



НИЕ ВЯРВАМЕ ВЪВ ВАШЕТО БЪДЕЩЕ

Как да пишем работещ JS код?

Работете итеративно (на малки, последователни стъпки).

След всяка нова стъпка, проверявайте как и дали скрипта ви все още работи.

Използвайте `console.log()`, за да си изведете текущите стойности на променливите и резултатите от функциите ви.

Използвайте дебъгера, за да виждате къде какво точно става.

Често грешката е нещо много дребно като лиспваща запетайка или скоба, но пък чупи цялата програма.

Често допускани грешки - 1

1. Лошо форматиране

- използването на правилна индентация е нещо много важно при писането на програмен код. Тя ни помага да се ориентираме къде започват и къде свършват блоковете от код

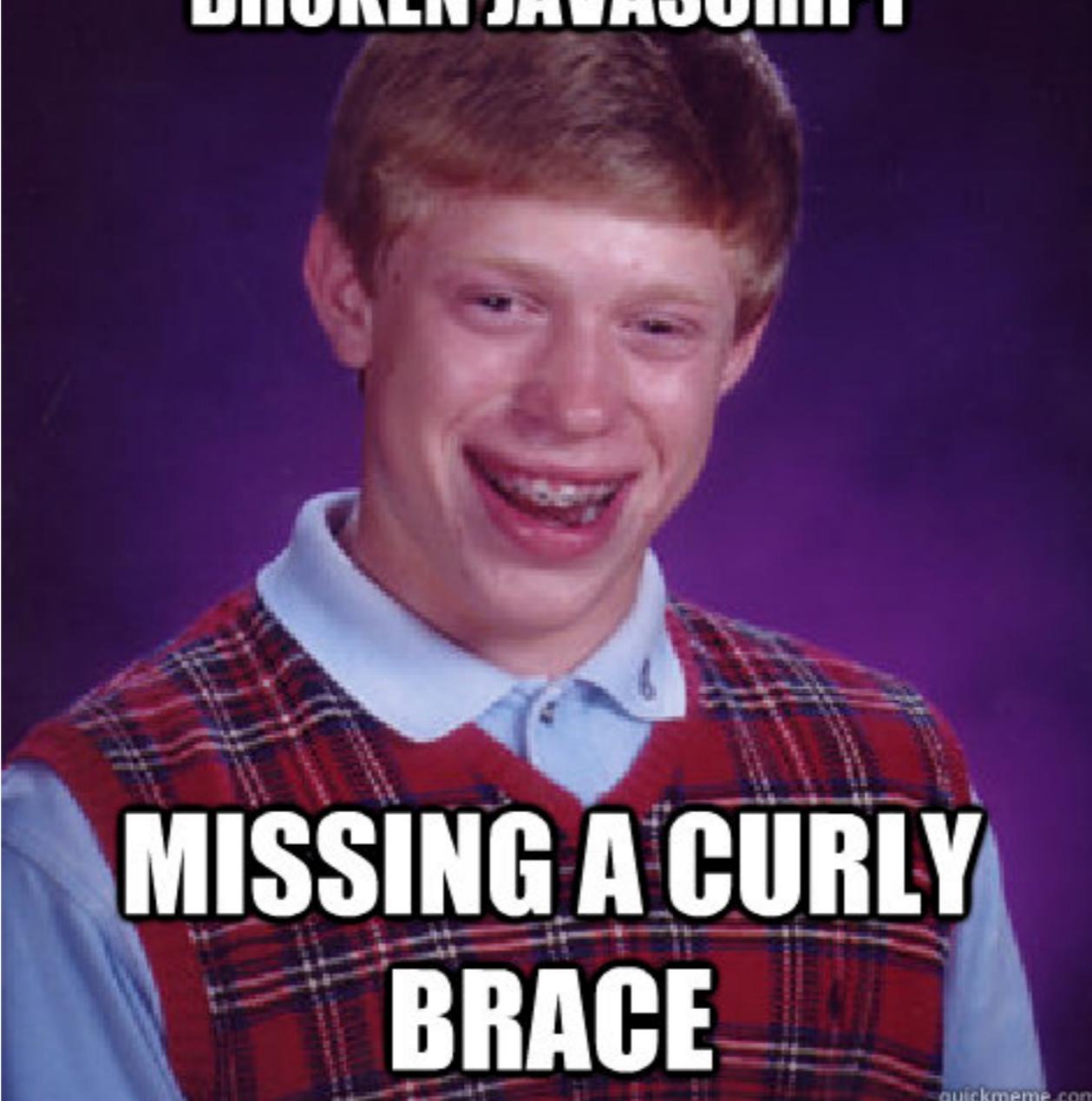
2. Синтактични грешки

- синтактичните грешки са грешно написани думи от езика, грешно поставени скоби, запетайки и т.н.
- за да можете да работите лесно, трябва да научите синтаксиса на езика
 - това включва научаването на запазените думи и на правилата за поставяне на скоби и запетайки

3. Грешно изписано име на свойство (property) на обект

```
[1,2,3,4,5].length // => 5  
[1,2,3,4,5].lenght // => undefined
```

**WASTES 2 HOURS DEBUGGING
BROKEN JAVASCRIPT**



Scope



& Namespace

Контекст

- Обхват (зона на действие) на това за което се говори. Извън тази зона на действие, това за което се говори няма смисъл.
- Пример от реалния живот:
 - Видя ли Лора?
 - Да.
 - Как е тя!извън контекста на разговора, това изречение няма смисъл

- От компютърните езици:

```
function calc(x) {  
    return x / 2;    // ok, I know x  
}  
console.log(x); // who the hell is x?..
```

Контекст в JavaScript

- Scope-а в JavaScript се определя от функциите
- Scope е онази част от кода, където създадените променливи и именувани функции имат смисъл. Извън тази зона те не съществуват
- Зоната на действие на една променлива или функция е тялото на функцията, в която те са дефинирани
- Т.е. тази зона започва там където е отварящата скоба на обграждащата функция (“{”) и свършва със затварящата скоба (“}”) или при първия return, който ще бъде изпълнен
- Понякога ще казваме “зона на видимост” или “текущ контекст”

Глобален контекст

- Това е най-външния възможен контекст. Той съдържа всички останали.
- Глобалният контекст е обвързан с глобалната променлива `window`, която е служебна и се създава от браузъра по време на инициализация. Т.е. всичко което дефинираме директно в глобалният контекст, отива в обекта `window`
- Всички променливи и функции, които се декларират в глобалния контекст, са глобални и съответно видими отвсякъде
- Създаването на глобални функции и променливи се счита за лош стил на програмиране и може да доведе до непредвидени грешки

Локален контекст

- Всичко, което не е глобален контекст е локален контекст
- Т.е. - всичко, което се намира в тялото на функция (което е оградено от `function() { .. }`)
- **Локалните функции и променливи не се виждат отвън!**
- Всяка локална функция или променлива, се вижда във всички вложени в нея контексти (scopes)
- Локалният контекст се нарича още *closure*

```
var x = "I'm global";  
  
function level1 () {  
  
    var y = "level 1 variable";  
  
    function level2 (param) {  
  
        var z = "level 2 variable";  
  
        console.log("I can see x and y and z");  
        console.log("I can also see param");  
  
        return x + y + z + param;  
  
    }  
  
    console.log("I can see x and y and level2");  
    return level2("surprise!");  
  
}
```

Глобални:
x, level1

Локални за level1:
y, level2

Локални за level2:
z, param

Функционално програмиране

- Всяка функция прави едно конкретно нещо
- Функцията работи единствено с параметри (без да ползва променливи, които са декларирани извън нея)
- Функцията връща стойност
- За всяко действие създаваме функция, като се стараем тя да е достатъчно универсална, за да може да се пре-използва (да се използва повече от 1 път)

Функциите в JS

- Синтаксис:

```
function име_на_функция(аргумент1, аргумент2, ...) {  
    // изчисления  
    return;  
}
```

- Извикване:

```
име_на_функция(стойност1, стойност2, ...);
```

return

когато направим return във функция, кодът който се намира след return-а не се изпълнява:

```
function example() {  
    console.log("This text will be printed");  
    return;  
    console.log("This text will NOT be printed");  
}
```

Счита се за лош стил ако има прекалено много returns в една функция. В идеалният случай има само един и той е най накрая във функцията, точно преди затварящата скоба ("}")

Свойства на функциите в JS

- могат да се присвояват (задават като стойност) на променливи
- могат да се подават на други функции като параметри
- функциите могат да бъдат връщани като резултат от други функции
- **Заключение:** можем да боравим с функцията по същият начин както и с обикновенна стойност!

I DON'T OFTEN DO FUNCTIONAL
PROGRAMMING



BUT WHEN I DO, IT'S ALWAYS IN
JAVASCRIPT

memegenerator.net

Pure functions

- A pure function is a function where the return value is only determined by its input values, without observable side effects. This is how functions in math work: `Math.cos(x)` will, for the same value of `x`, always return the same result. ([SitePoint](#))
- Функционалното програмиране използва само чисти функции!
- Чистите функции в JS не използват променливи отвън (работят само в своя контекст) и връщат стойност изчислена на база на входните си параметри (аргументи)
- Използването на променлива, създадена извън функцията е Code Smell (termin описващ програмен код, като рисков)

Анонимни функции & IIFE

- Когато създадем функция без име, тя е анонимна
- За да изолираме контекста, в който работим можем да използваме самоизвикваща се анонимна функция (IIFE)

Синтаксис:

```
(function () {  
    "use strict";  
    // всичкия код на програмата  
})();
```

- Още по темата:

[Code smell](#)

[properly-isolate-variables-in-javascript](#)

[IIFE - Immediately Invoked Function Expressions](#)

Namespace

- Когато даден обект съдържа функции, то достъпът до тях става чрез името на обекта
- Обекти, които са създадени специално за тази цел, се наричат *namespace* и се пишат с главна буква
- Неймспейса е **именуван контекст**. Той ни позволява да използваме неговите функции, въпреки че те не са част от текущия контекст
- Пример за глобален неймспейс са обекти като Math и Date
`Math.round(1.23) // => 1`
`round(1.23) // => ReferenceError: round is not defined`



Hoisting & strict mode

Hoisting

- Пример:

```
sayHi();  
function sayHi() { console.log("Hi!"); } // => "Hi!"
```

- Причината този код да работи е, че създаването на функциите става още преди изпълнението на кода от текущия контекст
- Същото важи и за променливите - ако използваме променлива, преди да сме я декларирали (създали), то нейната стойност ще е `undefined`:

```
console.log(missingVariable);  
var missingVariable = 1; // => undefined
```

Hoisting problem

```
/* hoisting-а може да доведе до грешка, ако използваме  
променливата като функция, преди да е инициализирана */  
  
console.log(func1(5)); // => 10  
  
console.log(func2(5));  
TypeError: func2 is not a function  
  
function func1 (x) {  
    return x + x;  
}  
  
var func2 = func1;
```

“hoisting teaches that variable and function declarations are physically moved to the top of your coding, but this is not what happens at all.

What does happen is that variable and function declarations are put into memory during the compile phase, but stays exactly where you typed it in your coding.”

- MDN: Hoisting

Недекларириани променливи

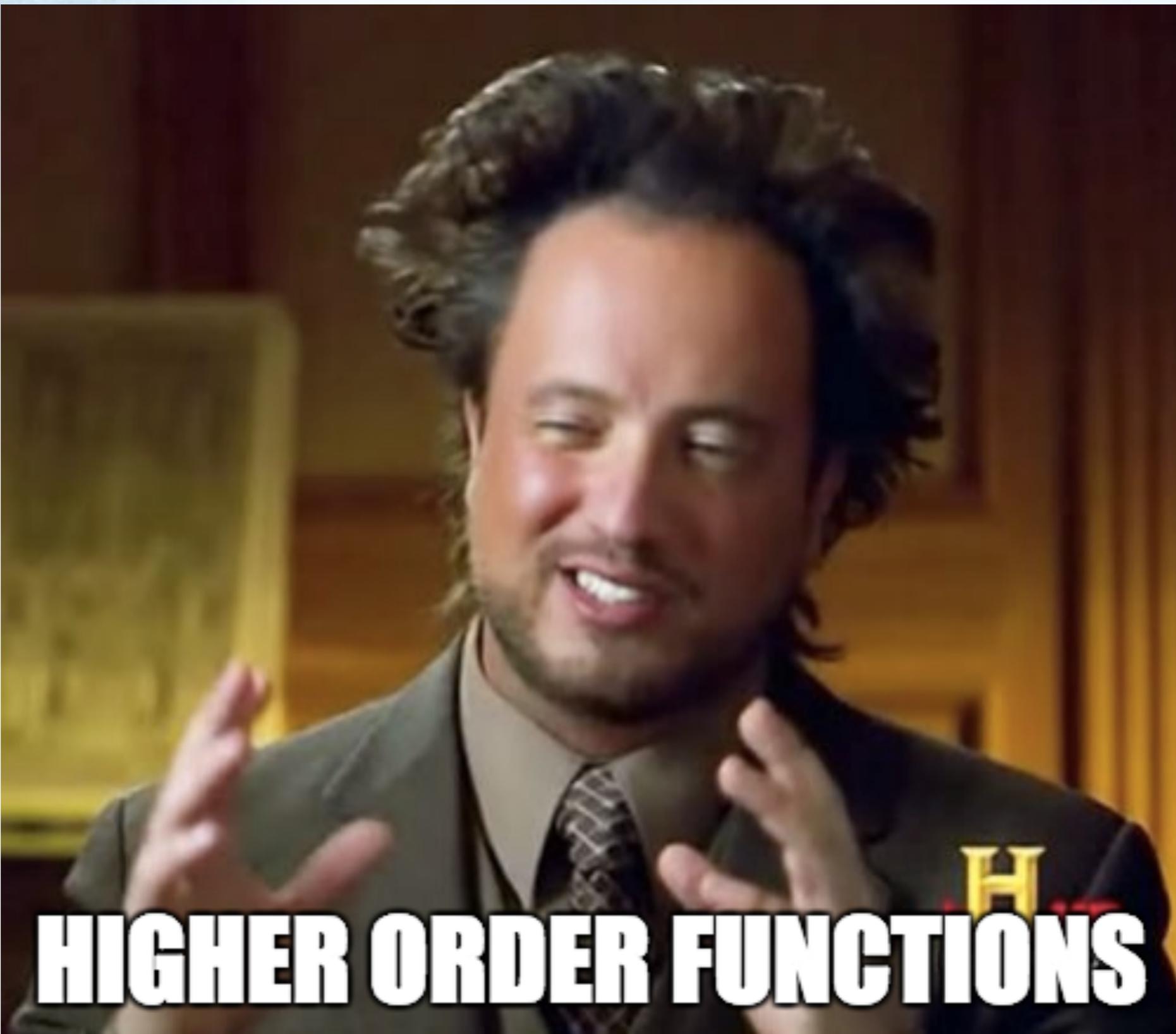
- В JavaScript има и още една особеност - ако директно зададем стойност на една променлива, без да сме я декларирали с var, то тази променлива ще се създаде автоматично и ще се закачи за най-глобалният контекст (обекта window)
- Това не е част от хостинг-а, но общото между двете е че и двете могат да създават потенциални проблеми
- Например:

```
var myVar;  
myvar = "alabala";  
// незабелязана грешка в името на променливата
```

Strict mode

- "use strict"; // Converting mistakes into errors
- Стриктният режим помага за т.нар. "fast fail" или иначе казано, предизвиква грешка там, където има потенциална опасност от грешка
- С други думи - помага на програмиста да пише по-правилен и стабилен код
- Поставя се в началото на функцията или директно в глобалният контекст
- https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Strict_mode

Въпроси?



HIGHER ORDER FUNCTIONS

Да започнем с примери

- Всеки списък има вградената функция `forEach`:

```
var array = ["John", "Eddie", "Anna"];
typeof array.forEach; // => "function"
```

- Тази функция приема като аргумент друга функция:

```
function iKnow(name) {
  console.log("I know", name);
}
array.forEach(iKnow);
```

- Принципа, на който работи `forEach`, е че итерира по списъка (обхожда елементите на списъка един по един) и за свеки елемент изпълнява функцията, която сме му подали като аргумент (в примера по-горе това е функцията `iKnow`)

Функции от по-висок ред (HOF)

- Функция, която приема като аргумент друга функция и след това я използва за изчисляването на резултата си, се нарича Функция от по-висок ред (Higher order function)
- Функцията, която подаваме като аргумент, се нарича **callback** функция
- Функциите, които създаваме и съхраняваме в променлива или подаваме като аргументи, могат да бъдат **анонимни** (без наименование)

Пример

- Конструкцията **for**:

```
for (let i=0; i<array.length; i++) {  
    var element = array[i];  
    console.log(element);  
}
```

- и функцията **forEach()**:

```
array.forEach(function(element) {  
    console.log(element);  
});
```

- Са напълно аналогични. Разликата е, че във вторият пример използваме композиция на функции и за това казваме, че **forEach()** е функция от по-висок ред

Използване на анонимна функция като callback функция

- Ако имаме следния списък:

```
var array = ["John", "Eddie", "Anna"];
```

- И използваме следната композиция на функции:

```
function iKnow(name) {  
    console.log("I know", name);  
}  
  
array.forEach(iKnow);
```

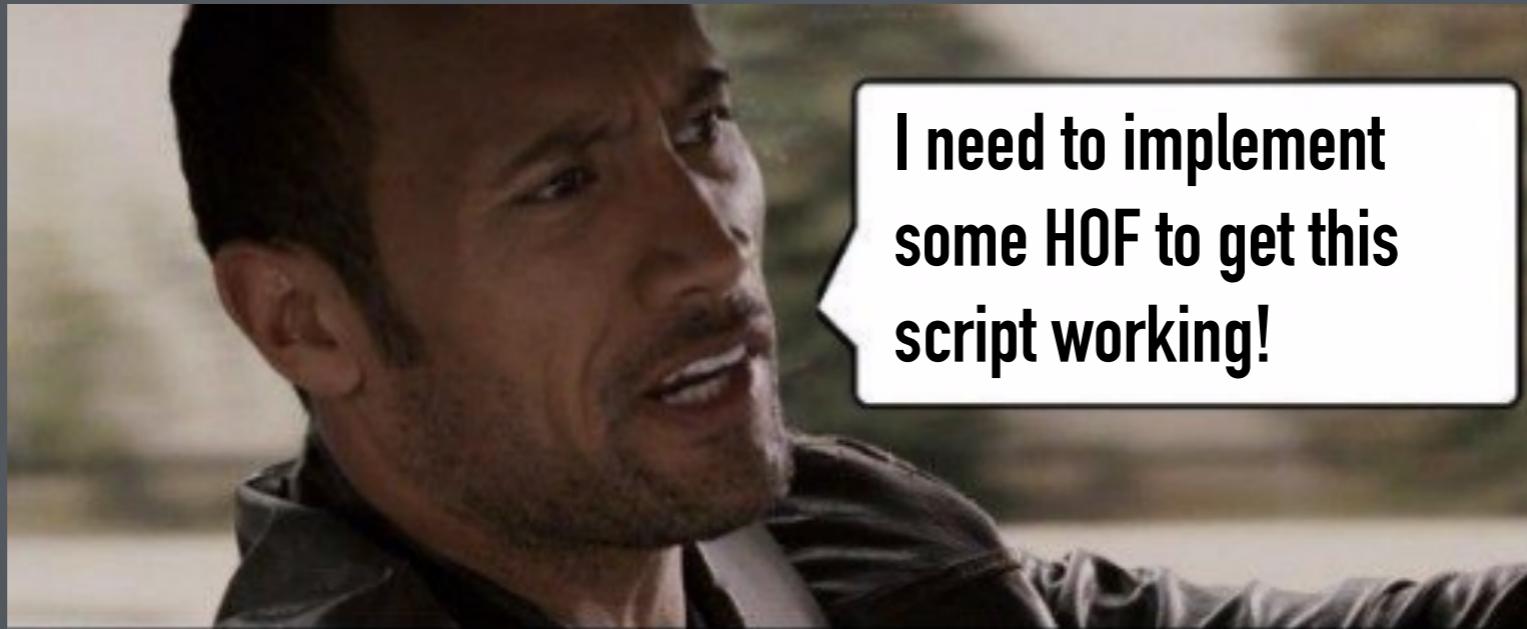
- Можем да си спестим създаването на именувана функция iKnow:

```
array.forEach(function (name) {  
    console.log("I know", name);  
});
```

Още функции на списъците, които използват композиция

- **find:** търсене в списък
- **map:** прилагане на едно и също действие върху всички елементи от списъка
- **reduce:** редуциране на списък (ES6)
- **filter:** филтриране на списък
- **sort:** сортиране на списък
- Пример:

```
array.filter(function (name) {  
    return name.length > 4;  
});
```



I need to implement
some HOF to get this
script working!



Why not using
many nested loops
instead?



Въпроси?

Упражнение

- Направете задачата, която изписва думата на обратно, като използвате `split`, `reverse` и `join`:
<https://repl.it/teacher/assignments/88090>
- Направете задачата, която обръща думите в едно изречение, като използвате тар и функцията `reverseWord` от горната задача:
<https://repl.it/teacher/assignments/88086>



KEEP
CALM
AND
LEARN
JAVASCRIPT

Полезни връзки

- **JavaScript best practices:**

https://www.w3.org/wiki/JavaScript_best_practices

(задължително го прочетете и научете, ако все още не сте)

- MDN документация:

[forEach](#)

[filter](#)

[reduce](#)

- Интересни (и малко шумни) видео уроци:

<https://www.youtube.com/watch?list=PL0zVEGEvSaeEd9hImCXrk5yUyqUagn84&v=BMUiFMZr7vk>

<https://www.youtube.com/channel/UCO1cgjhGzsSYb1rsB4bFe4Q/videos>

- Code wars

<https://www.codewars.com/dashboard>

Примери

<http://swift-academy.zenlabs.pro/lessons/lesson15/examples/download.zip>

obfuscate() решение: <https://repl.it/HObN/0>

Анонимни функции и IIFE: <https://repl.it/HOKk/3>

HOF: <https://repl.it/HO9g/0>

Домашно

<http://swift-academy.zenlabs.pro/lessons/lesson15/homework>