JONAS.IO
SCHMEDTMANN

Subscribe here

THE COMPLETE JAVASCRIPT COURSE

FROM ZERO TO EXPERT!

SLIDES FOR THEORY LECTURES

(DON'T SKIP THEM, THEY ARE SUPER IMPORTANT 🤓)

Follow me here

@JONASSCHMEDTMAN

📖 TABLE OF CONTENTS: THEORY LECTURES (CLICK THE TITLES)

WHAT IS JAVASCRIPT?

JAVASCRIPT

We don't have to worry about complex stuff like memory management

JAVASCRIPT IS A HIGH-LEVEL, OBJECT-ORIENTED, MULTI-PARADIGM PROGRAMMING LANGUAGE. 🤯

Based on objects, for storing most kinds of data

We can use different styles of programming

Instruct computer to *do* things

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.

JONAS.IO
SCHMEDTMANN

THE COMPLETE JAVASCRIPT COURSE

FROM ZERO TO EXPERT!

@JONASSCHMEDTMAN

SECTION

JAVASCRIPT IN THE BROWSER: DOM AND EVENTS FUNDAMENTALS

LECTURE

WHAT'S THE DOM AND DOM MANIPULATION

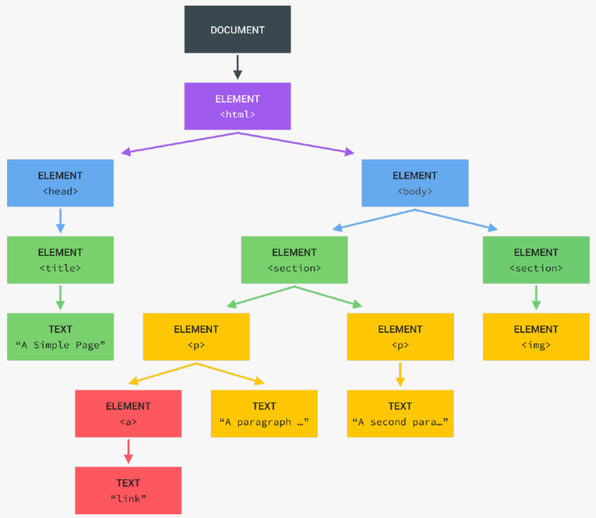
JS

WHAT IS THE DOM?

DOM

DOCUMENT OBJECT MODEL: STRUCTURED REPRESENTATION OF HTML DOCUMENTS. ALLOWS JAVASCRIPT TO ACCESS HTML ELEMENTS AND STYLES TO MANIPULATE THEM.

Tree structure, generated by browser on HTML load



Change text, HTML attributes, and even CSS styles

DOM !== JAVASCRIPT 🧐

DOM Methods and Properties for DOM Manipulation

👎

NOT PART OF

JS

ecma INTERNATIONAL

For example
`document.querySelector()`

API: Application Programming Interface

WEB APIs

DOM Methods and Properties

Timers

Fetch

👍

CAN INTERACT WITH

JS

JONAS.IO
SCHMEDTMANN

THE COMPLETE JAVASCRIPT COURSE

FROM ZERO TO EXPERT!

@JONASSCHMEDTMAN

SECTION

HOW JAVASCRIPT WORKS BEHIND THE SCENES

LECTURE

AN HIGH-LEVEL OVERVIEW OF JAVASCRIPT

JS

WHAT IS JAVASCRIPT: REVISITED

JAVASCRIPT

JAVASCRIPT IS A HIGH-LEVEL, PROTOTYPE-BASED OBJECT-ORIENTED, MULTI-PARADIGM, INTERPRETED OR JUST-IN-TIME COMPILED, DYNAMIC, SINGLE-THREADED, GARBAGE-COLLECTED PROGRAMMING LANGUAGE WITH FIRST-CLASS FUNCTIONS AND A NON-BLOCKING EVENT LOOP CONCURRENCY MODEL. 🤔 🤯 🤣

DECONSTRUCTING THE MONSTER DEFINITION

- High-level
- Garbage-collected
- Interpreted or just-in-time compiled
- Multi-paradigm
- Prototype-based object-oriented
- First-class functions
- Dynamic
- Single-threaded
- Non-blocking event loop

👉 **Paradigm**: An approach and mindset of structuring code, which will direct your coding style and technique.

The one we've been using so far

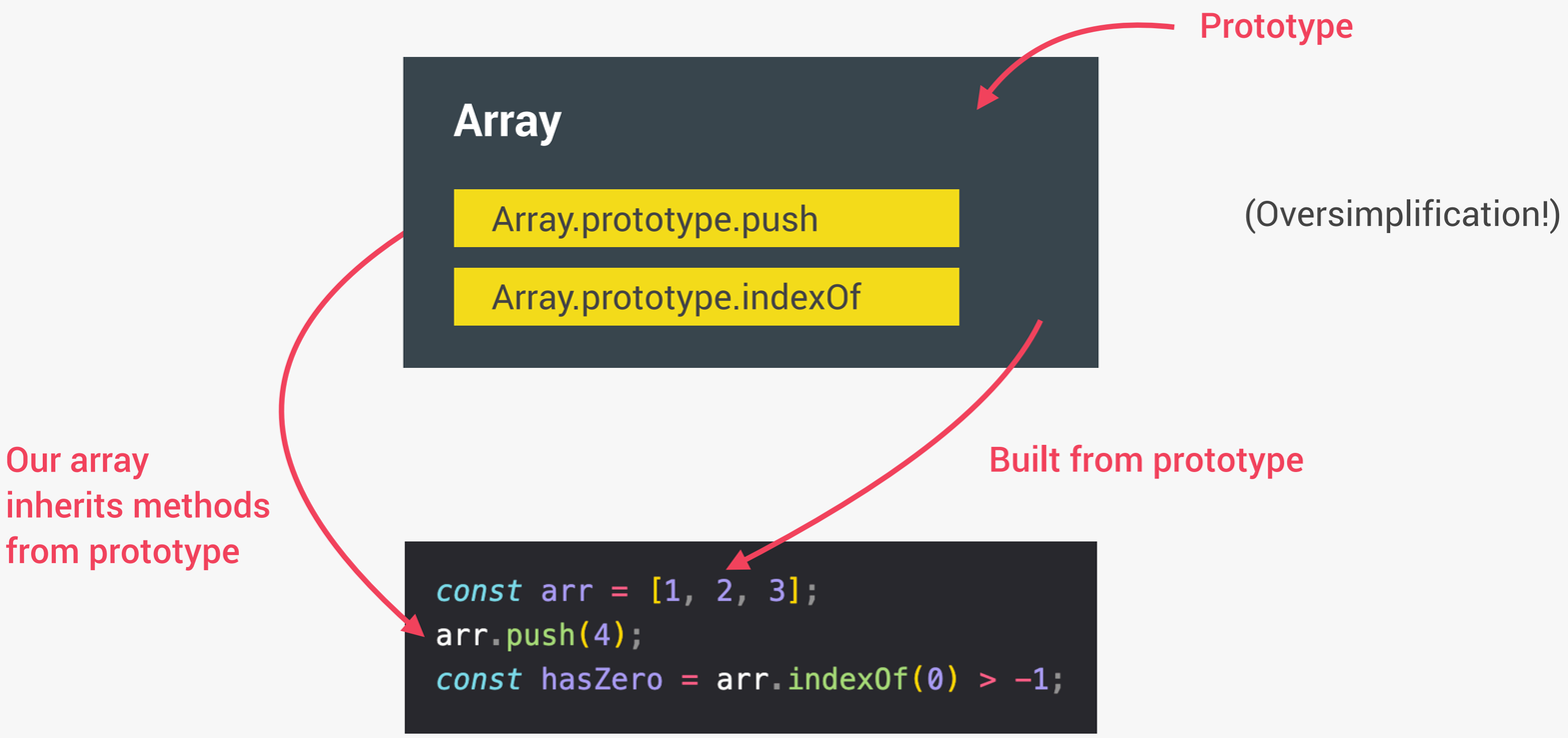
1. Procedural programming
2. Object-oriented programming (OOP)
3. Functional programming (FP)

☝️ **Imperative vs.**
👋 **Declarative**

More about this later in **Multiple Sections** 👉

DECONSTRUCTING THE MONSTER DEFINITION

- High-level
- Garbage-collected
- Interpreted or just-in-time compiled
- Multi-paradigm
- Prototype-based object-oriented
- First-class functions
- Dynamic
- Single-threaded
- Non-blocking event loop



More about this in Section **Object Oriented Programming** 👉

DECONSTRUCTING THE MONSTER DEFINITION

- High-level
- Garbage-collected
- Interpreted or just-in-time compiled
- Multi-paradigm
- Prototype-based object-oriented
- First-class functions
- Dynamic
- Single-threaded
- Non-blocking event loop

👉 In a language with **first-class functions**, functions are simply **treated as variables**. We can pass them into other functions, and return them from functions.

```
const closeModal = () => {
  modal.classList.add("hidden");
  overlay.classList.add("hidden");
};

overlay.addEventListener("click", closeModal);
```

Passing a function into another function as an argument: First-class functions!

More about this in Section **A Closer Look at Functions** 👉

DECONSTRUCTING THE MONSTER DEFINITION

- High-level
- Garbage-collected
- Interpreted or just-in-time compiled
- Multi-paradigm
- Prototype-based object-oriented
- First-class functions
- Dynamic
- Single-threaded
- Non-blocking event loop

👉 **Concurrency model**: how the JavaScript engine handles multiple tasks happening at the same time.

↓ Why do we need that?

👉 JavaScript runs in one **single thread**, so it can only do one thing at a time.

↓ So what about a long-running task?

👉 Sounds like it would block the single thread. However, we want non-blocking behavior!

↓ How do we achieve that?

(Oversimplification!)

👉 By using an **event loop**: takes long running tasks, executes them in the "background", and puts them back in the main thread once they are finished.

More about this **Later in this Section** 👇

JONAS.IO
SCHMEDTMANN

THE COMPLETE JAVASCRIPT COURSE

FROM ZERO TO EXPERT!

@JONASSCHMEDTMAN

SECTION

HOW JAVASCRIPT WORKS BEHIND THE SCENES

LECTURE

THE JAVASCRIPT ENGINE AND RUNTIME

JS

MODERN JUST-IN-TIME COMPILATION OF JAVASCRIPT

```
document.querySelector(".again").addEventListener("click", () => {
  document.querySelector(".message").textContent = "Start guessing...";
  document.querySelector(".number").textContent = "?";
  document.querySelector(".guess").value = "";
});
```

~~"JavaScript is an interpreted language"~~

JS ENGINE

Happens in special threads that we can't access from code

Parsing

AST

Example

Compilation

Just-in-time compilation

```
11010110101110101011101110100100101001111101110111111010001000010100101110000011101001001001111010
```

Optimization

Execution

Happens in Call Stack

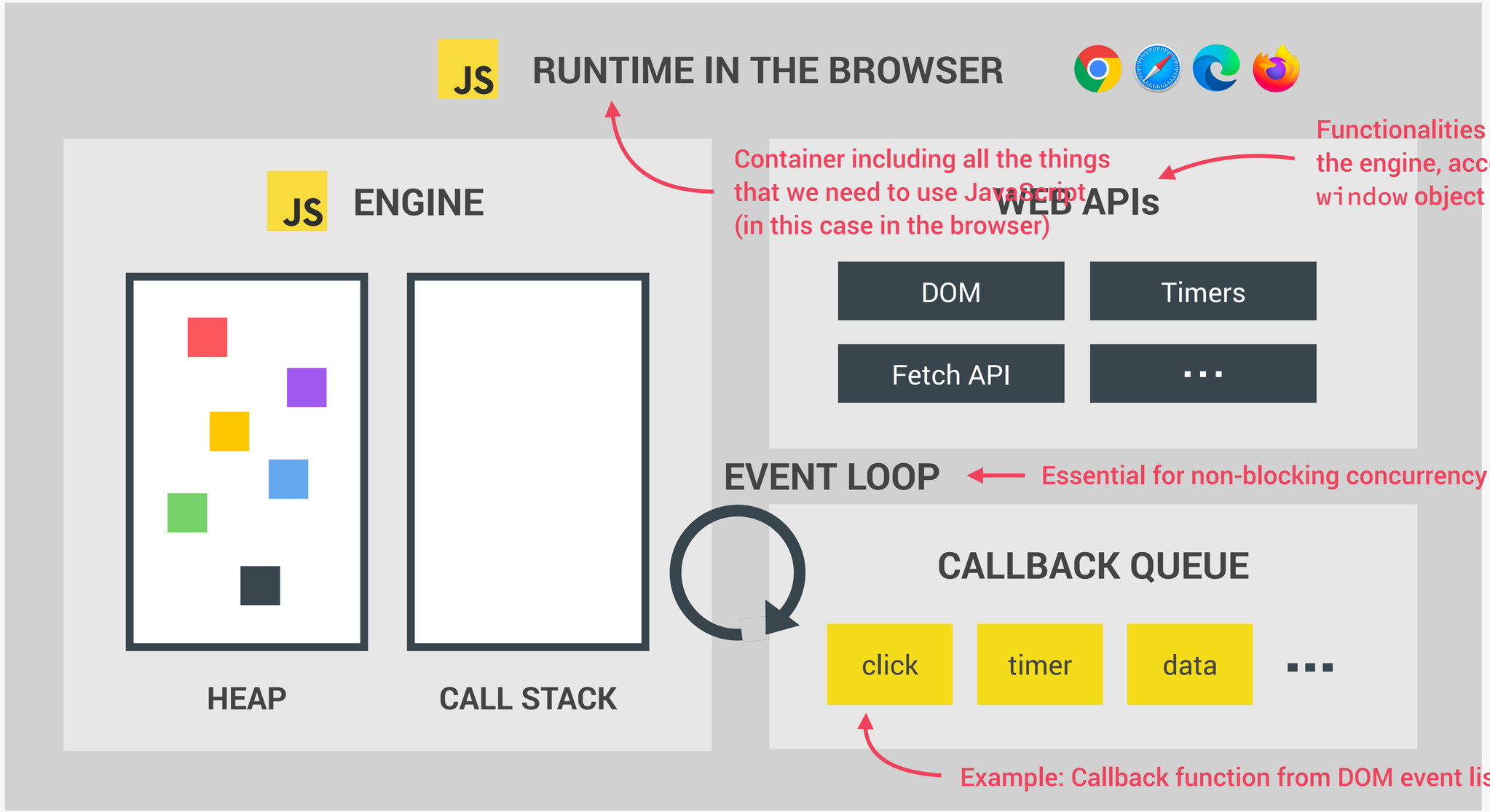
During execution!

👉 AST Example

```
const x = 23;
```

```
- VariableDeclaration {
    start: 0
    end: 13
  - declarations: [
    - VariableDeclarator {
        start: 6
        end: 12
      - id: Identifier {
          start: 6
          end: 7
          name: "x"
        }
      - init: Literal = $node {
          start: 10
          end: 12
          value: 23
          raw: "23"
        }
    }
  ]
  kind: "const"
```

THE BIGGER PICTURE: JAVASCRIPT RUNTIME



JONAS.IO
SCHMEDTMANN

THE COMPLETE JAVASCRIPT COURSE

FROM ZERO TO EXPERT!

@JONASSCHMEDTMAN

SECTION

HOW JAVASCRIPT WORKS BEHIND THE SCENES

LECTURE

EXECUTION CONTEXTS AND THE CALL STACK

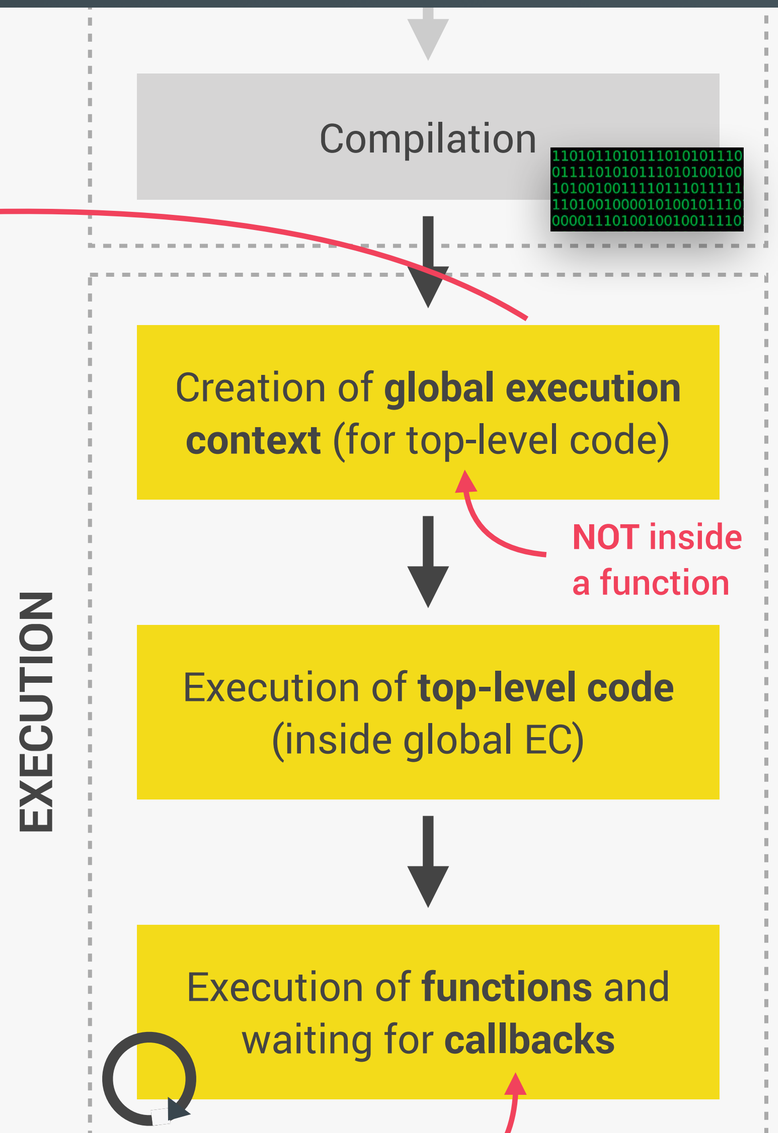
JS

WHAT IS AN EXECUTION CONTEXT?

👉 Human-readable code:

```
const name = 'Jonas';

const first = () => {
  let a = 1;
  const b = second();
  a = a + b;
  return a;
};

function second() {
  var c = 2;
  return c;
}
```

Function body only executed when called!



EXECUTION CONTEXT

Environment in which a piece of JavaScript is executed. Stores all the necessary information for some code to be executed.

👉 **Exactly <u>one</u> global execution context (EC):** Default context, created for code that is not inside any function (top-level).

👉 **One execution context <u>per function</u>:** For each function call, a new execution context is created.

All together make the call stack

EXECUTION CONTEXT IN DETAIL

WHAT'S INSIDE EXECUTION CONTEXT?

1. Variable Environment
   - 👉 `let`, `const` and `var` declarations
   - 👉 Functions
   - 👉 ~~`arguments`~~ object
2. Scope chain
3. ~~this~~ keyword

NOT in arrow functions!

Generated during “creation phase”, right before execution

```
const name = 'Jonas';

const first = () => {
  let a = 1;
  const b = second(7, 9);
  a = a + b;
  return a;
};

function second(x, y) {
  var c = 2;
  return c;
}

const x = first();
```

Global

```
name = ‘Jonas’
first = <function>
second = <function>
x = <unknown>
```

Literally the function code

Need to run `first()` first

first()

```
a = 1
b = <unknown>
```

Need to run `second()` first

second()

```
c = 2
arguments = [7, 9]
```

Array of passed arguments. Available in all “regular” functions (not arrow)

(Technically, values only become known during execution)

JONAS.IO
SCHMEDTMANN

THE COMPLETE JAVASCRIPT COURSE

FROM ZERO TO EXPERT!

@JONASSCHMEDTMAN

SECTION

HOW JAVASCRIPT WORKS BEHIND THE SCENES

LECTURE

SCOPE AND THE SCOPE CHAIN

JS

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';
}
```

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

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

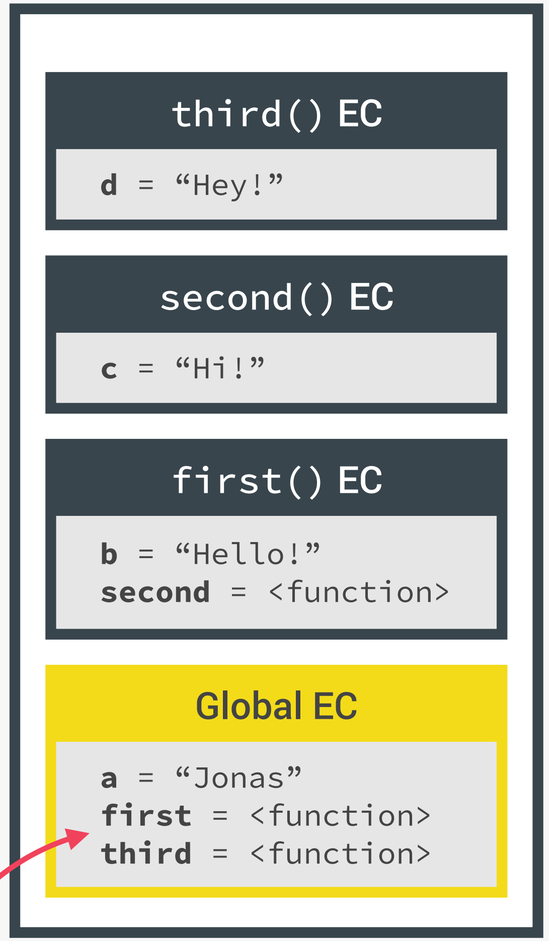
- 👉 Variables are accessible only **inside block** (block scoped)
- ⚠️ **HOWEVER**, this only applies to **`let`** and **`const`** variables!
- 👉 Functions are **also block scoped** (only in strict mode)

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
}
```

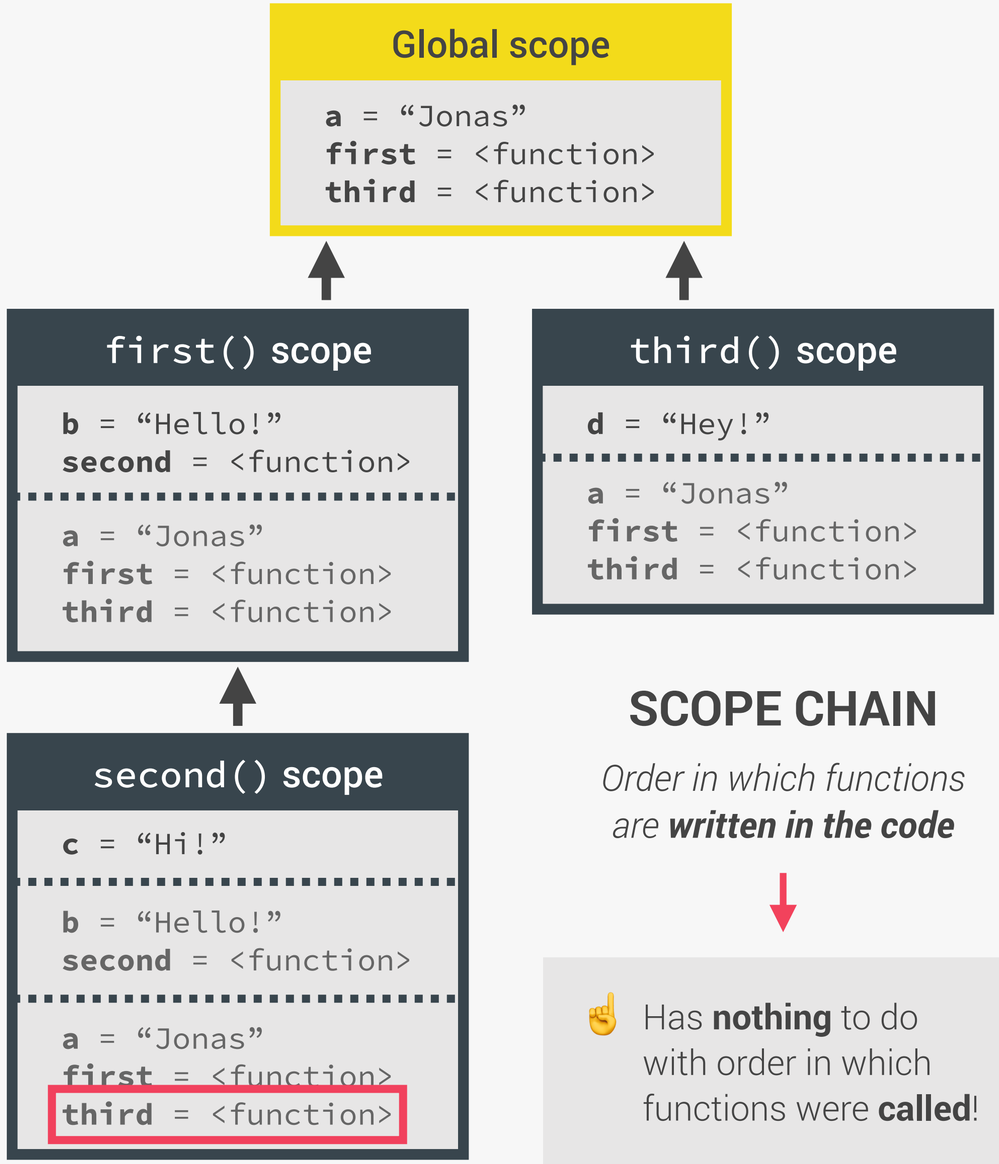
c and b can NOT be found in third() scope!

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**!

JONAS.IO
SCHMEDTMANN

THE COMPLETE JAVASCRIPT COURSE

FROM ZERO TO EXPERT!

@JONASSCHMEDTMAN

SECTION

HOW JAVASCRIPT WORKS BEHIND THE SCENES

LECTURE

VARIABLE ENVIRONMENT: HOISTING AND THE TDZ

JS

HOISTING IN JAVASCRIPT

👉 **Hoisting:** Makes some types of variables accessible/usable in the code before they are actually declared. “Variables lifted to the top of their scope”.

BEHIND THE SCENES

Before execution, code is scanned for variable declarations, and for each variable, a new property is created in the **variable environment object**.

EXECUTION CONTEXT

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

	HOISTED? 👇	INITIAL VALUE 👇	SCOPE 👇
`function` declarations	✅ YES	Actual function	Block
`var` variables	✅ YES	`undefined`	Function
`let` and `const` variables	🚫 NO	`<uninitialized>`, TDZ	Block
`function` expressions and arrows	🤷‍♂️ Depends if using `var` or `let/const`		

In strict mode. Otherwise: function!

Technically, yes. But not in practice

Temporal Dead Zone

TEMPORAL DEAD ZONE, LET AND CONST

```
const myName = 'Jonas';

if (myName === 'Jonas') {
  console.log(`Jonas is a ${job}`);
  const age = 2037 - 1989;
  console.log(age);
  const job = 'teacher';
  console.log(x);
}
```

TEMPORAL DEAD ZONE FOR `job` VARIABLE

👉 **Different kinds of error messages:**

```
ReferenceError: Cannot access 'job' before initialization
ReferenceError: x is not defined
```

WHY HOISTING?

👉 **Using functions before actual declaration;**

👉 `var` hoisting is just a byproduct.

WHY TDZ?

👉 **Makes it easier to avoid and catch errors:** accessing variables before declaration is bad practice and should be avoided;

👉 **Makes `const` variables actually work**

JONAS.IO
SCHMEDTMANN

THE COMPLETE JAVASCRIPT COURSE

FROM ZERO TO EXPERT!

@JONASSCHMEDTMAN

SECTION

HOW JAVASCRIPT WORKS BEHIND THE SCENES

LECTURE

THE THIS KEYWORD

JS

HOW THE THIS KEYWORD WORKS

👉 **`this` keyword/variable:** Special variable that is created for every execution context (every function). Takes the value of (points to) the "owner" of the function in which the `this` keyword is used.

☝️ `this` is **NOT** static. It depends on **how** the function is called, and its value is only assigned when the function **is actually called**.

EXECUTION CONTEXT

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

Method 👉 `this` = <Object that is calling the method>

Simple function call 👉 `this = undefined`

In strict mode! Otherwise: `window` (in the browser)

Don't get own `this`

Arrow functions 👉 `this` = <`this` of surrounding function (lexical `this`)>

Event listener 👉 `this` = <DOM element that the handler is attached to>

`new`, `call`, `apply`, `bind` 👉 *<Later in the course... ⏳>*

☝️ `this` does **NOT** point to the function itself, and also **NOT** the its variable environment!

👉 Method example:

```
const jonas = {
  name: 'Jonas',
  year: 1989,
  calcAge: function() {
    return 2037 - this.year;
  }
};
jonas.calcAge(); // 48
```

`calcAge` is method

`jonas`

`1989`

Way better than using `jonas.year`!

JONAS.IO
SCHMEDTMANN

THE COMPLETE JAVASCRIPT COURSE

FROM ZERO TO EXPERT!

@JONASSCHMEDTMAN

SECTION

HOW JAVASCRIPT WORKS BEHIND THE SCENES

LECTURE

MEMORY MANAGEMENT: PRIMITIVES VS. OBJECTS

JS

WHERE IS MEMORY ALLOCATED?

1️⃣ **Allocate memory**

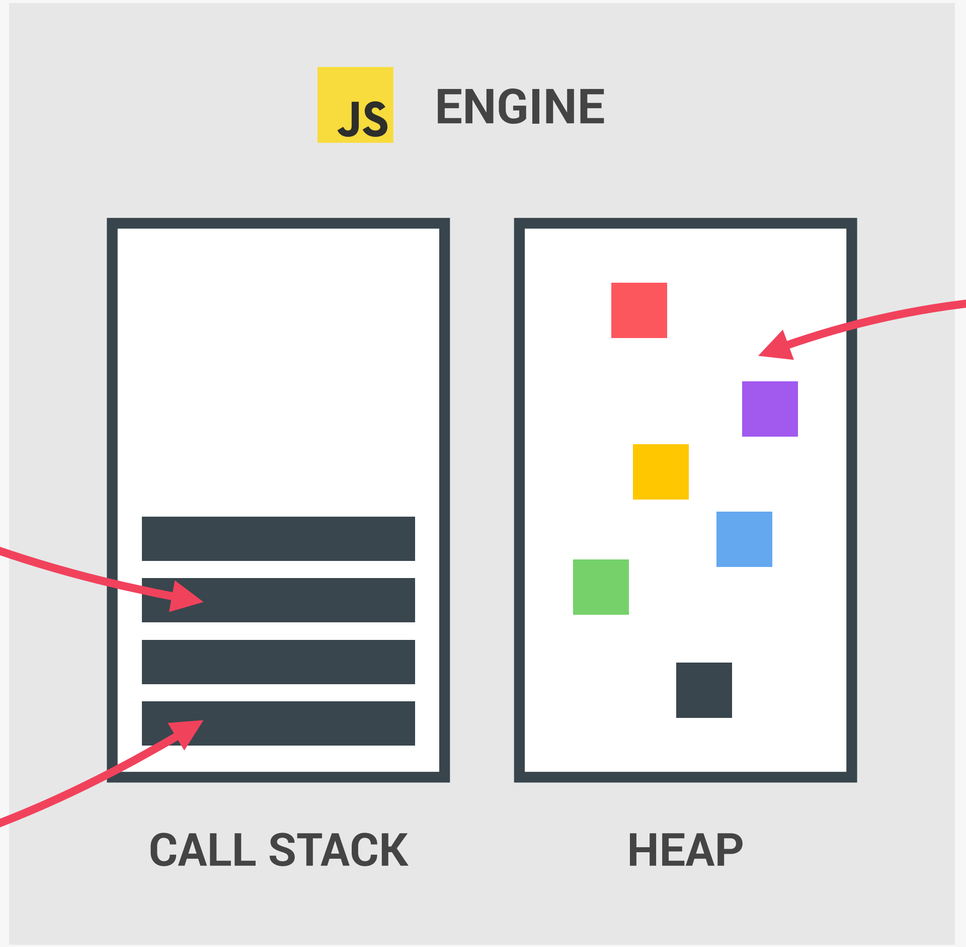
PRIMITIVES

- 👉 Number
- 👉 String
- 👉 Boolean
- 👉 Undefined
- 👉 Null
- 👉 Symbol
- 👉 BigInt

- 👉 **References to objects**

STORED IN

STORED IN



STORED IN

OBJECTS

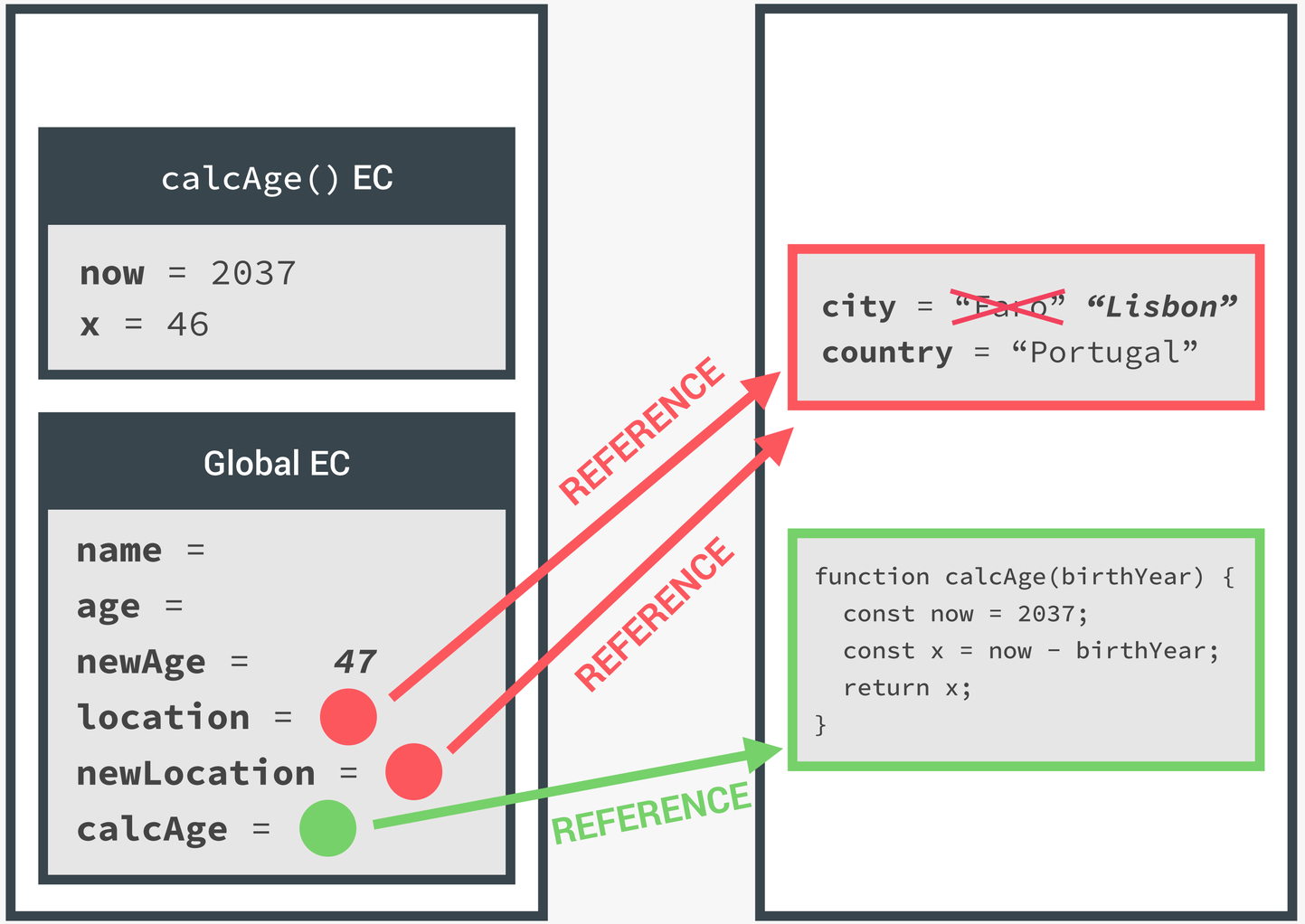
- 👉 Object literals
- 👉 Arrays
- 👉 Functions
- 👉 Many more...

UNDERSTANDING OBJECT REFERENCES

1️⃣ Allocate memory

```
const name = 'Jonas';
const age = calcAge(1991);
let newAge = age;
newAge++;

const location = {
  city: 'Faro',
  country: 'Portugal',
};

const newLocation = location;
newLocation.city = 'Lisbon';

console.log(location);
// { city: 'Lisbon', country: 'Portugal' }

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

city in location has been destroyed 💥



JONAS.IO
SCHMEDTMANN

THE COMPLETE JAVASCRIPT COURSE

FROM ZERO TO EXPERT!

@JONASSCHMEDTMAN

SECTION

HOW JAVASCRIPT WORKS BEHIND THE SCENES

LECTURE

MEMORY MANAGEMENT: GARBAGE COLLECTION

JS

GARBAGE COLLECTION

3 Release memory

🤔 **How is memory freed up after we no longer need a value?**

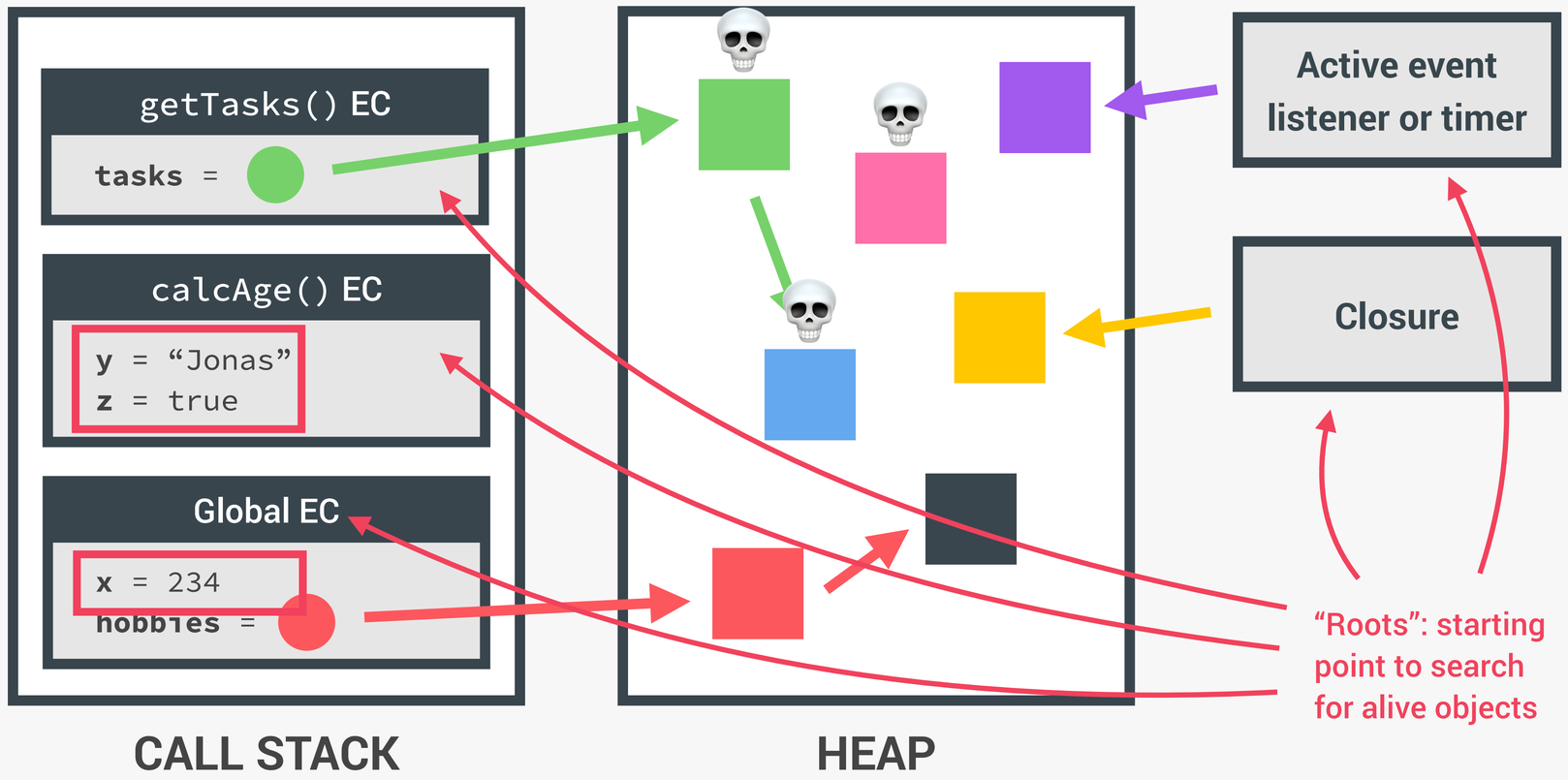
→ **CALL STACK** 💥 Variable environment is **simply deleted when EC pops off stack**

→ **HEAP** 🚛 **Garbage collection** (central memory management tool)

MARK-AND-SWEEP ALGORITHM:

1. **Mark:** Mark all objects that are ***reachable*** from a root as “alive”
2. **Sweep:** Delete un-marked (***unreachable***) objects and reclaim memory for future allocations

👉 **Memory leak:** When objects that are no longer needed are **incorrectly still reachable**, and therefore **not being garbage collected**



JONAS.IO
SCHMEDTMANN

THE COMPLETE JAVASCRIPT COURSE

FROM ZERO TO EXPERT!

@JONASSCHMEDTMAN

SECTION

DATA STRUCTURES, MODERN OPERATORS AND STRINGS

LECTURE

SUMMARY: WHICH DATA STRUCTURE TO USE?

JS

ARRAYS VS. SETS AND OBJECTS VS. MAPS

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

VS.

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

```
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)

VS.

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

JONAS.IO
SCHMEDTMANN

THE COMPLETE JAVASCRIPT COURSE

FROM ZERO TO EXPERT!

@JONASSCHMEDTMAN

SECTION

A CLOSER LOOK AT FUNCTIONS

LECTURE

FIRST-CLASS AND HIGHER-ORDER FUNCTIONS

JS

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

JONAS.IO
SCHMEDTMANN

THE COMPLETE JAVASCRIPT COURSE

FROM ZERO TO EXPERT!

@JONASSCHMEDTMAN

SECTION

A CLOSER LOOK AT FUNCTIONS

LECTURE

CLOSURES

JS

UNDERSTANDING CLOSURES

secureBooking() EC

`passengerCount = 0`

- 👉 Variable environment (VE) that popped off stack after `secureBooking`
- 👉 Because of the closure, VE was **moved to heap and NOT garbage collected**
- 👉 **This execution context is where `booker` function was created!**

booker() EC

`<empty>`

Global EC

`secureBooking = <f>`
`booker = <f>`

CALL STACK

```
const secureBooking = function () {
  let passengerCount = 0;

  return function () {
    passengerCount++;
    console.log(`${passengerCount}
    passengers`);
  };
};

const booker = secureBooking();

booker(); // 1 passengers
booker(); // 2 passengers
```

This is the function

Global scope

`secureBooking = <f>`
`booker = <f>`

secureBooking() scope

`passengerCount = 0`

`secureBooking = <f>`
`booker = <f>`

booker() scope

`<empty>`

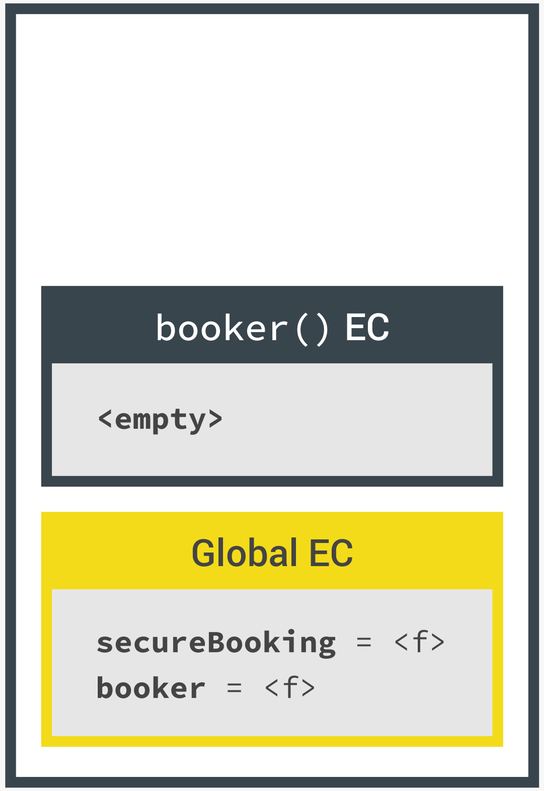
`secureBooking = <f>`
`booker = <f>`

How to access passengerCount?

SCOPE CHAIN

UNDERSTANDING CLOSURES

- 👉 A function has access to the variable environment (VE) of the execution context in which it was created
- 👉 **Closure:** VE attached to the function, exactly as it was at the time and place the function was created



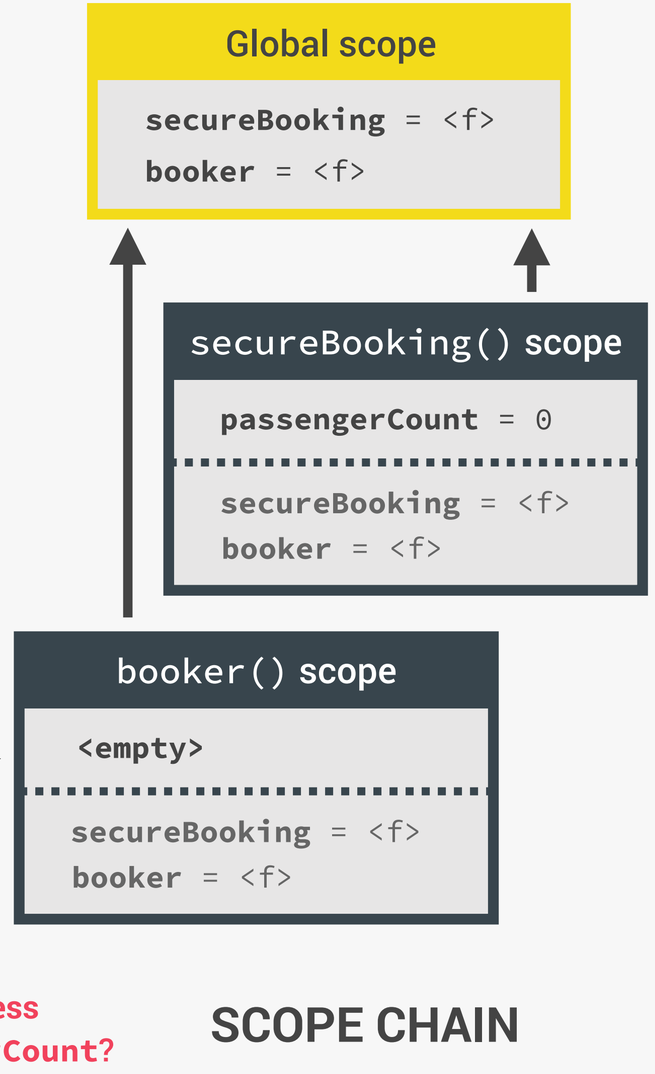
CALL STACK

```
const secureBooking = function () {
  let passengerCount = 0;

  return function () {
    passengerCount++;
    console.log(`${passengerCount}
    passengers`);
  };
};

const booker = secureBooking();

booker(); // 1 passengers
booker(); // 2 passengers
```

This is the function

passengerCount = 2

CLOSURE



How to access passengerCount?

SCOPE CHAIN

CLOSURES SUMMARY 🥳

👉 A closure is the closed-over **variable environment** of the execution context **in which a function was created**, even ***after*** that execution context is gone;

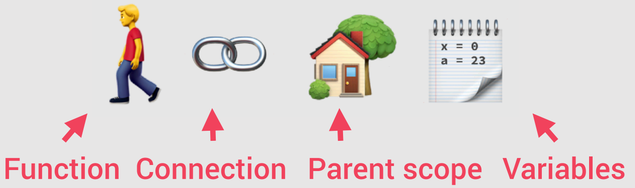
↓ Less formal

👉 A closure gives a function access to all the variables **of its parent function**, even ***after*** that parent function has returned. The function keeps a **reference** to its outer scope, which ***preserves*** the scope chain throughout time.

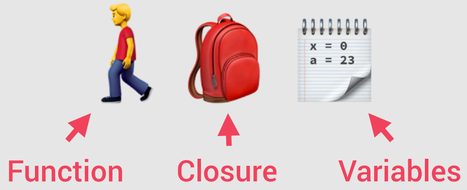
↓ Less formal

👉 A closure makes sure that a function doesn't loose connection to **variables that existed at the function's birth place**;

↓ Less formal



👉 A closure is like a **backpack** that a function carries around wherever it goes. This backpack has all the **variables that were present in the environment where the function was created**.



☝️ We do **NOT** have to manually create closures, this is a JavaScript feature that happens automatically. We can't even access closed-over variables explicitly. A closure is **NOT** a tangible JavaScript object.

JONAS.IO
SCHMEDTMANN

THE COMPLETE JAVASCRIPT COURSE

FROM ZERO TO EXPERT!

@JONASSCHMEDTMAN

SECTION

WORKING WITH ARRAYS

LECTURE

SUMMARY: WHICH ARRAY METHOD TO USE?

JS

WHICH ARRAY METHOD TO USE? 🤔

"I WANT...:"

To *mutate* original

👉 Add to original:

- `.push` *(end)*
- `.unshift` *(start)*

👉 Remove from original:

- `.pop` *(end)*
- `.shift` *(start)*
- `.splice` *(any)*

👉 Others:

- `.reverse`
- `.sort`
- `.fill`

☝️ **These should usually be avoided!**

A *new* array based on original

👉 Same length as original:

- `.map` *(loop)*

👉 Filtered using condition:

- `.filter`

👉 Taking portion of original:

- `.slice`

👉 With one item replaced:

- `.with`

👉 Flattened:

- `.flat`
- `.flatMap`

👉 Reversed:

- `.toReversed`

👉 Sorted:

- `.toSorted`

👉 With deleted items:

- `.toSpliced`

👉 Joining two arrays:

- `.concat`

An array index

👉 Based on value:

- `.indexOf`

👉 Based on test condition:

- `.findIndex`
- `.findLastIndex`

An array element

👉 Based on test condition:

- `.find`
- `.findLast`

👉 Based on position:

- `.at`

Know if array includes

👉 Based on value:

- `.includes`

👉 Based on test condition:

- `.some`
- `.every`

A new string

👉 Based on separator:

- `.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)

MORE ARRAY TOOLS AND TECHNIQUES

👉 **Grouping** an array by categories:

```
Object.groupBy
```

👉 Creating a new array **from scratch**:

```
Array.from
```

👉 Creating a new array **from scratch** with n empty positions (use together with `.fill` method):

```
new Array(n)
```

👉 **Joining** 2 or more arrays:

```
[...arr1, ...arr2]
```

👉 Creating a new array containing **unique** values from `arr`

```
[...new Set(arr)]
```

👉 Creating a new array containing unique elements that are present **in both** `arr1` and `arr2`

```
[...new Set(arr1).intersection(new Set(arr2))]
```

JONAS.IO
SCHMEDTMANN

THE COMPLETE JAVASCRIPT COURSE

FROM ZERO TO EXPERT!

@JONASSCHMEDTMAN

SECTION

ADVANCED DOM AND EVENTS

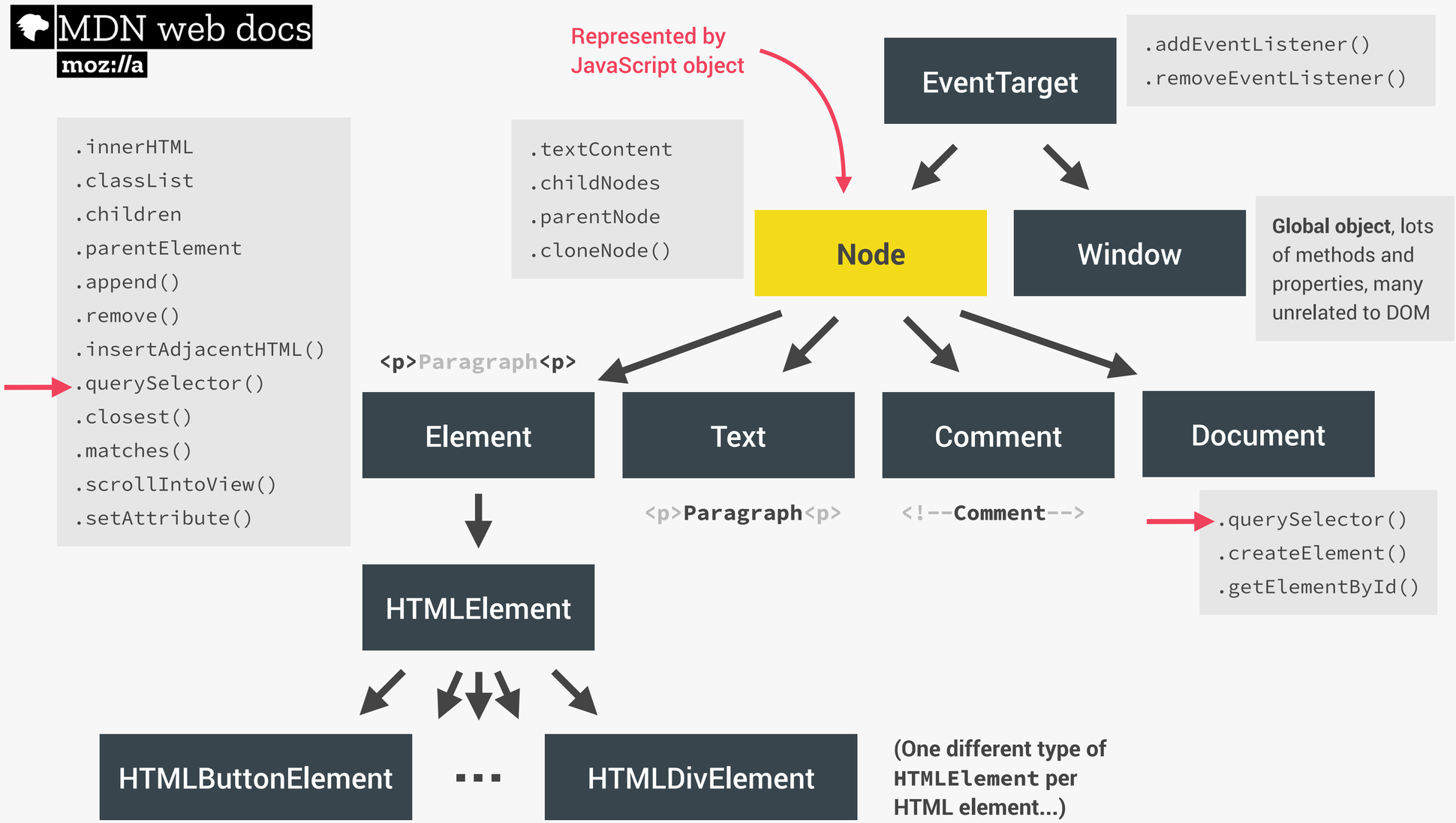
LECTURE

HOW THE DOM REALLY WORKS

JS

HOW THE DOM API IS ORGANIZED BEHIND THE SCENES

MDN web docs
moz://a



INHERITANCE OF METHODS AND PROPERTIES

Example:

Any `HTMLElement` will have access to `.addEventListener()`, `.cloneNode()` or `.closest()` methods.

(THIS IS NOT A DOM TREE)

JONAS.IO
SCHMEDTMANN

THE COMPLETE JAVASCRIPT COURSE

FROM ZERO TO EXPERT!

@JONASSCHMEDTMAN

SECTION

ADVANCED DOM AND EVENTS

LECTURE

EVENT PROPAGATION: BUBBLING AND CAPTURING

JS

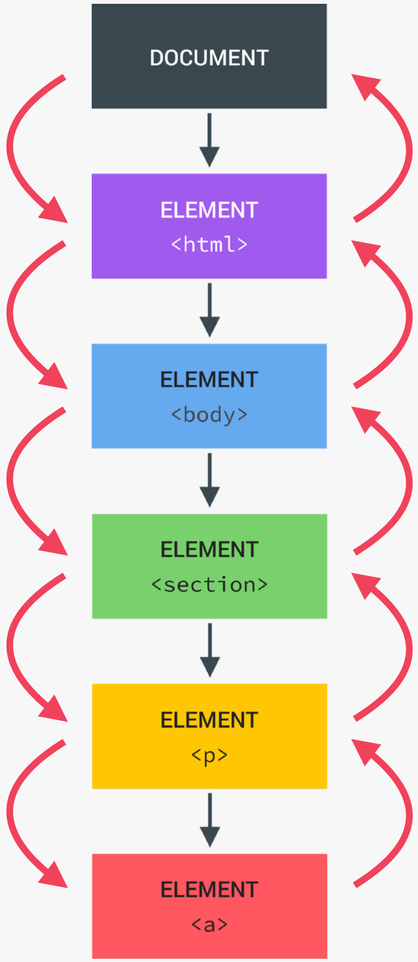
BUBBLING AND CAPTURING

```
<html>
  <head>
    <title>A Simple Page</title>
  </head>
  <body>
    <section>
      <p>A paragraph with a <a>link</a></p>
      <p>A second paragraph</p>
    </section>
    <section>
      <img src="dom.png" alt="The DOM" />
    </section>
  </body>
</html>
```

(THIS DOES NOT HAPPEN ON *ALL* EVENTS)

Click event

1 CAPTURING PHASE



2 TARGET PHASE

3 BUBBLING PHASE

```
document
  .querySelector('section')
  .addEventListener('click', () => {
  alert('You cliked me 😀');
});
```

127.0.0.1:8080 says

You cliked me 😀

```
document
  .querySelector('a')
  .addEventListener('click', () => {
  alert('You cliked me 😀');
});
```

127.0.0.1:8080 says

You cliked me 😀

JONAS.IO
SCHMEDTMANN

THE COMPLETE JAVASCRIPT COURSE

FROM ZERO TO EXPERT!

@JONASSCHMEDTMAN

SECTION

ADVANCED DOM AND EVENTS

LECTURE

EFFICIENT SCRIPT LOADING: DEFER AND ASYNC

JS

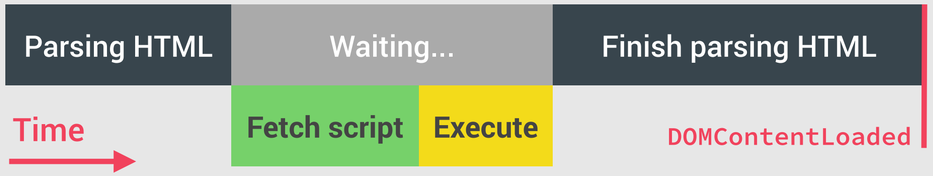
DEFER AND ASYNC SCRIPT LOADING

HEAD

BODY END

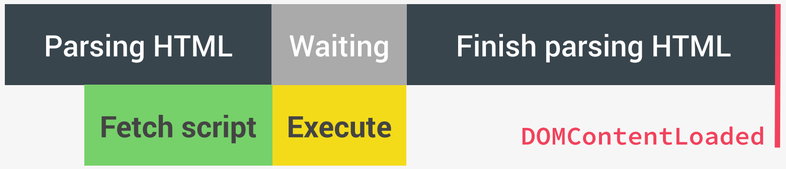
REGULAR

```
<script src="script.js">
```



ASYNC

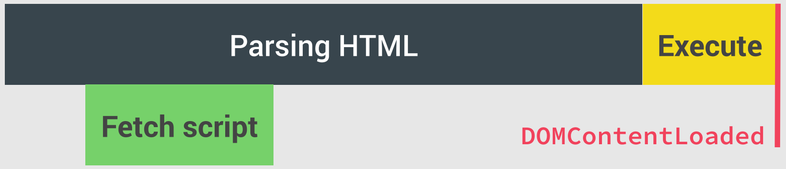
```
<script async src="script.js">
```



👉 Makes no sense 🤷

DEFER

```
<script defer src="script.js">
```



👉 Makes no sense 🤷

REGULAR VS. ASYNC VS. DEFER

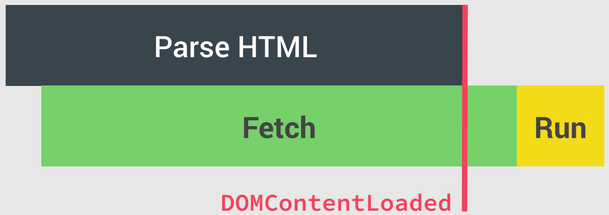
END OF BODY

- 👉 Scripts are fetched and executed ***after the HTML is completely parsed***
- 👉 **Use if you need to support old browsers**

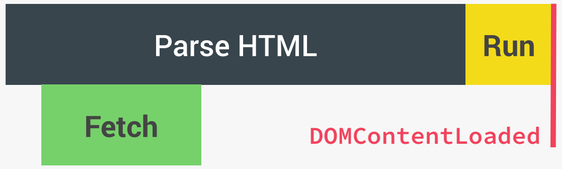
You can, of course, use **different strategies for different scripts**. Usually a complete web applications includes more than just one script

ASYNC IN HEAD

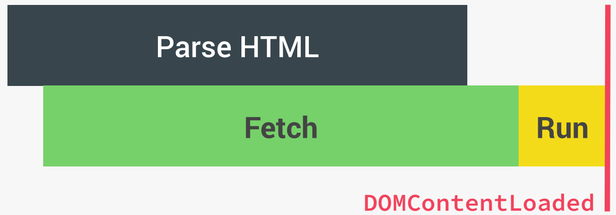
- 👉 Scripts are fetched ***asynchronously*** and executed ***immediately***
- 👉 Usually the `DOMContentLoaded` event waits for ***all*** scripts to execute, except for `async` scripts. So, `DOMContentLoaded` does ***not*** wait for an `async` script
- 👉 Scripts ***not*** guaranteed to execute in order
- 👉 **Use for 3rd-party scripts where order doesn't matter (e.g. Google Analytics)**



DEFER IN HEAD



- 👉 Scripts are fetched ***asynchronously*** and executed ***after the HTML is completely parsed***
- 👉 `DOMContentLoaded` event fires ***after*** `defer` script is executed
- 👉 Scripts are executed ***in order***
- 👉 **This is overall the best solution! Use for your own scripts, and when order matters (e.g. including a library)**



JONAS.IO
SCHMEDTMANN

THE COMPLETE JAVASCRIPT COURSE

FROM ZERO TO EXPERT!

SECTION

OBJECT ORIENTED PROGRAMMING (OOP) WITH JAVASCRIPT

LECTURE

OOP IN JAVASCRIPT

@JONASSCHMEDTMAN

JS

OOP IN JAVASCRIPT: PROTOTYPES

"CLASSICAL OOP": CLASSES

Class

INSTANTIATION

Instance

- Objects (instances) are **instantiated** from a class, which functions like a blueprint;
- Behavior (methods) is **copied** from class to all instances.

OOP IN JS: PROTOTYPES

Prototype ← Contains methods

PROTOTYPAL INHERITANCE/ DELEGATION

Object ← Can access methods

- Objects are **linked** to a prototype object;
- **Prototypal inheritance:** The prototype contains methods (behavior) that are **accessible to all objects linked to that prototype;**
- Behavior is **delegated** to the linked prototype object.

👉 **Example: Array**

```
const num = [1, 2, 3];
num.map(v => v * 2);
```

MDN web docs
moz://a

```
Array.prototype.keys()
Array.prototype.lastIndexOf()
Array.prototype.map()
```

`Array.prototype` is the **prototype** of all array objects we create in JavaScript

Therefore, **all** arrays have access to the `map` method!

```
ƒ Array()
  arguments: (...)
  caller: (...)
  length: 1
  name: "Array"
  prototype: Array(0)
    unique: ƒ ()
    length: 0
    constructor: ƒ Array()
    concat: ƒ concat()
    map: ƒ map()
```

3 WAYS OF IMPLEMENTING PROTOTYPAL INHERITANCE IN JAVASCRIPT

🤔 *“How do we actually create prototypes? And how do we link objects to prototypes? How can we create new objects, without having classes?”*

☝️ **The 4 pillars of OOP are still valid!**

- 👉 Abstraction
- 👉 Encapsulation
- 👉 Inheritance
- 👉 Polymorphism

1 Constructor functions

- 👉 Technique to create objects from a function;
- 👉 This is how built-in objects like Arrays, Maps or Sets are actually implemented.

2 ES6 Classes

- 👉 Modern alternative to constructor function syntax;
- 👉 “Syntactic sugar”: behind the scenes, ES6 classes work **exactly** like constructor functions;
- 👉 ES6 classes do **NOT** behave like classes in “classical OOP” (last lecture).

3 `Object.create()`

- 👉 The easiest and most straightforward way of linking an object to a prototype object.

JONAS.IO
SCHMEDTMANN

THE COMPLETE JAVASCRIPT COURSE

FROM ZERO TO EXPERT!

@JONASSCHMEDTMAN

SECTION

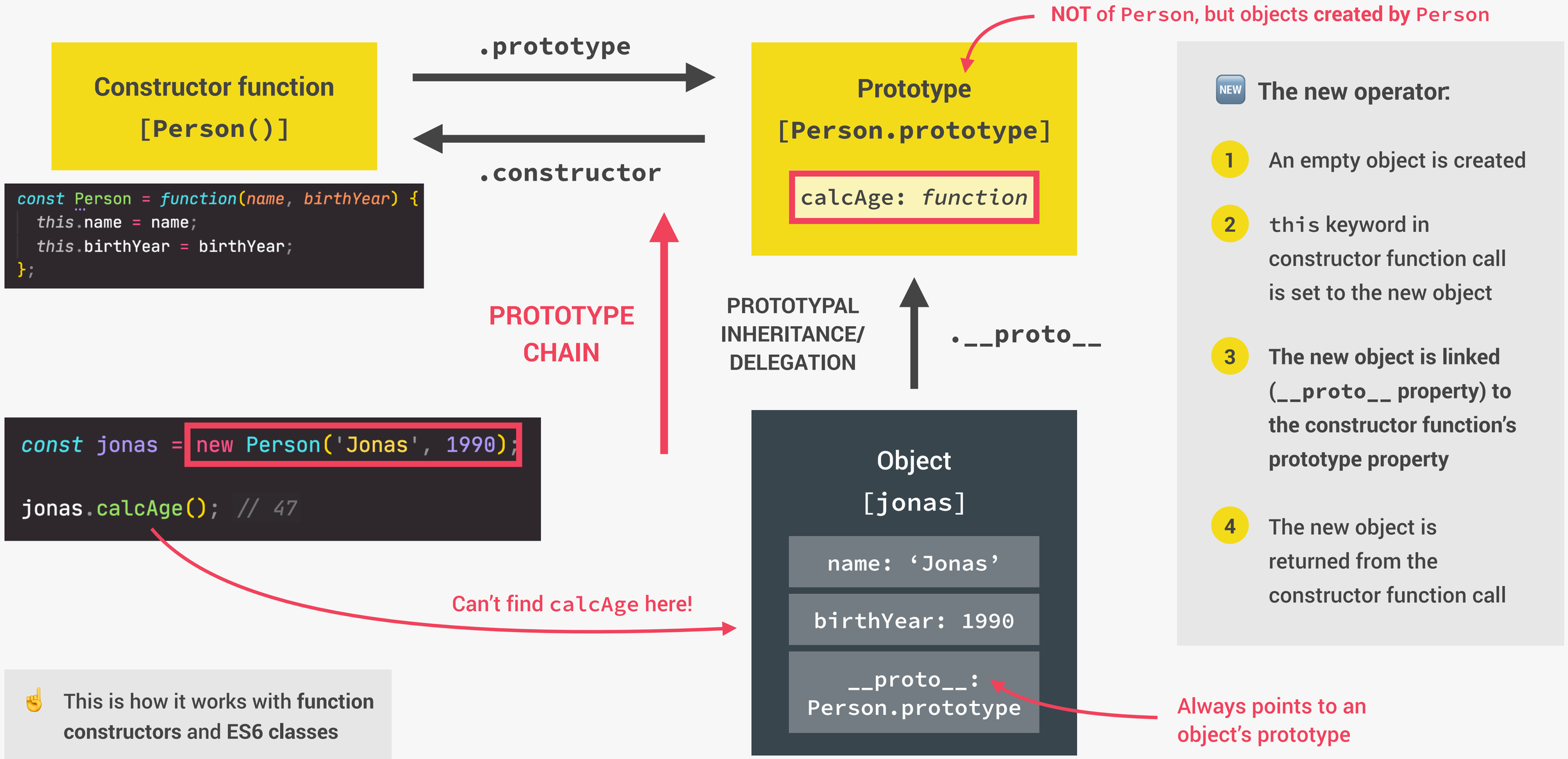
OBJECT ORIENTED PROGRAMMING (OOP) WITH JAVASCRIPT

LECTURE

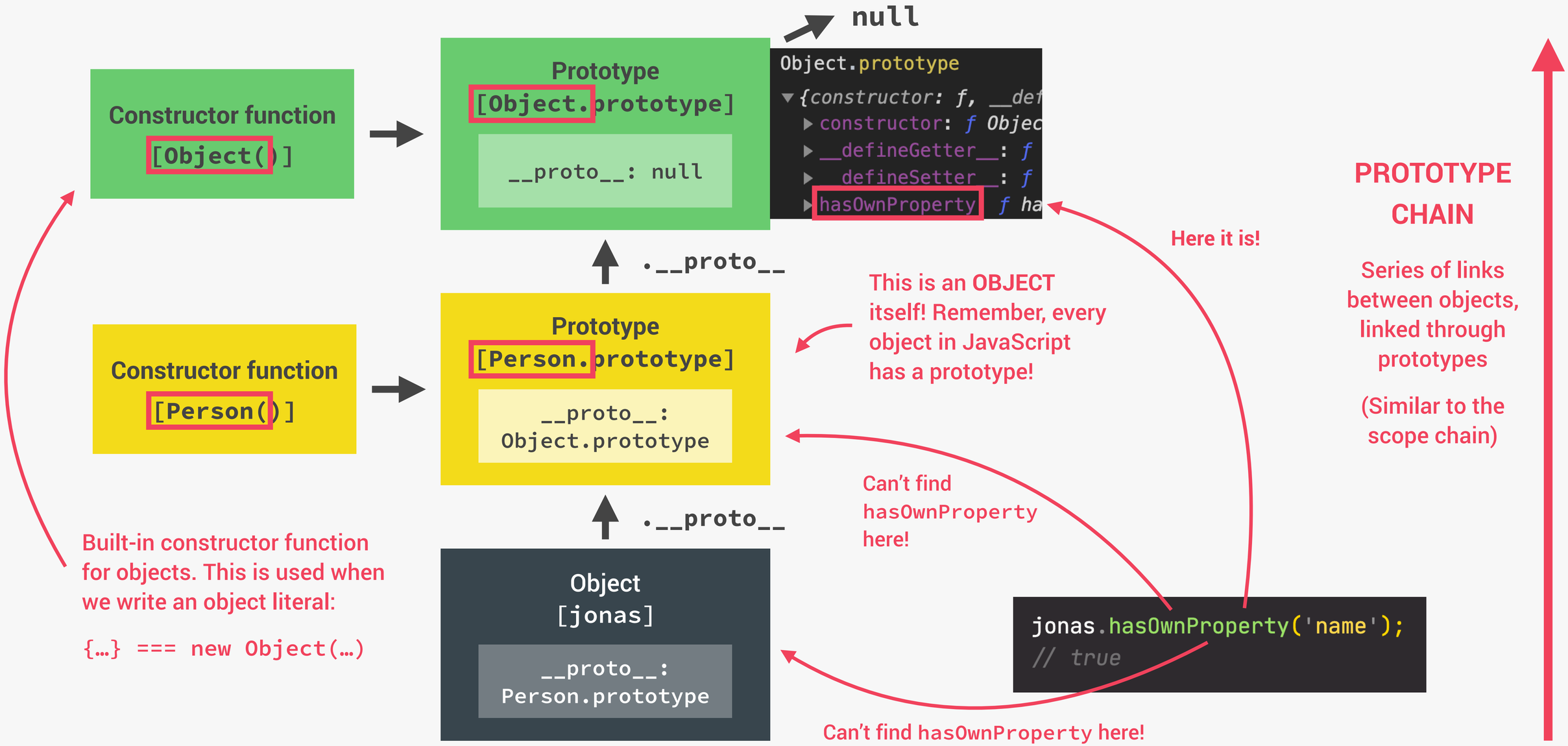
PROTOTYPAL INHERITANCE AND THE PROTOTYPE CHAIN

JS

HOW PROTOTYPAL INHERITANCE / DELEGATION WORKS



THE PROTOTYPE CHAIN



JONAS.IO
SCHMEDTMANN

THE COMPLETE JAVASCRIPT COURSE

FROM ZERO TO EXPERT!

@JONASSCHMEDTMAN

SECTION

OBJECT ORIENTED PROGRAMMING (OOP) WITH JAVASCRIPT

LECTURE

OBJECT.CREATE

JS

HOW OBJECT.CREATE WORKS

Prototype

[PersonProto]

calcAge: *function*

```
const PersonProto = {
  calcAge() {
    console.log(2037 - this.birthYear);
  },
};
```

.__proto__

```
const steven = Object.create(PersonProto);
```

Object

[steven]

name: ‘Steven’

birthYear: 2002

__proto__ : PersonProto

OBJECT.CREATE

CONSTRUCTOR FUNCTIONS

~~**Constructor function**~~

~~**[Person()]**~~

~~.prototype~~

Prototype

[Person.prototype]

calcAge: *function*

```
const jonas = new Person('Jonas', 1990);
```

AUTOMATIC

.__proto__

Object

[jonas]

name: ‘Jonas’

birthYear: 1990

__proto__: Person.prototype

JONAS.IO
SCHMEDTMANN

THE COMPLETE JAVASCRIPT COURSE

FROM ZERO TO EXPERT!

@JONASSCHMEDTMAN

SECTION

OBJECT ORIENTED PROGRAMMING (OOP) WITH JAVASCRIPT

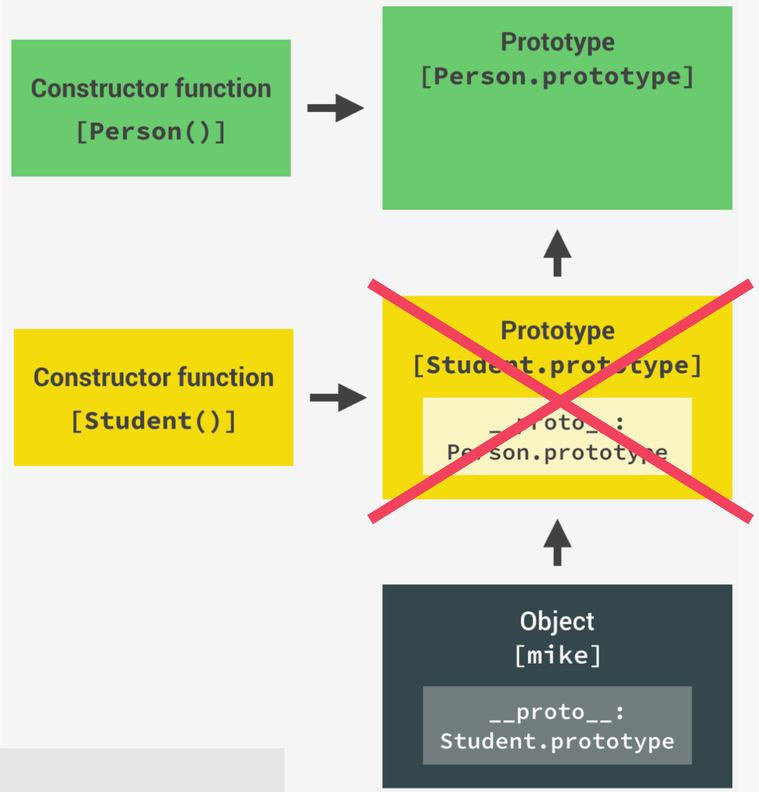
LECTURE

INHERITANCE BETWEEN "CLASSES": CONSTRUCTOR FUNCTIONS

JS

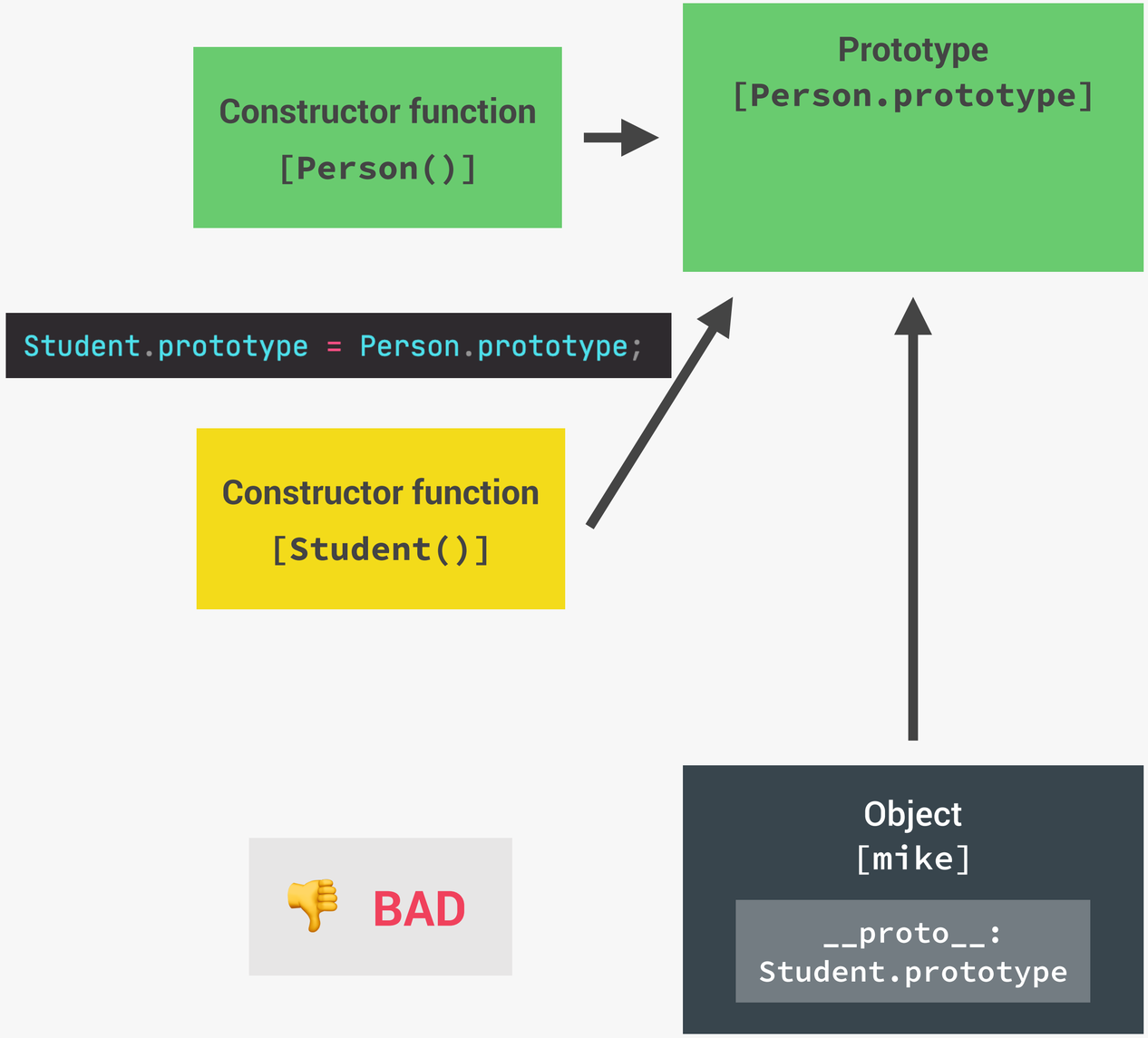
INHERITANCE BETWEEN "CLASSES"

```
Student.prototype = Object.create(Person.prototype);
```



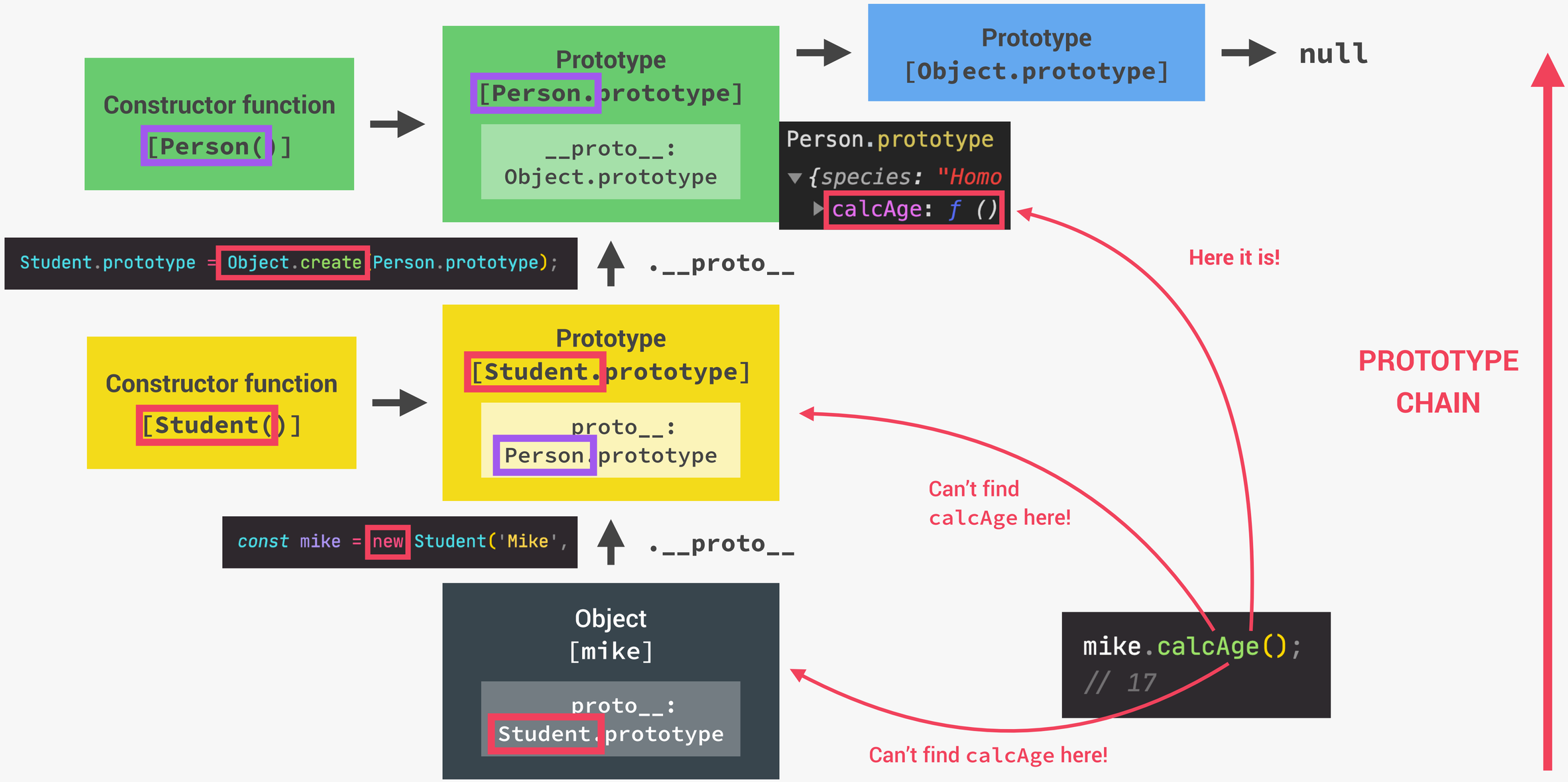
👍 GOOD

```
Student.prototype = Person.prototype;
```



👎 BAD

INHERITANCE BETWEEN "CLASSES"



JONAS.IO
SCHMEDTMANN

THE COMPLETE JAVASCRIPT COURSE

FROM ZERO TO EXPERT!

@JONASSCHMEDTMAN

SECTION

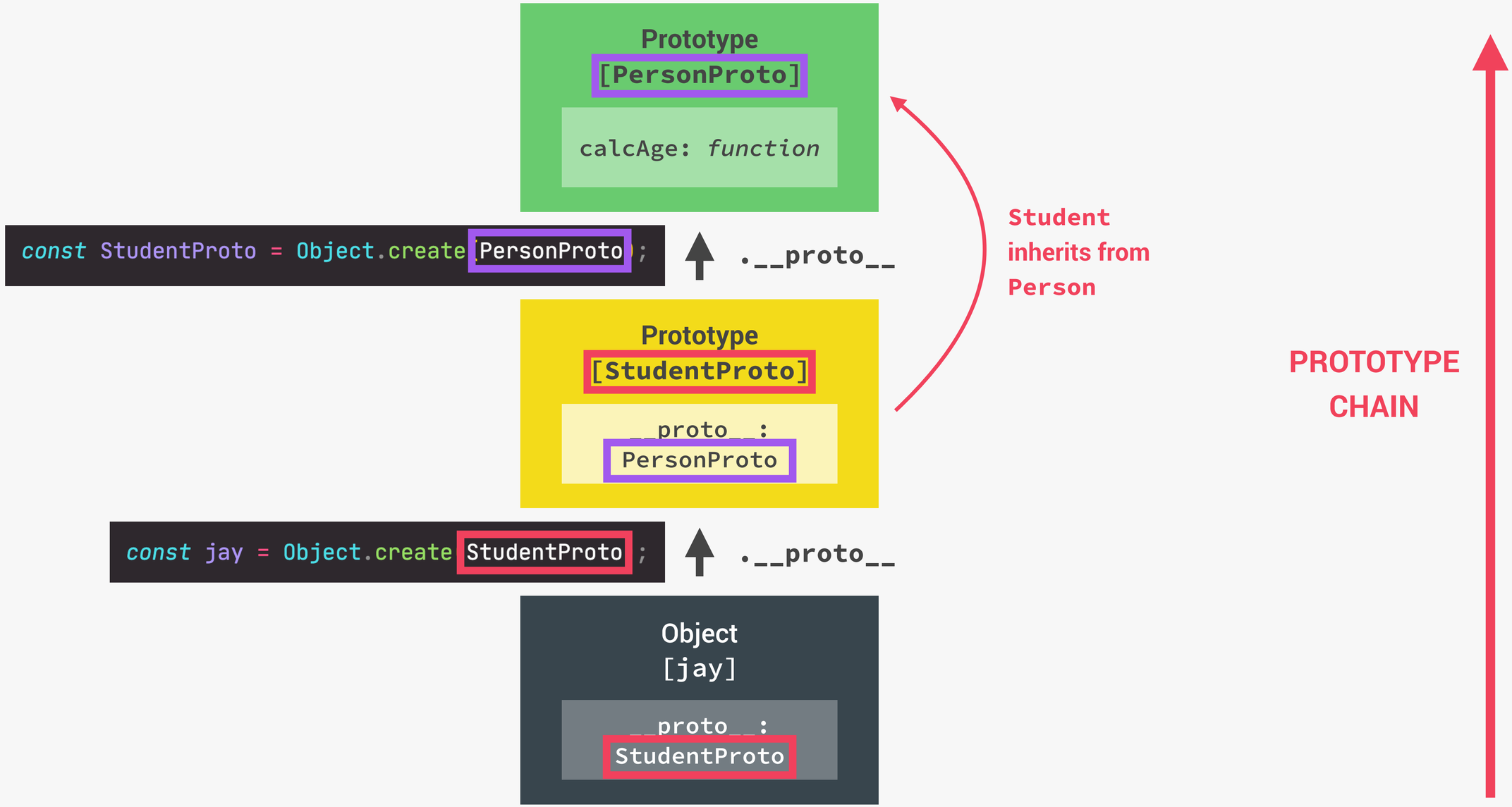
OBJECT ORIENTED PROGRAMMING (OOP) WITH JAVASCRIPT

LECTURE

INHERITANCE BETWEEN "CLASSES": OBJECT.CREATE

JS

INHERITANCE BETWEEN "CLASSES": OBJECT.CREATE



JONAS.IO
SCHMEDTMANN

THE COMPLETE JAVASCRIPT COURSE

FROM ZERO TO EXPERT!

@JONASSCHMEDTMAN

SECTION

OBJECT ORIENTED PROGRAMMING (OOP) WITH JAVASCRIPT

LECTURE

ES6 CLASSES SUMMARY

JS


Public field (similar to property, available on created object)
Private fields (not accessible outside of class)
Static public field (available only on class)
Call to parent (super) class (necessary with extend). Needs to happen before accessing this
Instance property (available on created object)
Redefining private field
Public method
Referencing private field and method
Private method (⚠️ Might not yet work in your browser. “Fake” alternative: _ instead of #)
Getter method
Setter method (use _ to set property with same name as method, and also add getter)
Static method (available only on class. Can not access instance properties nor methods, only static ones)
Creating new object with new operator
class Student extends Person {
university = 'University of Lisbon';
#studyHours = 0;
#course;
static numSubjects = 10;
constructor(fullName, birthYear, startYear, course) {
super(fullName, birthYear);
this.startYear = startYear;
this.#course = course;
}
introduce() {
console.log(`I study ${this.#course} at ${this.university}`);
}
study(h) {
this.#makeCoffe();
this.#studyHours += h;
}
#makeCoffe() {
return 'Here is a coffe for you 👼';
}
get testScore() {
return this._testScore;
}
set testScore(score) {
this._testScore = score ⩽ 20 ? score : 0;
}
static printCurriculum() {
console.log(`There are ${this.numSubjects} subjects`);
}
}
const student = new Student('Jonas', 2020, 2037, 'Medicine');
Parent class
Inheritance between classes, automatically sets prototype
Child class
Constructor method, called by new operator. Mandatory in regular class, might be omitted in a child class
👉 Classes are just “syntactic sugar” over constructor functions
👉 Classes are not hoisted
👉 Classes are first-class citizens
👉 Class body is always executed in strict mode

JONAS.IO
SCHMEDTMANN

THE COMPLETE JAVASCRIPT COURSE

FROM ZERO TO EXPERT!

@JONASSCHMEDTMAN

SECTION

MAPTY APP: OOP, GEOLOCATION, EXTERNAL LIBRARIES, AND MORE!

LECTURE

HOW TO PLAN A WEB PROJECT

JS

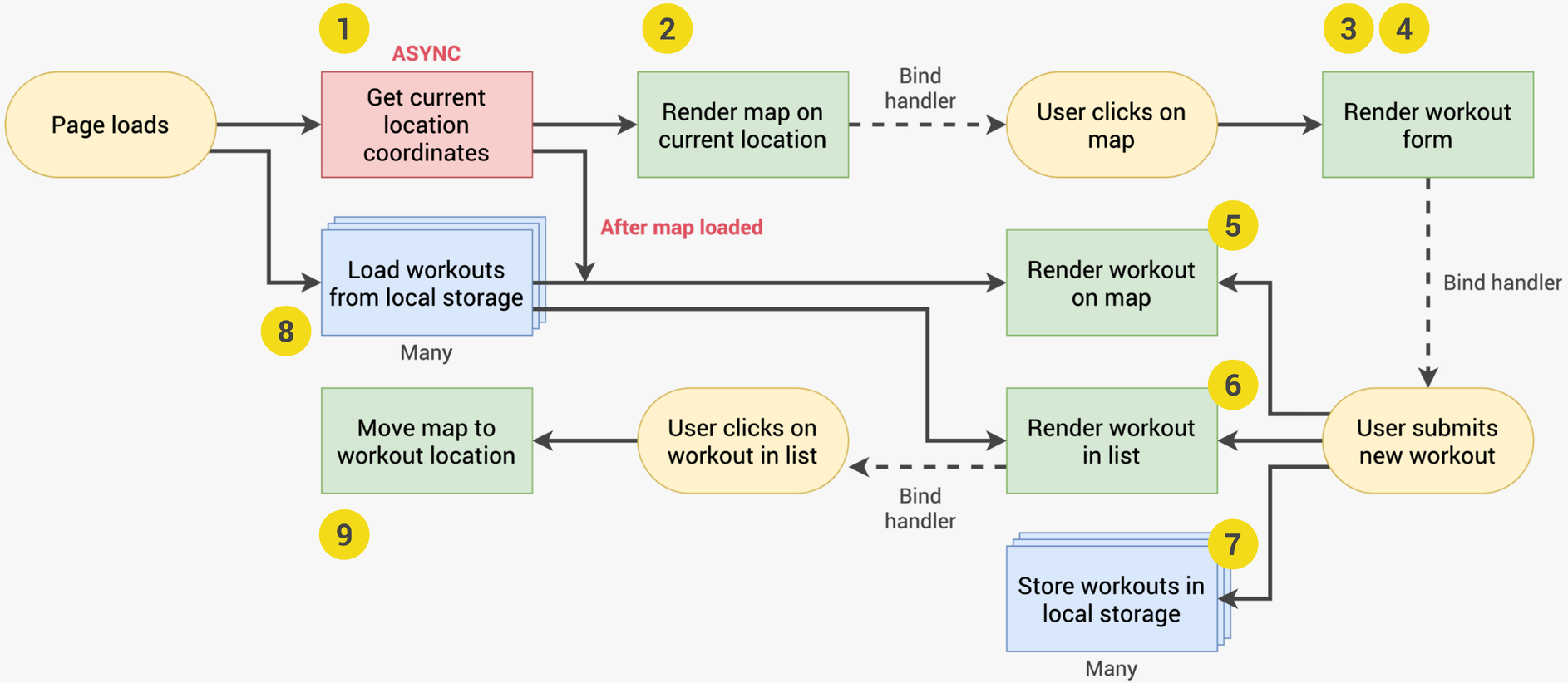
3. FLOWCHART

mapty

FEATURES

1. Geolocation to display map at current location
2. Map where user clicks to add new workout
3. Form to input distance, time, pace, steps/minute
4. Form to input distance, time, speed, elevation gain
5. Display workouts in a list
6. Display workouts on the map
7. Store workout data in the browser
8. On page load, read the saved data and display
9. Move map to workout location on click

Added later

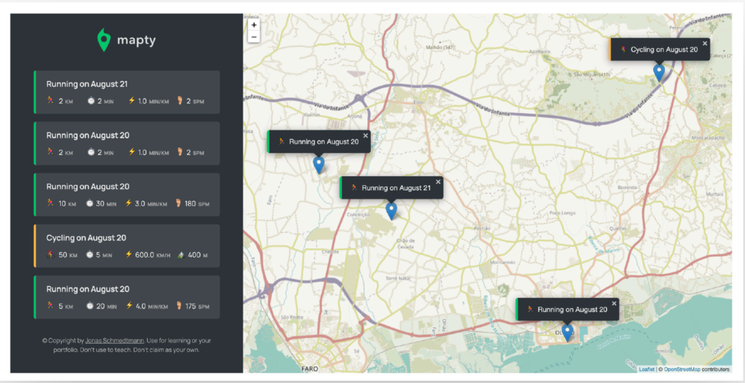


☝ In the real-world, you don't have to come with the final flowchart right in the planning phase. It's normal that it changes throughout implementation!

10 ADDITIONAL FEATURE IDEAS: CHALLENGES 🤓

mapty

- 👉 Ability to **edit** a workout;
- 👉 Ability to **delete** a workout;
- 👉 Ability to **delete all** workouts;
- 👉 Ability to **sort** workouts by a certain field (e.g. distance);
- 👉 **Re-build** `Running` and `Cycling` objects coming from Local Storage;
- 👉 More realistic error and confirmation **messages**;
- 👉 Ability to position the map to **show all workouts** [very hard];
- 👉 Ability to **draw lines and shapes** instead of just points [very hard];
- 👉 **Geocode location** from coordinates (“Run in Faro, Portugal”) [only after asynchronous JavaScript section];
- 👉 **Display weather** data for workout time and place [only after asynchronous JavaScript section].



JONAS.IO
SCHMEDTMANN

THE COMPLETE JAVASCRIPT COURSE

FROM ZERO TO EXPERT!

@JONASSCHMEDTMAN

SECTION

ASYNCHRONOUS JAVASCRIPT: PROMISES, ASYNC/AWAIT AND AJAX

LECTURE

ASYNCHRONOUS JAVASCRIPT, AJAX AND APIS

JS

ASYNCHRONOUS CODE

Asynchronous

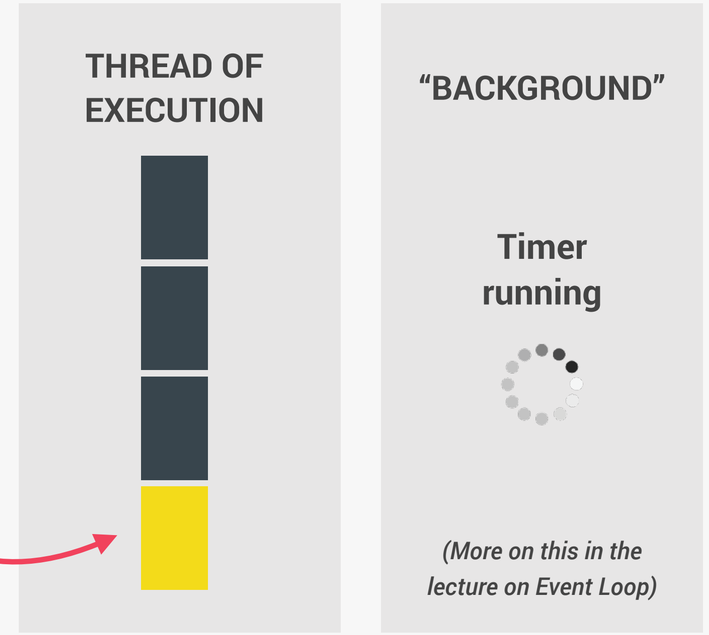
```
const p = document.querySelector('.p');
setTimeout(function () {
  p.textContent = 'My name is Jonas!';
}, 5000);
p.style.color = 'red';
```

CALLBACK WILL RUN AFTER TIMER

👉 **Example:** Timer with callback

```
[1, 2, 3].map(v => v * 2);
```

Callback does NOT automatically make code asynchronous!



ASYNCHRONOUS

Coordinating behavior of a program over a period of time

- 👉 Asynchronous code is executed **after a task that runs in the "background" finishes**;
- 👍 Asynchronous code is **non-blocking**;
- 👉 Execution doesn't wait for an asynchronous task to finish its work;
- 👉 Callback functions alone do **NOT** make code asynchronous!

ASYNCHRONOUS CODE

Asynchronous

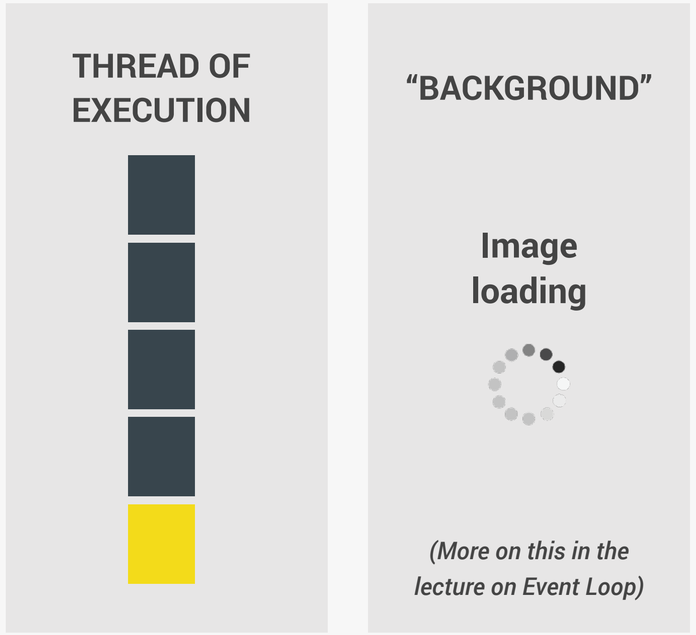
```
const img = document.querySelector('.dog');
img.src = 'dog.jpg';
img.addEventListener('load', function () {
  img.classList.add('fadeIn');
});
p.style.width = '300px';
```

CALLBACK WILL RUN AFTER IMAGE LOADS

`addEventListener` does NOT automatically make code asynchronous!

👉 **Example:** Asynchronous image loading with event and callback

👉 **Other examples:** Geolocation API or **AJAX calls**



ASYNCHRONOUS

Coordinating behavior of a program over a period of time

👉 Asynchronous code is executed **after a task that runs in the "background" finishes**;

👍 Asynchronous code is **non-blocking**;

👉 Execution doesn't wait for an asynchronous task to finish its work;

👉 Callback functions alone do **NOT** make code asynchronous!

WHAT ARE AJAX CALLS?

AJAX

Asynchronous **J**avaScript **A**nd **X**ML: Allows us to communicate with remote web servers in an **asynchronous way**. With AJAX calls, we can **request data** from web servers dynamically.

WHAT IS AN API?

API

- **A**pplication **P**rogramming **I**nterface: Piece of software that can be used by another piece of software, in order to allow **applications to talk to each other**;
- There are be many types of APIs in web development:

  DOM API | Geolocation API | Own Class API | **"Online" API**

  Just "API"
- **"Online" API:** Application running on a server, that receives requests for data, and sends data back as response;
- We can build **our own** web APIs (requires back-end development, e.g. with node.js) or use **3rd-party** APIs.

There is an API for everything

- Weather data
- Data about countries
- Flights data
- Currency conversion data
- APIs for sending email or SMS
- Google Maps
- *Millions of possibilities...*

✈️ 🌍

AJAX

XML

~~XML data format~~

JSON data format


```
{
  "publisher": "101 Cookbooks",
  "title": "Best Pizza Dough Ever",
  "source_url": "http://www.101cookbo
  "recipe_id": "47746",
  "image_url": "http://forkify-api.he
  "social_rank": 100,
  "publisher_url": "http://www.101coo
},
```


Most popular API data format

JONAS.IO
SCHMEDTMANN

THE COMPLETE JAVASCRIPT COURSE

FROM ZERO TO EXPERT!

@JONASSCHMEDTMAN

SECTION

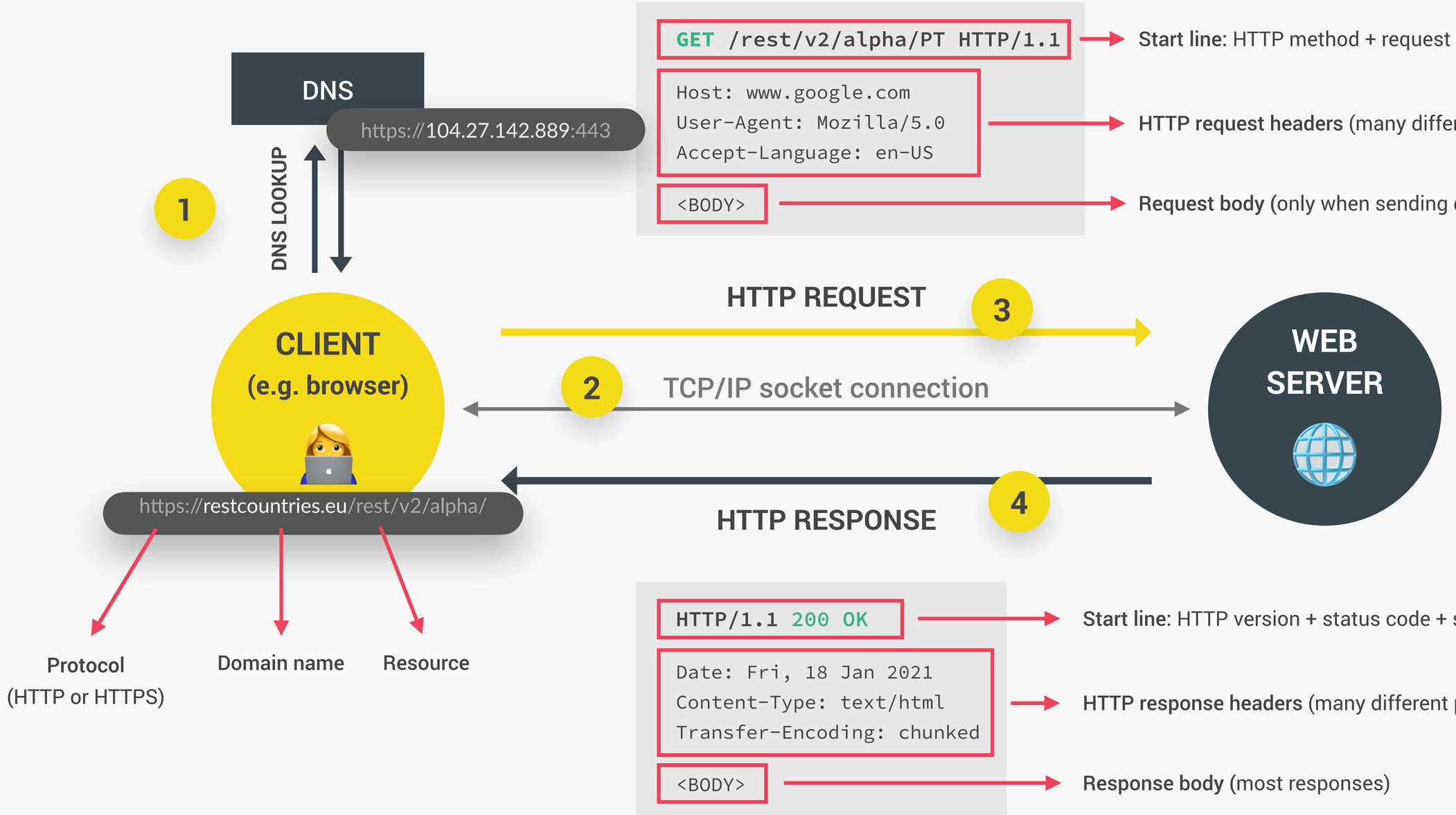
ASYNCHRONOUS JAVASCRIPT: PROMISES, ASYNC/AWAIT AND AJAX

LECTURE

HOW THE WEB WORKS: REQUESTS AND RESPONSES

JS

WHAT HAPPENS WHEN WE ACCESS A WEB SERVER



JONAS.IO
SCHMEDTMANN

THE COMPLETE JAVASCRIPT COURSE

FROM ZERO TO EXPERT!

@JONASSCHMEDTMAN

SECTION

ASYNCHRONOUS JAVASCRIPT: PROMISES, ASYNC/AWAIT AND AJAX

LECTURE

PROMISES AND THE FETCH API

JS

WHAT ARE PROMISES?

PROMISE

👉 **Promise:** An object that is used as a placeholder for the future result of an asynchronous operation.

↓ Less formal

👉 **Promise:** A container for an asynchronously delivered value.

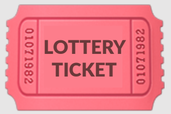
↓ Less formal

👉 **Promise:** A container for a future value.

Example: Response from AJAX call

👉 We no longer need to rely on events and callbacks passed into asynchronous functions to handle asynchronous results;

👉 Instead of nesting callbacks, we can **chain promises** for a sequence of asynchronous operations: **escaping callback hell** 🎉



Promise that I will receive money if I guess correct outcome

🤞 I buy lottery ticket (promise) right now

↓

🔮 Lottery draw happens asynchronously

↓

💰 If correct outcome, I receive money, because it was promised

THE PROMISE LIFECYCLE

Before the future value is available

Asynchronous task **has finished**

PENDING

ASYNC TASK

SETTLED

FULFILLED

REJECTED

Success! The value is now **available**

An **error** happened

👉 We are able **handle** these different states in our code!

E.g. Fetch API returns promise

BUILD PROMISE

CONSUME PROMISE

When we already have a promise. E.g. promise returned from Fetch API

JONAS.IO
SCHMEDTMANN

THE COMPLETE JAVASCRIPT COURSE

FROM ZERO TO EXPERT!

@JONASSCHMEDTMAN

SECTION

ASYNCHRONOUS JAVASCRIPT: PROMISES, ASYNC/AWAIT AND AJAX

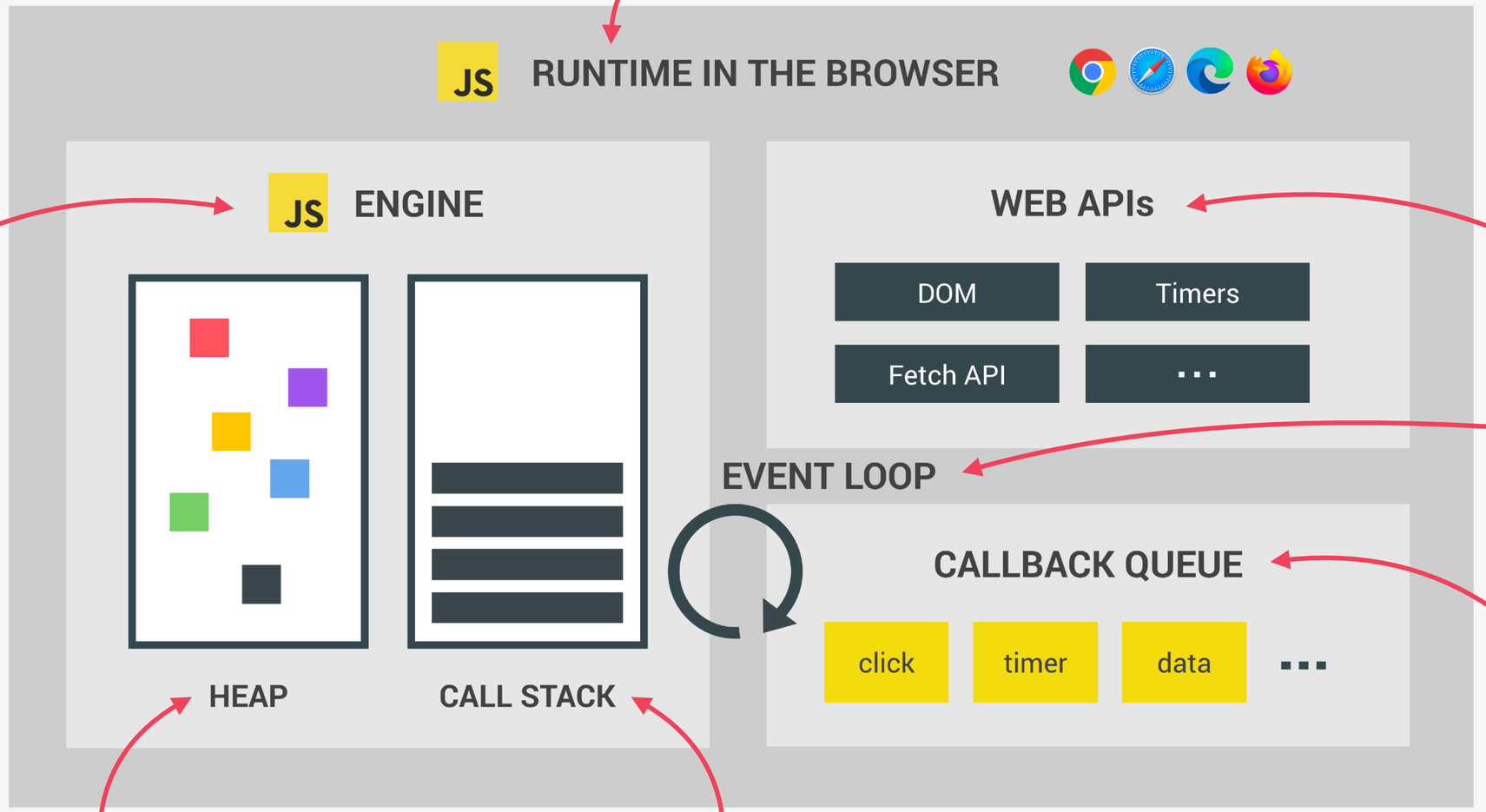
LECTURE

ASYNCHRONOUS BEHIND THE SCENES: THE EVENT LOOP

JS

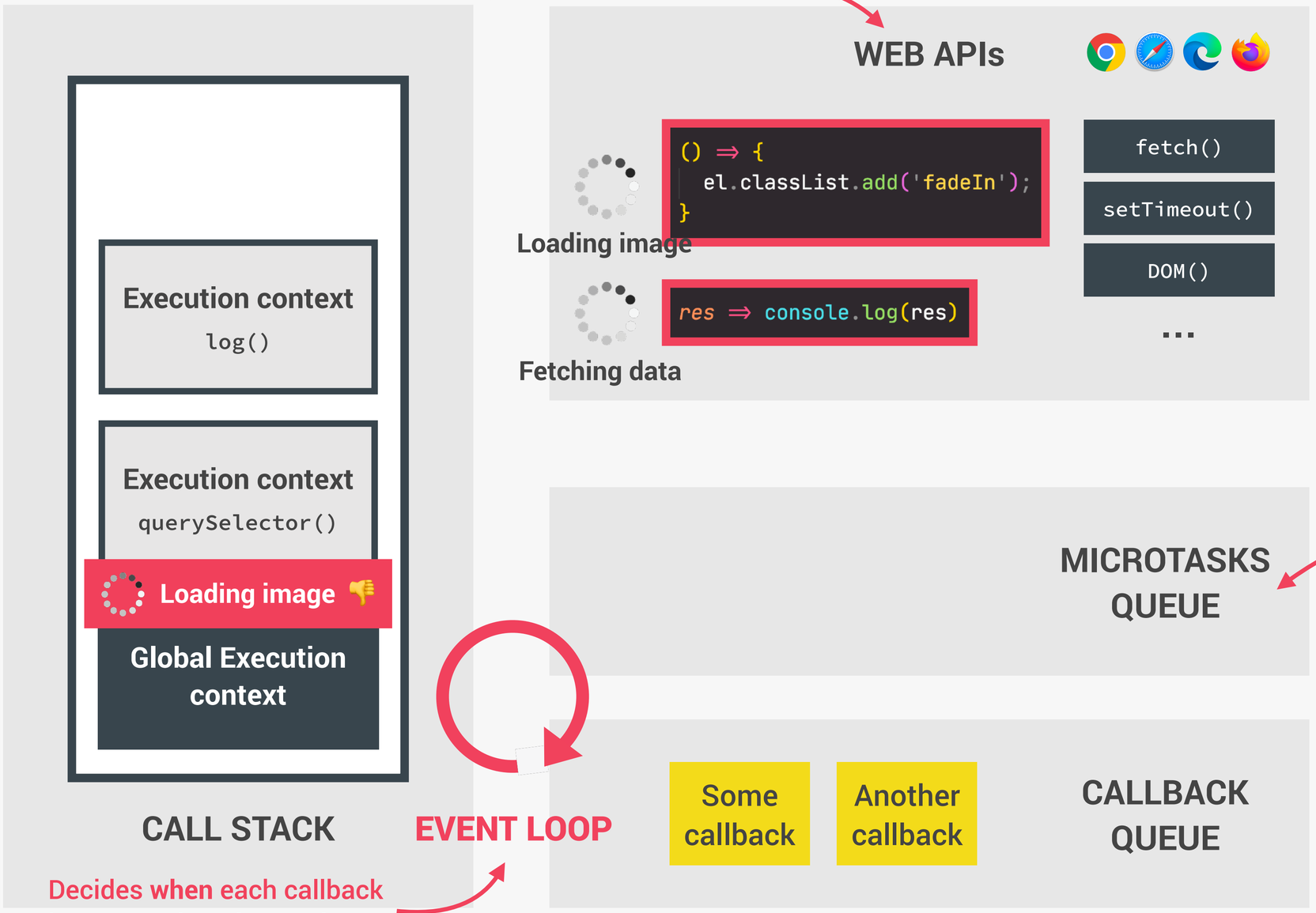
REVIEW: JAVASCRIPT RUNTIME

👉 **Concurrency model**: How JavaScript handles multiple tasks happening at the same time.



HOW ASYNCHRONOUS JAVASCRIPT WORKS BEHIND THE SCENES

Where asynchronous tasks run




```
el = document.querySelector('img');
el.src = 'dog.jpg';
el.addEventListener('load', () ⇒ {
  el.classList.add('fadeIn');
});

fetch('https://someurl.com/api')
  .then(res ⇒ console.log(res));

// More code...
```

🤔 How can **asynchronous** code be executed in a **non-blocking way**, if there is **only one thread** of execution in the engine?

JONAS.IO
SCHMEDTMANN

THE COMPLETE JAVASCRIPT COURSE

FROM ZERO TO EXPERT!

@JONASSCHMEDTMAN

SECTION

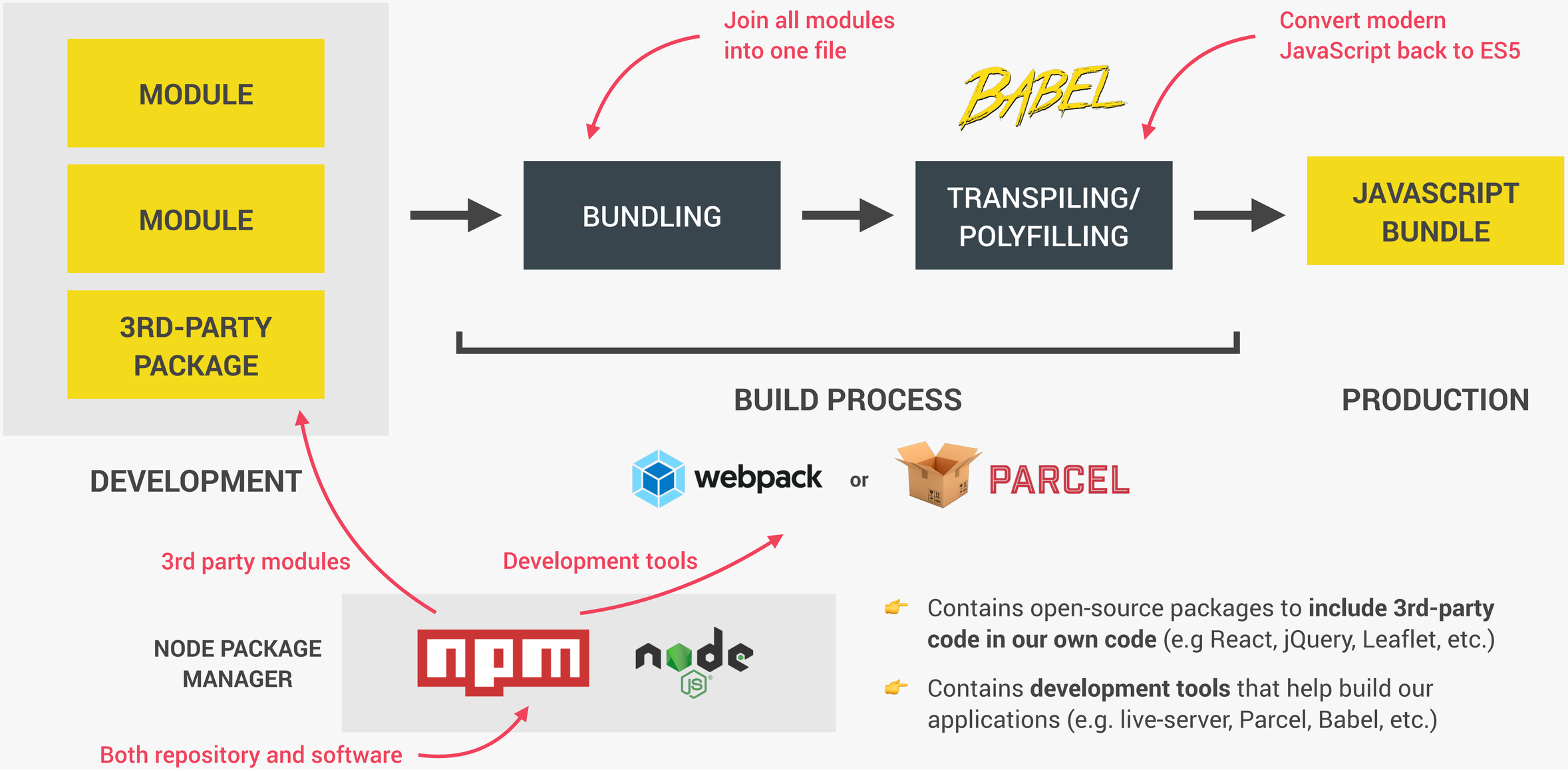
MODERN JAVASCRIPT DEVELOPMENT: MODULES AND TOOLING

LECTURE

AN OVERVIEW OF MODERN JAVASCRIPT DEVELOPMENT

JS

MODERN JAVASCRIPT DEVELOPMENT



- 👉 Contains open-source packages to **include 3rd-party code in our own code** (e.g React, jQuery, Leaflet, etc.)
- 👉 Contains **development tools** that help build our applications (e.g. live-server, Parcel, Babel, etc.)

JONAS.IO
SCHMEDTMANN

THE COMPLETE JAVASCRIPT COURSE

FROM ZERO TO EXPERT!

@JONASSCHMEDTMAN

SECTION

MODERN JAVASCRIPT DEVELOPMENT: MODULES AND TOOLING

LECTURE

AN OVERVIEW OF MODULES IN JAVASCRIPT

JS

NATIVE JAVASCRIPT (ES6) MODULES

ES6 MODULES

Modules stored in files, **exactly one module per file.**

```
import { rand } from './math.js';
const diceP1 = rand(1, 6, 2);
const diceP2 = rand(1, 6, 2);
const scores = { diceP1, diceP2 };
export { scores };
```

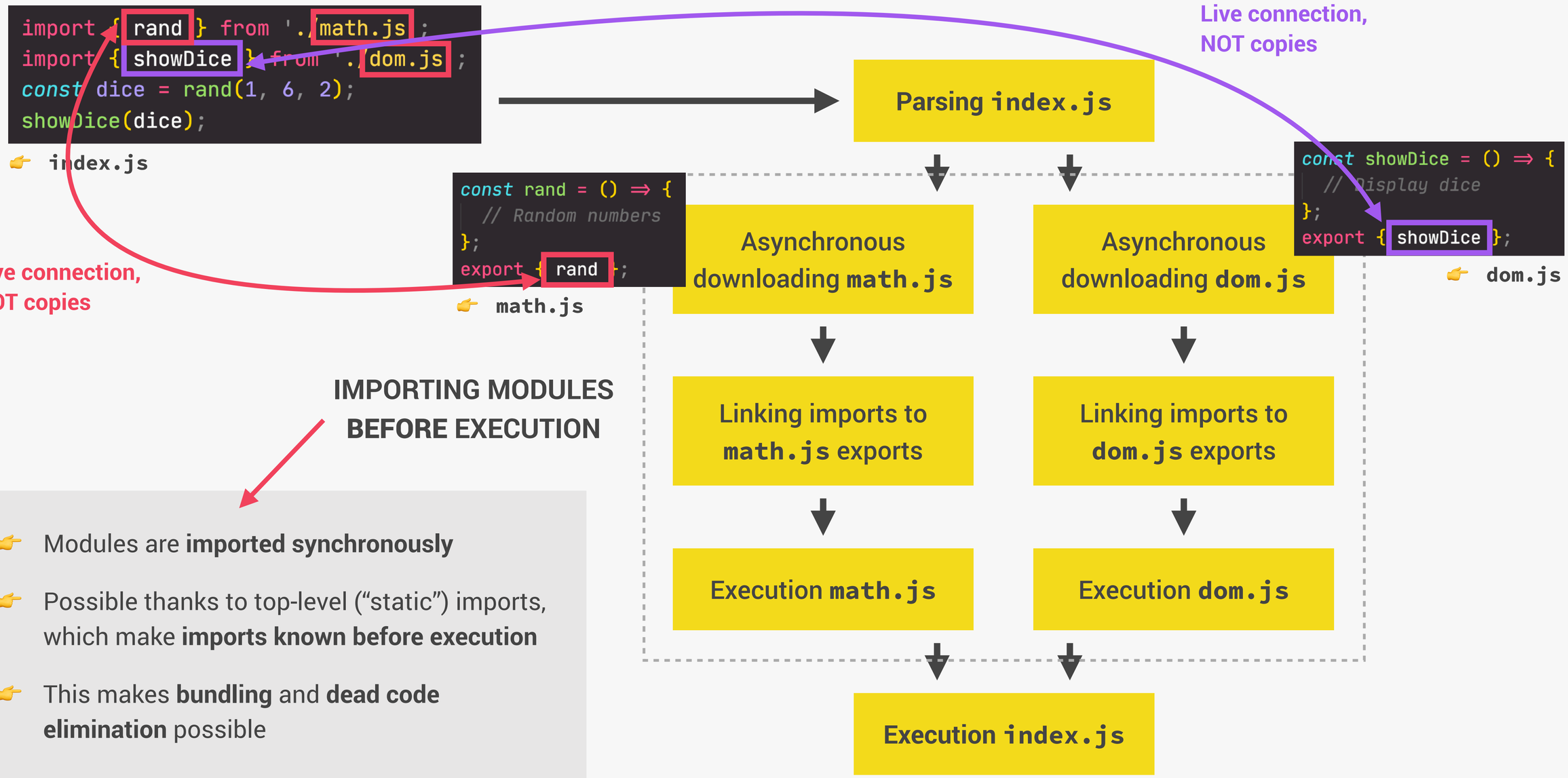
`import` and `export` syntax

	ES6 MODULE	SCRIPT
👉 **Top-level variables**	Scoped to module	Global
👉 **Default mode**	Strict mode	“Sloppy” mode
👉 **Top-level `this`**	`undefined`	`window`
👉 **Imports and exports**	✅ YES	🚫 NO
👉 **HTML linking**	`<script type="module">`	`<script>`
👉 **File downloading**	Asynchronous	Synchronous

☝️ Need to happen at top-level

Imports are hoisted!

HOW ES6 MODULES ARE IMPORTED



IMPORTING MODULES BEFORE EXECUTION

- Modules are **imported synchronously**
- Possible thanks to top-level (“static”) imports, which make **imports known before execution**
- This makes **bundling** and **dead code elimination** possible

JONAS.IO
SCHMEDTMANN

THE COMPLETE JAVASCRIPT COURSE

FROM ZERO TO EXPERT!

@JONASSCHMEDTMAN

SECTION

MODERN JAVASCRIPT DEVELOPMENT: MODULES AND TOOLING

LECTURE

REVIEW: WRITING CLEAN AND MODERN JAVASCRIPT

JS

REVIEW: MODERN AND CLEAN CODE

READABLE CODE

- 👉 Write code so that **others** can understand it
- 👉 Write code so that **you** can understand it in 1 year
- 👉 Avoid too “clever” and overcomplicated solutions
- 👉 Use descriptive variable names: **what they contain**
- 👉 Use descriptive function names: **what they do**

GENERAL

- 👉 Use DRY principle (refactor your code)
- 👉 Don’t pollute global namespace, encapsulate instead
- 👉 Don’t use `var`
- 👉 Use strong type checks (=== and `!==`)

FUNCTIONS

- 👉 Generally, functions should do **only one thing**
- 👉 Don’t use more than 3 function parameters
- 👉 Use default parameters whenever possible
- 👉 Generally, return same data type as received
- 👉 Use arrow functions when they make code more readable

OOP

- 👉 Use ES6 classes
- 👉 Encapsulate data and **don’t mutate** it from outside the class
- 👉 Implement method chaining
- 👉 Do **not** use arrow functions as methods (in regular objects)

REVIEW: MODERN AND CLEAN CODE

AVOID NESTED CODE

- 👉 Use early `return` (guard clauses)
- 👉 Use ternary (conditional) or logical operators instead of `if`
- 👉 Use multiple `if` instead of `if/else-if`
- 👉 Avoid `for` loops, use array methods instead
- 👉 Avoid callback-based asynchronous APIs

ASYNCHRONOUS CODE

- 👉 Consume promises with async/await for best readability
- 👉 Whenever possible, run promises in **parallel** (`Promise.all`)
- 👉 Handle errors and promise rejections

JONAS.IO
SCHMEDTMANN

THE COMPLETE JAVASCRIPT COURSE

FROM ZERO TO EXPERT!

@JONASSCHMEDTMAN

SECTION

MODERN JAVASCRIPT DEVELOPMENT: MODULES AND TOOLING

LECTURE

DECLARATIVE AND FUNCTIONAL JAVASCRIPT PRINCIPLES

JS

IMPERATIVE VS. DECLARATIVE CODE

Two fundamentally different ways of writing code (paradigms)

IMPERATIVE

- 👉 Programmer explains **“HOW to do things”**
- 👉 We explain the computer *every single step* it has to follow to achieve a result
- 👉 **Example:** Step-by-step recipe of a cake

```
const arr = [2, 4, 6, 8];
const doubled = [];
for (let i = 0; i < arr.length; i++)
  doubled[i] = arr[i] * 2;
```

DECLARATIVE

- 👉 Programmer tells **“WHAT do do”**
- 👉 We simply *describe* the way the computer should achieve the result
- 👉 The **HOW** (step-by-step instructions) gets abstracted away
- 👉 **Example:** Description of a cake

```
const arr = [2, 4, 6, 8];
const doubled = arr.map(n ⇒ n * 2);
```

FUNCTIONAL PROGRAMMING PRINCIPLES

FUNCTIONAL PROGRAMMING

- **Declarative** programming paradigm
- Based on the idea of writing software by combining many **pure functions**, avoiding **side effects** and **mutating** data

- **Side effect:** Modification (mutation) of any data **outside** of the function (mutating external variables, logging to console, writing to DOM, etc.)
- **Pure function:** Function without side effects. Does not depend on external variables. **Given the same inputs, always returns the same outputs**.
- **Immutability:** State (data) is **never** modified! Instead, state is **copied** and the copy is mutated and returned.

- **Examples:**   Redux

FUNCTIONAL PROGRAMMING TECHNIQUES

- Try to avoid data mutations
- Use built-in methods that don't produce side effects
- Do data transformations with methods such as `.map()`, `.filter()` and `.reduce()`
- Try to avoid side effects in functions: this is of course not always possible!

DECLARATIVE SYNTAX

- Use array and object destructuring
- Use the spread operator (`...`)
- Use the ternary (conditional) operator
- Use template literals

JONAS.IO
SCHMEDTMANN

THE COMPLETE JAVASCRIPT COURSE

FROM ZERO TO EXPERT!

@JONASSCHMEDTMAN

SECTION

FORKIFY APP: BUILDING A MODERN APPLICATION

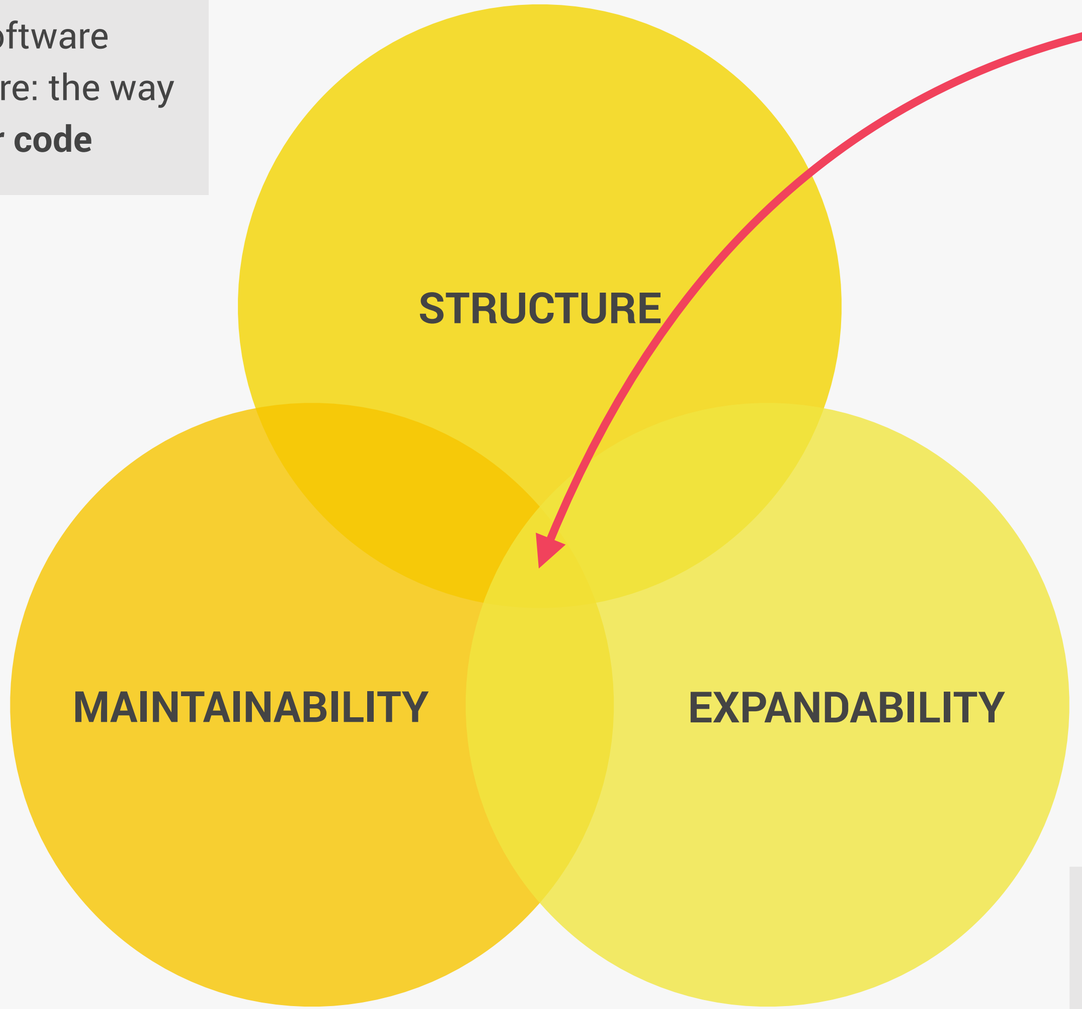
LECTURE

THE MVC ARCHITECTURE

JS

WHY WORRY ABOUT ARCHITECTURE?

👉 Like a house, software needs a structure: the way we **organize our code**



The perfect architecture

👉 We can create our own architecture (Mapty project)

👉 We can use a well-established architecture pattern like MVC, MVP, Flux, etc. (**this project**)

👉 We can use a framework like React, Angular, Vue, Svelte, etc.

👉 A project is never done! We need to be able to easily **change it in the future**

👉 We also need to be able to easily **add new features**

COMPONENTS OF ANY ARCHITECTURE

BUSINESS LOGIC

- 👉 Code that **solves the actual business problem**;
- 👉 Directly related to what business does and what it needs;
- 👉 **Example:** sending messages, storing transactions, calculating taxes, …

STATE

- 👉 Essentially **stores all the data** about the application
- 👉 Should be the “single source of truth”
- 👉 UI should be kept in sync with the state
- 👉 State libraries exist


HTTP LIBRARY

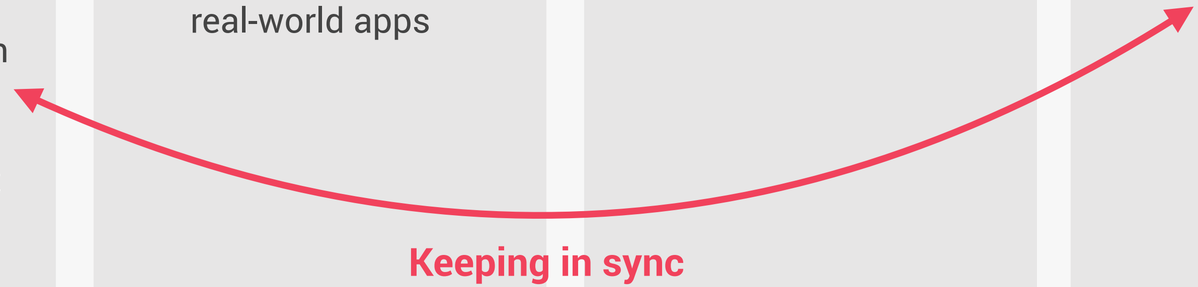
- 👉 Responsible for making and receiving AJAX requests
- 👉 Optional but almost always necessary in real-world apps

APPLICATION LOGIC (ROUTER)

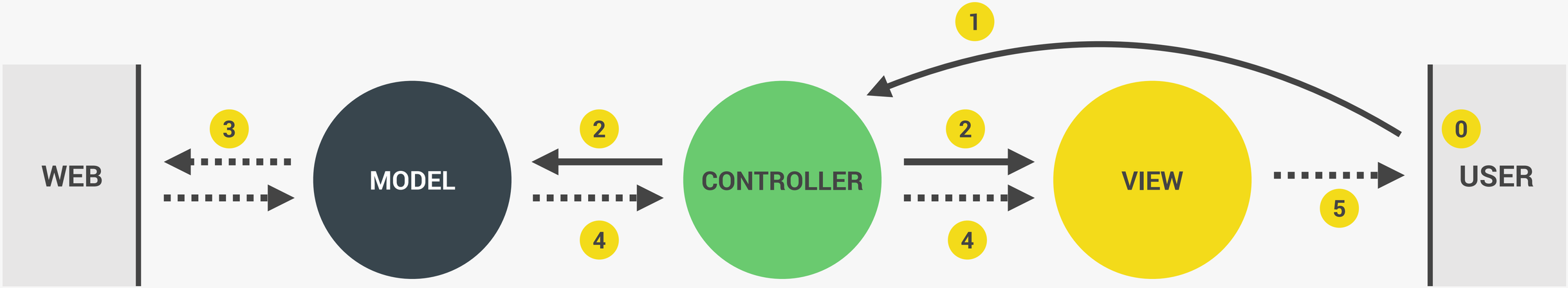
- 👉 Code that is only concerned about the **implementation of application itself**;
- 👉 Handles navigation and UI events

PRESENTATION LOGIC (UI LAYER)

- 👉 Code that is concerned about the **visible part** of the application
- 👉 Essentially displays application state



THE MODEL-VIEW-CONTROLLER (MVC) ARCHITECTURE



BUSINESS LOGIC

STATE

HTTP LIBRARY

APPLICATION LOGIC

- 👉 Bridge between model and views (which don't know about one another)
- 👉 Handles UI events and **dispatches tasks to model and view**

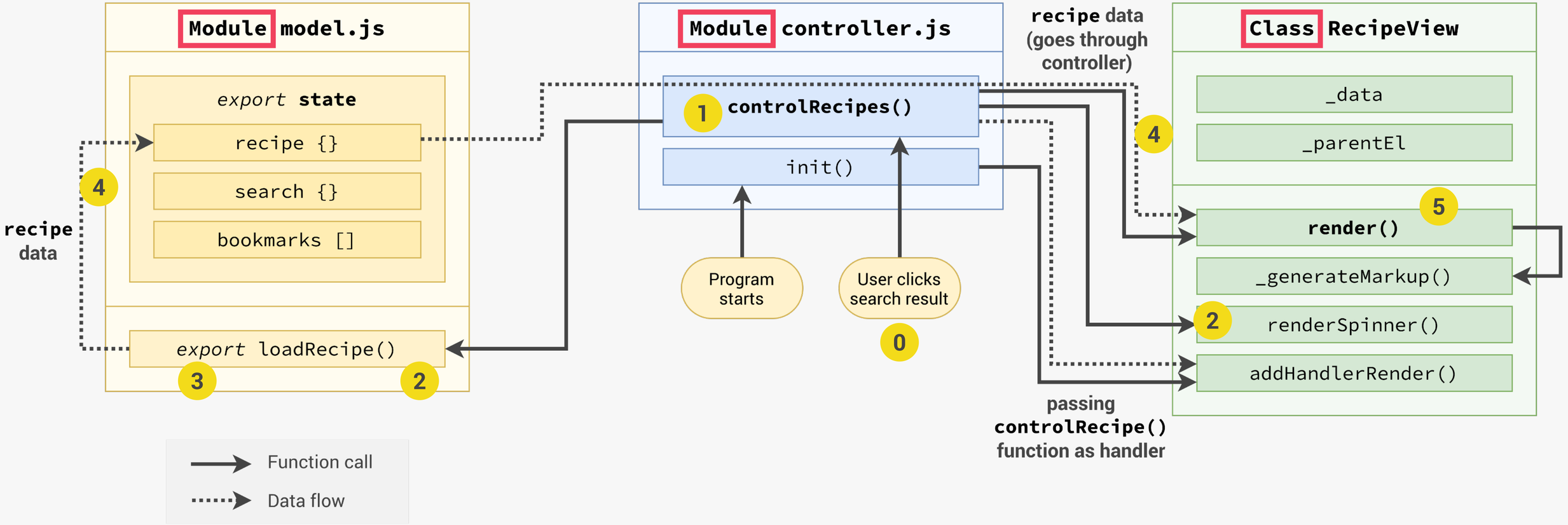
PRESENTATION LOGIC

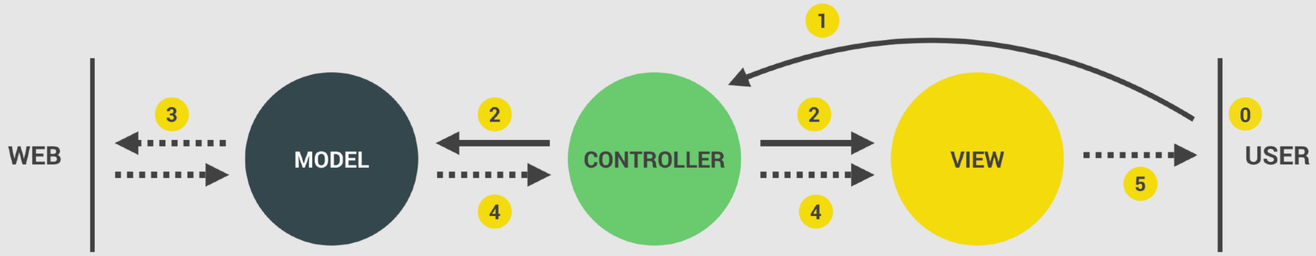
→ Connected by function call and import

⇢ Data flow

MVC IMPLEMENTATION (RECIPE DISPLAY ONLY)

forkify





JONAS.IO
SCHMEDTMANN

THE COMPLETE JAVASCRIPT COURSE

FROM ZERO TO EXPERT!

SECTION

FORKIFY APP: BUILDING A MODERN APPLICATION

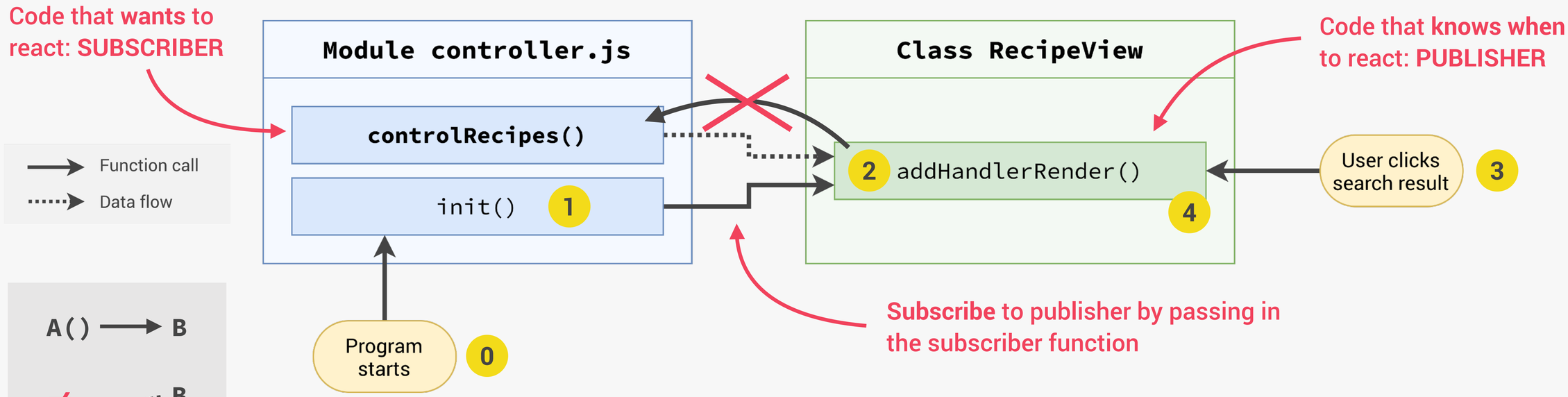
LECTURE

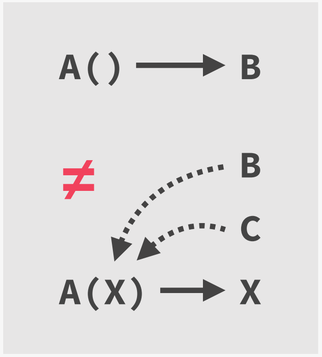
EVENT HANDLERS IN MVC: PUBLISHER-SUBSCRIBER PATTERN

@JONASSCHMEDTMAN

JS

EVENT HANDLING IN MVC: PUBLISHER-SUBSCRIBER PATTERN





- 👉 `controlRecipes` will be passed into `addHandlerRender` when program starts
- 👉 `addHandlerRender` listens for events (`addEventListener`), and uses `controlRecipes` as callback

- 👉 Events should be **handled** in the **controller** (otherwise we would have application logic in the view)
- 👉 Events should be **listened for** in the **view** (otherwise we would need DOM elements in the controller)

JONAS.IO
SCHMEDTMANN

THE COMPLETE JAVASCRIPT COURSE

FROM ZERO TO EXPERT!

@JONASSCHMEDTMAN

SECTION

FORKIFY APP: BUILDING A MODERN APPLICATION

LECTURE

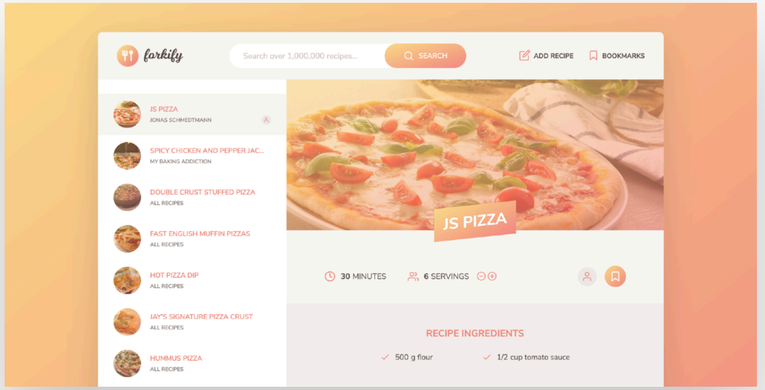
WRAPPING UP: FINAL CONSIDERATIONS

JS

IMPROVEMENT AND FEATURE IDEAS: CHALLENGES 🤓

forkify

- 👉 Display **number of pages** between the pagination buttons;
- 👉 Ability to **sort** search results by duration or number of ingredients;
- 👉 Perform **ingredient validation** in view, before submitting the form;
- 👉 **Improve recipe ingredient input**: separate in multiple fields and allow more than 6 ingredients;



- 👉 **Shopping list feature**: button on recipe to add ingredients to a list;
- 👉 **Weekly meal planning feature**: assign recipes to the next 7 days and show on a weekly calendar;
- 👉 **Get nutrition data** on each ingredient from spoonacular API (https://spoonacular.com/food-api) and calculate total calories of recipe.