Charlie Koster | @ckoster22

FUNCTIONAL PROGRAMMING FOR THE DIS-FUNCTIONAL

TITANIUM SPONSORS

epiq

PAIGE
TECHNOLOGIES
INTELLIGENT PAIRING. PERPETUAL SUCCESS.

ADAPTIVE™
SOLUTIONS GROUP

okta

Platinum Sponsors

VinSolutions
Make every connection count.

jack henry
& ASSOCIATES INC.®

VALOREM

NAIC
National Association of Insurance Commissioners

NIPR
NATIONAL INSURANCE
PRODUCER REGISTRY

TEKsystems
Our people make IT possible.

KEYHOLE
SOFTWARE

smartsheet
Work Better™

Couchbase

EquipmentShare

KOCH™

PLURALSIGHT

Google Cloud

VML

mylo™
POWERED
BY LOCKTON

Gold Sponsors

Advantage Tech
IT Staffing & Recruiting Services

PeopleAdmin

CENTRIQ
TRAINING

Cerner

KU
EDWARDS
CAMPUS
The University of Kansas

twilio

{Track:js}
JAVASCRIPT ERROR MONITORING

OBJECT
PARTNERS

smg
service
management
group®

stackify

GARMIN

Balance
Innovations

Allied Global
SERVICES

dsi

LRS
CONSULTING SERVICES

Aviture
ingenuity. engineered.

TouchNet
+ Heartland

Mediware®

An affiliate of CSA Industries, Inc.
AUREUS
GROUP
Finance | Systems | Executive

Children's Mercy
KANSAS CITY

Commerce Bank

SECURITY PS
STRATEGIC INFORMATION SECURITY

ProKarma

VS**

- Object Oriented Programming
- Imperative Programming
- Stateful Programming

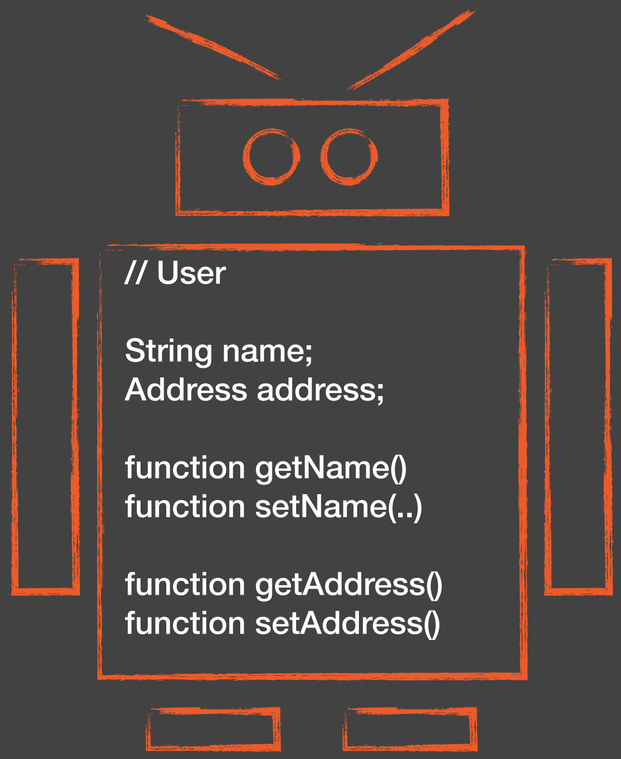
- Functional Programming
- Declarative Programming
- Stateless Programming

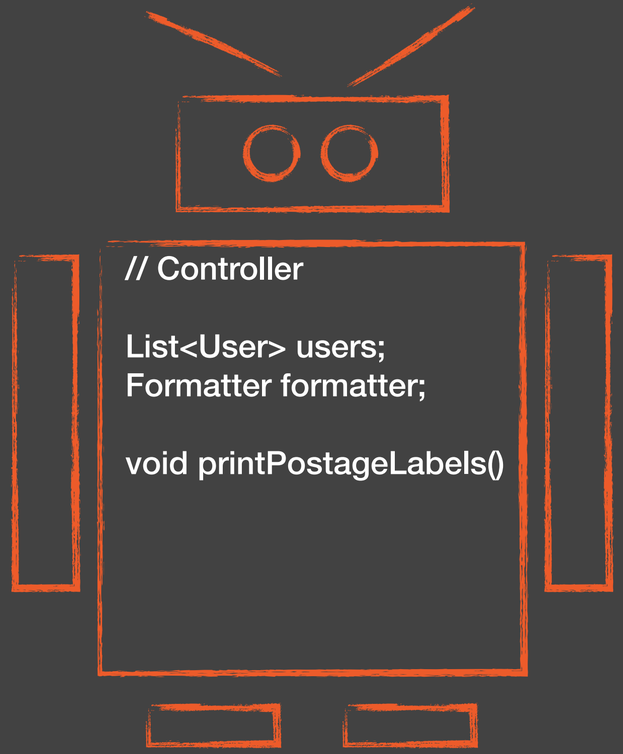
* Not necessarily mutually incompatible paradigms
** Mutually exclusive approaches to problem solving

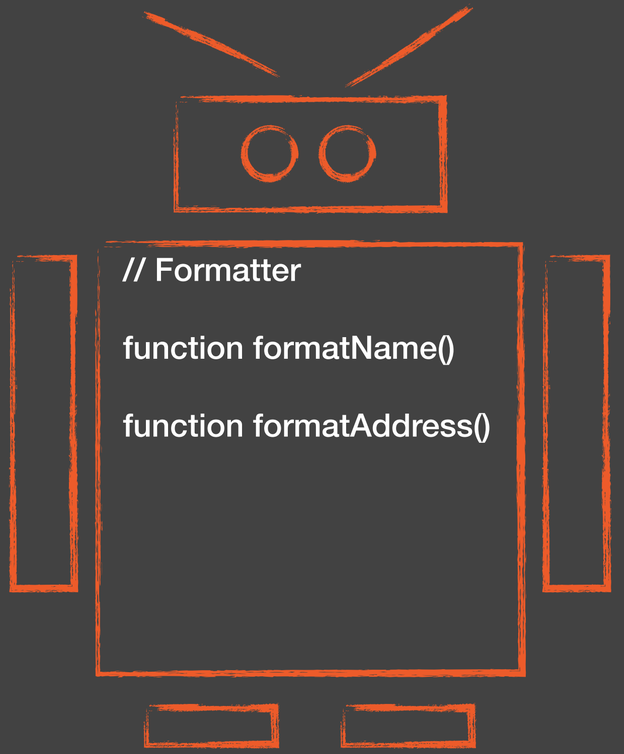
```
// User

String name;
Address address;

function getName()
function setName(..)

function getAddress()
function setAddress()
```

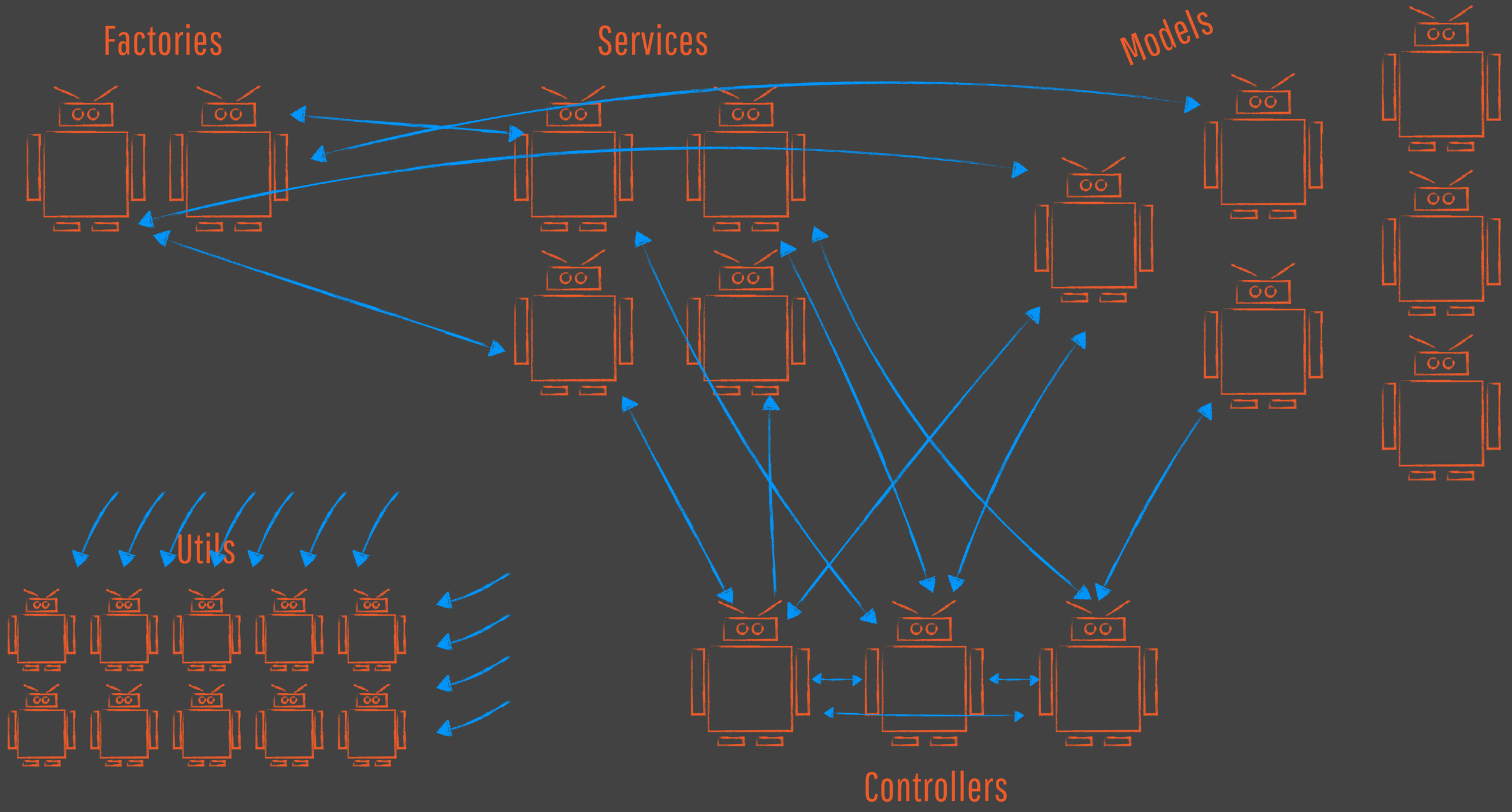
```
// Formatter

function formatName()

function formatAddress()
```

```
// Controller

List<User> users;
Formatter formatter;

void printPostageLabels()
```


// User
String name;
Address address;
function getName()
function setName(..)
function getAddress()
function setAddress()


// Controller
List<User> users;
Formatter formatter;
void printPostageLabels()


// Formatter
function formatName()
function formatAddress()

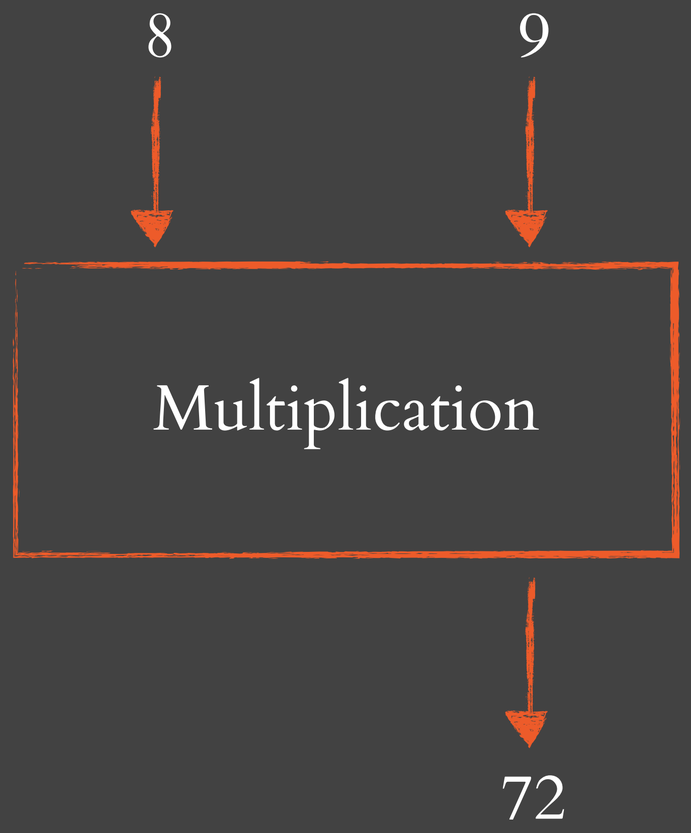

Factories
Services
Models
Utils
Controllers

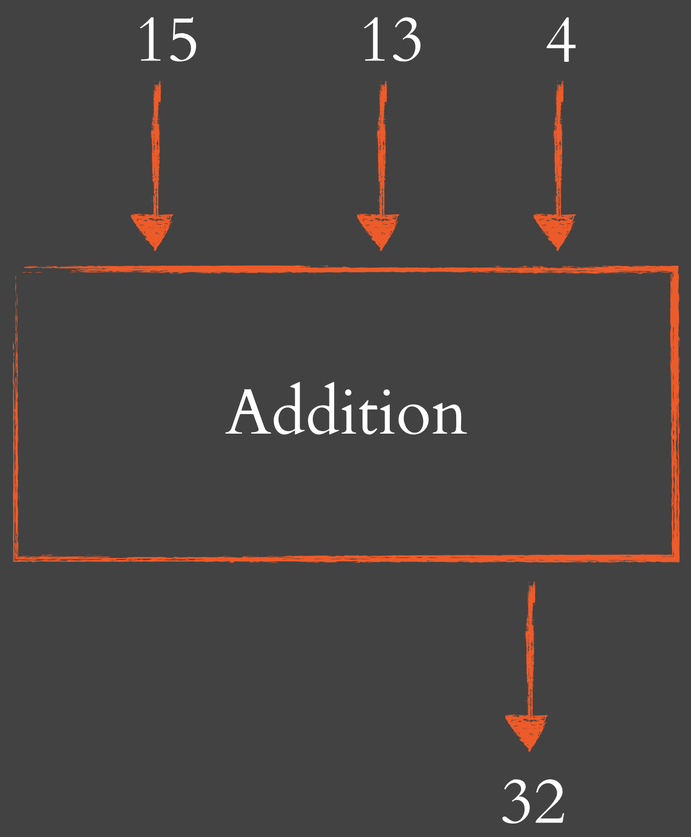
Stateful Programming complects data, behavior, and time

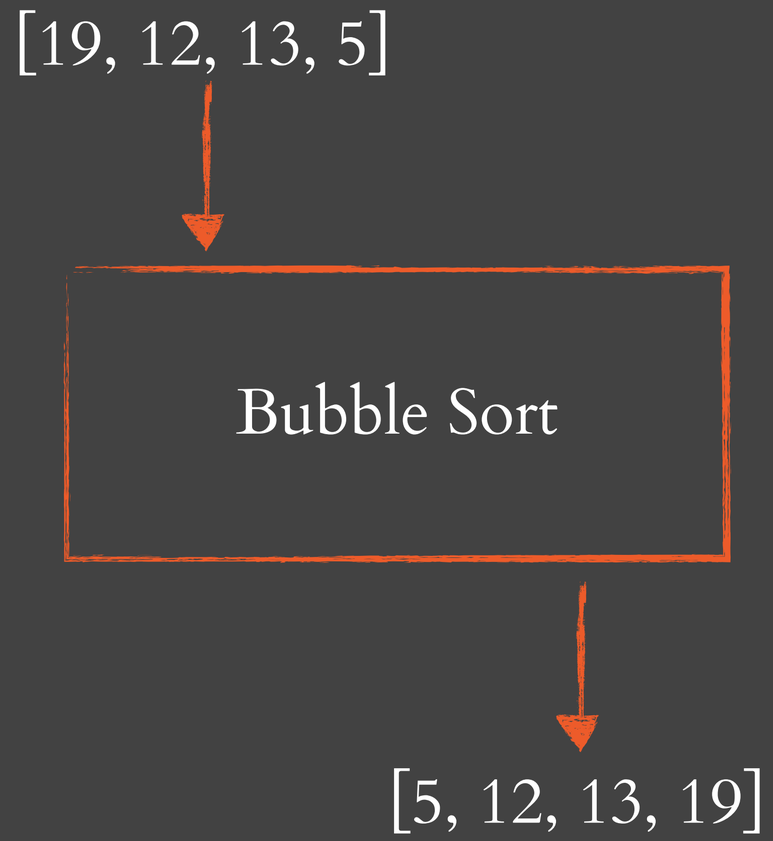
How

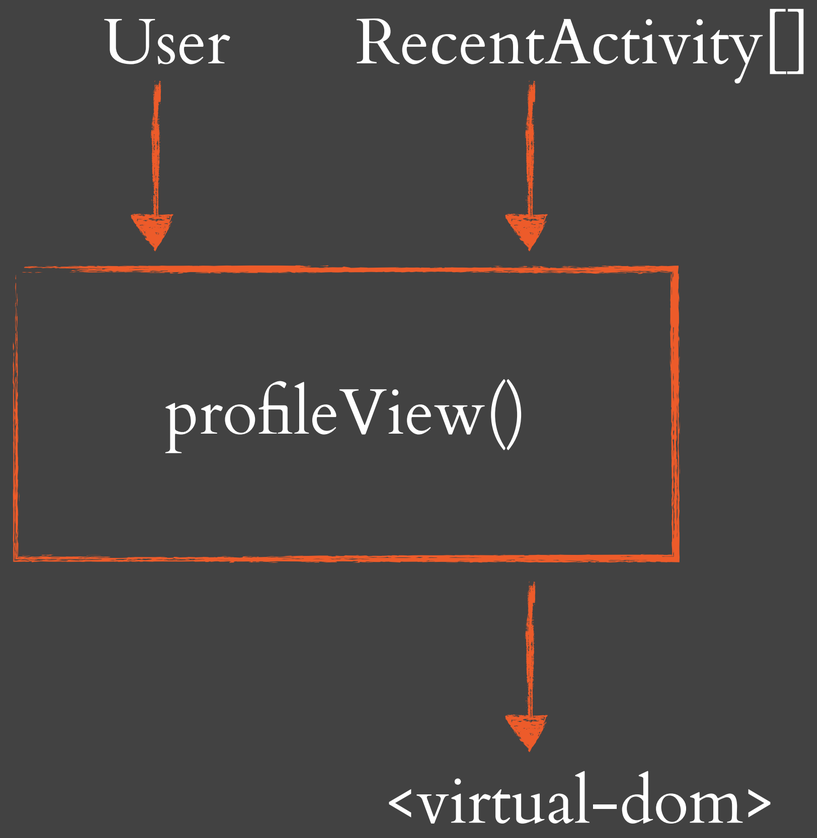
When

What


8
9
Multiplication
72

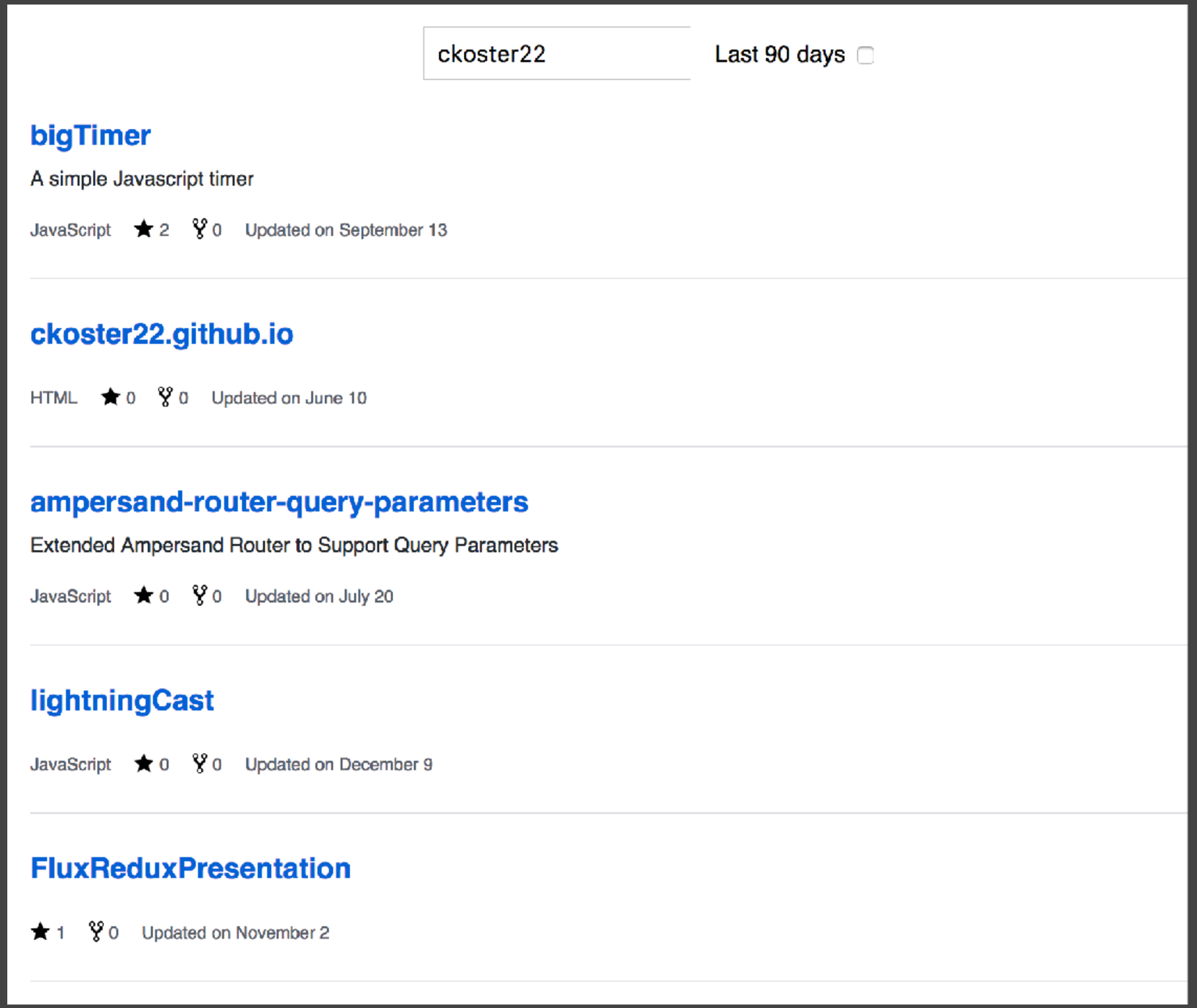

15
13
4
Addition
32


[19, 12, 13, 5]
Bubble Sort
[5, 12, 13, 19]


User
RecentActivity[]
profileView()
<virtual-dom>

Functional Programming separates the what,
the how, and the when

WHAT IS FUNCTIONAL PROGRAMMING?


ckoster22
Last 90 days
bigTimer
A simple Javascript timer
JavaScript 2 0 Updated on September 13
ckoster22.github.io
HTML 0 0 Updated on June 10
ampersand-router-query-parameters
Extended Ampersand Router to Support Query Parameters
JavaScript 0 0 Updated on July 20
lightningCast
JavaScript 0 0 Updated on December 9
FluxReduxPresentation
1 0 Updated on November 2

FUNCTIONAL PROGRAMMING

- First class functions
- Higher order functions
- Map, filter, and reduce
- Pure functions
- Immutability

First class functions

A first class function is a function that can be treated as a value

FIRST CLASS FUNCTION EXAMPLE

```
const repoFilter = (repo) => {
    return repo.lastPushed.getTime() >= ninetyDaysAgo;
};

…

repositories.filter(repoFilter);
```

Higher order functions

Higher order functions can receive functions as arguments or return functions

```
function isDivisibleBy(divisor) {
    return function(num) {
        return num % divisor === 0;
    }
}

const isEven = isDivisibleBy(2);

isEven(10); // true
isEven(11); // false
```

```
function isDivisibleBy(divisor) {
    return function(num) {
        return num % divisor === 0;
    }
}

const isEven = isDivisibleBy(2);

isEven(10); // true
isEven(11); // false
```

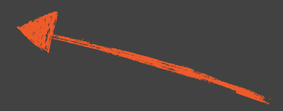
```
function isDivisibleBy(divisor) {
    return function(num) {
        return num % divisor === 0;
    }
}

const isEven = isDivisibleBy(2);

isEven(10); // true
isEven(11); // false
```

```
function isDivisibleBy(divisor) {
    return function(num) {
        return num % divisor === 0;
    }
}

const isEven = isDivisibleBy(2);

isEven(10); // true
isEven(11); // false
```

Map, filter, and reduce

TRANSFORMING ARRAYS WITH MAP

```
const users = [
    {
        first: 'John',
        last: 'Doe'
    },
    {
        first: 'Jane',
        last: 'Doe'
    }
];

const userFullNames = users.map(user => {
    return user.first + ' ' + user.last
});
// [ 'John Doe', 'Jane Doe' ]
```



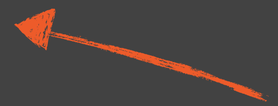
Return transformed item

```
const jsonToRepository = (json) => {
    const repo = json.node;
    let stars = repo.stargazers.edges.length;
    let forks = repo.forks.edges.length;

    return {
        owner: owner,
        name: repo.name,
        description: repo.description,
        language: repo.primaryLanguage.name,
        stars: stars,
        forks: forks,
        lastPushed: new Date(repo.pushedAt)
    };
};

…

jsonRepos.map(repoJsonToRepository);
```

Mapped to a more useful format

FILTERING ARRAYS

```
const nums = [1, 2, 3, 4, 5, 6, 7, 8];

const evens = nums.filter(num => {
    return num % 2 === 0;
});
// [2, 4, 6, 8]
```



Return boolean

```
const ninetyDaysAgo = currentMillis - (1000 * 60 * 60 * 24 * 90);

const repoFilter = (repo) => {
    return repo.lastPushed.getTime() >= ninetyDaysAgo;
};

…

repositories.filter(repoFilter);
```

REDUCE - THE ARRAY SWISS ARMY KNIFE

```
const nums = [1, 2, 3, 4, 5, 6, 7, 8];

const sum = nums.reduce((total, num) => {
    return total + num;
}, 0);
// 36
```

Initial value

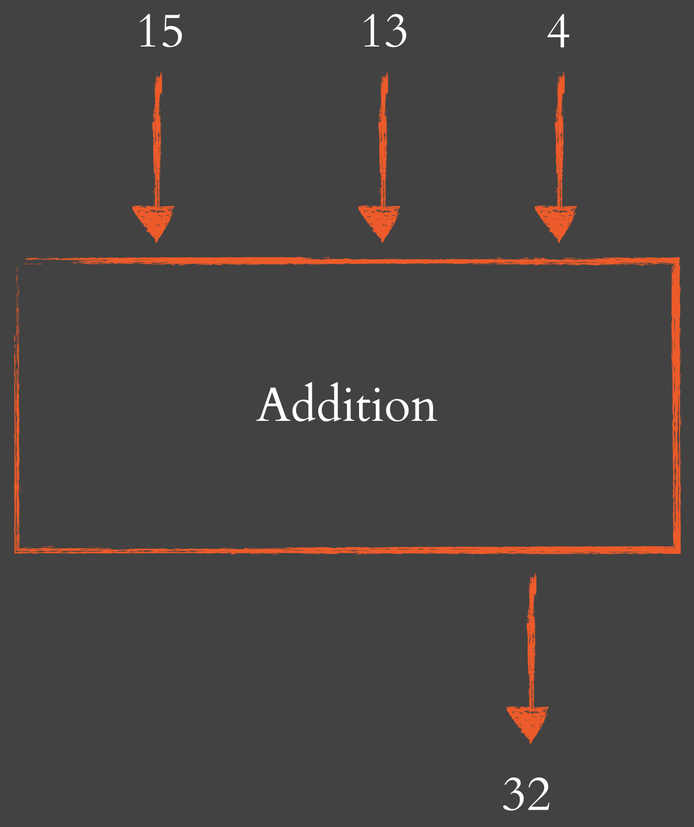
Accumulator

Total
0
1
3
6
10
15
21
28
36

Pure functions

PURE FUNCTIONS

- Pure functions have **no side effects**
  - No mutating variables outside of the function's scope
  - No mutating the function arguments
  - No invoking impure functions (DOM manip, Http, DB, etc)
- Pure functions accept 0 or more args and return a value



NOT A PURE FUNCTION

```
let total = 0;

function calculateSum(numArr) {
    numArr.forEach((num) => {
        total += num;
    });

    return total;
}
```

Mutation outside of inner scope

NOT A PURE FUNCTION

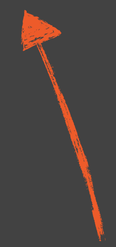
```
let myArray = [{val: 1}, {val: 2}];

function doubleNums(arr) {
    arr.forEach((item) => {
        item.val = item.val * 2;
    });
}
```

Mutation to original reference

No return statement

NOT A PURE FUNCTION

```
function getProductById(id) {
    return http.get(`http://localhost/products`, {
        id: id
    });
}
```



Side effect

PURE FUNCTIONS

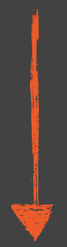
```
const repoView = (repo) => {
    return (
        <div key={repo.name} >
            {repoNameView(repo.owner, repo.name)}
            {descriptionView(repo.description)}

            …

        </div>
    );
};
```

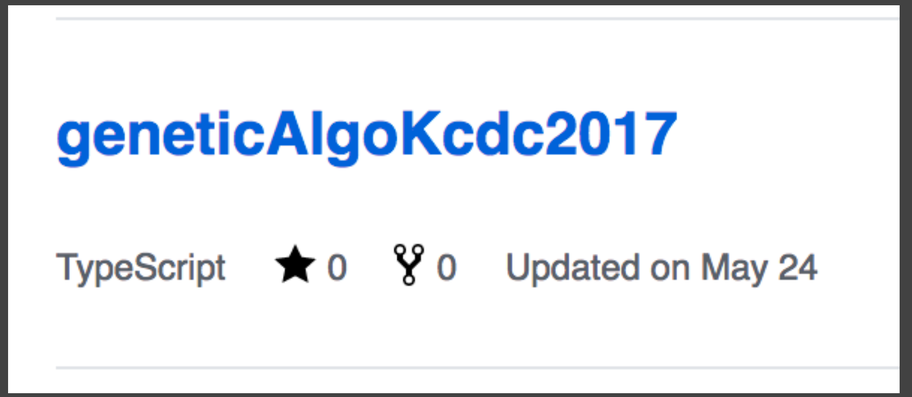
PURE FUNCTIONS

```
const jsonToRepoViews = (json) => {
    return json.map(jsonToRepository)   // convert to repo
               .filter(repoFilter)      // filter out old repos
               .map(repoView);          // map to virtual dom
};
```

PURE FUNCTIONS

```
const jsonToRepoViews = (json) => {
    return json.map(jsonToRepository)
               .filter(repoFilter)
               .map(repoView);
};
```





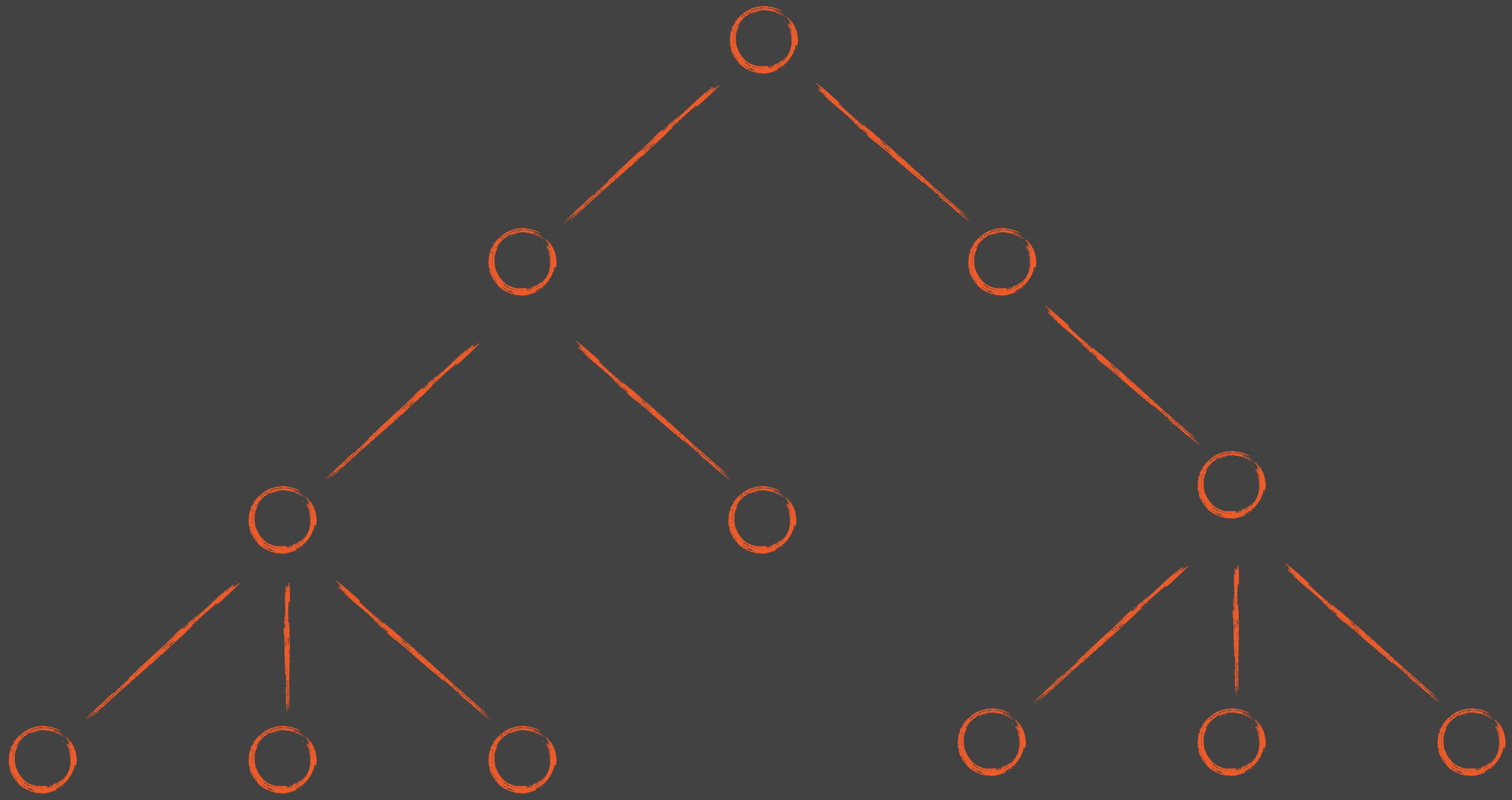
WHY PURE FUNCTIONS?

- Less mental overhead
- Simpler to test
  - No mocking requests
  - No mocking side effects
  - No bootstrapping frameworks
  - No stubbing impure functions / components
- Pure functions can be memoized

Immutability

WHY IMMUTABILITY IS IMPORTANT

- Pure functions
- Change detection
- Performance

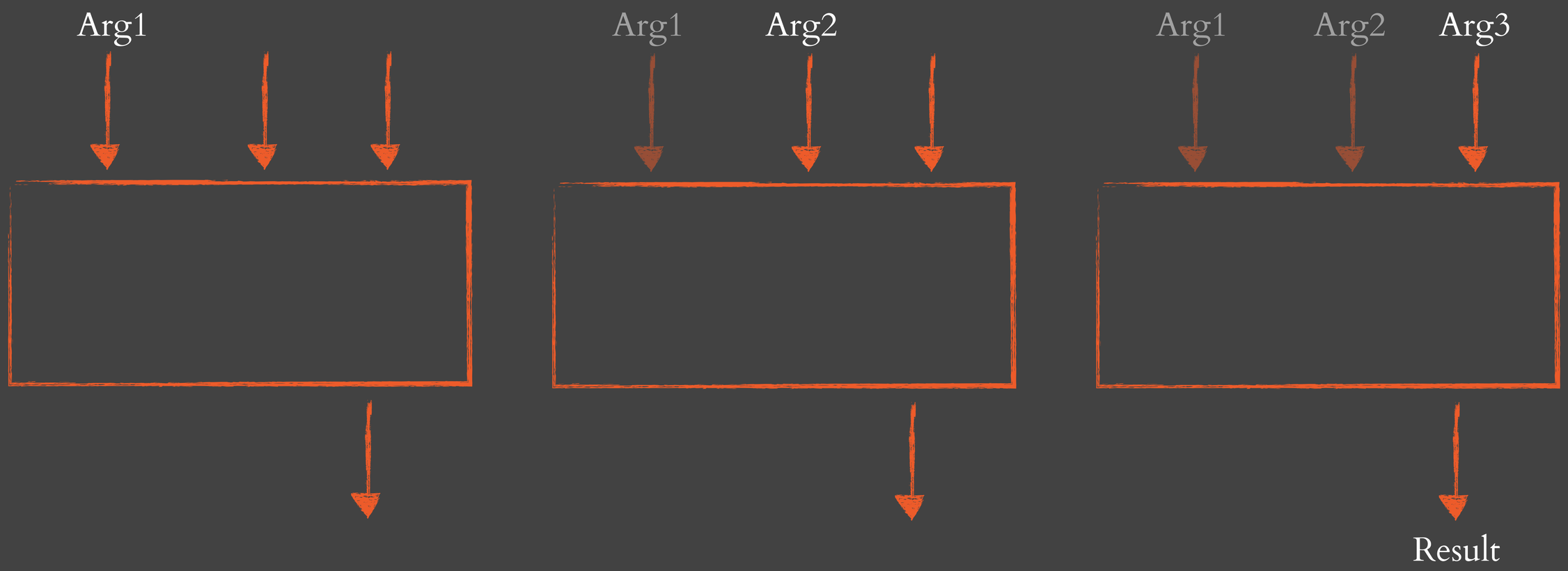


Immutability

@ckoster22

CURRY / COMPOSE

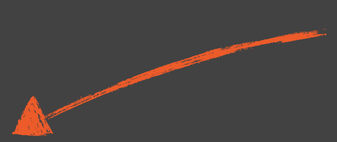
A curried function can be invoked with fewer arguments than it was expecting, and a new function will be returned that expects the remaining arguments to be supplied.


Arg1
Arg1
Arg2
Arg1
Arg2
Arg3
Result

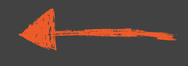
CURRY EXAMPLE


```
const addThreeNums = R.curry((a, b, c) => a + b + c);

console.log(addThreeNums(1, 2, 3)); // 6

const addFiveToTwoNums = addThreeNums(5);
console.log(addFiveToTwoNums(3, 4)); // 12

const addNine = addFiveToTwoNums(4);
console.log(addNine(10)); // 19
```

CURRY EXAMPLE




```
const addThreeNums = R.curry((a, b, c) => a + b + c);

console.log(addThreeNums(1, 2, 3)); // 6

const addFiveToTwoNums = addThreeNums(5);
console.log(addFiveToTwoNums(3, 4)); // 12

const addNine = addFiveToTwoNums(4);
console.log(addNine(10)); // 19
```



CURRY EXAMPLE

Curried function

```
const addThreeNums = R.curry((a, b, c) => a + b + c);

console.log(addThreeNums(1, 2, 3)); // 6

const addFiveToTwoNums = addThreeNums(5);
console.log(addFiveToTwoNums(3, 4)); // 12

const addNine = addFiveToTwoNums(4);
console.log(addNine(10)); // 19
```

Regular invocation

Partial application

CURRY EXAMPLE

Curried function

```
const addThreeNums = R.curry((a, b, c) => a + b + c);

console.log(addThreeNums(1, 2, 3)); // 6

const addFiveToTwoNums = addThreeNums(5);
console.log(addFiveToTwoNums(3, 4)); // 12

const addNine = addFiveToTwoNums(4);
console.log(addNine(10)); // 19
```

Regular invocation

Partial application

Partial application

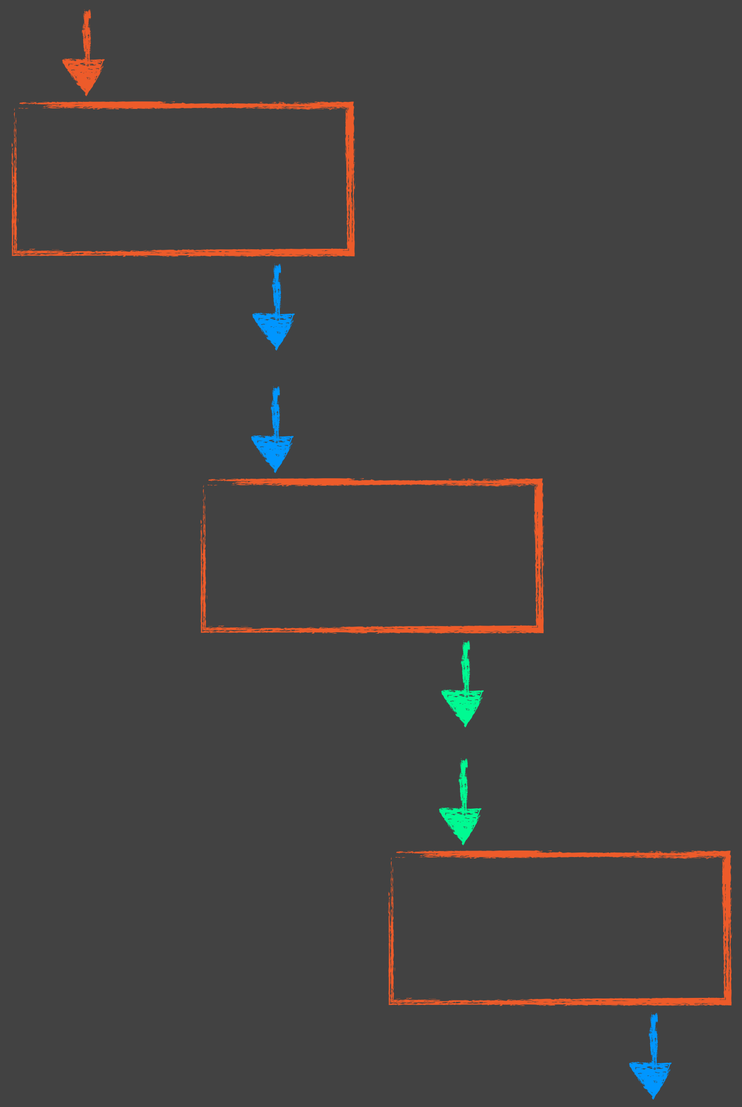
CURRY EXAMPLE

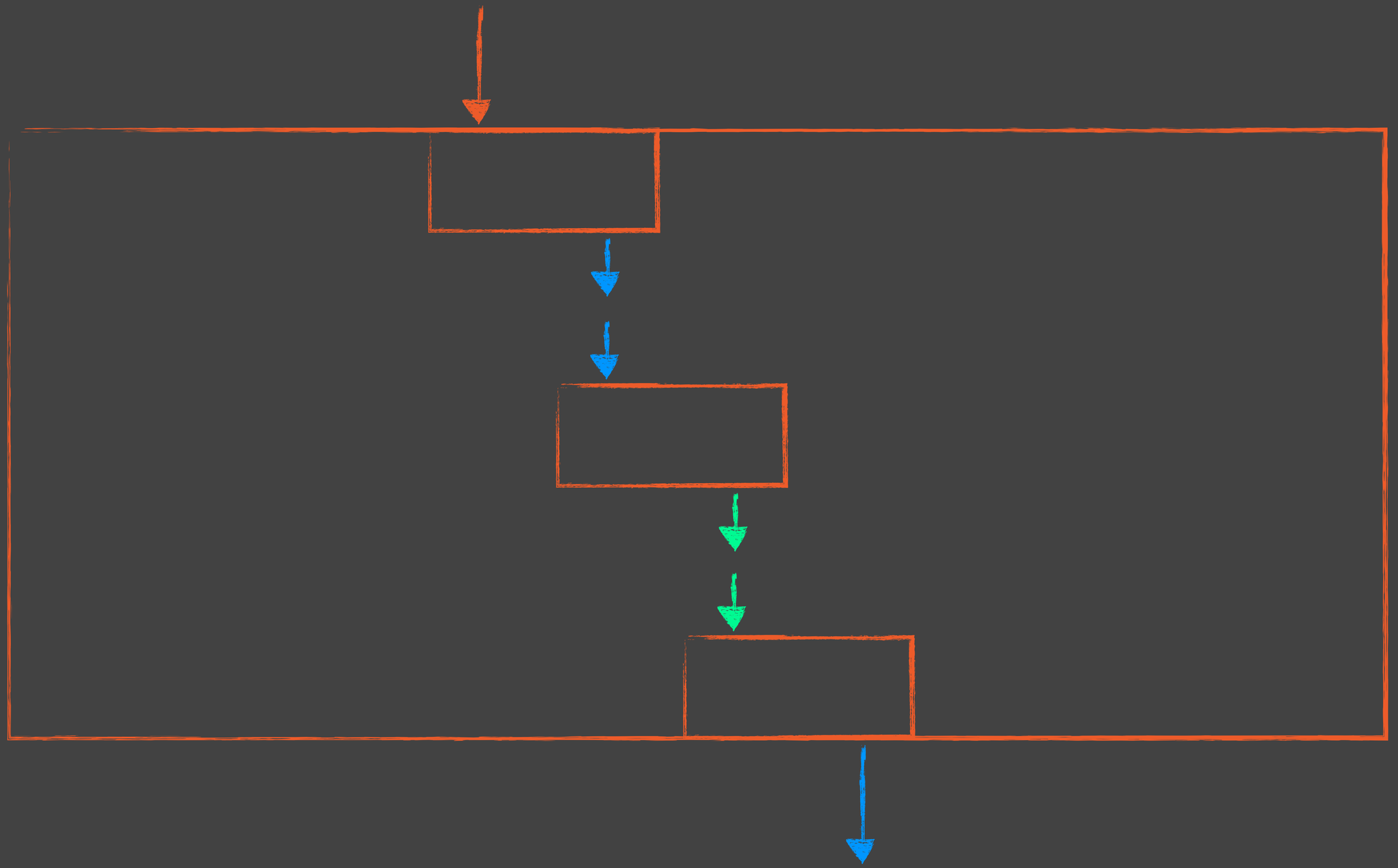
```
const repoFilter = R.curry((isFiltered, repo) => {
    if (isFiltered) {
        return repo.lastPushed.getTime() >= ninetyDaysAgo;
    } else {
        return true;
    }
});

…

const filteredRepos = repos.filter(repoFilter(appState.isFiltered));
```

Partial application

Function composition

Functions can be composed together to produce a new function



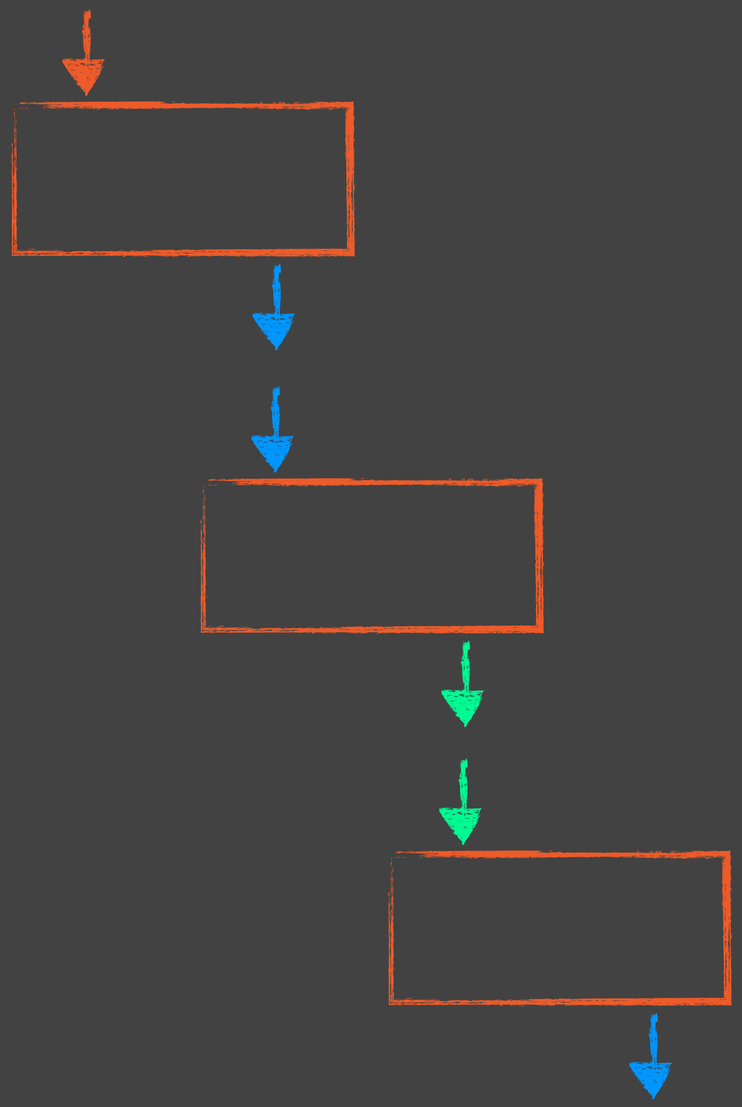


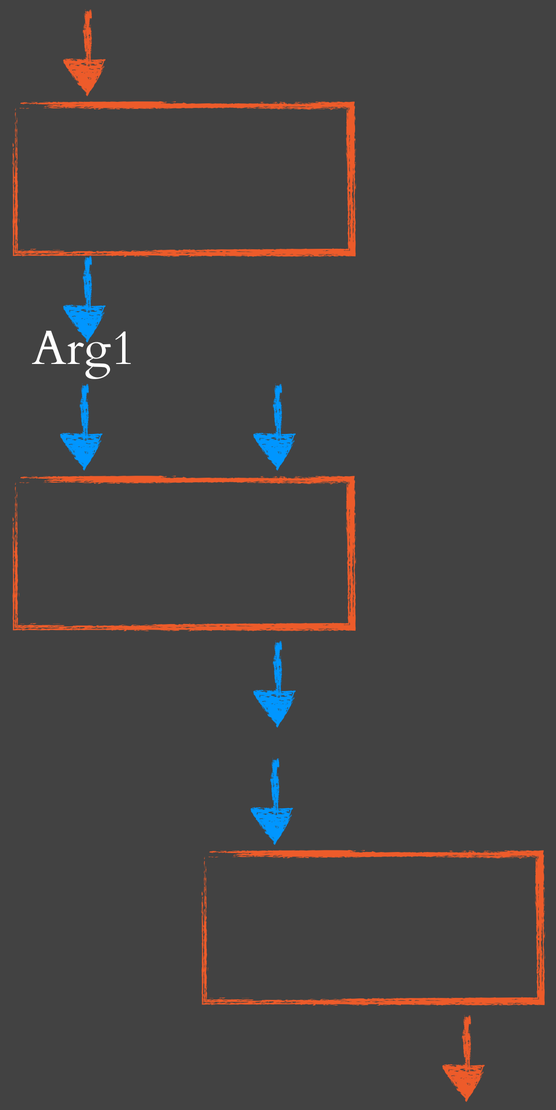
FUNCTION COMPOSITION EXAMPLE

```
const squareRoot = (num: number): number => {
    return Math.pow(num, 0.5);
};
const isEven = (num: number): boolean => {
    return num % 2 === 0;
};

const rootIsEven = R.compose(isEven, squareRoot);
rootIsEven(9); // false
rootIsEven(16); // true
```

FUNCTION COMPOSITION EXAMPLE

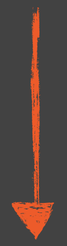
```
const squareRoot = (num: number): number => {
    return Math.pow(num, 0.5);
};
const isEven = (num: number): boolean => {
    return num % 2 === 0;
};

const rootIsEven = R.compose(isEven, squareRoot);
rootIsEven(9); // false
rootIsEven(16); // true
```

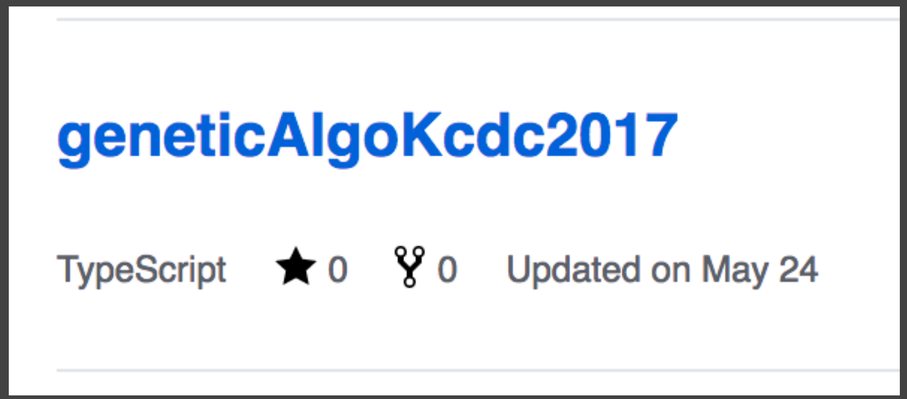



Arg1

```
const jsonToRepoViews = (json) => {
    return json.map(jsonToRepository)   // convert to repo
               .filter(repoFilter)      // filter out old repos
               .map(repoView);          // map to virtual dom
};
```

```
const jsonToRepoViews = R.compose(
    R.map(repoView),
    R.filter(repoFilter(appState.isFiltered)),
    R.map(jsonToRepository)
);

return jsonToRepoViews(json);
```





INVARIANTS

An invariant is a condition, or rule, that can be relied upon to be true during the execution of the program

INVARIANTS WE'VE SEEN SO FAR

- Immutability
- Types

INVARIANTS WE'VE SEEN SO FAR

- Immutability
- Types

```
const user = {
    name: 'John Doe',
    address: '123 Fake Street',
    city: 'Kansas City',
    state: 'Kansas'
};

const updatedUser = {
    ...user,
    name: 'Jane Doe'
};
```

Implicit / unenforced

*Except through unit testing

INVARIANTS WE'VE SEEN SO FAR

- Immutability
- Types

```
const user = Immutable({
    name: 'John Doe',
    address: '123 Fake Street',
    city: 'Kansas City',
    state: 'Kansas'
});

const updatedUser =
    user.set('name', 'Jane Doe');

user.name = 'Waldo';
```

Enforced at runtime

INVARIANTS WE'VE SEEN SO FAR

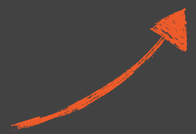
- Immutability
- Types

```
user =
    { name = "John Doe"
    , address = "123 Fake Street"
    , city = "Kansas City"
    , state = "Kansas"
    }

updatedUser =
    { user | name = "Jane Doe" }

user.name = "Waldo"
```

Enforced at compile time

INVARIANTS WE'VE SEEN SO FAR

- Immutability
- Types

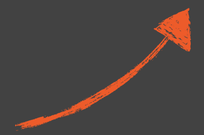


Unenforced

INVARIANTS WE'VE SEEN SO FAR

- Immutability
- Types

```
const fullname: string = 'John Doe';

console.log(fullname.toUpperCase());
console.log(fullname * 5);
```

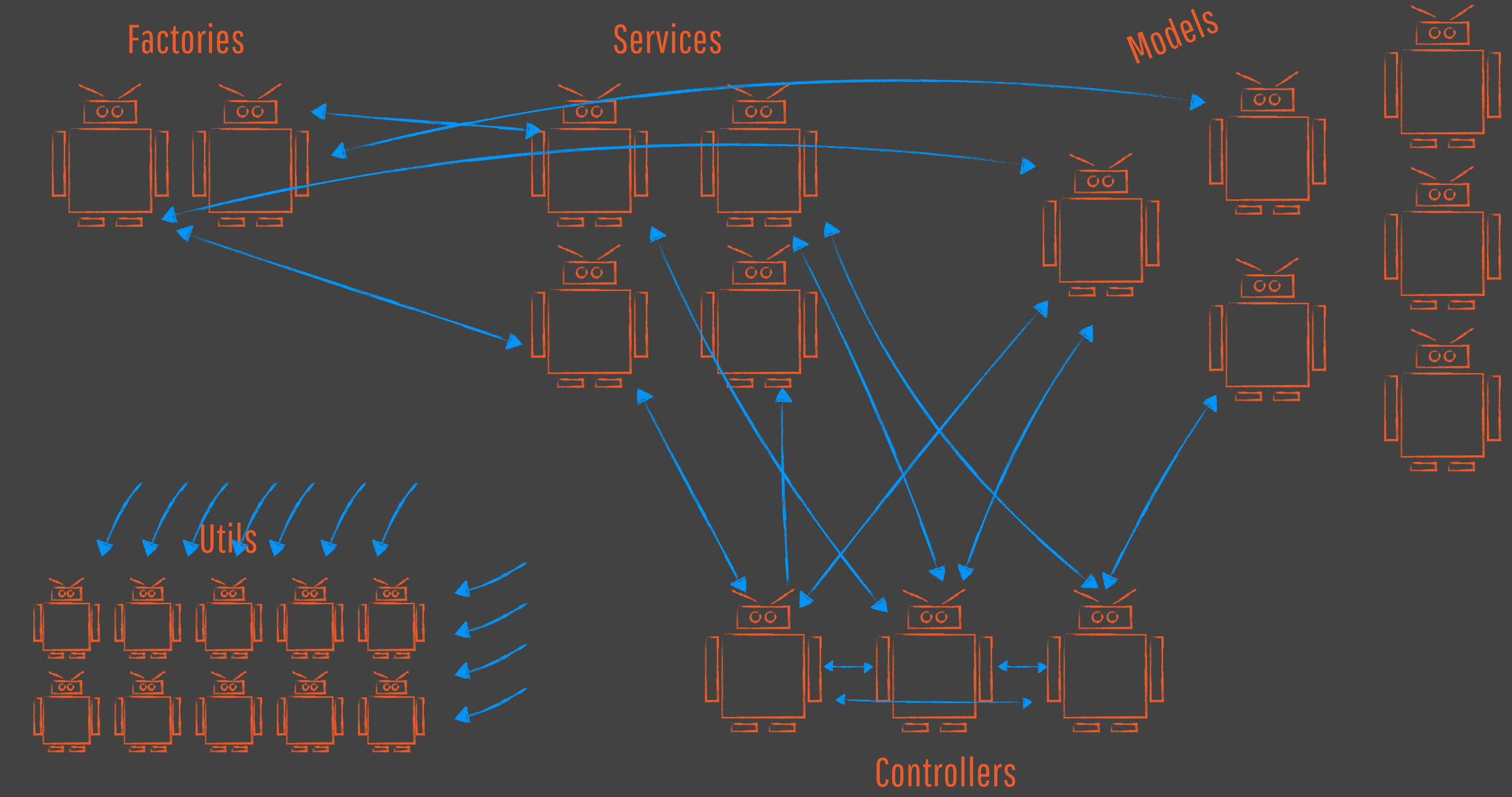


Enforced at
compile time

MORE INVARIANTS

- When to show / clear errors
- Form validation
- Autofocus the first input field
- Prevent multiple submissions
- Many, many more!

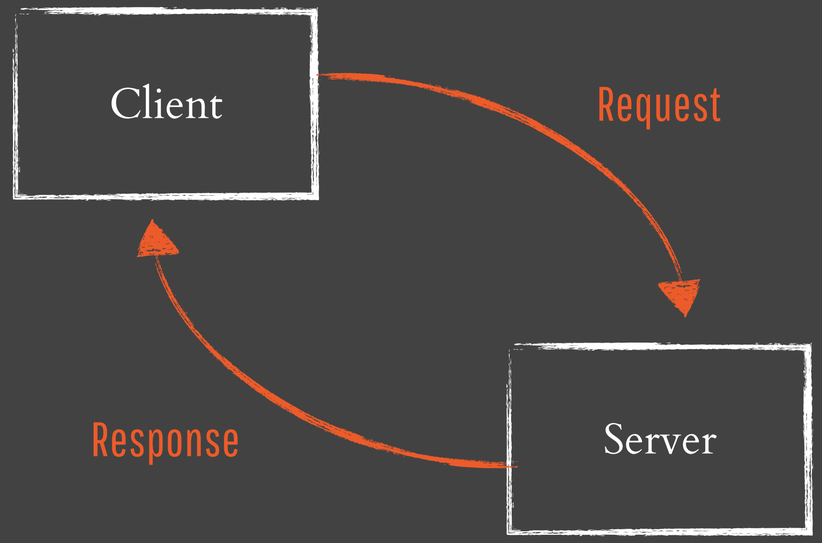
Imperative, stateful programming typically* involves invariants being maintained through state transitions


Factories
Services
Models
Utils
Controllers

Functional programming typically* involves invariants being enforced through deliberately modeled data and interfaces

USER EXPERIENCE INVARIANTS

ASYNCHRONOUS DATA



- Success
- Error
- Retrieving
- “I didn’t ask yet” (initial state)

```ts
export type AppState = {
    githubUser: string,
    isFiltered: boolean,
    repositories?: Repository[],
    isFetching: boolean,
    hasFetched: boolean,
    fetchError?: string
};

export type Repository = {
    owner: string,
    name: string,
    description?: string,
    language?: string,
    stars: number,
    forks: number,
    lastPushed: Date
};
```

- Problems
  - Implicit invariants
  - Late enforcement, if at all
    - Runtime
    - Unit tests
  - Capable of representing many bugs

```
export type AppState = {
    githubUser: string,
    isFiltered: boolean,
    data: RepositoryData
};

export type RepositoryData = Initial | Retrieving | Success | Failure;
export type Initial = {
    kind: 'Initial'
};
export type Retrieving = {
    kind: 'Retrieving'
};
export type Success = {
    kind: 'Success',
    repositories: Repository[]
};
export type Failure = {
    kind: 'Failure',
    message: string
};
```

Enforced at compile time

Explicit invariant

Only valid states are representable

```
let view;

switch (appState.data.kind) {
    case 'Initial':
        view = <span>About to retrieve repository data..</span>;
        break;
    case 'Retrieving':
        view = <span>Retrieving. Please wait..</span>;
        break;
    case 'Success':
        view = appState.data.repositories
                            .filter(repoFilter(appState.isFiltered))
                            .map(repoView);
        break;
    case 'Failure':
        view = <span>There was a problem retrieving the repositories.</span>;
        break;
}
```

STATEFUL PROGRAMMING

- What / how / when are complected
- As the app scales, so does complexity
- Invariants managed through state transitions

STATELESS PROGRAMMING

- Separation of what / how / when
- Simplicity & performance
- Invariance through deliberately modeled data and functions

RESOURCES

Lee Byron - Immutable App Architecture

Rich Hickey - Simple Made Easy

Richard Feldman - Making Impossible States Impossible

Thanks!

@ckoster22

THANKS

Charlie Koster |   @ckoster22

https://github.com/ckoster22

https://medium.com/@ckoster22

https://twitter.com/ckoster22