Functional JavaScript

marcus@gratex.com

FIIT , 2018

Functional JavaScript

- JavaScript is a multi-paradigm language
- It can be used to program functionally

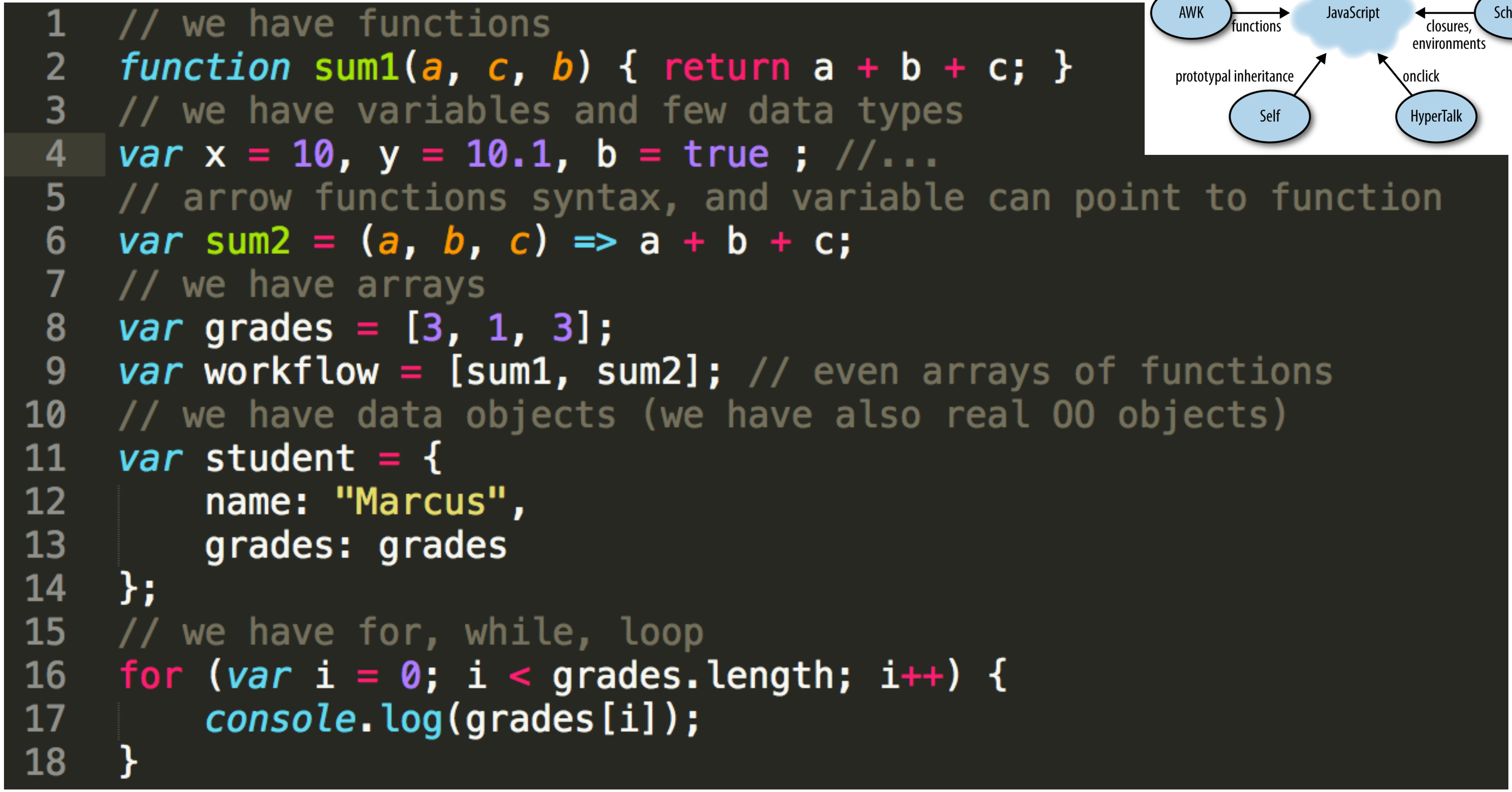
FP idioms:

- ➡ iterative functions, which can replace loops,
- ➡ list processing
- function manipulations
- immutability
- pure functions
- branching
- ... and many other things,

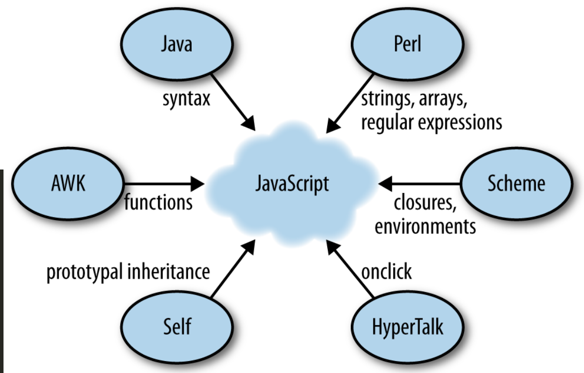
Can help us to keep code:

- smaller
- cleaner/readable/semantic
- testable
- reusable
- maintainable
- ….
- more fun

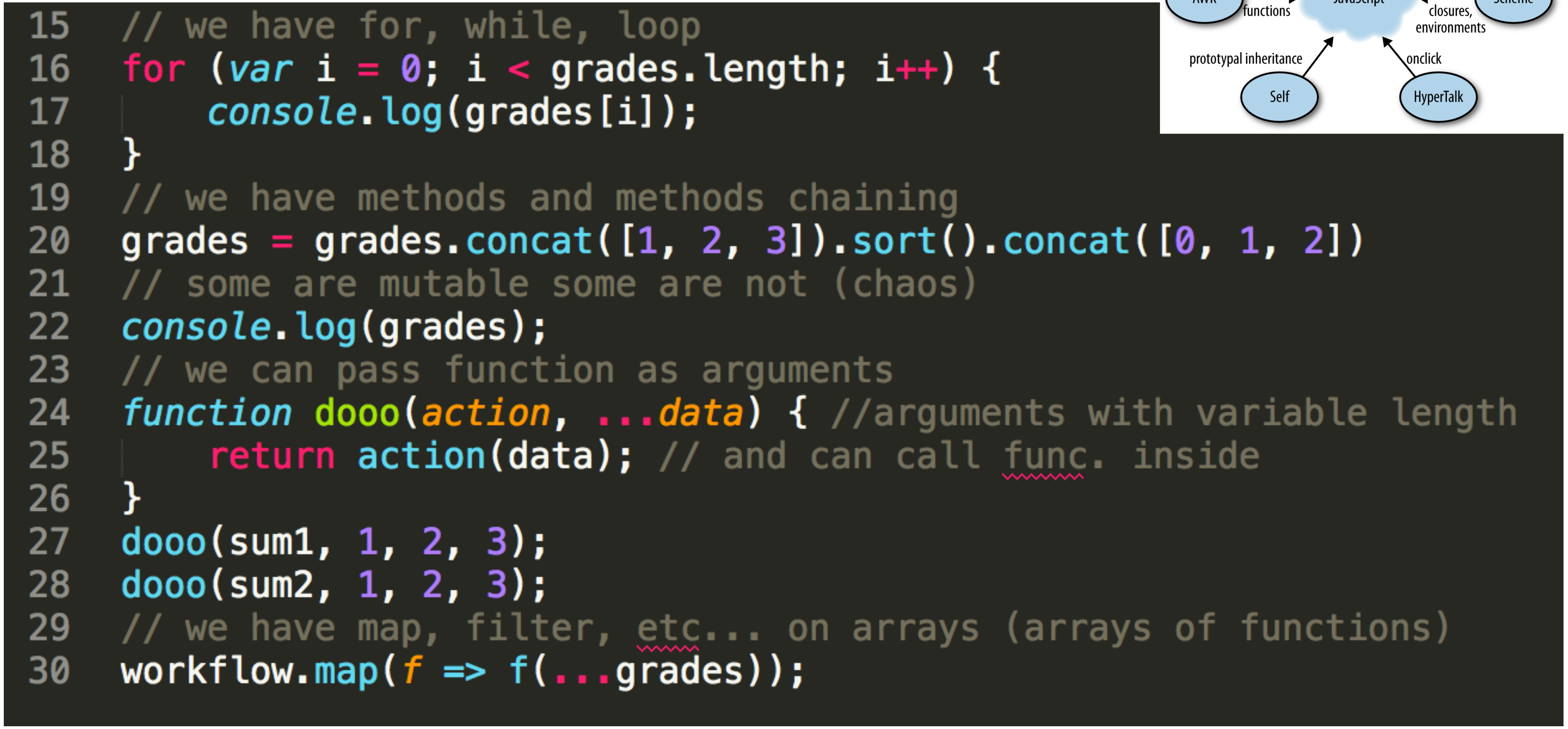
JavaScript (intro in 30 lines)



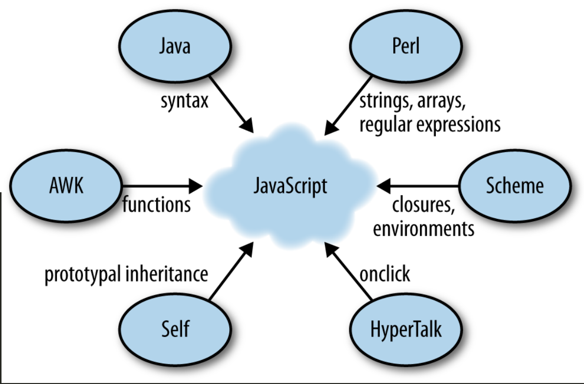
```
// we have functions
function sum1(a, c, b) { return a + b + c; }
// we have variables and few data types
var x = 10, y = 10.1, b = true ; //...
// arrow functions syntax, and variable can point to function
var sum2 = (a, b, c) => a + b + c;
// we have arrays
var grades = [3, 1, 3];
var workflow = [sum1, sum2]; // even arrays of functions
// we have data objects (we have also real OO objects)
var student = {
     name: "Marcus",
     grades: grades
};
// we have for, while, loop
for (var i = 0; i < grades.length; i++) {
     console.log(grades[i]);
}
```



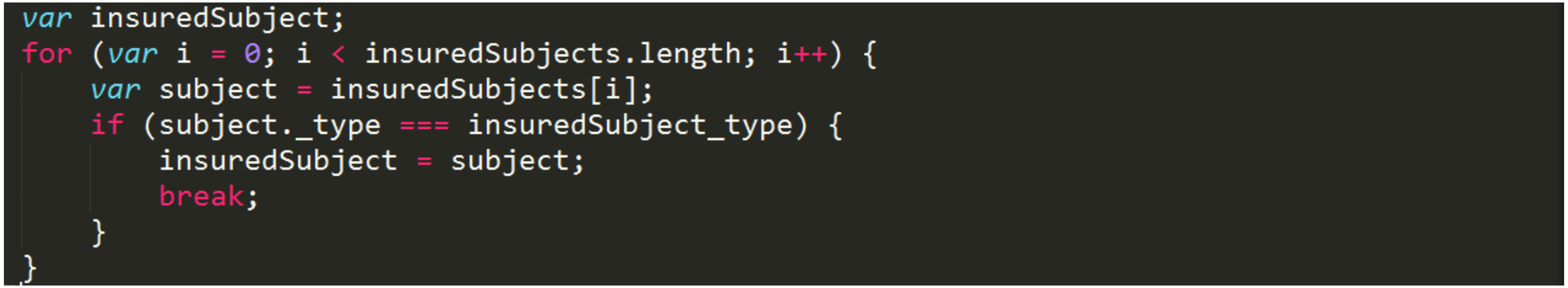
JavaScript (intro in 30 lines)



```javascript
// we have for, while, loop
for (var i = 0; i < grades.length; i++) {
    console.log(grades[i]);
}
// we have methods and methods chaining
grades = grades.concat([1, 2, 3]).sort().concat([0, 1, 2])
// some are mutable some are not (chaos)
console.log(grades);
// we can pass function as arguments
function dooo(action, ...data) { //arguments with variable length
    return action(data); // and can call func. inside
}
dooo(sum1, 1, 2, 3);
dooo(sum2, 1, 2, 3);
// we have map, filter, etc... on arrays (arrays of functions)
workflow.map(f => f(...grades));
```



JavaScript – "ugly" "for"



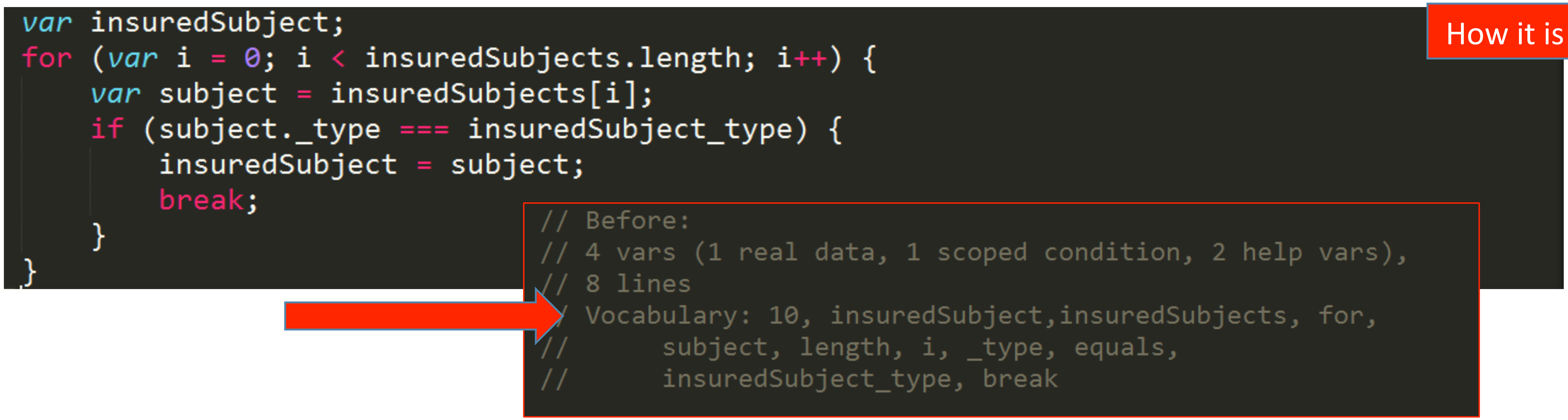
```
var insuredSubject;
for (var i = 0; i < insuredSubjects.length; i++) {
    var subject = insuredSubjects[i];
    if (subject._type === insuredSubject_type) {
        insuredSubject = subject;
        break;
    }
}
```

- for, while, do while
- they exist from Basic … Java
- Bad:
  - Verbose
  - Not semantic
  - Not reusable
- Good:
  - …

- keď sa pozriem na FOR cyklus neviem čo robí, lebo
  - robiť hocičo
  - veľa vecí naraz

- cyclomatic complexity
  - for, if,. for, while all nested

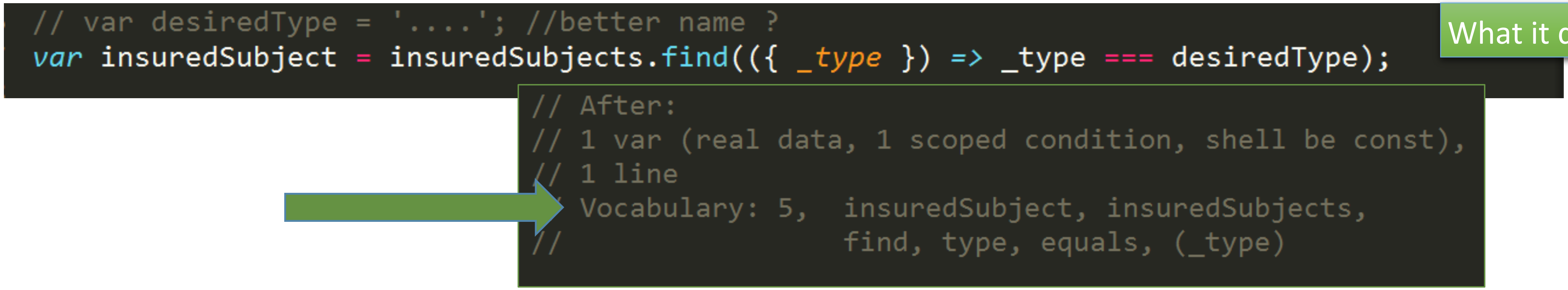
JavaScript - "ugly" "for"



How it is done

```
var insuredSubject;
for (var i = 0; i < insuredSubjects.length; i++) {
    var subject = insuredSubjects[i];
    if (subject._type === insuredSubject_type) {
        insuredSubject = subject;
        break;
    }
}
```

```
// Before:
// 4 vars (1 real data, 1 scoped condition, 2 help vars),
// 8 lines
/ Vocabulary: 10, insuredSubject,insuredSubjects, for,
//      subject, length, i, _type, equals,
//      insuredSubject_type, break
```



What it does

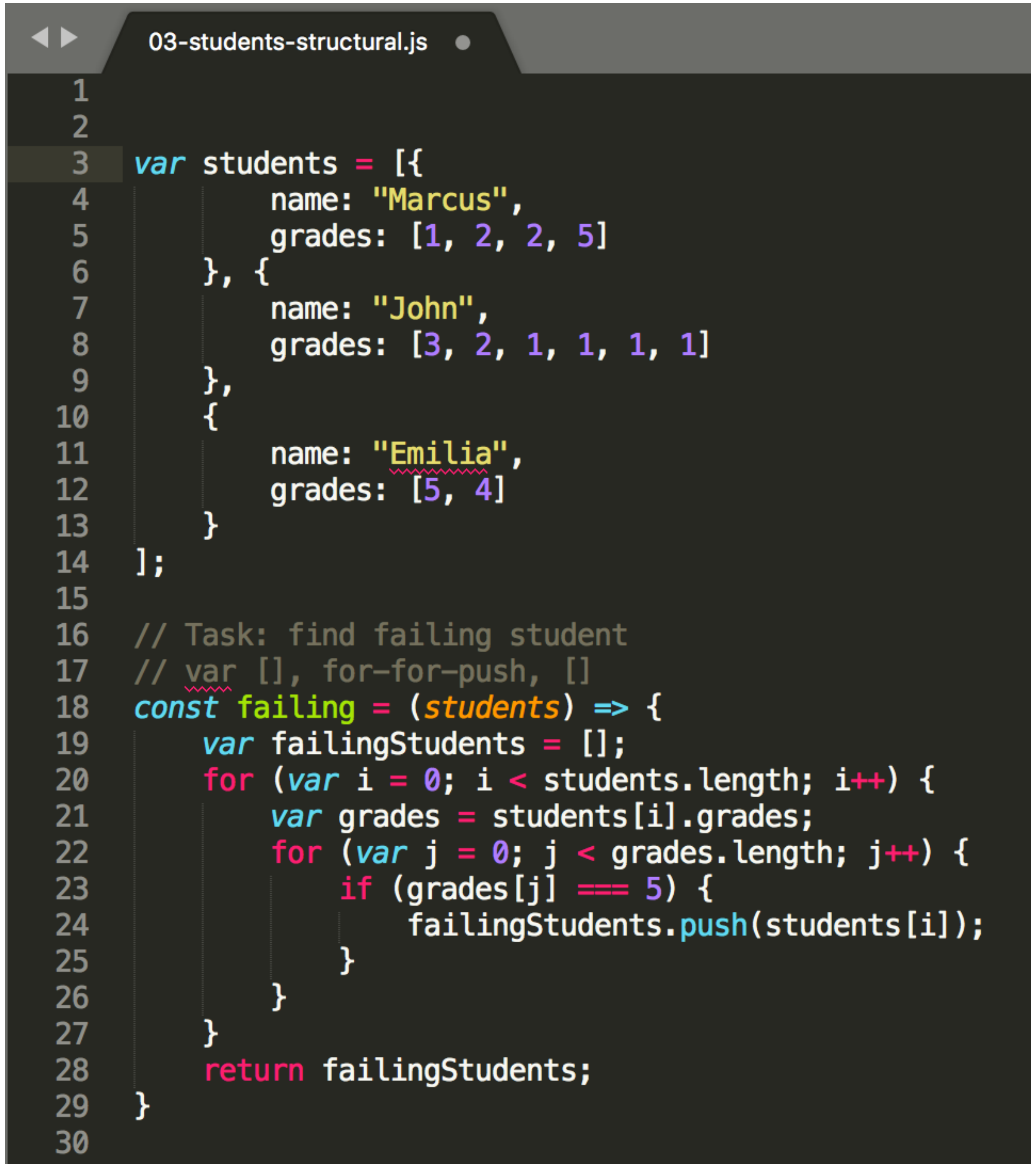
```
 // var desiredType = '....'; //better name ?
 var insuredSubject = insuredSubjects.find(({ _type }) => _type === desiredType);
```

```
// After:
// 1 var (real data, 1 scoped condition, shell be const),
// 1 line
/ Vocabulary: 5,  insuredSubject, insuredSubjects,
//                find, type, equals, (_type)
```

JavaScript - array extras vs ugly “for”

mapping variants of for, to semantic methods

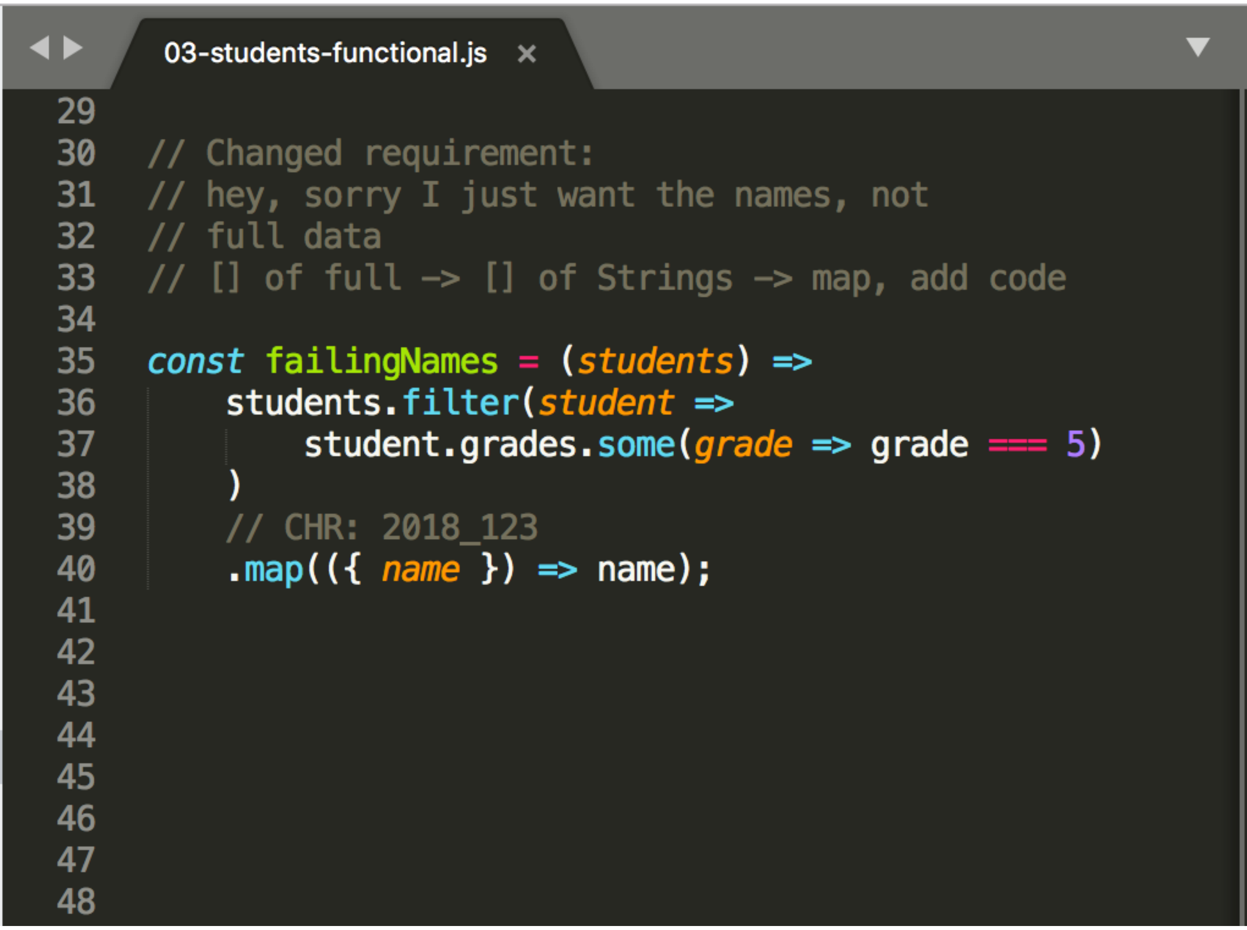
Function	In	Out	loop eq.	ES
map	[], N	[], N	var [], for, push, return []	
filter	[], N	[], M<N	var [], for, if, push, return []	
reduce	[]	{}, [], whatever,…	var [], for, if, push, return {}	
reduceRight	[]	{}, [], whatever,…	var [], for (i--), if, push, return {}	
some	[]	boolean	var b, for, if return true	
every	[]	boolean	var b, for, if return false	
forEach	[]		for	
find	[] of items	item	for, if return a[i]; return;	
fill	[], item	[] of items	var [], for, push, return []	
from	[], iterable	[]	var [], for, push, return []	



03-students-structural.js

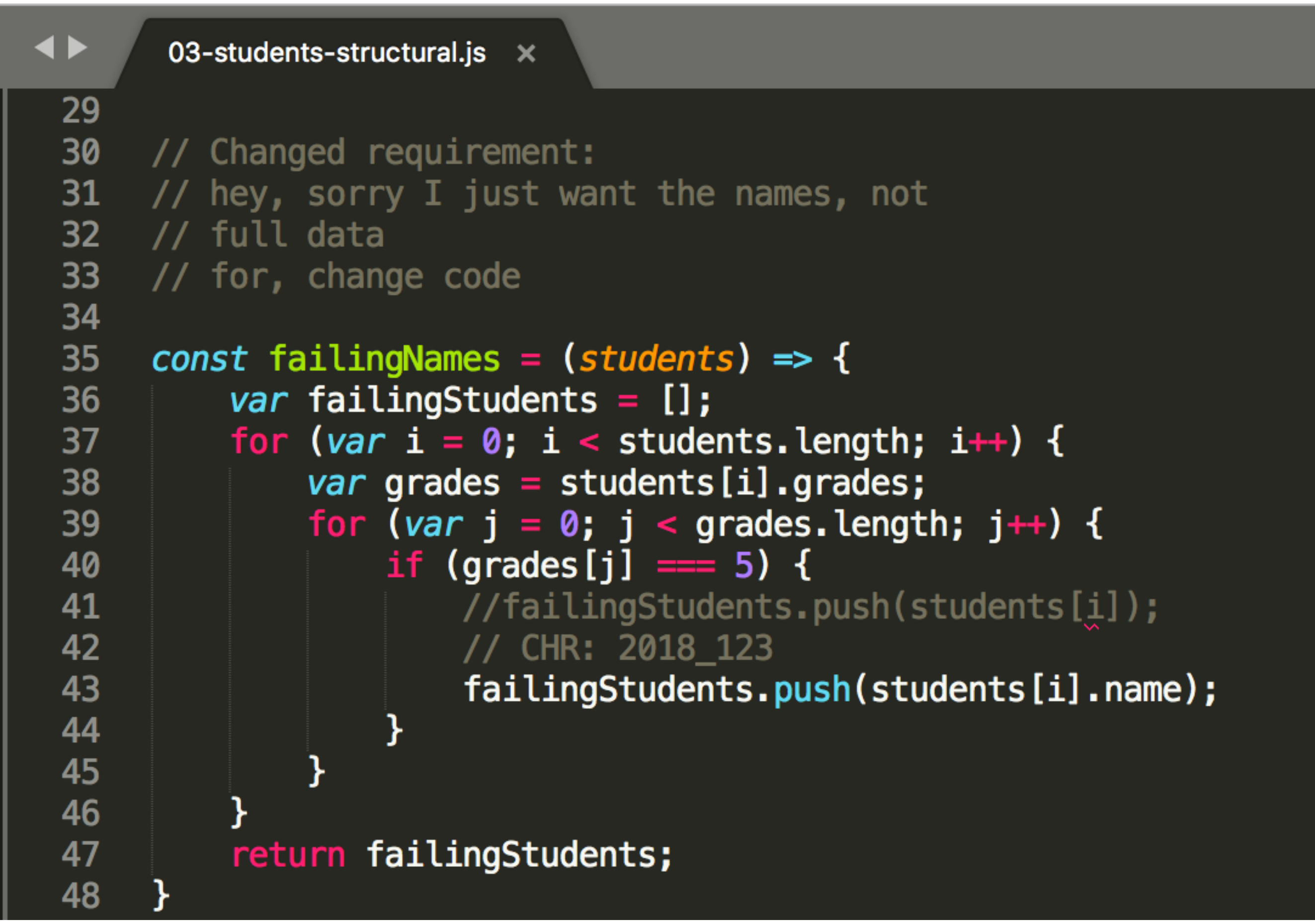
```javascript


var students = [{
        name: "Marcus",
        grades: [1, 2, 2, 5]
    }, {
        name: "John",
        grades: [3, 2, 1, 1, 1, 1]
    },
    {
        name: "Emilia",
        grades: [5, 4]
    }
];

// Task: find failing student
// var [], for-for-push, []
const failing = (students) => {
    var failingStudents = [];
    for (var i = 0; i < students.length; i++) {
        var grades = students[i].grades;
        for (var j = 0; j < grades.length; j++) {
            if (grades[j] === 5) {
                failingStudents.push(students[i]);
            }
        }
    }
    return failingStudents;
}

```

JavaScript - Array Extras vs ugly “for”



03-students-functional.js

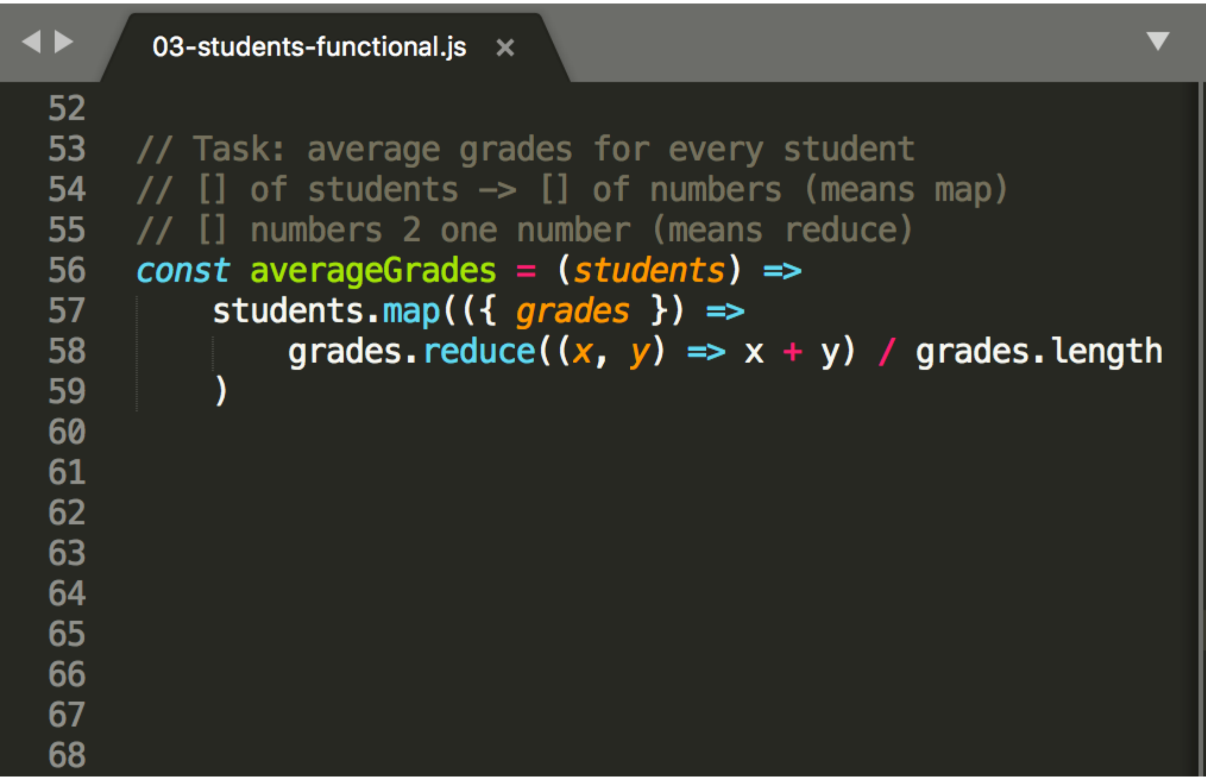
```javascript

// Changed requirement:
// hey, sorry I just want the names, not
// full data
// [] of full -> [] of Strings -> map, add code

const failingNames = (students) =>
    students.filter(student =>
        student.grades.some(grade => grade === 5)
    )
    // CHR: 2018_123
    .map(({ name }) => name);
```



03-students-structural.js

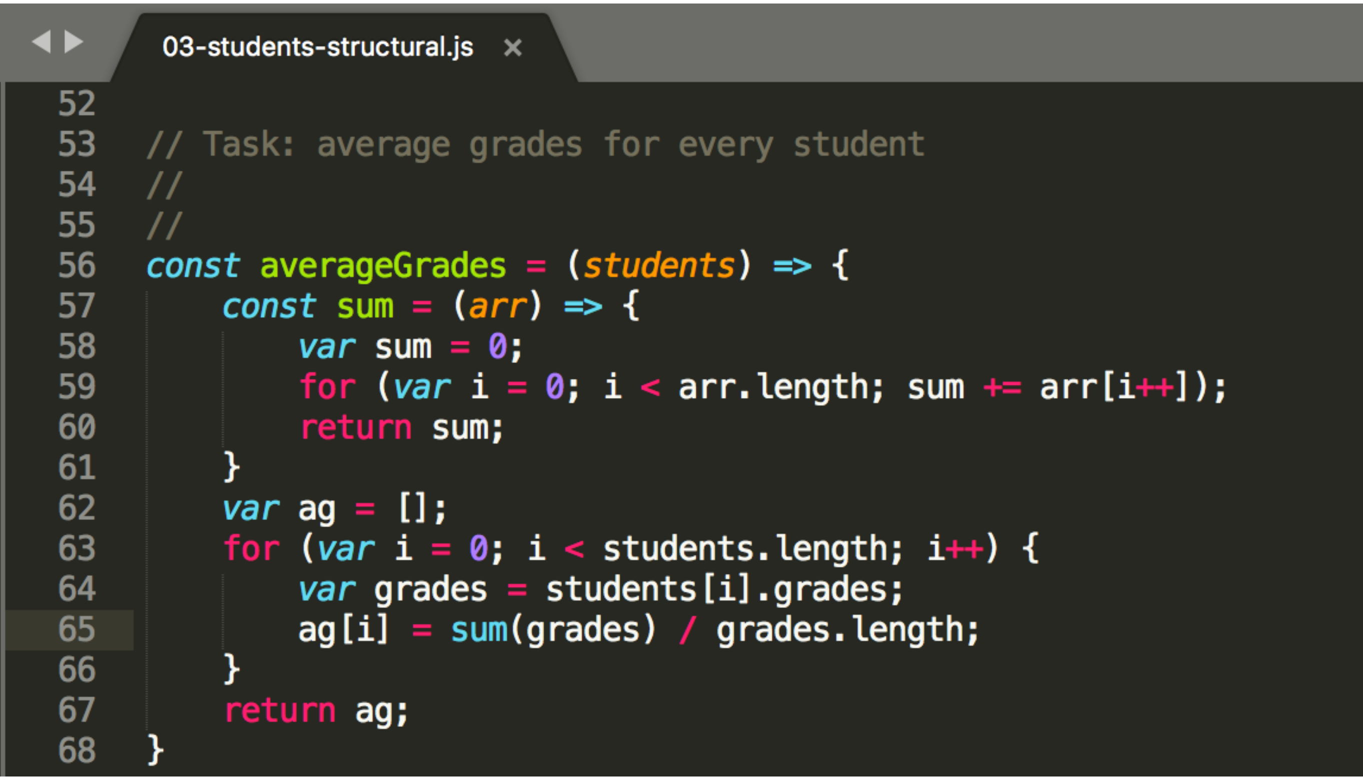
```javascript

// Changed requirement:
// hey, sorry I just want the names, not
// full data
// for, change code

const failingNames = (students) => {
    var failingStudents = [];
    for (var i = 0; i < students.length; i++) {
        var grades = students[i].grades;
        for (var j = 0; j < grades.length; j++) {
            if (grades[j] === 5) {
                //failingStudents.push(students[i]);
                // CHR: 2018_123
                failingStudents.push(students[i].name);
            }
        }
    }
    return failingStudents;
}
```

JavaScript - Array Extras vs ugly “for”



03-students-functional.js

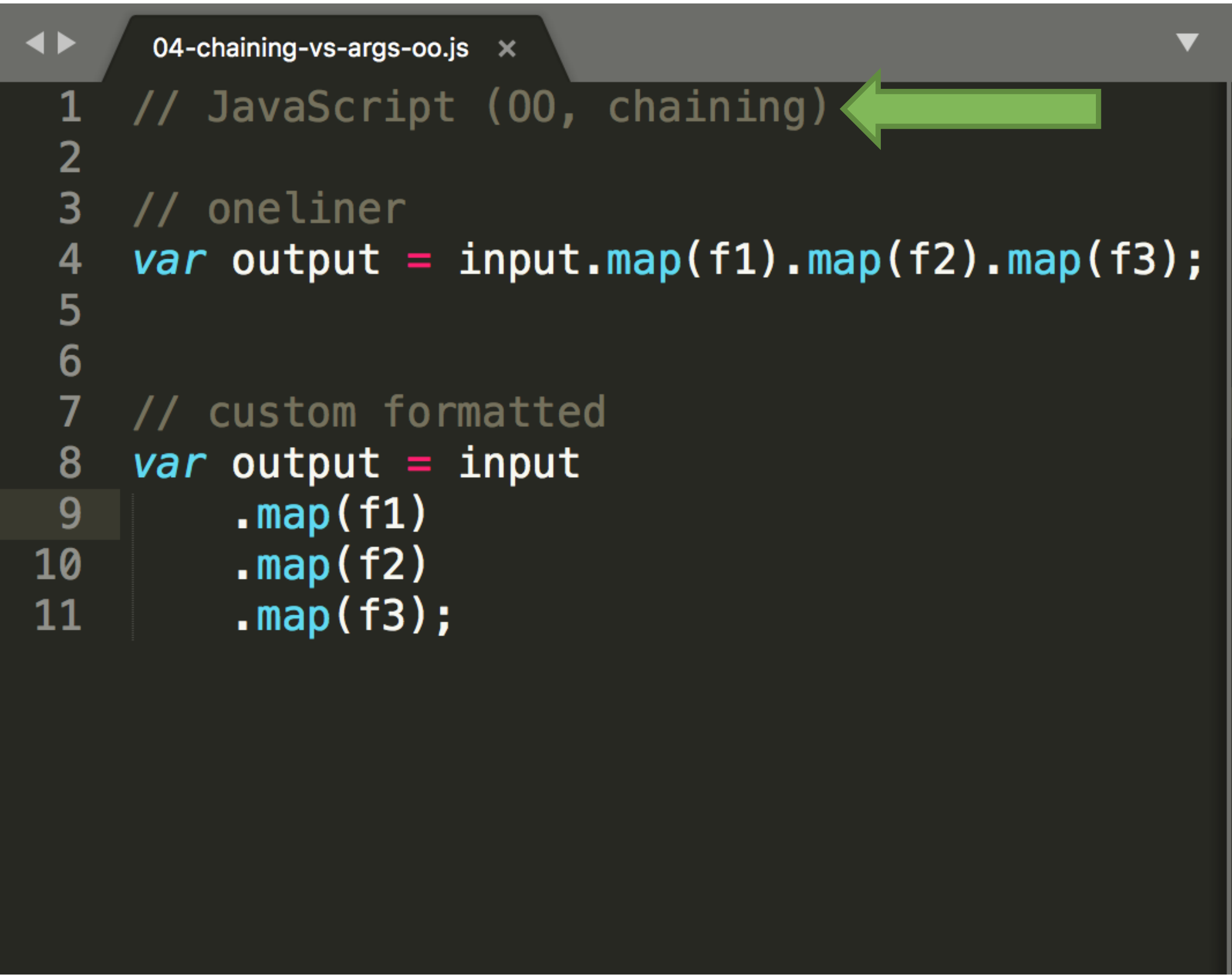
```javascript

// Task: average grades for every student
// [] of students -> [] of numbers (means map)
// [] numbers 2 one number (means reduce)
const averageGrades = (students) =>
    students.map(({ grades }) =>
        grades.reduce((x, y) => x + y) / grades.length
    )
```



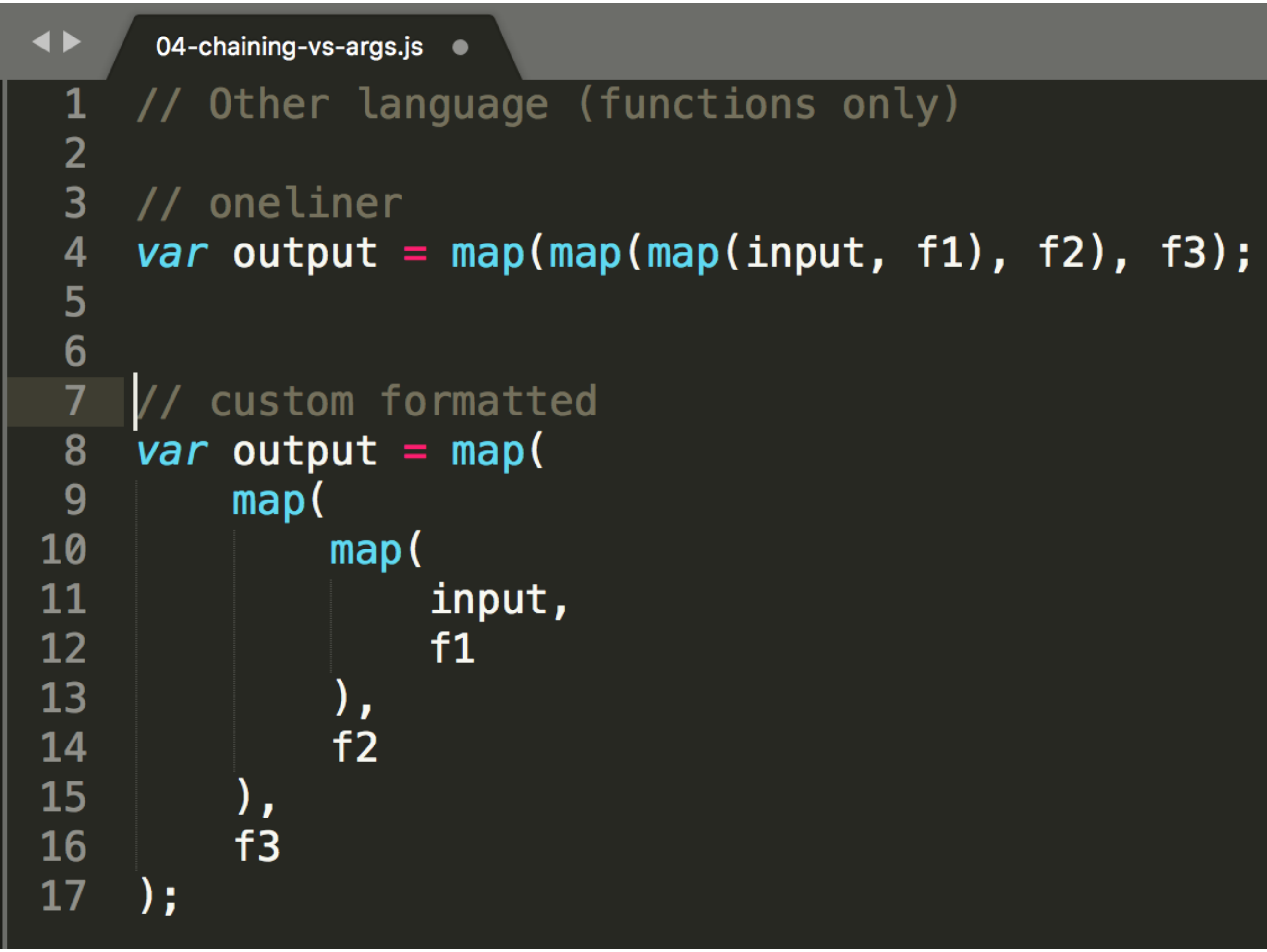
03-students-structural.js

```javascript

// Task: average grades for every student
//
//
const averageGrades = (students) => {
    const sum = (arr) => {
        var sum = 0;
        for (var i = 0; i < arr.length; sum += arr[i++]);
        return sum;
    }
    var ag = [];
    for (var i = 0; i < students.length; i++) {
        var grades = students[i].grades;
        ag[i] = sum(grades) / grades.length;
    }
    return ag;
}
```

JavaScript - best of both worlds

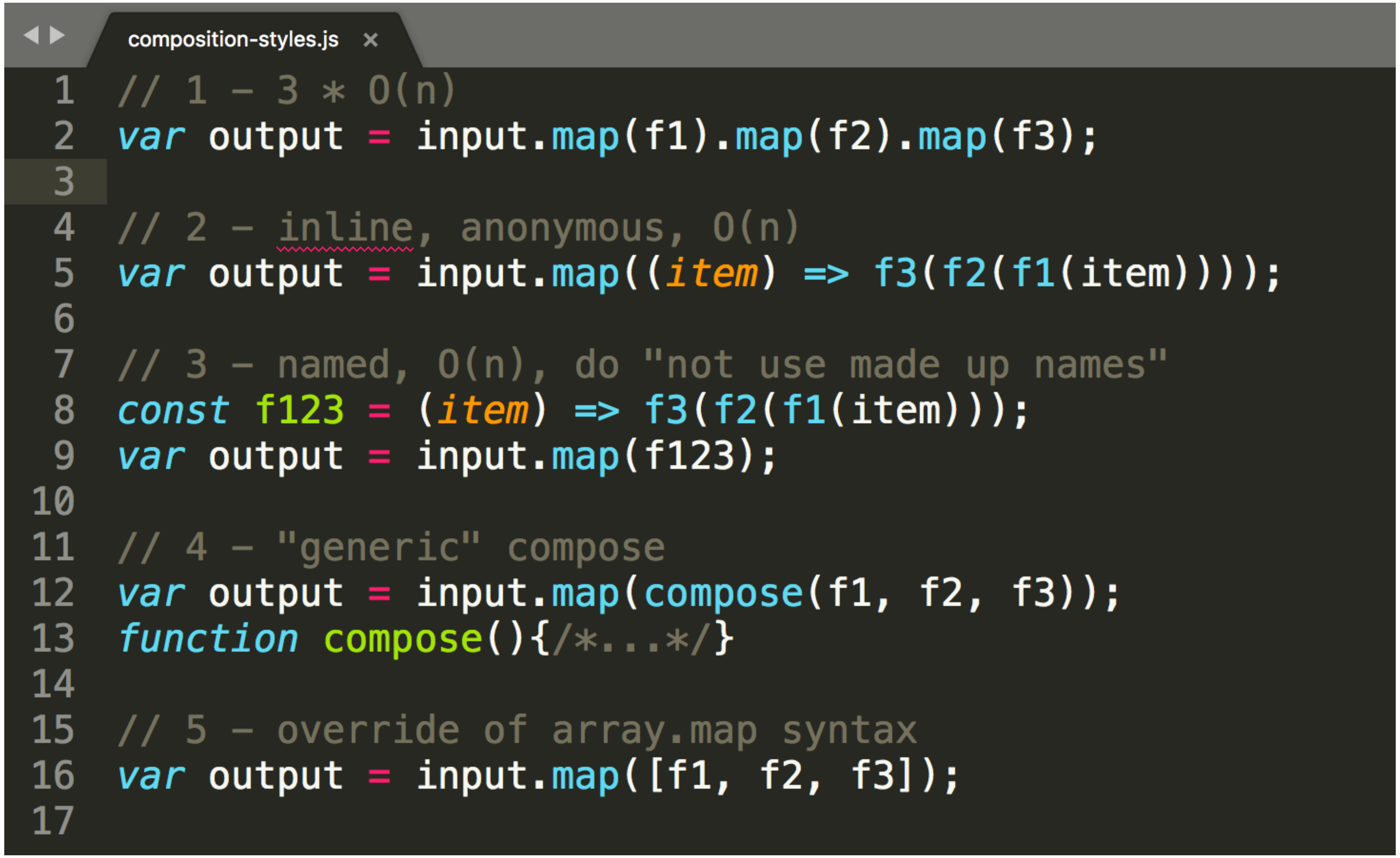


```
// JavaScript (OO, chaining)

// oneliner
var output = input.map(f1).map(f2).map(f3);


// custom formatted
var output = input
    .map(f1)
    .map(f2)
    .map(f3);
```



```
// Other language (functions only)

// oneliner
var output = map(map(map(input, f1), f2), f3);


// custom formatted
var output = map(
    map(
        map(
            input,
            f1
        ),
        f2
    ),
    f3
);
```

Composition styles inline, adhoc, generic



composition-styles.js ×

```js
// 1 - 3 * O(n)
var output = input.map(f1).map(f2).map(f3);

// 2 - inline, anonymous, O(n)
var output = input.map((item) => f3(f2(f1(item))));

// 3 - named, O(n), do "not use made up names"
const f123 = (item) => f3(f2(f1(item)));
var output = input.map(f123);

// 4 - "generic" compose
var output = input.map(compose(f1, f2, f3));
function compose(){/*...*/}

// 5 - override of array.map syntax
var output = input.map([f1, f2, f3]);
```

readable
reused,
slow 400ms

readable,
reused
fast 120ms

readable,
reused
fast 120ms

readable,
reused
fast 153ms
slow 300ms

readable
reused,
non standard

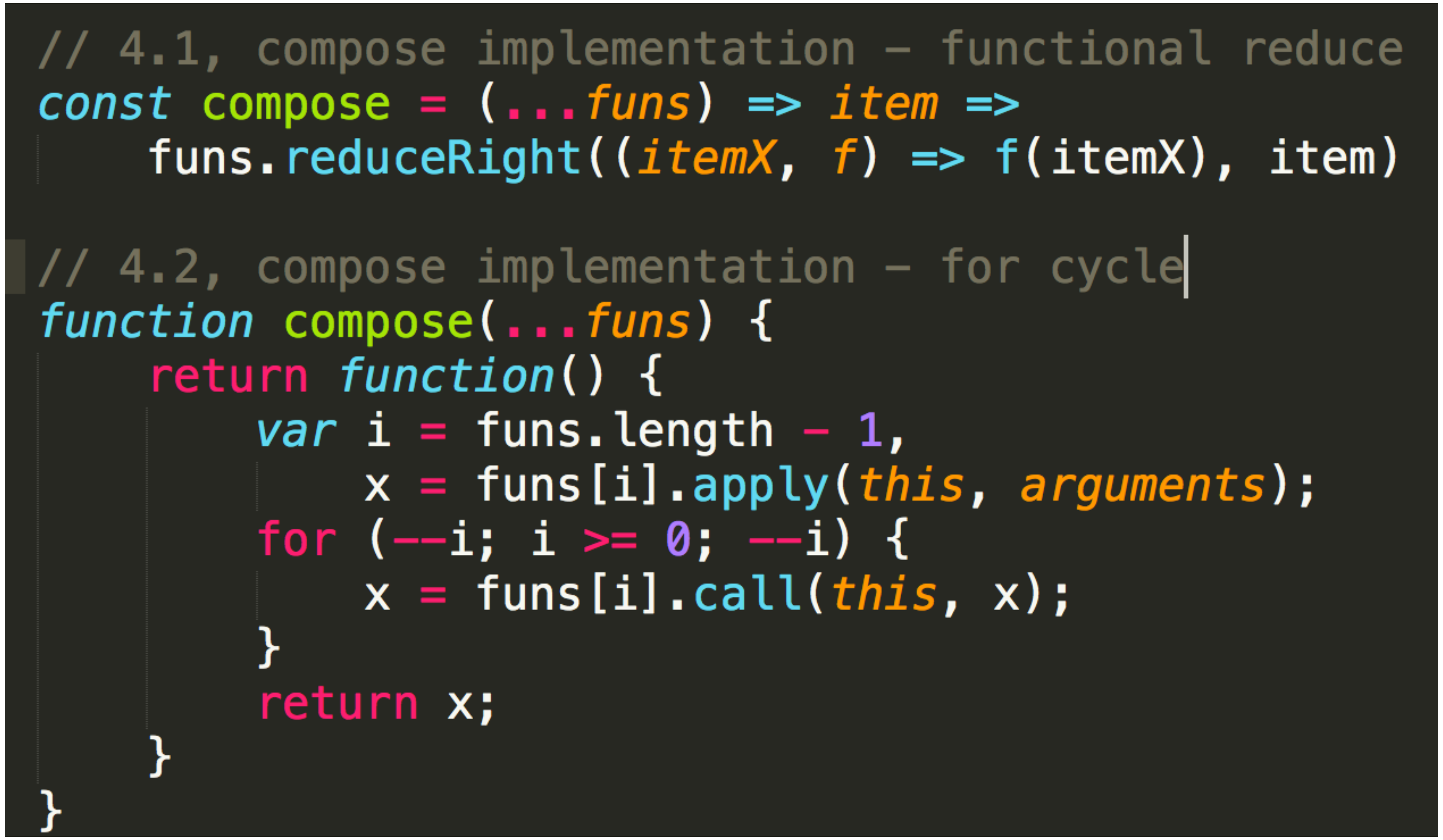
Composition - compose() implementations

JavaScript language does not have API for "generic compose" or "construct"

Observe these 2 sample implementations.

still naive,
not exact semantics as map.map.map
because of (i, items).

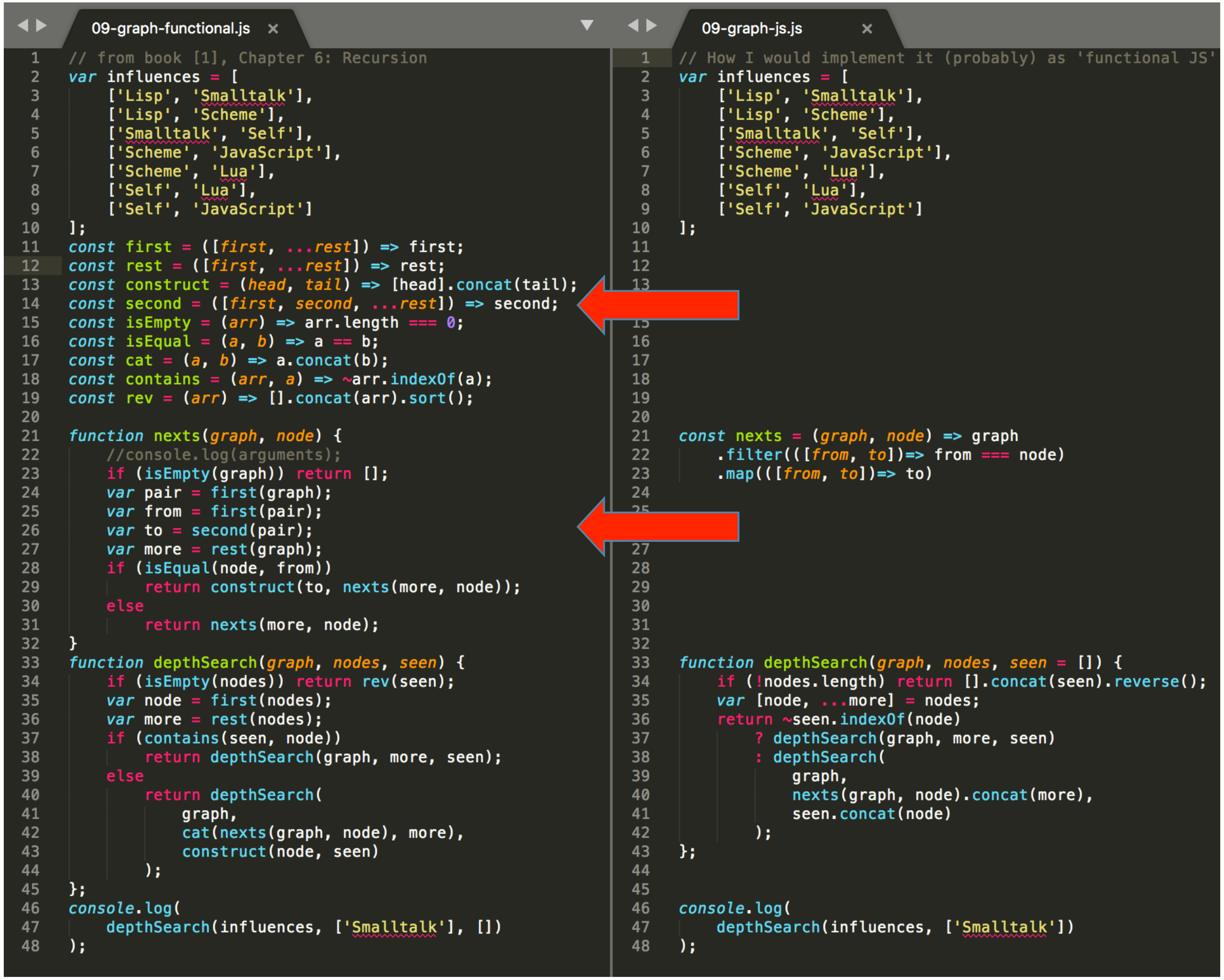
Just as an example of performance of good old "for" [1]



```
// 4.1, compose implementation - functional reduce
const compose = (...funs) => item =>
    funs.reduceRight((itemX, f) => f(itemX), item)

// 4.2, compose implementation - for cycle
function compose(...funs) {
    return function() {
        var i = funs.length - 1,
            x = funs[i].apply(this, arguments);
        for (--i; i >= 0; --i) {
            x = funs[i].call(this, x);
        }
        return x;
    }
}
```

300ms

153ms



```javascript
// from book [1], Chapter 6: Recursion
var influences = [
    ['Lisp', 'Smalltalk'],
    ['Lisp', 'Scheme'],
    ['Smalltalk', 'Self'],
    ['Scheme', 'JavaScript'],
    ['Scheme', 'Lua'],
    ['Self', 'Lua'],
    ['Self', 'JavaScript']
];
const first = ([first, ...rest]) => first;
const rest = ([first, ...rest]) => rest;
const construct = (head, tail) => [head].concat(tail);
const second = ([first, second, ...rest]) => second;
const isEmpty = (arr) => arr.length === 0;
const isEqual = (a, b) => a == b;
const cat = (a, b) => a.concat(b);
const contains = (arr, a) => ~arr.indexOf(a);
const rev = (arr) => [].concat(arr).sort();

function nexts(graph, node) {
    //console.log(arguments);
    if (isEmpty(graph)) return [];
    var pair = first(graph);
    var from = first(pair);
    var to = second(pair);
    var more = rest(graph);
    if (isEqual(node, from))
        return construct(to, nexts(more, node));
    else
        return nexts(more, node);
}
function depthSearch(graph, nodes, seen) {
    if (isEmpty(nodes)) return rev(seen);
    var node = first(nodes);
    var more = rest(nodes);
    if (contains(seen, node))
        return depthSearch(graph, more, seen);
    else
        return depthSearch(
            graph,
            cat(nexts(graph, node), more),
            construct(node, seen)
        );
};
console.log(
    depthSearch(influences, ['Smalltalk'], [])
);
```

```javascript
// How I would implement it (probably) as 'functional JS'
var influences = [
    ['Lisp', 'Smalltalk'],
    ['Lisp', 'Scheme'],
    ['Smalltalk', 'Self'],
    ['Scheme', 'JavaScript'],
    ['Scheme', 'Lua'],
    ['Self', 'Lua'],
    ['Self', 'JavaScript']
];

const nexts = (graph, node) => graph
    .filter(([from, to])=> from === node)
    .map(([from, to])=> to)

function depthSearch(graph, nodes, seen = []) {
    if (!nodes.length) return [].concat(seen).reverse();
    var [node, ...more] = nodes;
    return ~seen.indexOf(node)
        ? depthSearch(graph, more, seen)
        : depthSearch(
            graph,
            nexts(graph, node).concat(more),
            seen.concat(node)
        );
};

console.log(
    depthSearch(influences, ['Smalltalk'])
);
```

JavaScript - too much functional ? Alebo čo sa stane, keď to "preženiete".

- is this still JavaScript ?

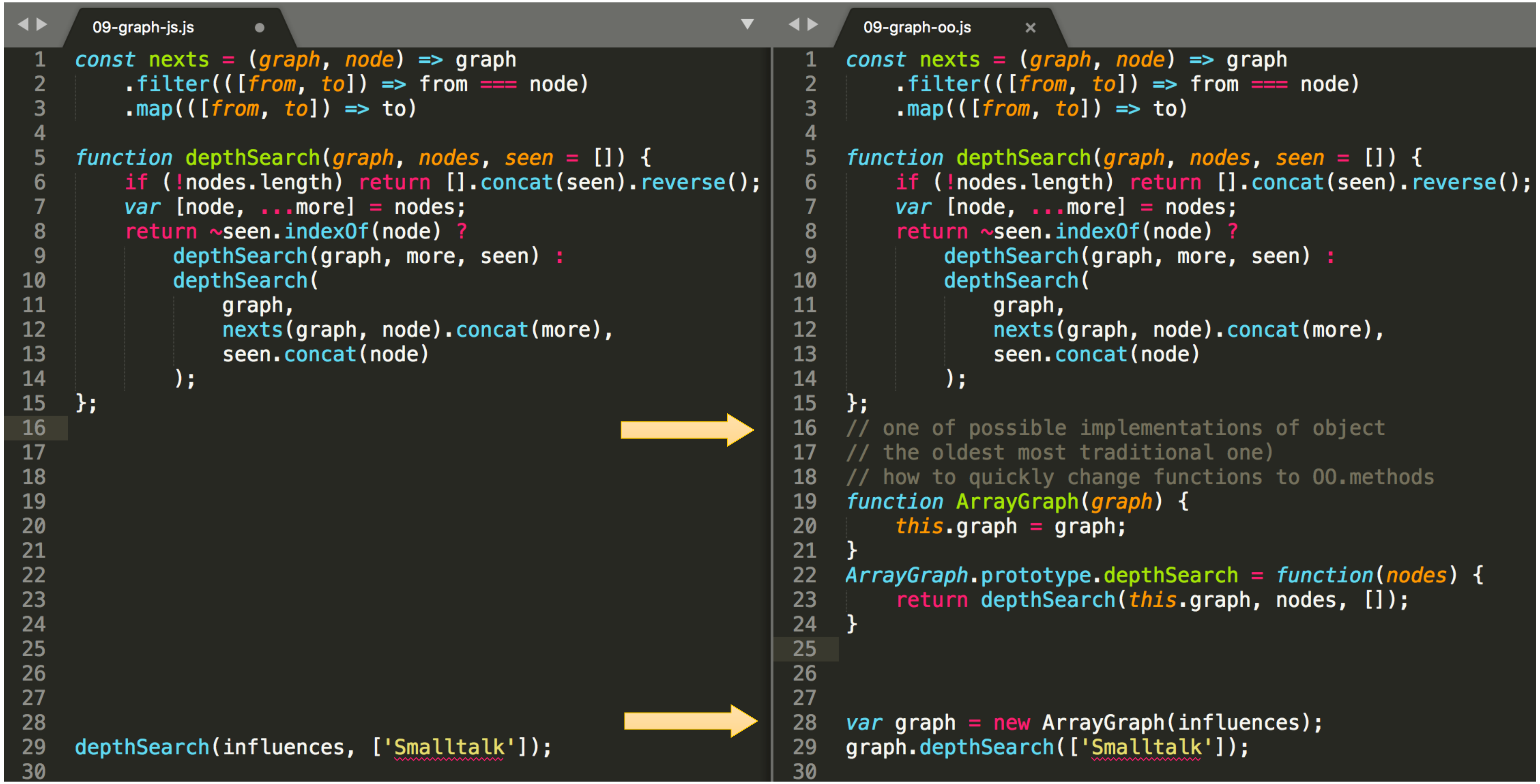
how many JS people

- will have to read it,
- and will understand ?

- Do you want to study
    - custom APIs (vocabularies) or use
    - "idiomatic JS" (for common oneliners) ?
- is recursion really "best" for this structure parsing ?

- is functional really best for this ? (internal data structure hiding)
    - which functions are really reusable ?

JavaScript - from functions to object+methods



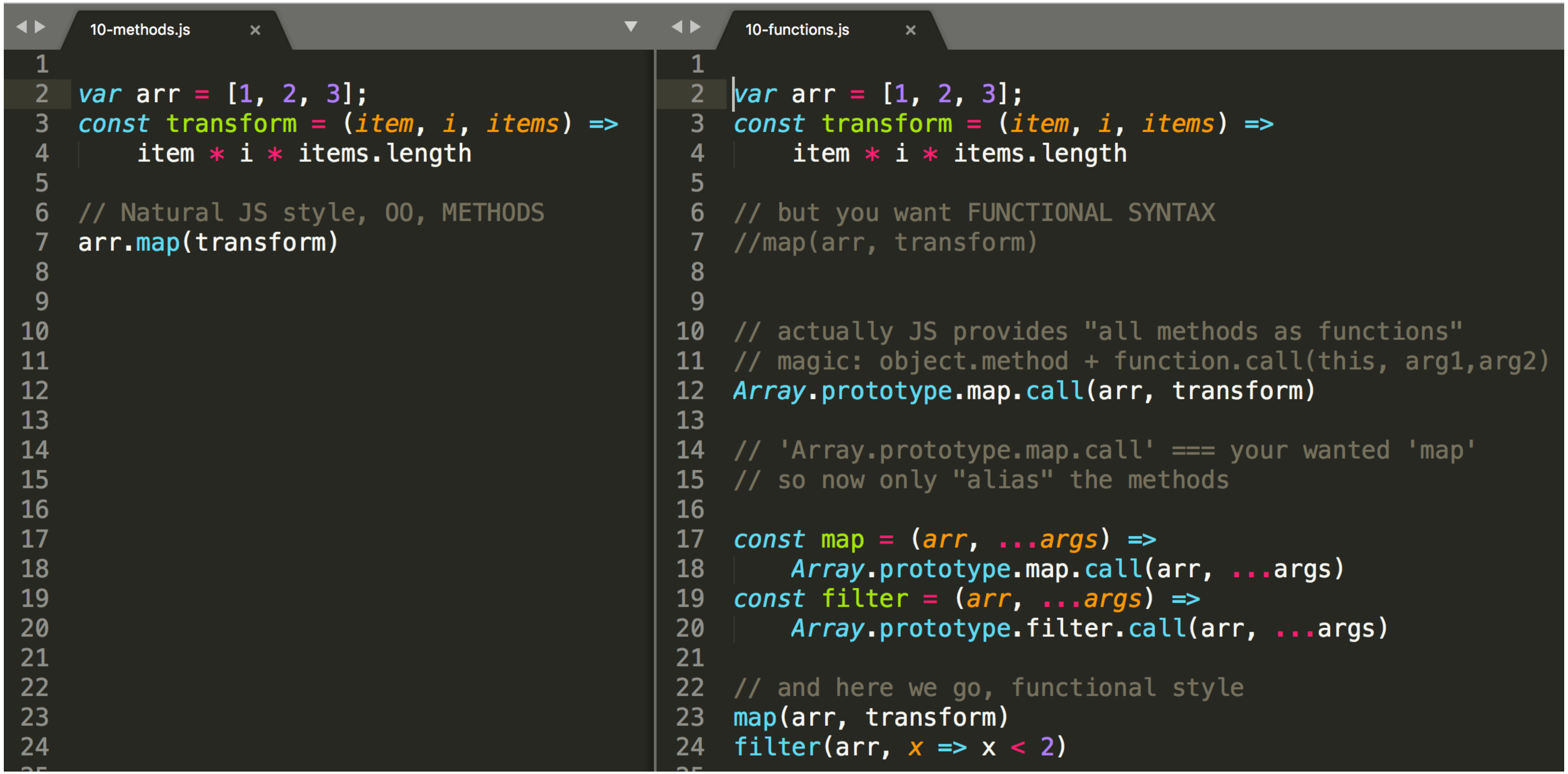
09-graph-js.js

```javascript
const nexts = (graph, node) => graph
    .filter(([from, to]) => from === node)
    .map(([from, to]) => to)

function depthSearch(graph, nodes, seen = []) {
    if (!nodes.length) return [].concat(seen).reverse();
    var [node, ...more] = nodes;
    return ~seen.indexOf(node) ?
        depthSearch(graph, more, seen) :
        depthSearch(
            graph,
            nexts(graph, node).concat(more),
            seen.concat(node)
        );
};













depthSearch(influences, ['Smalltalk']);

```

09-graph-oo.js

```javascript
const nexts = (graph, node) => graph
    .filter(([from, to]) => from === node)
    .map(([from, to]) => to)

function depthSearch(graph, nodes, seen = []) {
    if (!nodes.length) return [].concat(seen).reverse();
    var [node, ...more] = nodes;
    return ~seen.indexOf(node) ?
        depthSearch(graph, more, seen) :
        depthSearch(
            graph,
            nexts(graph, node).concat(more),
            seen.concat(node)
        );
};
// one of possible implementations of object
// the oldest most traditional one)
// how to quickly change functions to OO.methods
function ArrayGraph(graph) {
    this.graph = graph;
}
ArrayGraph.prototype.depthSearch = function(nodes) {
    return depthSearch(this.graph, nodes, []);
}



var graph = new ArrayGraph(influences);
graph.depthSearch(['Smalltalk']);

```

JavaScript - from methods to functions



```
var arr = [1, 2, 3];
const transform = (item, i, items) =>
    item * i * items.length

// Natural JS style, OO, METHODS
arr.map(transform)
```

```
var arr = [1, 2, 3];
const transform = (item, i, items) =>
    item * i * items.length

// but you want FUNCTIONAL SYNTAX
//map(arr, transform)


// actually JS provides "all methods as functions"
// magic: object.method + function.call(this, arg1,arg2)
Array.prototype.map.call(arr, transform)

// 'Array.prototype.map.call' === your wanted 'map'
// so now only "alias" the methods

const map = (arr, ...args) =>
    Array.prototype.map.call(arr, ...args)
const filter = (arr, ...args) =>
    Array.prototype.filter.call(arr, ...args)

// and here we go, functional style
map(arr, transform)
filter(arr, x => x < 2)
```

Functional JavaScript (Summary)

- JavaScript is a multi-paradigm language
- It can be used to program functionally (specially with “modern JS”)
- But (my opinion, my current “functional JS POV”), so far on covered topics
  - use **functional concepts for business**, keep the rest “as needed” (procedural, declarative)
  - use **functional concepts for functional problems**
  - use **chaining** (nicer syntax, more readable programs)
  - use (parameter) **destructuring**, rest parameters, defaults (less bloated code, less need for low level FP primitives, branching)
  - use (study) **JS syntax**, if exists whenever possible, do **not hide known JS** under unknown libs
  - **do not implement low level functional features** by reusing functional features (e.g use raw loops to implement _ranges, etc…, beware performance, O(n), call stack price, memory)
  - do **not implement** functional low level features, **use libs** (eg. _underescore.js), use only what needed, more and more is replaced by standard JS syntax)
  - use **arrow functions only for inlines** (I like hoisting, more readable programs, top down reading, code first, then functions, function(){}, vs const=()=>{})
  - do not follow blindly FP concepts eg. “using functions instead of values” f(f()) vs f(v), we can have f(funOrValue)
  - **use recursion only where appropriate** for “problem solving”, **reduce is almost always fine** if not needing quick exits
  - implement **mappers, filters, reducers**, instead of implementing “whole methods”