



JONAS.IO  
SCHMEDTMANN

Subscribe here

# THE COMPLETE JAVASCRIPT COURSE

FROM ZERO TO EXPERT!



@JONASSCHMEDTMAN

Follow me here

## SLIDES FOR THEORY LECTURES

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





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

- 1 Watch before you start!
- 2 A Brief Introduction to JavaScript
- 3 Data Types
- 4 Boolean Logic
- 5 JavaScript Releases: ES5, ES6+ and ESNext
- 6 Functions Calling Other Functions
- 7 Reviewing Functions
- 8 Learning How to Code
- 9 How to Think Like a Developer
- 10 Debugging (Fixing Errors)
- 11 What's the DOM and DOM Manipulation
- 12 An high-level Overview of JavaScript
- 13 The JavaScript Engine and Runtime
- 14 Execution Contexts and The Call Stack
- 15 Scope and The Scope Chain
- 16 Variable environment: Hoisting and The TDZ
- 17 The this Keyword
- 18 Primitives vs. Objects (Primitive vs. Reference Types)
- 19 Summary: Which Data Structure to Use?
- 20 First-Class and Higher-Order Functions
- 21 Closures
- 22 Data Transformations: map, filter, reduce
- 23 Summary: Which Array Method to Use?
- 24 How the DOM Really Works
- 25 Event Propagation: Bubbling and Capturing
- 26 Efficient Script Loading: defer and async
- 27 What is Object-Oriented Programming?
- 28 OOP in JavaScript
- 29 Prototypal Inheritance and The Prototype Chain
- 30 Object.create
- 31 Inheritance Between "Classes": Constructor Functions
- 32 Inheritance Between "Classes": Object.create
- 33 ES6 Classes summary
- 34 Mapty Project: How to Plan a Web Project
- 35 Mapty Project: Final Considerations
- 36 Asynchronous JavaScript, AJAX and APIs
- 37 How the Web Works: Requests and Responses
- 38 Promises and the Fetch API
- 39 Asynchronous Behind the Scenes: The Event Loop
- 40 An Overview of Modern JavaScript Development
- 41 An Overview of Modules in JavaScript
- 42 Modern, Clean and Declarative JavaScript Programming
- 43 Forkify: Project Overview and Planning
- 44 The MVC Architecture
- 45 Event Handlers in MVC: Publisher-Subscriber Pattern
- 46 Forkify Project: Final Considerations





WELCOME, WELCOME,  
WELCOME!





JONAS.IO  
SCHMEDTMANN

# THE COMPLETE JAVASCRIPT COURSE

FROM ZERO TO EXPERT!



@JONASSCHMEDTMAN

SECTION

WELCOME, WELCOME, WELCOME!

LECTURE

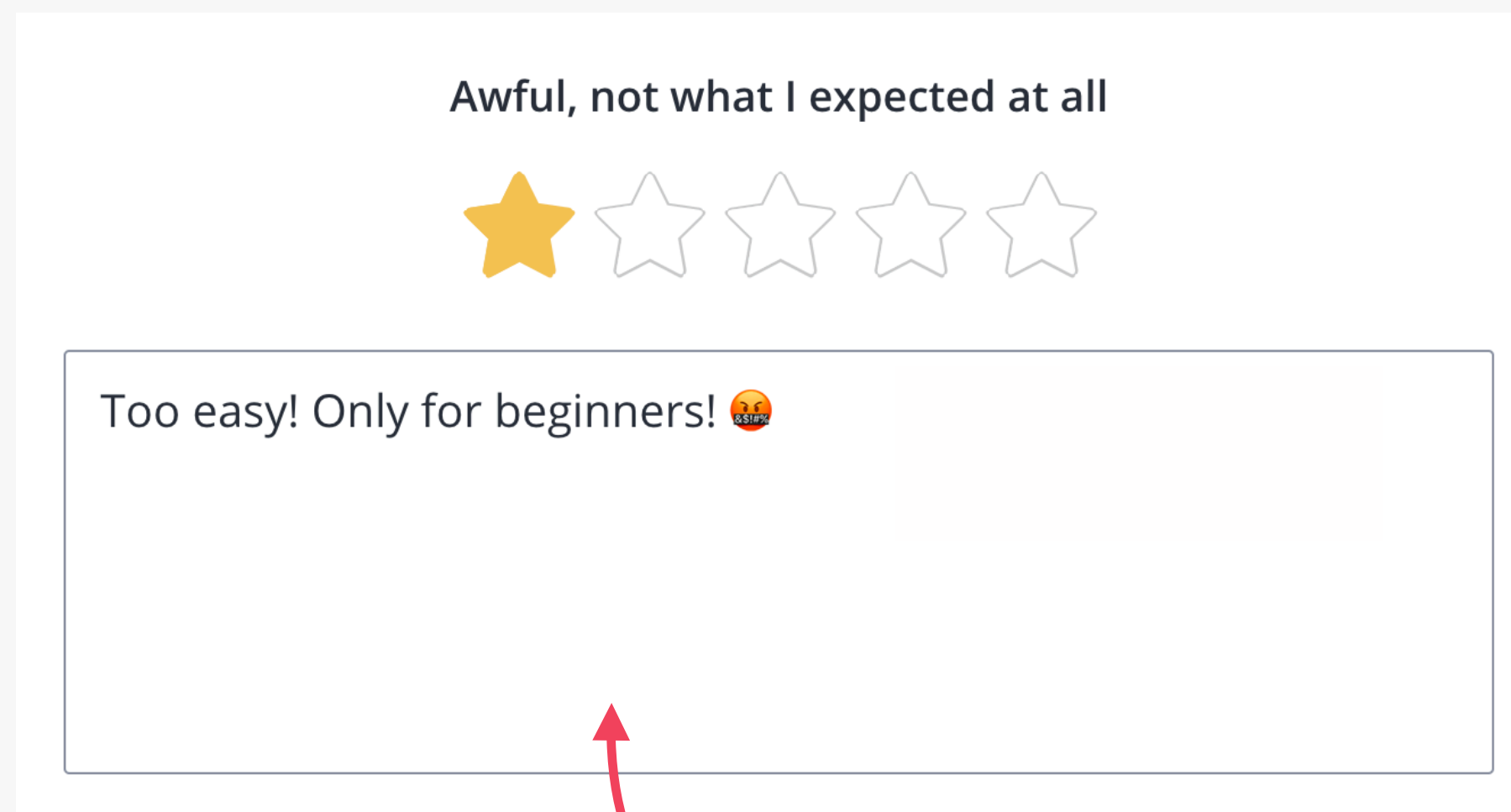
WATCH BEFORE YOU START!

JS

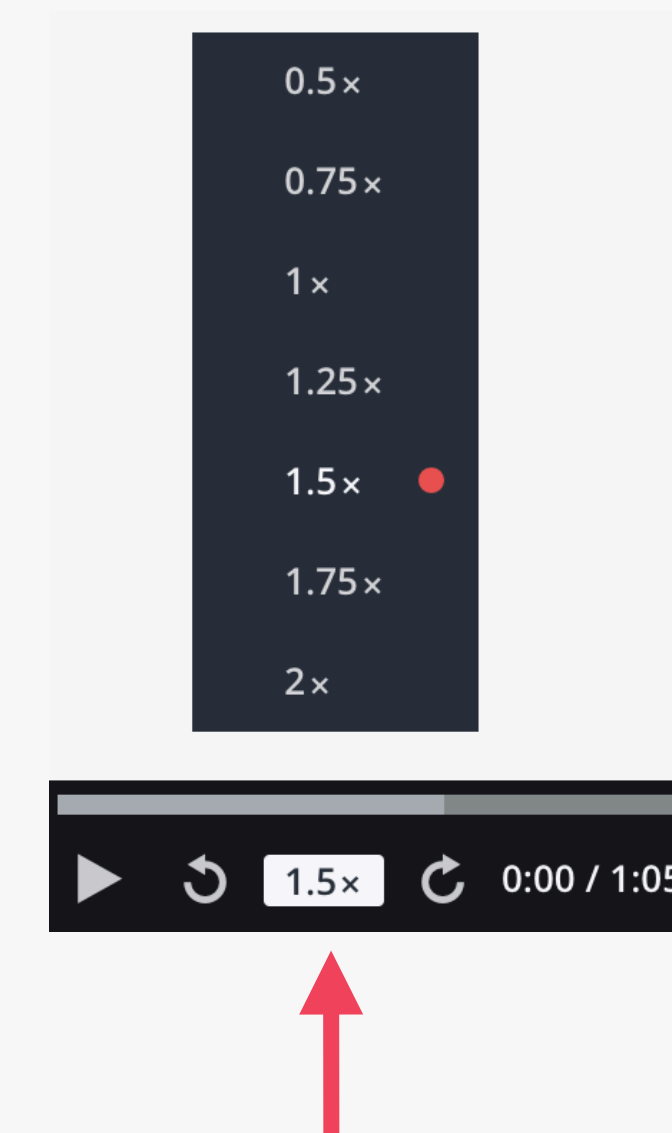
# SOME QUICK CONSIDERATIONS BEFORE WE START... 🚀



**This course is for all of you!** So please don't write a bad review right away if the course is too easy, or too hard, or progressing too slow, or too fast for you. To make it perfect for **YOU**, you can rewatch lectures, jump to other sections, watch the course with slower or faster playback speed, or ask questions.

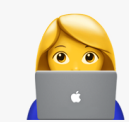


Please don't be that person.  
Everyone is different...  
(Unless the course *itself* is truly terrible)





# SOME QUICK CONSIDERATIONS BEFORE WE START... 🚀



**You need to code along with me!** You will learn **ZERO** JavaScript skills by just sitting and watching me code. You have to code **YOURSELF!**





# SOME QUICK CONSIDERATIONS BEFORE WE START... 🚀



**Try all the coding challenges!** Try to do your best, but if you get stuck for too long, watch the solution. **Don't beat yourself up if you can't figure it out!** Just rewatch the lectures that were covered in the challenge, try to understand them better, and move on.



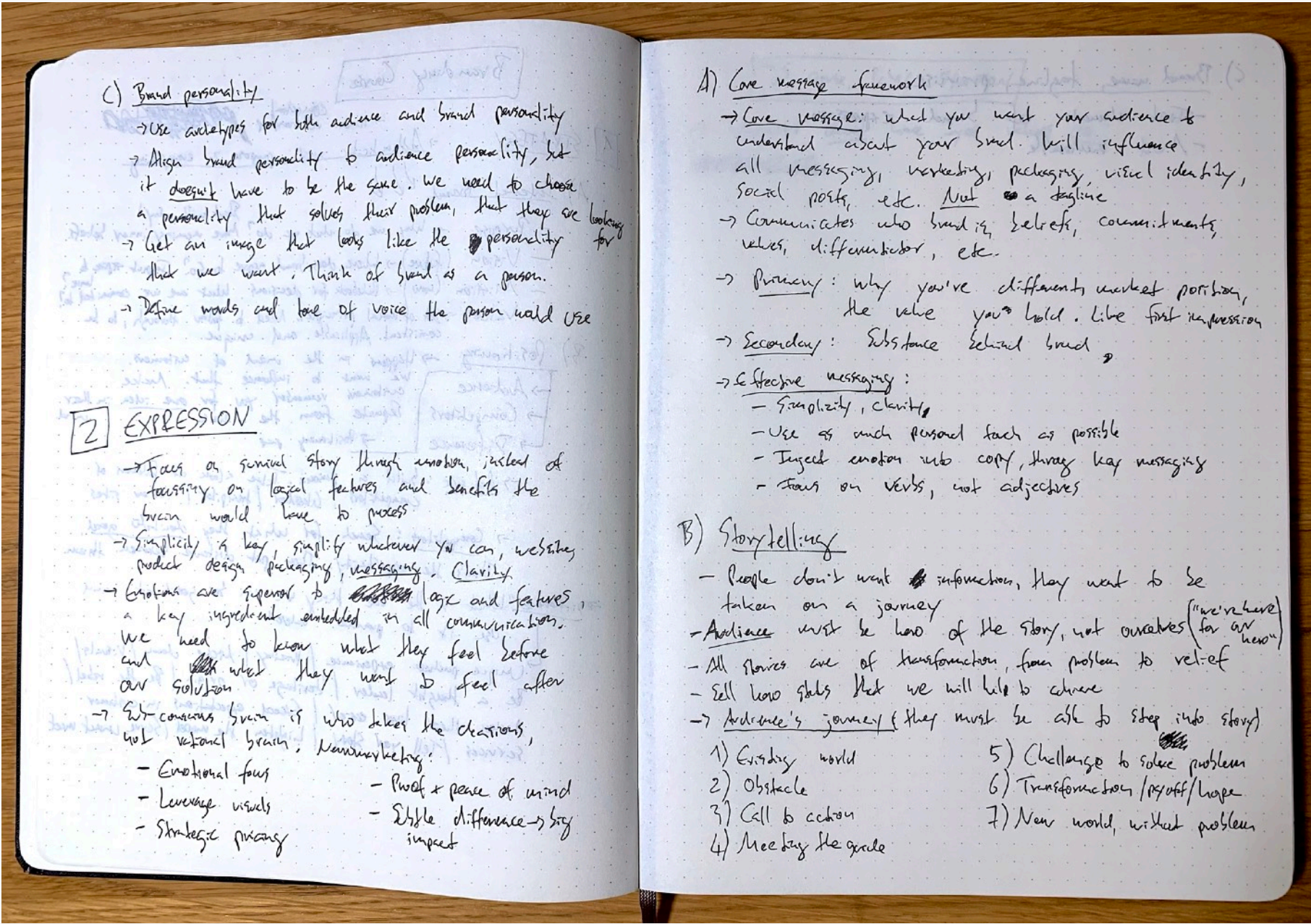
Watch for this sign!

PAUSE THE VIDEO  
FOR CHALLENGE

# SOME QUICK CONSIDERATIONS BEFORE WE START.. 🚀



If you want the course material to stick, take notes. Notes on code syntax, notes on theory concepts, notes on everything!



Totally non-coding... Try to understand a single word 😂



# SOME QUICK CONSIDERATIONS BEFORE WE START... 🚀

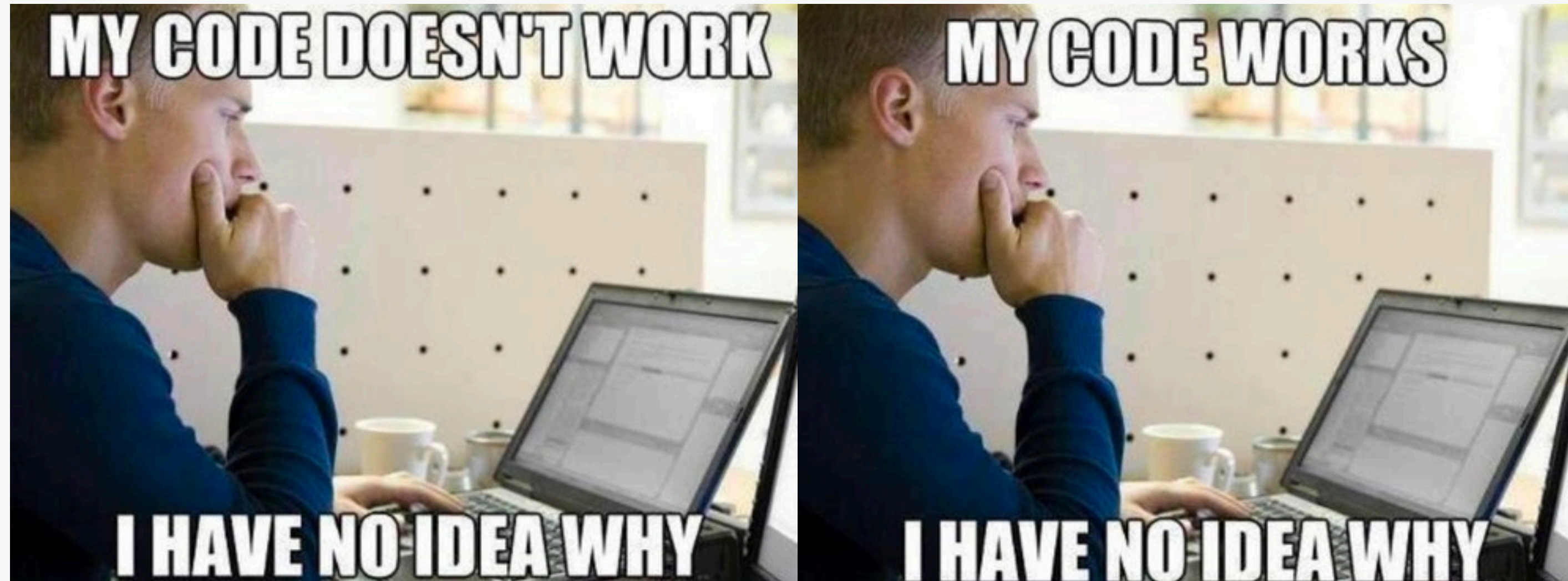
😱 If this is your first time ever programming, please don't get overwhelmed. It's 100% normal that **you will not understand everything** at the beginning. ***Just don't think "I guess coding is not for me"!***





# SOME QUICK CONSIDERATIONS BEFORE WE START... 🚀

😅 In the first sections of the course, don't bother understanding **WHY** things work the way they do in JavaScript. Also, don't stress about **efficient code**, or **fast code**, or **clean code**. While learning, we just want to make things **WORK**. We will understand the **WHY** later in the course.

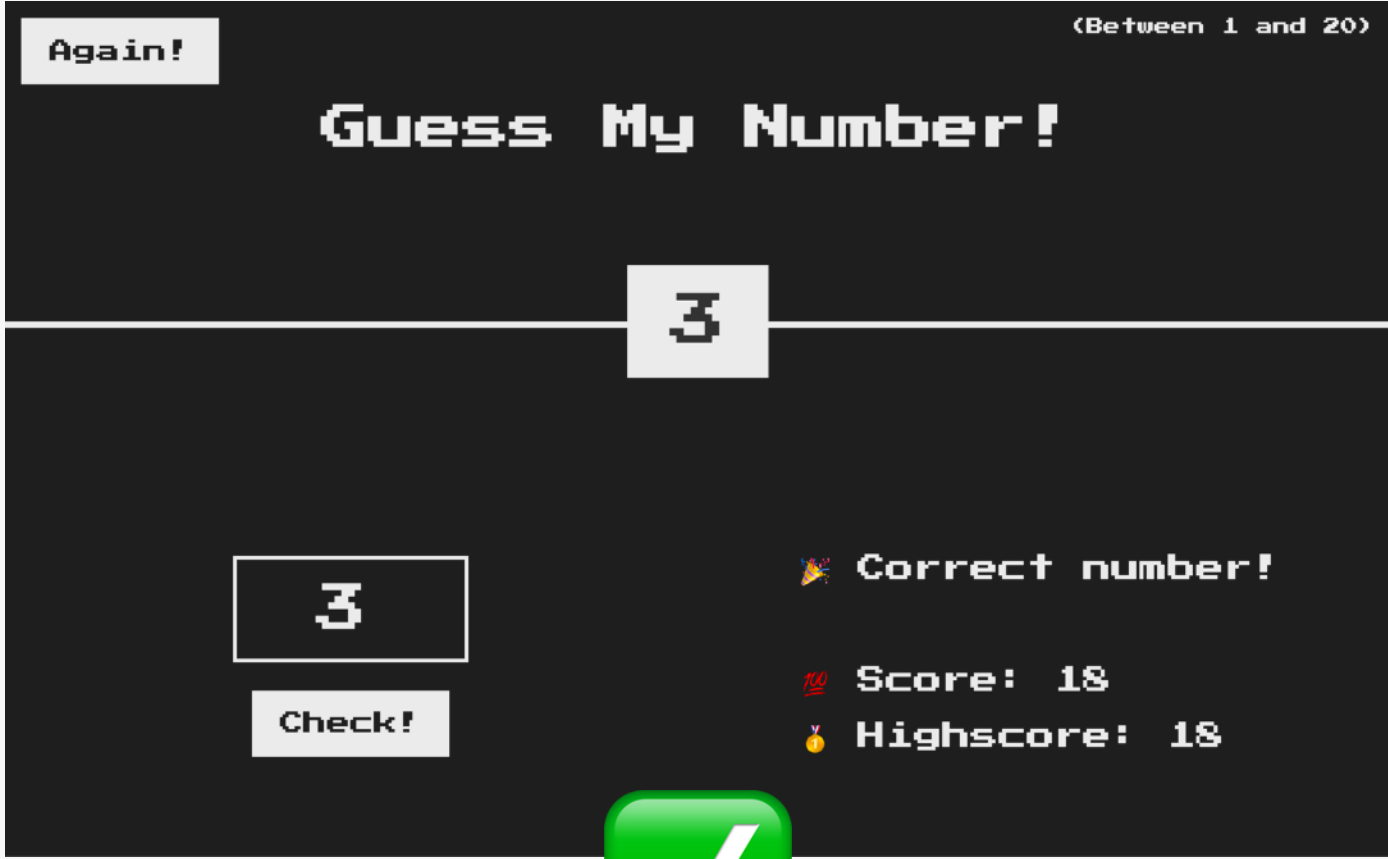
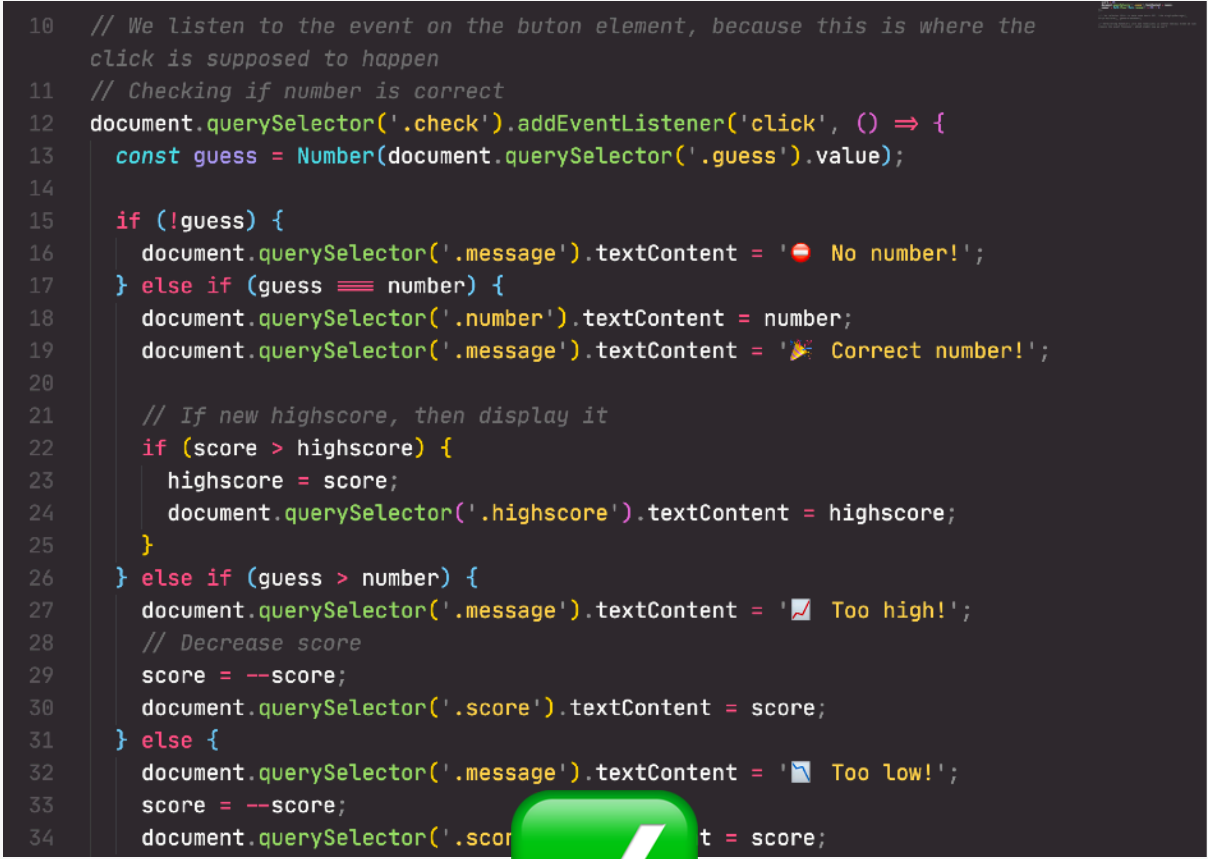
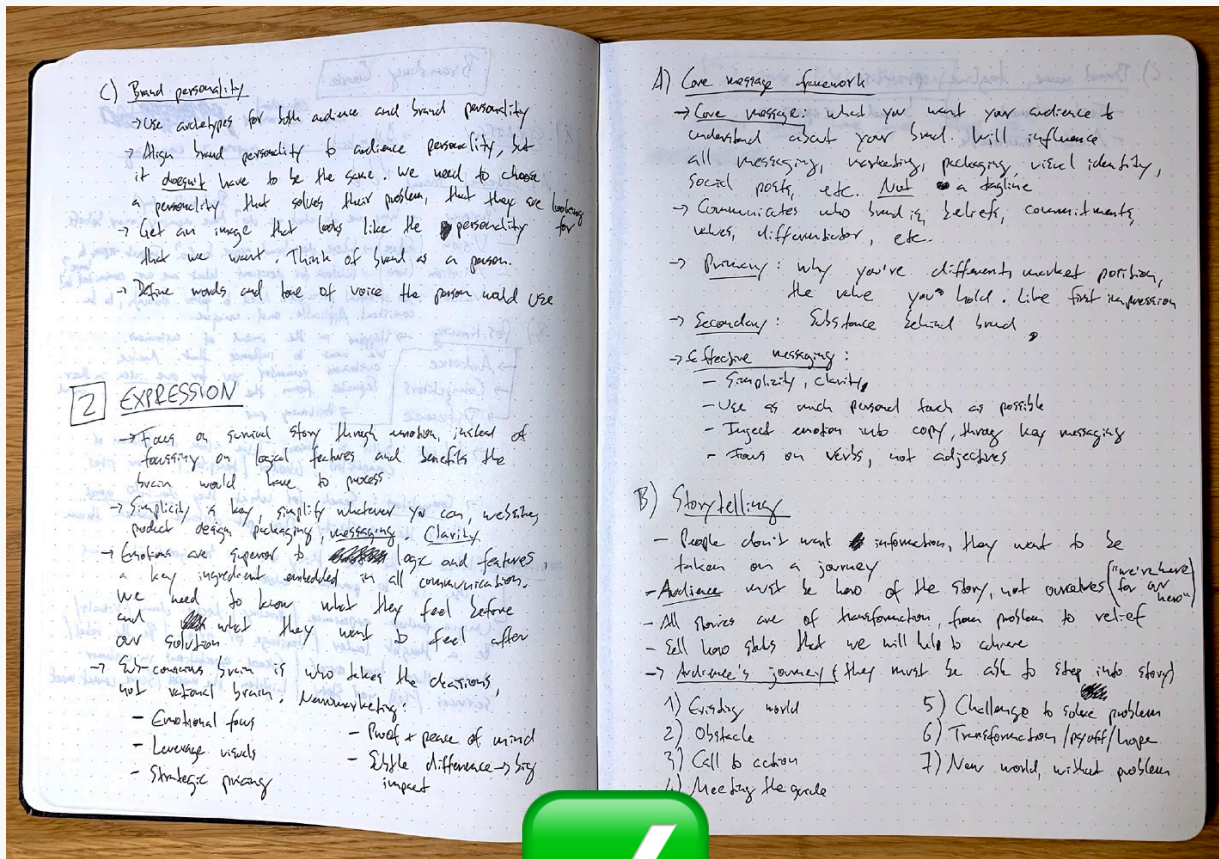




# SOME QUICK CONSIDERATIONS BEFORE WE START... 🚀



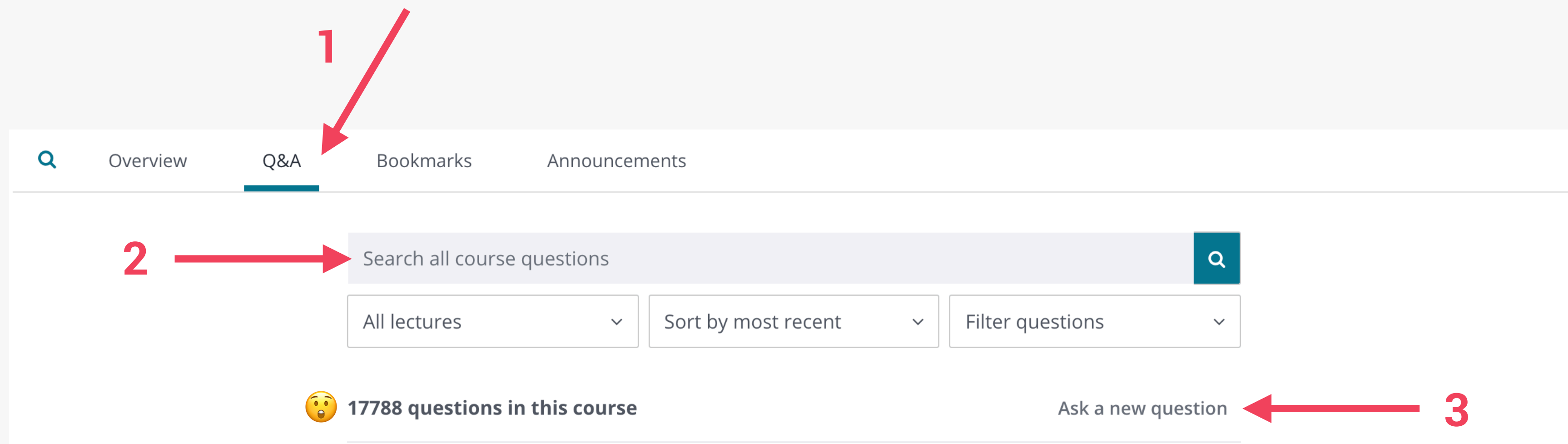
Before moving on from a section, make sure that you understand exactly what was covered. Take a break, review the code we wrote, review your notes, review the projects we built, and maybe even write some code yourself.





# SOME QUICK CONSIDERATIONS BEFORE WE START...

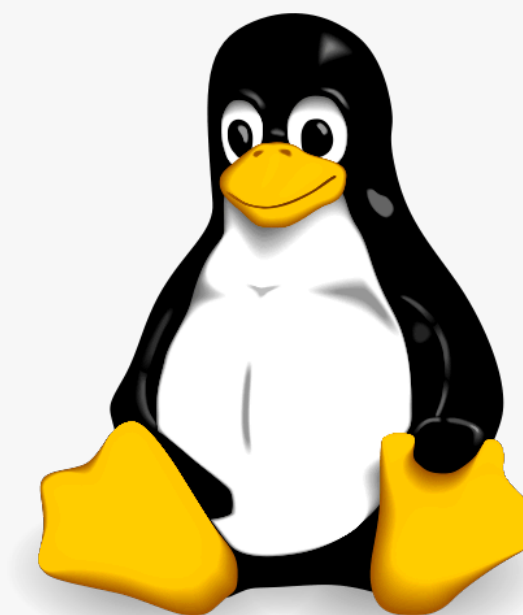
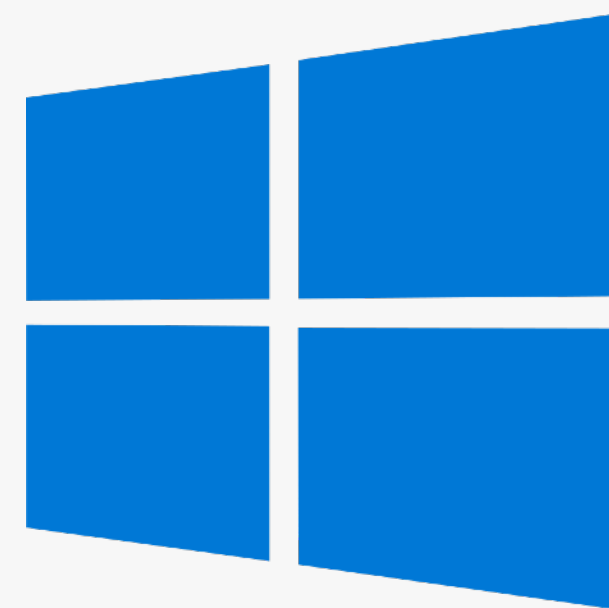
**!?** If you have an error or a question, start by trying to solve it yourself! This is essential for your progress. If you can't solve it, check the Q&A section. If that doesn't help, just ask a new question. Use a short description, and post relevant code.



# SOME QUICK CONSIDERATIONS BEFORE WE START... 🚀



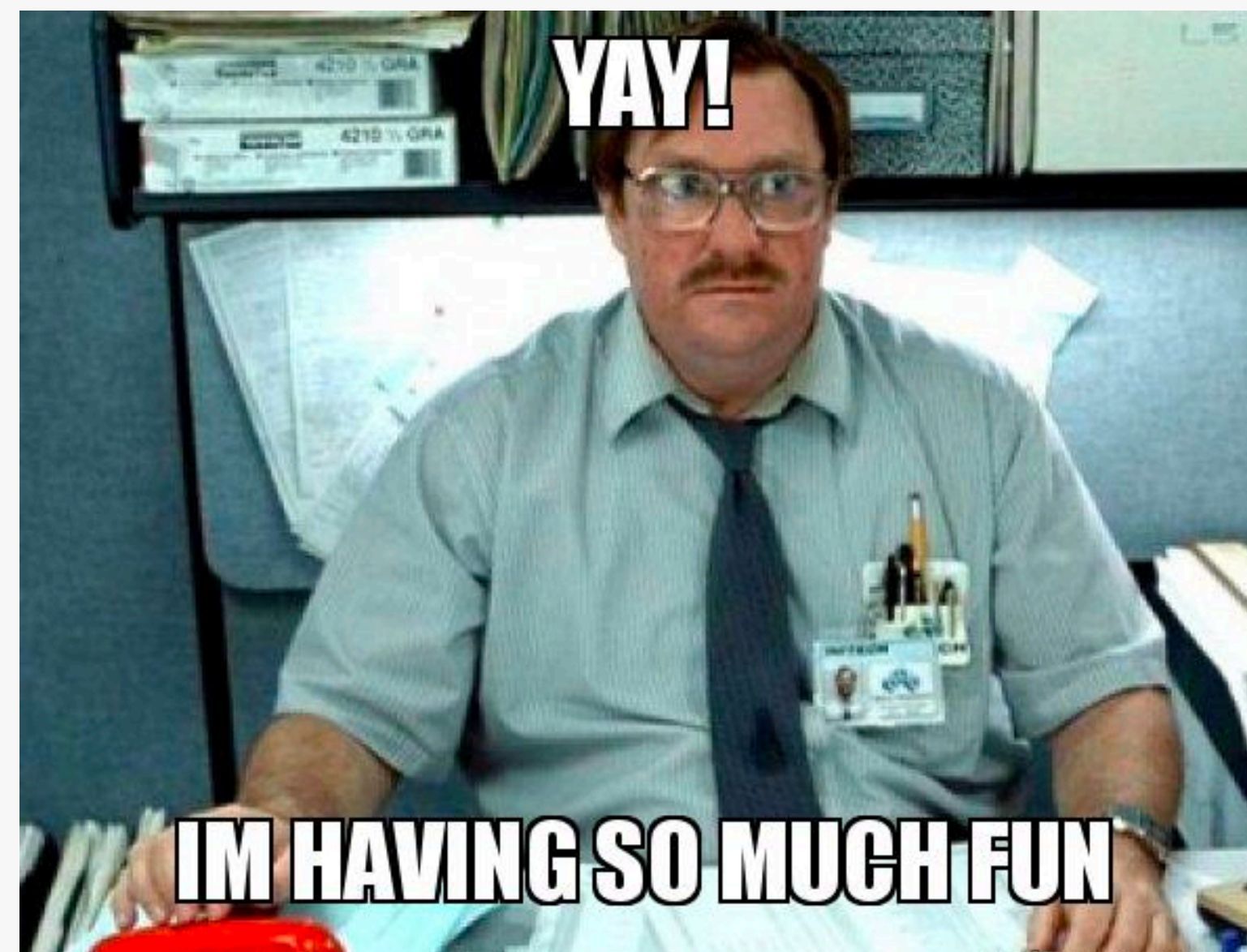
I recorded this course on a Mac, but everything works the exact same way on Windows or Linux. If something doesn't work on your computer, it's **NOT** because you're using a different OS.





# SOME QUICK CONSIDERATIONS BEFORE WE START... 🚀

💖 **Most importantly, have fun!** It's so rewarding to see something that **YOU** have built **YOURSELF!** So if you're feeling frustrated, stop whatever you're doing, and come back later!



And I mean **REAL** fun 😄





# JAVASCRIPT FUNDAMENTALS – PART 1



JONAS.IO  
SCHMEDTMANN

# THE COMPLETE JAVASCRIPT COURSE

FROM ZERO TO EXPERT!



@JONASSCHMEDTMAN

SECTION

JAVASCRIPT FUNDAMENTALS - PART 1

LECTURE

A BRIEF INTRODUCTION TO  
JAVASCRIPT





# WHAT IS JAVASCRIPT?

JAVASCRIPT

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

Based on objects, for  
storing most kinds of data

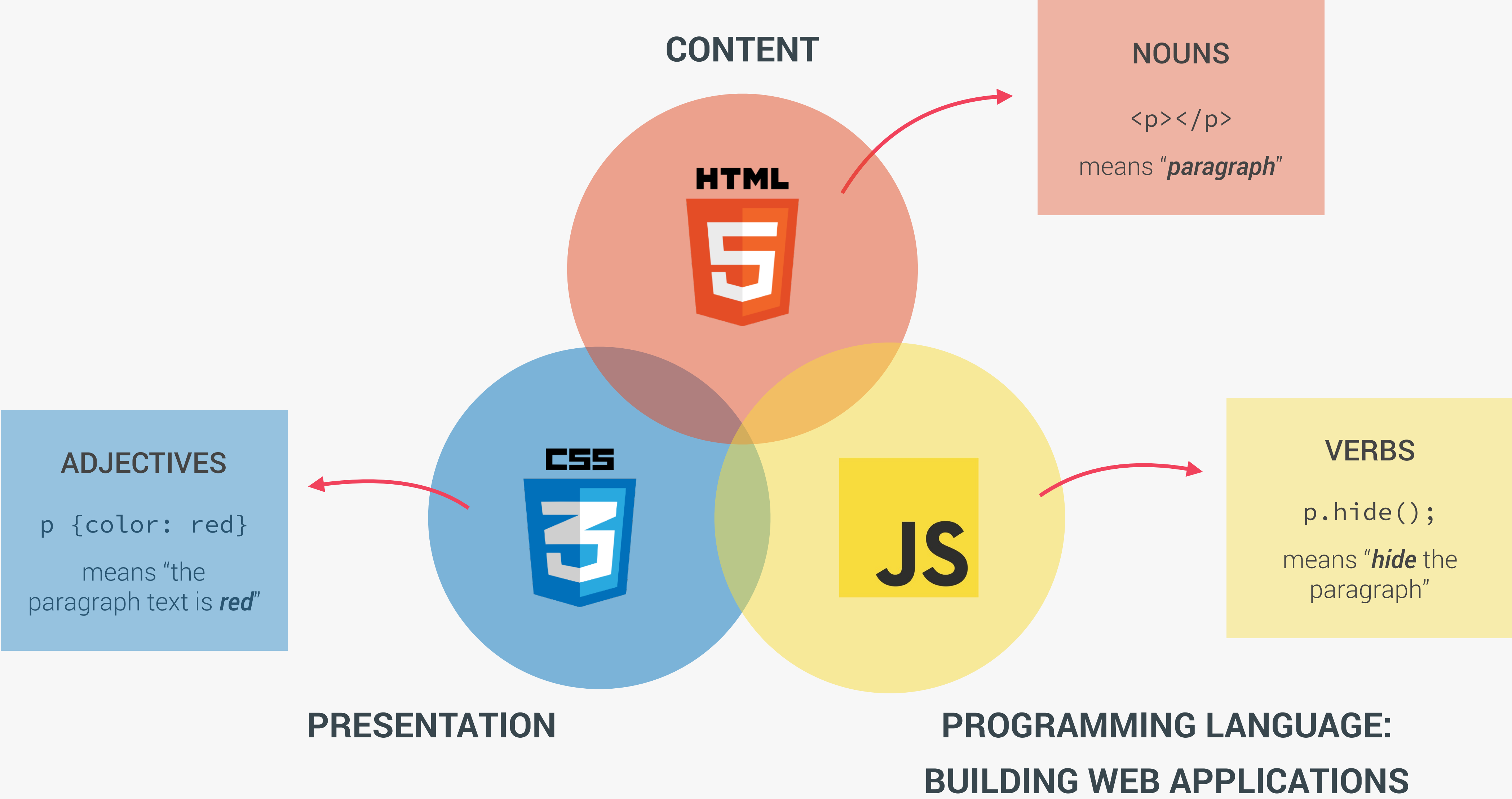
We don't have to worry about complex  
stuff like memory management

We can use different styles  
of programming

Instruct computer to *do* things

JS

# THE ROLE OF JAVASCRIPT IN WEB DEVELOPMENT



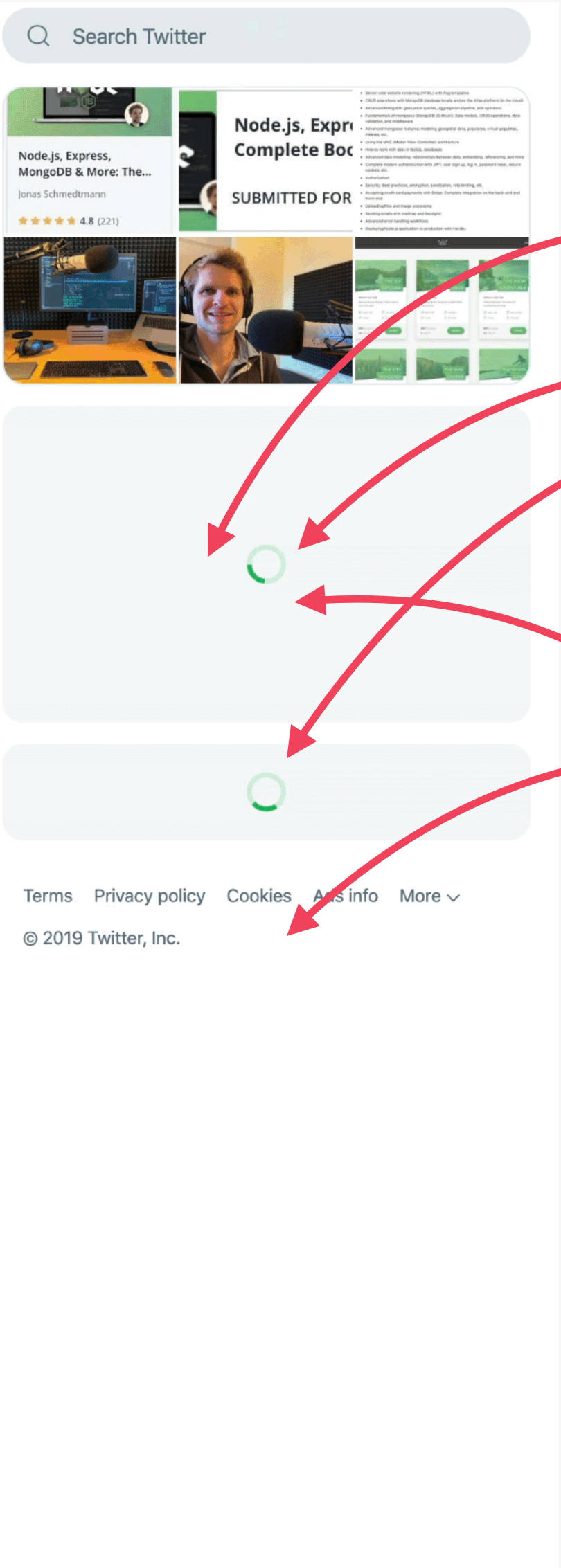
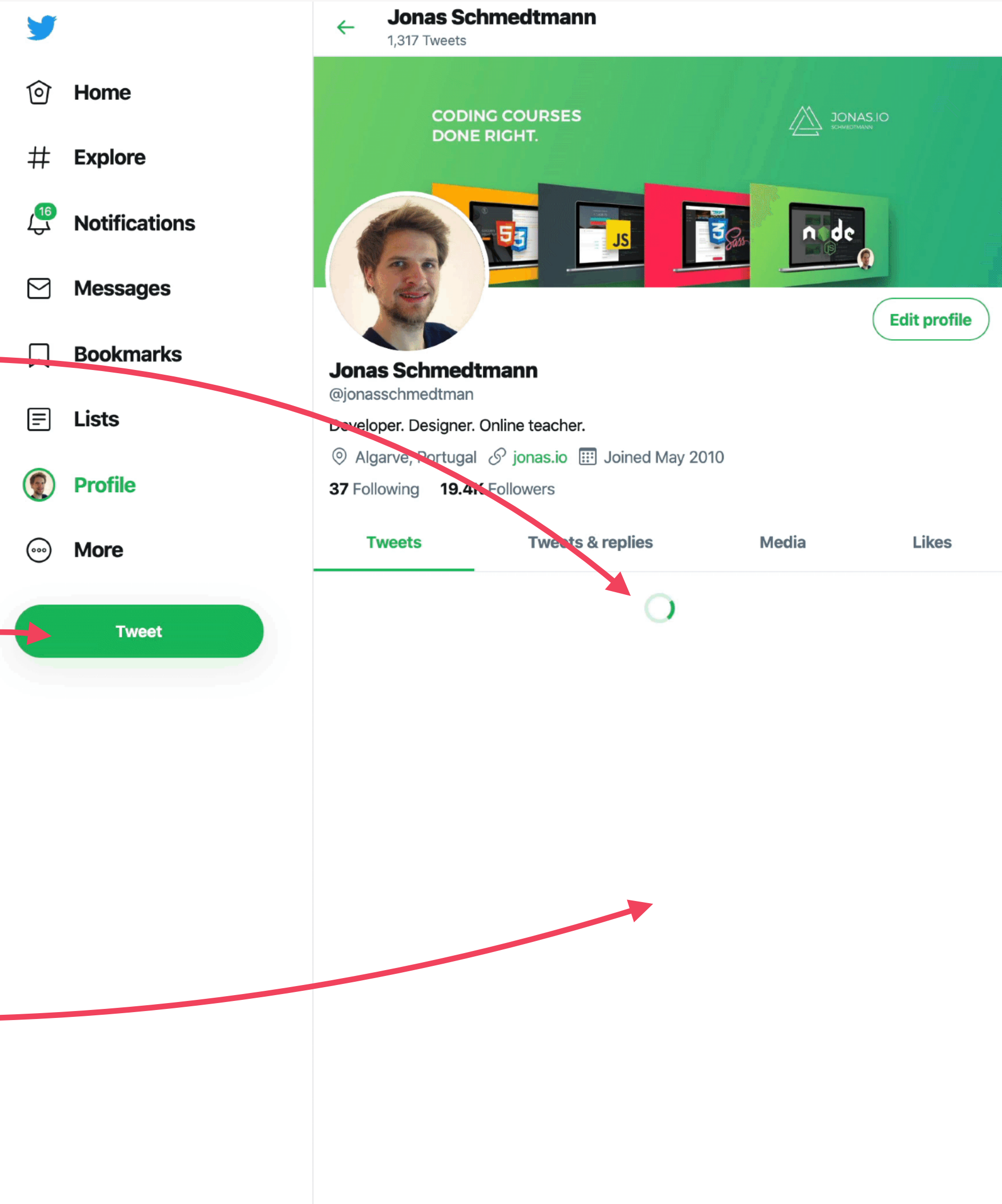


# EXAMPLE OF DYNAMIC EFFECTS / WEB APPLICATION

Show spinner + loading data in the background

Show tweet box after clicking

Display tweets after loading data

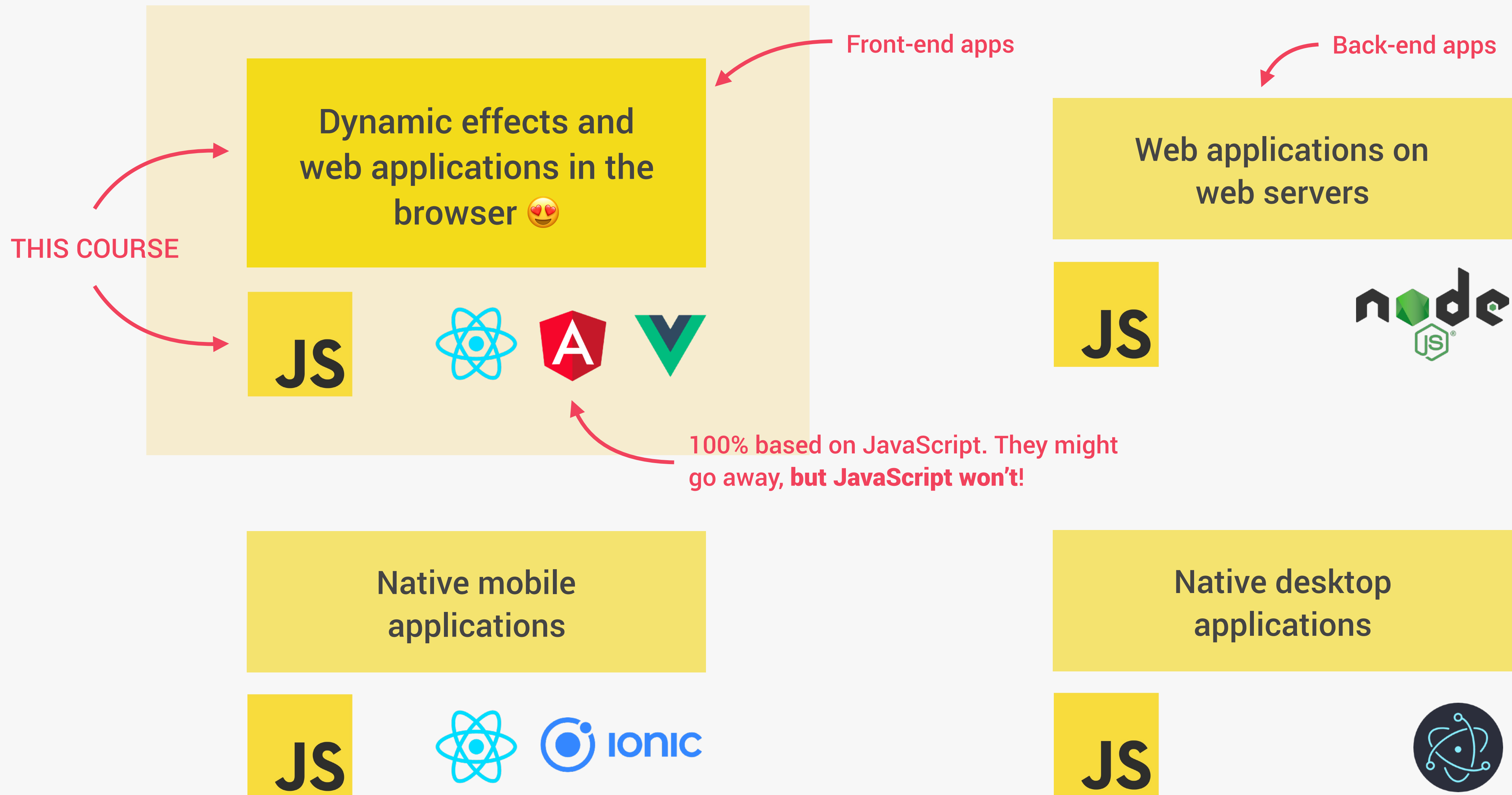


Display user info on hover

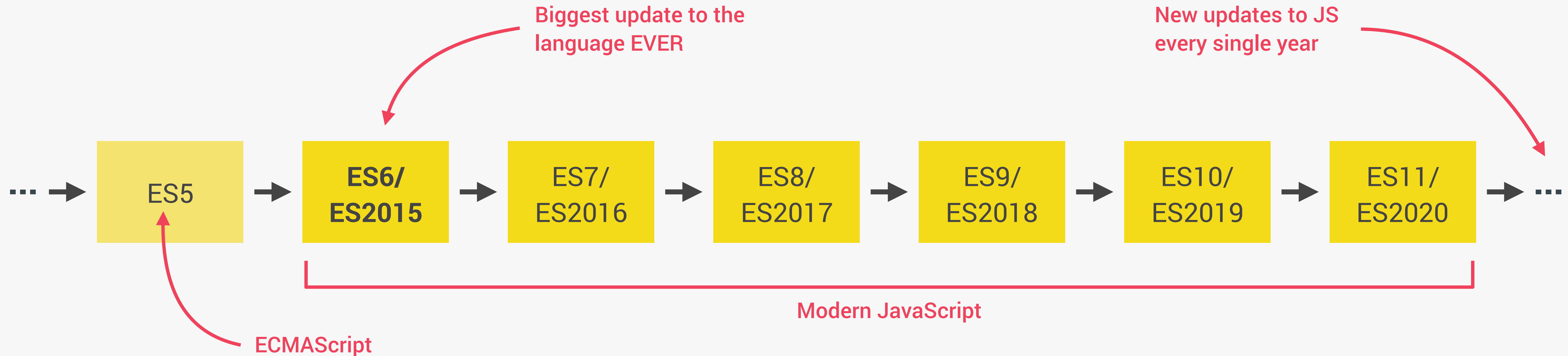
Show spinner + loading data in the background

Show data after loading

# THERE IS NOTHING YOU CAN'T DO WITH JAVASCRIPT (WELL, ALMOST...)



# JAVASCRIPT RELEASES... (MORE ABOUT THIS LATER)



🔥 Learn **modern JavaScript from the beginning**, but without forgetting the older parts!

🎉 Let's finally get started!







JONAS.IO  
SCHMEDTMANN

# THE COMPLETE JAVASCRIPT COURSE

FROM ZERO TO EXPERT!



@JONASSCHMEDTMAN

SECTION

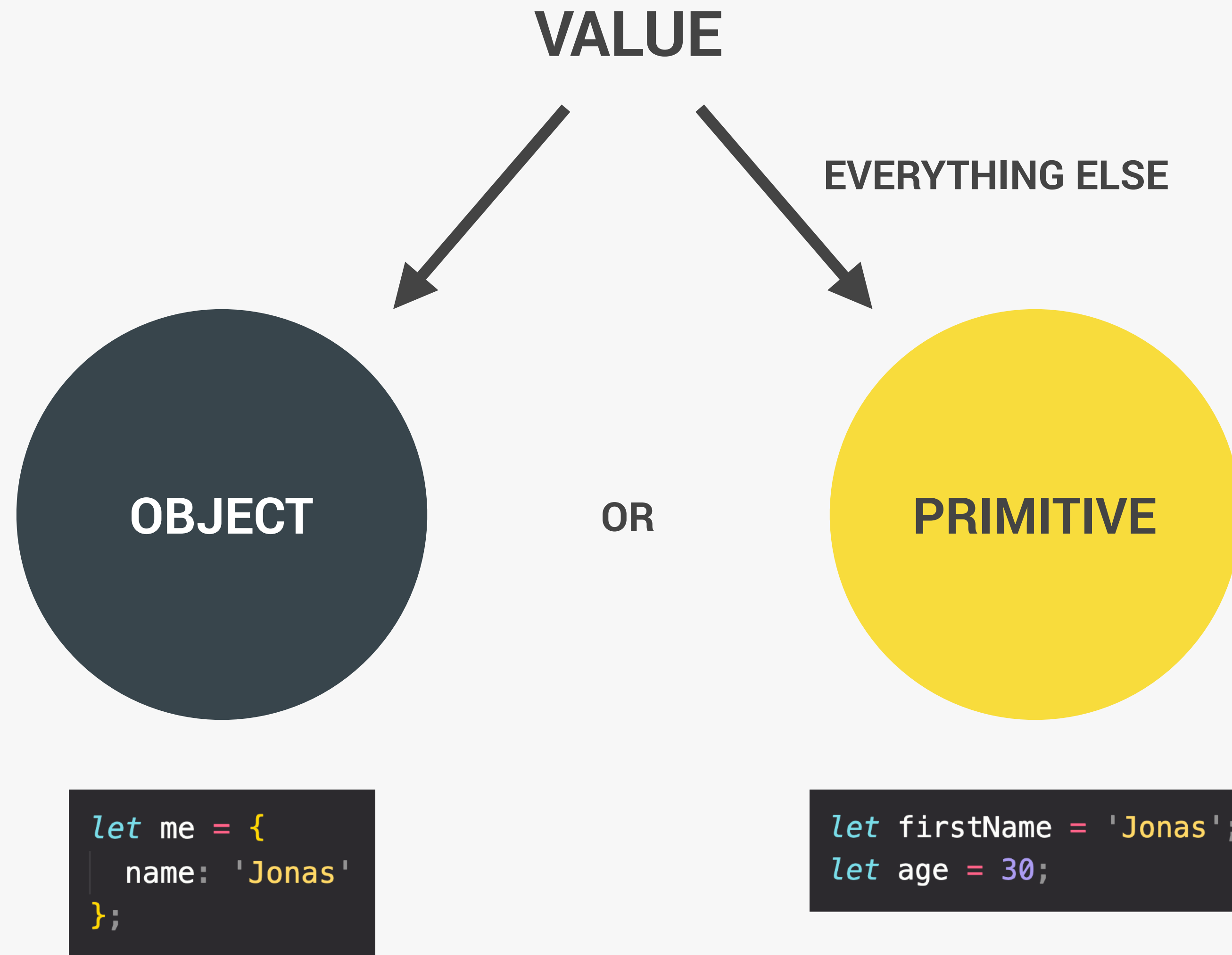
JAVASCRIPT FUNDAMENTALS – PART 1

LECTURE

DATA TYPES

JS

# OBJECTS AND PRIMITIVES





# 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!





JONAS.IO  
SCHMEDTMANN

# THE COMPLETE JAVASCRIPT COURSE

FROM ZERO TO EXPERT!



@JONASSCHMEDTMAN

SECTION

JAVASCRIPT FUNDAMENTALS - PART 1

LECTURE

BOOLEAN LOGIC





# BASIC BOOLEAN LOGIC: THE AND, OR & NOT OPERATORS

A AND B

"Sarah has a driver's license  
**AND** good vision"

Possible values

|   |       |       |
|---|-------|-------|
|   | A     |       |
|   | TRUE  | FALSE |
| B | TRUE  | TRUE  |
|   | FALSE | FALSE |

Results of operation, depending on 2 variables

👉  
**true** when **ALL** are **true**

No matter how many variables

A OR B

"Sarah has a driver's license  
**OR** good vision"

A

|   |       |       |
|---|-------|-------|
|   | A     |       |
|   | TRUE  | FALSE |
| B | TRUE  | TRUE  |
|   | FALSE | TRUE  |

👉  
**true** when **ONE** is **true**

NOT A, NOT B



Inverts **true/false** value

👉 **EXAMPLE:**

A: Sarah has a driver's license

B: Sarah has good vision

Boolean variables that can be either TRUE or FALSE

# AN EXAMPLE 🧑💻

## BOOLEAN VARIABLES

👉 A: Age is greater or equal 20

false

👉 B: Age is less than 30

true

age = 16

## LET'S USE OPERATORS!

👉 !A

false

true

👉 A AND B

false

true

false

👉 A OR B

false

true

true

👉 !A AND B

true

true

true

👉 A OR !B

false

false

false

|   |       | A     |       |
|---|-------|-------|-------|
| B | AND   | TRUE  | FALSE |
|   | TRUE  | TRUE  | FALSE |
|   | FALSE | FALSE | FALSE |

|   |       | A    |       |
|---|-------|------|-------|
| B | OR    | TRUE | FALSE |
|   | TRUE  | TRUE | TRUE  |
|   | FALSE | TRUE | FALSE |







JONAS.IO  
SCHMEDTMANN

# THE COMPLETE JAVASCRIPT COURSE

FROM ZERO TO EXPERT!



@JONASSCHMEDTMAN

SECTION

JAVASCRIPT FUNDAMENTALS - PART 1

LECTURE

JAVASCRIPT RELEASES: ES5, ES6+  
AND ESNEXT

JS

# A BRIEF HISTORY OF JAVASCRIPT

1995

- 👉 Brendan Eich creates the **very first version of JavaScript in just 10 days**. It was called Mocha, but already had many fundamental features of modern JavaScript!



1996

- 👉 Mocha changes to LiveScript and then to JavaScript, in order to attract Java developers. However, **JavaScript has almost nothing to do with Java** 🙅

- 👉 Microsoft launches IE, **copying JavaScript from Netscape** and calling it JScript;



1997

- 👉 With a need to standardize the language, ECMA releases ECMAScript 1 (ES1), the first **official standard for JavaScript** (ECMAScript is the standard, JavaScript the language in practice);



2009

- 👉 ES5 (ECMAScript 5) is released with lots of great new features;

2015

- 👉 ES6/ES2015 (ECMAScript 2015) was released: **the biggest update to the language ever!**

- 👉 ECMAScript changes to an **annual release cycle** in order to ship less features per update 🙏

2016 – ∞

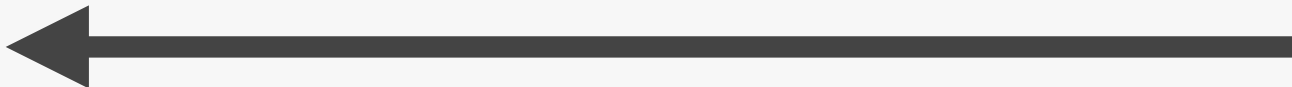
- 👉 Release of ES2016 / ES2017 / ES2018 / ES2019 / ES2020 / ES2021 / ... / ES2089 😄



# BACKWARDS COMPATIBILITY: DON'T BREAK THE WEB!

```
// ES1 Code
function add(n) {
  var x = 5 + add.arguments[0];
  return x;
}
```

1997



BACKWARDS  
COMPATIBLE

Modern JavaScript  
Engine

2020

DON'T BREAK THE WEB!

- 👉 Old features are **never** removed;
- 👉 Not really new versions, just **incremental updates** (releases)
- 👉 Websites keep working **forever!**

Modern JavaScript  
Engine

2020



NOT FORWARDS  
COMPATIBLE

```
// ES2089 Code 🤔
c int add n <=> int 5 + n
```

2089



# HOW TO USE MODERN JAVASCRIPT TODAY



**During development:** Simply use the latest Google Chrome!



**During production:** Use Babel to transpile and polyfill your code (converting back to ES5 to ensure browser compatibility for all users).

<http://kangax.github.io/compat-table>

ES5

- ➡ Fully supported in all browsers (down to IE 9 from 2011);
- ➡ Ready to be used today 👍

BABEL

ES6/ES2015



ES2020

- ➡ **ES6+:** Well supported in all **modern** browsers;
- ➡ No support in **older** browsers;
- ➡ Can use **most** features in production with transpiling and polyfilling 😊

ES2021 – ∞

- ➡ **ESNext:** Future versions of the language (new feature proposals that reach Stage 4);
- ➡ Can already use **some** features in production with transpiling and polyfilling.

Will add new videos

(As of 2020)

# MODERN JAVASCRIPT FROM THE BEGINNING



Learn **modern JavaScript from the beginning!**



But, also learn how some things used to be done **before** modern JavaScript (e.g. `const` & `let` vs `var` and function constructors vs ES6 `class`).

## 3 reasons why we should not forget the Good Ol' JavaScript:



You will better understand how JavaScript actually works;



Many tutorials and code you find online today are still in ES5;



When working on old codebases, these will be written in ES5.





# JAVASCRIPT FUNDAMENTALS – PART 2



JONAS.IO  
SCHMEDTMANN

# THE COMPLETE JAVASCRIPT COURSE

FROM ZERO TO EXPERT!



@JONASSCHMEDTMAN

SECTION

JAVASCRIPT FUNDAMENTALS - PART 2

LECTURE

FUNCTIONS CALLING OTHER  
FUNCTIONS



# CALLING A FUNCTION INSIDE A FUNCTION: DATA FLOW

```
const cutPieces = function (fruit) {  
  return fruit * 4;  
};  
  
const fruitProcessor = function (apples, oranges) {  
  
  const applePieces = cutPieces(apples);  
  const orangePieces = cutPieces(oranges);  
  
  const juice = `Juice with ${applePieces} pieces of  
apple and ${orangePieces} pieces of orange.`;  
  return juice;  
};  
  
console.log(fruitProcessor(2, 3));
```

The diagram illustrates the data flow in the provided JavaScript code. Red arrows represent the flow of data from the console log call to the fruitProcessor function, then to the cutPieces function, and back to the fruitProcessor function. A yellow arrow shows the flow of data from the console log call to the fruitProcessor function.

- The console log call `console.log(fruitProcessor(2, 3));` has two arguments: `2` (highlighted in a red box) and `3` (highlighted in a yellow box).
- The `fruitProcessor` function is called with `apples = 2` and `oranges = 3`. The value `2` is passed to `cutPieces` to calculate `applePieces`, and the value `3` is passed to `cutPieces` to calculate `orangePieces`.
- The `cutPieces` function is called with `fruit = 2` and `fruit = 3`. The value `2` is passed to `cutPieces` to calculate `applePieces`, and the value `3` is passed to `cutPieces` to calculate `orangePieces`.
- The `cutPieces` function returns `fruit * 4`. For `applePieces`, the result is `2 * 4 = 8`. For `orangePieces`, the result is `3 * 4 = 12`.
- The `fruitProcessor` function returns the string `Juice with 8 pieces of apple and 12 pieces of orange.``.







JONAS.IO  
SCHMEDTMANN

# THE COMPLETE JAVASCRIPT COURSE

FROM ZERO TO EXPERT!



@JONASSCHMEDTMAN

SECTION

JAVASCRIPT FUNDAMENTALS - PART 2

LECTURE

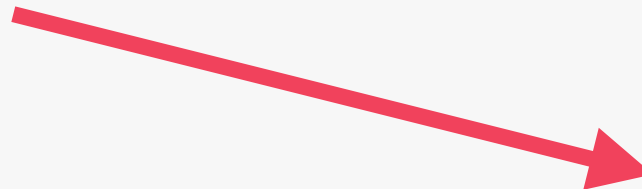
REVIEWING FUNCTIONS

JS

# FUNCTIONS REVIEW: 3 DIFFERENT FUNCTION TYPES

## 👉 Function declaration

Function that can be used before it's declared



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

## 👉 Function expression

Essentially a function *value* stored in a variable



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

## 👉 Arrow function

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

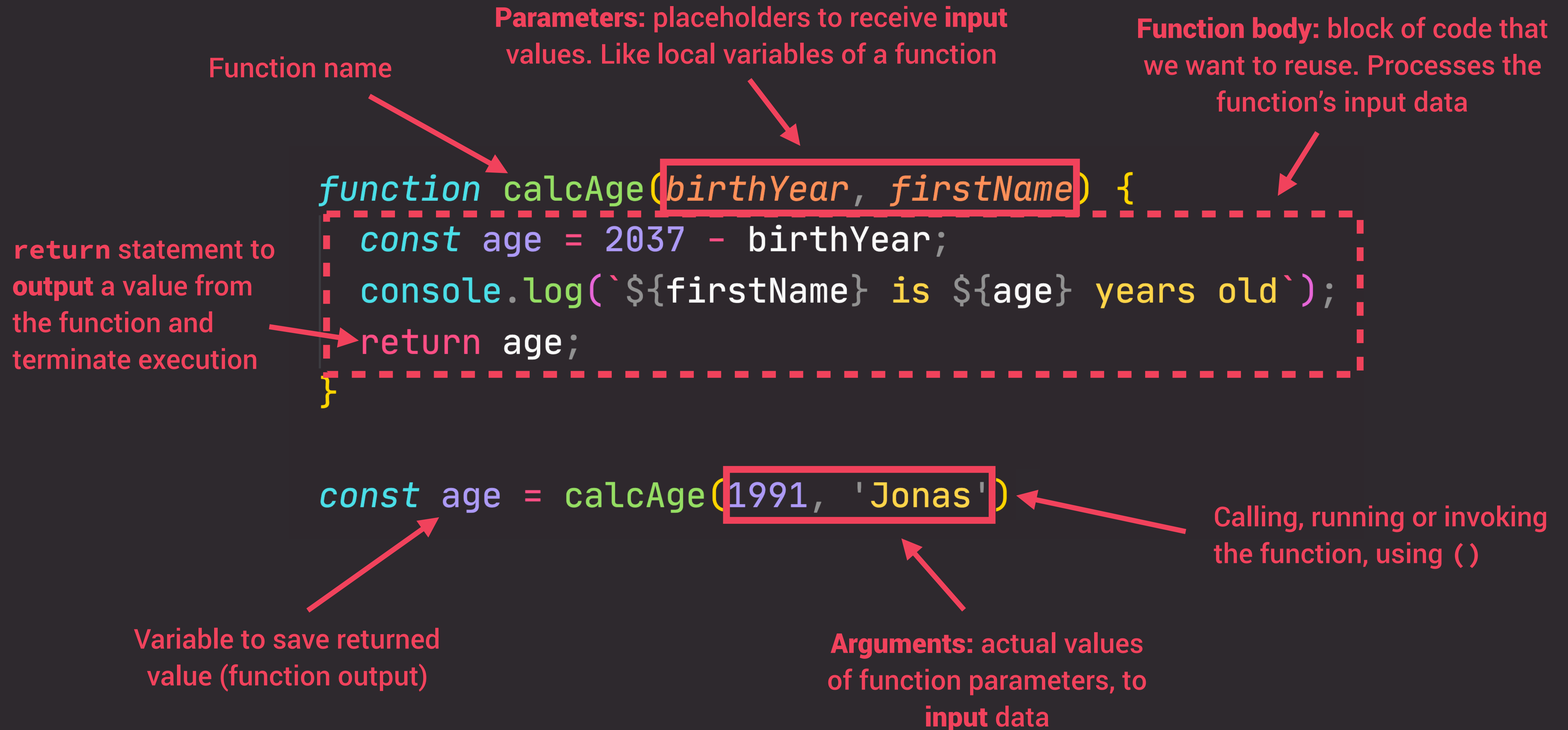


```
const calcAge = birthYear => 2037 - birthYear;
```

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



# FUNCTIONS REVIEW: ANATOMY OF A FUNCTION





# DEVELOPER SKILLS & EDITOR SETUP





JONAS.IO  
SCHMEDTMANN

# THE COMPLETE JAVASCRIPT COURSE

FROM ZERO TO EXPERT!



@JONASSCHMEDTMAN

SECTION

DEVELOPER SKILLS & EDITOR SETUP

LECTURE

LEARNING HOW TO CODE

JS



# HOW TO FAIL 🤔 AT LEARNING HOW TO CODE



John  
(not actually...)

- ✦ He **didn't have a clear goal** at the beginning of his journey
- ✦ He started by watching courses and reading tutorials, but he would just **copy the code without caring how it works**. Sometimes he would just copy and paste code!
- ✦ He **didn't reinforce** what he was learning by doing small challenges or taking notes
- ✦ He **didn't practice coding**, and didn't come up with his own project ideas
- ✦ He **quickly became frustrated** when his code was not perfectly clean or efficient
- ✦ He **lost motivation** because he thought he could never know everything
- ✦ He was **learning in isolation**
- ✦ After finishing a couple of courses, **he thought he now was a web developer** and could start applying to jobs. But he couldn't even build an app on his own!

# HOW TO SUCCEED 🎉 AT LEARNING HOW TO CODE

💣 He **didn't have a clear goal** at the beginning of his journey



**FIX**

- 👍 Set a **specific, measurable, realistic** and **time-based** goal
- 👍 Know exactly **why** you are learning to code: Switching careers? Finding a better job?
- 👍 **Imagine a big project** you want to be able to build!
- 👍 Research technologies you need and then learn them

💣 He would just **copy the code without caring how it works**. Sometimes he would just copy and paste code!



**FIX**

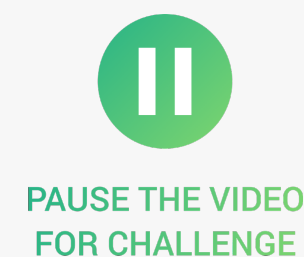
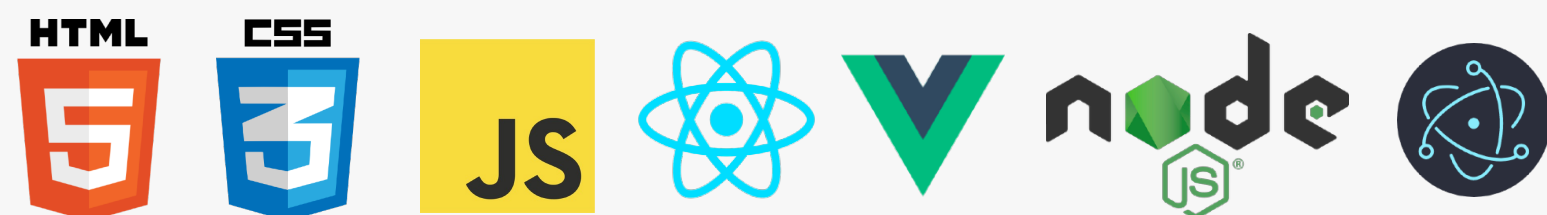
- 👍 Understand the code that you're studying and typing
- 👍 **Always type the code**, don't copy-paste!

💣 He **didn't reinforce** what he was learning by doing small challenges or taking notes



**FIX**

- 👍 After you learn a new feature or concept, **use it immediately**
- 👍 Take notes
- 👍 **Challenge yourself** and practice with small coding exercises and challenges
- 👍 Don't be in a hurry to complete the course fast!





# HOW TO SUCCEED 🎉 AT LEARNING HOW TO CODE

💣 He **didn't practice coding**, and didn't come up with his own project ideas



- 👍 Practicing on your own is the most important thing to do
- 👍 **This is NOT optional!** Without **practice outside of courses**, you won't go anywhere!
- 👍 Come up with your own project ideas or copy popular sites or applications, or just parts of them in the beginning
- 👍 Don't be stuck in "tutorial hell"

💣 He **quickly became frustrated** when his code was not perfectly clean or efficient



- 👍 **Don't get stuck** trying to write the perfect code!
- 👍 Just write tons of code, **no matter the quality!**
- 👍 Clean and efficient code will come with time
- 👍 You can always refactor code later

💣 He **lost motivation** because he thought he could never know everything



- 👍 Embrace the fact that **you will never you know everything**
- 👍 Just focus on what you need to achieve your goal!



**getify**  
@getify



20+ yrs dev exp, 8 books w/ 100k+ copies sold, 300k+ hours watched of my videos, 4k+ taught in person...

And you know what? I still struggle to get my code to work and it's still a tedious slog. And my code still confuses me the next day.

You're not alone in these struggles.

♡ 6,015 3:33 PM - Mar 10, 2018



# HOW TO SUCCEED 🎉 AT LEARNING HOW TO CODE

💣 He was **learning in isolation**



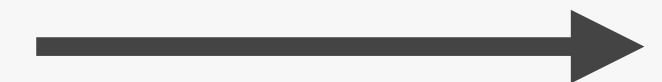
- 👍 Explain new concepts to other people. If you can explain it, you truly understand it!
- 👍 Share your goals to make **yourself accountable**
- 👍 Share your learning progress with the web dev community (#100DaysOfCode, , #CodeNewbie, #webdev, etc.)

💣 After finishing a couple of courses, **he thought he now was a web developer** and could start applying to jobs



- 👍 The **biggest misconception** that people have!
- 👍 Courses are an amazing starting point, but are only the **beginning of your journey!**

**NEXT SLIDE**



# LEARNING HOW TO CODE IS HARD, BUT YOU CAN DO IT!









JONAS.IO  
SCHMEDTMANN

# THE COMPLETE JAVASCRIPT COURSE

FROM ZERO TO EXPERT!



@JONASSCHMEDTMAN

SECTION

DEVELOPER SKILLS & EDITOR SETUP

LECTURE

HOW TO THINK LIKE A DEVELOPER:  
BECOME A PROBLEM SOLVER!





# HOW TO FAIL 🤔 AT SOLVING PROBLEMS



John can  
code now 😊

## WHENEVER JOHN ENCOUNTERS A PROBLEM:

- 💥 He jumps at the problem **without much thinking**
- 💥 He implements his solution in an **unstructured way**
- 💥 He **gets stressed out** when things don't work
- 💥 He is **too proud to research** solutions

↓ **FIX**

- 👍 **Stay calm and slow down**, don't just jump at a problem without a plan
- 👍 Take a very **logical and rational approach** (programming is just logic, in the end...)
- 👍 Use my **4-step framework** to solve any problem

→  
**NEXT SLIDE**

👉 **Example:** *In an array of GPS coordinates, find the two closest points*



# 4 STEPS TO SOLVE ANY PROBLEM

1

Make sure you 100% understand the problem. **Ask the right questions** to get a clear picture of the problem

## EXAMPLE

💬 Project Manager: *"We need a function that reverses **whatever** we pass into it"*

1

- 👉 What does "whatever" even mean in this context? What should be reversed? **Answer:** Only strings, numbers, and arrays make sense to reverse...
- 👉 What to do if something else is passed in?
- 👉 What should be returned? Should it always be a string, or should the type be the same as passed in?
- 👉 How to recognize whether the argument is a number, a string, or an array?
- 👉 How to reverse a number, a string, and an array?

# 4 STEPS TO SOLVE ANY PROBLEM

1

Make sure you 100% understand the problem. **Ask the right questions** to get a clear picture of the problem



2

**Divide and conquer:** Break a big problem into smaller sub-problems.

## EXAMPLE

💬 Project Manager: *"We need a function that reverses whatever we pass into it"*

2

### SUB-PROBLEMS:

- 👉 Check if argument is a number, a string, or an array
- 👉 Implement reversing a number
- 👉 Implement reversing a string
- 👉 Implement reversing an array
- 👉 Return reversed value

Looks like a task list that we need to implement

# 4 STEPS TO SOLVE ANY PROBLEM

1

Make sure you 100% understand the problem. **Ask the right questions** to get a clear picture of the problem



2

**Divide and conquer:** Break a big problem into smaller sub-problems.



3

Don't be afraid to do as much **research** as you have to

## EXAMPLE

💬 Project Manager: *"We need a function that reverses whatever we pass into it"*

3

- 👉 How to check if a value is a number in JavaScript?
- 👉 How to check if a value is a string in JavaScript?
- 👉 How to check if a value is an array in JavaScript?
- 👉 How to reverse a number in JavaScript?
- 👉 How to reverse a string in JavaScript?
- 👉 How to reverse an array in JavaScript?





# 4 STEPS TO SOLVE ANY PROBLEM

1

Make sure you 100% understand the problem. **Ask the right questions** to get a clear picture of the problem



2

**Divide and conquer:** Break a big problem into smaller sub-problems.



3

Don't be afraid to do as much **research** as you have to



4

For bigger problems, **write pseudo-code** before writing the actual code

## EXAMPLE

💬 Project Manager: *"We need a function that reverses whatever we pass into it"*

4

```
function reverse(value)
  if value type != string || != number || != array
    return value

  if value type == string
    reverse string
  if value type == number
    reverse number
  if value type == array
    reverse array

  return reversed value
```





JONAS.IO  
SCHMEDTMANN

# THE COMPLETE JAVASCRIPT COURSE

FROM ZERO TO EXPERT!



@JONASSCHMEDTMAN

SECTION

DEVELOPER SKILLS & EDITOR SETUP

LECTURE

DEBUGGING (FIXING ERRORS)

JS



# WHAT IS A SOFTWARE BUG?

- 👉 **Software bug:** Defect or problem in a computer program. Basically, any **unexpected or unintended behavior** of a computer program is a software bug.
- 👉 Bugs are **completely normal** in software development!

- 👉 Previous example: "We need a function that reverses whatever we pass into it"

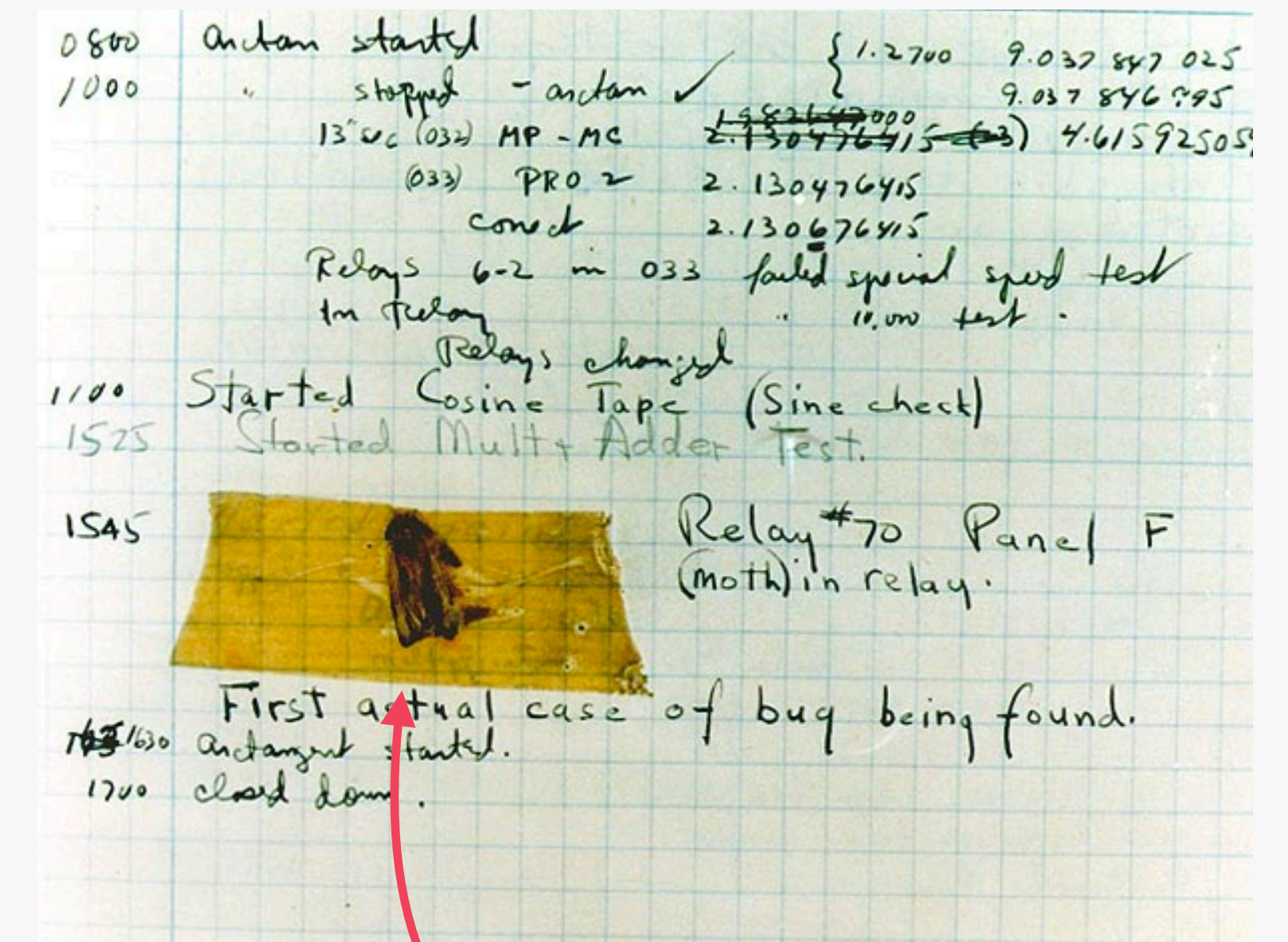
`reverse([1, 3, 5, 7])`



`[5, 1, 7, 3]`

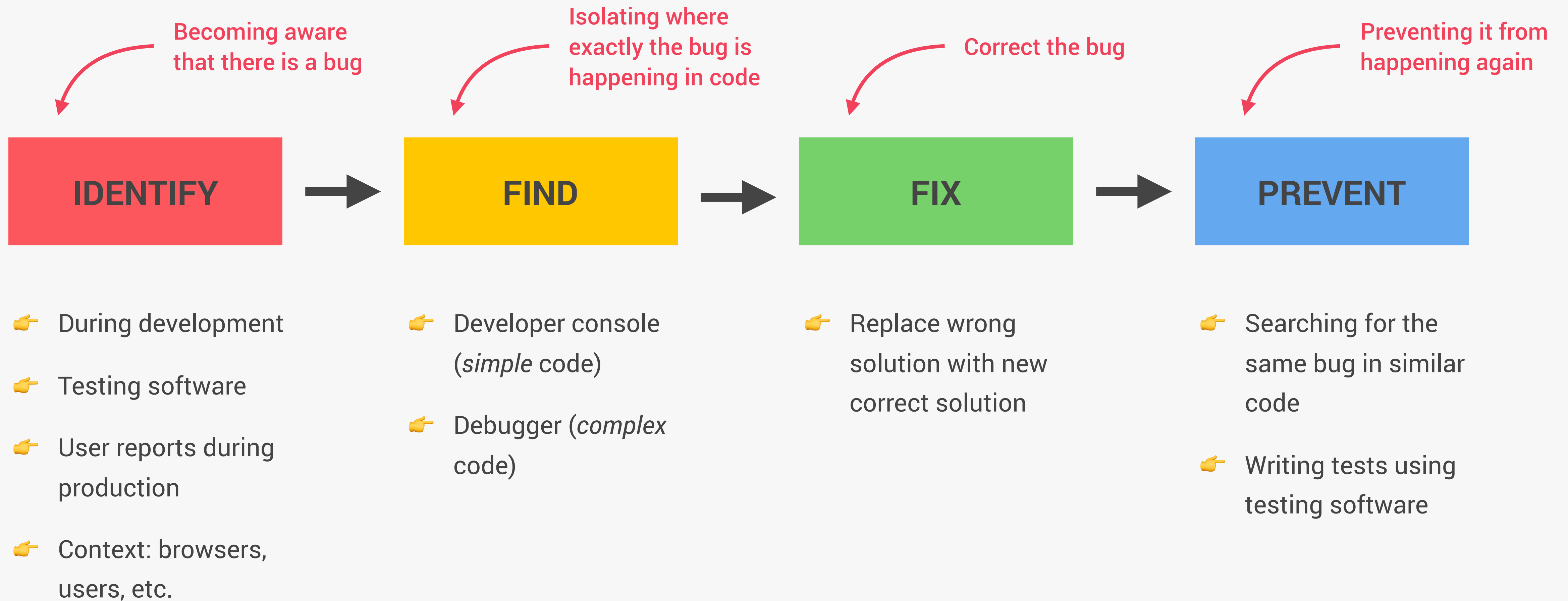
Unexpected result: the array is scrambled, NOT reversed. So there is a bug in the reverse function 🐛

- 👉 **Debugging:** Process of finding, fixing and preventing bugs.



A real bug which was causing an error in Harvard's computer in the 1940s

# THE DEBUGGING PROCESS







# JAVASCRIPT IN THE BROWSER: DOM AND EVENTS FUNDAMENTALS



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

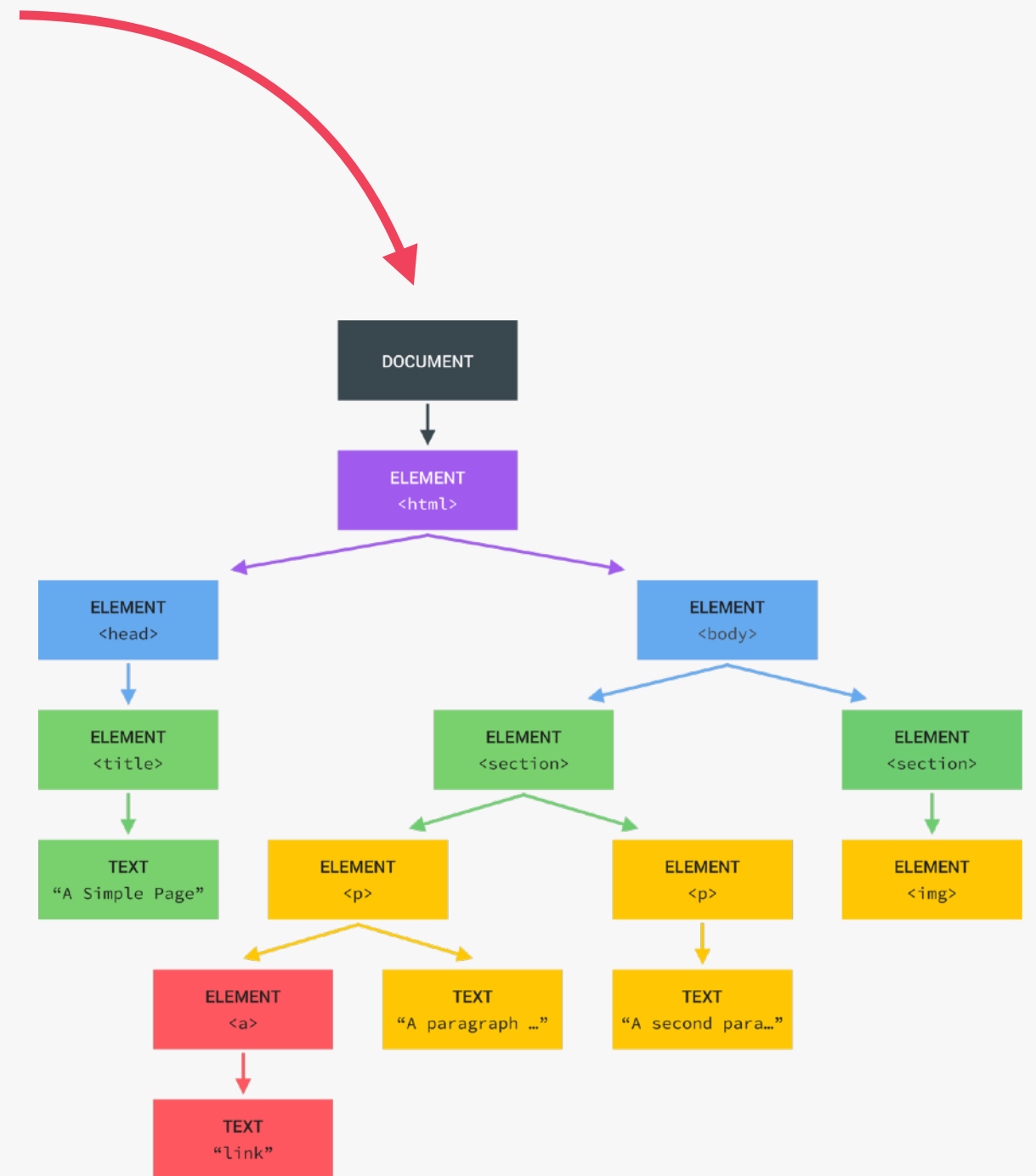
JS

# WHAT IS THE DOM?

DOM

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

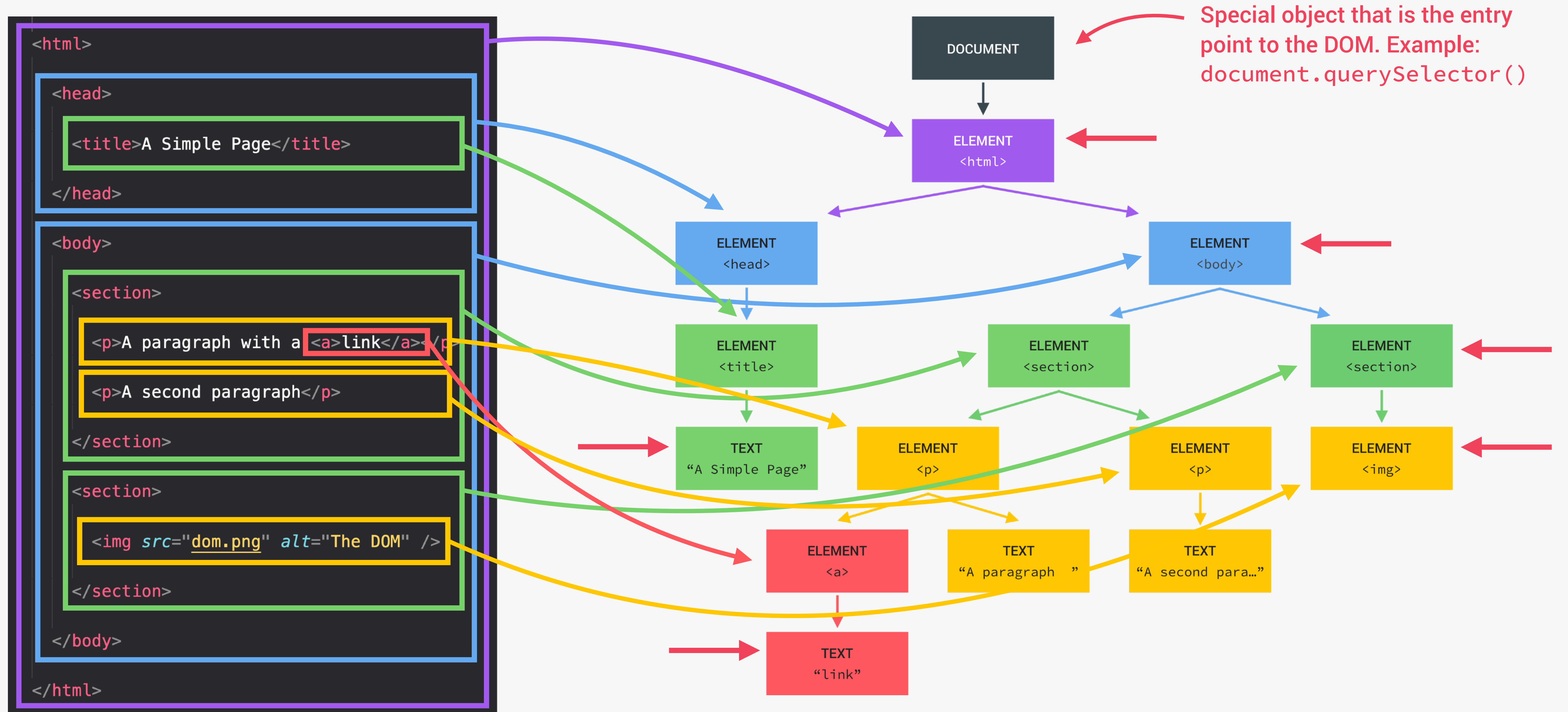
Tree structure, generated by browser on HTML load



Change text, HTML attributes, and even CSS styles



# THE DOM TREE STRUCTURE



DOM !== JAVASCRIPT 🤔

DOM Methods and Properties for DOM Manipulation



JS



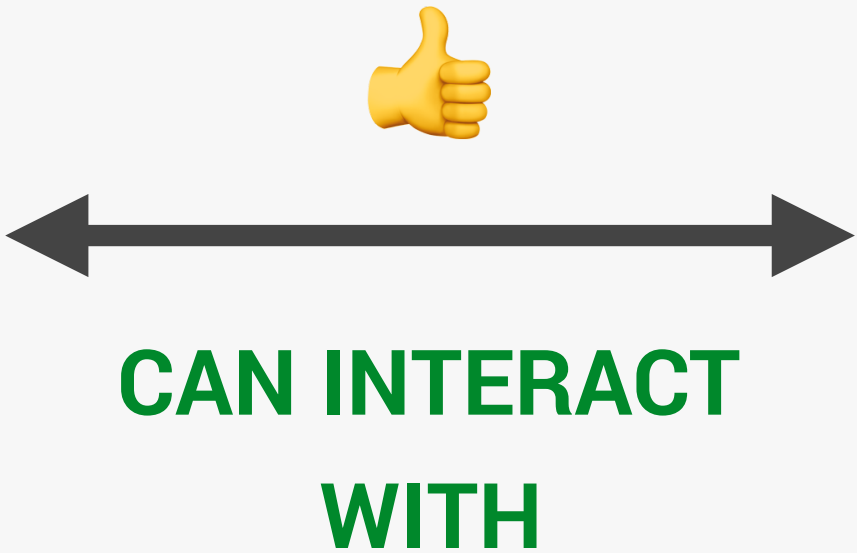
For example  
`document.querySelector()`

API: Application Programming Interface

WEB APIs



- DOM Methods and Properties
- Timers
- Fetch



JS





# HOW JAVASCRIPT WORKS BEHIND THE SCENES



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



# WHAT IS JAVASCRIPT: REVISITED

JAVASCRIPT

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

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 🤔💥😂

JS

# 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

# DECONSTRUCTING THE MONSTER DEFINITION

High-level

Garbage-collected

Interpreted or just-in-time compiled

Multi-paradigm

Prototype-based object-oriented

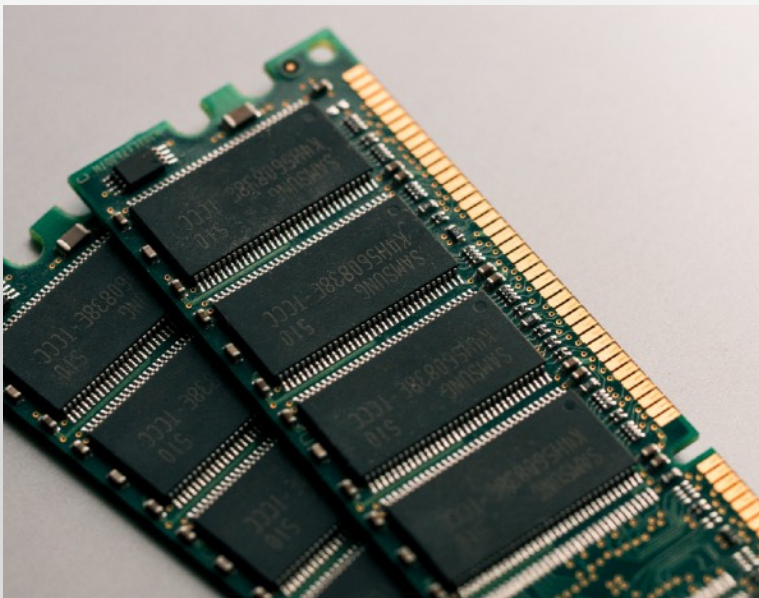
First-class functions

Dynamic

Single-threaded

Non-blocking event loop

👉 Any computer program needs resources:



+



LOW-LEVEL

Developer has to manage resources manually



HIGH-LEVEL

Developer does NOT have to worry, everything happens automatically



# DECONSTRUCTING THE MONSTER DEFINITION

High-level

Garbage-collected

Interpreted or just-in-time compiled

Multi-paradigm

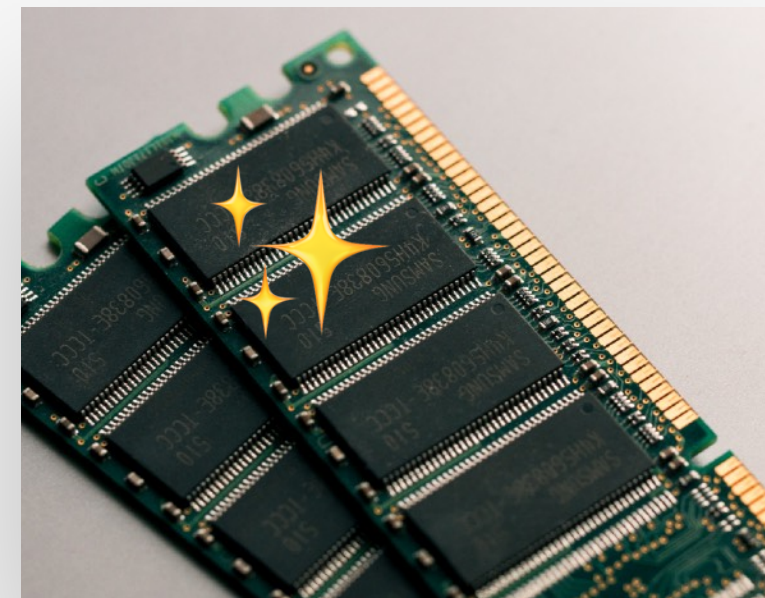
Prototype-based object-oriented

First-class functions

Dynamic

Single-threaded

Non-blocking event loop



Cleaning the memory  
so we don't have to

# 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

```
document.querySelector(".again").addEventListener("click", () => {
  document.querySelector(".message").textContent = "Start guessing...";
  document.querySelector(".number").textContent = "?";
  document.querySelector(".guess").value = "";
  score = 20;
  document.querySelector(".score").textContent = score;
  number = Math.floor(Math.random() * 20) + 1;
});
```

Abstraction over  
0s and 1s

↓  
CONVERT TO MACHINE CODE = COMPILING

```
1101011010111010101110110110010111010101010111101010
01111010101110101001001110101110101011100010101100010
101001001111011101111001110000001110101011110111010
1101001000010100101110101011010101110101011101010010
000011101001001001111010101110101011100101011111010
10010101001001001111010011101010001010101001011010100
11100100010001111010000101011100010100010101110101101
01010010101000101010001110100100101110101001000101011
11101010010111010100010101110101001011101010100101001
111001011101110101011101001010101010101010100101010
01110101101101010100101010111010111010101011100111010
111010100111010100111010110101010101010101011101010
```

Happens inside the  
JavaScript engine

More about this **Later in this Section** 🖱️

# 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

**Array**

Array.prototype.push

Array.prototype.indexOf

Prototype

(Oversimplification!)

Our array  
inherits methods  
from prototype

Built from prototype

```
const arr = [1, 2, 3];  
arr.push(4);  
const hasZero = arr.indexOf(0) > -1;
```

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

👉 Dynamically-typed language:

No data type definitions. Types becomes known at runtime

Data type of variable is automatically changed

```
let x = 23;  
let y = 19;  
x = "Jonas";
```





# 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





# WHAT IS A JAVASCRIPT ENGINE?

JS ENGINE

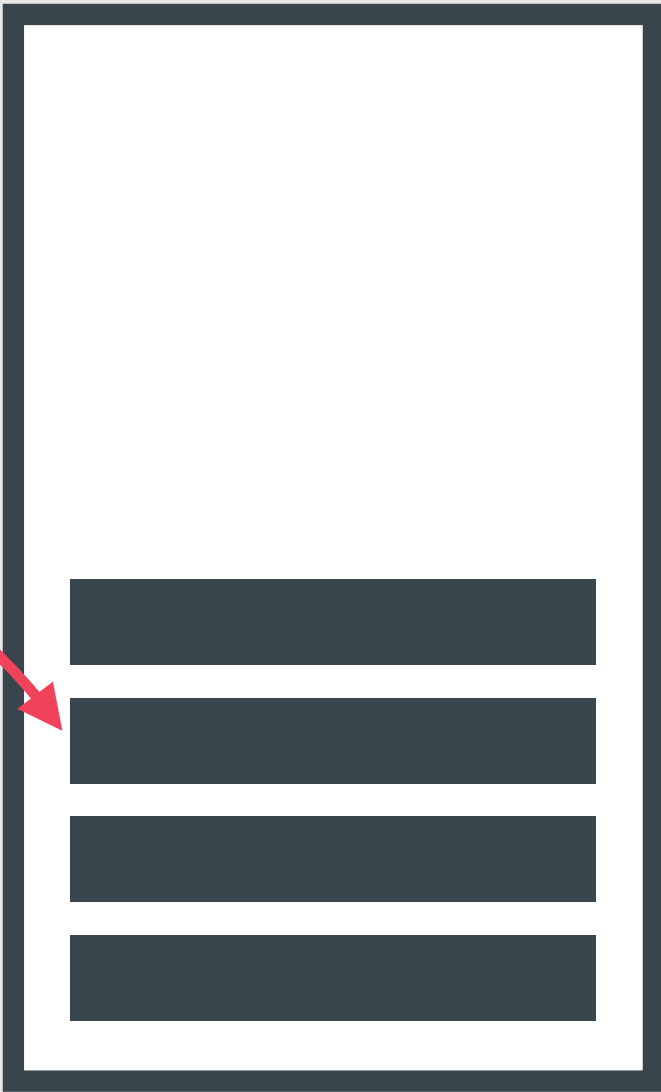
PROGRAM THAT **EXECUTES**  
JAVASCRIPT CODE.



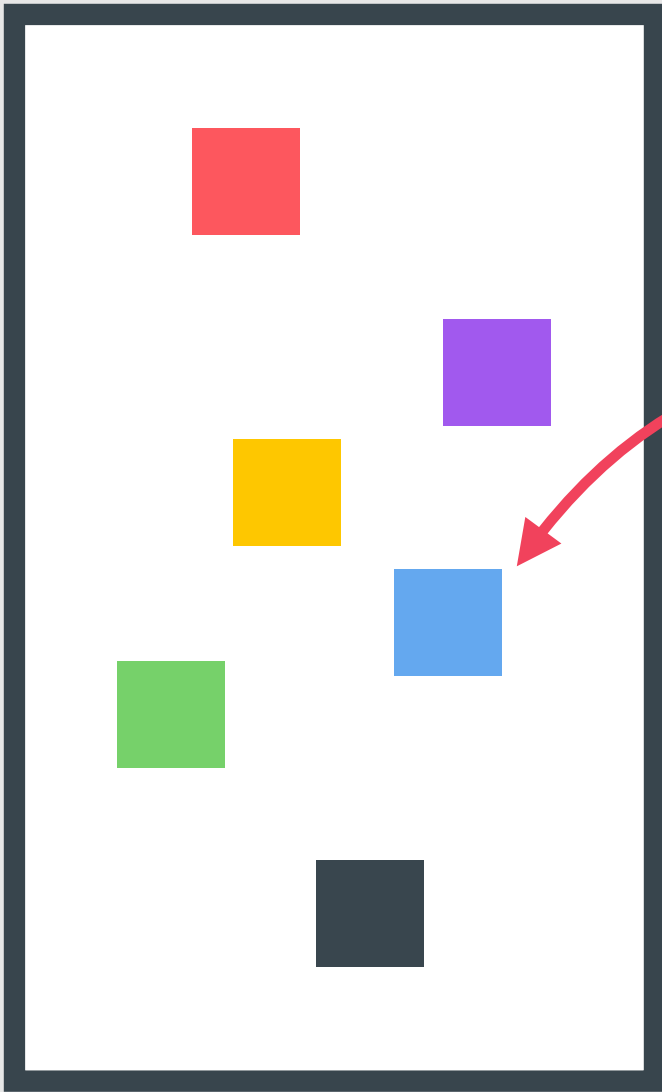
Example: V8 Engine



Execution  
context



CALL STACK



HEAP

Object in  
memory

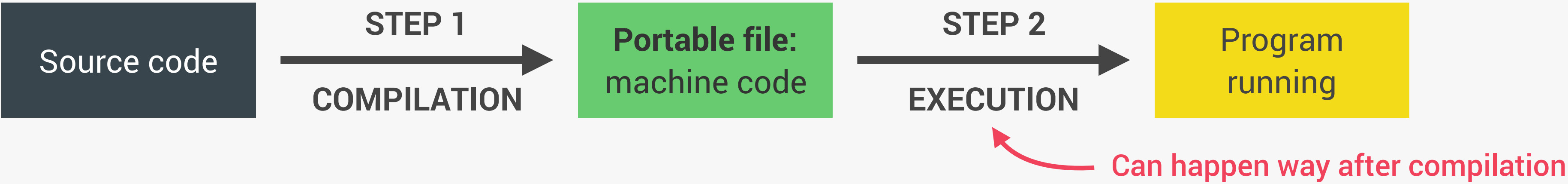
How is it  
compiled?

Where our code  
is executed

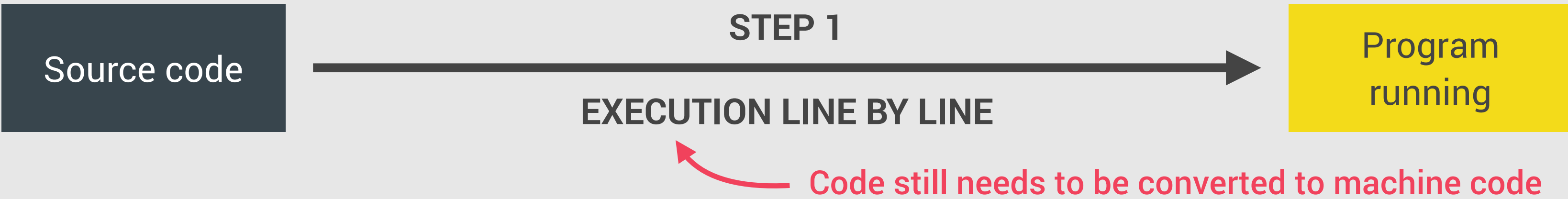
Where objects  
are stored

# COMPUTER SCIENCE SIDENOTE: COMPIRATION VS. INTERPRETATION 🧐

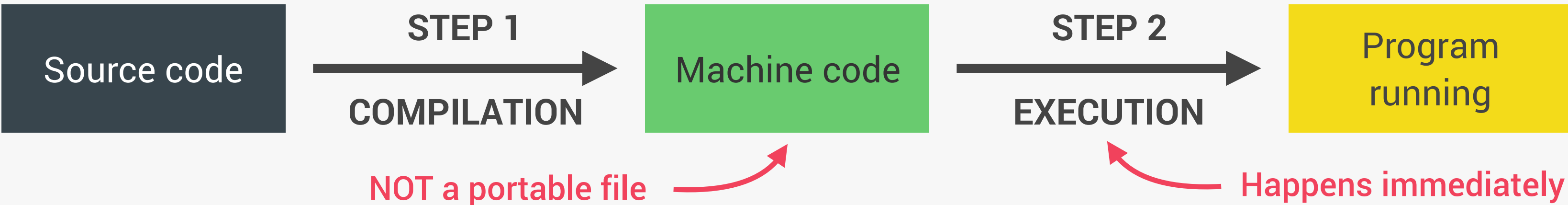
👉 **Compilation:** Entire code is converted into machine code at once, and written to a binary file that can be executed by a computer.



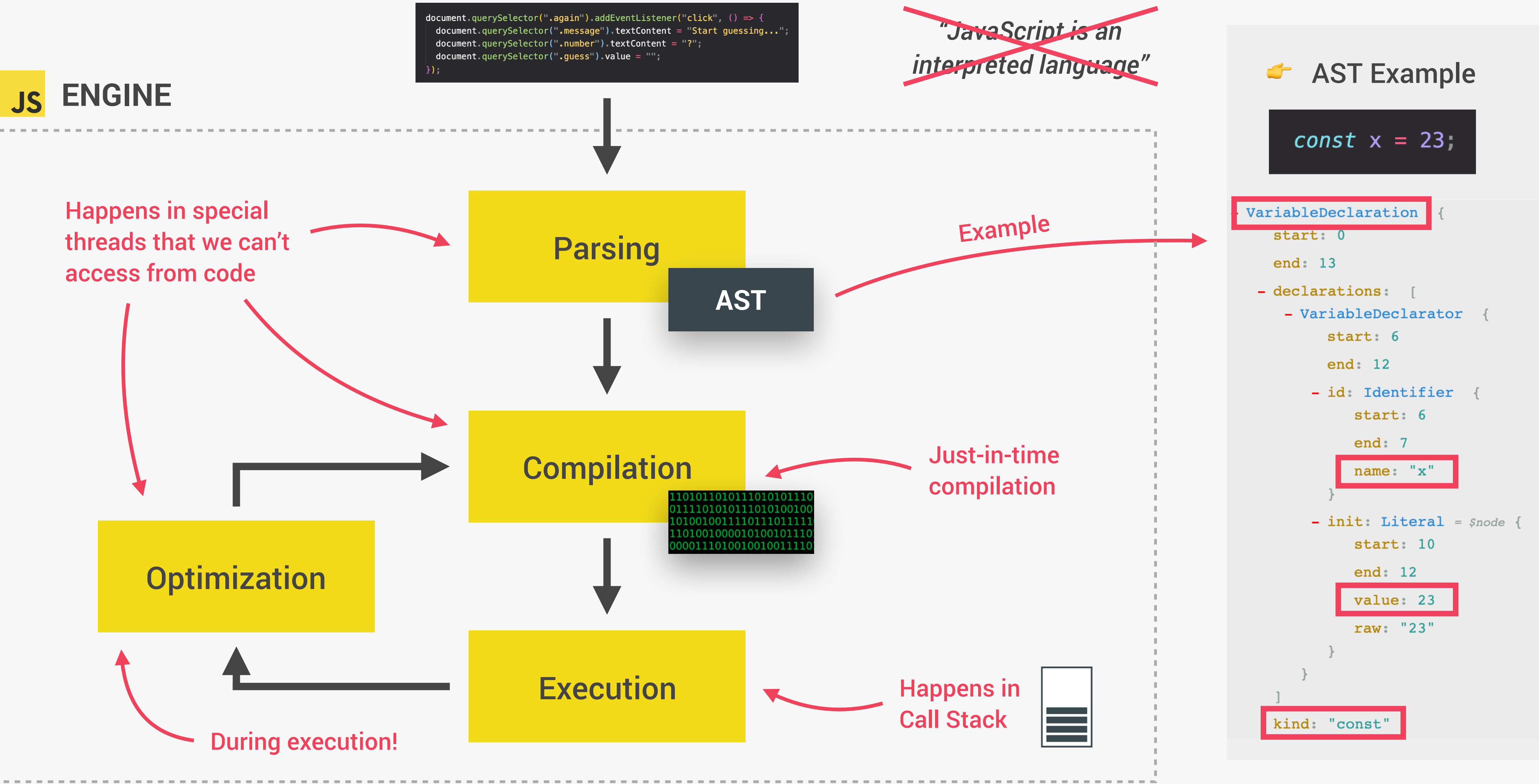
👉 **Interpretation:** Interpreter runs through the source code and executes it line by line.



👉 **Just-in-time (JIT) compilation:** Entire code is converted into machine code at once, then executed immediately.

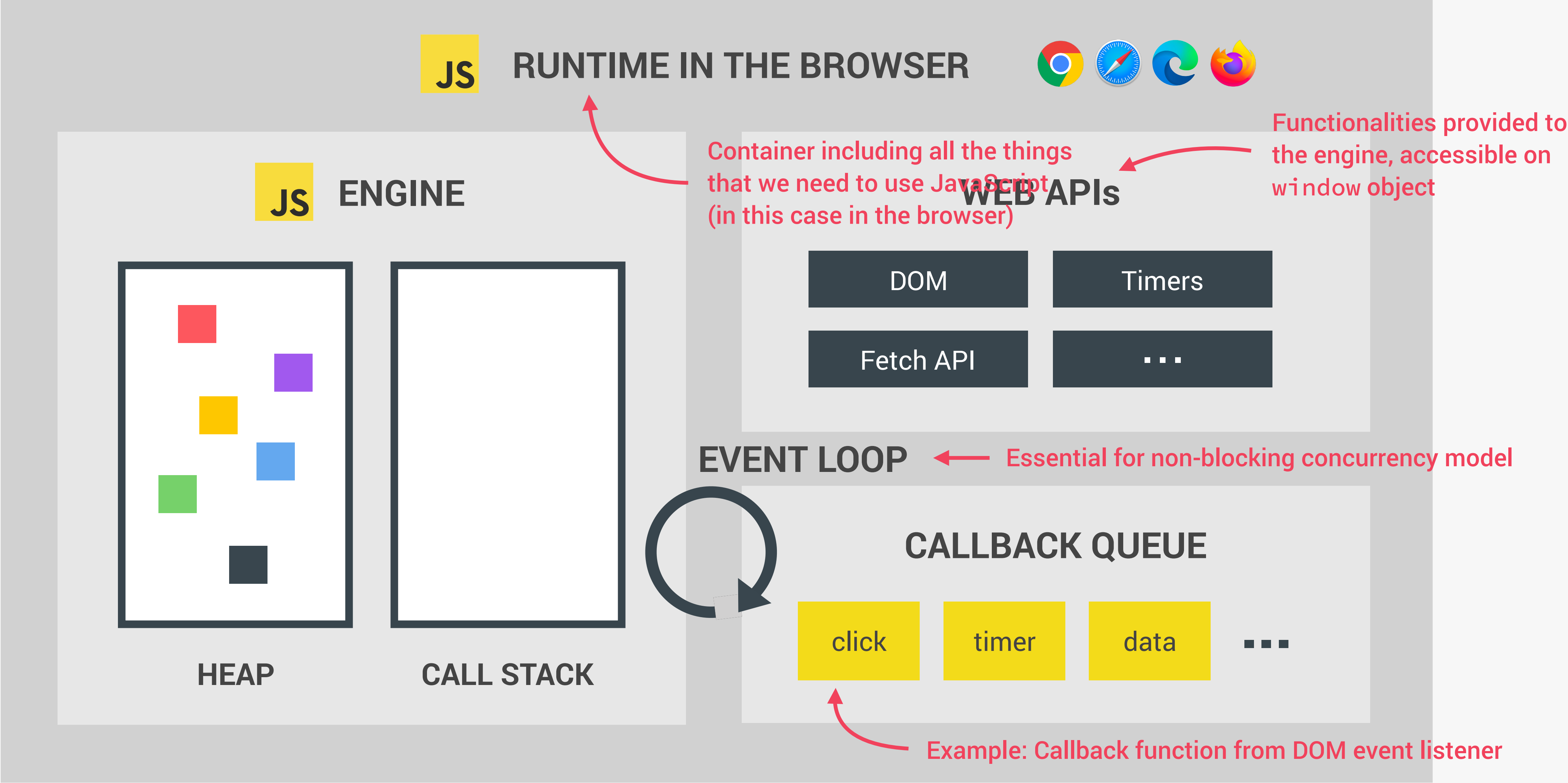


# MODERN JUST-IN-TIME COMPILATION OF JAVASCRIPT

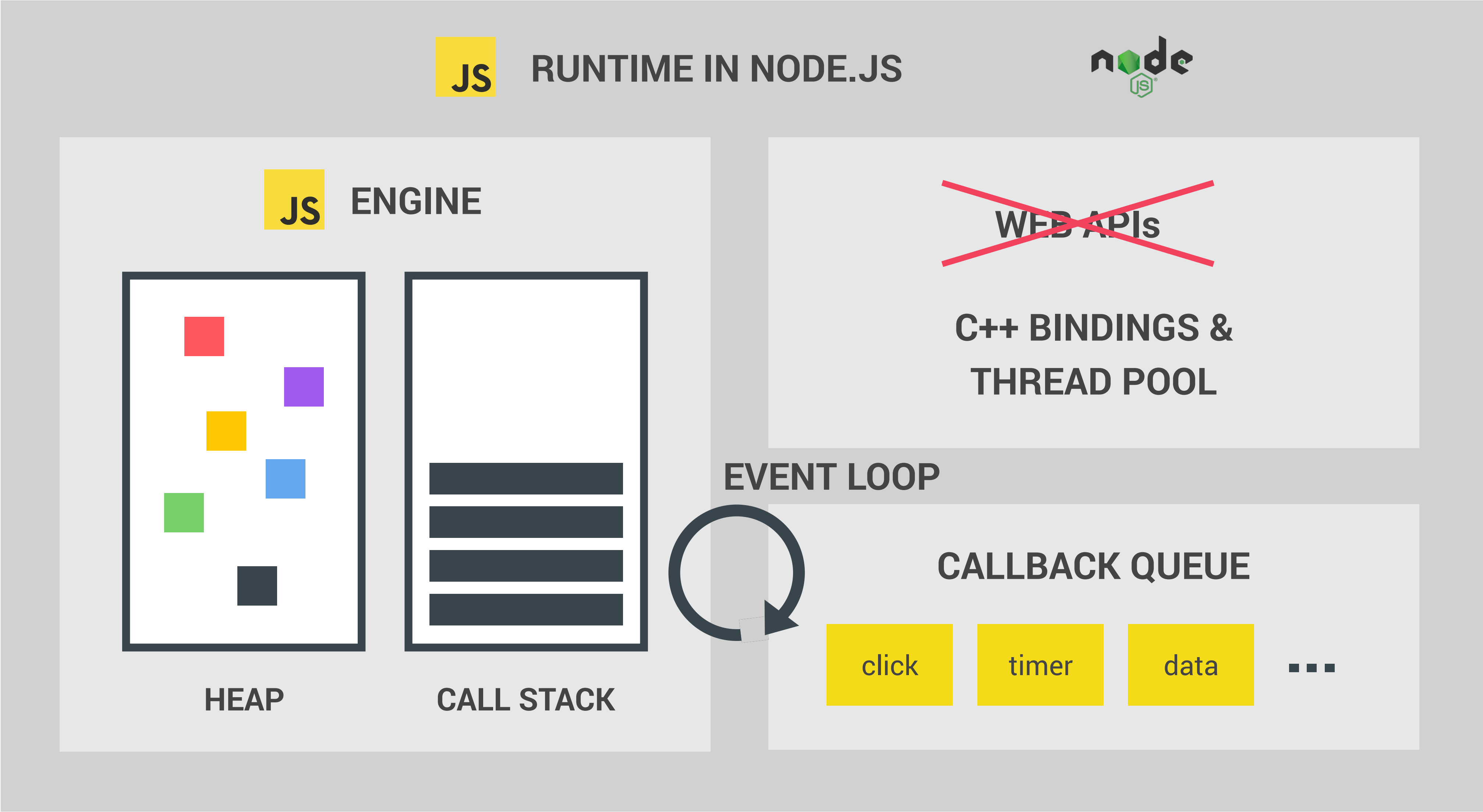




# THE BIGGER PICTURE: JAVASCRIPT RUNTIME



# THE BIGGER PICTURE: JAVASCRIPT RUNTIME









JONAS.IO  
SCHMEDTMANN

# THE COMPLETE JAVASCRIPT COURSE

FROM ZERO TO EXPERT!



@JONASSCHMEDTMAN

## SECTION

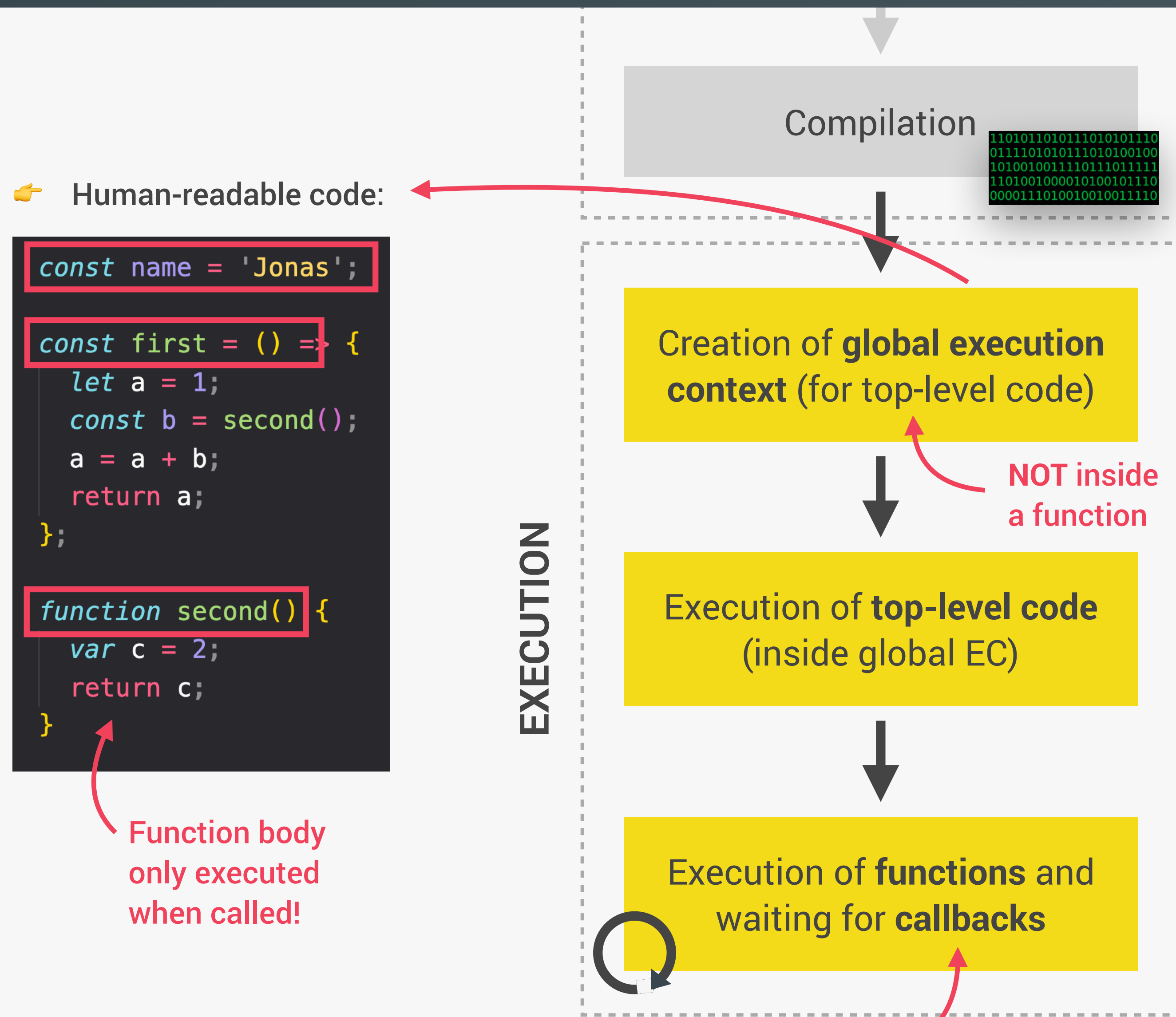
HOW JAVASCRIPT WORKS BEHIND THE  
SCENES

## LECTURE

EXECUTION CONTEXTS AND THE  
CALL STACK



# WHAT IS AN EXECUTION CONTEXT?



EXECUTION CONTEXT

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

"JavaScript code"

Pizza "execution context"

- Exactly **one** global execution context (EC): Default context, created for code that is not inside any function (top-level).
- One execution context **per function**: 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)



# THE CALL STACK

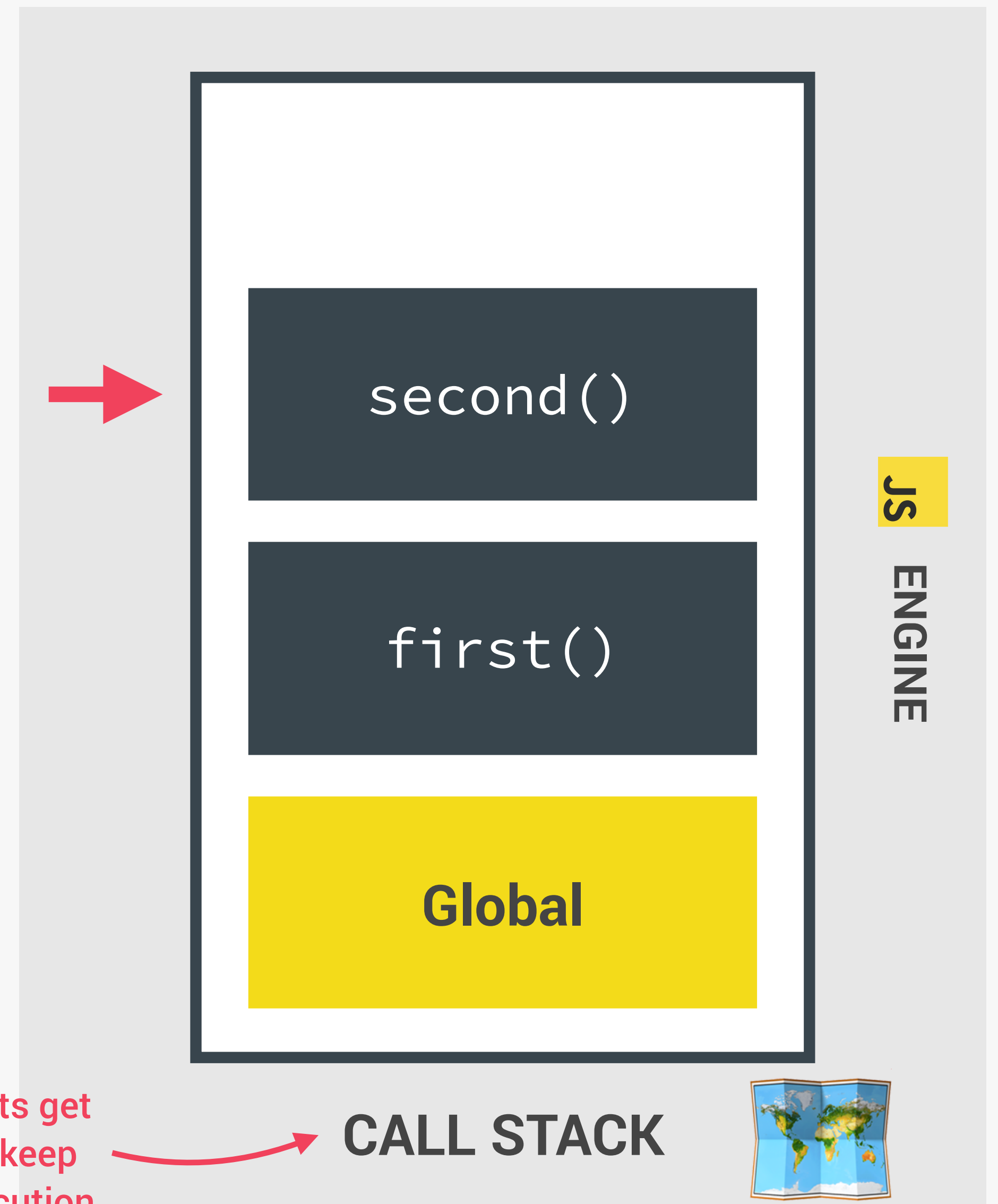
👉 Compiled code starts execution

```
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();
```



“Place” where execution contexts get stacked on top of each other, to keep track of where we are in the 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





# SCOPING AND SCOPE IN JAVASCRIPT: CONCEPTS

## SCOPE CONCEPTS

### EXECUTION CONTEXT

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

- 👉 **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';  
} ← 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)

# THE SCOPE CHAIN

```
const myName = 'Jonas';
```

```
function first() {
```

```
  const age = 30;
```

```
  if (age >= 30) { // true
```

```
    const decade = 3;
```

```
    var millennial = 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

VARIABLE LOOKUP IN SCOPE CHAIN

Global scope

myName = "Jonas"

SCOPE CHAIN

first() scope

age = 30

millennial = true

myName = "Jonas"

(Considering only  
variable declarations)

Scope has access  
to variables from  
all outer scopes

if block scope

decade = 3

age = 30

millennial = true

myName = "Jonas"

second() scope

job = "teacher"

age = 30

millennial = true

myName = "Jonas"



# SCOPE CHAIN VS. CALL STACK

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

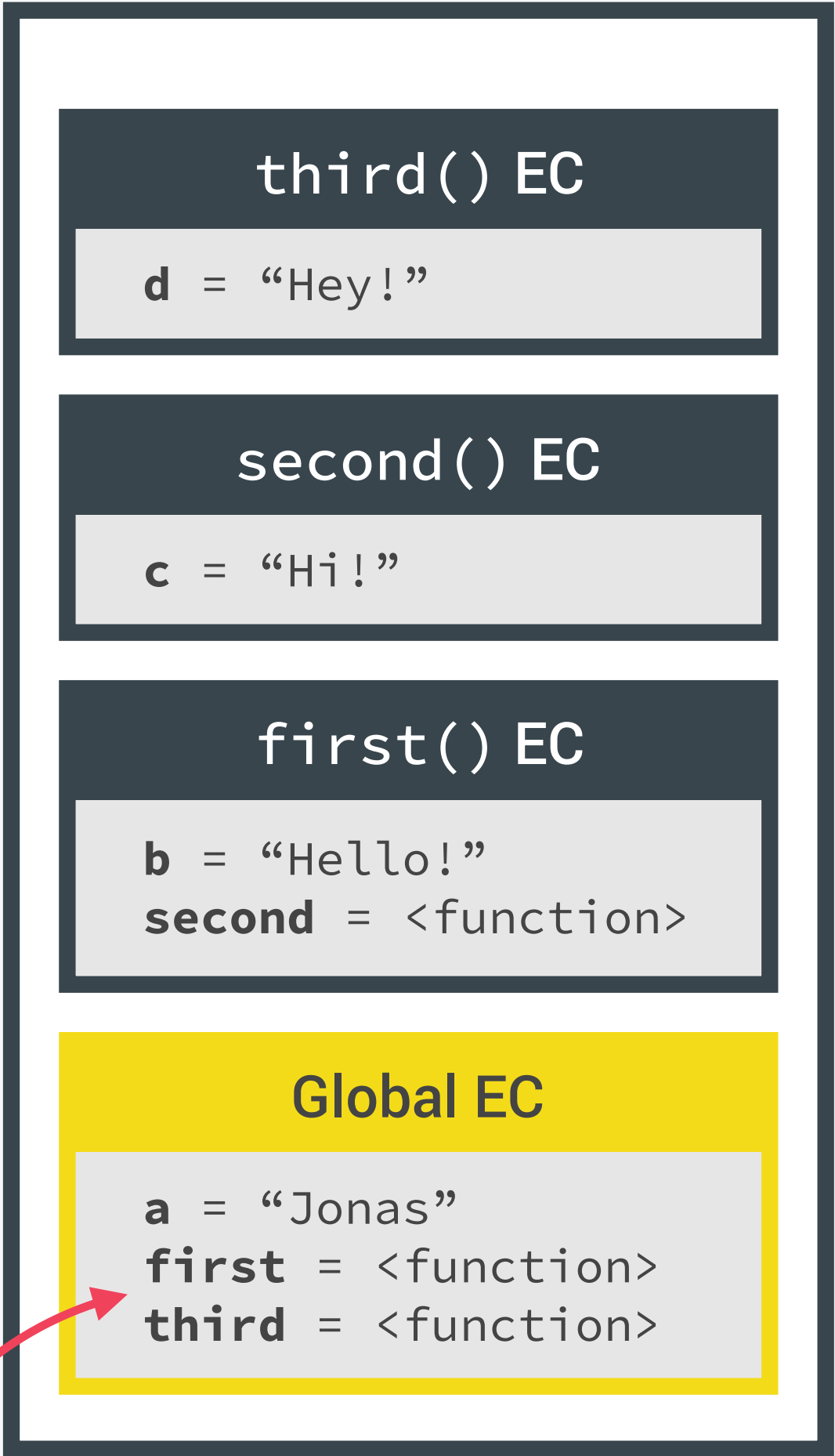
function first() {
  const b = 'Hello!';
  second();

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

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

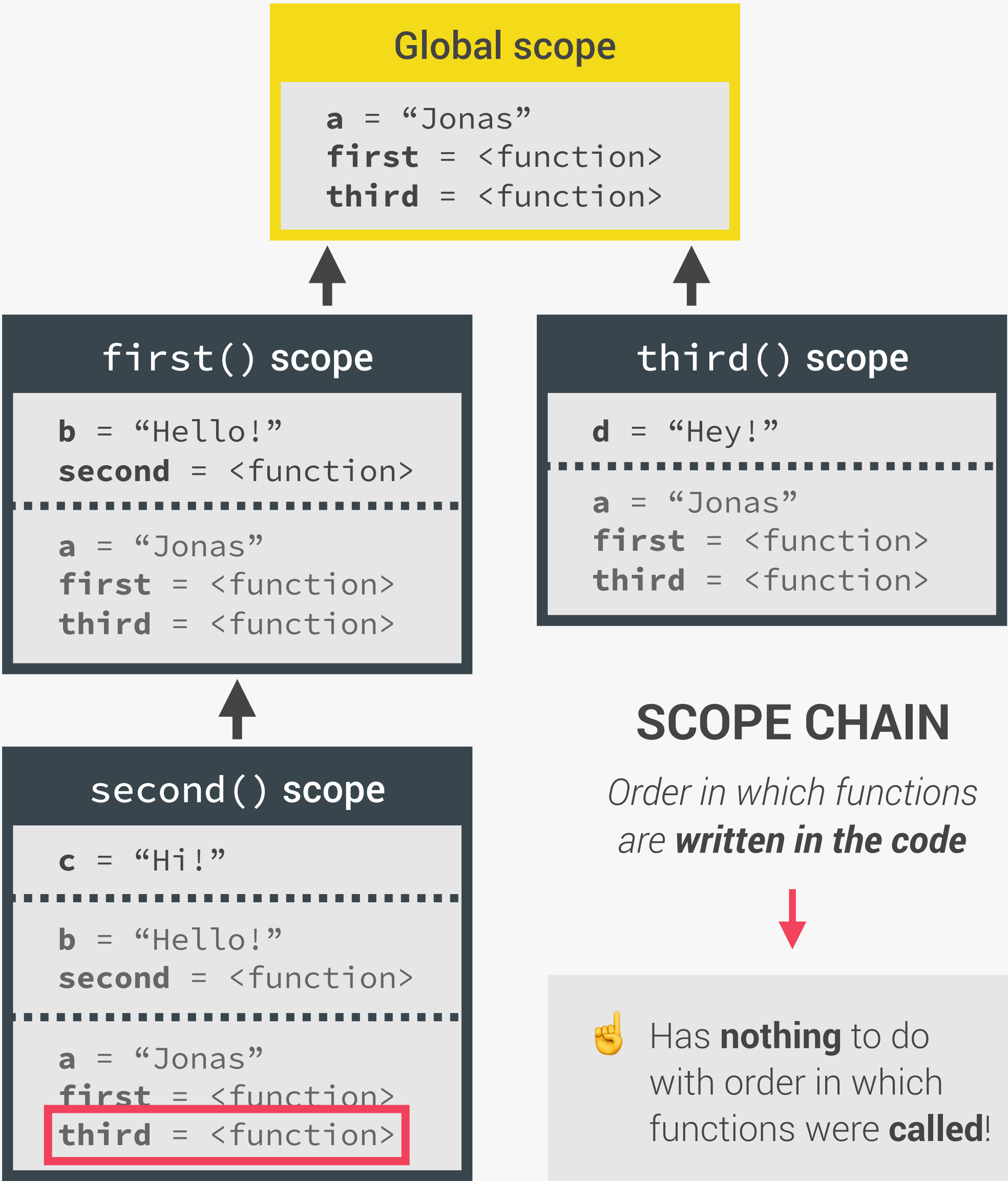
c and b can NOT be found in third() scope!

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**!

# SUMMARY 🎉

- 👉 Scoping asks the question “*Where do variables live?*” or “*Where can we access a certain variable, and where not?*”;
- 👉 There are 3 types of scope in JavaScript: the global scope, scopes defined by functions, and scopes defined by blocks;
- 👉 Only `let` and `const` variables are block-scoped. Variables declared with `var` end up in the closest function scope;
- 👉 In JavaScript, we have lexical scoping, so the rules of where we can access variables are based on exactly where in the code functions and blocks are written;
- 👉 Every scope always has access to all the variables from all its outer scopes. This is the scope chain!
- 👉 When a variable is not in the current scope, the engine looks up in the scope chain until it finds the variable it’s looking for. This is called variable lookup;
- 👉 The scope chain is a one-way street: a scope will never, ever have access to the variables of an inner scope;
- 👉 The scope chain in a certain scope is equal to adding together all the variable environments of the all parent scopes;
- 👉 The scope chain has nothing to do with the order in which functions were called. It does not affect the scope chain at all!







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?<br>👉 | INITIAL VALUE<br>👉                  | SCOPE<br>👉 |
|---------------------------------|---------------|-------------------------------------|------------|
| 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 |            |

Technically, yes. But not in practice

In strict mode. Otherwise: function!

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



# 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... ⌚>

## 👉 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`!

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





JONAS.IO  
SCHMEDTMANN

# THE COMPLETE JAVASCRIPT COURSE

FROM ZERO TO EXPERT!



@JONASSCHMEDTMAN

GOT QUESTIONS? FEEDBACK?  
JUST POST IT IN THE Q&A OF THIS  
VIDEO, AND YOU WILL GET HELP  
THERE!









JONAS.IO  
SCHMEDTMANN

# THE COMPLETE JAVASCRIPT COURSE

FROM ZERO TO EXPERT!



@JONASSCHMEDTMAN

## SECTION

HOW JAVASCRIPT WORKS BEHIND THE  
SCENES

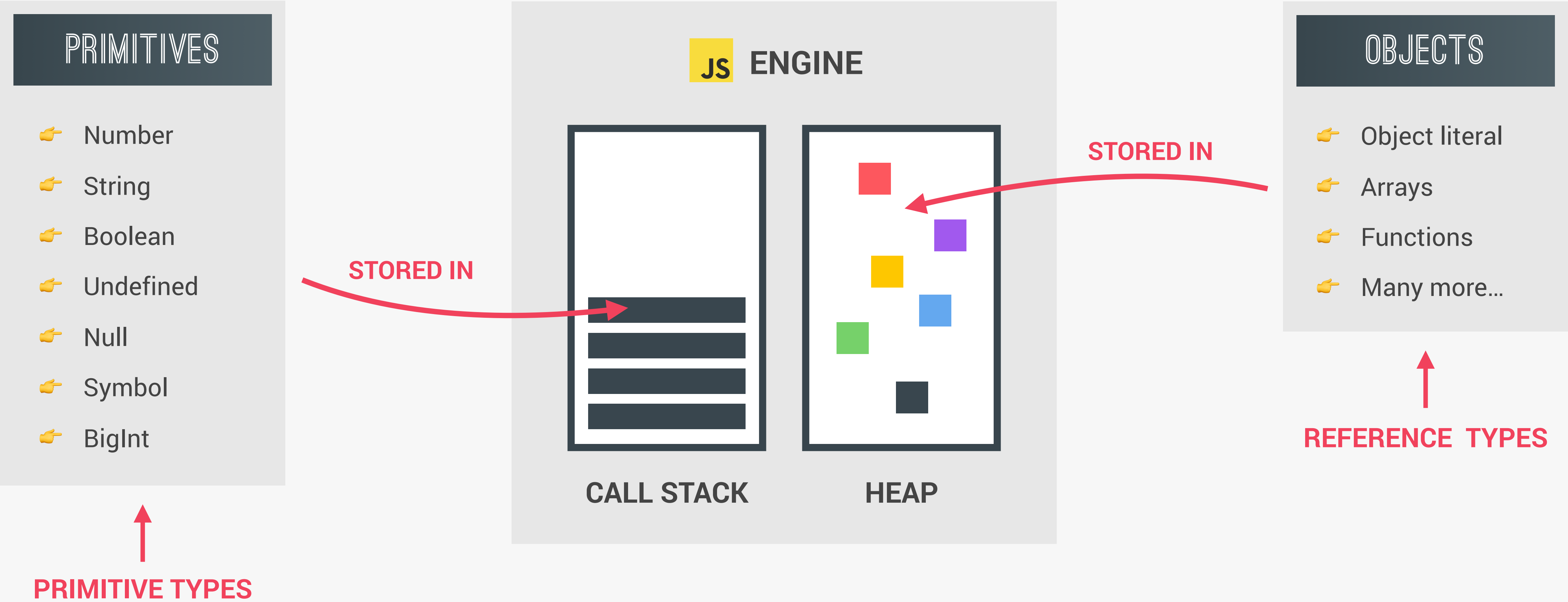
## LECTURE

PRIMITIVES VS. OBJECTS (PRIMITIVE  
VS. REFERENCE TYPES)





# REVIEW: PRIMITIVES, OBJECTS AND THE JAVASCRIPT ENGINE



# PRIMITIVE VS. REFERENCE VALUES

## 👉 Primitive values example:

```
let age = 30;
let oldAge = age;
age = 31;
console.log(age); // 31
console.log(oldAge); // 30
```

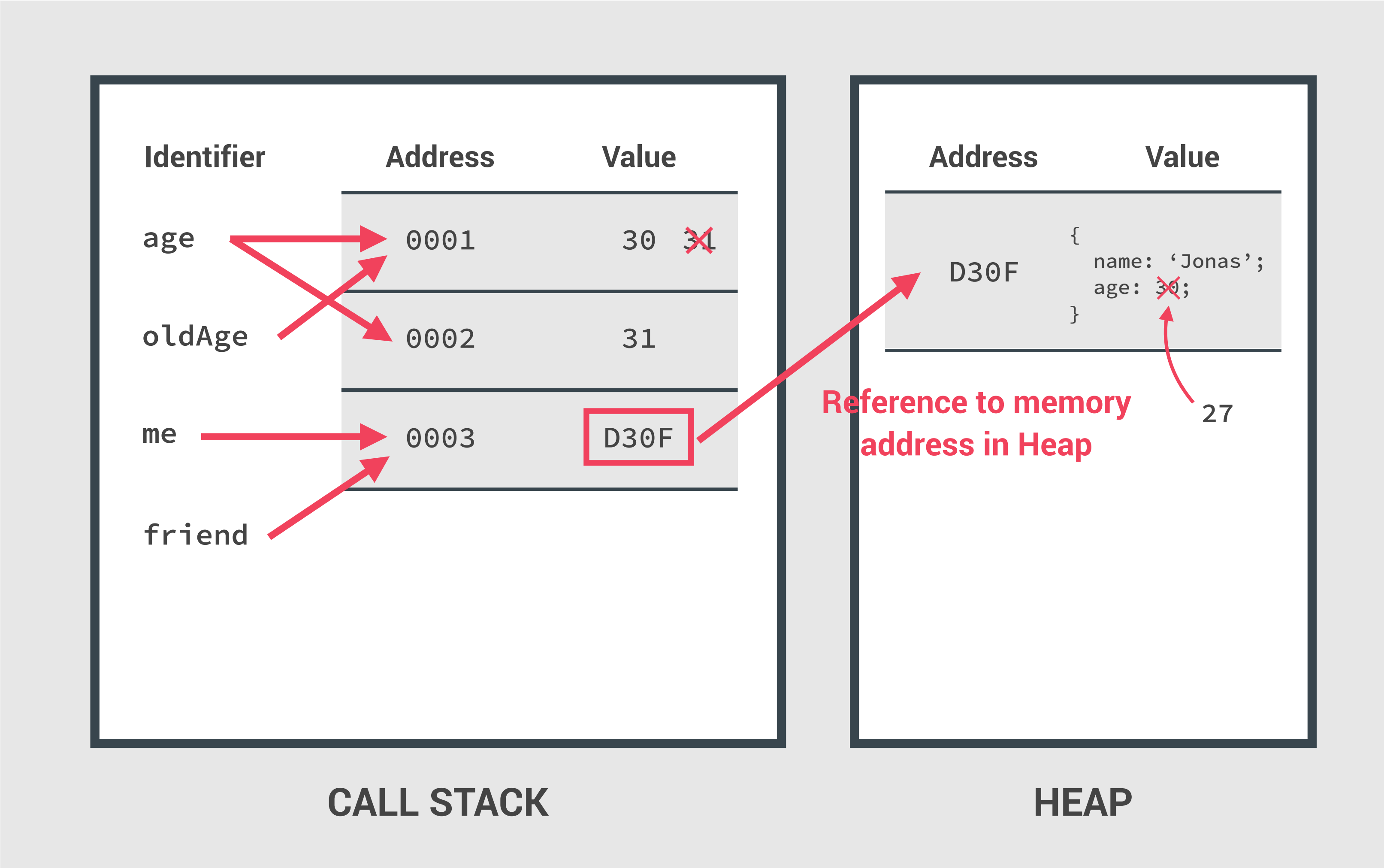
## 👉 Reference values example:

```
const me = {
  name: 'Jonas',
  age: 30
};
const friend = me;
friend.age = 27;

console.log('Friend:', friend);
// { name: 'Jonas', age: 27 }

console.log('Me:', me);
// { name: 'Jonas', age: 27 }
```

No problem, because we're NOT changing the **value** at address 0003!



# "HOW JAVASCRIPT WORKS BEHIND THE SCENES" TOPICS FOR LATER..

**1** **Prototypal Inheritance** 📌 Object Oriented Programming (OOP) With JavaScript

**2** **Event Loop** 📌 Asynchronous JavaScript: Promises, Async/Await and AJAX

**3** **How the DOM Really Works** 📌 Advanced DOM and Events





DATA STRUCTURES,  
MODERN OPERATORS  
AND STRINGS



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



# DATA STRUCTURES OVERVIEW

## SOURCES OF DATA

- 1 From the program itself:** Data written directly in source code (e.g. status messages)
- 2 From the UI:** Data input from the user or data written in DOM (e.g tasks in todo app)
- 3 From external sources:** Data fetched for example from web API (e.g. recipe objects)

Collection of data

Data structure

SIMPLE LIST?

KEY/VALUE PAIRS?

Arrays or Sets

Objects or Maps

### OTHER BUILT-IN:

- 👉 WeakMap
- 👉 WeakSet

### NON-BUILT IN:

- 👉 Stacks
- 👉 Queues
- 👉 Linked lists
- 👉 Trees
- 👉 Hash tables

Application  
Programming  
Interface

“Object”

Array

“Object”

```
{
  "count": 3,
  "recipes": [
    {
      "publisher": "101 Cookbooks",
      "title": "Best Pizza Dough Ever",
      "source_url": "http://www.101cookbooks.com/archiv",
      "recipe_id": "47746",
      "image_url": "http://forkify-api.herokuapp.com/im",
      "social_rank": 100,
      "publisher_url": "http://www.101cookbooks.com"
    },
    {
      "publisher": "The Pioneer Woman",
      "title": "Deep Dish Fruit Pizza",
      "source_url": "http://thepioneerwoman.com/cooking",
      "recipe_id": "46956",
      "image_url": "http://forkify-api.herokuapp.com/im",
      "social_rank": 100,
      "publisher_url": "http://thepioneerwoman.com"
    },
    {
      "publisher": "Closet Cooking",
      "title": "Pizza Dip",
      "source_url": "http://www.closetcooking.com/2011/",
      "recipe_id": "35477",
      "image_url": "http://forkify-api.herokuapp.com/im",
      "social_rank": 99.99999999999994,
      "publisher_url": "http://closetcooking.com"
    }
  ]
}
```

Keys allow us to  
describe values

👉 JSON data format example

# ARRAYS VS. SETS AND OBJECTS VS. MAPS

## ARRAYS

VS.

## SETS

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

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

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

- 👉 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

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

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

- 👉 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)

- 👉 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





# A CLOSER LOOK AT FUNCTIONS



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





# 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

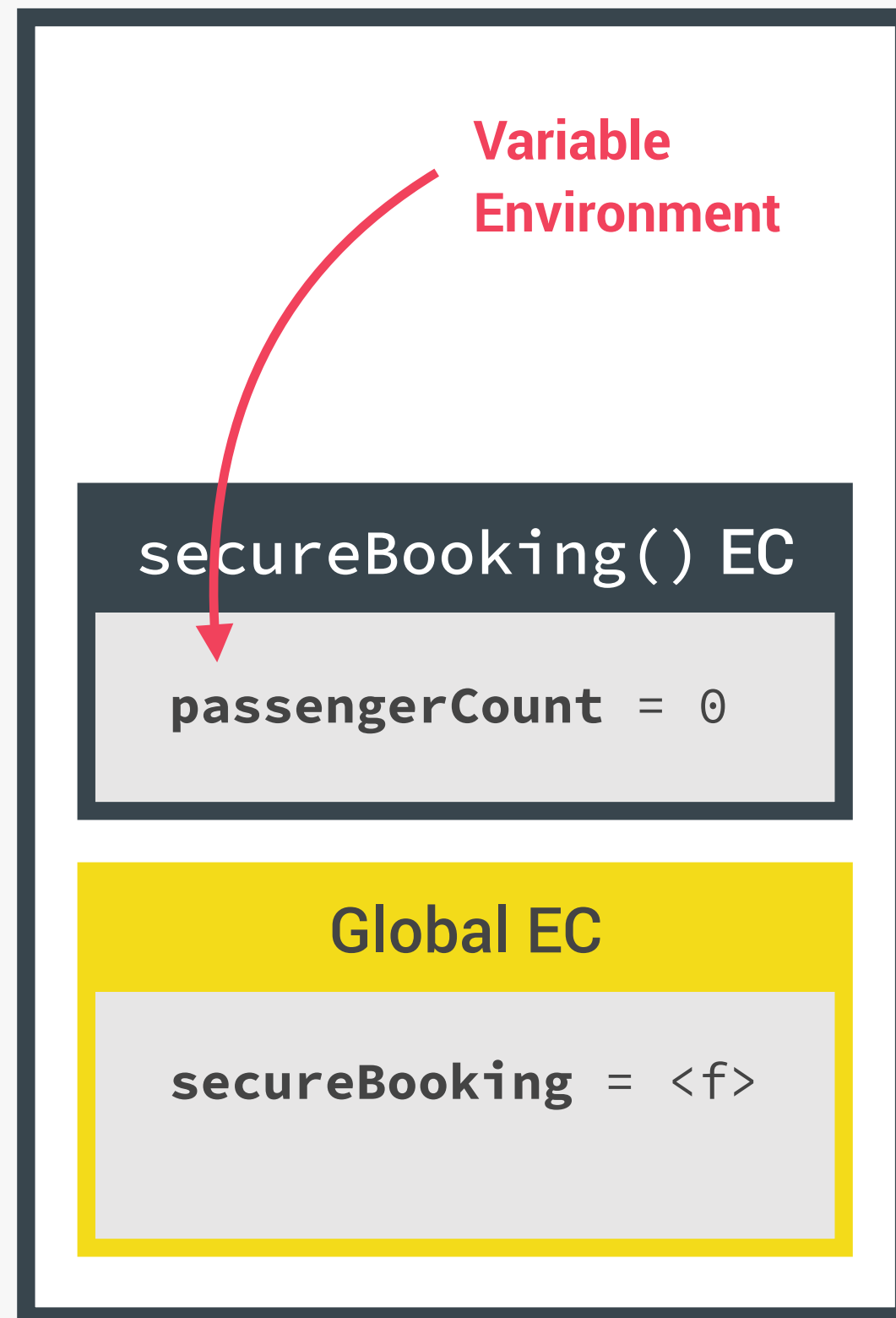
A CLOSER LOOK AT FUNCTIONS

LECTURE

CLOSURES

JS

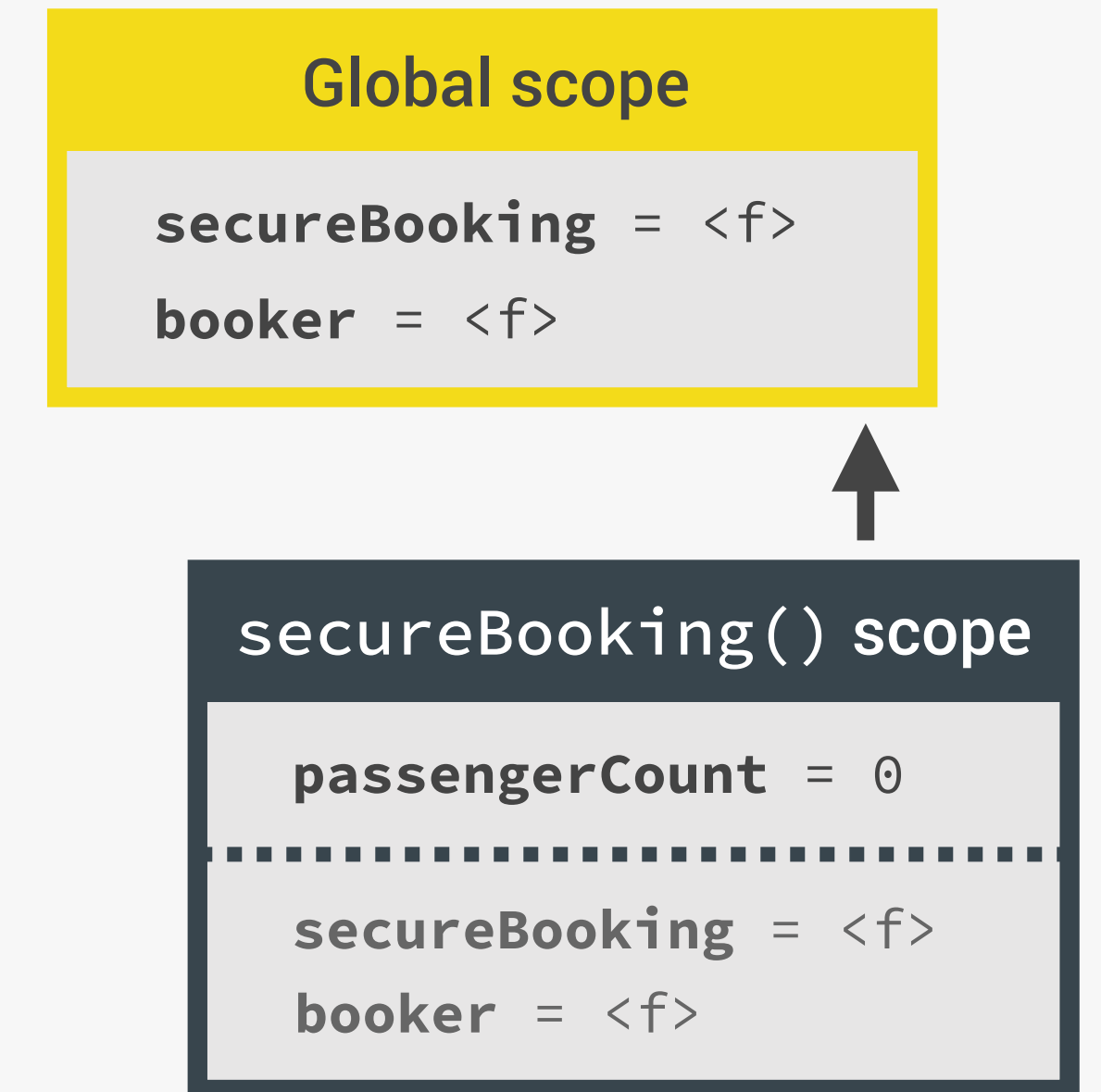
# "CREATING" A CLOSURE



CALL STACK

Order in which  
functions were *called*

```
const secureBooking = function () {  
  let passengerCount = 0;  
  
  return function () {  
    passengerCount++;  
    console.log(`${passengerCount}  
    passengers`);  
  };  
};  
  
const booker = secureBooking();
```



SCOPE CHAIN

Order in which functions  
are *written in the code*

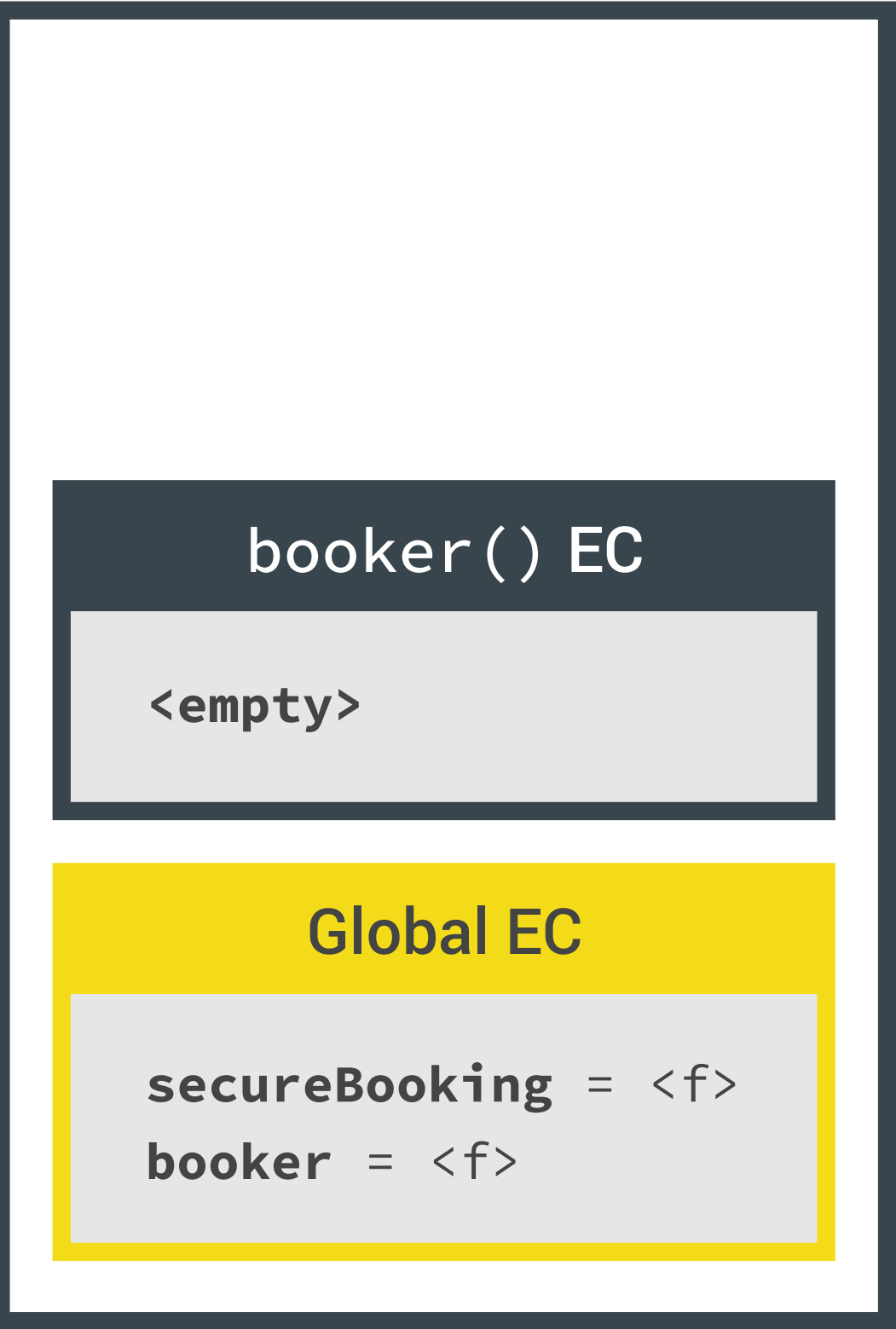


# UNDERSTANDING CLOSURES



# 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

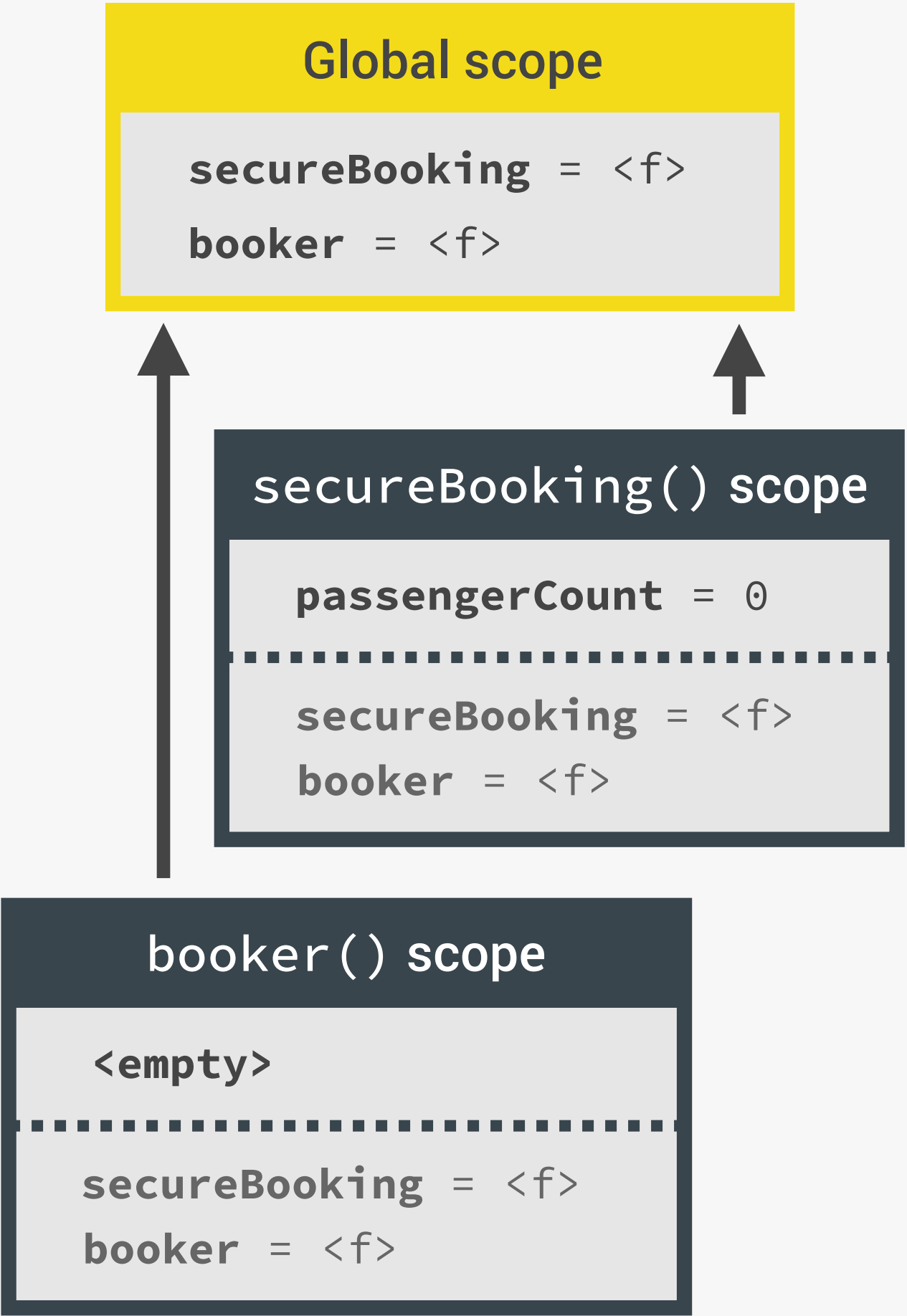


CALL STACK

```
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

(Priority over scope chain)  
**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

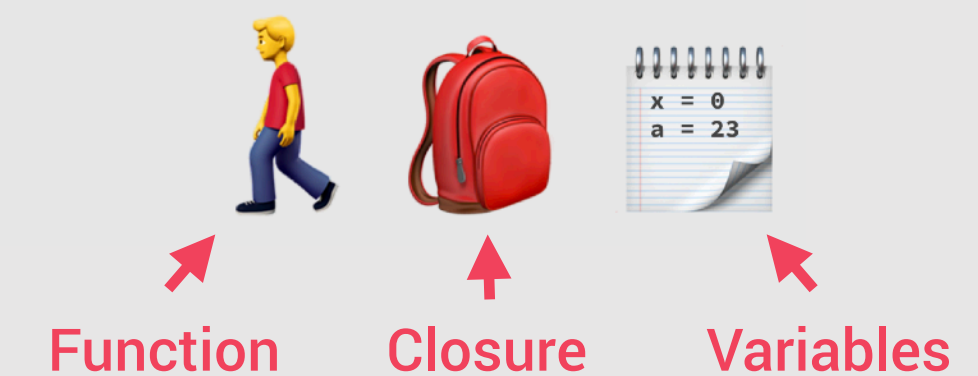
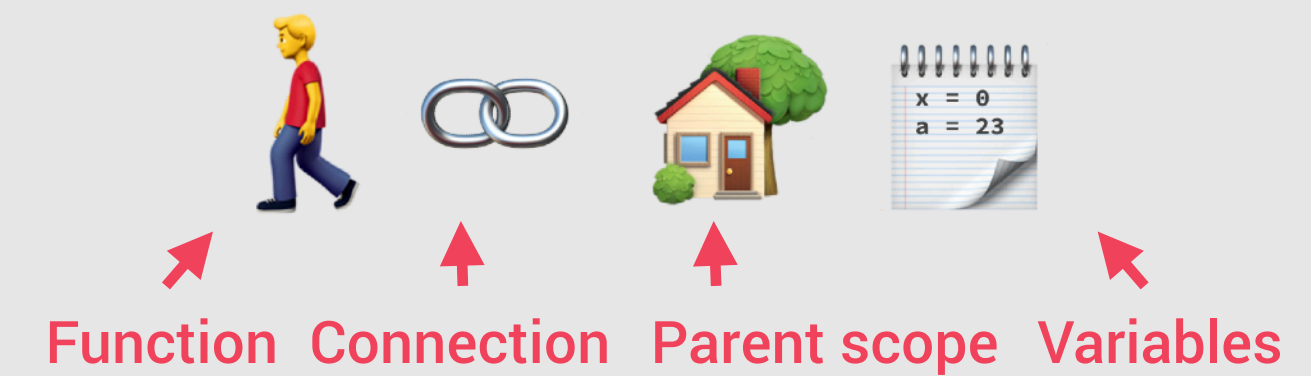
- 👉 A closure gives a function access to all the variables **of its parent function**, even **after** that parent function has returned. The function keeps a **reference** to its outer scope, which **preserves** the scope chain throughout time.

↓ Less formal

- 👉 A closure makes sure that a function doesn't lose 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.





# WORKING WITH ARRAYS



JONAS.IO  
SCHMEDTMANN

# THE COMPLETE JAVASCRIPT COURSE

FROM ZERO TO EXPERT!



@JONASSCHMEDTMAN

SECTION

WORKING WITH ARRAYS

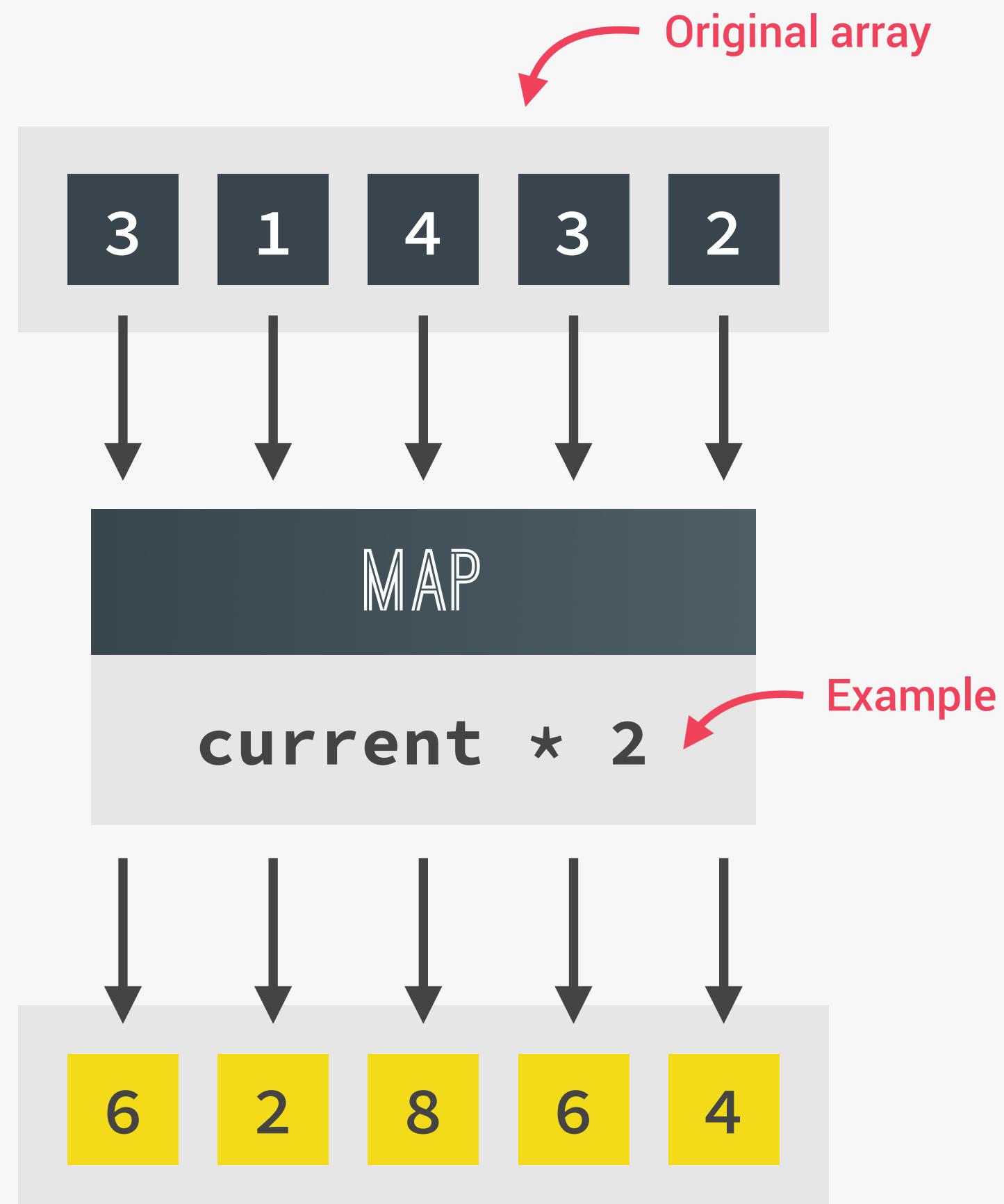
LECTURE

DATA TRANSFORMATIONS: MAP, FILTER,  
REDUCE

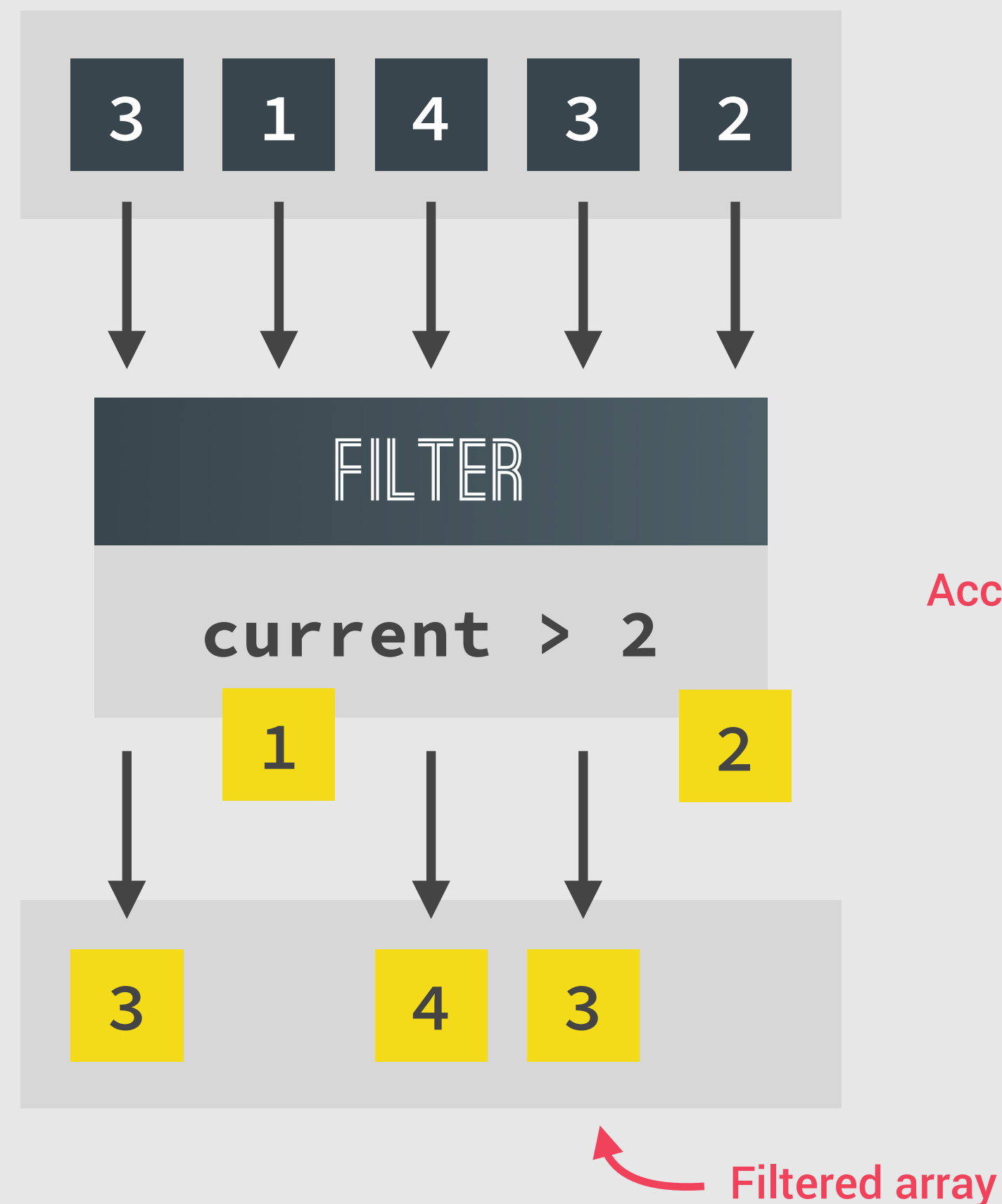
JS



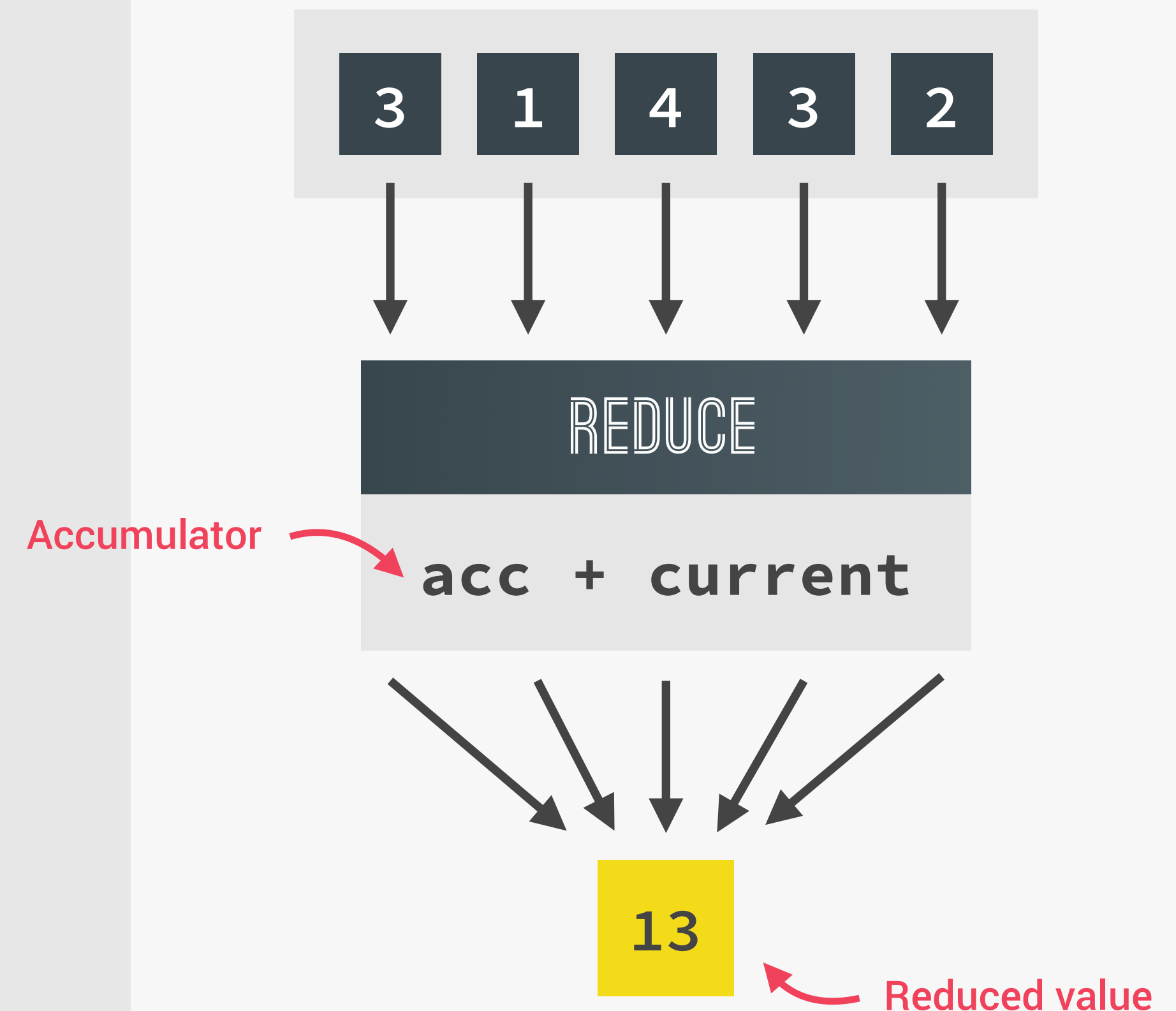
# DATA TRANSFORMATIONS WITH MAP, FILTER AND REDUCE



👉 `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)





JONAS.IO  
SCHMEDTMANN

# THE COMPLETE JAVASCRIPT COURSE

FROM ZERO TO EXPERT!



@JONASSCHMEDTMAN

SECTION

WORKING WITH ARRAYS

LECTURE

SUMMARY: WHICH ARRAY METHOD TO  
USE?





# 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)*



# ADVANCED DOM AND EVENTS





JONAS.IO  
SCHMEDTMANN

# THE COMPLETE JAVASCRIPT COURSE

FROM ZERO TO EXPERT!



@JONASSCHMEDTMAN

SECTION

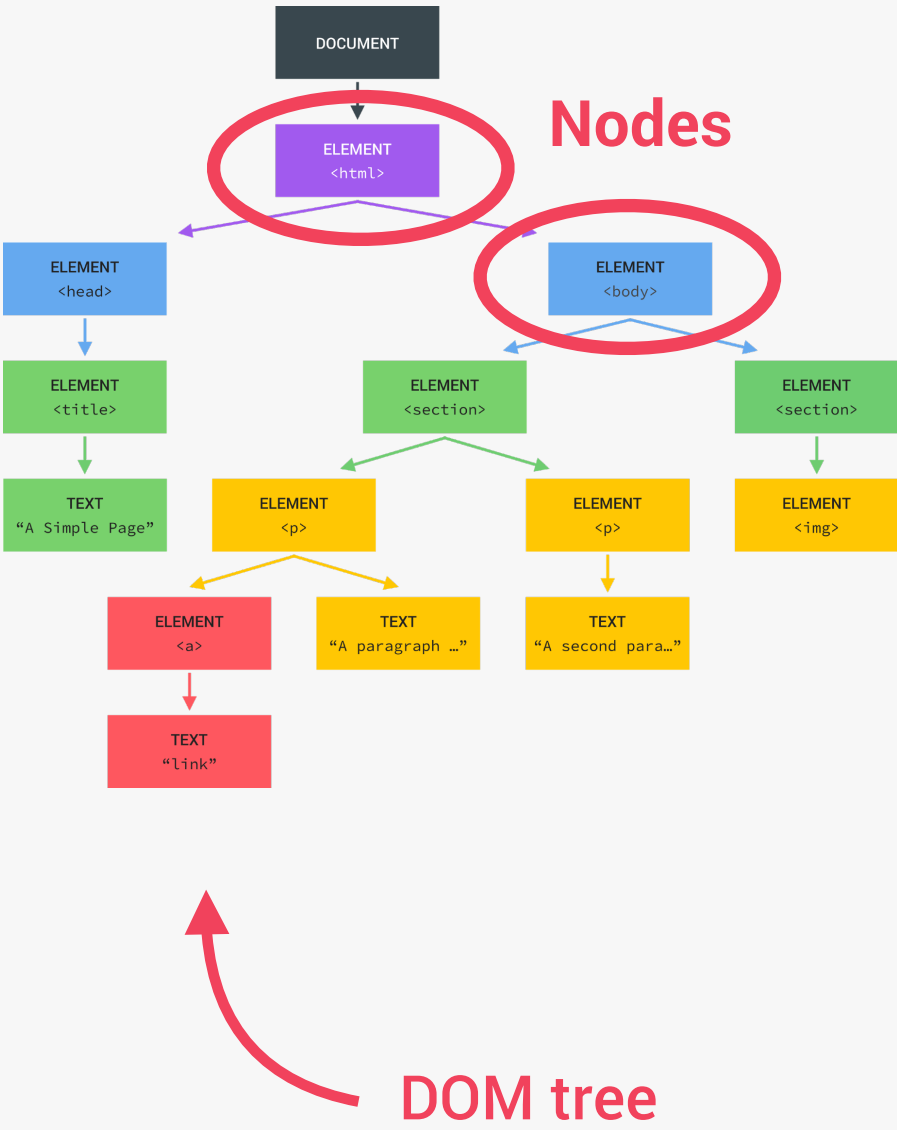
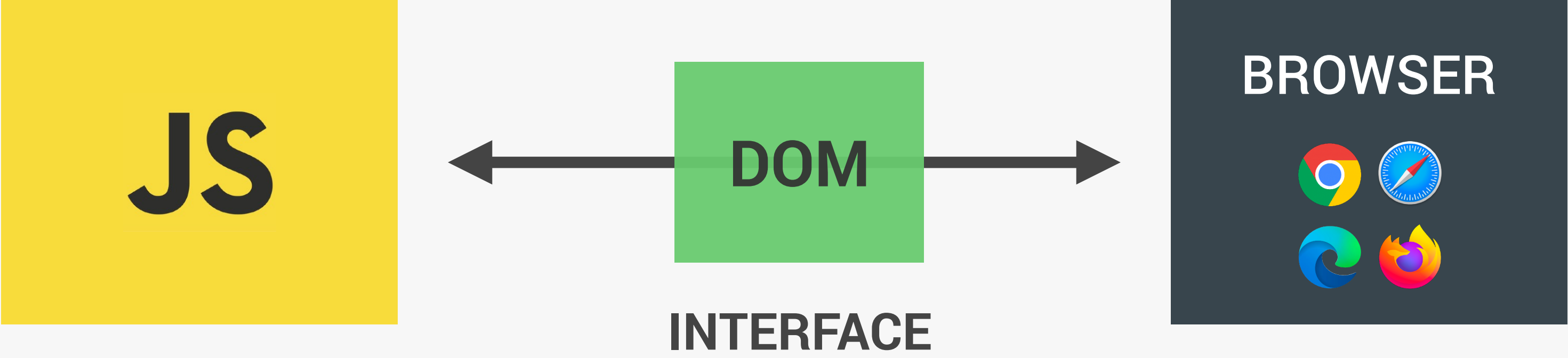
ADVANCED DOM AND EVENTS

LECTURE

HOW THE DOM REALLY WORKS

JS

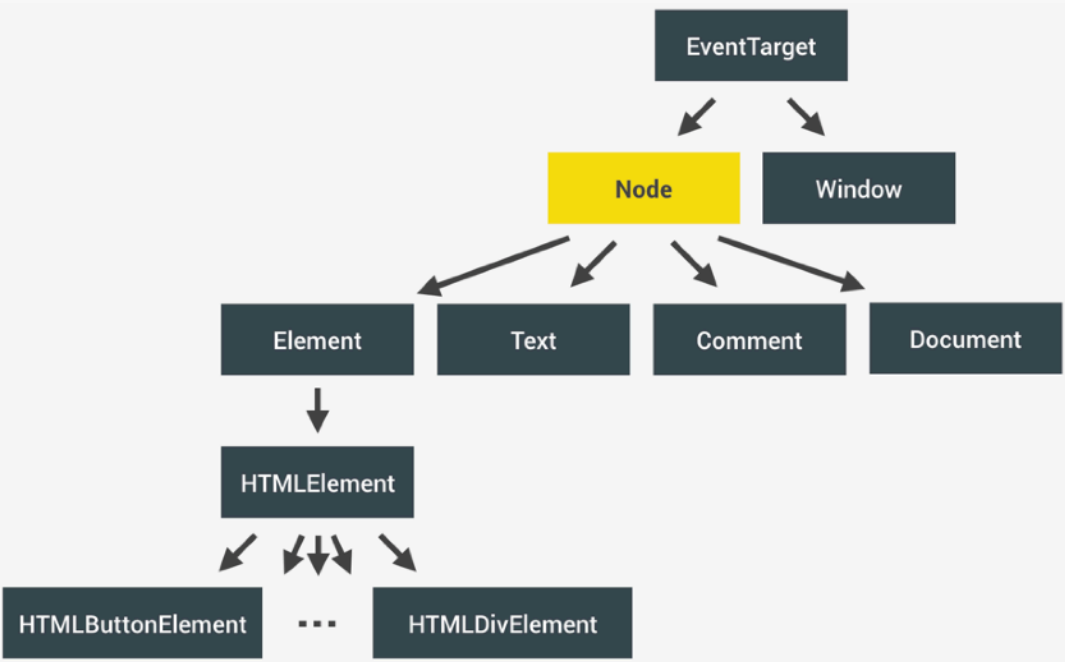
# REVIEW: WHAT IS THE DOM?



- 👉 Allows us to make JavaScript interact with the browser;
- 👉 We can write JavaScript to create, modify and delete HTML elements; set styles, classes and attributes; and listen and respond to events;
- 👉 DOM tree is generated from an HTML document, which we can then interact with;
- 👉 DOM is a very complex API that contains lots of methods and properties to interact with the DOM tree

Application Programming Interface

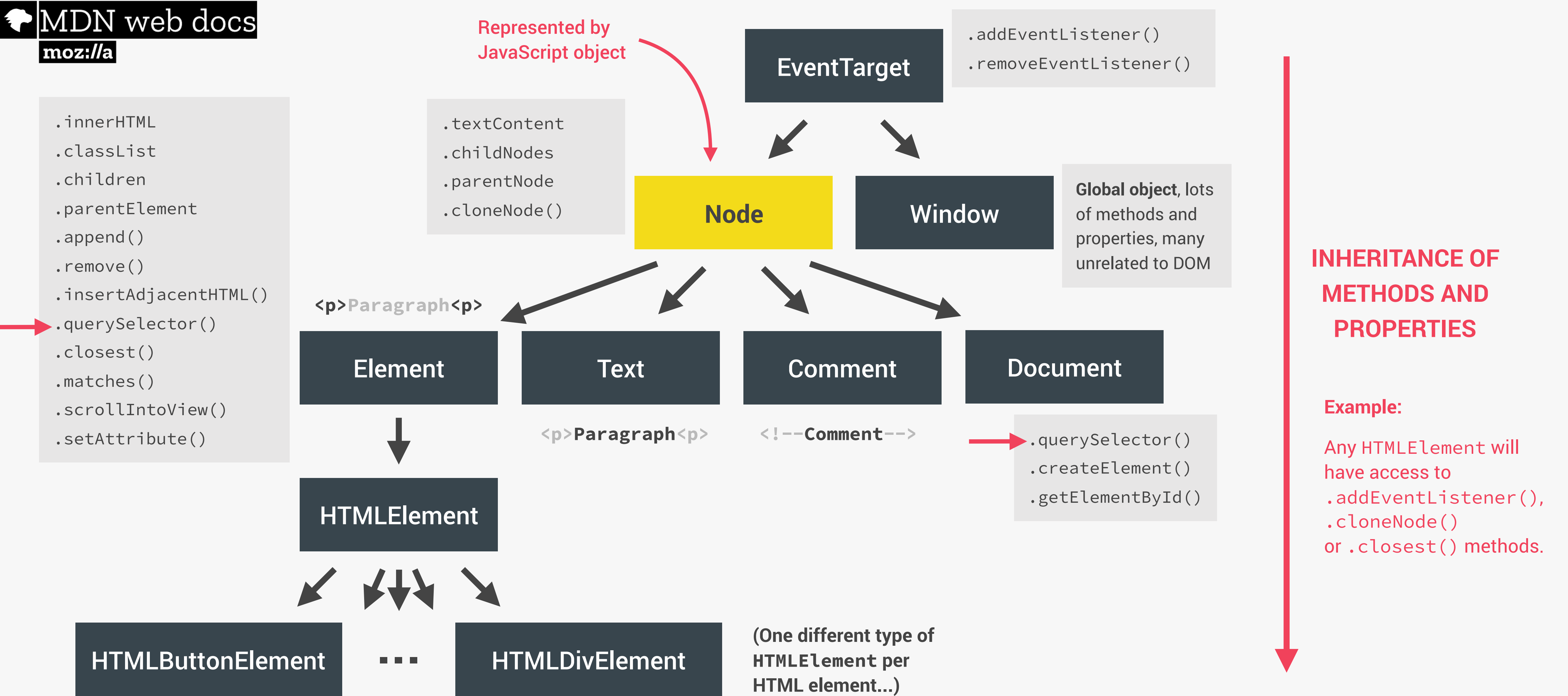
```
.querySelector() / .addEventListener() / .createElement() /  
.innerHTML / .textContent / .children / etc ...
```





# HOW THE DOM API IS ORGANIZED BEHIND THE SCENES

 **MDN web docs**  
**moz://a**



(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



# BUBBLING AND CAPTURING

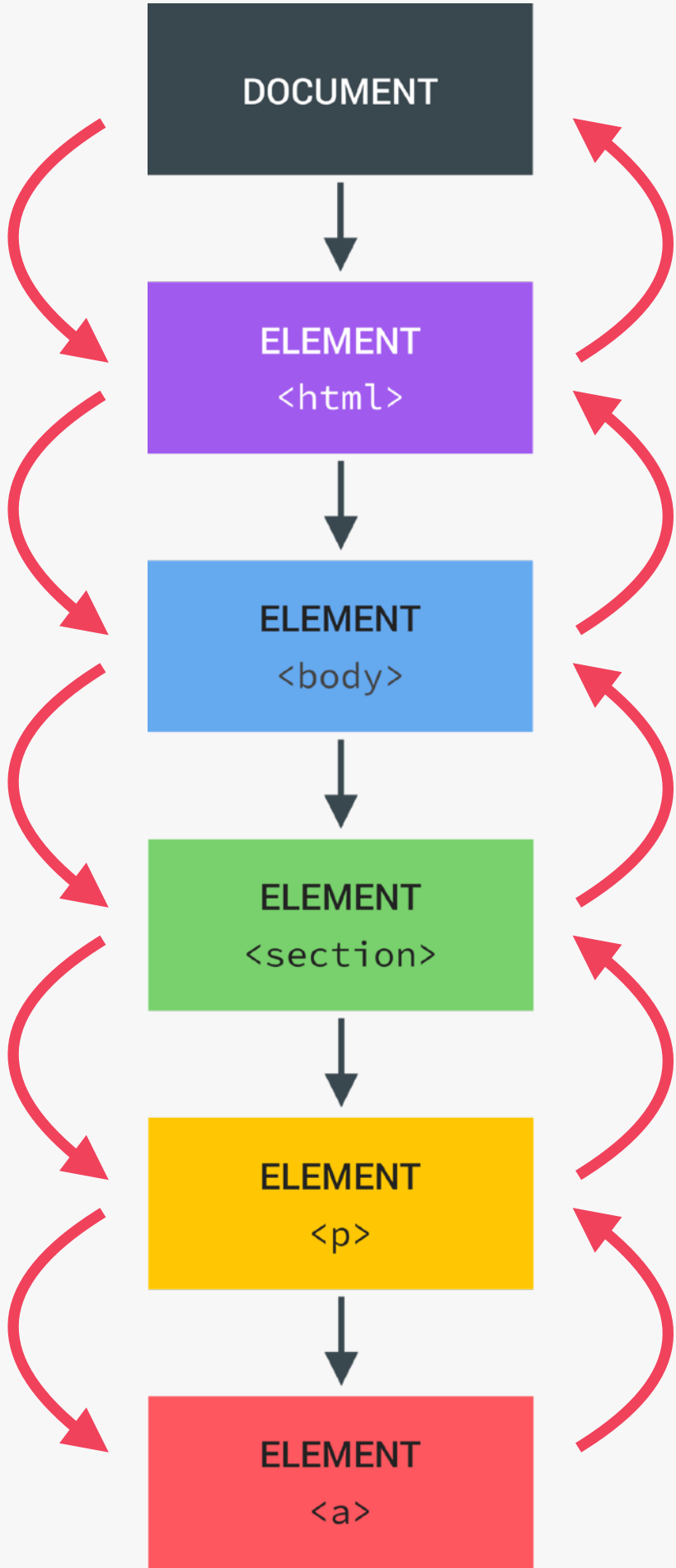
```
<html>
<head>
  <title>A Simple Page</title>
</head>
<body>
  <section>
    <p>A paragraph with a <a>link</a></p>
    <p>A second paragraph</p>
  </section>
  <section>
    
  </section>
</body>
</html>
```

(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">
```

Parsing HTML

Waiting...

Finish parsing HTML

Time



Fetch script

Execute

DOMContentLoaded

Parsing HTML

Fetch script

Execute

DOMContentLoaded

ASYNC

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

Parsing HTML

Waiting

Finish parsing HTML

Fetch script

Execute

DOMContentLoaded

👉 Makes no sense 🙄

DEFER

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

Parsing HTML

Execute

Fetch script

DOMContentLoaded

👉 Makes no sense 🙄



# REGULAR VS. ASYNC VS. DEFER

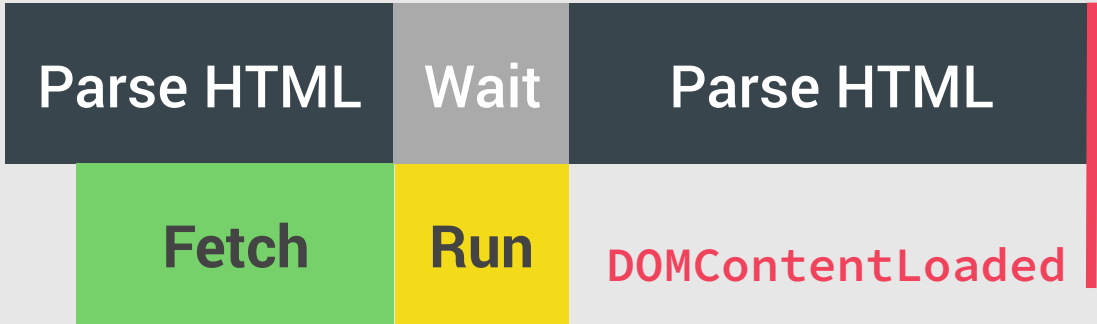
## END OF BODY



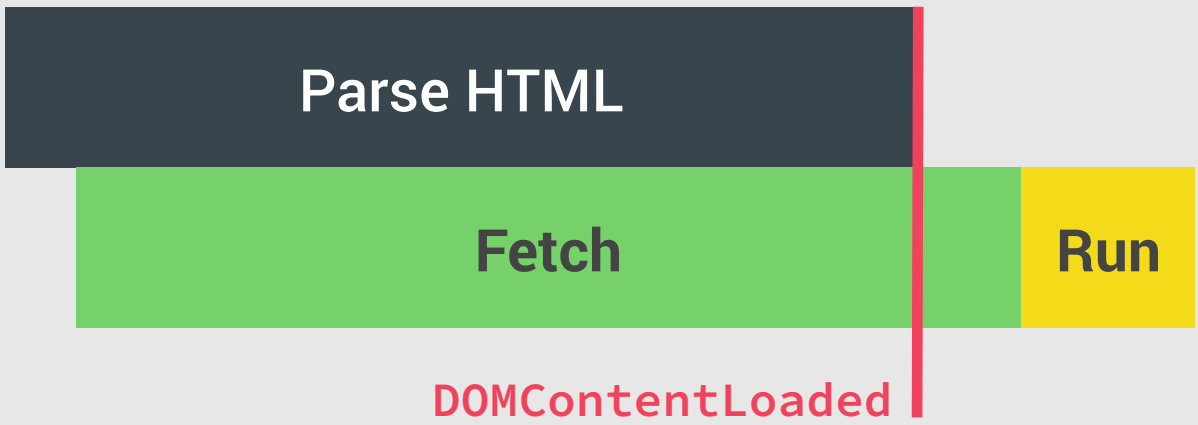
- 👉 Scripts are fetched and executed **after the HTML is completely parsed**
- 👉 **Use if you need to support old browsers**

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

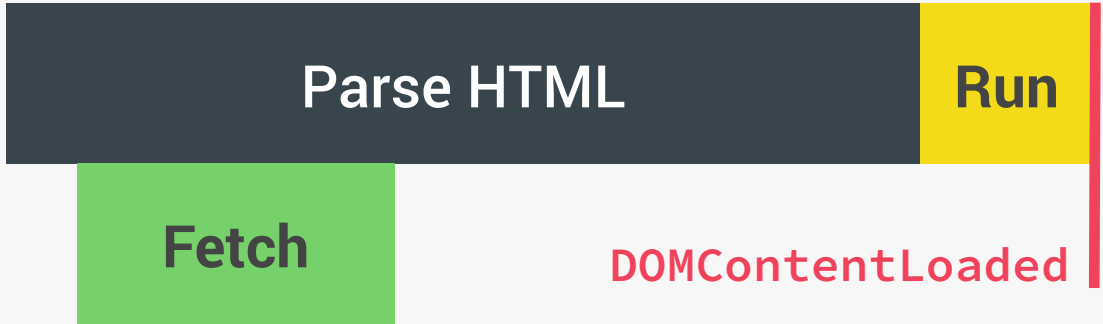
## ASYNC IN HEAD



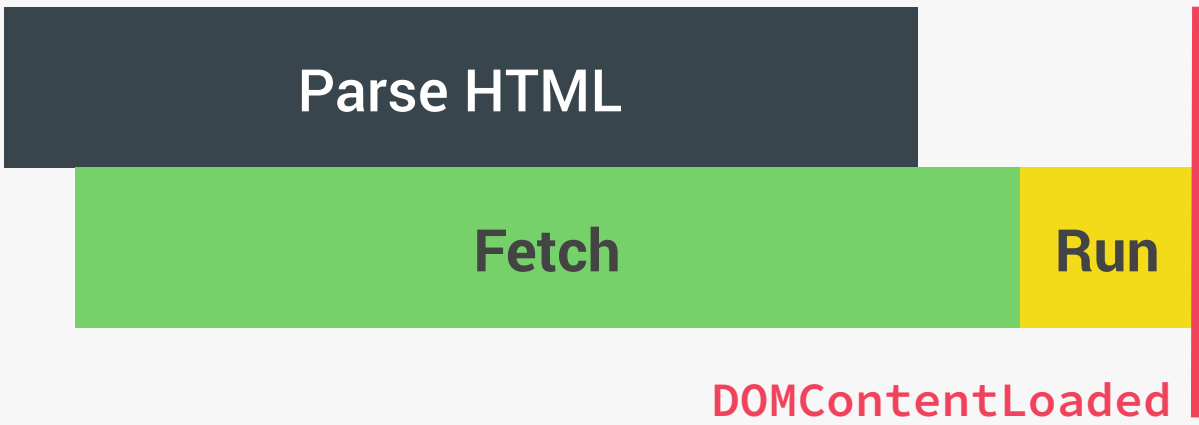
- 👉 Scripts are fetched **asynchronously** and executed **immediately**
- 👉 Usually the DOMContentLoaded event waits for **all** scripts to execute, except for async scripts. So, DOMContentLoaded does **not** wait for an async script
- 👉 Scripts **not** guaranteed to execute in order
- 👉 **Use for 3rd-party 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)**





# OBJECT ORIENTED PROGRAMMING (OOP) WITH JAVASCRIPT





JONAS.IO  
SCHMEDTMANN

# THE COMPLETE JAVASCRIPT COURSE

FROM ZERO TO EXPERT!



@JONASSCHMEDTMAN

## SECTION

OBJECT ORIENTED PROGRAMMING  
(OOP) WITH JAVASCRIPT

## LECTURE

WHAT IS OBJECT-ORIENTED  
PROGRAMMING?

JS



# WHAT IS OBJECT-ORIENTED PROGRAMMING? (OOP)

OOP

Style of code, "how" we write and organize code

👉 Object-oriented programming (OOP) is a programming paradigm based on the concept of objects;

E.g. user or todo list item

👉 We use objects to **model** (describe) real-world or abstract features;

E.g. HTML component or data structure

👉 Objects may contain data (properties) and code (methods). By using objects, we pack **data and the corresponding behavior** into one block;

👉 In OOP, objects are **self-contained** pieces/blocks of code;

👉 Objects are **building blocks** of applications, and **interact** with one another;

👉 Interactions happen through a **public interface** (API): methods that the code **outside** of the object can access and use to communicate with the object;

👉 OOP was developed with the goal of **organizing** code, to make it **more flexible and easier to maintain** (avoid "spaghetti code").

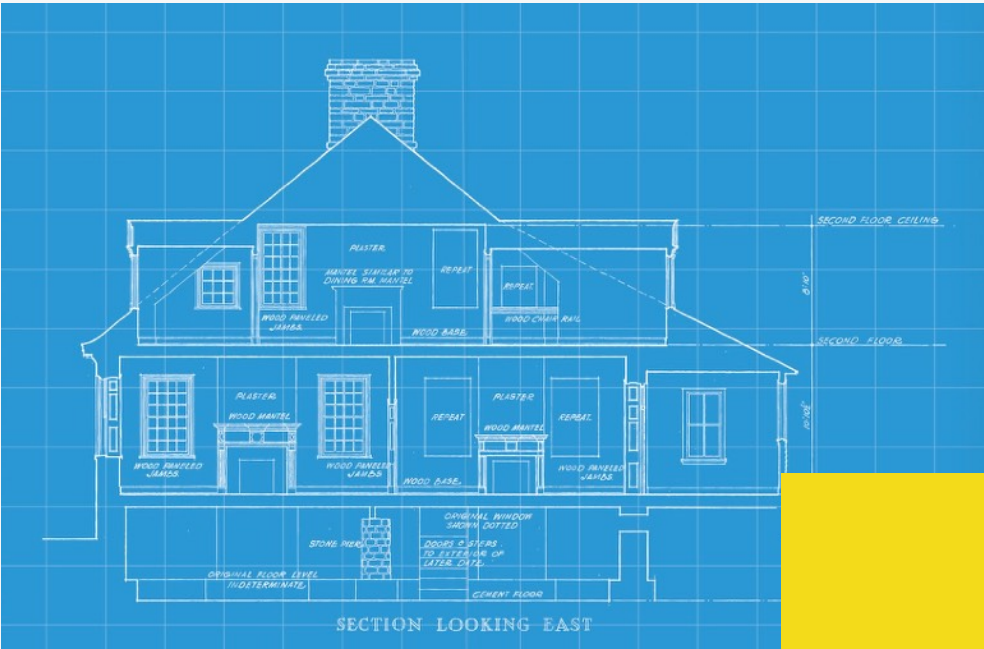
Data

```
const user = {  
  user: 'jonas',  
  password: 'dk23s',  
  
  login(password) {  
    // Login logic  
  },  
  sendMessage(str) {  
    // Sending logic  
  }  
}
```

Behaviour



# CLASSES AND INSTANCES (TRADITIONAL OOP)



CLASS

Like a blueprint from which we can create new objects

```
User {
  user
  password
  email

  login(password) {
    // Login logic
  }
  sendMessage(str) {
    // Sending logic
  }
}
```

Just a representation, NOT actual JavaScript syntax!

JavaScript does NOT support *real* classes like represented here

Instance

```
{
  user = 'jonas'
  password = 'dk23s'
  email = 'hello@jonas.io'

  login(password) {
    // Login logic
  }
  sendMessage(str) {
    // Sending logic
  }
}
```



New object created from the class. Like a *real* house created from an *abstract* blueprint

Instance

```
{
  user = 'mary'
  password = 'qwerty23'
  email = 'mary@test.com'

  login(password) {
    // Login logic
  }
  sendMessage(str) {
    // Sending logic
  }
}
```



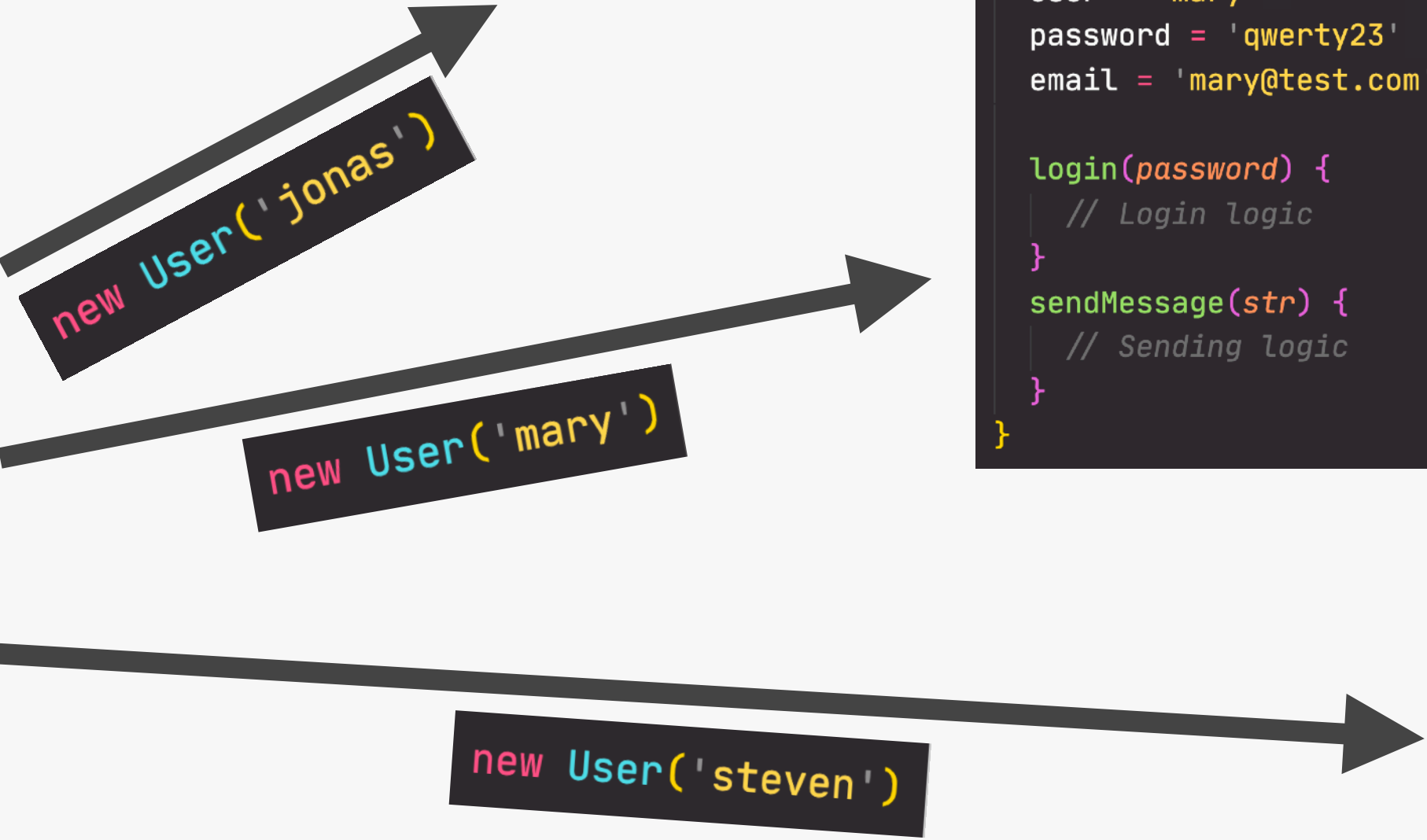
Instance

```
{
  user = 'steven'
  password = '5p8dz32dd'
  email = 'steven@tes.co'

  login(password) {
    // Login logic
  }
  sendMessage(str) {
    // Sending logic
  }
}
```



👉 Conceptual overview: it works a bit differently in JavaScript. Still important to understand!





The diagram consists of four yellow rectangular boxes stacked vertically on a light gray background. Each box contains one of the four pillars of Object-Oriented Programming (OOP) in bold black text. From top to bottom, the boxes are labeled: Abstraction, Encapsulation, Inheritance, and Polymorphism.

**Abstraction**

**Encapsulation**

**Inheritance**

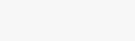
**Polymorphism**

## Abstraction

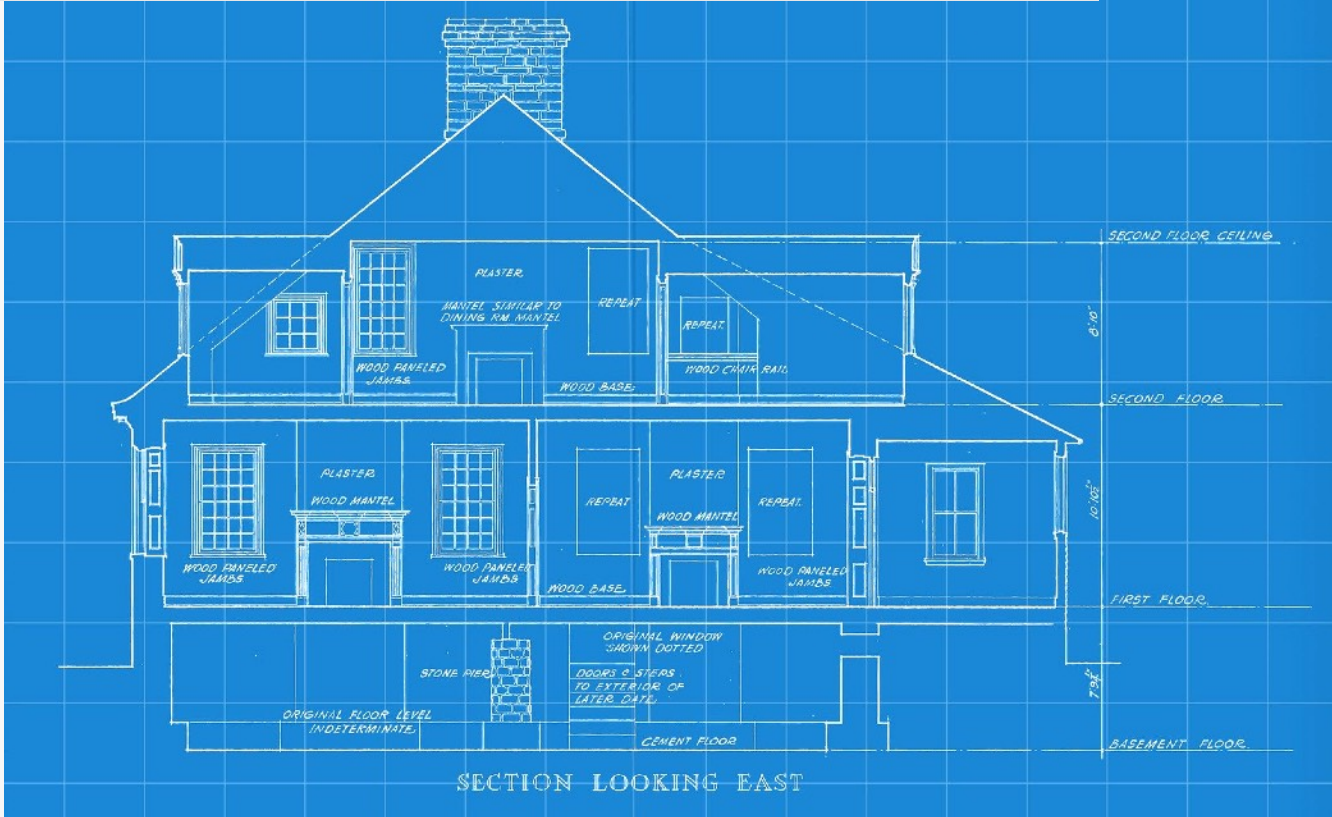
## Encapsulation

# Inheritance

# Polymorphism



🤔 *“How do we actually design classes? How do we model real-world data into classes?”*



# PRINCIPLE 1: ABSTRACTION

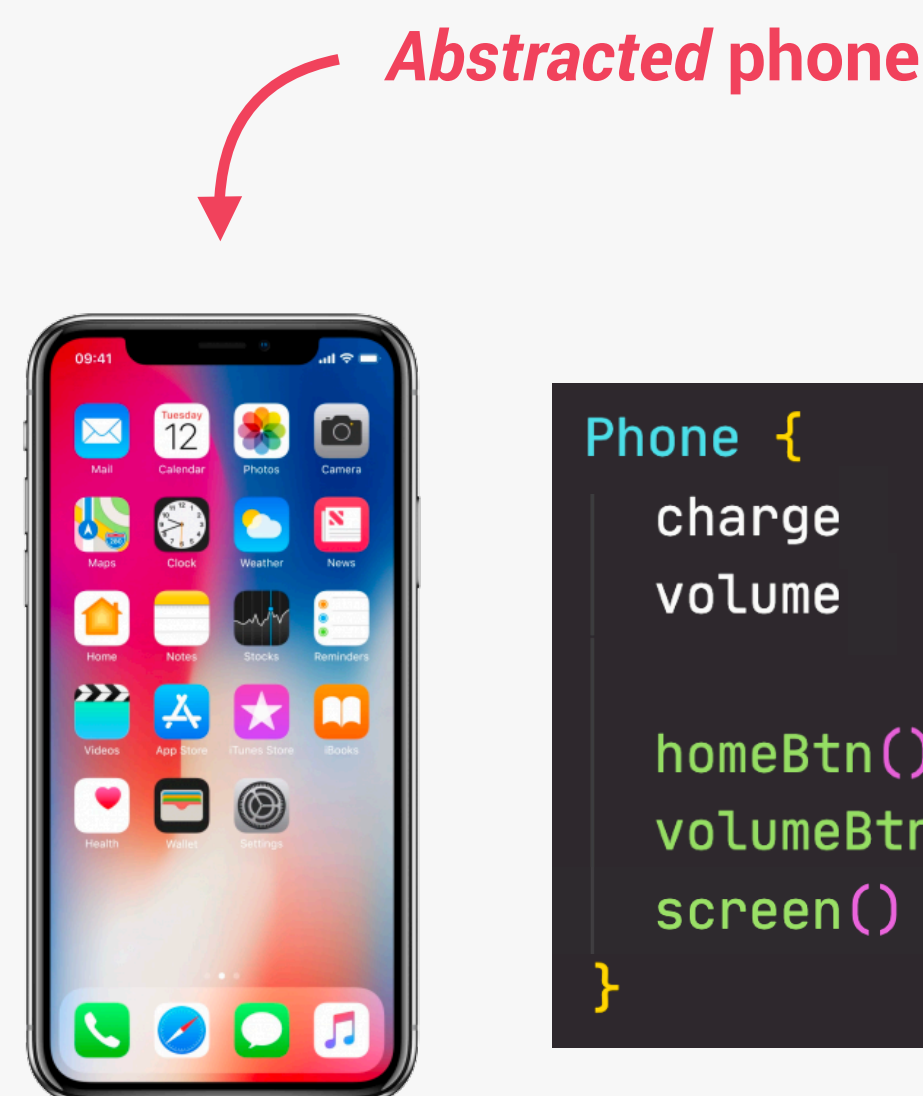
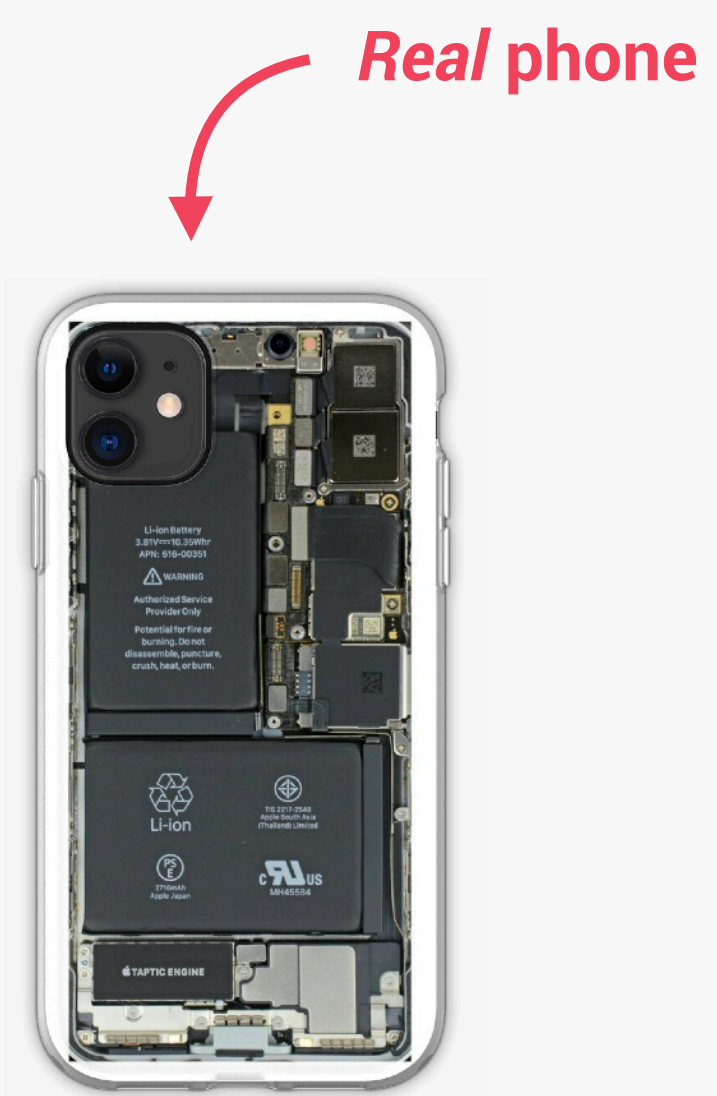
Abstraction

Encapsulation

Inheritance

Polymorphism

```
Phone {  
  charge  
  volume  
  voltage  
  temperature  
  
  homeBtn() {}  
  volumeBtn() {}  
  screen() {}  
  verifyVolt() {}  
  verifyTemp() {}  
  vibrate() {}  
  soundSpeaker() {}  
  soundEar() {}  
  frontCamOn() {}  
  frontCamOff() {}  
  rearCamOn() {}  
  rearCamOff() {}  
}
```



```
Phone {  
  charge  
  volume  
  
  homeBtn() {}  
  volumeBtn() {}  
  screen() {}  
}
```

Do we really need all these low-level details?

Details have been abstracted away

👉 **Abstraction:** Ignoring or hiding details that **don't matter**, allowing us to get an **overview** perspective of the *thing* we're implementing, instead of messing with details that don't really matter to our implementation.

# PRINCIPLE 2: ENCAPSULATION

Abstraction

Encapsulation

Inheritance

Polymorphism

NOT accessible from  
outside the class!

STILL accessible from  
within the class!

STILL accessible from  
within the class!

NOT accessible from  
outside the class!

```
User {  
  user  
  private password  
  private email  
  
  login(word) {  
    this.password === word  
  }  
  comment(text) {  
    this.checkSPAM(text)  
  }  
  private checkSPAM(text) {  
    // Verify logic  
  }  
}
```

Again, NOT actually JavaScript  
syntax (the private keyword  
doesn't exist)

WHY?

- 👉 Prevents external code from accidentally manipulating internal properties/state
- 👉 Allows to change internal implementation without the risk of breaking external code

👉 **Encapsulation:** Keeping properties and methods **private** inside the class, so they are **not accessible from outside the class**. Some methods can be **exposed** as a public interface (API).



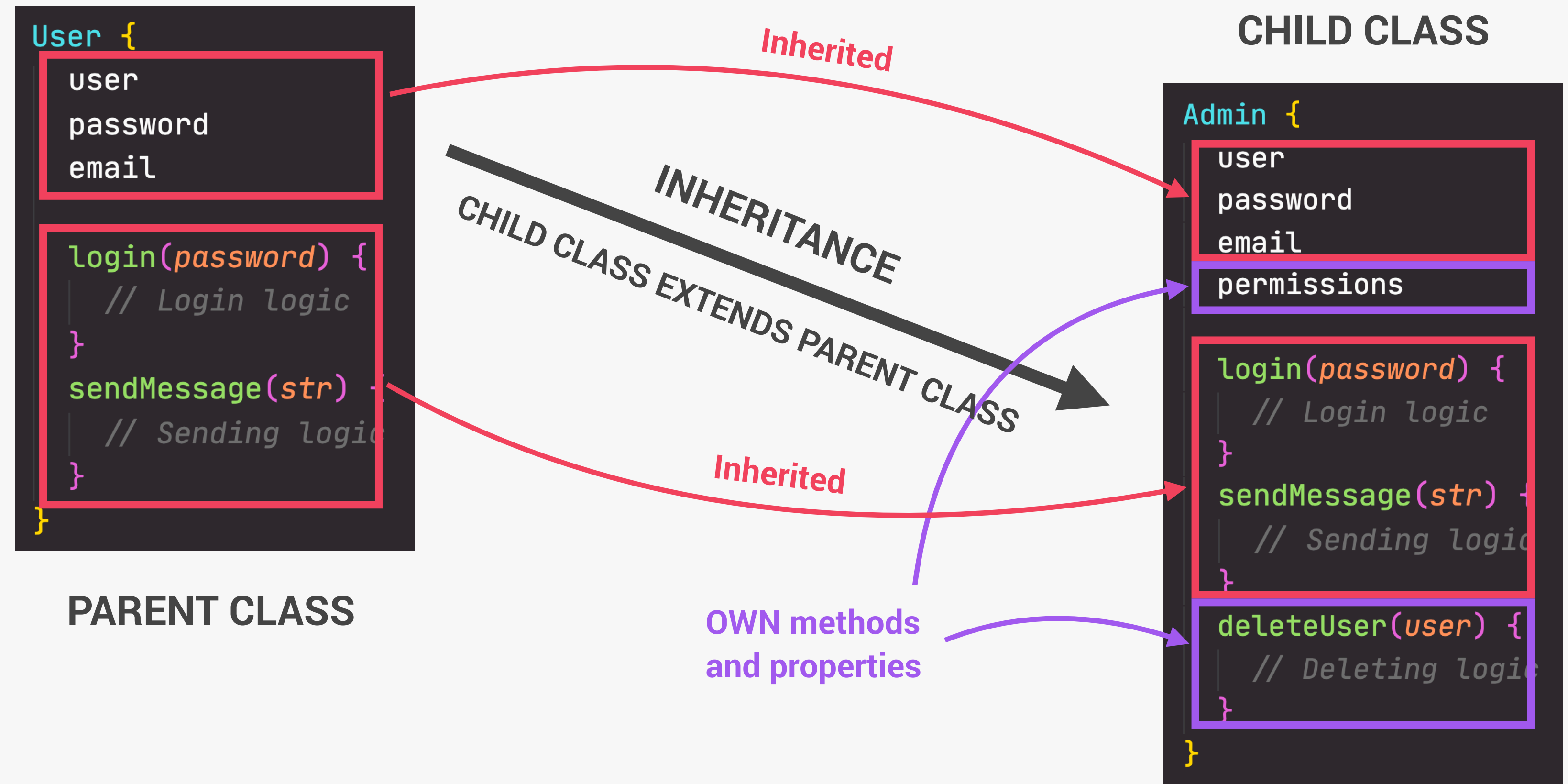
# PRINCIPLE 3: INHERITANCE

Abstraction

Encapsulation

Inheritance

Polymorphism



👉 **Inheritance:** Making all properties and methods of a certain class **available to a child class**, forming a hierarchical relationship between classes. This allows us to **reuse common logic** and to model real-world relationships.

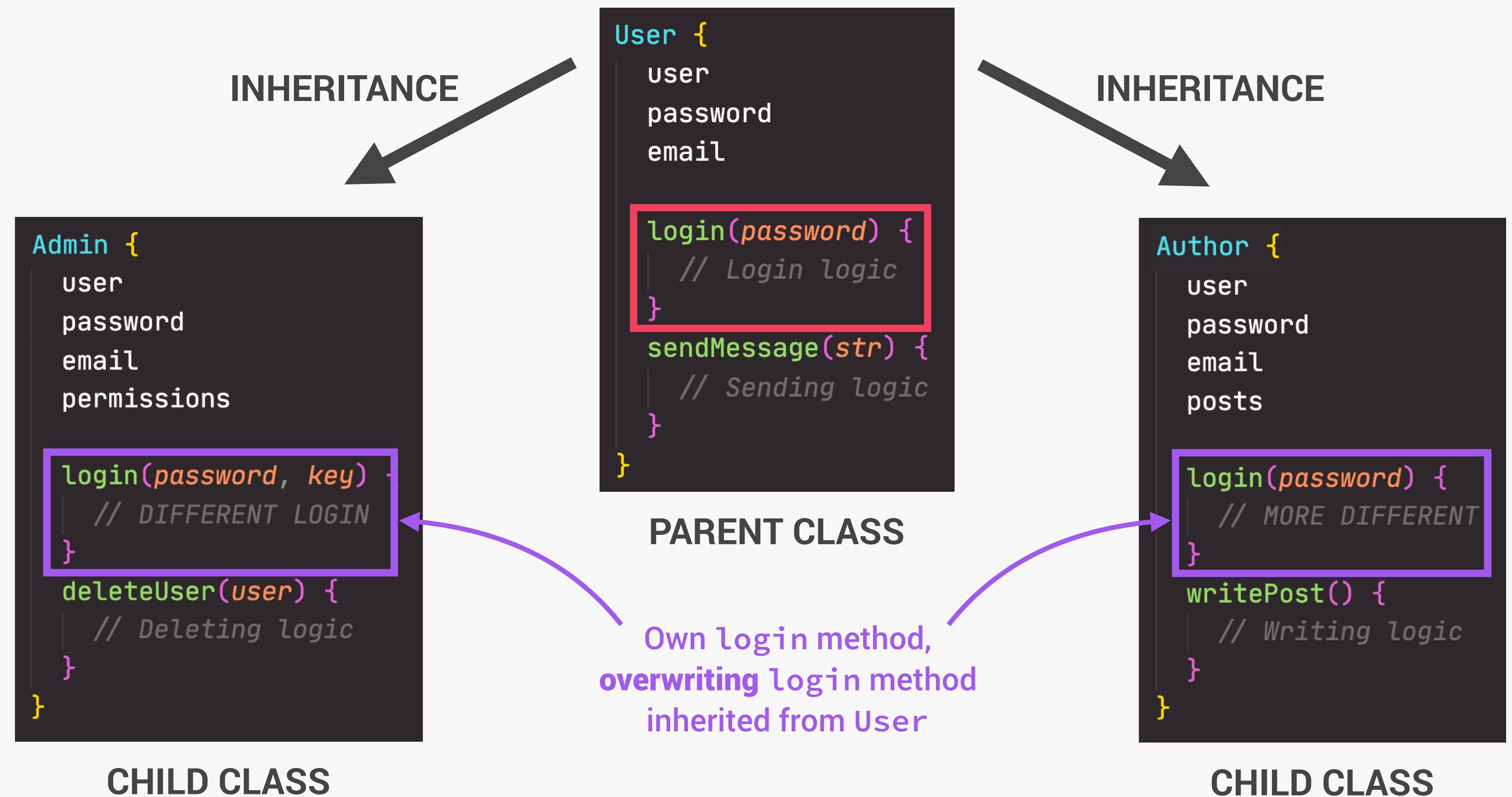
# PRINCIPLE 4: POLYMORPHISM

Abstraction

Encapsulation

Inheritance

Polymorphism



👉 **Polymorphism:** A child class can **overwrite** a method it inherited from a parent class [it's more complex than that, but enough for our purposes].







JONAS.IO  
SCHMEDTMANN

# THE COMPLETE JAVASCRIPT COURSE

FROM ZERO TO EXPERT!



@JONASSCHMEDTMAN

SECTION

OBJECT ORIENTED PROGRAMMING  
(OOP) WITH JAVASCRIPT

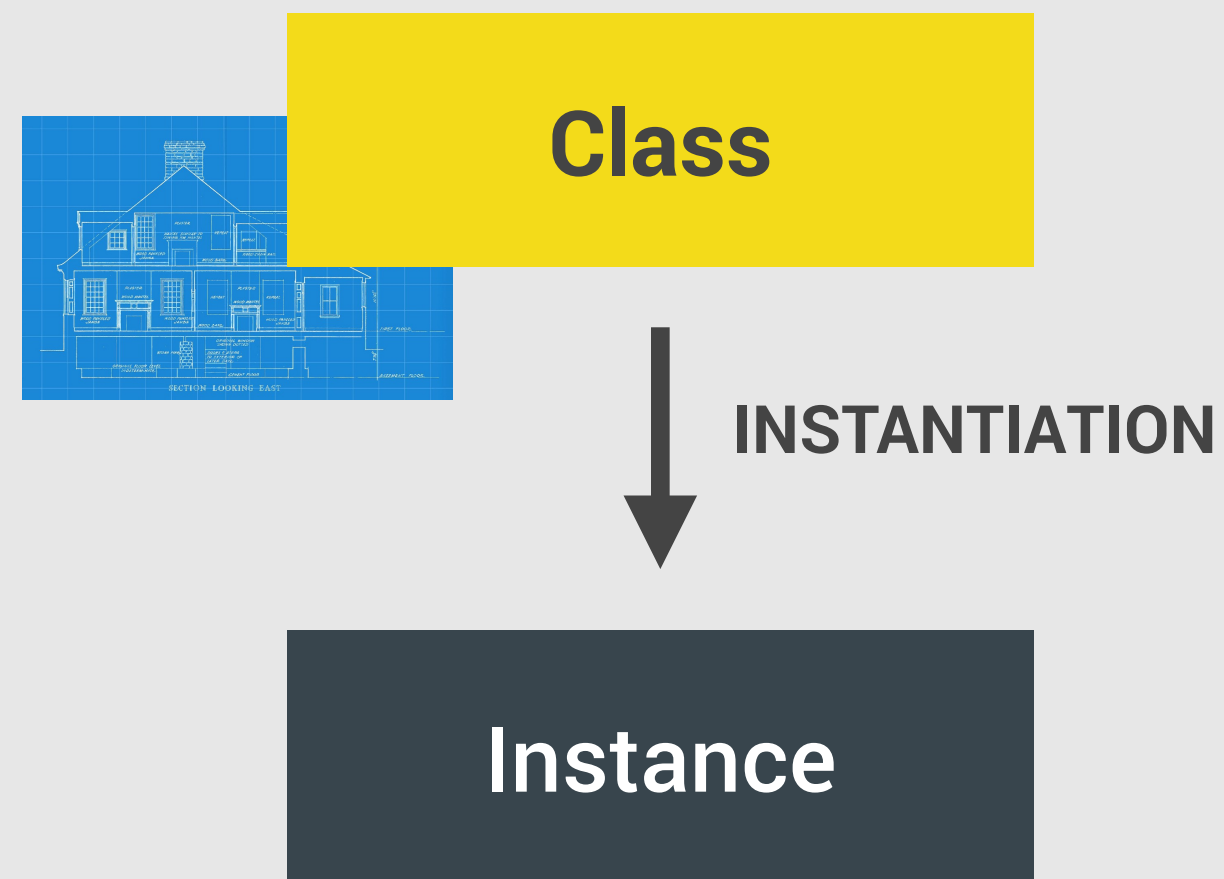
LECTURE

OOP IN JAVASCRIPT

JS

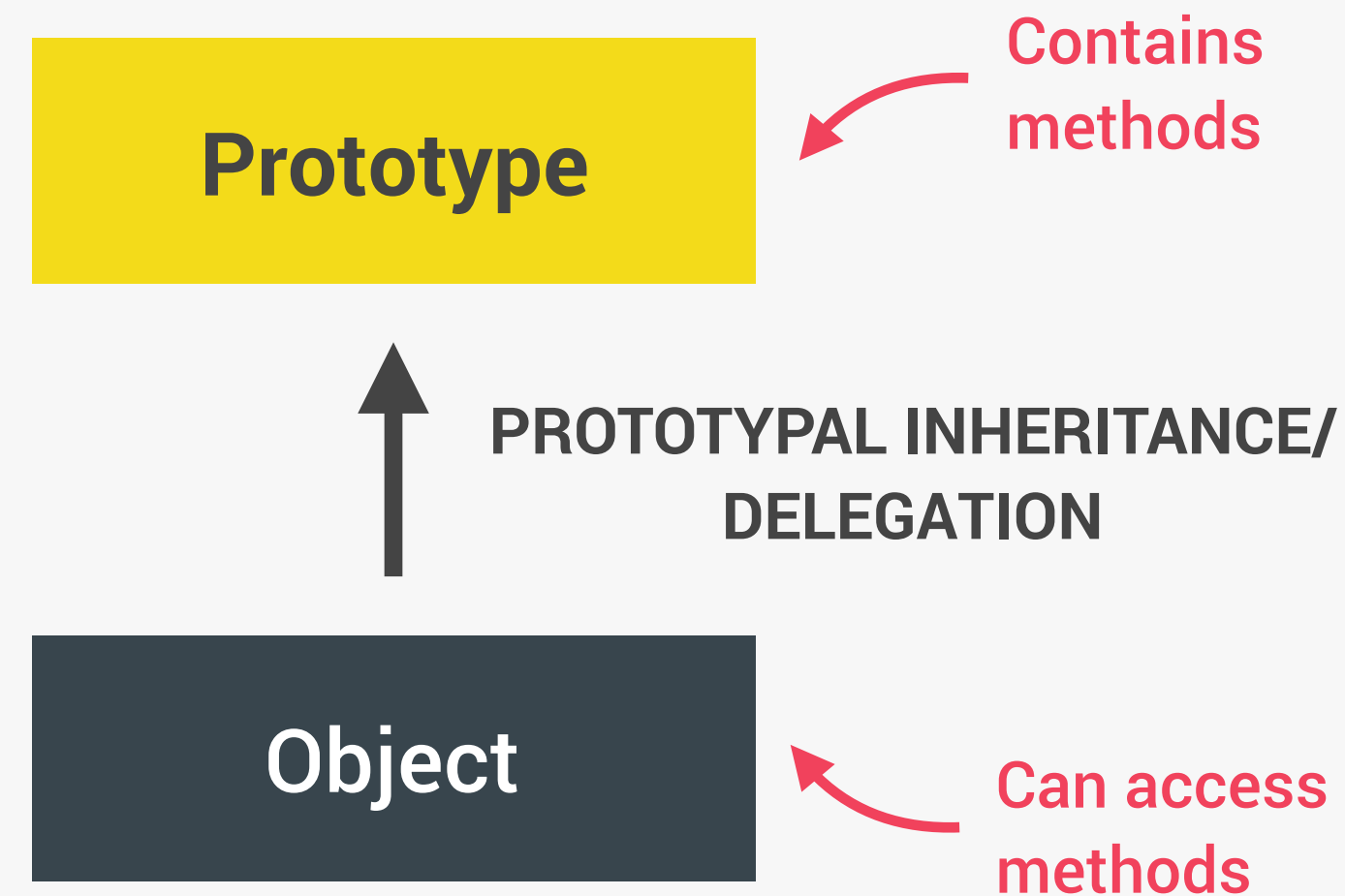
# OOP IN JAVASCRIPT: PROTOTYPES

## "CLASSICAL OOP": CLASSES



- 👉 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



- 👉 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!**

```
▼ f Array() ⓘ  
  arguments: (...)  
  caller: (...)  
  length: 1  
  name: "Array"  
  prototype: Array(0)  
    ▶ unique: f ()  
      length: 0  
    ▶ constructor: f Array()  
    ▶ concat: f concat()  
    ▶ map: f 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

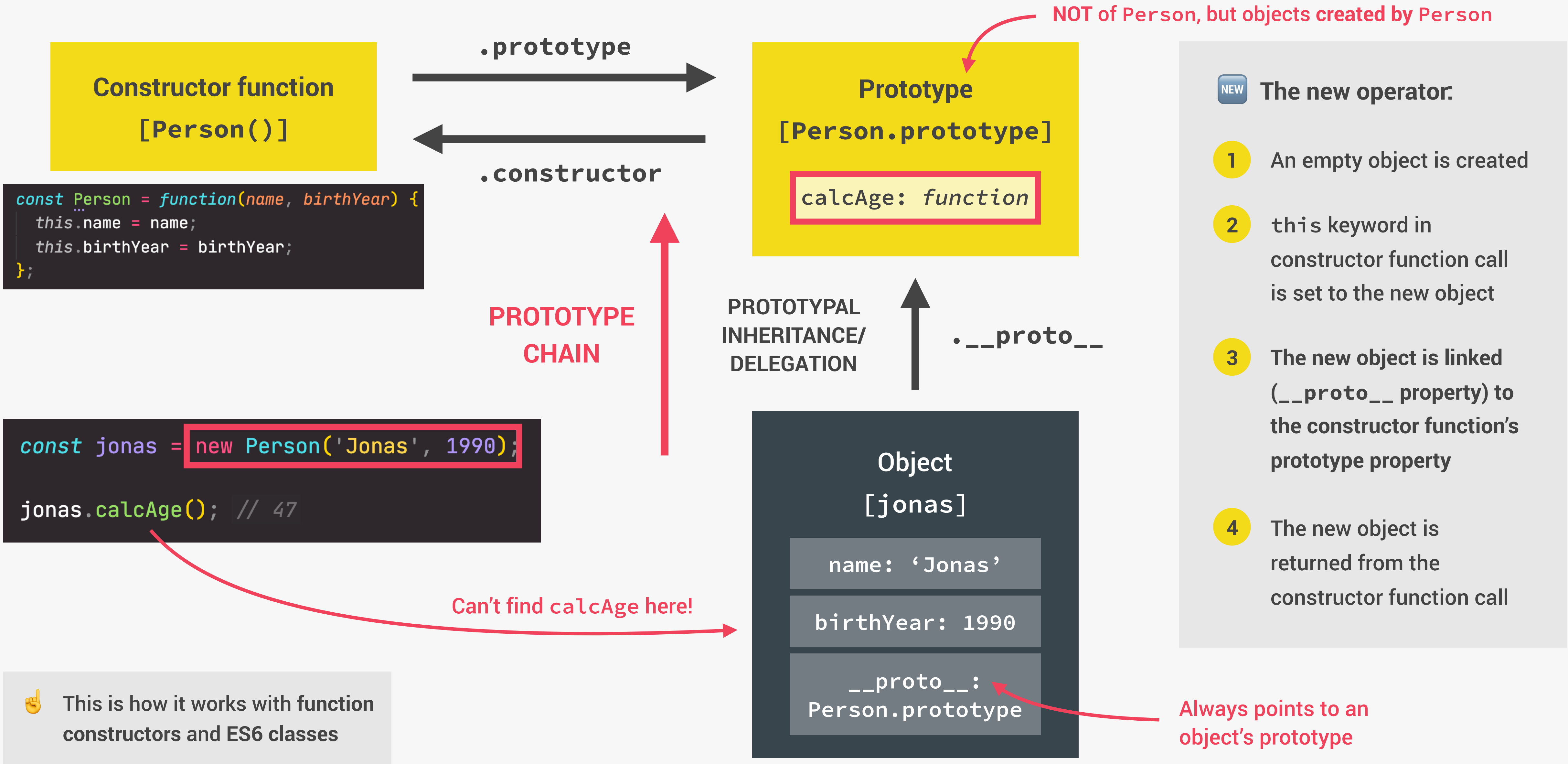
OBJECT ORIENTED PROGRAMMING  
(OOP) WITH JAVASCRIPT

## LECTURE

PROTOTYPAL INHERITANCE AND THE  
PROTOTYPE CHAIN

JS

# HOW PROTOTYPAL INHERITANCE / DELEGATION WORKS











JONAS.IO  
SCHMEDTMANN

# THE COMPLETE JAVASCRIPT COURSE

FROM ZERO TO EXPERT!



@JONASSCHMEDTMAN

SECTION

OBJECT ORIENTED PROGRAMMING  
(OOP) WITH JAVASCRIPT

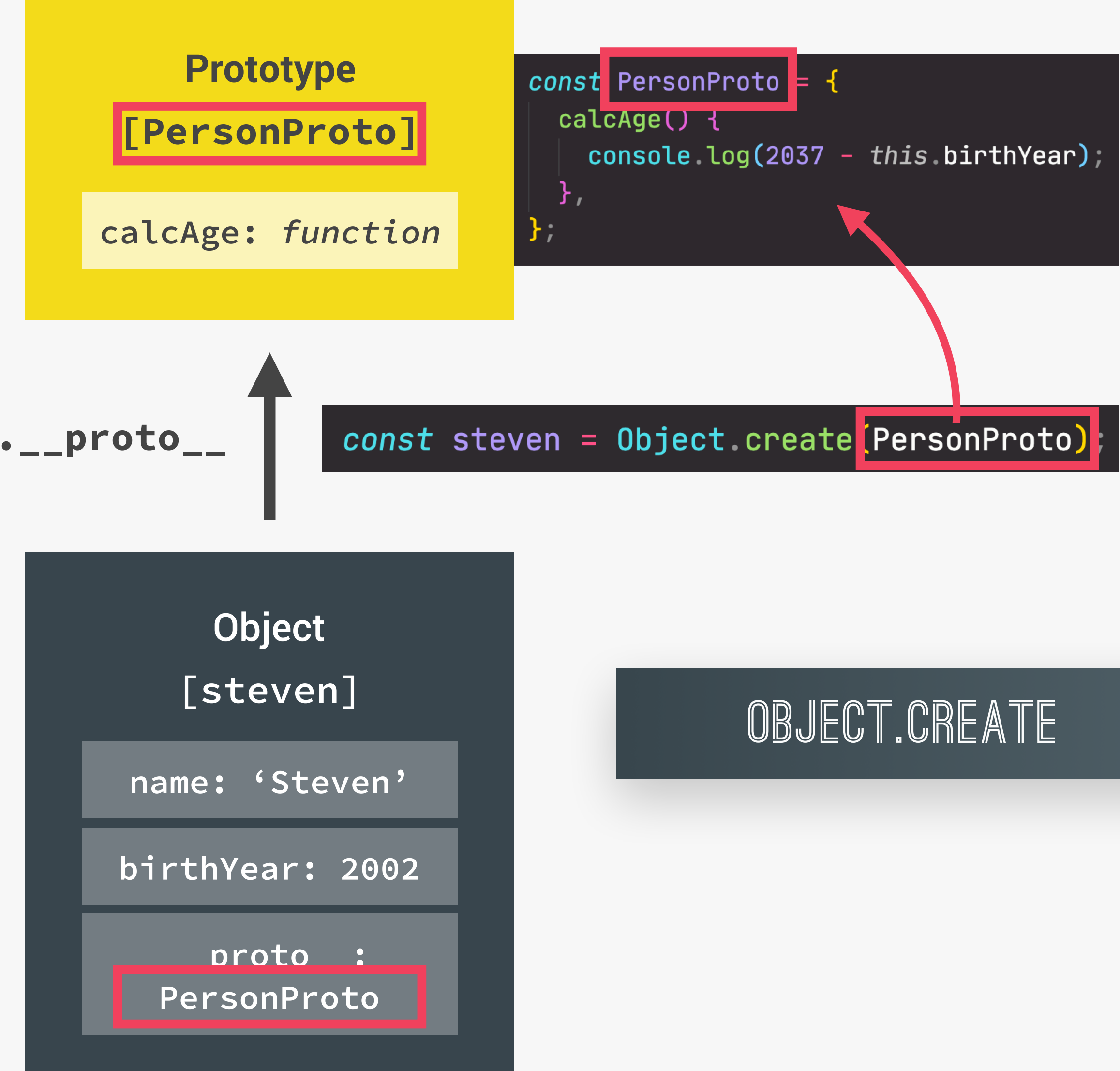
LECTURE

OBJECT.CREATE

JS

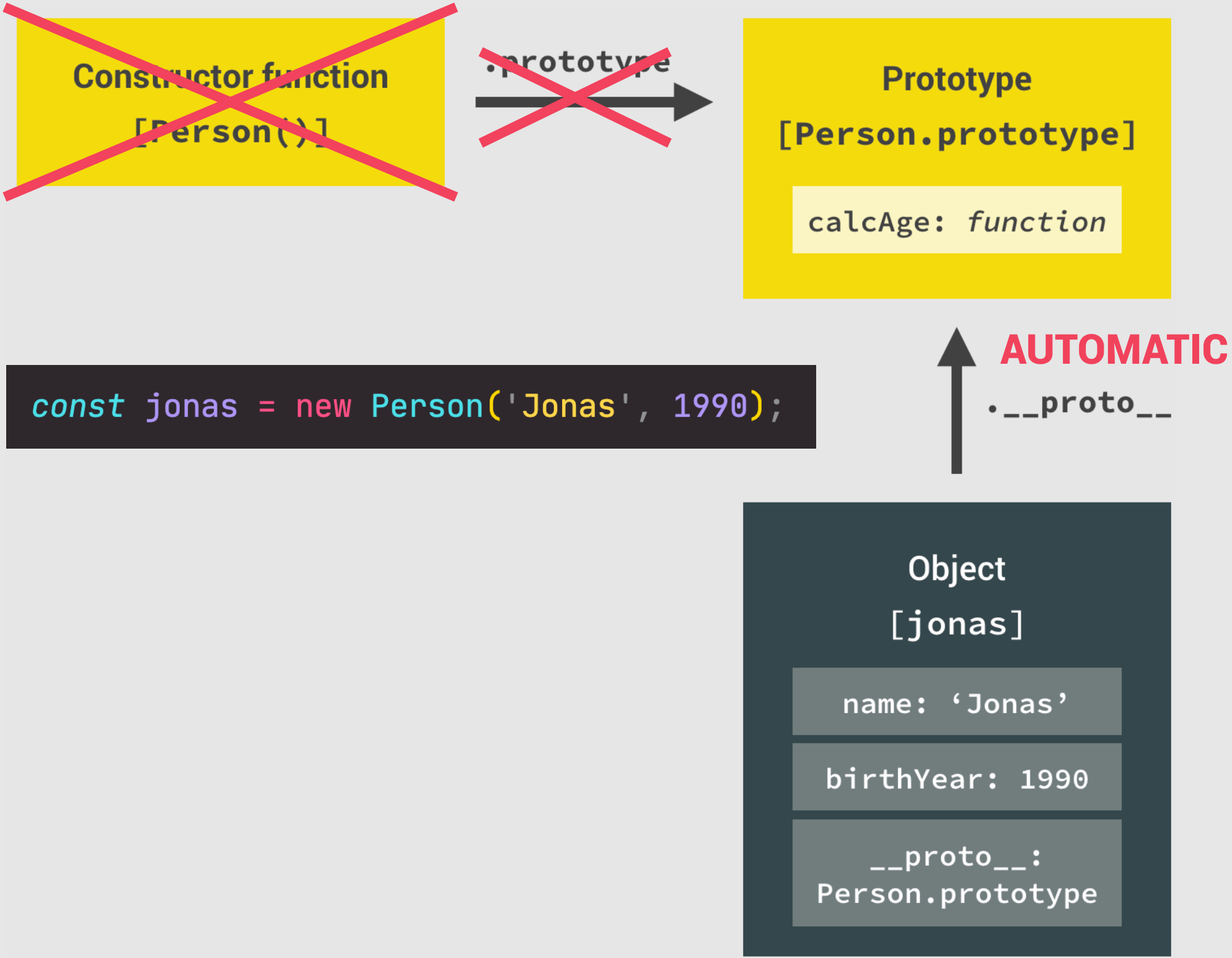


# HOW OBJECT.CREATE WORKS



## OBJECT.CREATE

## CONSTRUCTOR FUNCTIONS







JONAS.IO  
SCHMEDTMANN

# THE COMPLETE JAVASCRIPT COURSE

FROM ZERO TO EXPERT!



@JONASSCHMEDTMAN

## SECTION

OBJECT ORIENTED PROGRAMMING  
(OOP) WITH JAVASCRIPT

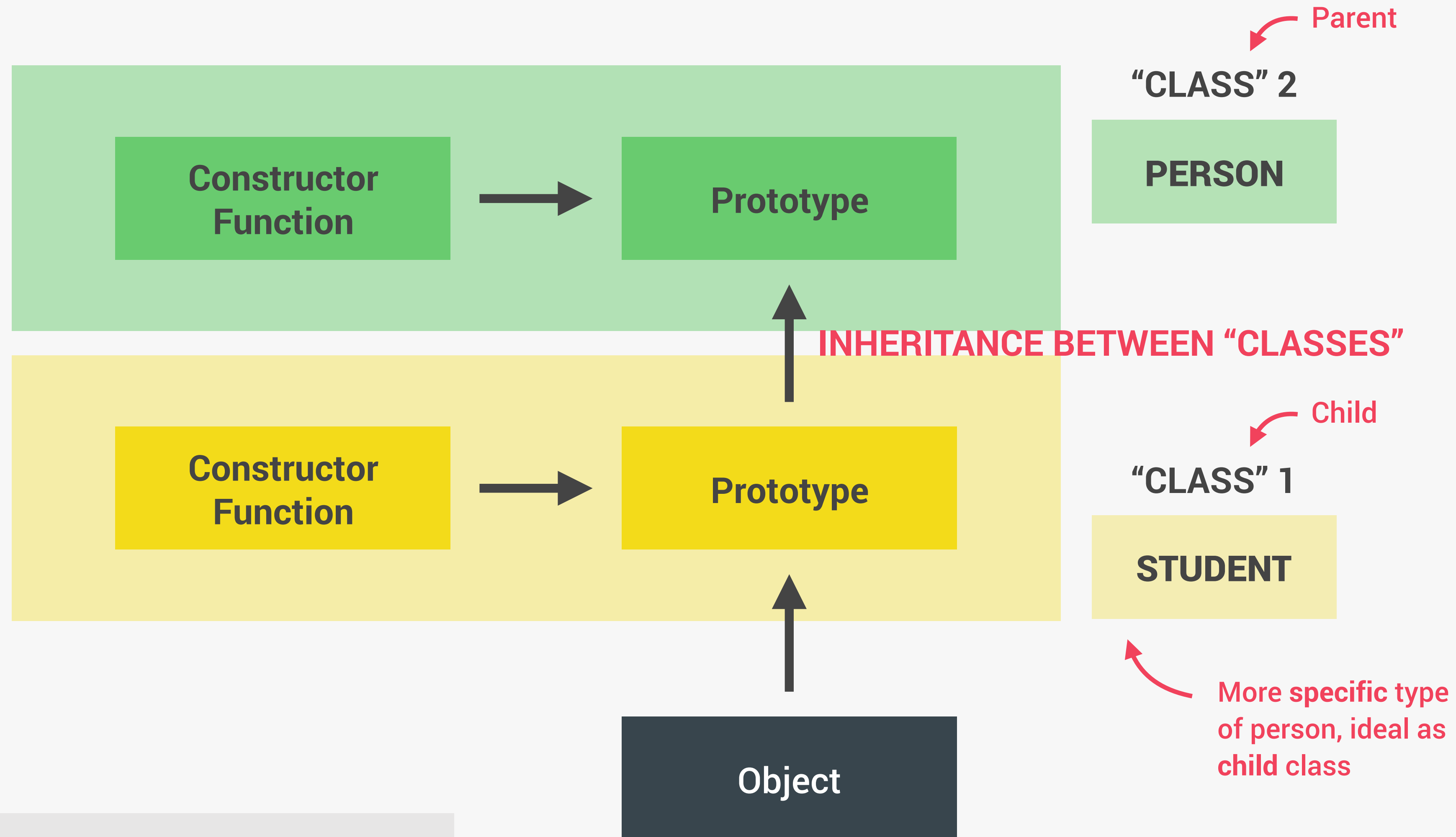
## LECTURE

INHERITANCE BETWEEN "CLASSES":  
CONSTRUCTOR FUNCTIONS

JS



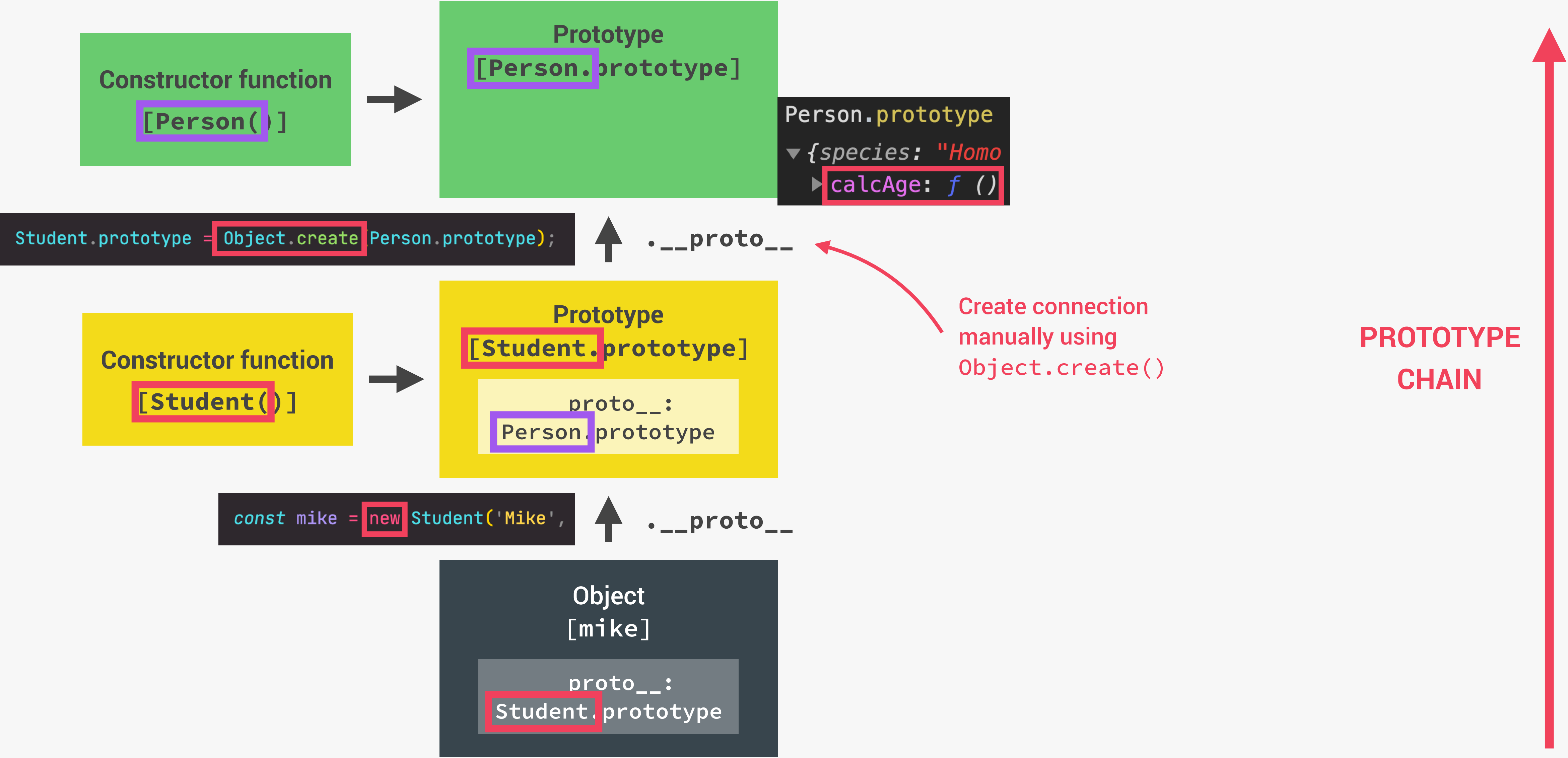
# INHERITANCE BETWEEN "CLASSES"



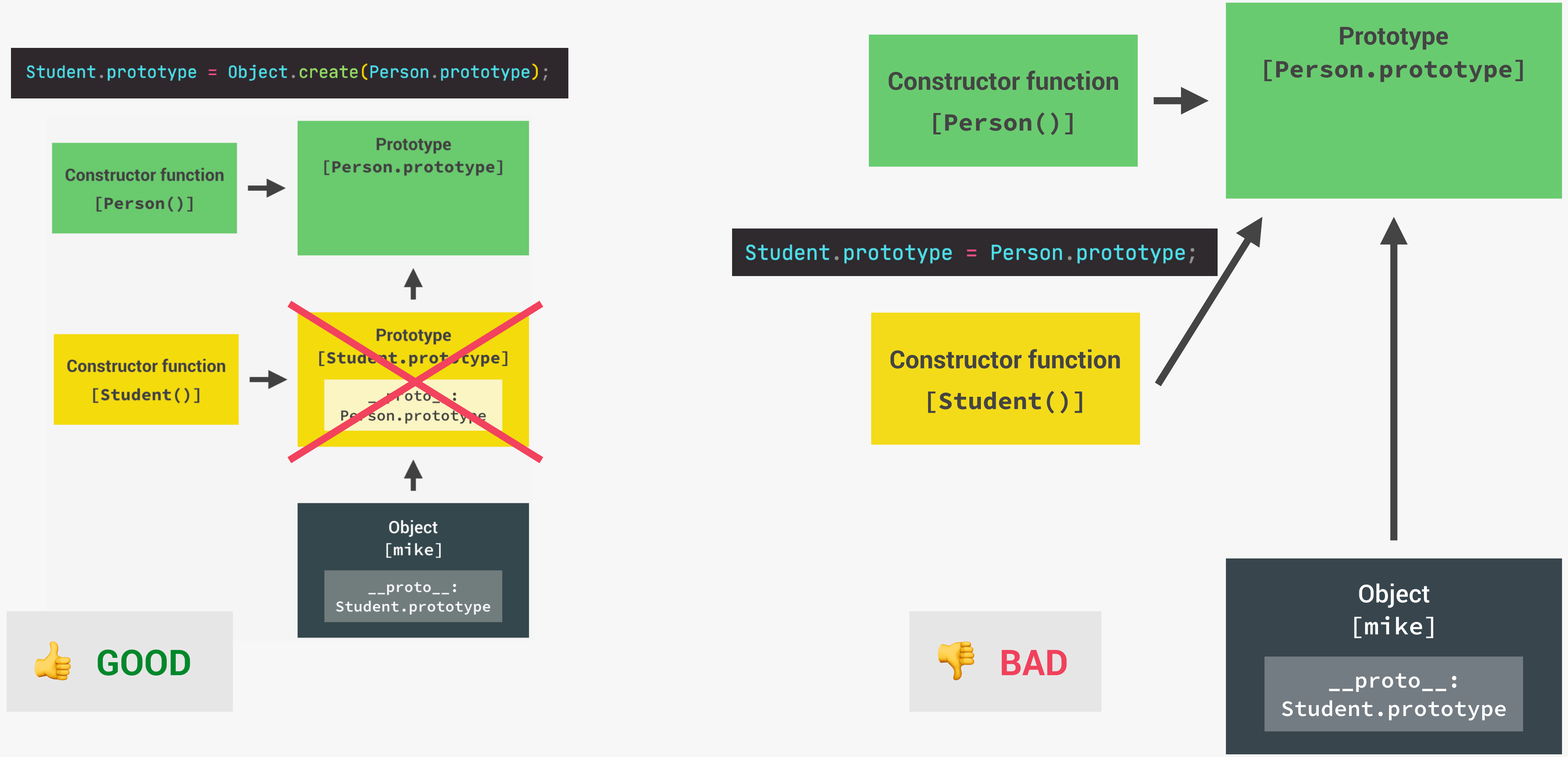
👉 Using class terminology here to make it easier to understand.

- 1 Constructor functions
- 2 ES6 Classes
- 3 `Object.create()`

# INHERITANCE BETWEEN "CLASSES"

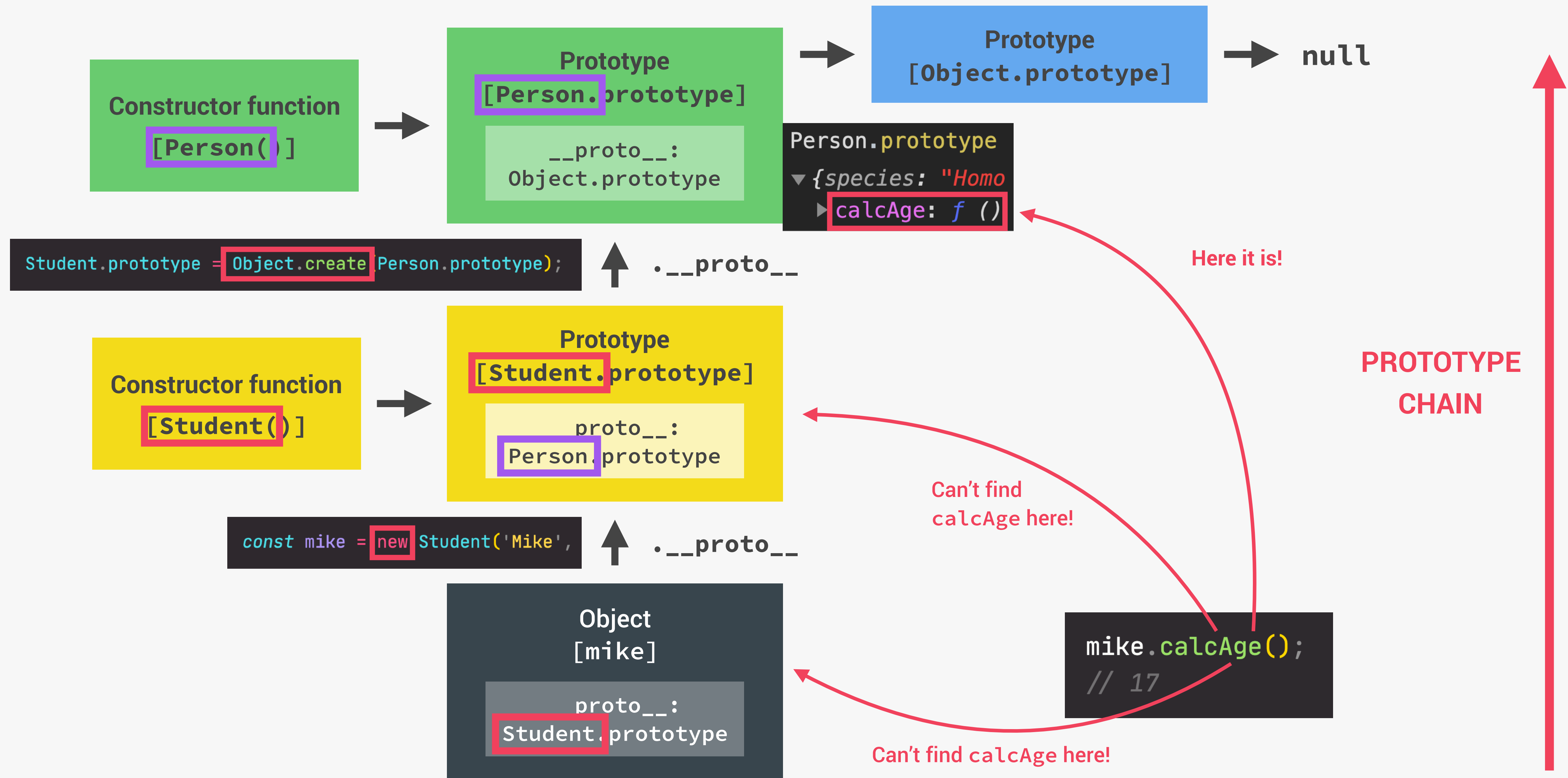


# INHERITANCE BETWEEN "CLASSES"





# INHERITANCE BETWEEN "CLASSES"







JONAS.IO  
SCHMEDTMANN

# THE COMPLETE JAVASCRIPT COURSE

FROM ZERO TO EXPERT!



@JONASSCHMEDTMAN

## SECTION

OBJECT ORIENTED PROGRAMMING  
(OOP) WITH JAVASCRIPT

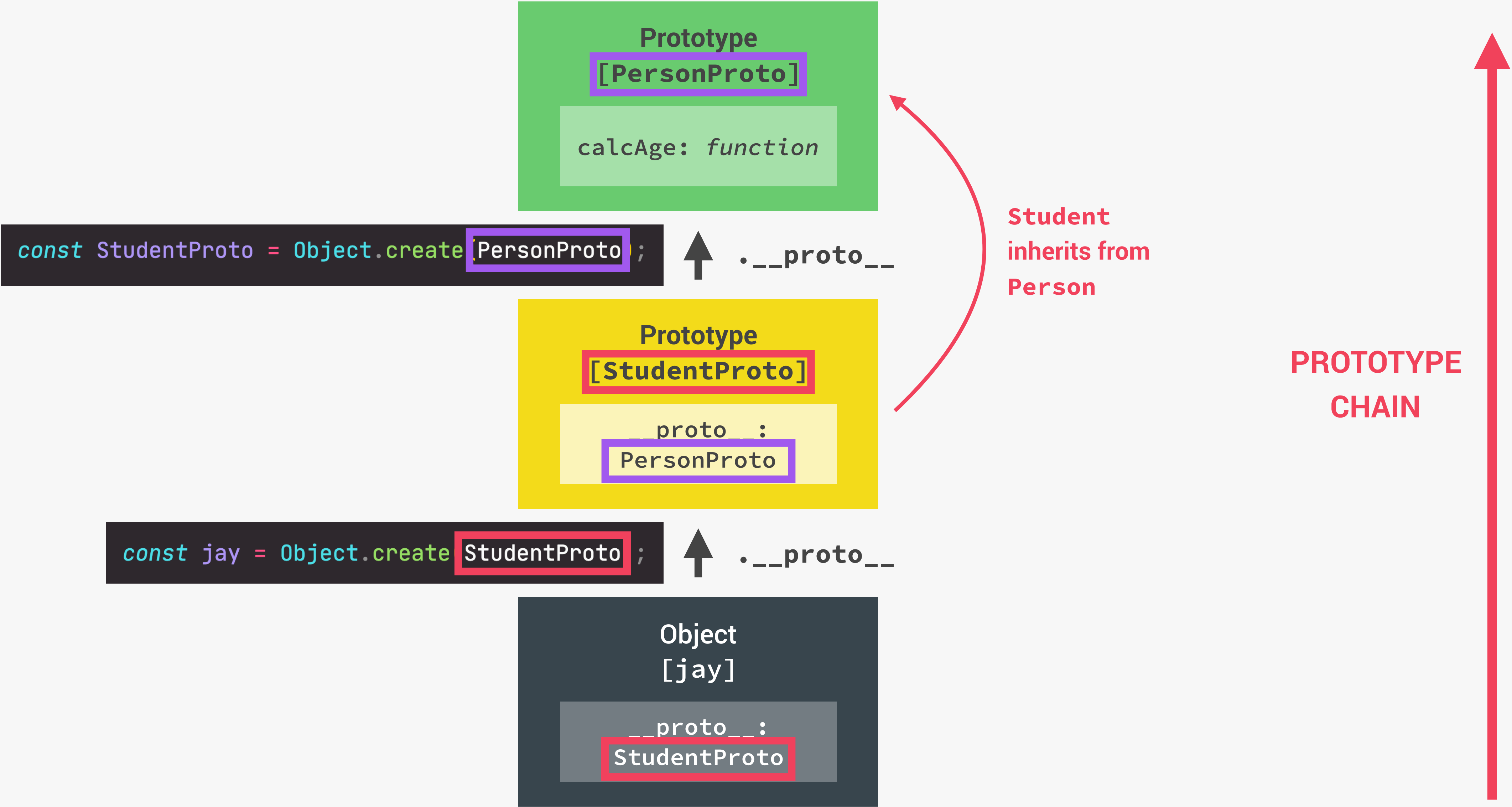
## LECTURE

INHERITANCE BETWEEN "CLASSES":  
OBJECT.CREATE





# 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**



MAPTYP APP. OOP,  
GEOLOCATION,  
EXTERNAL LIBRARIES,  
AND MORE!





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

# PROJECT PLANNING

Description of the application's functionality from the user's perspective. All user stories put together describe the entire application.

WHAT we will build

HOW we will build it



PLANNING STEP

Implementation of our plan using code

👉 **User story:** Description of the application's functionality from the user's perspective.

👉 **Common format:** As a *[type of user]*, I want *[an action]* so that *[a benefit]*

Who?

Example: user, admin, etc.

What?

Why?

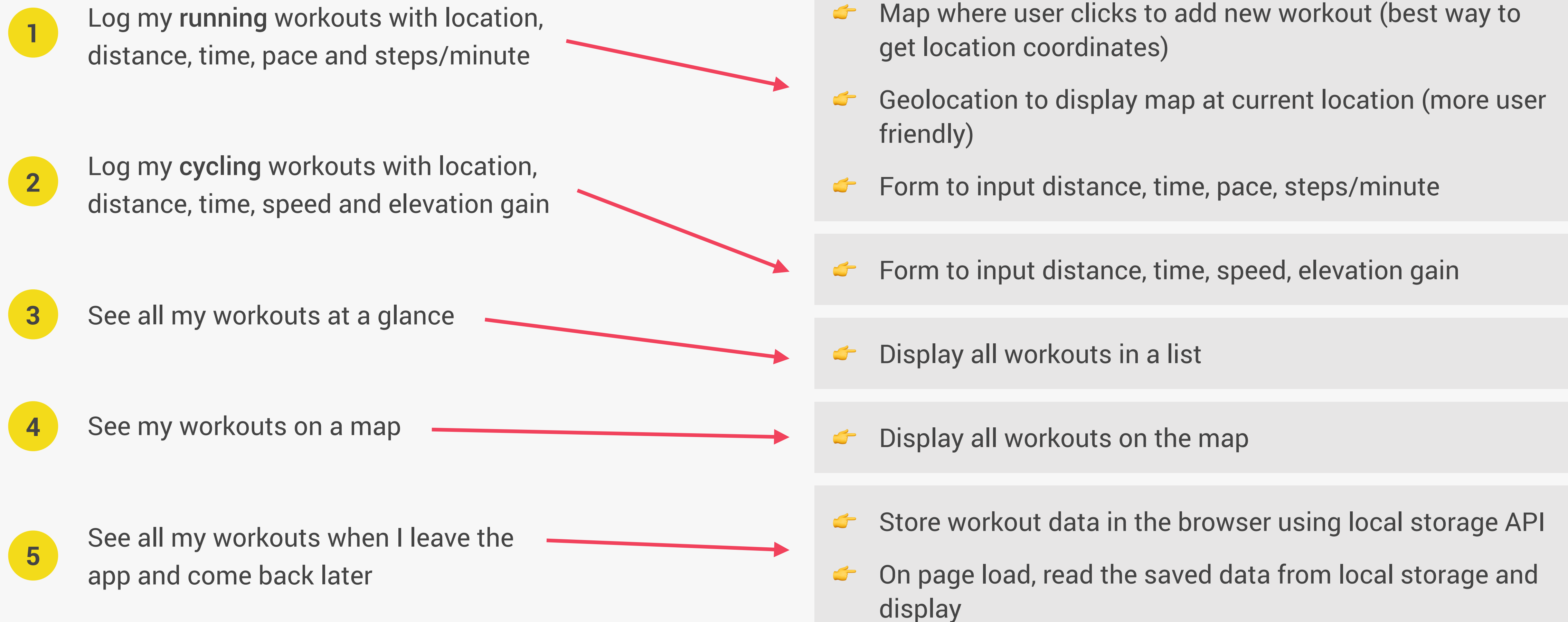
- 1 As a user, I want to log my running workouts with location, distance, time, pace and steps/minute, so I can keep a log of all my running
- 2 As a user, I want to log my cycling workouts with location, distance, time, speed and elevation gain, so I can keep a log of all my cycling
- 3 As a user, I want to see all my workouts at a glance, so I can easily track my progress over time
- 4 As a user, I want to also see my workouts on a map, so I can easily check where I work out the most
- 5 As a user, I want to see all my workouts when I leave the app and come back later, so that I can keep using there app over time



## 2. FEATURES

### USER STORIES

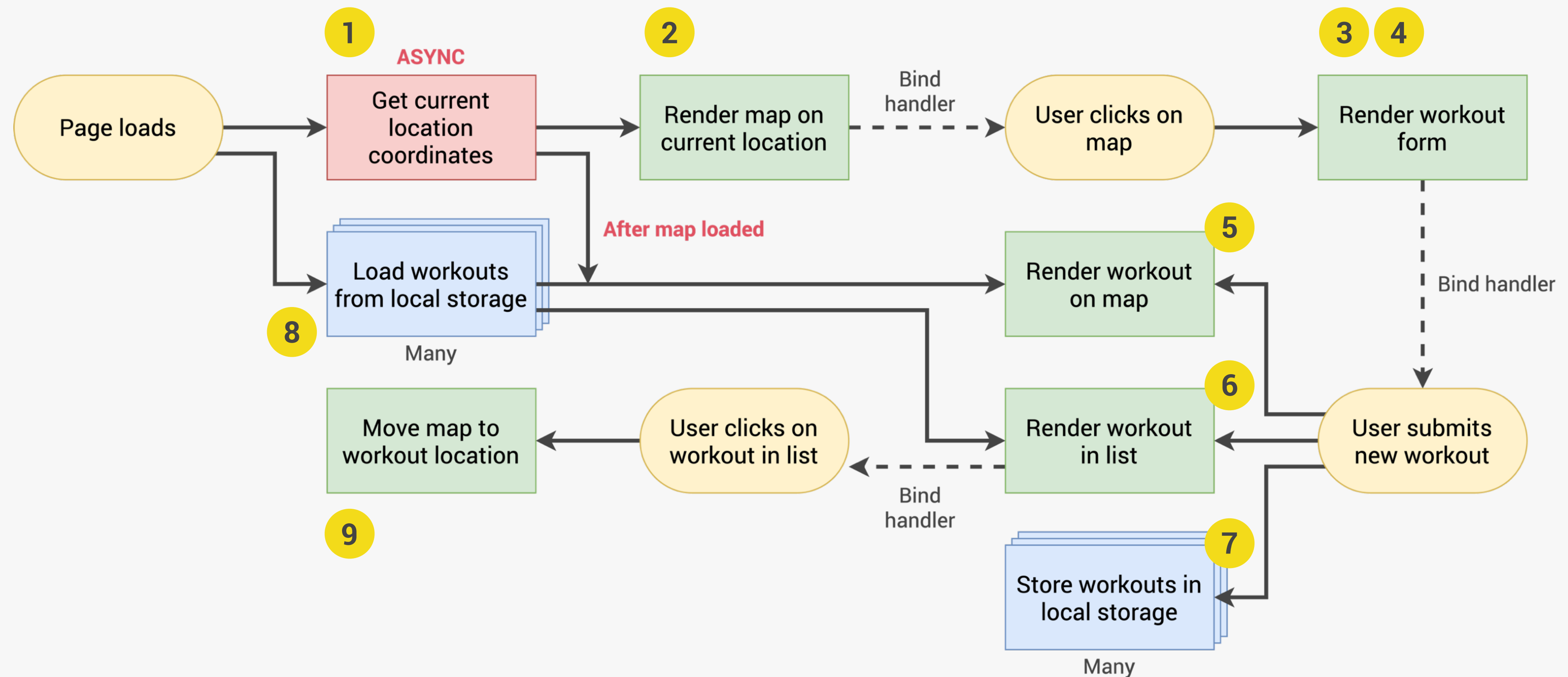
### FEATURES



# 3. FLOWCHART

## 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



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!

Added later

FOR NOW, LET'S JUST  
START CODING 🥰







JONAS.IO  
SCHMEDTMANN

# THE COMPLETE JAVASCRIPT COURSE

FROM ZERO TO EXPERT!



@JONASSCHMEDTMAN

SECTION

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

LECTURE

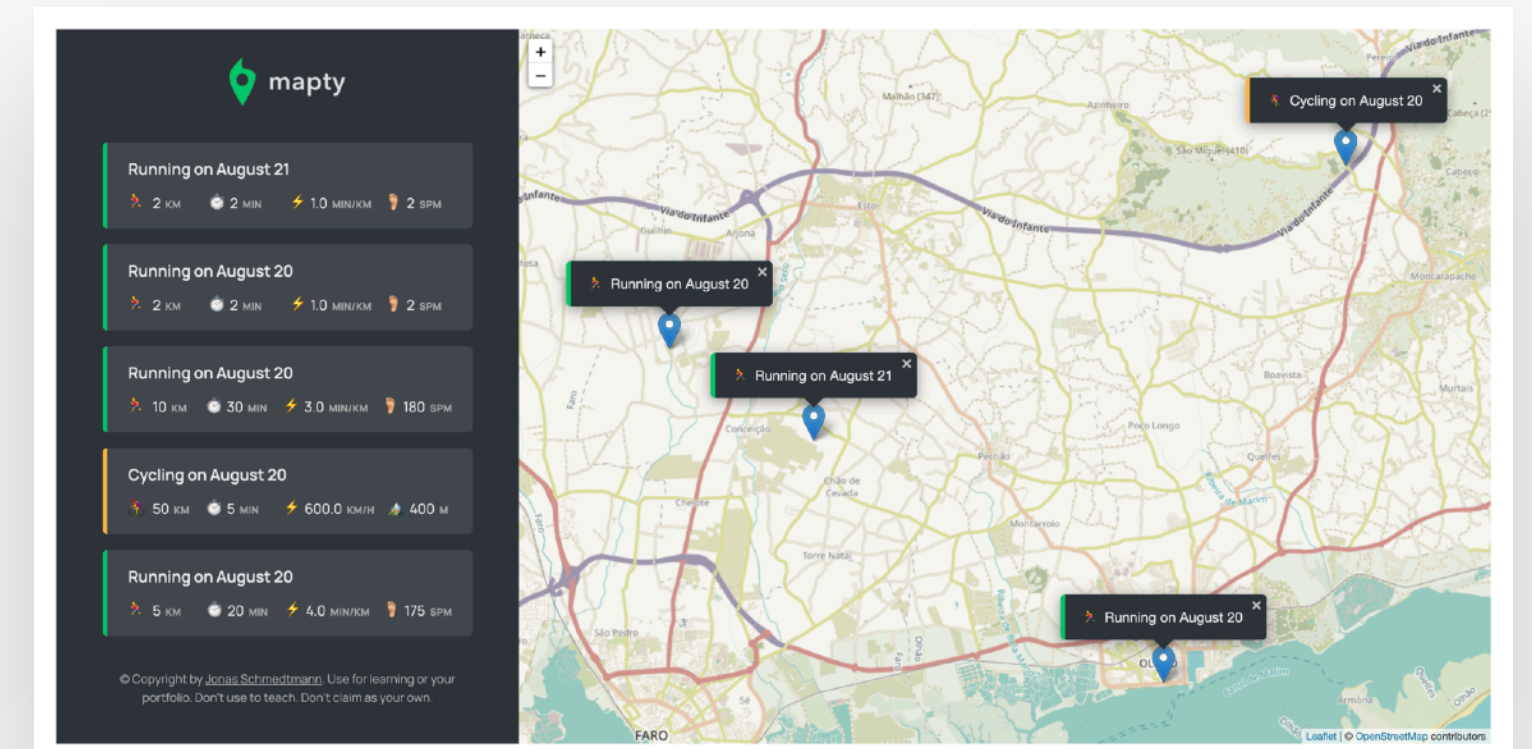
FINAL CONSIDERATIONS

JS



# 10 ADDITIONAL FEATURE IDEAS: CHALLENGES 🧐

- 👉 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].







ASYNCHRONOUS  
JAVASCRIPT:  
PROMISES, ASYNC/  
AWAIT AND AJAX



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



# SYNCHRONOUS CODE

BLOCKING

```
const p = document.querySelector('.p');  
p.textContent = 'My name is Jonas!';  
alert('Text set!');  
p.style.color = 'red';
```

127.0.0.1:8080 says  
Text set!

OK

THREAD OF  
EXECUTION



SYNCHRONOUS

- 👉 Most code is **synchronous**;
- 👉 Synchronous code is **executed line by line**;
- 👉 Each line of code **waits** for previous line to finish;
- 👉 Long-running operations **block** code execution.

Part of execution  
context that actually  
executes the code in  
computer's CPU

# ASYNCHRONOUS CODE

CALLBACK WILL RUN AFTER TIMER

Asynchronous

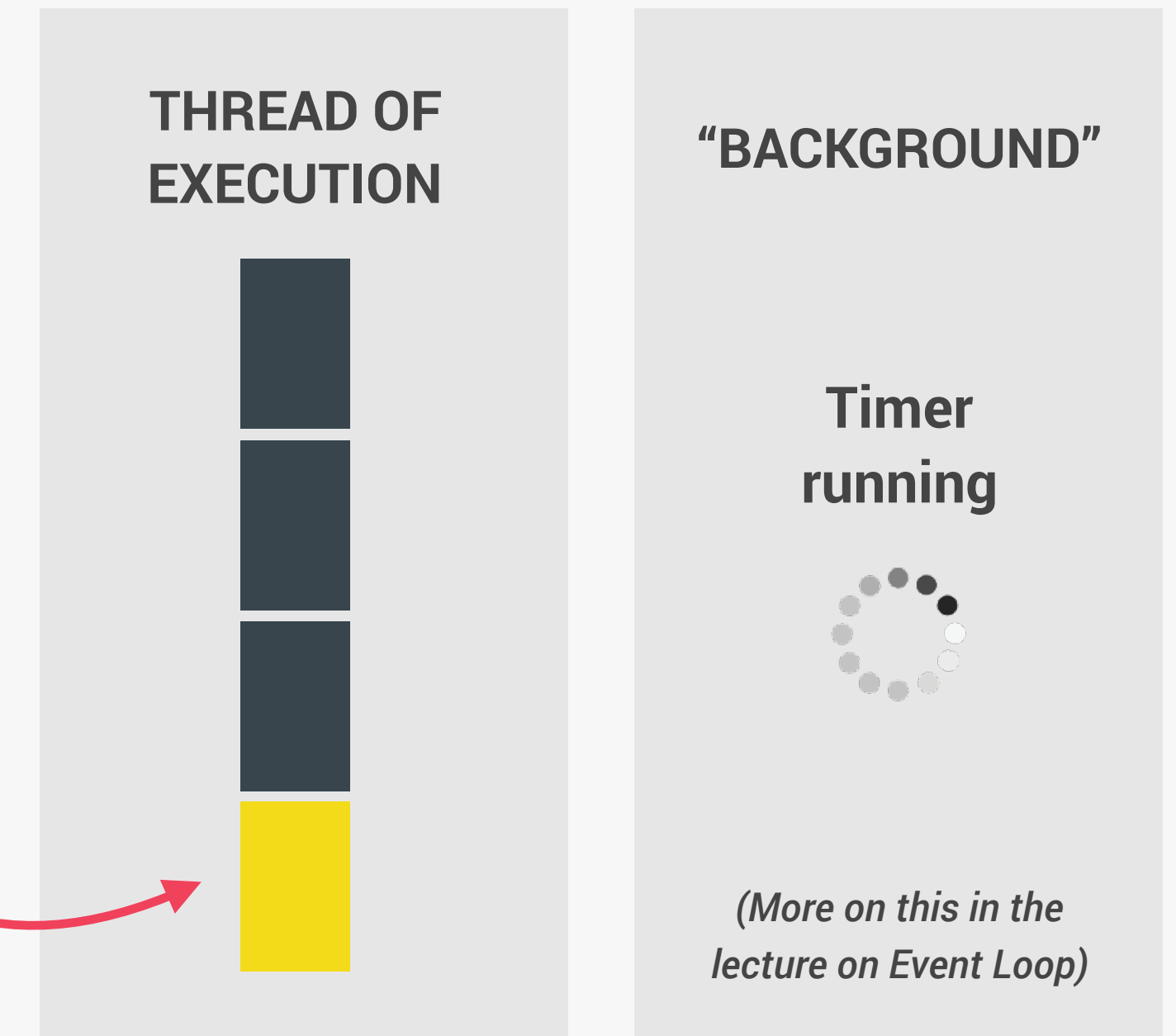
```
const p = document.querySelector('.p');
setTimeout(function () {
  p.textContent = 'My name is Jonas!';
}, 5000);
p.style.color = 'red';
```

👉 Example: Timer with callback

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

Callback does NOT automatically make code asynchronous!

Executed after all other code



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**

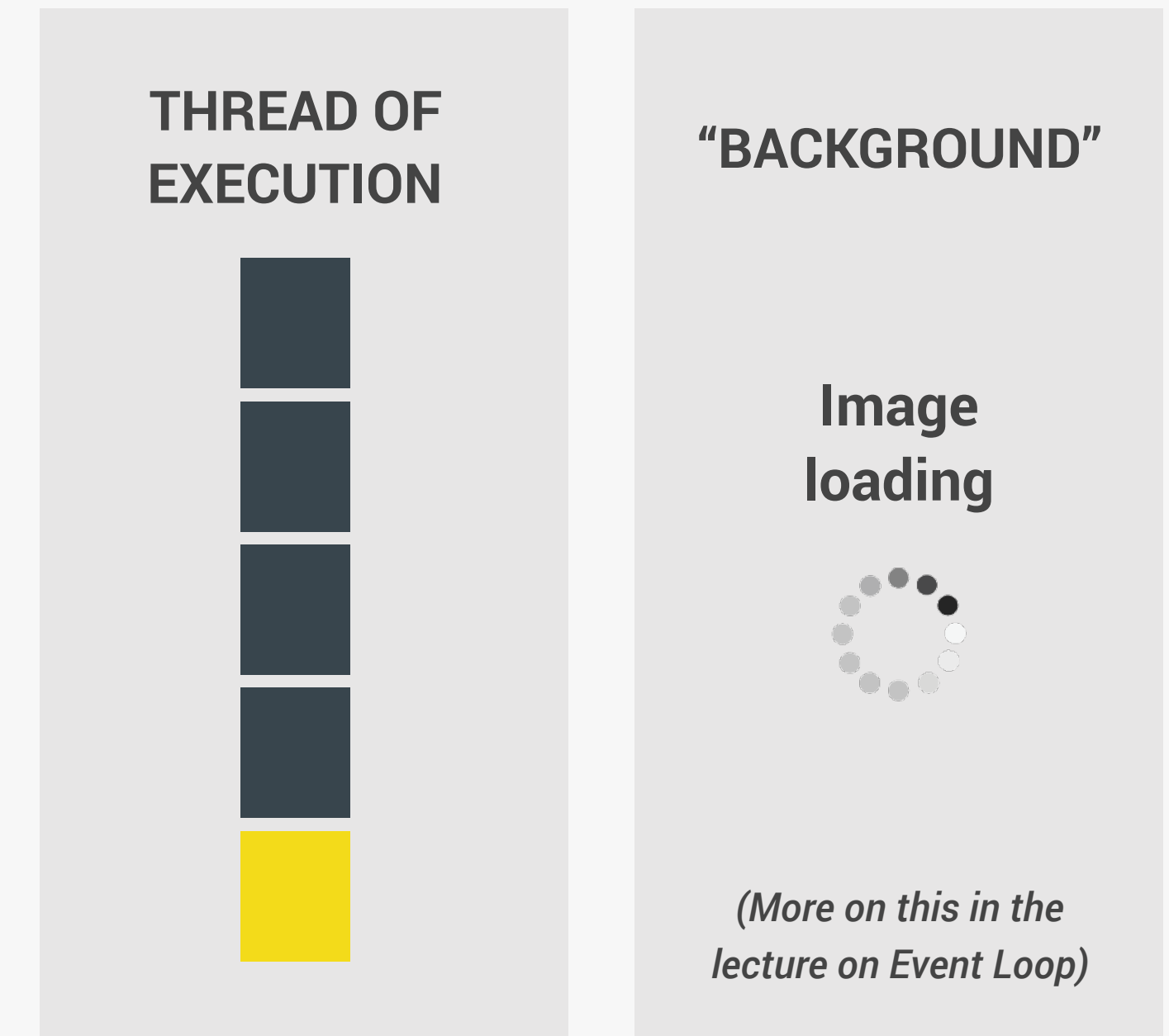
```
const img = document.querySelector('.dog');
img.src = 'dog.jpg';
img.addEventListener('load', function () {
  img.classList.add('fadeIn');
});
p.style.width = '300px';
```

CALLBACK WILL RUN AFTER IMAGE LOADS

👉 Example: Asynchronous image loading with event and callback

👉 Other examples: Geolocation API or AJAX calls

addEventListener does NOT automatically make code asynchronous!



ASYNCHRONOUS

Coordinating behavior of a program over a period of time

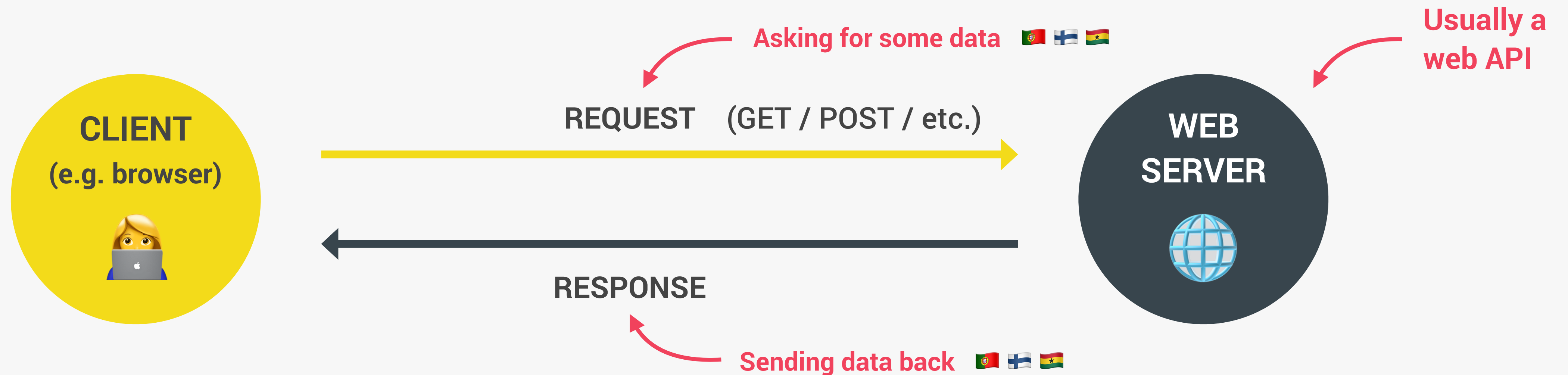
- 👉 Asynchronous code is executed **after** a task that runs in the "background" finishes;
- 👉 Asynchronous code is **non-blocking**;
- 👉 Execution doesn't wait for an asynchronous task to finish its work;
- 👉 Callback functions alone do **NOT** make code asynchronous!



# WHAT ARE AJAX CALLS?

## AJAX

**Asynchronous JavaScript And XML:** 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

👉 **Application Programming Interface:** 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;

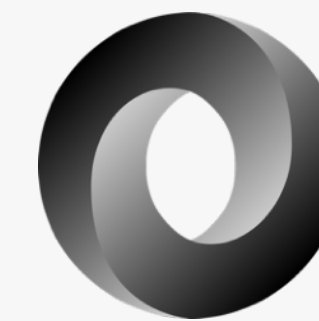
👉 We can build **our own** web APIs (requires back-end development, e.g. with node.js) or use **3rd-party** APIs.



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

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...









JONAS.IO  
SCHMEDTMANN

# THE COMPLETE JAVASCRIPT COURSE

FROM ZERO TO EXPERT!



@JONASSCHMEDTMAN

## SECTION

ASYNCHRONOUS JAVASCRIPT:  
PROMISES, ASYNC/AWAIT AND AJAX

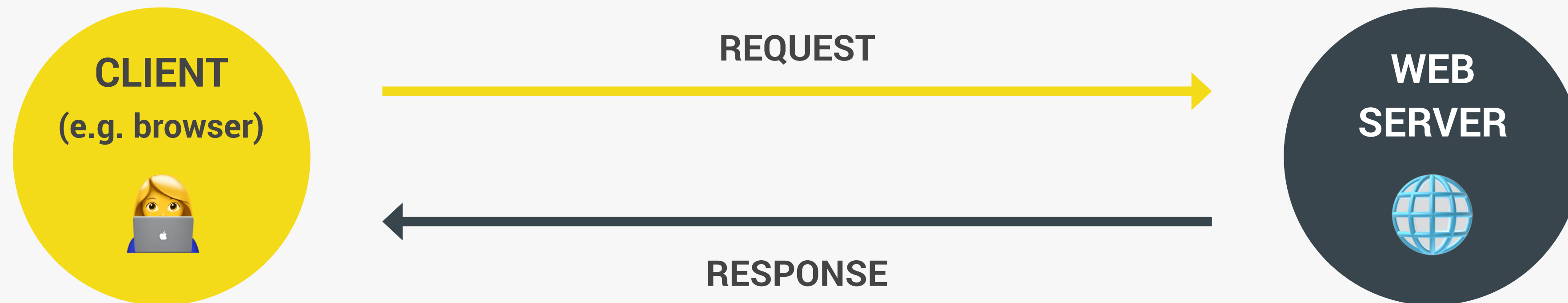
## LECTURE

HOW THE WEB WORKS: REQUESTS  
AND RESPONSES

JS

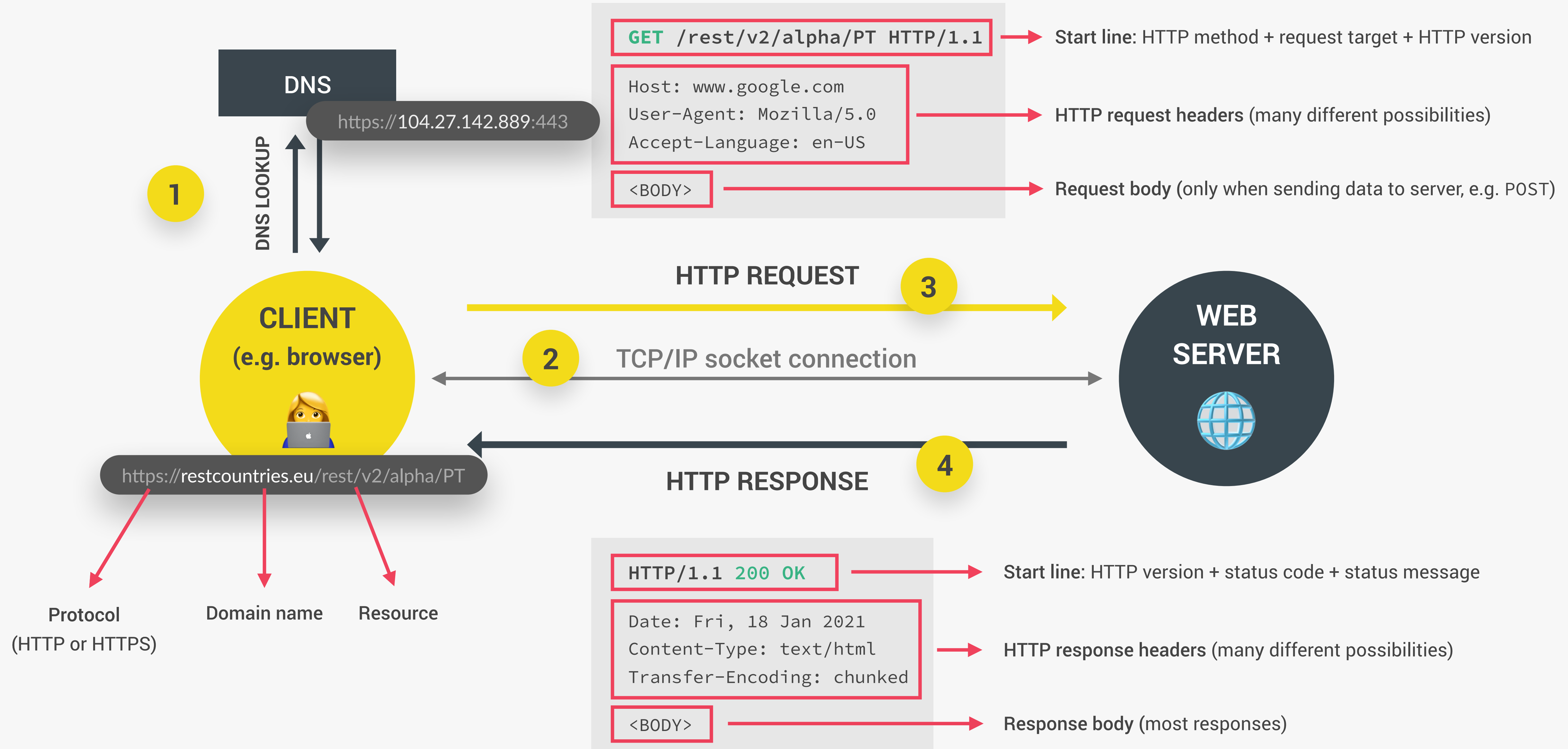
# WHAT HAPPENS WHEN WE ACCESS A WEB SERVER

👉 Request-response model or Client-server architecture





# 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.

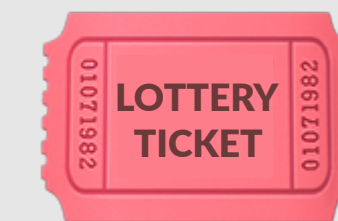
↓ Less formal

👉 **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



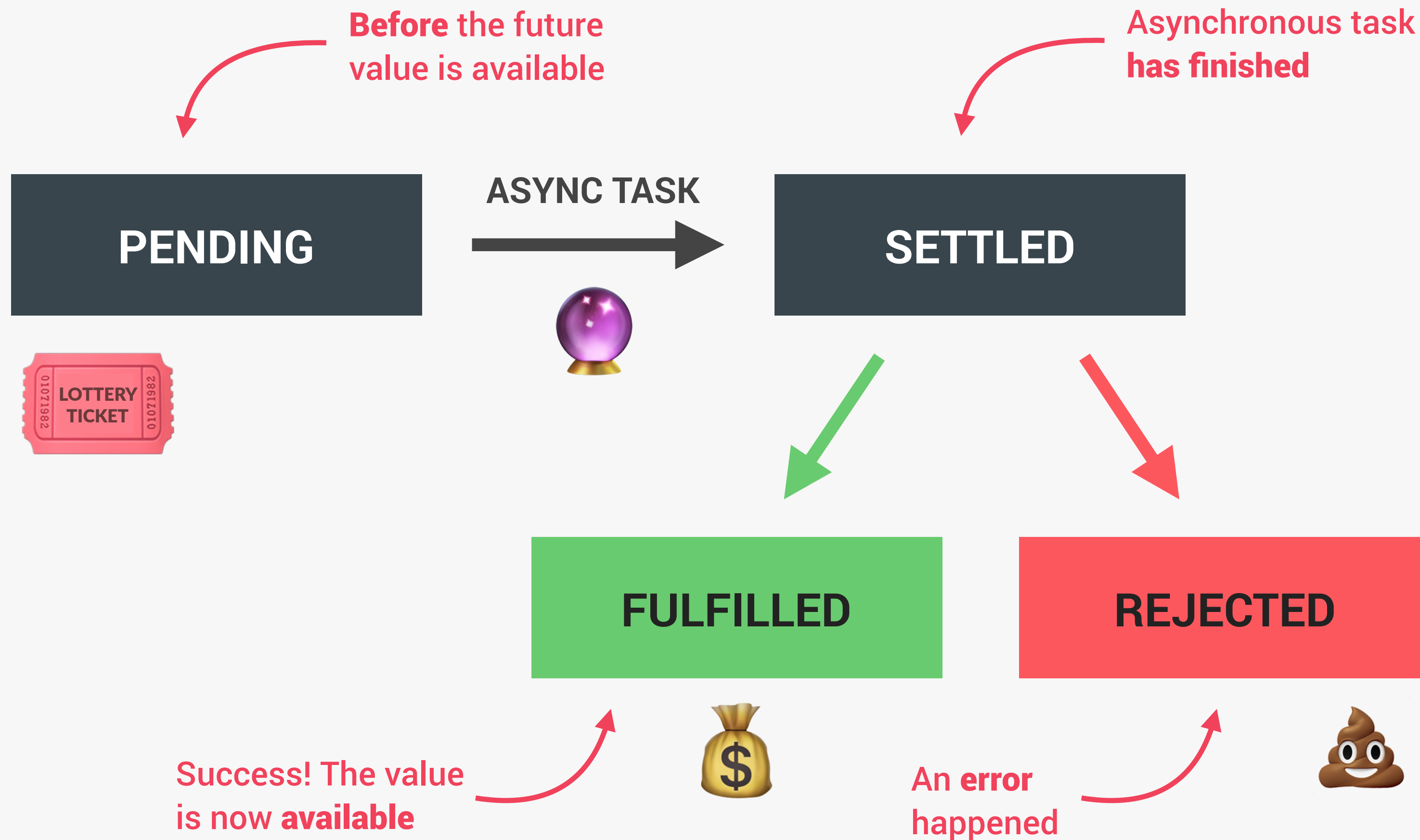
🎰 Lottery draw happens asynchronously



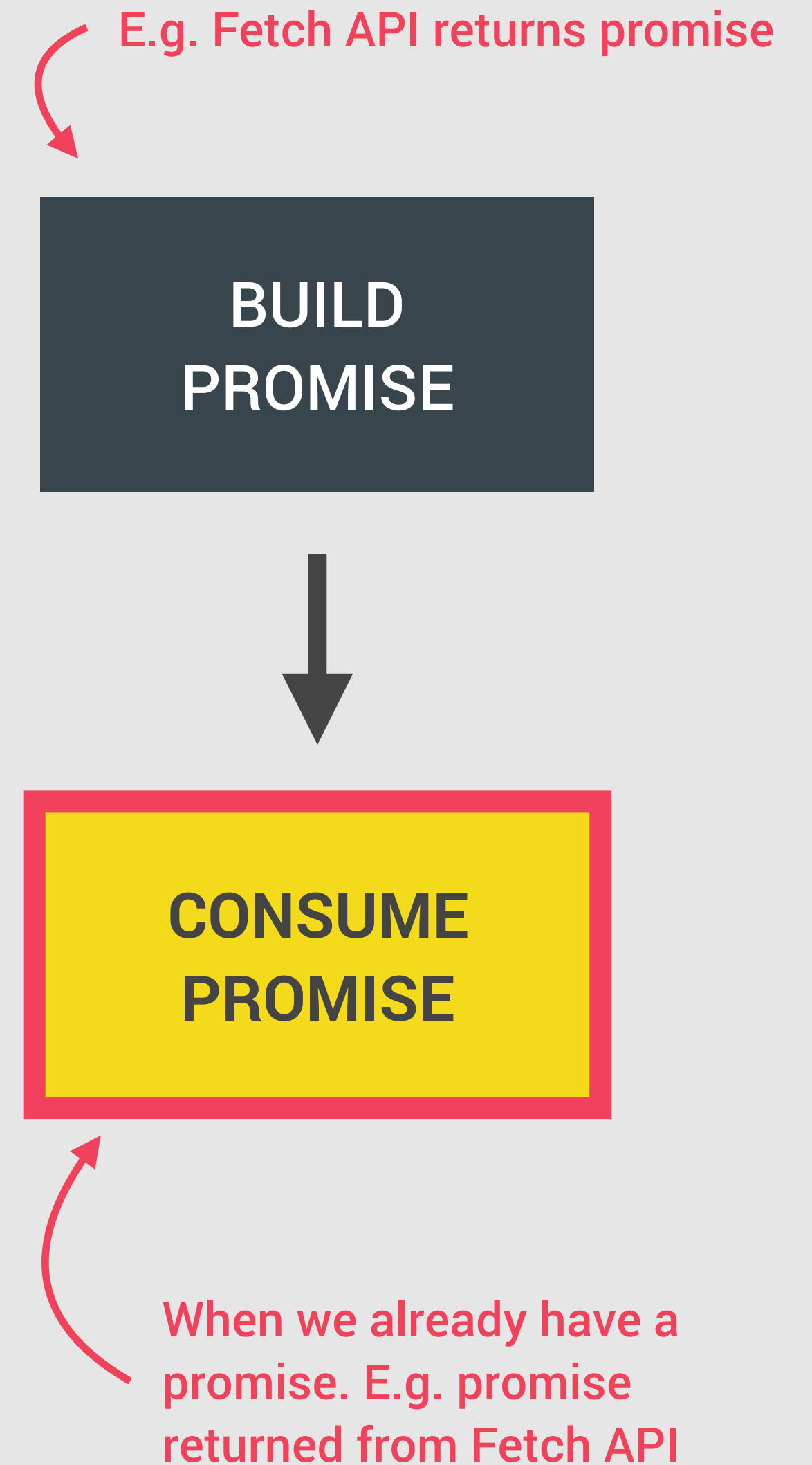
💰 If correct outcome, I receive money, because it was promised



# THE PROMISE LIFECYCLE



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







JONAS.IO  
SCHMEDTMANN

# THE COMPLETE JAVASCRIPT COURSE

FROM ZERO TO EXPERT!



@JONASSCHMEDTMAN

## SECTION

ASYNCHRONOUS JAVASCRIPT:  
PROMISES, ASYNC/AWAIT AND AJAX

## LECTURE

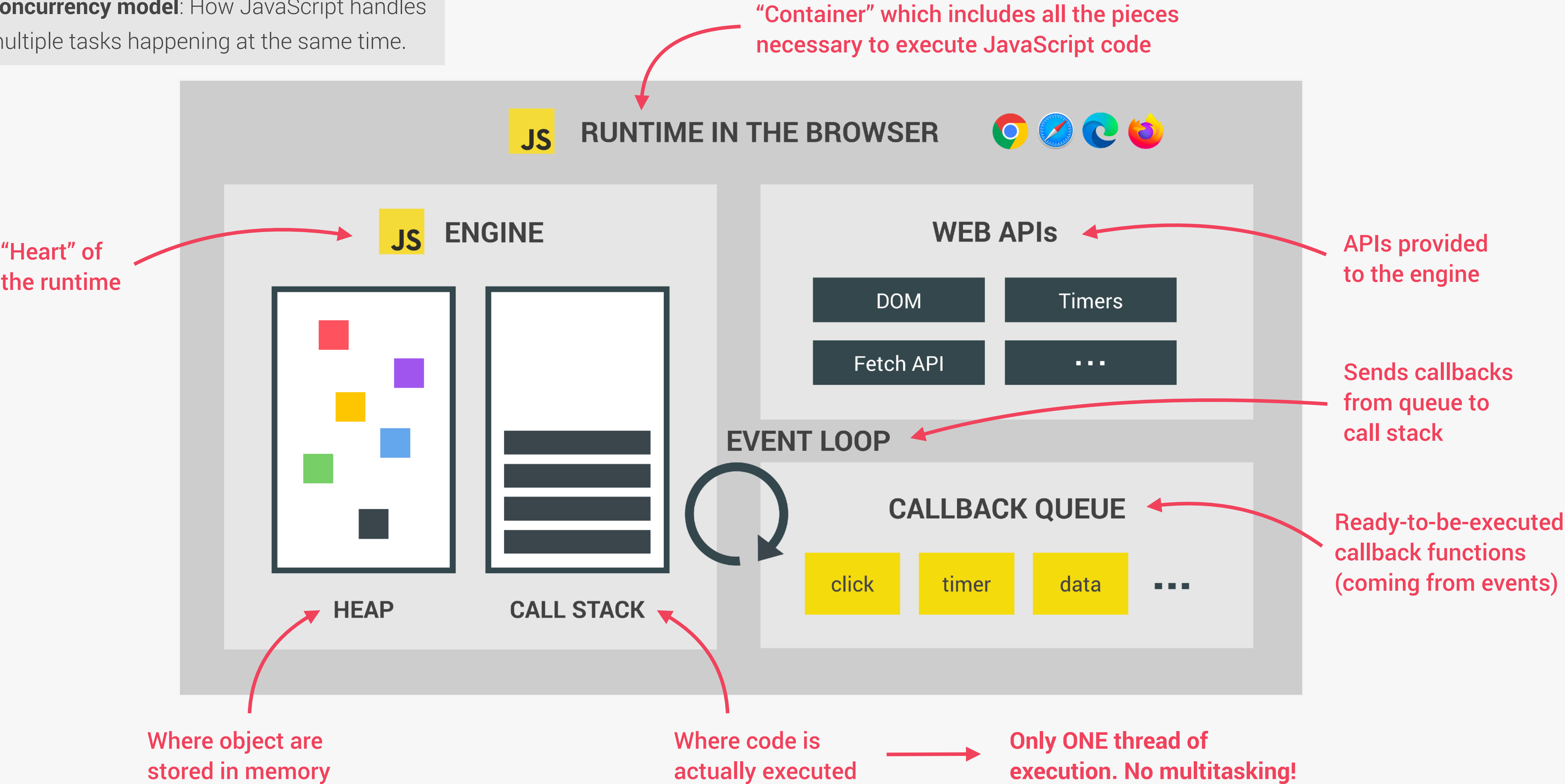
ASYNCHRONOUS BEHIND THE SCENES:  
THE EVENT LOOP





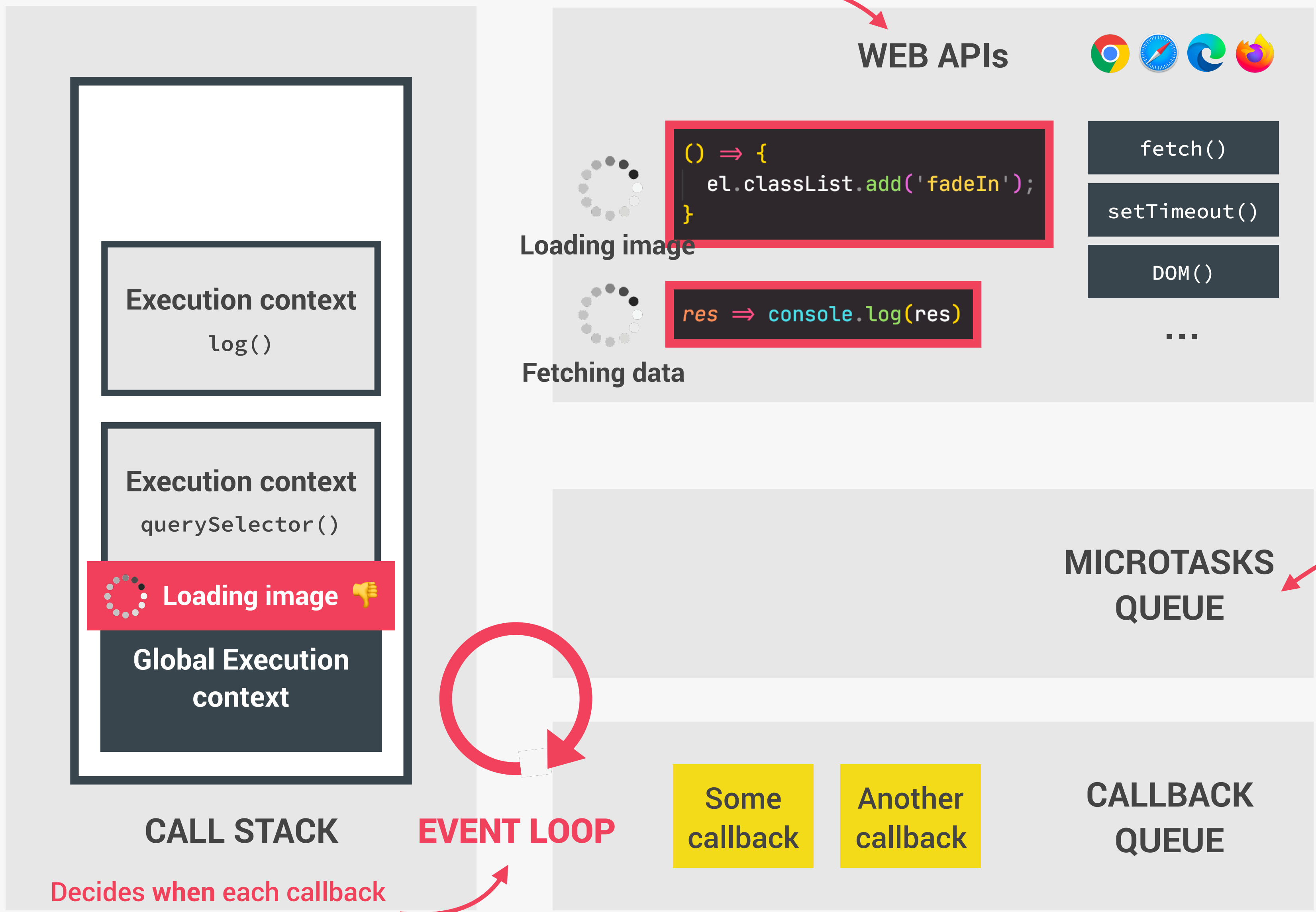
# REVIEW: JAVASCRIPT RUNTIME

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



# HOW ASYNCHRONOUS JAVASCRIPT WORKS BEHIND THE SCENES

Where asynchronous tasks run



Decides when each callback is executed: **orchestration**

**WEB APIs**

Loading image

```
() => {  
  el.classList.add('fadeIn');  
}
```

Fetching data

```
res => console.log(res)
```

fetch()  
setTimeout()  
DOM()  
...

```
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 ...
```

Like callback queue, but for callbacks related to **promises**.  
Has **priority** over callback queue!

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





# MODERN JAVASCRIPT DEVELOPMENT: MODULES AND TOOLING



JONAS.IO  
SCHMEDTMANN

# THE COMPLETE JAVASCRIPT COURSE

FROM ZERO TO EXPERT!



@JONASSCHMEDTMAN

## SECTION

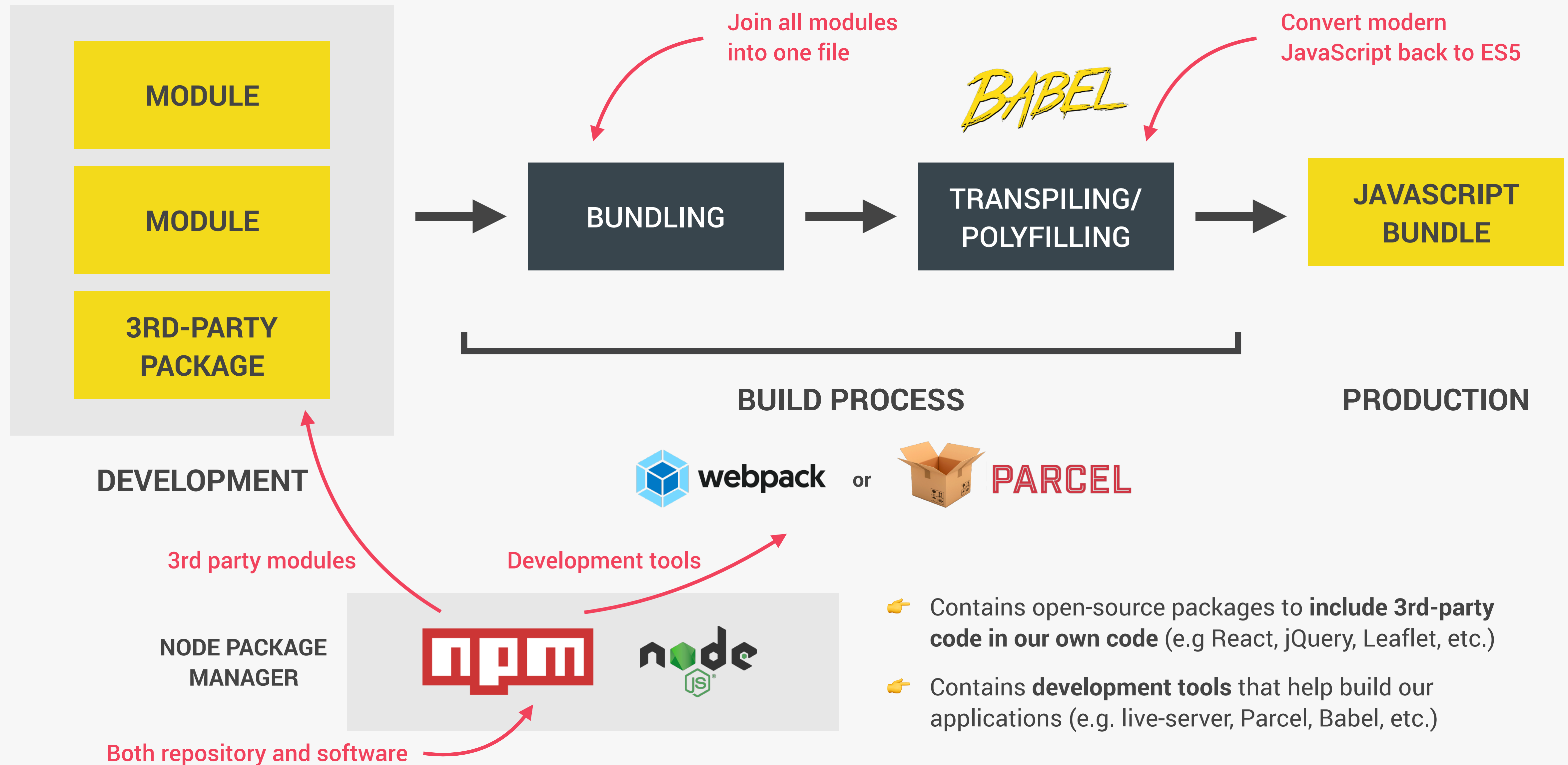
MODERN JAVASCRIPT DEVELOPMENT:  
MODULES AND TOOLING

## LECTURE

AN OVERVIEW OF MODERN  
JAVASCRIPT DEVELOPMENT



# MODERN JAVASCRIPT DEVELOPMENT









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



# AN OVERVIEW OF MODULES

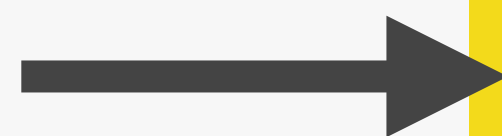
## MODULE

- ➔ Reusable piece of code that **encapsulates** implementation details;
- ➔ Usually a **standalone file**, but it doesn't have to be.

### WHY MODULES?

- ➔ **Compose software:** Modules are small building blocks that we put together to build complex applications;
- ➔ **Isolate components:** Modules can be developed in isolation without thinking about the entire codebase;
- ➔ **Abstract code:** Implement low-level code in modules and import these abstractions into other modules;
- ➔ **Organized code:** Modules naturally lead to a more organized codebase;
- ➔ **Reuse code:** Modules allow us to easily reuse the same code, even across multiple projects.

IMPORT  
(DEPENDENCY)



## MODULE

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

Module code



EXPORT  
(PUBLIC API)



# NATIVE JAVASCRIPT (ES6) MODULES

## ES6 MODULES

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

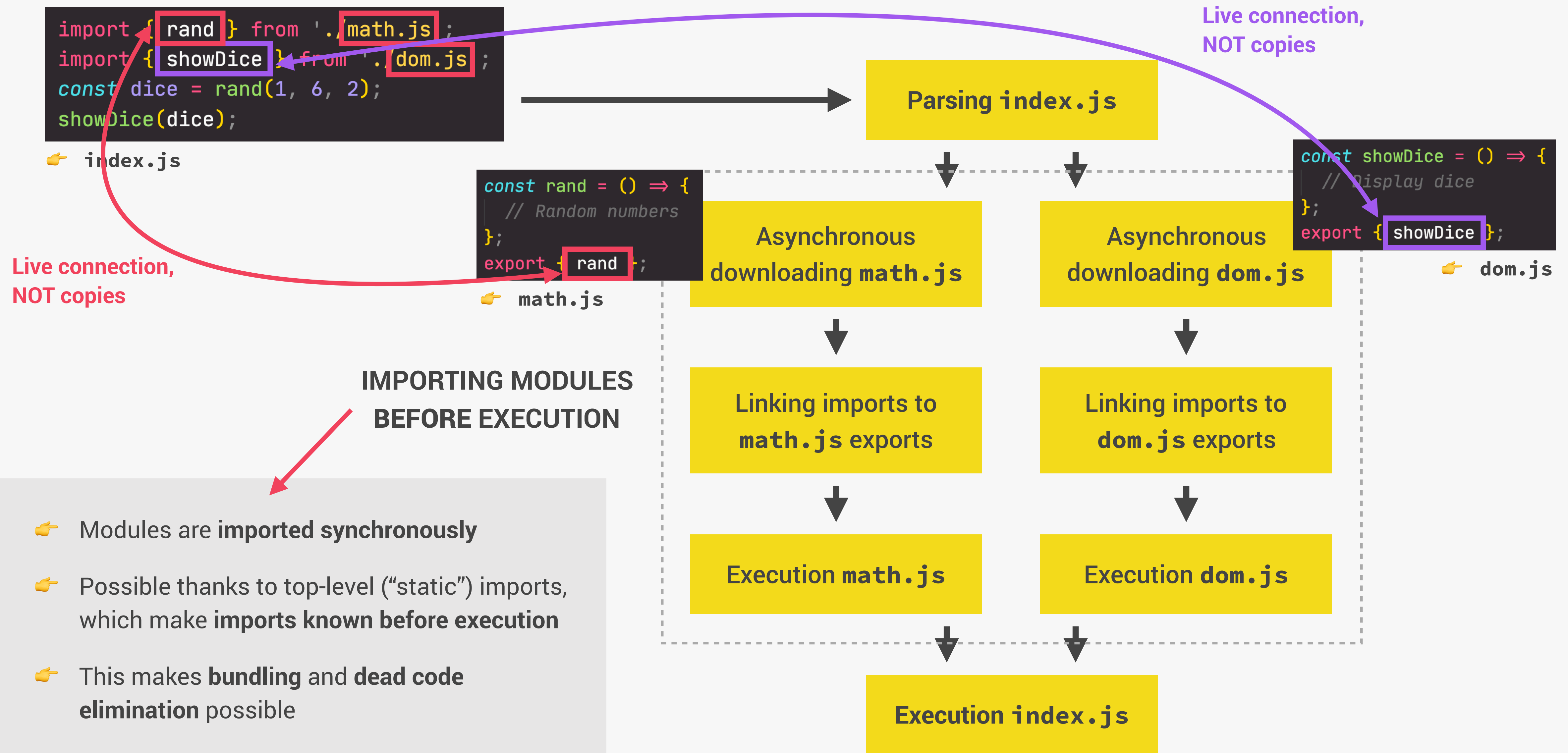
```
import { rand } from './math.js';
const diceP1 = rand(1, 6, 2);
const diceP2 = rand(1, 6, 2);
const scores = { diceP1, diceP2 };
export { scores };
```

import and export syntax

Need to happen at top-level  
Imports are hoisted!

|                       | 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   |

# HOW ES6 MODULES ARE IMPORTED









JONAS.IO  
SCHMEDTMANN

# THE COMPLETE JAVASCRIPT COURSE

FROM ZERO TO EXPERT!



@JONASSCHMEDTMAN

## SECTION

MODERN JAVASCRIPT DEVELOPMENT:  
MODULES AND TOOLING

## LECTURE

MODERN, CLEAN AND DECLARATIVE  
JAVASCRIPT PROGRAMMING

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



# 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);
```

# USE DECLARATIVE AND FUNCTIONAL CODE

## FUNCTIONAL PROGRAMMING

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

More about this in Section **Functional Programming With JavaScript** 👉

## 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





# FORKIFY APP. BUILDING A MODERN APPLICATION



JONAS.IO  
SCHMEDTMANN

# THE COMPLETE JAVASCRIPT COURSE

FROM ZERO TO EXPERT!



@JONASSCHMEDTMAN

SECTION

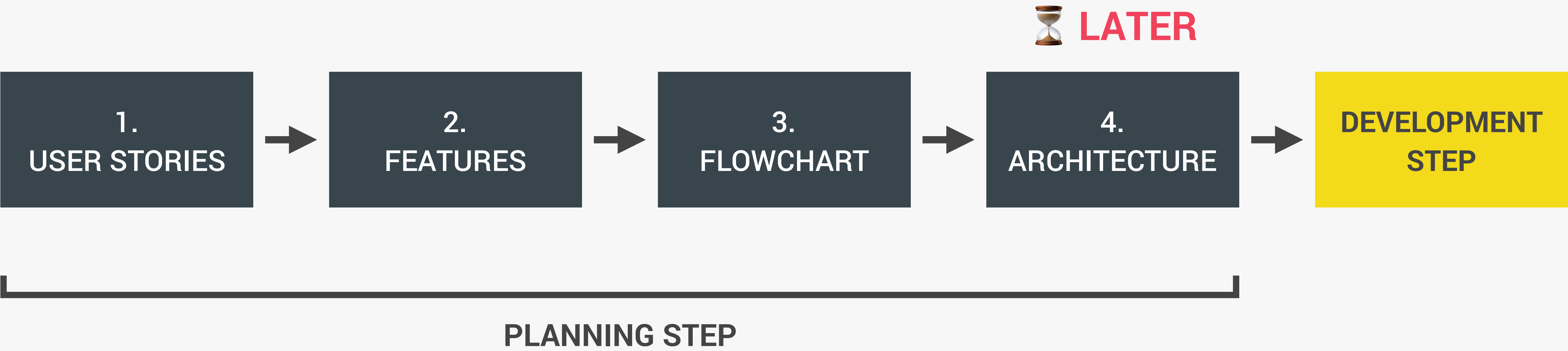
FORKIFY APP: BUILDING A MODERN  
APPLICATION

LECTURE

PROJECT OVERVIEW AND PLANNING

JS

# PROJECT PLANNING





👉 **User story:** Description of the application's functionality from the user's perspective.

👉 **Common format:** As a *[type of user]*, I want *[an action]* so that *[a benefit]*

- 1 As a user, I want to **search for recipes**, so that I can find new ideas for meals
- 2 As a user, I want to be able to **update the number of servings**, so that I can cook a meal for different number of people
- 3 As a user, I want to **bookmark recipes**, so that I can review them later
- 4 As a user, I want to be able to **create my own recipes**, so that I have them all organized in the same app
- 5 As a user, I want to be able to **see my bookmarks and own recipes when I leave the app and come back later**, so that I can close the app safely after cooking

## 2. FEATURES



### USER STORIES

### FEATURES

- 1 Search for recipes
- 2 Update the number of servings
- 3 Bookmark recipes
- 4 Create my own recipes
- 5 See my bookmarks and own recipes when I leave the app and come back later

- 👉 Search functionality: input field to send request to API with searched keywords
- 👉 Display results with pagination
- 👉 Display recipe with cooking time, servings and ingredients

- 👉 Change servings functionality: update all ingredients according to current number of servings

- 👉 Bookmarking functionality: display list of all bookmarked recipes

- 👉 User can upload own recipes
- 👉 User recipes will automatically be bookmarked
- 👉 User can only see their own recipes, not recipes from other users

- 👉 Store bookmark data in the browser using local storage
- 👉 On page load, read saved bookmarks from local storage and display

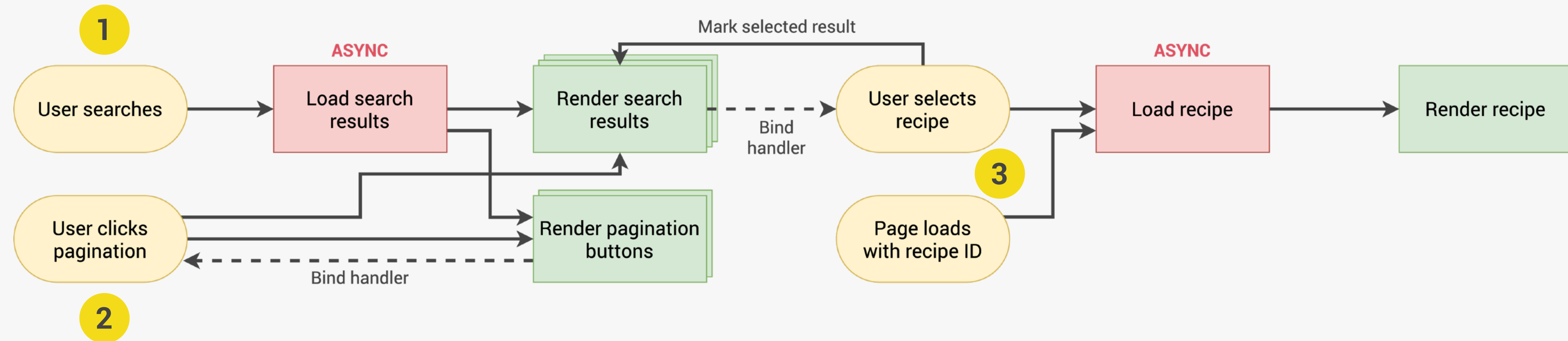
# 3. FLOWCHART (PART 1)



## FEATURES

1. Search functionality: API search request
2. Results with pagination
3. Display recipe

Other features later









JONAS.IO  
SCHMEDTMANN

# THE COMPLETE JAVASCRIPT COURSE

FROM ZERO TO EXPERT!



@JONASSCHMEDTMAN

SECTION

FORKIFY APP: BUILDING A MODERN  
APPLICATION

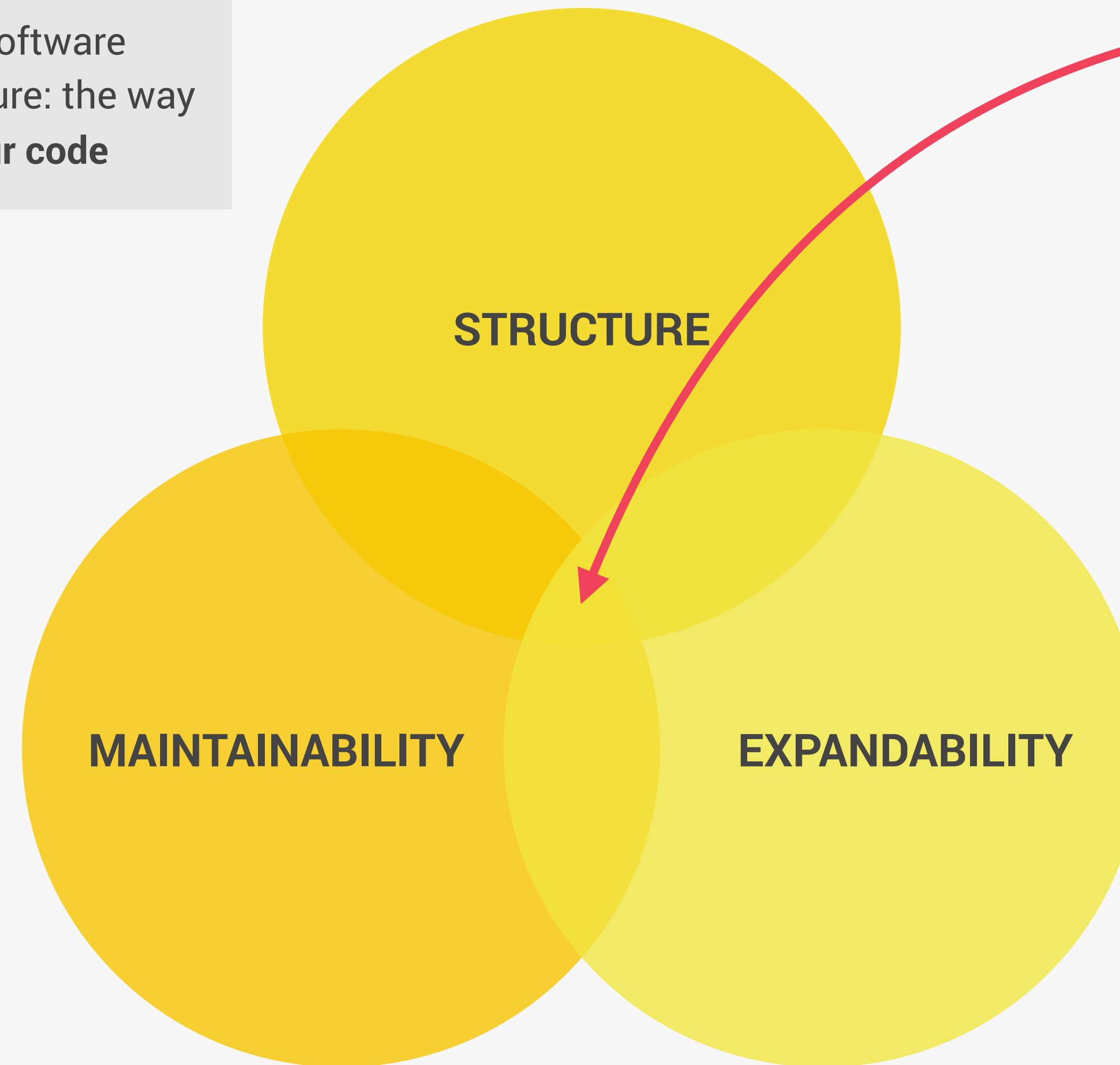
LECTURE

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.

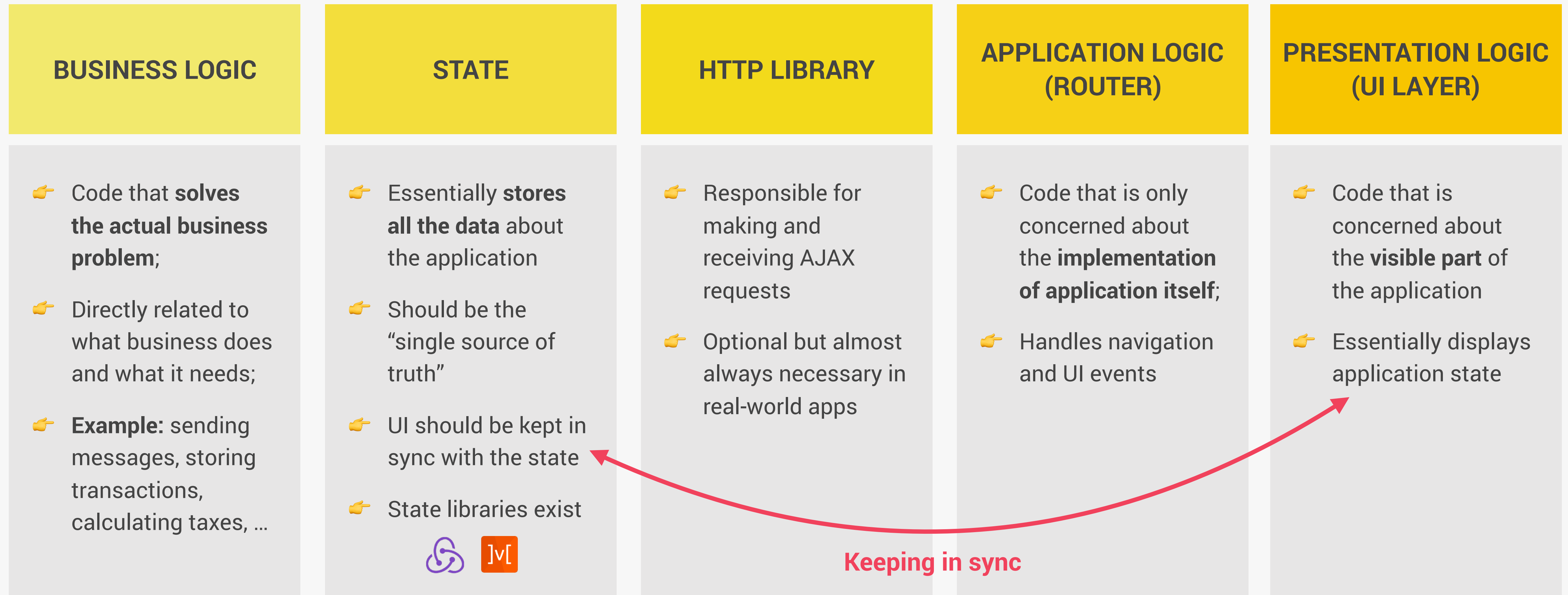


👉 A project is never done!  
We need to be able to easily **change it in the future**

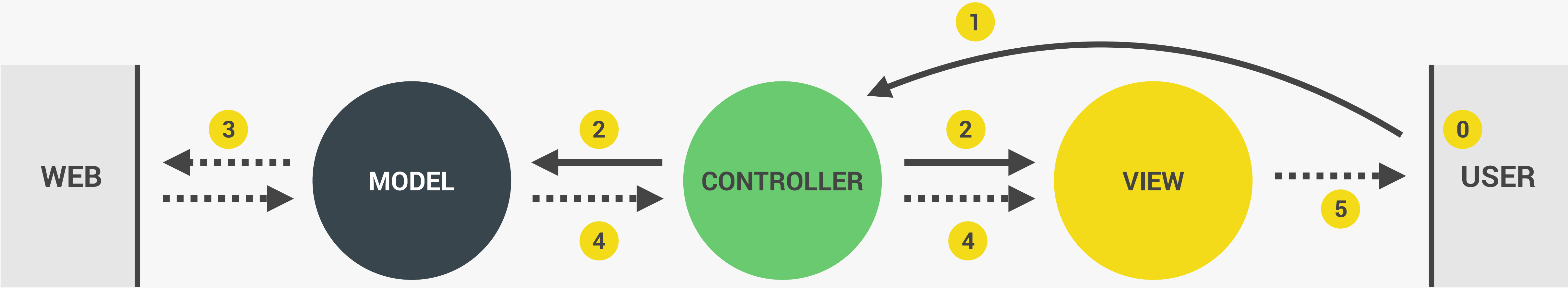
👉 We also need to be able to easily **add new features**



# COMPONENTS OF ANY ARCHITECTURE



# THE MODEL-VIEW-CONTROLLER (MVC) ARCHITECTURE



BUSINESS LOGIC

STATE

HTTP LIBRARY

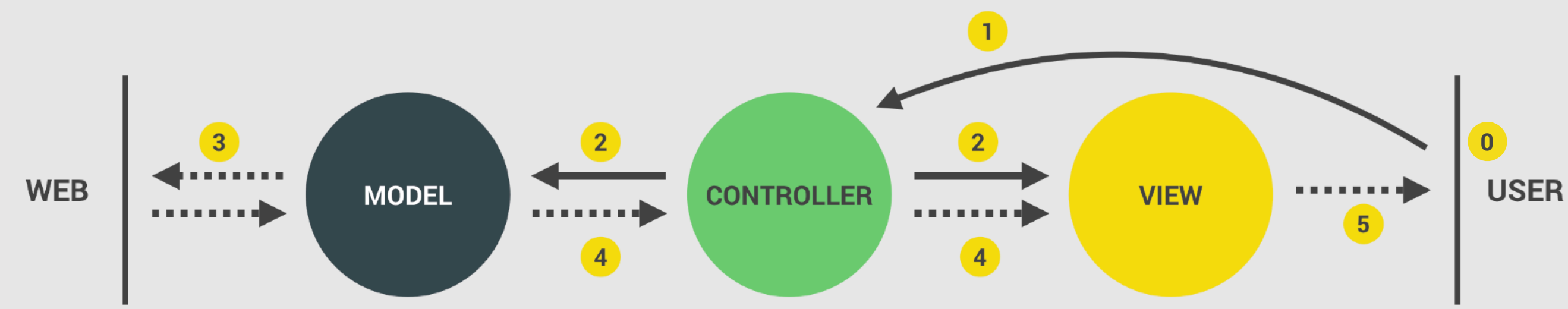
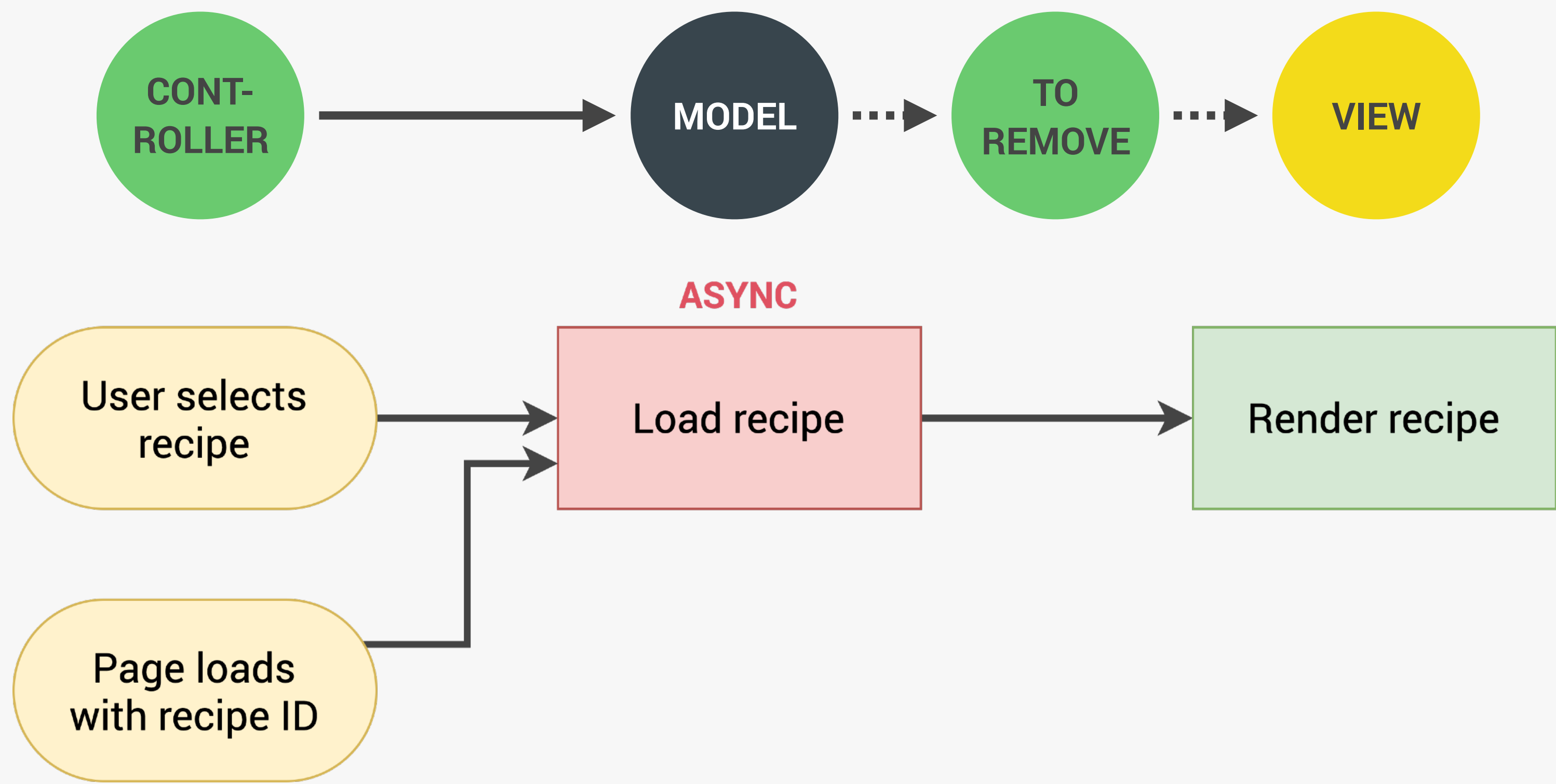
APPLICATION LOGIC

👉 Bridge between model and views (which don't know about one another)

👉 Handles UI events and **dispatches tasks to model and view**

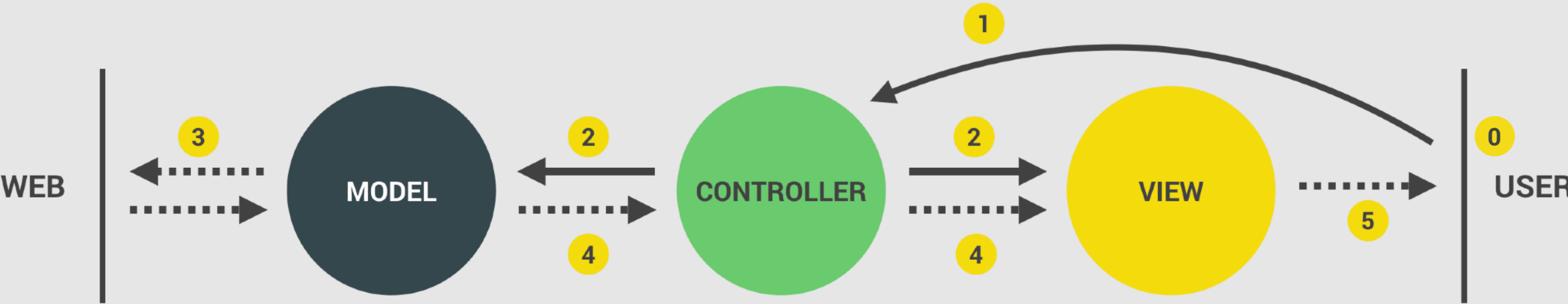
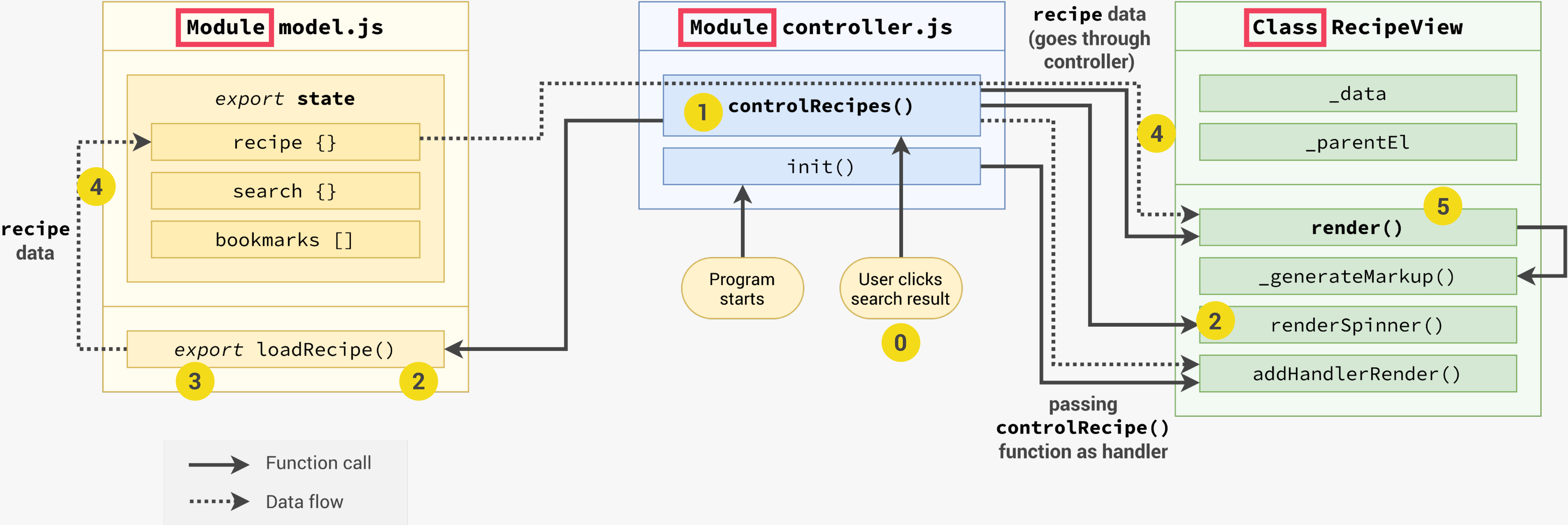
PRESENTATION LOGIC

→ Connected by function call and import  
.....→ Data flow





# MVC IMPLEMENTATION (RECIPE DISPLAY ONLY)







JONAS.IO  
SCHMEDTMANN

# THE COMPLETE JAVASCRIPT COURSE

FROM ZERO TO EXPERT!



@JONASSCHMEDTMAN

## SECTION

FORKIFY APP: BUILDING A MODERN  
APPLICATION

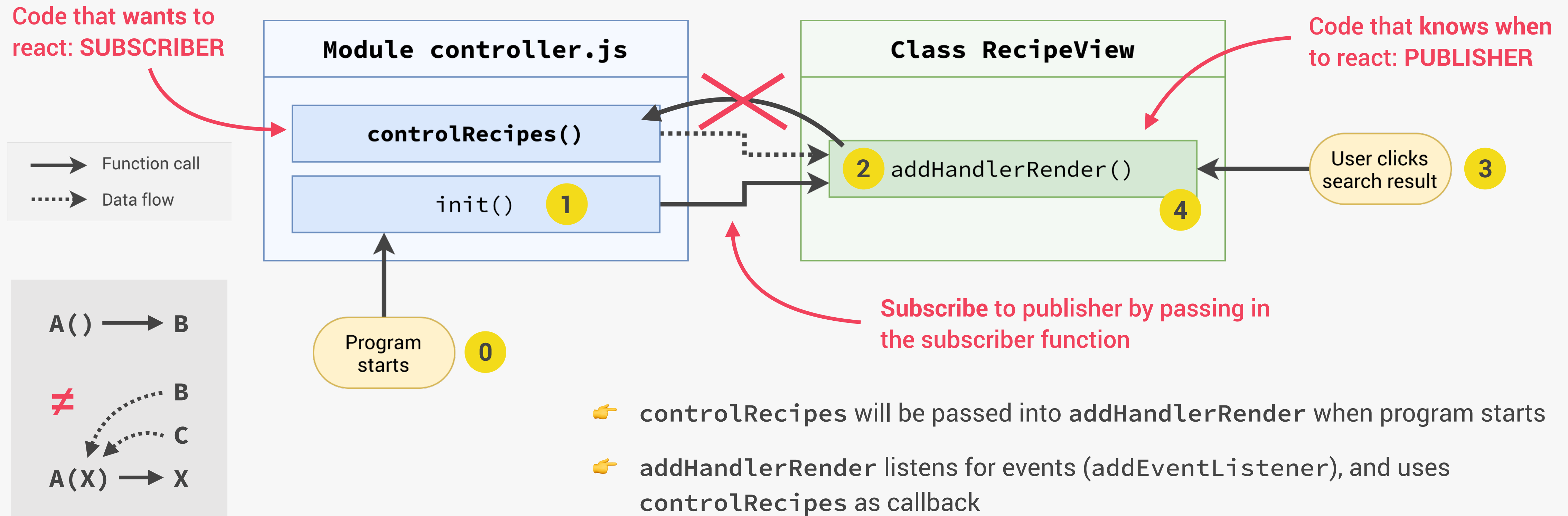
## LECTURE

EVENT HANDLERS IN MVC:  
PUBLISHER-SUBSCRIBER PATTERN

JS



# EVENT HANDLING IN MVC: PUBLISHER-SUBSCRIBER PATTERN



- 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

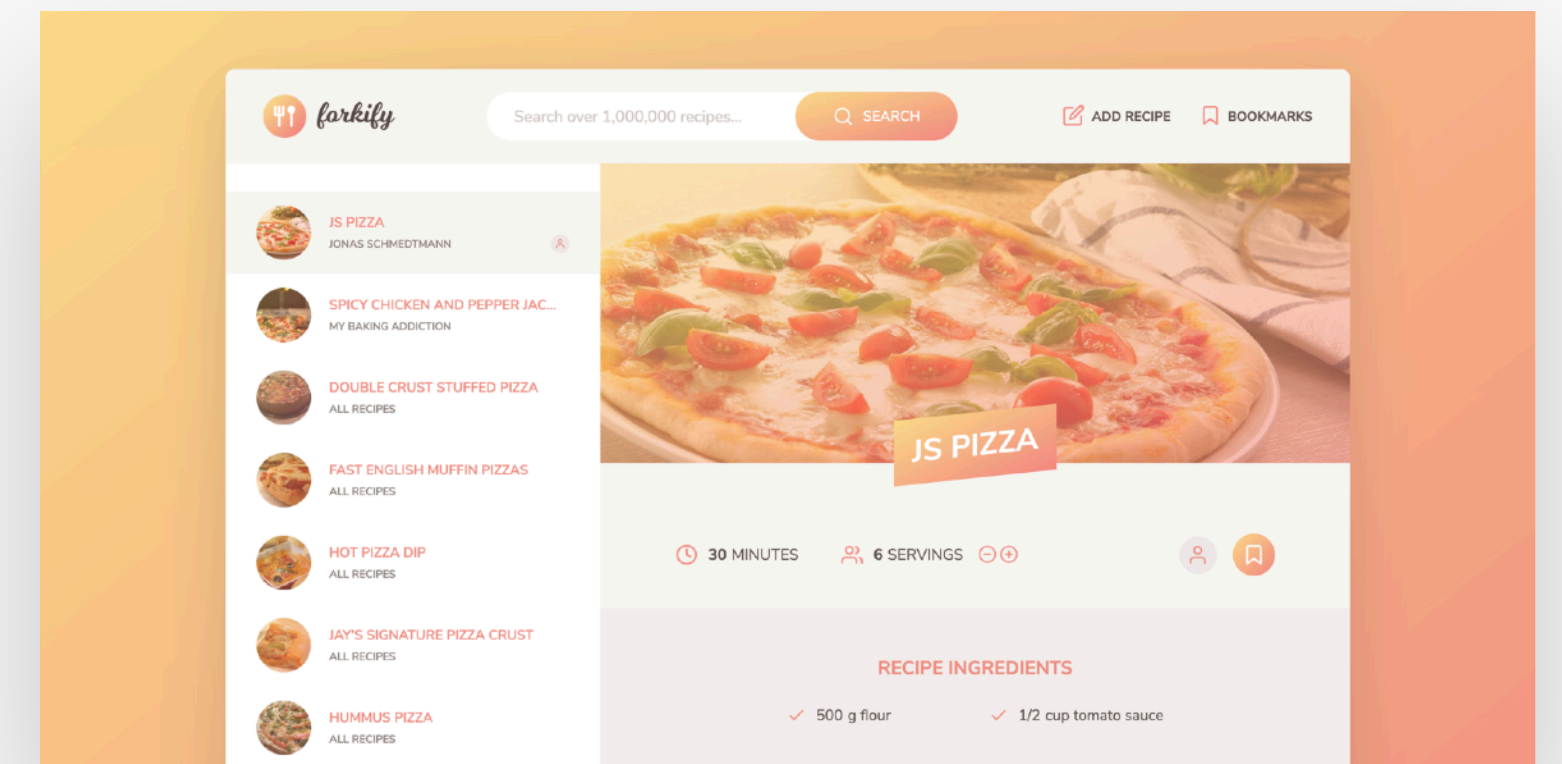
LECTURE

WRAPPING UP: FINAL  
CONSIDERATIONS

JS



- 👉 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.





END