Think functionally

•••

$f(x) = y$

Presenters

Pavel Struhar
pavel.struhar@accenture.com
Twitter: @pavestru

Daniel Derevjanik
daniel.derevjanik@accenture.com
Twitter: @dderevjanik

Both from Lightweight Arch. core team

EVERYTHING IS
AWESOME!

Spoiler

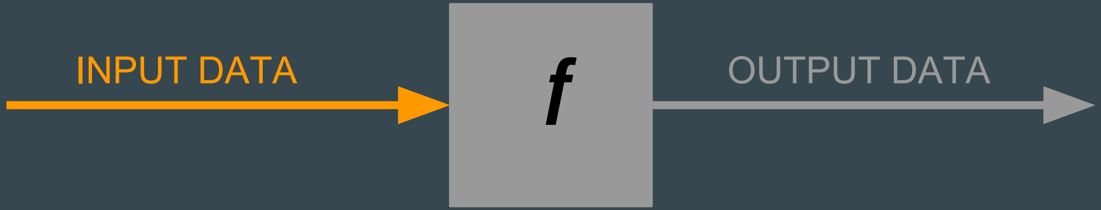
Everything is a function

OO pattern/principle	FP pattern/principle
Single Responsibility Principle	Functions
Open/Closed principle	Functions
Dependency Inversion Principle	Functions, also
Interface Segregation Principle	Functions
Factory pattern	Yes, functions
Strategy pattern	Oh my, functions again!
Decorator pattern	Functions
Visitor pattern	Functions □

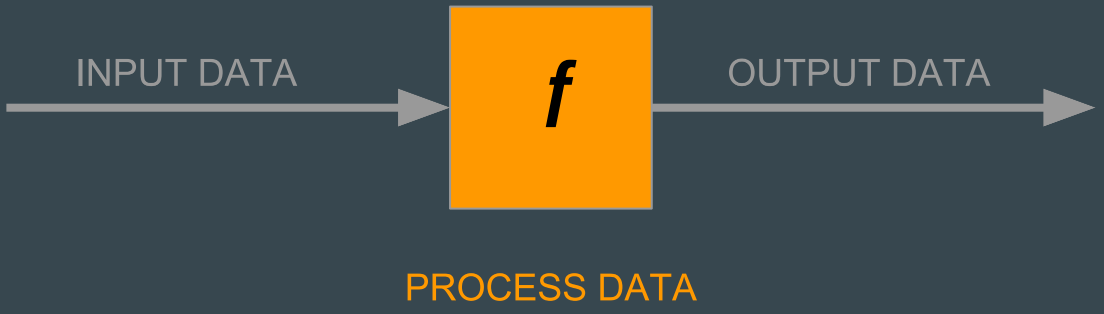
A monad is just a monoid in the category of endofunctors.
What's the problem?

Data and Functions

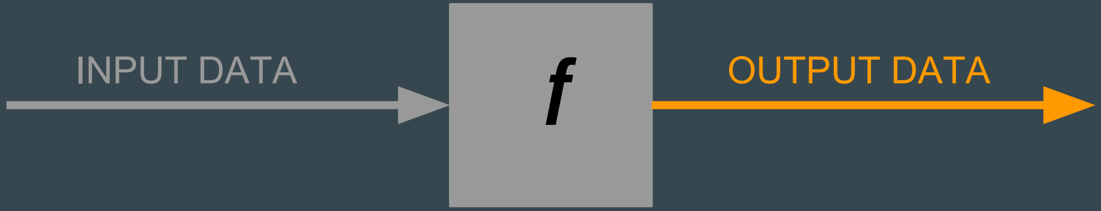
Data Flow - Expectations



Data Flow - Expectations

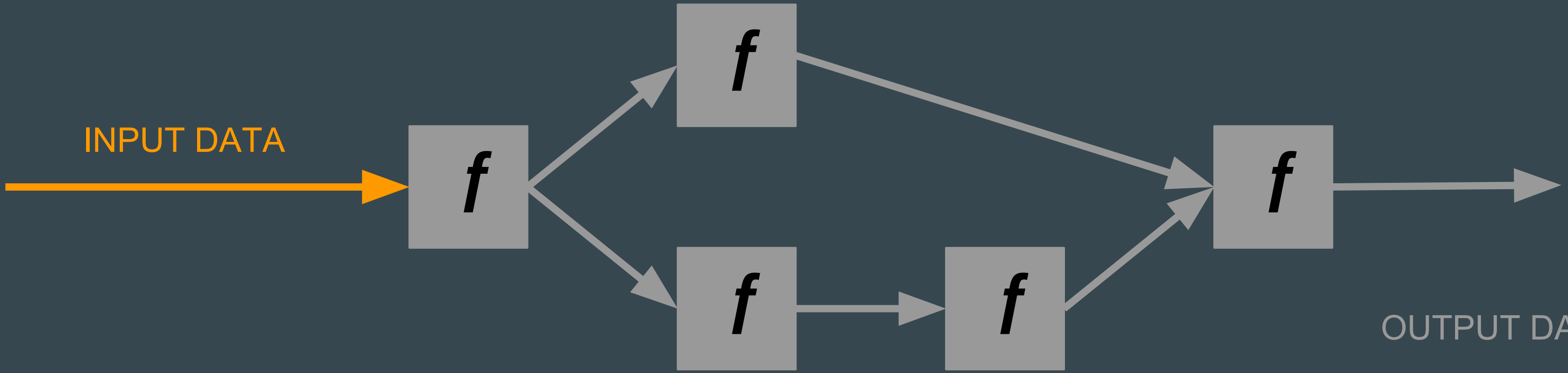


Data Flow - Expectations

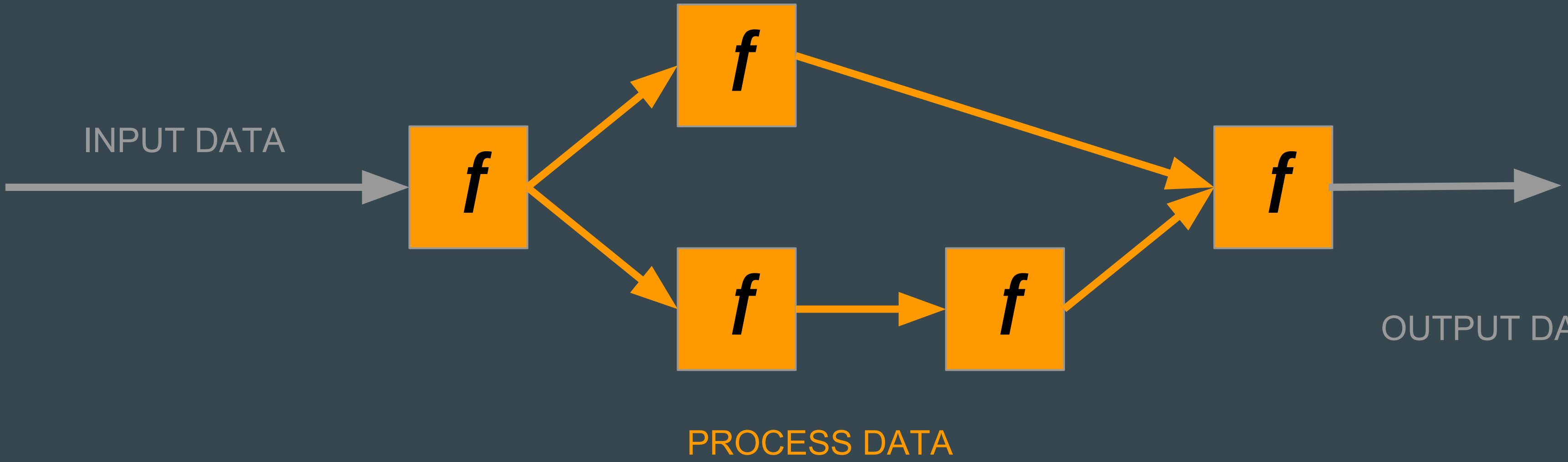


Simplicity is a key

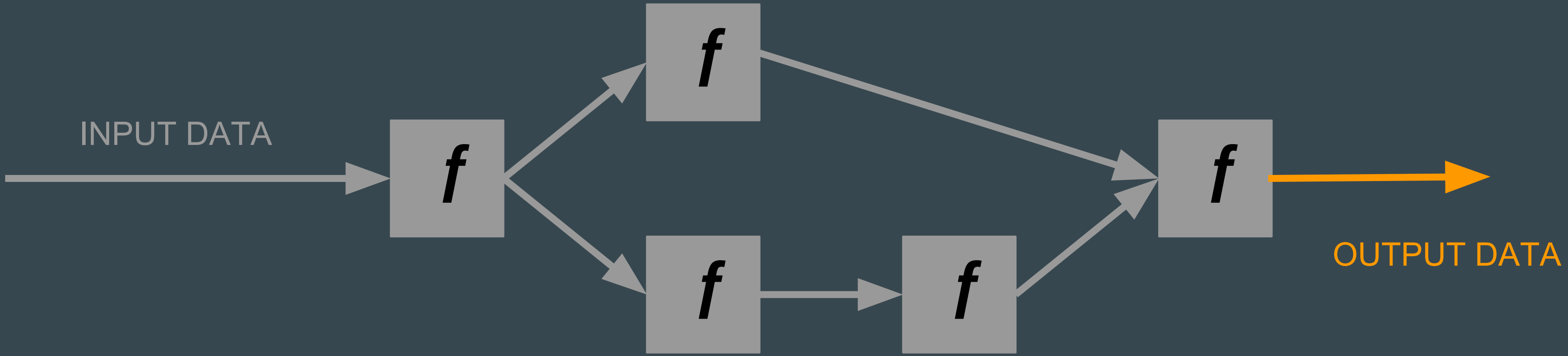
Data Flow - Expectations



Data Flow - Expectations

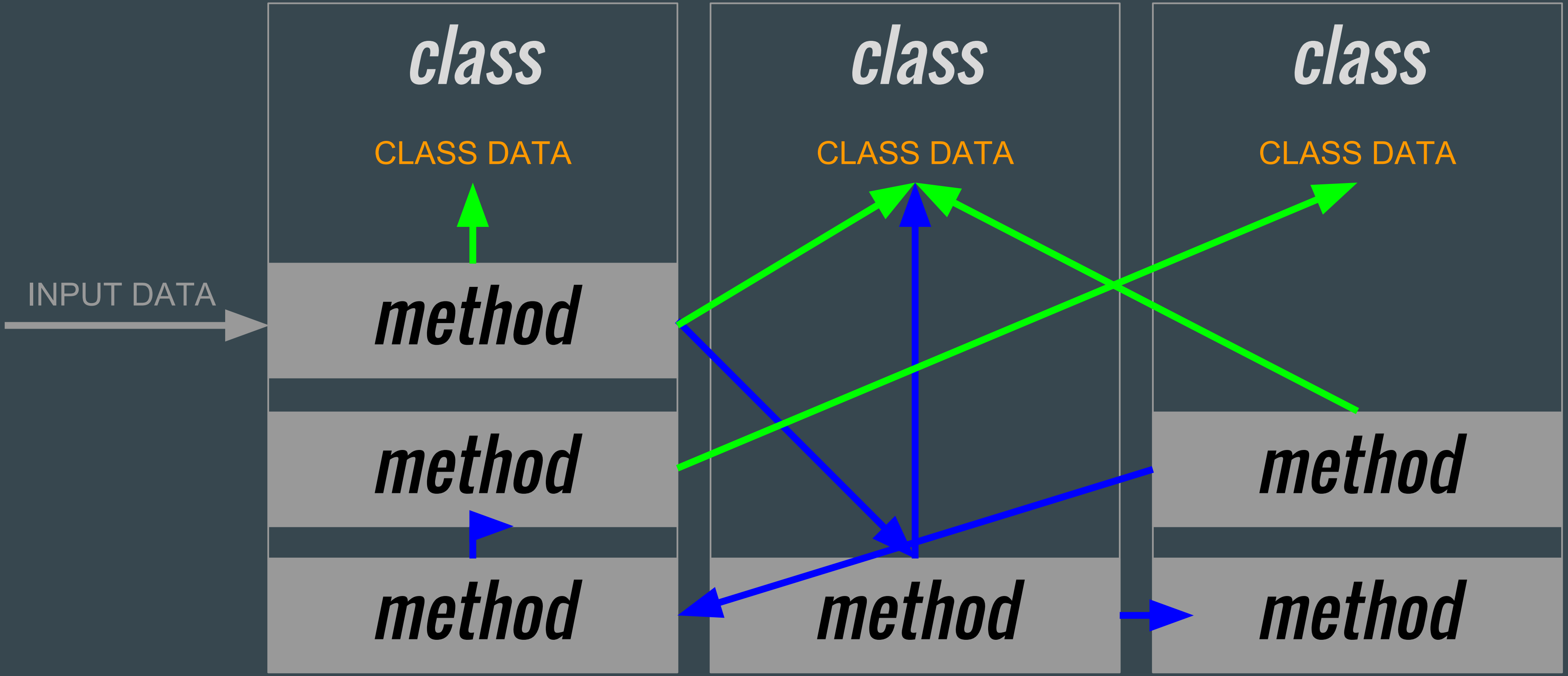


Data Flow - Expectations



Simplicity is a key, still...

Data Flow - Reality



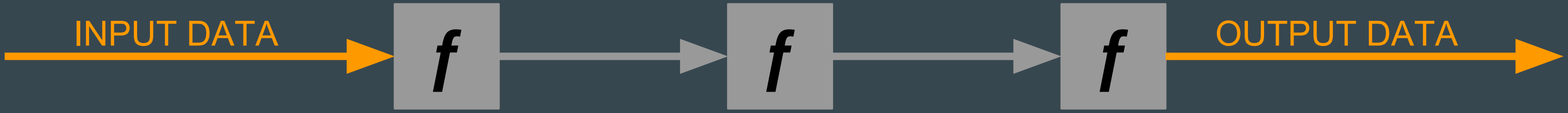
Data Flow - Reality - OOP in nutshell

Simplicity ? Errr….

What can we achieve ?

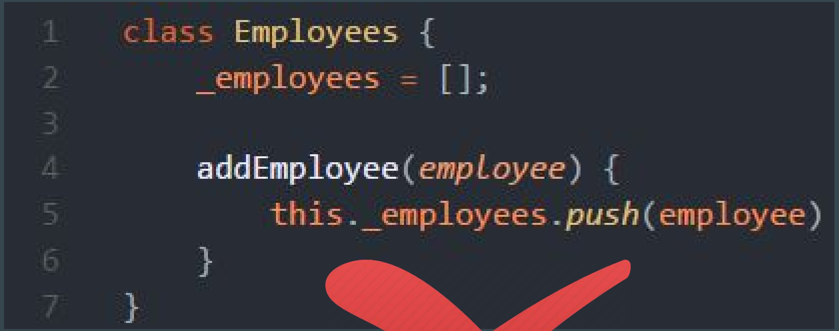
What can we achieve ?

- one directional dataflow



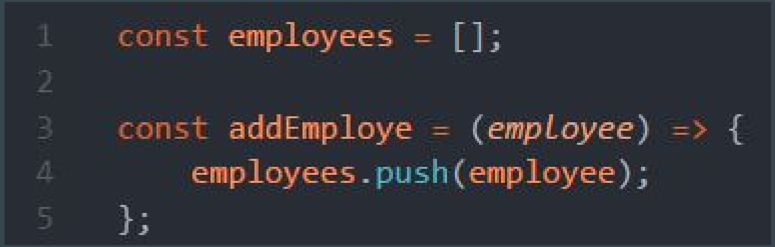
How can we achieve that ?

Separation of Data and Functions



```
class Employees {
    _employees = [];

    addEmployee(employee) {
        this._employees.push(employee)
    }
}
```

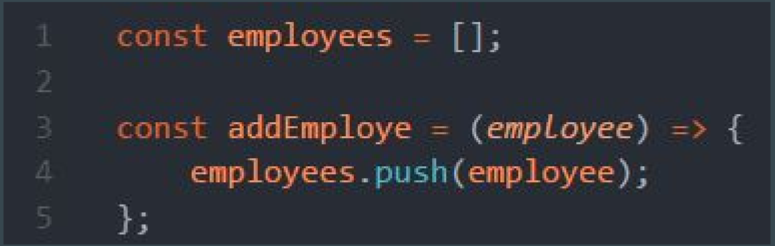
✗



```
const employees = [];

const addEmploye = (employee) => {
    employees.push(employee);
};
```

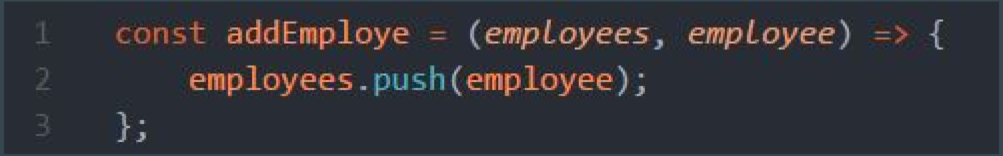
✓

No Hidden Information



```javascript
const employees = [];

const addEmploye = (employee) => {
    employees.push(employee);
};
```

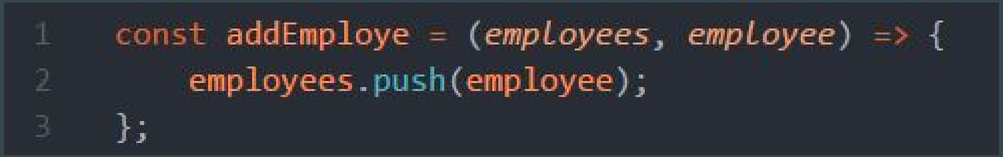
✗



```javascript
const addEmploye = (employees, employee) => {
    employees.push(employee);
};
```

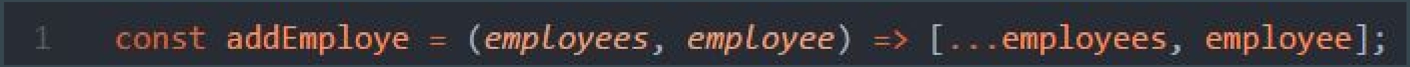
✓

No Side-effects



```
const addEmploye = (employees, employee) => {
    employees.push(employee);
};
```

✗



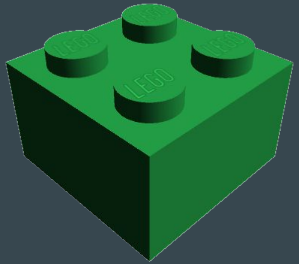
```
const addEmploye = (employees, employee) => [...employees, employee];
```

✓

Simple

is

Pure Function

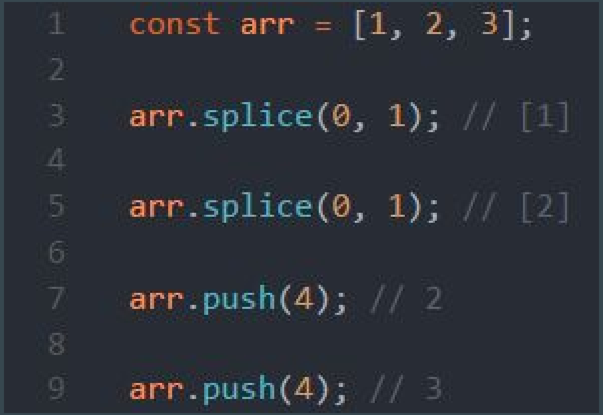


Pure Function

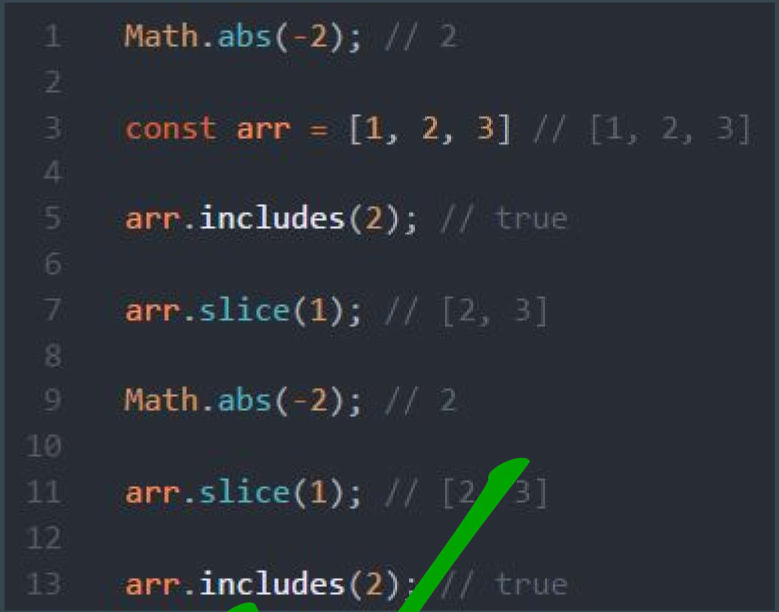
Function without time,

(Always the same output for the same arguments.)

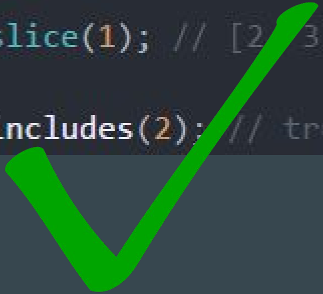
Pure Function is Timeless



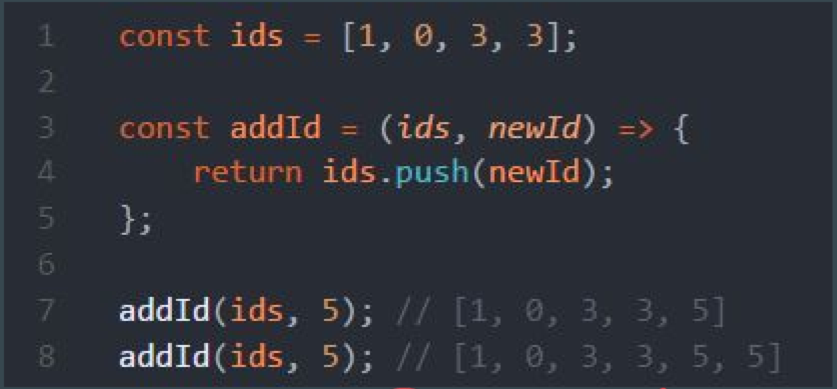
```
const arr = [1, 2, 3];

arr.splice(0, 1); // [1]

arr.splice(0, 1); // [2]

arr.push(4); // 2

arr.push(4); // 3
```



```
Math.abs(-2); // 2

const arr = [1, 2, 3] // [1, 2, 3]

arr.includes(2); // true

arr.slice(1); // [2, 3]

Math.abs(-2); // 2

arr.slice(1); // [2, 3]

arr.includes(2); // true
```

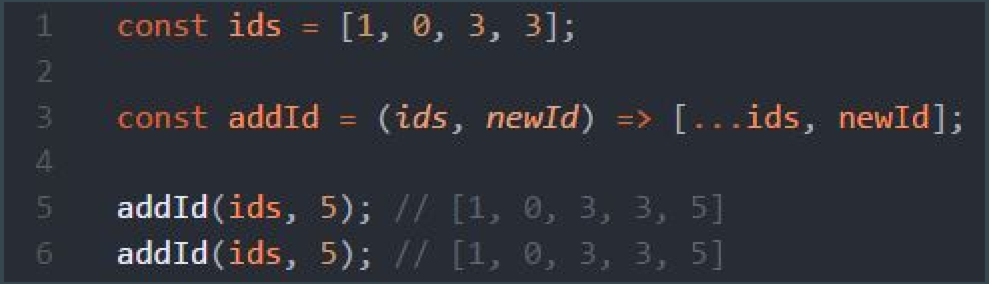


Pure Function has No side-effects



```javascript
const ids = [1, 0, 3, 3];

const addId = (ids, newId) => {
    return ids.push(newId);
};

addId(ids, 5); // [1, 0, 3, 3, 5]
addId(ids, 5); // [1, 0, 3, 3, 5, 5]
```

✗

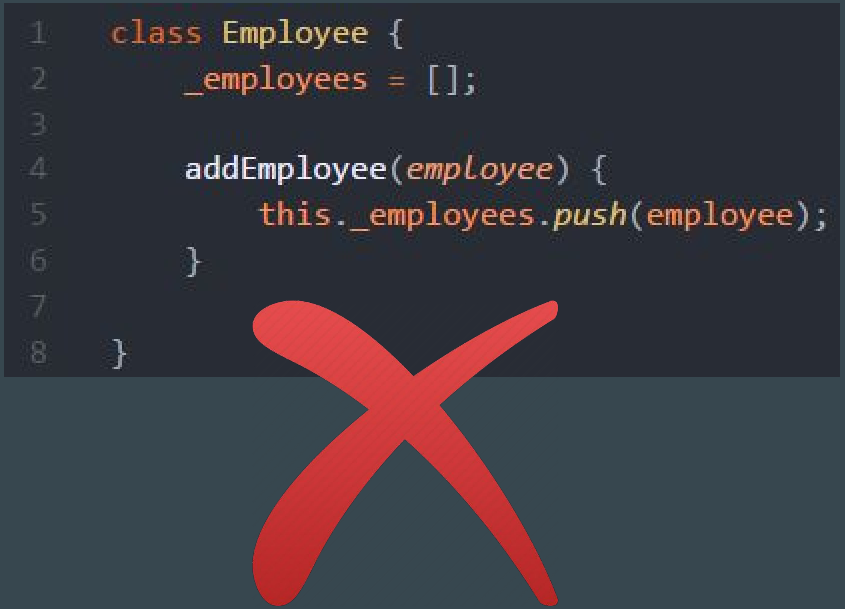


```javascript
const ids = [1, 0, 3, 3];

const addId = (ids, newId) => [...ids, newId];

addId(ids, 5); // [1, 0, 3, 3, 5]
addId(ids, 5); // [1, 0, 3, 3, 5]
```

✓

Pure Function is Opposite to Class Method

No this keyword



```
class Employee {
    _employees = [];

    addEmployee(employee) {
        this._employees.push(employee);
    }

}
```

Why should you care about pure functions ?

Pure functions advantages

- Easier to maintain
- Testability
- Easy to debug
- Simply reusable
- Thread-safe
- One-directional data-flow

Time for Demo

Immutability

Immutability

default in Functional languages

Immutability by default in JavaScript?

ECMAScript 6

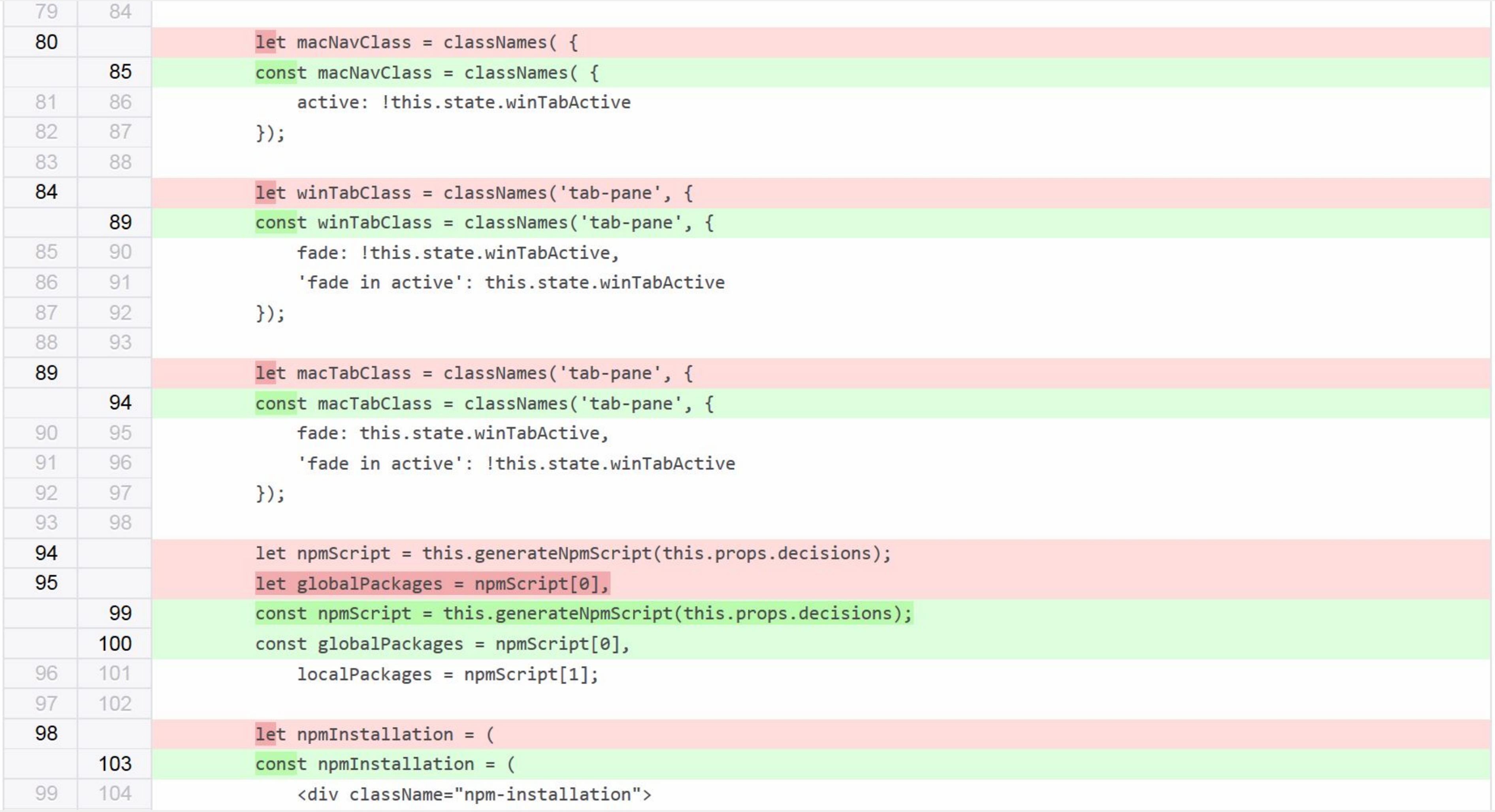
introduced let keyword

ECMAScript 6

introduced let keyword

also introduced const

Real life: Code reviews



```
        let macNavClass = classNames( {
        const macNavClass = classNames( {
            active: !this.state.winTabActive
        });

        let winTabClass = classNames('tab-pane', {
        const winTabClass = classNames('tab-pane', {
            fade: !this.state.winTabActive,
            'fade in active': this.state.winTabActive
        });

        let macTabClass = classNames('tab-pane', {
        const macTabClass = classNames('tab-pane', {
            fade: this.state.winTabActive,
            'fade in active': !this.state.winTabActive
        });

        let npmScript = this.generateNpmScript(this.props.decisions);
        let globalPackages = npmScript[0],
        const npmScript = this.generateNpmScript(this.props.decisions);
        const globalPackages = npmScript[0],
            localPackages = npmScript[1];

        let npmInstallation = (
        const npmInstallation = (
            <div className="npm-installation">
```

```
const firstName = 'Emmet';
```

```javascript
const person = {
    firstName: 'Emmet',
    lastName: 'Hutchinson'
};
```

```javascript
const person = {
    firstName: 'Emmet',
    lastName: 'Hutchinson'
};

person.firstName = 'Gandalf';
```

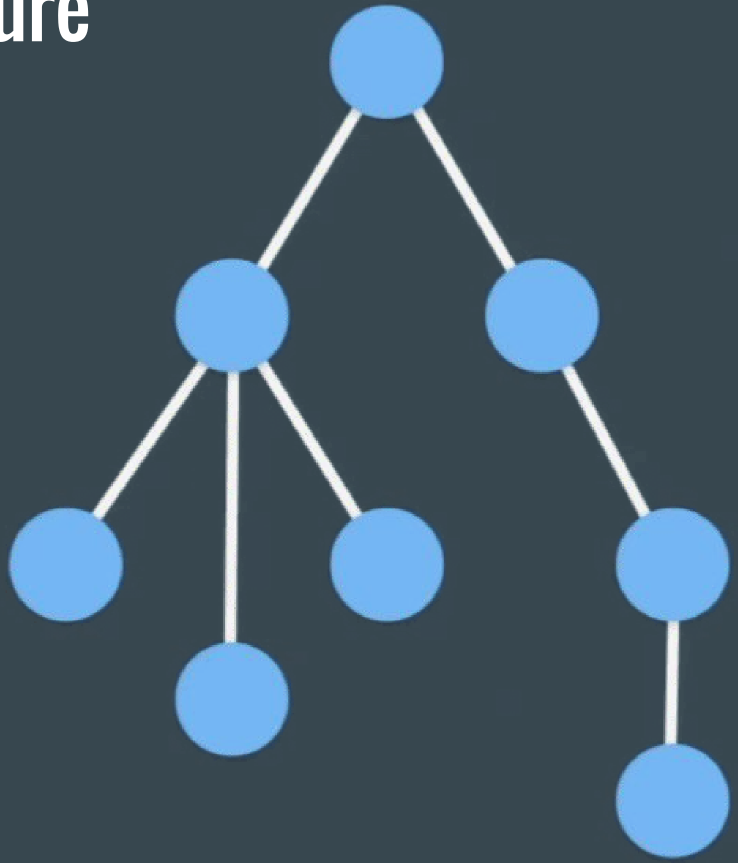
Immutable Data Structures FTW

Immutable building blocks

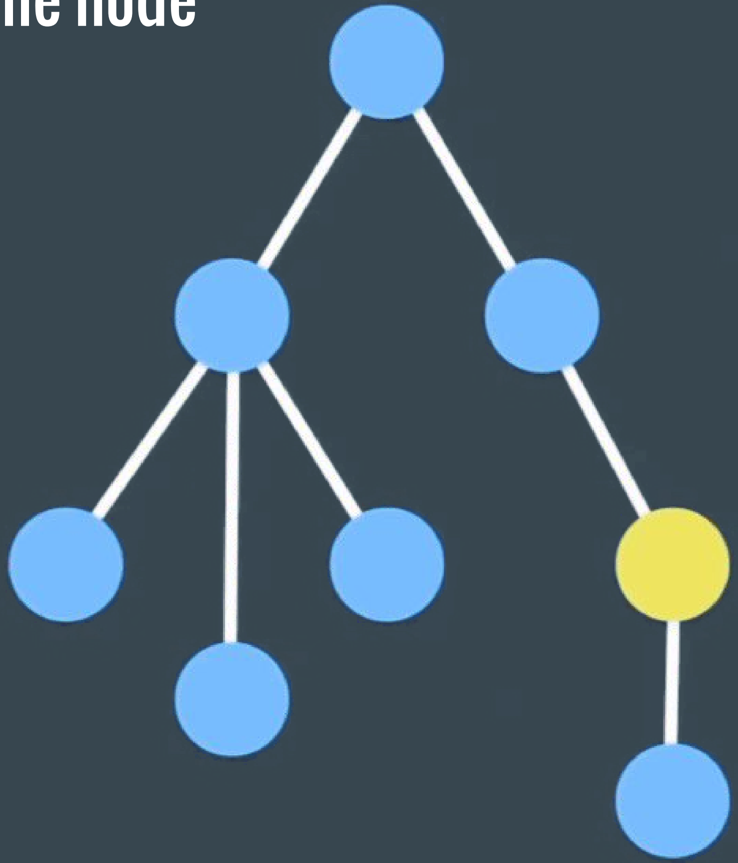
- Records
- Vectors
- Maps
- Sets

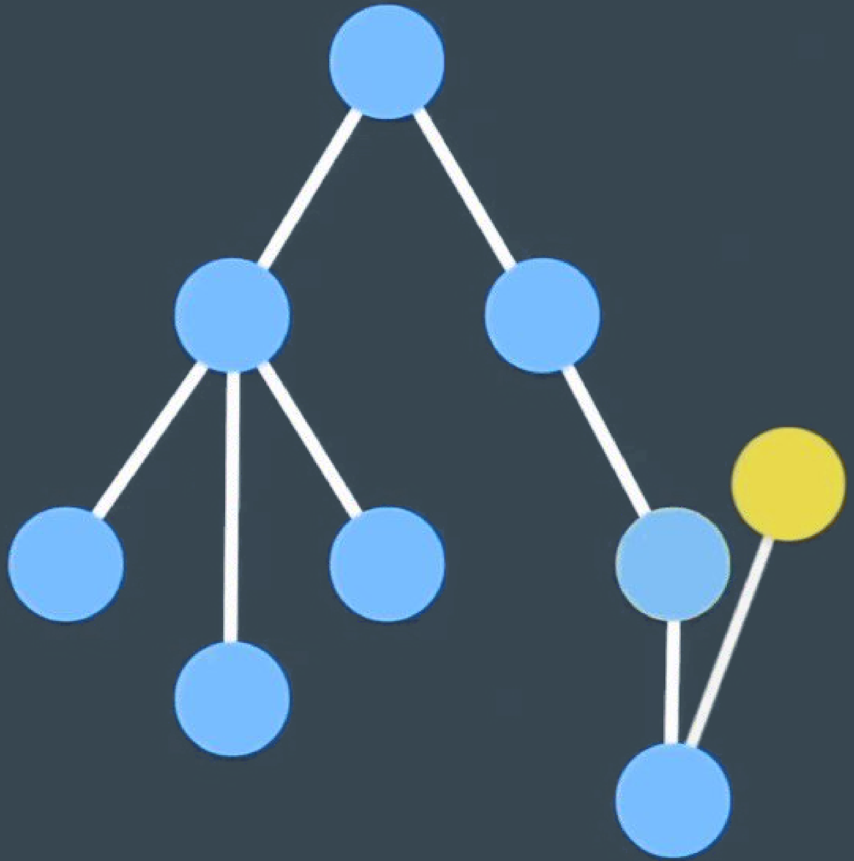
… build anything you want

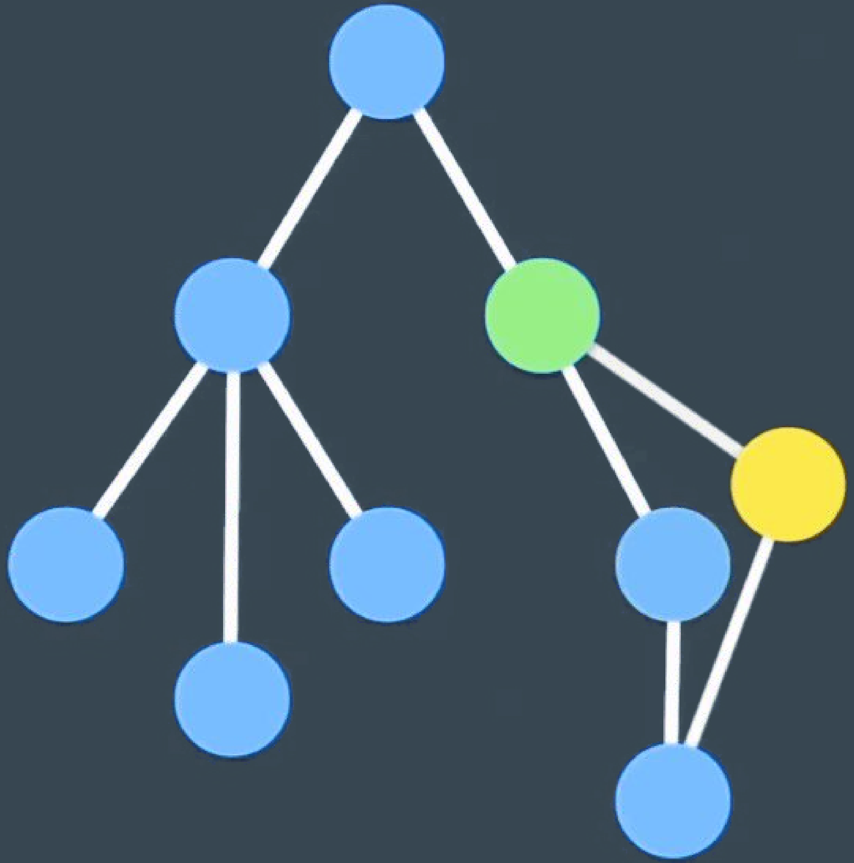
Tree data structure

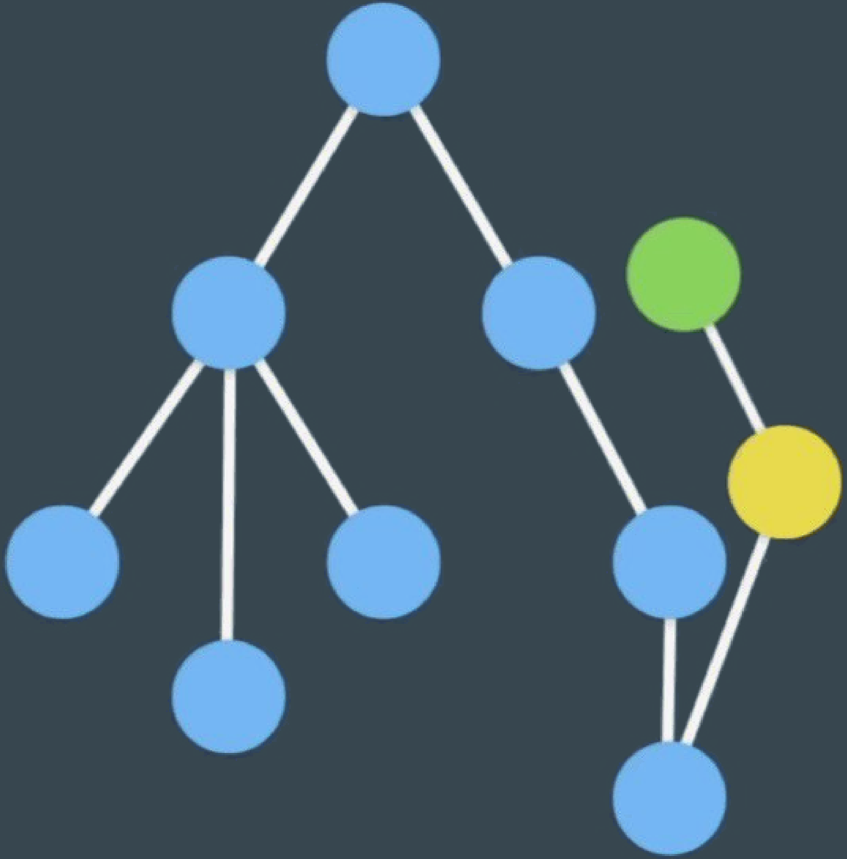


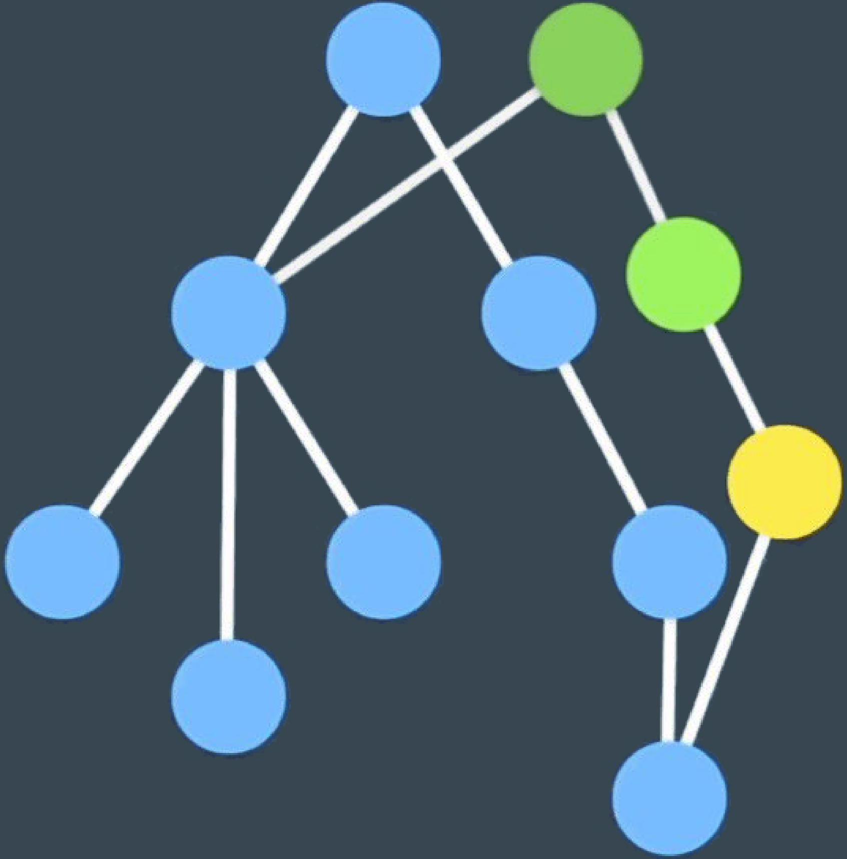
We want to change one node

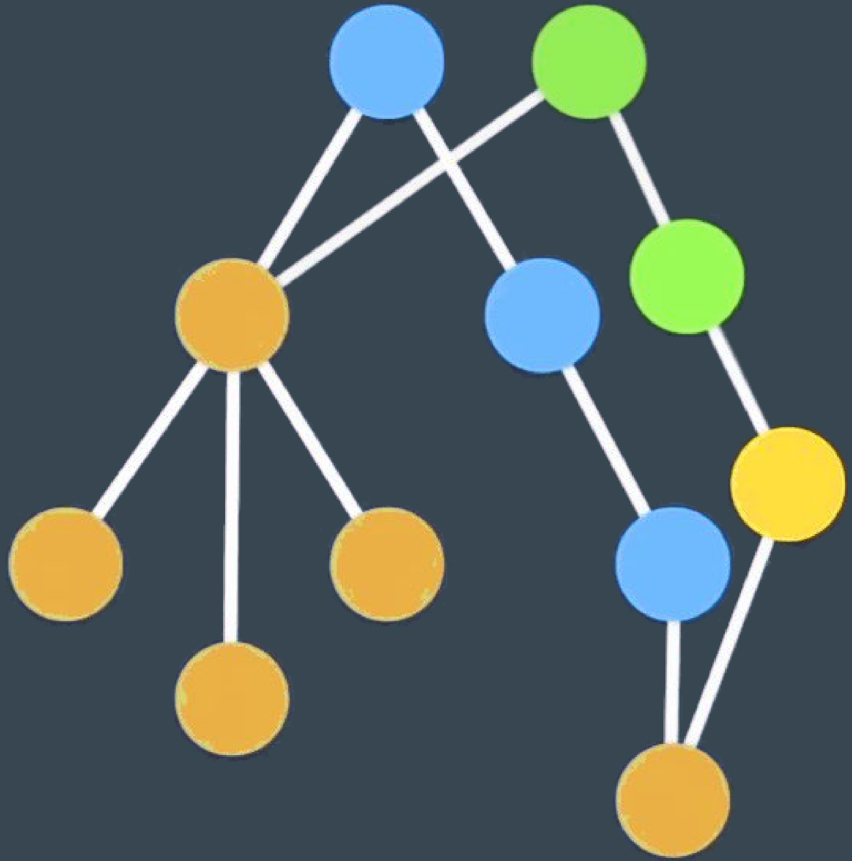


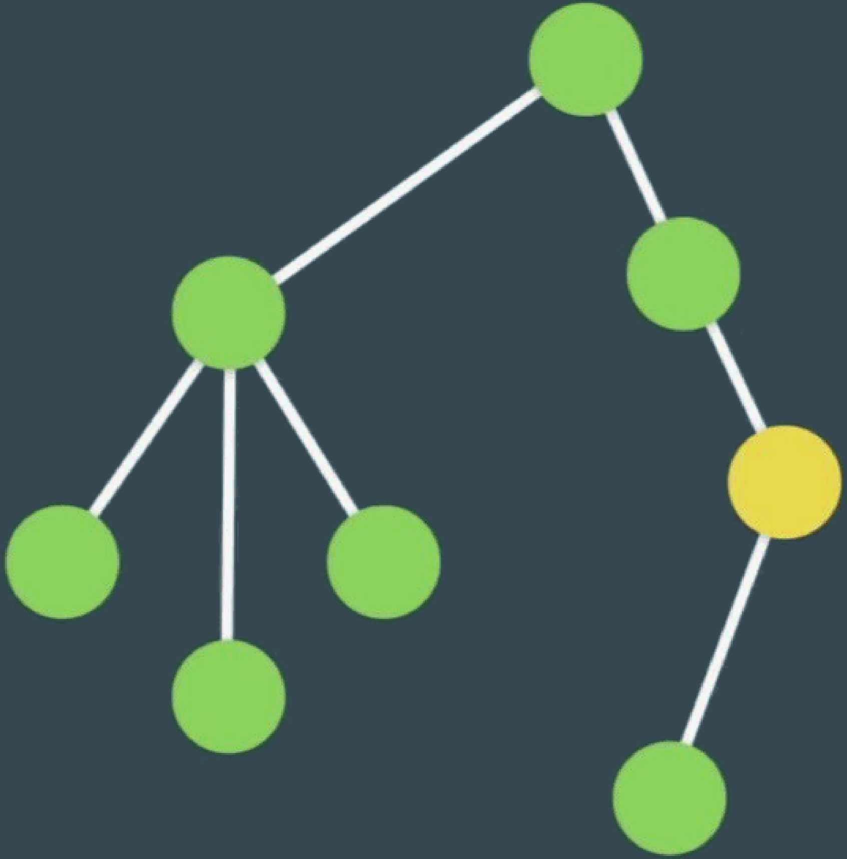




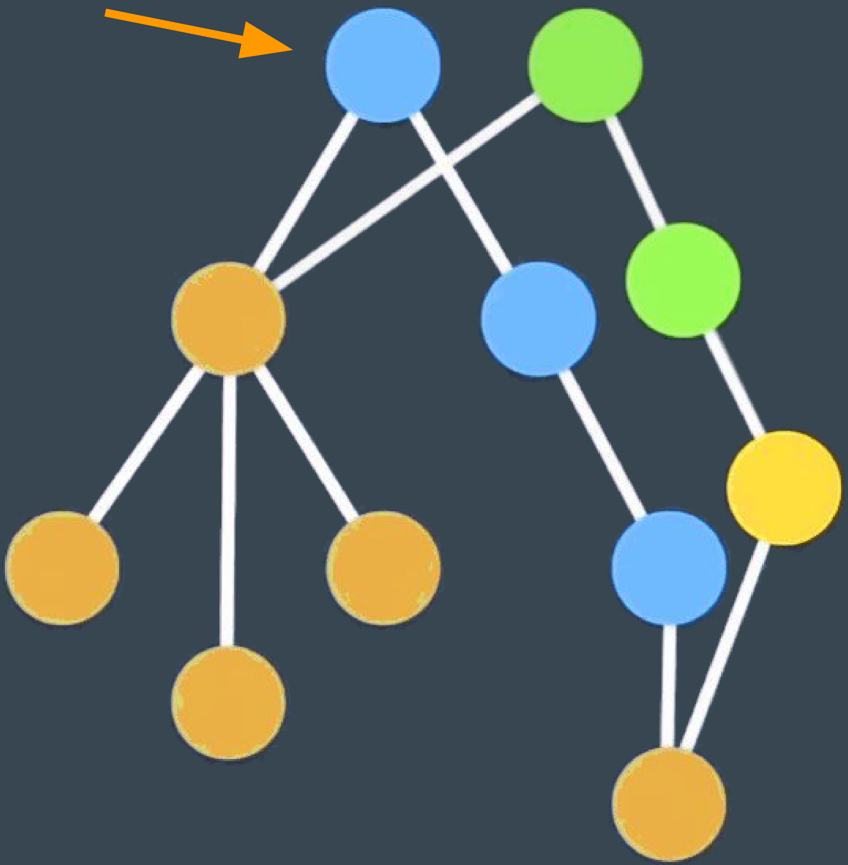








...or keep it for the history record



Immutability in Java

Practices for immutability in Java (make them habits)

- Mark the class final
- Mark all the fields private and final
- Force all the callers to construct an object of the class directly, i.e. do not use any setter methods
- Do not change the state of the objects in any methods of the class

“Classes should be immutable unless there's a very good reason to make them mutable.... If a class cannot be made immutable, limit its mutability as much as possible.”

-- Joshua Bloch (taken from the book Effective Java)

Benefits of immutable objects

- Thread-safety
- Easier to parallelize
- Consistent internal state (in spite of exception)
- References to immutable objects can be cached
    - Easy to implement UNDO - REDO functionality

Time for a success story

Redux.js

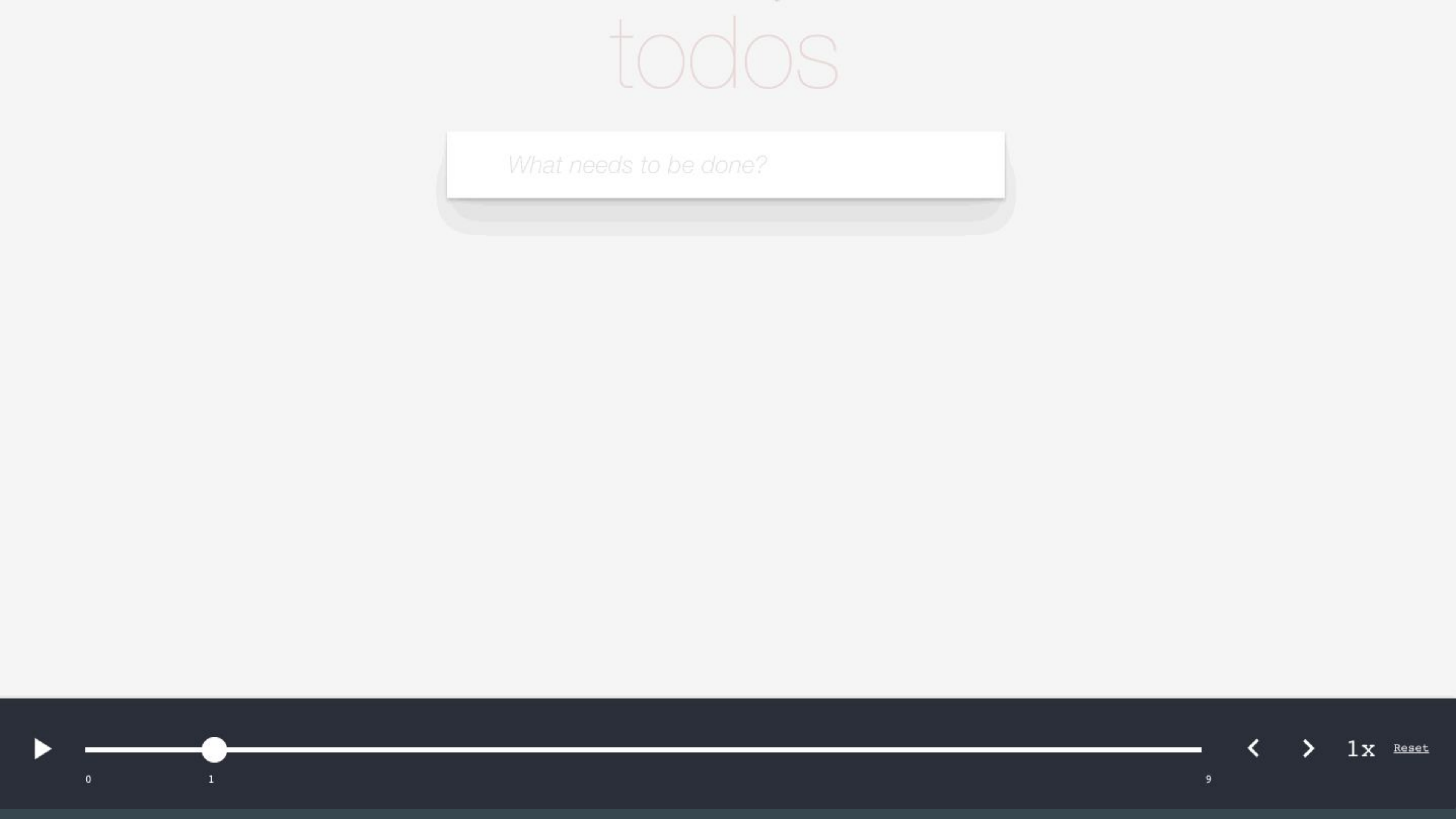
- FLUX-ish library by Dan Abramov
  - Creator of Hot Module Reloading for ReactJS
- Motivation:
  - We want Hot Module Reloading not only for View Components
  - but also for Business logic inside FLUX stores (like Models in MVC)

Redux

- Solution
    - Take state out of Stores
    - Make Store methods immutable / pure functions
    - No classes, just functions - reducers

Redux

- Result
  - HMR also for reducers
  - Application keeps state after HMR !!!
  - Time travelling
  - Awesome debugging tools
  - Community excitement - this is how FLUX was supposed to work from the start


todos
What needs to be done?
0
1
9
1x
Reset

Summary

What can we improve in our daily programming ?

Shared mutable state is the root of all evil.

Pete Hunt, ReactJS team @ Facebook

Avoid mutability

Don’t use global scoped variables

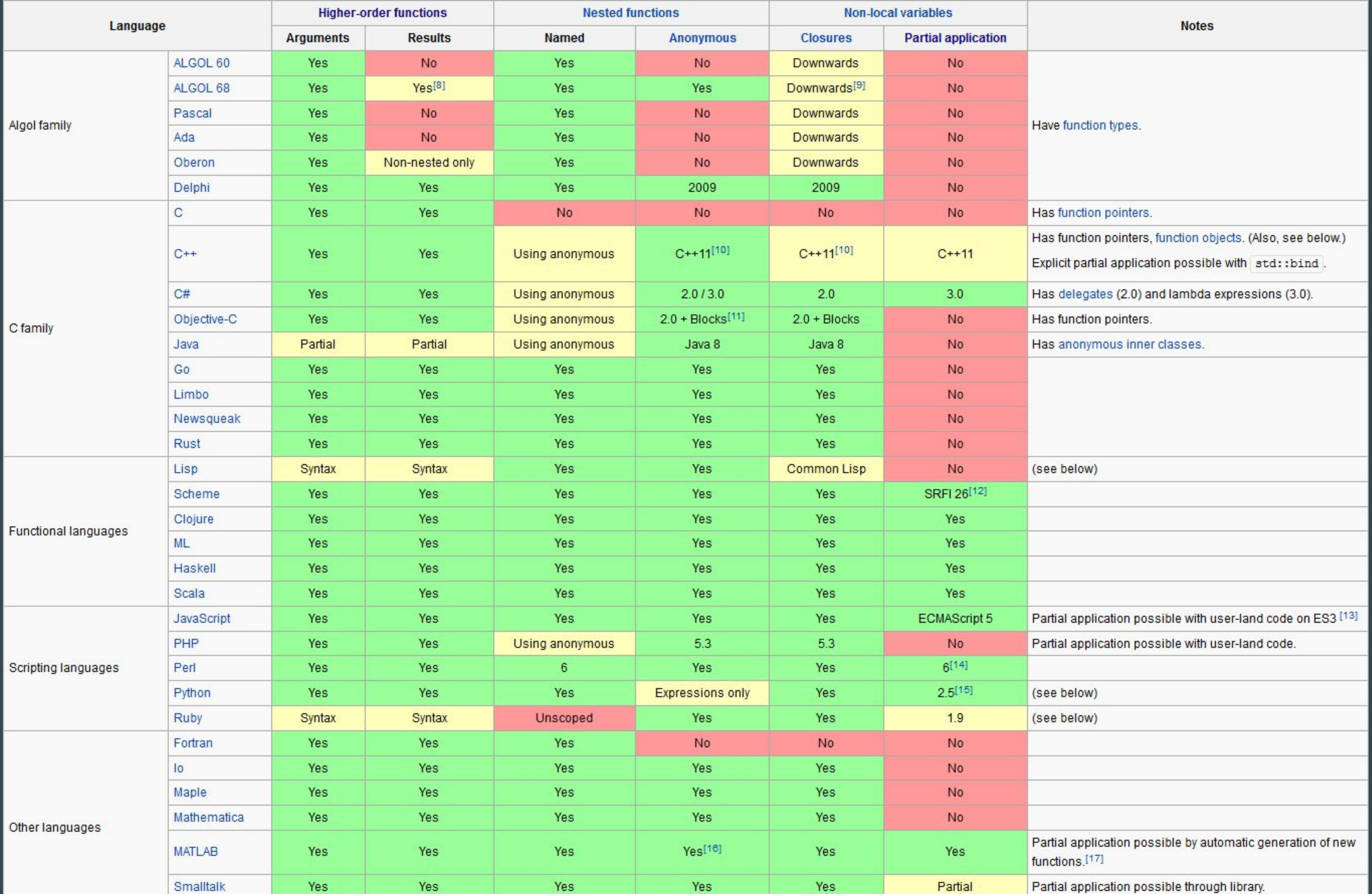
Make const not var

Separate Data from Business logic

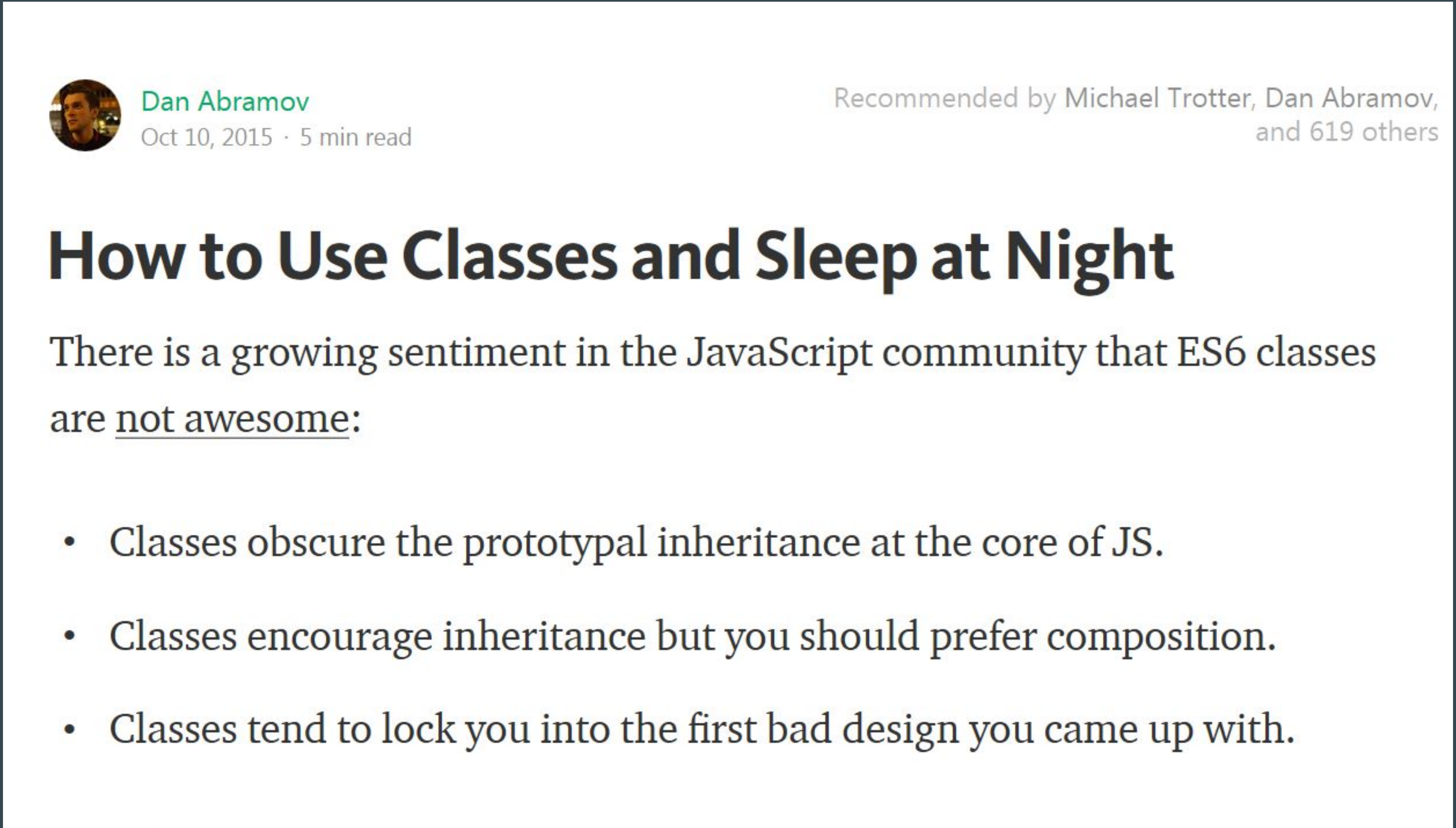
Use Pure functions

Learning Haskell will help you write better Java(Script) code

OOP vs FP ?



Language		Higher-order functions		Nested functions		Non-local variables		Notes
		Arguments	Results	Named	Anonymous	Closures	Partial application	
Algol family	ALGOL 60	Yes	No	Yes	No	Downwards	No	Have function types.
	ALGOL 68	Yes	Yes[8]	Yes	Yes	Downwards[9]	No	
	Pascal	Yes	No	Yes	No	Downwards	No	
	Ada	Yes	No	Yes	No	Downwards	No	
	Oberon	Yes	Non-nested only	Yes	No	Downwards	No	
	Delphi	Yes	Yes	Yes	2009	2009	No	
C family	C	Yes	Yes	No	No	No	No	Has function pointers.
	C++	Yes	Yes	Using anonymous	C++11[10]	C++11[10]	C++11	Has function pointers, function objects. (Also, see below.) Explicit partial application possible with `std::bind`.
	C#	Yes	Yes	Using anonymous	2.0 / 3.0	2.0	3.0	Has delegates (2.0) and lambda expressions (3.0).
	Objective-C	Yes	Yes	Using anonymous	2.0 + Blocks[11]	2.0 + Blocks	No	Has function pointers.
	Java	Partial	Partial	Using anonymous	Java 8	Java 8	No	Has anonymous inner classes.
	Go	Yes	Yes	Yes	Yes	Yes	No	
	Limbo	Yes	Yes	Yes	Yes	Yes	No	
	Newsqueak	Yes	Yes	Yes	Yes	Yes	No	
	Rust	Yes	Yes	Yes	Yes	Yes	No	
Functional languages	Lisp	Syntax	Syntax	Yes	Yes	Common Lisp	No	(see below)
	Scheme	Yes	Yes	Yes	Yes	Yes	SRFI 26[12]	
	Clojure	Yes	Yes	Yes	Yes	Yes	Yes	
	ML	Yes	Yes	Yes	Yes	Yes	Yes	
	Haskell	Yes	Yes	Yes	Yes	Yes	Yes	
	Scala	Yes	Yes	Yes	Yes	Yes	Yes	
Scripting languages	JavaScript	Yes	Yes	Yes	Yes	Yes	ECMAScript 5	Partial application possible with user-land code on ES3 [13]
	PHP	Yes	Yes	Using anonymous	5.3	5.3	No	Partial application possible with user-land code.
	Perl	Yes	Yes	6	Yes	Yes	6[14]	
	Python	Yes	Yes	Yes	Expressions only	Yes	2.5[15]	(see below)
	Ruby	Syntax	Syntax	Unscoped	Yes	Yes	1.9	(see below)
Other languages	Fortran	Yes	Yes	Yes	No	No	No	
	Io	Yes	Yes	Yes	Yes	Yes	No	
	Maple	Yes	Yes	Yes	Yes	Yes	No	
	Mathematica	Yes	Yes	Yes	Yes	Yes	No	
	MATLAB	Yes	Yes	Yes	Yes[16]	Yes	Yes	Partial application possible by automatic generation of new functions.[17]
	Smalltalk	Yes	Yes	Yes	Yes	Yes	Partial	Partial application possible through library.



Dan Abramov
Oct 10, 2015 · 5 min read

Recommended by Michael Trotter, Dan Abramov, and 619 others

How to Use Classes and Sleep at Night

There is a growing sentiment in the JavaScript community that ES6 classes are not awesome:

- Classes obscure the prototypal inheritance at the core of JS.
- Classes encourage inheritance but you should prefer composition.
- Classes tend to lock you into the first bad design you came up with.

Composition over Inheritance

Topics for future sessions

Functional programming

- Higher order functions
- Partial application
- Currying
- Closures
- Functors
- ...
- Monads?

Topics for future sessions

Functional Reactive Programming

- ReactiveX (RxJS, RxJava), …
  - underscore / lodash for streams of events / data
- Observables
- Reactive (user) interfaces
- …

Topics for future sessions

Clojure

Haskell

Ocaml

Elm

ClojureScript

PureScript

ReasonML

Happy building