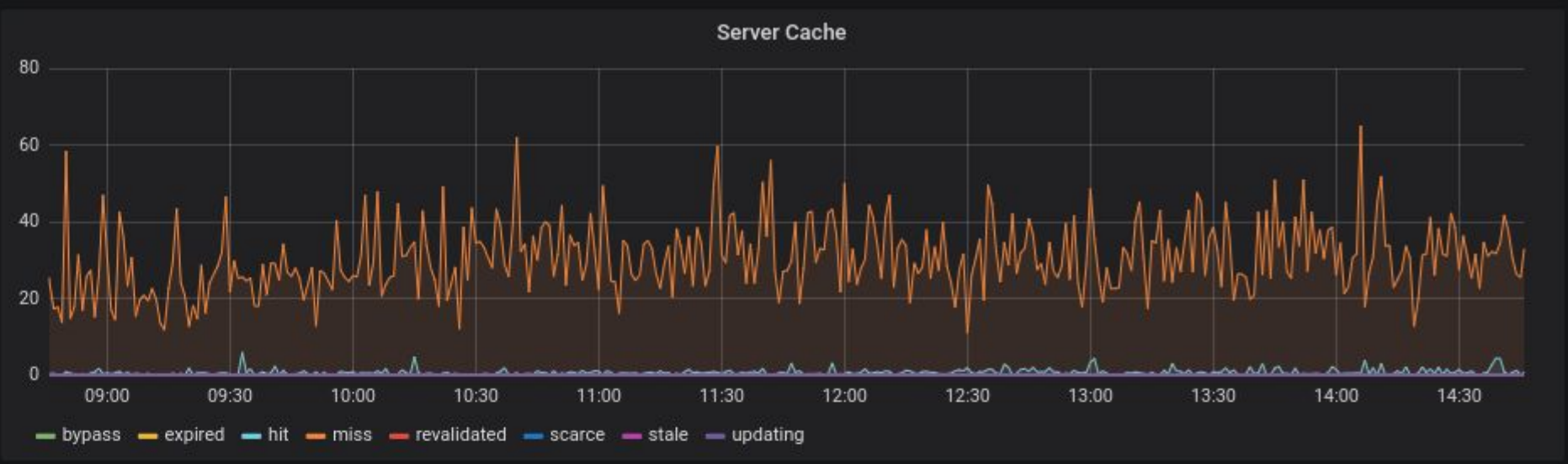
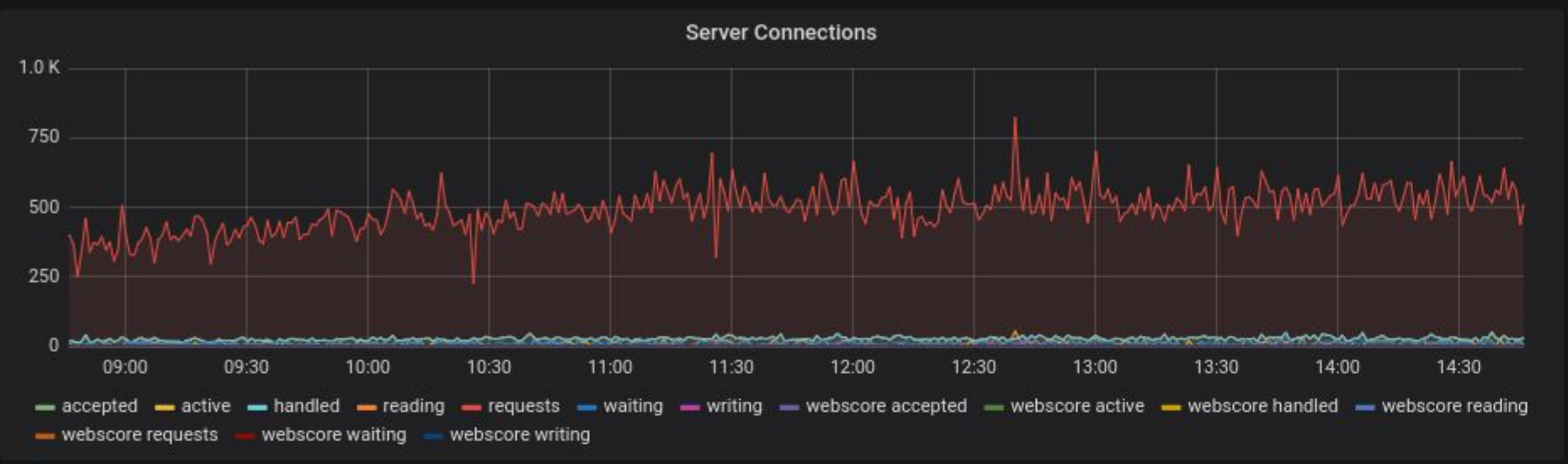


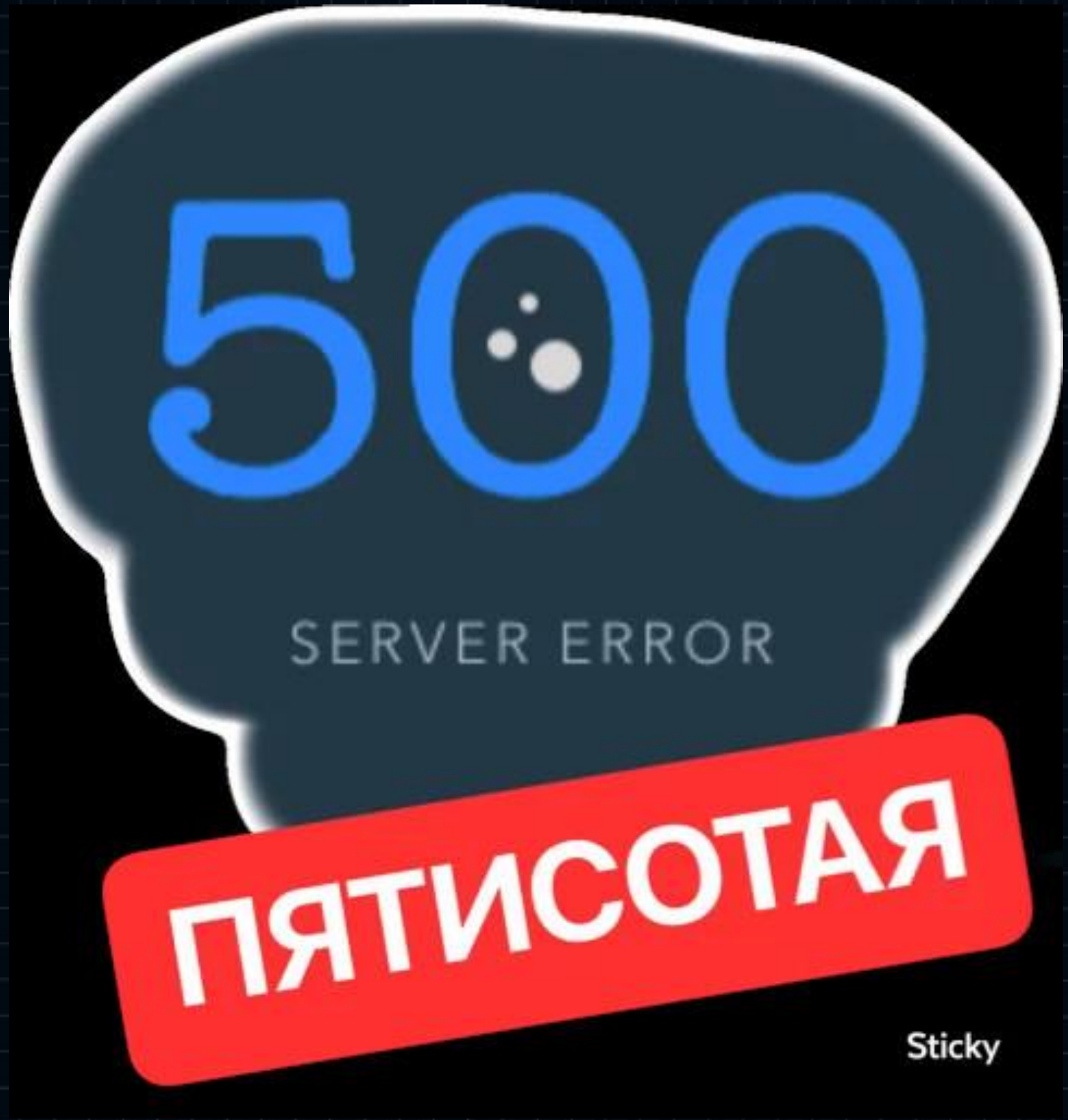
3 years



N years

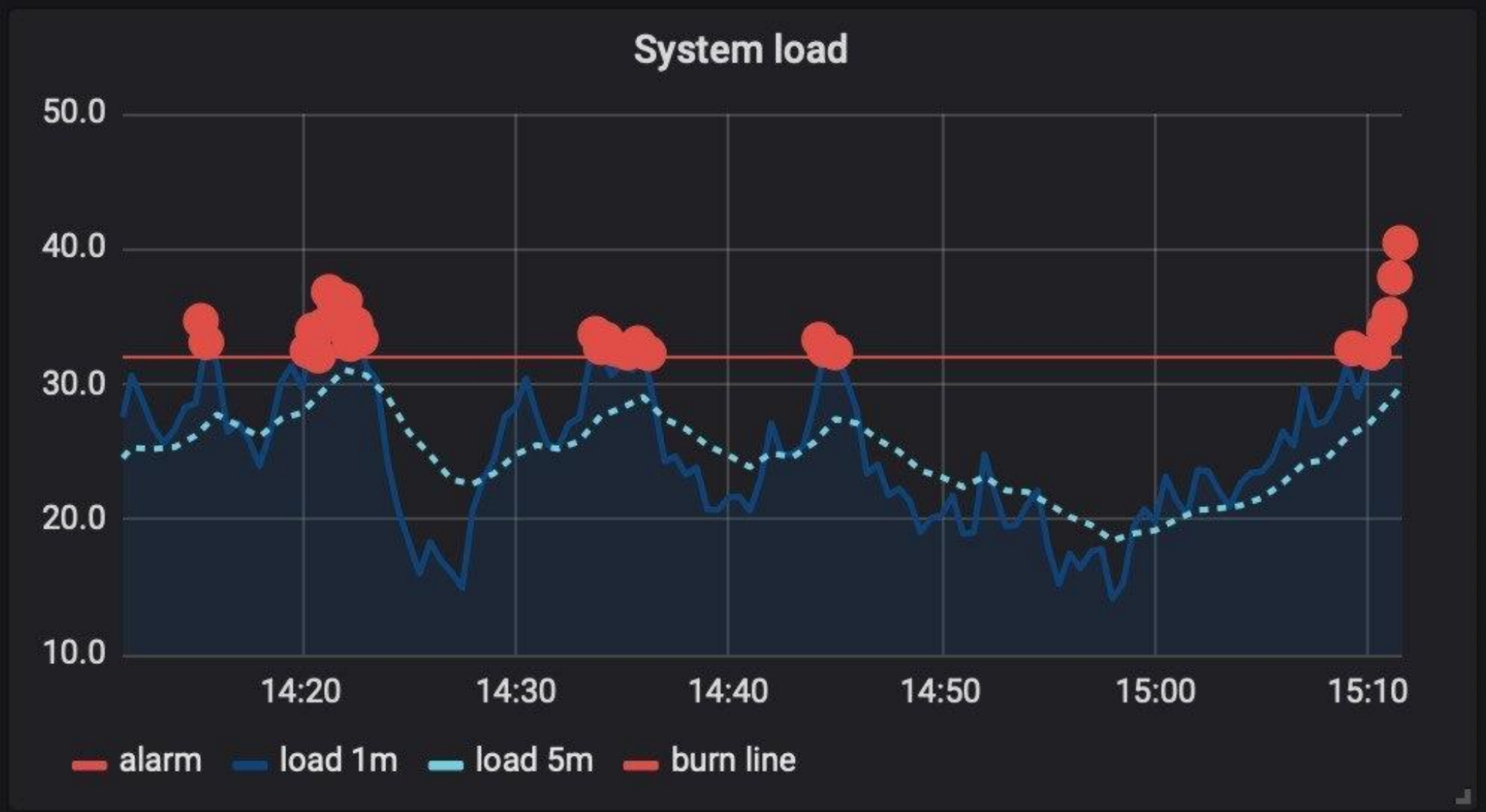
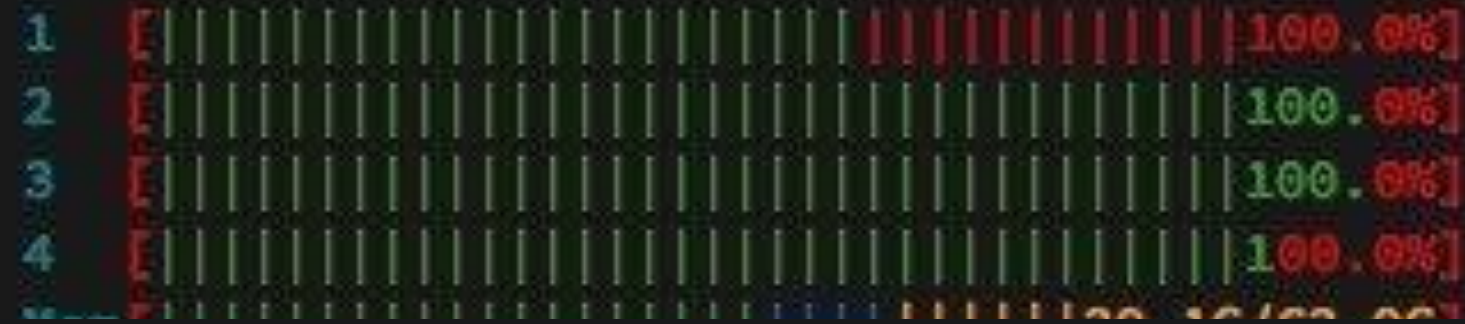




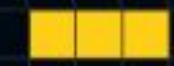


```
1 [||||| 100.0%]
2 [||||| 100.0%]
3 [||||| 100.0%]
4 [||||| 100.0%]
Mem [||||| 29.1G/62.9G]
```

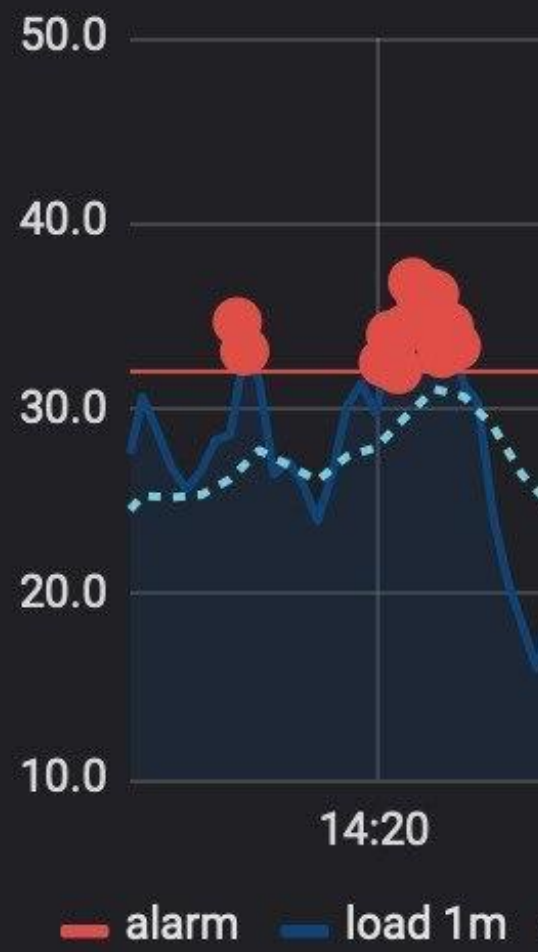
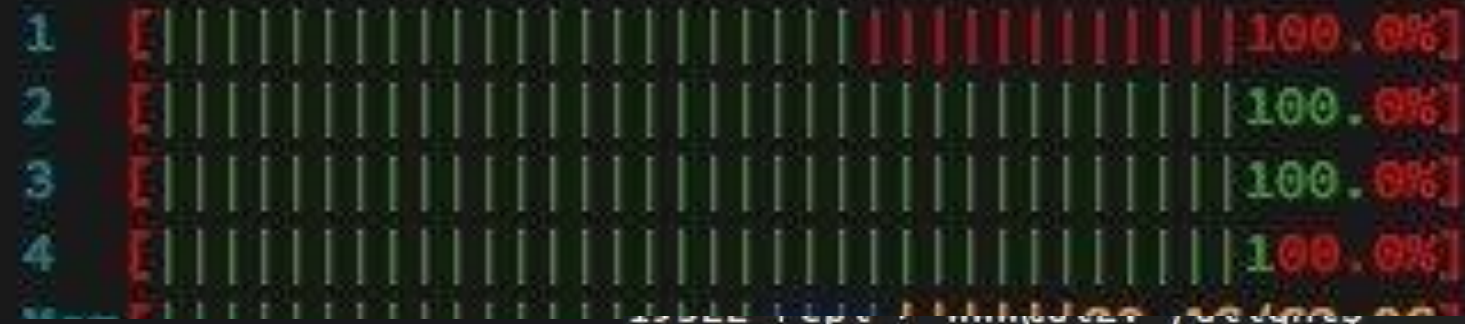
```
5 [||||| 100.0%]
6 [||||| 100.0%]
7 [||||| 100.0%]
8 [||||| 100.0%]
Tasks: 420; 9 running
```



Current free memory %



35



```

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

3!

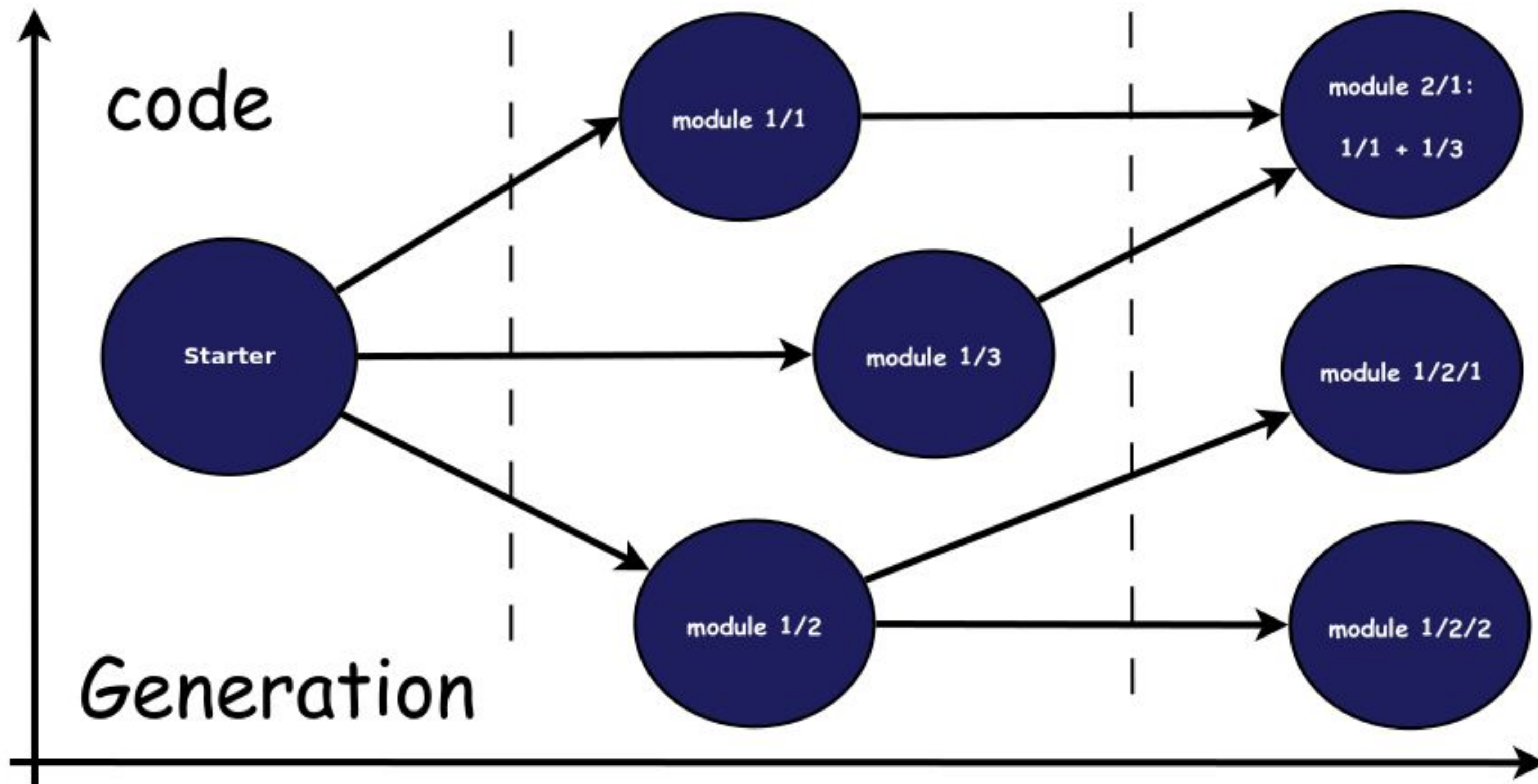
Prototypal Inheritance in JavaScript

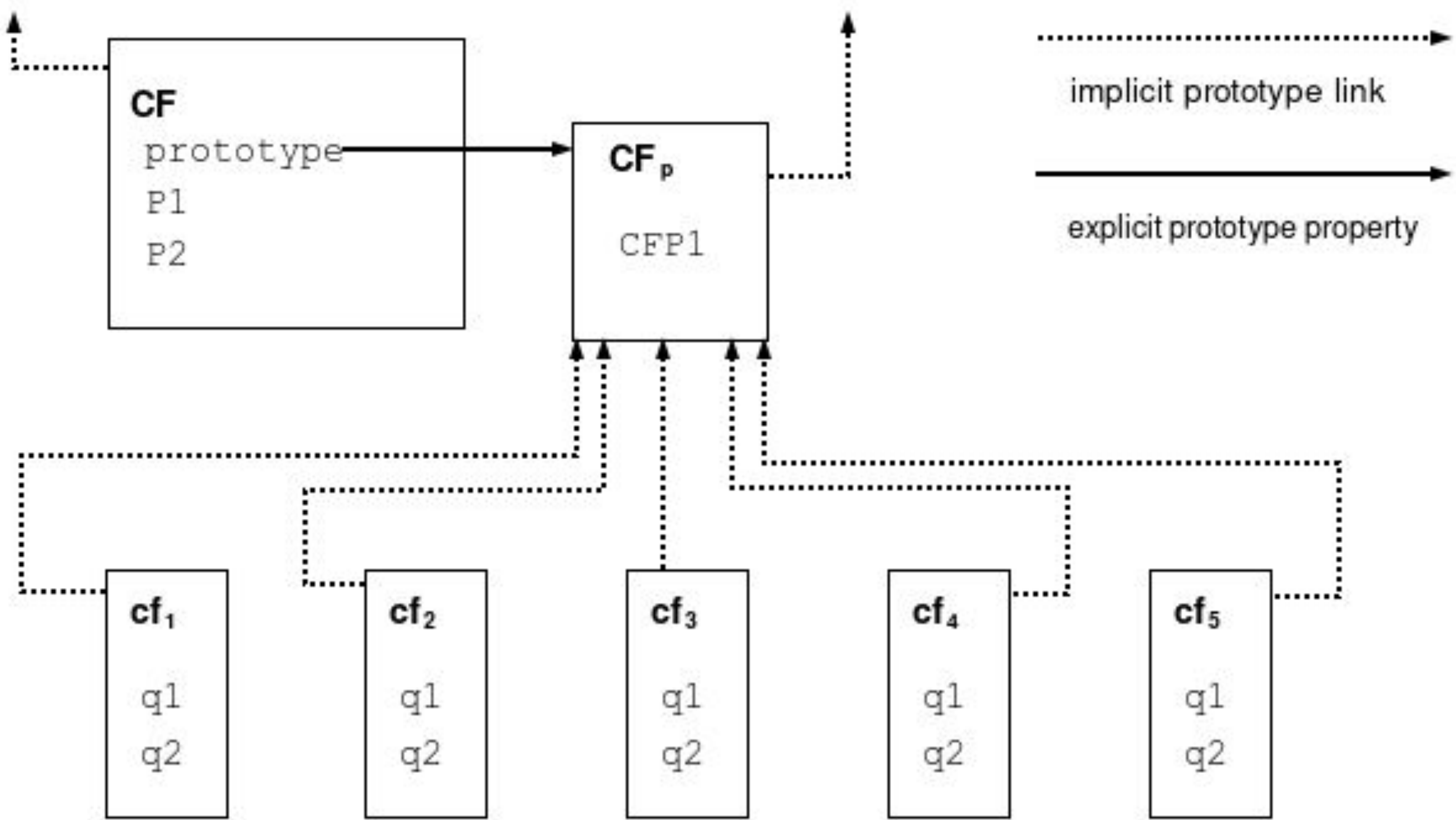
[Douglas Crockford](#)
www.crockford.com

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.

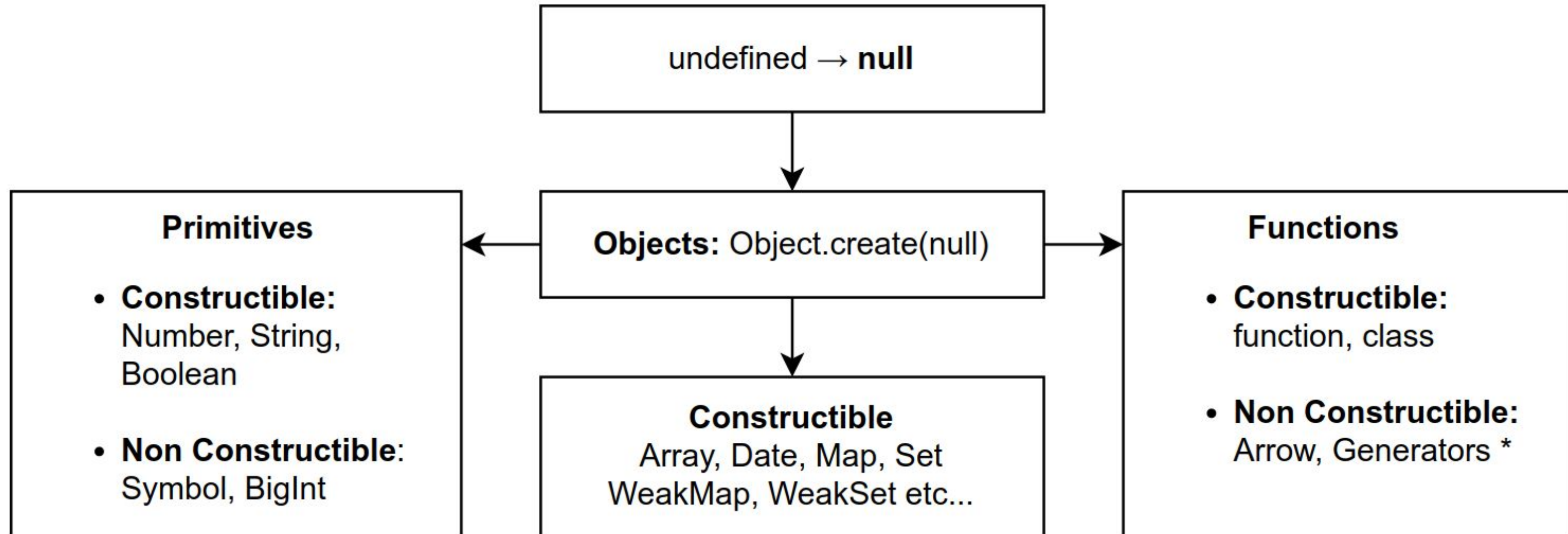
My journey was circuitous because JavaScript itself is conflicted about its prototypal nature. In a prototypal system, objects inherit from objects. JavaScript, however, lacks an operator that performs that operation. Instead it has a new operator, such that

`new f()`





JavaScript Objects Topology





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!

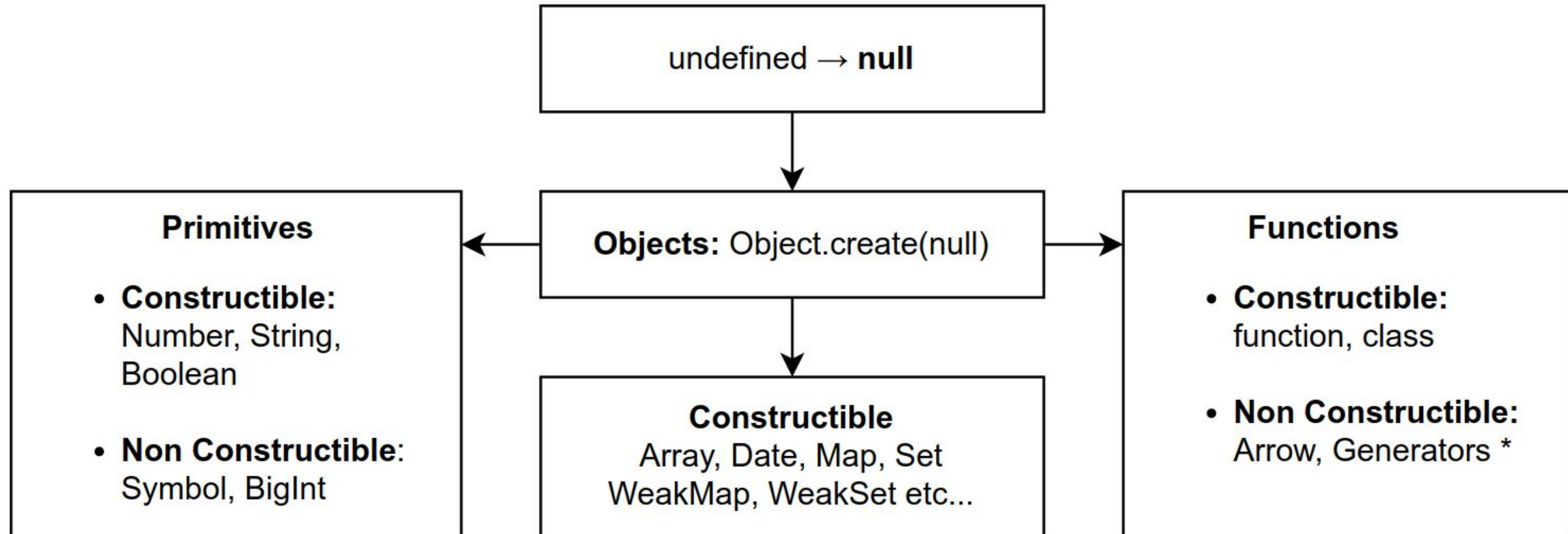


null is not a mistake

my apologies to Sir Charles Antony Richard Hoare



JavaScript Objects Topology



> next

◀ ◻ *MyConstructor* {state: 3} ⓘ

state: 3

◻ *__proto__*:

state: 2

◻ *__proto__*:

state: 1

▶ *__proto__*: Object

Filter

▶ Complete beginners

▶ JavaScript Guide

▶ Intermediate

▼ Advanced

Inheritance and the prototype chain

Memory Management

Concurrency model and Event Loop

References

▶ Built-in objects

▶ Expressions & operators

▶ Statements & declarations

▶ Functions

▶ Classes

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.

In this article

Inheritance w
chain

Constructors

Inspecting pro
dive

Different way:
mutating prot

Performance

Conclusion



Inheritance and the prototype chain

[Edit in wiki](#)

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

English ▼

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

[Edit in wiki](#)[Web technology for developers](#) > [JavaScript](#) > Inheritance and the prototype chain

English ▼

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

typeof null is also good

to my apologies to Brendan Eich



tc39 / ecma262

Unwatch releases 1k Unstar 10.5k Fork 868

Code Issues 226 Pull requests 90 Actions Security 0 Insights

Editorial: special note of null #1913

Edit Open with

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

Conversation 41 Commits 10 Checks 0 Files changed 1 +11 -0



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);
```

Reviewers

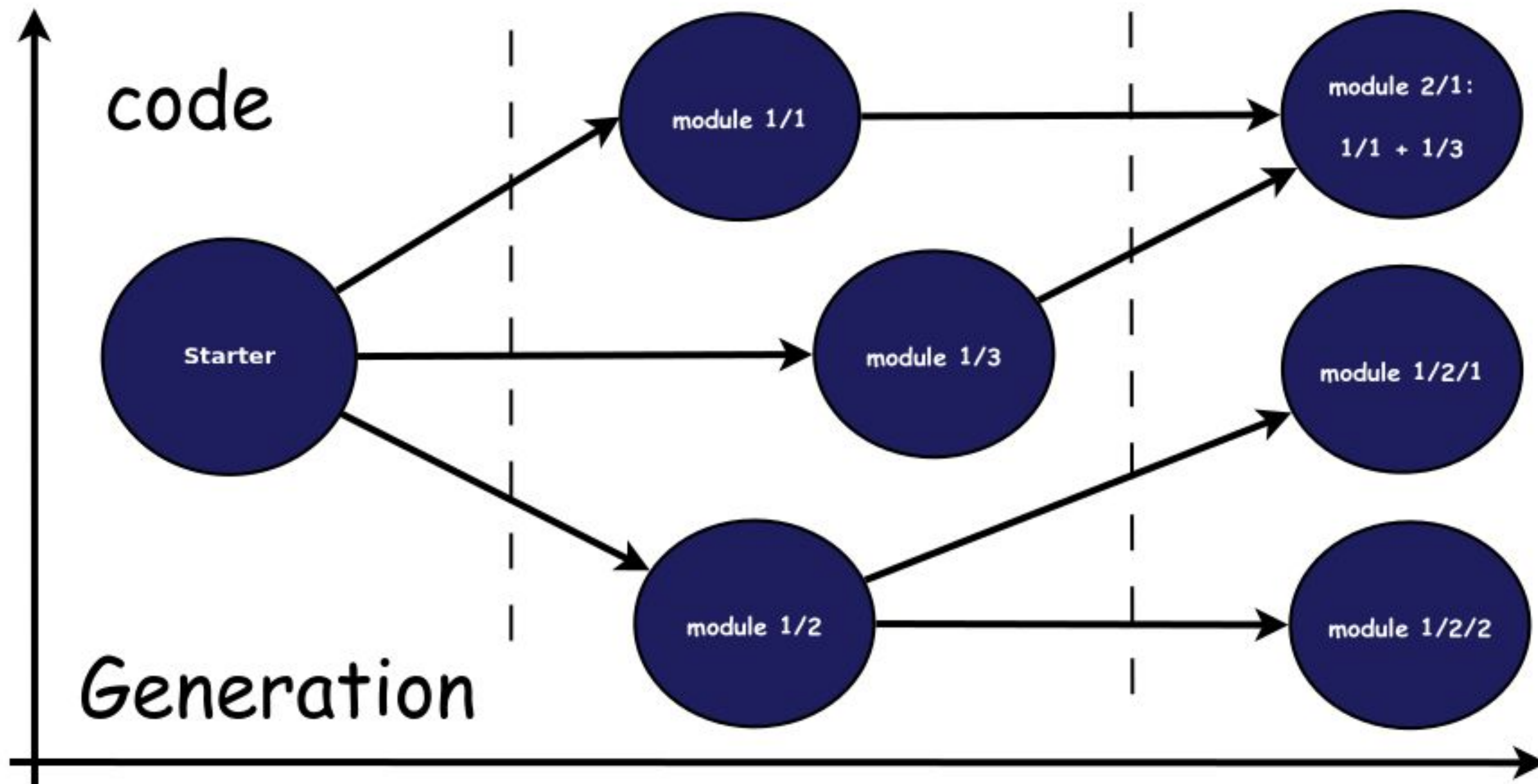
ljharb

Assignees

No one assigned

Labels

None yet

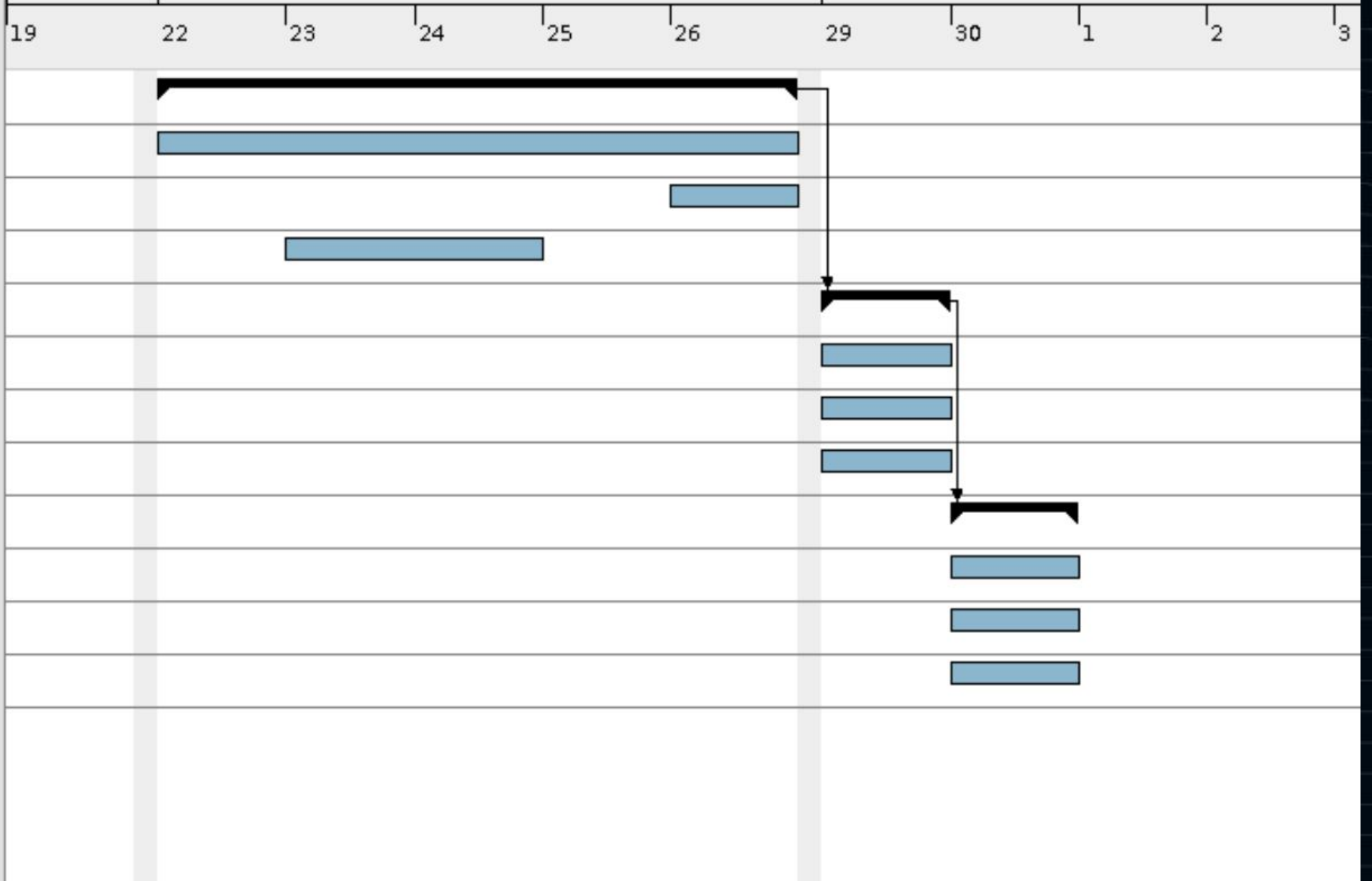




Name	Begin date	End date
• main task	22/06/20	26/06/20
• sub task 1	22/06/20	26/06/20
• sub task 2	26/06/20	26/06/20
• sub task 3	23/06/20	24/06/20
• main sub_task	29/06/20	29/06/20
• sub task 1	29/06/20	29/06/20
• sub task 2	29/06/20	29/06/20
• sub task 3	29/06/20	29/06/20
• sub sub task	30/06/20	30/06/20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

Zoom In | Zoom Out

Week 25 Week 26 Week 27



> next

◀ ◻ *MyConstructor* {state: 3} ⓘ

state: 3

◻ *__proto__*:

state: 2

◻ *__proto__*:

state: 1

▶ *__proto__*: Object

Common Misconceptions x +

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

``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:

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

Follow

 4.91K  70 



Common Misconceptions x +

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

JavaScript Scene

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

``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



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.

```
1 function foo() { };
2 const bar = { a: 'a' };
3 Object
4   .setPrototypeOf(
5     foo.prototype,
6     bar
7   );
8 const baz = Object.create(foo.prototype);
9 console.log(baz instanceof foo);
```

Filter

▼ Constructor

Symbol() constructor

▼ Properties

Symbol.asyncIterator

Symbol.prototype.description

Symbol.hasInstance

Symbol.isConcatSpreadable

Symbol.iterator

Symbol.match

Symbol.matchAll

Symbol.replace

Symbol.search

Symbol.species

Symbol.split

Symbol.toPrimitive

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

```
1 class Array1 {
2   static [Symbol.hasInstance](instance) {
3     return Array.isArray(instance);
4   }
5 }
6
7 console.log([] instanceof Array1);
8 // Expected output: true
9
```

In this article

[Try it](#)[Value](#)[Description](#)[Examples](#)[Specifications](#)[Browser compatibility](#)[See also](#)

63

63

63

решения



решения

4!

2021



Строгая типизация в JavaScript



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

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

76

76

76

примеры



примеры



04_Decorator.ts

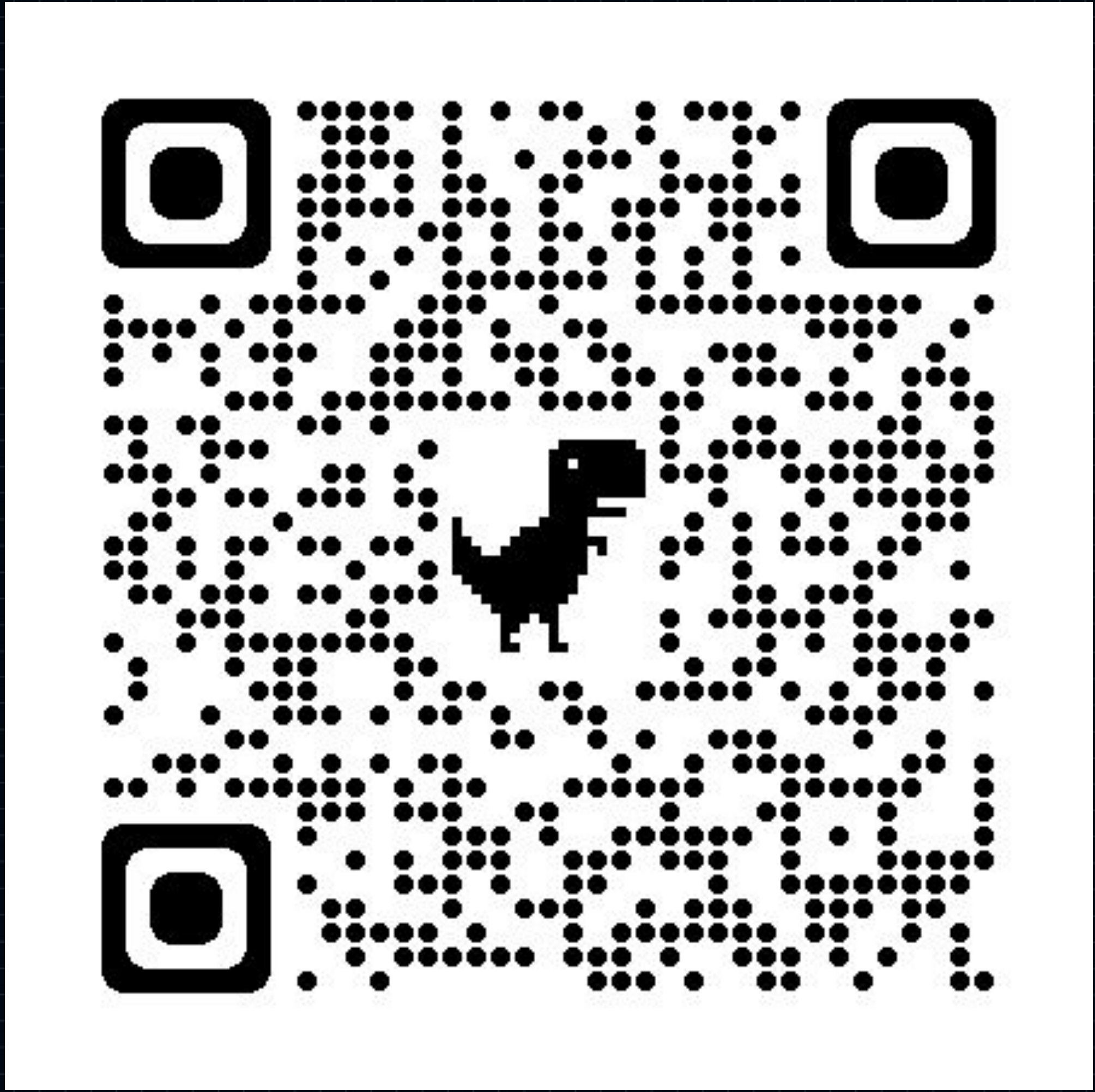


05_NextStep.js



ВЫВОДЫ

- Prototype Chain
 - getter'ы + setter'ы
 - Proxy + Symbol.hasInstance
- ... И НЕМНОЖКО МАГИИ ...



Спасибо !



85

85

85