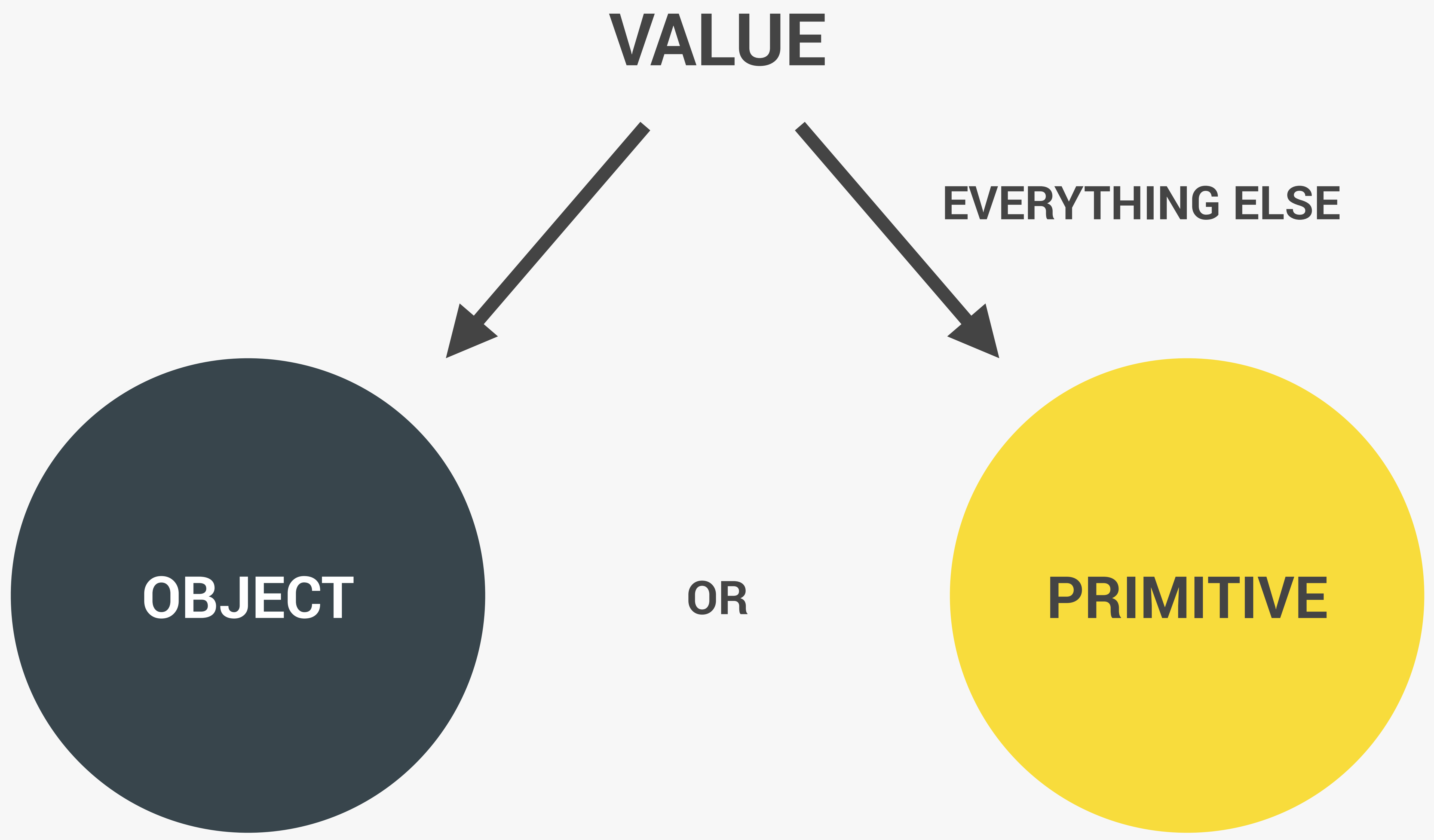
OBJECTS AND PRIMITIVES



```
let me = {
  name: 'Jonas'
};
```

```
let firstName = 'Jonas';
let age = 30;
```

THE 7 PRIMITIVE DATA TYPES

1. **Number:** Floating point numbers 👉 Used for decimals and integers `let age = 23;`
2. **String:** Sequence of characters 👉 Used for text `let firstName = 'Jonas';`
3. **Boolean:** Logical type that can only be `true` or `false` 👉 Used for taking decisions `let fullAge = true;`
4. **Undefined:** Value taken by a variable that is not yet defined ('empty value') `let children;`
5. **Null:** Also means 'empty value'
6. **Symbol (ES2015):** Value that is unique and cannot be changed *[Not useful for now]*
7. **BigInt (ES2020):** Larger integers than the Number type can hold

☝️ **JavaScript has dynamic typing:** We do ***not*** have to manually define the data type of the value stored in a variable. Instead, data types are determined **automatically**.

Value has type, NOT variable!

FUNCTIONS REVIEW: 3 DIFFERENT FUNCTION TYPES

👉 **Function declaration**

Function that can be used before it's declared

👉 **Function expression**

Essentially a function *value* stored in a variable

👉 **Arrow function**

Great for a quick one-line functions. Has no `this` keyword (more later...)

```
function calcAge(birthYear) {
  return 2037 - birthYear;
}

const calcAge = function (birthYear) {
  return 2037 - birthYear;
};

const calcAge = birthYear => 2037 - birthYear;
```

☝️ Three different ways of writing functions, but they all work in a similar way: receive **input** data, **transform** data, and then **output** data.

SCOPING AND SCOPE IN JAVASCRIPT: CONCEPTS

EXECUTION CONTEXT

- 👉 Variable environment
- 👉 Scope chain
- 👉 `this` keyword

SCOPE CONCEPTS

- 👉 **Scoping:** How our program's variables are **organized** and **accessed**. *"Where do variables live?"* or *"Where can we access a certain variable, and where not?"*;
- 👉 **Lexical scoping:** Scoping is controlled by **placement** of functions and blocks in the code;
- 👉 **Scope:** Space or environment in which a certain variable is **declared** (*variable environment in case of functions*). There is **global** scope, **function** scope, and **block** scope;
- 👉 **Scope of a variable:** Region of our code where a certain variable can be **accessed**.

THE 3 TYPES OF SCOPE

GLOBAL SCOPE

```
const me = 'Jonas';
const job = 'teacher';
const year = 1989;
```

- 👉 Outside of **any** function or block
- 👉 Variables declared in global scope are accessible **everywhere**

FUNCTION SCOPE

```
function calcAge(birthYear) {
  const now = 2037;
  const age = now - birthYear;
  return age;
}

console.log(now); // ReferenceError
```

- 👉 Variables are accessible only **inside function**, **NOT** outside
- 👉 Also called local scope

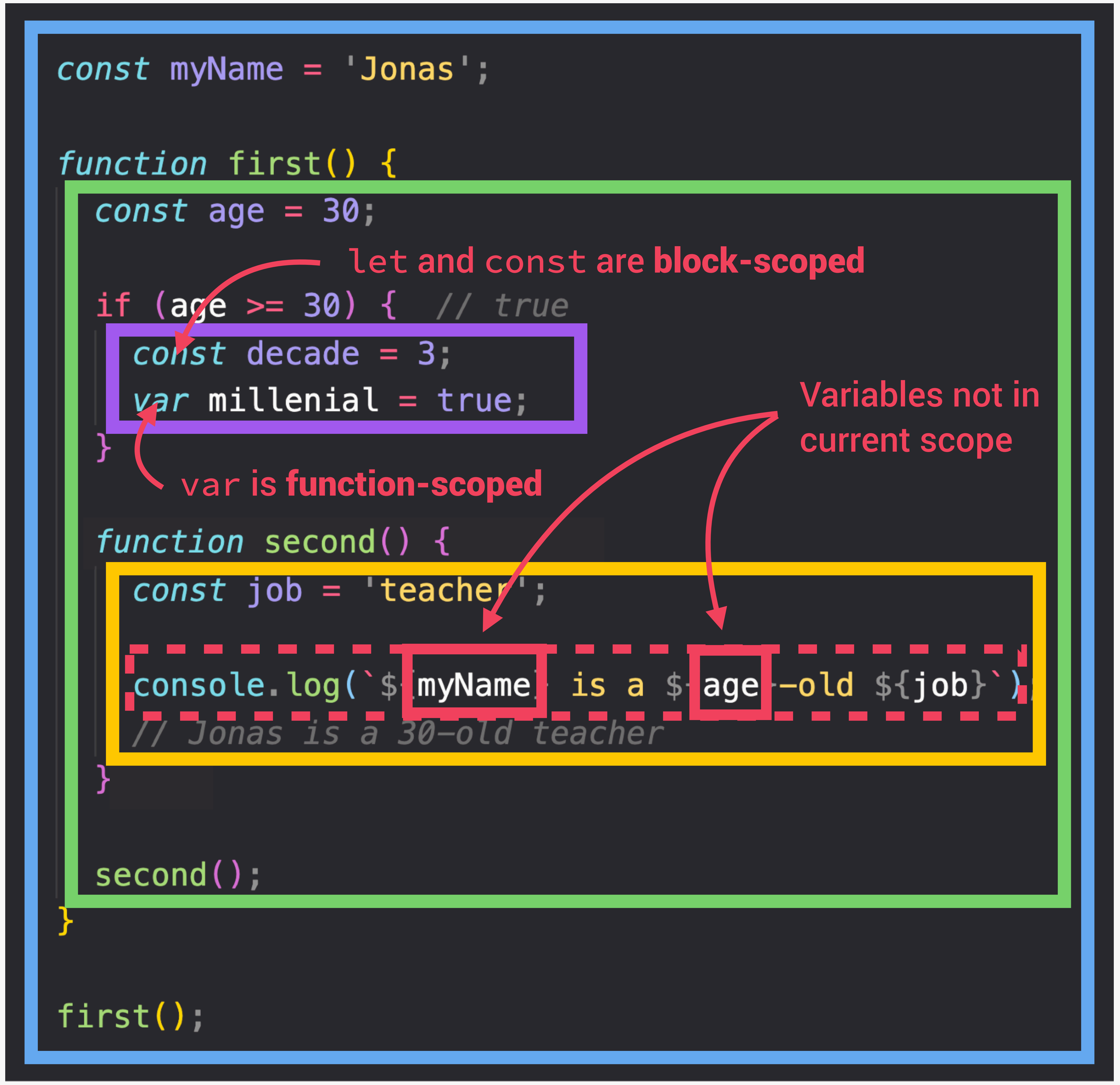
BLOCK SCOPE (ES6)

```
if (year >= 1981 && year <= 1996) {
  const millenial = true;
  const food = 'Avocado toast';
}

console.log(millenial); // ReferenceError
```

Example: `if` block, `for` loop block, etc.

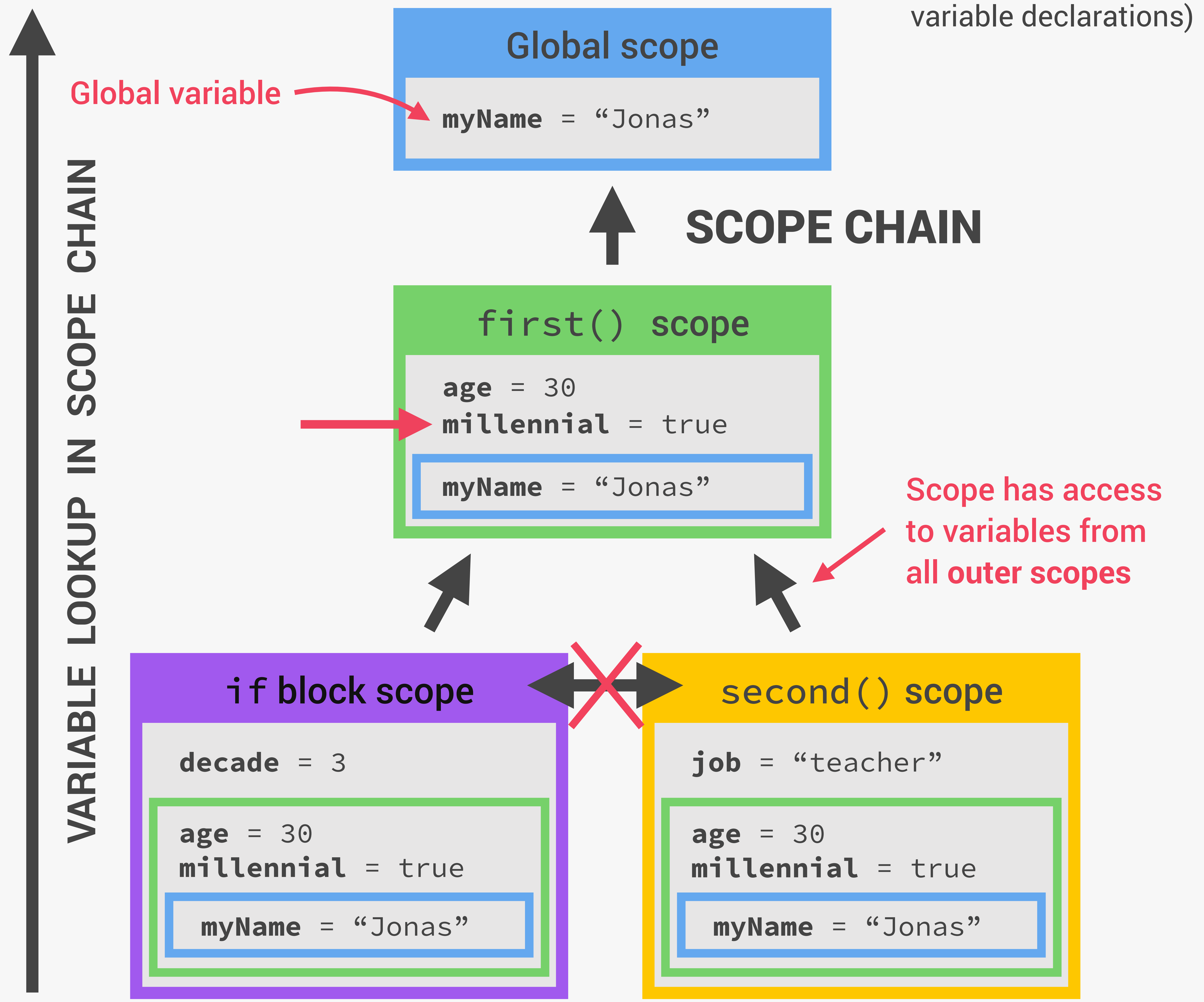
- 👉 Variables are accessible only **inside block** (block scoped)
- ⚠️ **HOWEVER**, this only applies to **`let`** and **`const`** variables!
- 👉 Functions are **also block scoped** (only in strict mode)

THE SCOPE CHAIN



```
const myName = 'Jonas';

function first() {
  const age = 30;

  if (age >= 30) {  // true
    const decade = 3;
    var millenial = true;
  }

  function second() {
    const job = 'teacher';

    console.log(`${myName} is a ${age}-old ${job}`);
    // Jonas is a 30-old teacher
  }

  second();
}

first();
```

let and const are block-scoped
var is function-scoped
Variables not in current scope



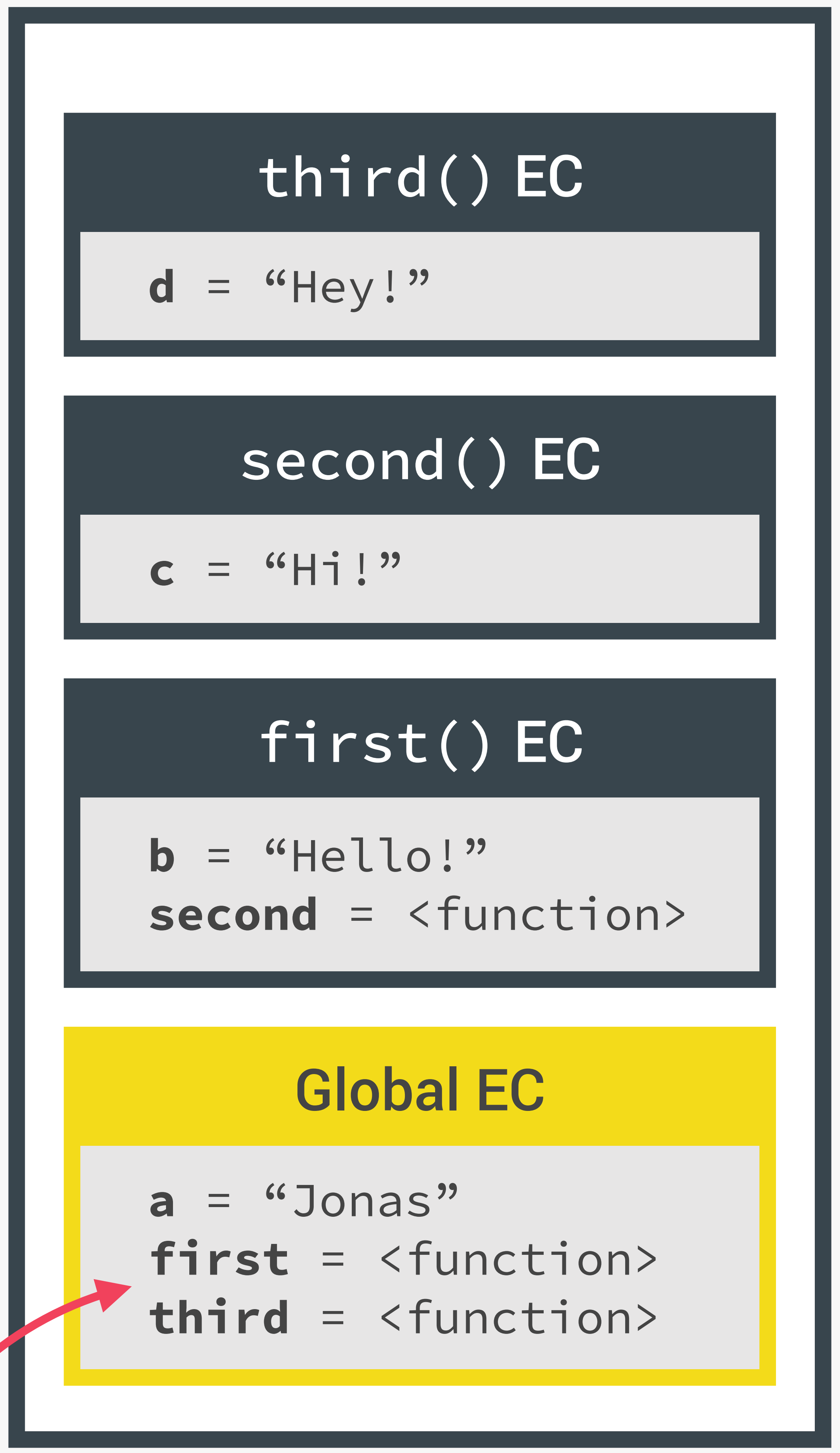
(Considering only variable declarations)

SCOPE CHAIN VS. CALL STACK

```
const a = 'Jonas';
first();

function first() {
  const b = 'Hello!';
  second();

  function second() {
    const c = 'Hi!';
    third();
  }
}

function third() {
  const d = 'Hey!';
  console.log(d + c + b + a);
  // ReferenceError
}
```

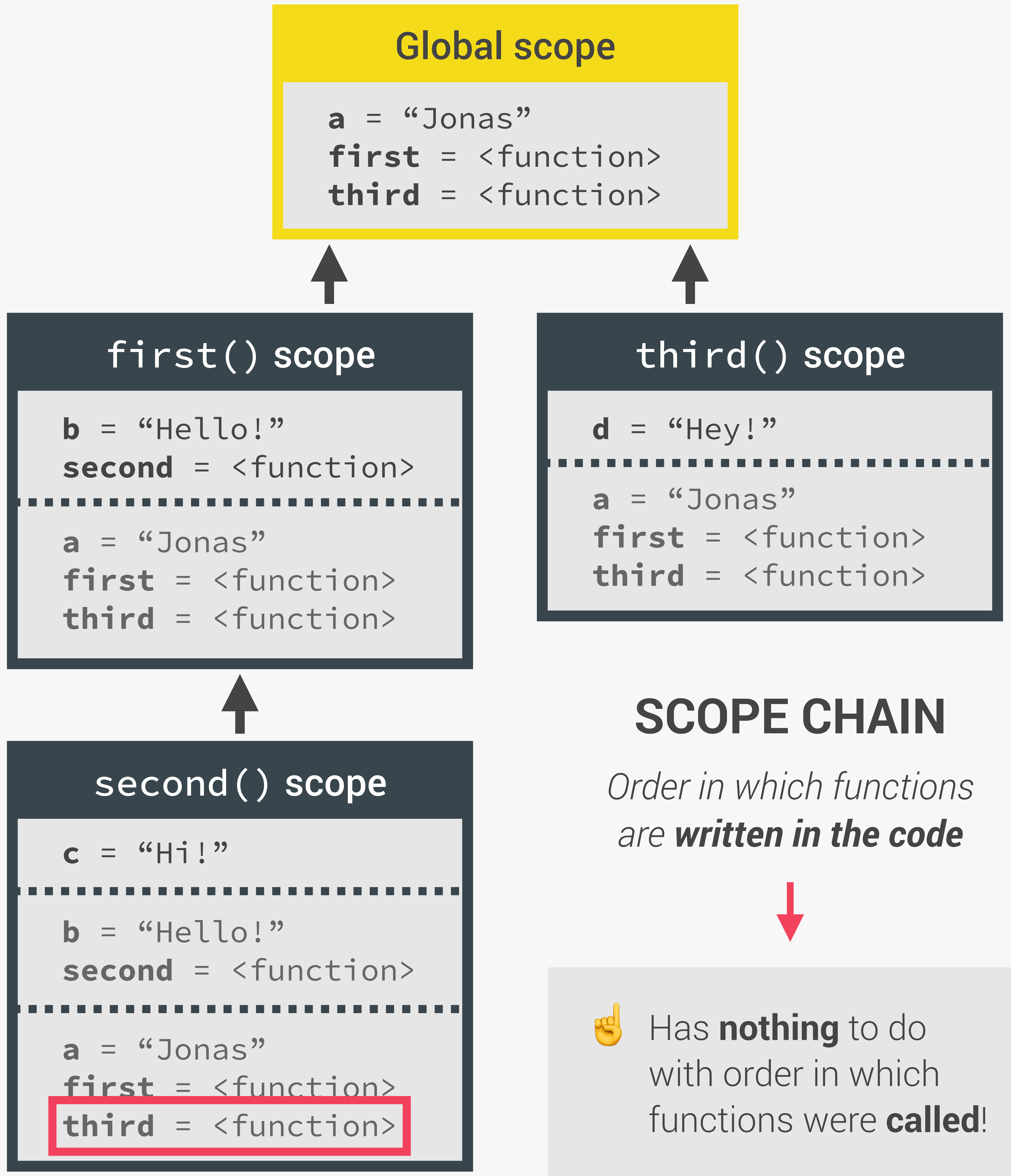
c and b can NOT be found in `third()` scope!

Variable environment (VE)



CALL STACK

*Order in which functions were **called***



SCOPE CHAIN

*Order in which functions are **written in the code***

☝️ Has **nothing** to do with order in which functions were **called**!

ARRAYS VS. SETS AND OBJECTS VS. MAPS

ARRAYS VS. SETS

ARRAYS

```
tasks = ['Code', 'Eat', 'Code'];
// ["Code", "Eat", "Code"]
```

- 👉 Use when you need **ordered** list of values (might contain duplicates)
- 👉 Use when you need to **manipulate** data

SETS

```
tasks = new Set(['Code', 'Eat', 'Code']);
// {"Code", "Eat"}
```

- 👉 Use when you need to work with **unique** values
- 👉 Use when **high-performance** is *really* important
- 👉 Use to **remove duplicates** from arrays

OBJECTS VS. MAPS

OBJECTS

```
task = {
  task: 'Code',
  date: 'today',
  repeat: true
};
```

- 👉 More "traditional" key/value store ("abused" objects)
- 👉 Easier to write and access values with `.` and `[]`

- 👉 Use when you need to include **functions** (methods)
- 👉 Use when working with JSON (can convert to map)

MAPS

```
task = new Map([
  ['task', 'Code'],
  ['date', 'today'],
  [false, 'Start coding!']
]);
```

- 👉 Better performance
- 👉 Keys can have **any** data type
- 👉 Easy to iterate
- 👉 Easy to compute size

- 👉 Use when you simply need to map key to values
- 👉 Use when you need keys that are **not** strings

FIRST-CLASS VS. HIGHER-ORDER FUNCTIONS

FIRST-CLASS FUNCTIONS

- 👉 JavaScript treats functions as **first-class citizens**
- 👉 This means that functions are **simply values**
- 👉 Functions are just another **"type" of object**

👉 Store functions in variables or properties:

```
const add = (a, b) => a + b;
```

```
const counter = {
  value: 23,
  inc: function() { this.value++; }
```

👉 Pass functions as arguments to OTHER functions:

```
const greet = () => console.log('Hey Jonas');
btnClose.addEventListener('click', greet)
```

👉 Return functions FROM functions

👉 Call methods on functions:

```
counter.inc.bind(someOtherObject);
```

HIGHER-ORDER FUNCTIONS

- 👉 A function that **receives** another function as an argument, that **returns** a new function, or **both**
- 👉 This is only possible because of first-class functions

1 Function that receives another function

```
const greet = () => console.log('Hey Jonas');
btnClose.addEventListener('click', greet)
```

Higher-order function

Callback function

✅📞💬

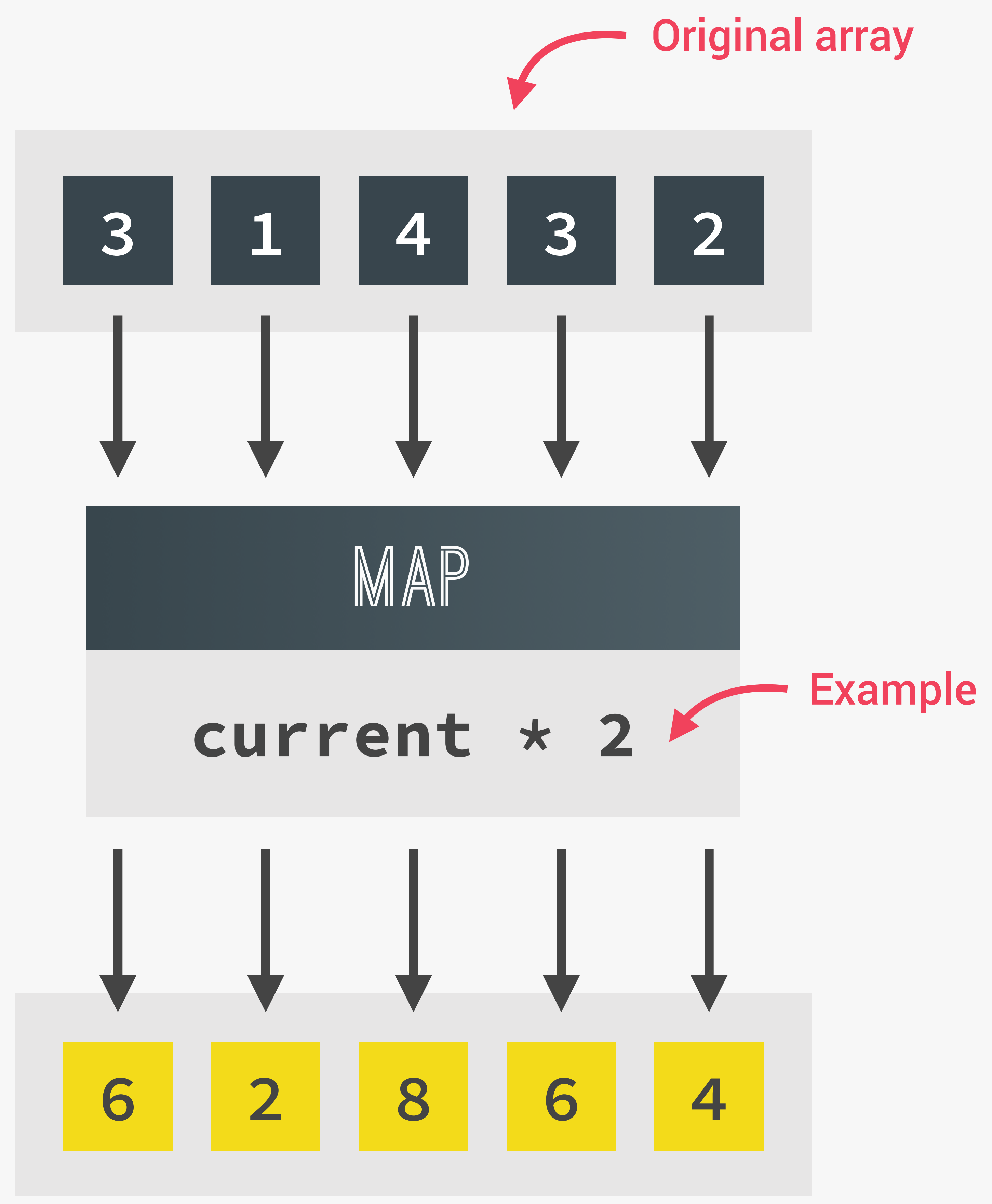
2 Function that returns new function

```
function count() {
  let counter = 0;
  return function() {
    counter++;
  };
}
```

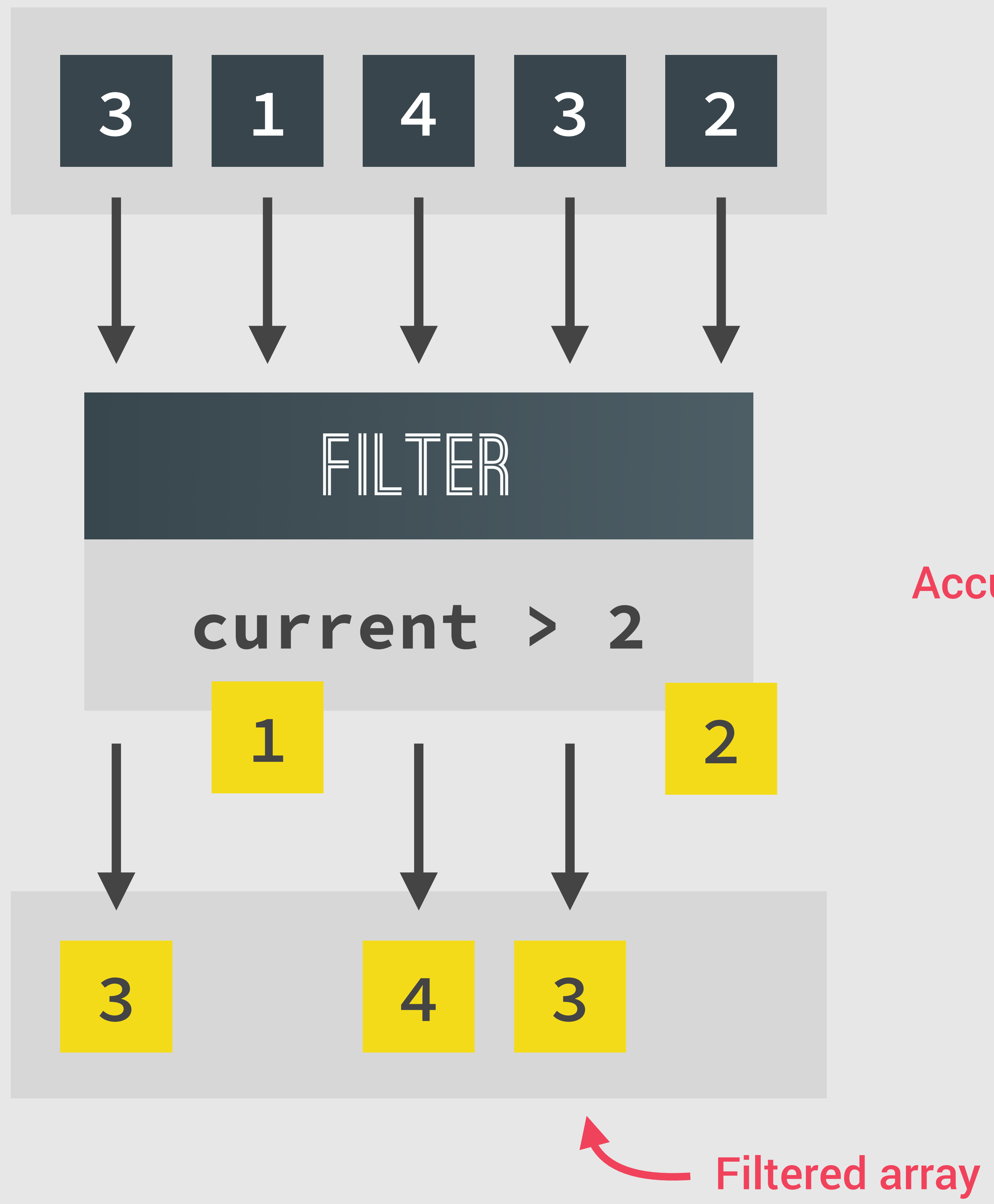
Higher-order function

Returned function

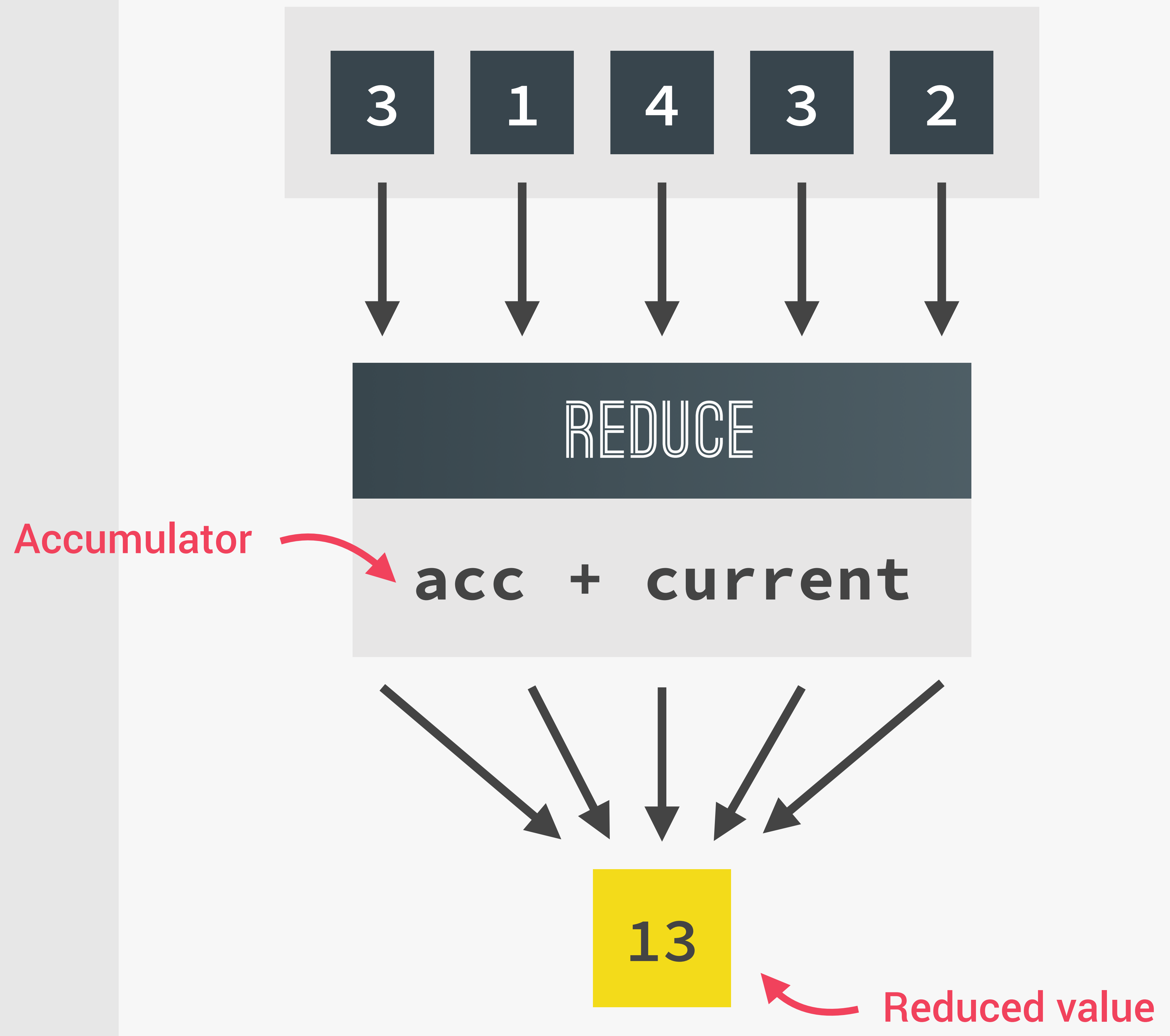
DATA TRANSFORMATIONS WITH MAP, FILTER AND REDUCE



👉 `map` returns a **new array** containing the results of applying an operation on all original array elements



👉 `filter` returns a **new array** containing the array elements that passed a specified **test condition**



👉 `reduce` boils ("reduces") all array elements down to one single value (e.g. adding all elements together)

WHICH ARRAY METHOD TO USE? 🤔

"I WANT...:"

To mutate original array

- Add to original:
  - `.push` *(end)*
  - `.unshift` *(start)*
- Remove from original:
  - `.pop` *(end)*
  - `.shift` *(start)*
  - `.splice` *(any)*
- Others:
  - `.reverse`
  - `.sort`
  - `.fill`

A new array

- Computed from original:
  - `.map` *(loop)*
- Filtered using condition:
  - `.filter`
- Portion of original:
  - `.slice`
- Adding original to other:
  - `.concat`
- Flattening the original:
  - `.flat`
  - `.flatMap`

An array index

- Based on value:
  - `.indexOf`
- Based on test condition:
  - `.findIndex`

An array element

- Based on test condition:
  - `.find`

Know if array includes

- Based on value:
  - `.includes`
- Based on test condition:
  - `.some`
  - `.every`

A new string

- Based on separator string:
  - `.join`

To transform to value

- Based on accumulator:
  - `.reduce`

(Boil down array to single value of any type: number, string, boolean, or even new array or object)

To just loop array

- Based on callback:
  - `.forEach`

(Does not create a new array, just loops over it)