THE COMPLETE JAVASCRIPT COURSE

FROM ZERO TO EXPERT!

slides for theory lectures

JS

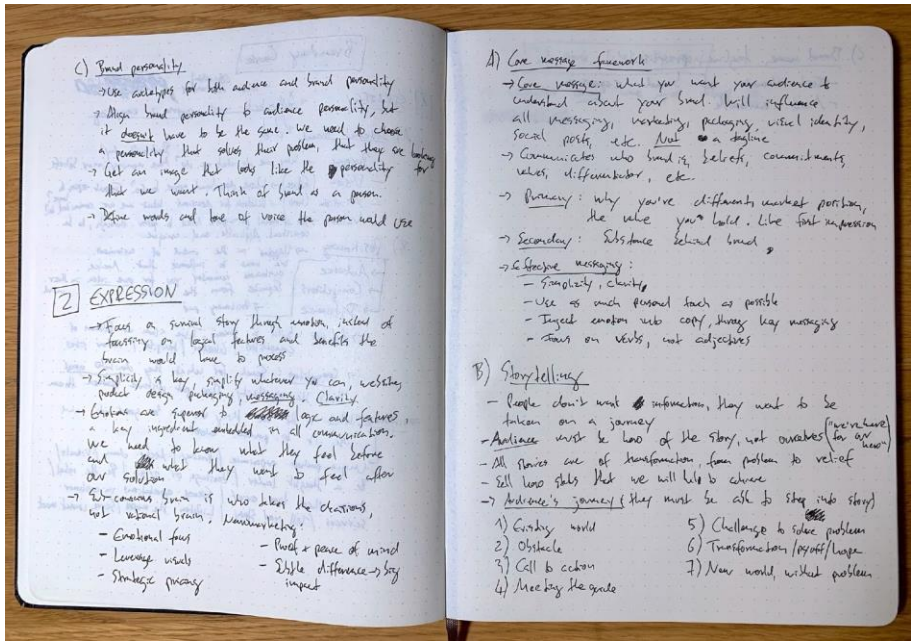
📖 TABLE OF CONTENTS: THEORY LECTURES

JS

WELCOME

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!



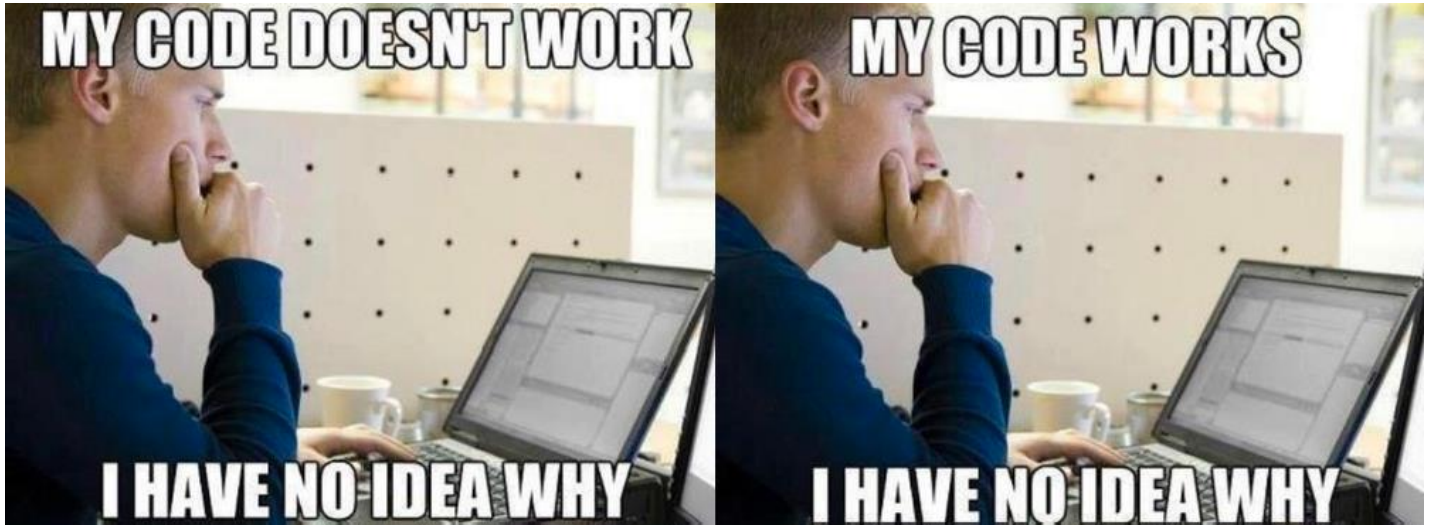
Totally non-coding... Try to understand a single word 😂

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"!***

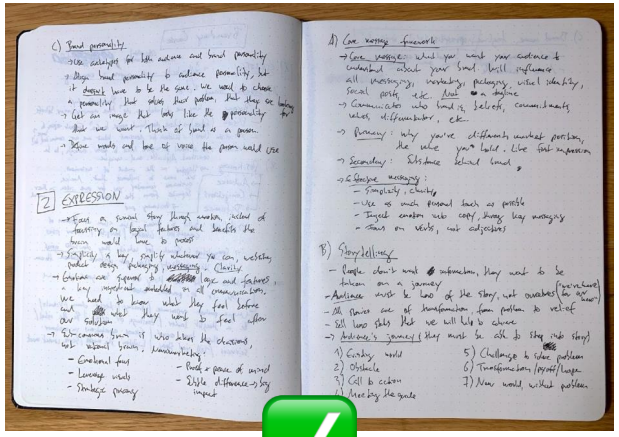
SOME QUICK CONSIDERATIONS BEFORE WE START... 🚀

😅 **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...🚀

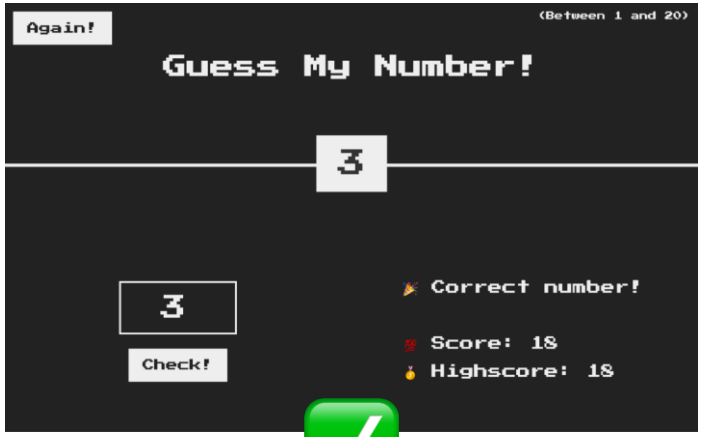
⏳ **Before moving on from a section, make sure that you understand exactly what was covered.** Take a break, review the code we wrote, review your notes, review the projects we built, and maybe even write some code yourself.



✅

```
// We listen to the event on the buton element, because this is where the
click is supposed to happen
// Checking if number is correct
document.querySelector('.check').addEventListener('click', () => {
  const guess = Number(document.querySelector('.guess').value);

  if (!guess) {
    document.querySelector('.message').textContent = '⛔ No number!';
  } else if (guess === number) {
    document.querySelector('.number').textContent = number;
    document.querySelector('.message').textContent = '🎉 Correct number!';

    // If new highscore, then display it
    if (score > highscore) {
      highscore = score;
      document.querySelector('.highscore').textContent = highscore;
    }
  } else if (guess > number) {
    document.querySelector('.message').textContent = '📈 Too high!';
    // Decrease score
    score = --score;
    document.querySelector('.score').textContent = score;
  } else {
    document.querySelector('.message').textContent = '📉 Too low!';
    score = --score;
    document.querySelector('.sco                    t = score;
```

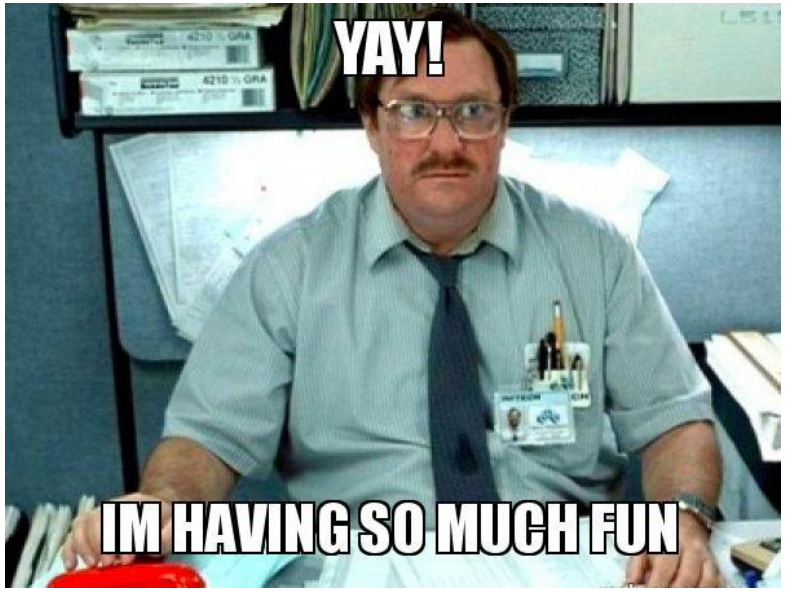
✅



✅

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 😅

JS

JAVASCRIPT FUNDAMENTALS – PART 1

THE COMPLETE JAVASCRIPT COURSE

FROM ZERO TO EXPERT!

SECTION

JAVASCRIPT FUNDAMENTALS – PART 1

LECTURE

A BRIEF INTRODUCTION TO JAVASCRIPT

WHAT IS JAVASCRIPT?

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

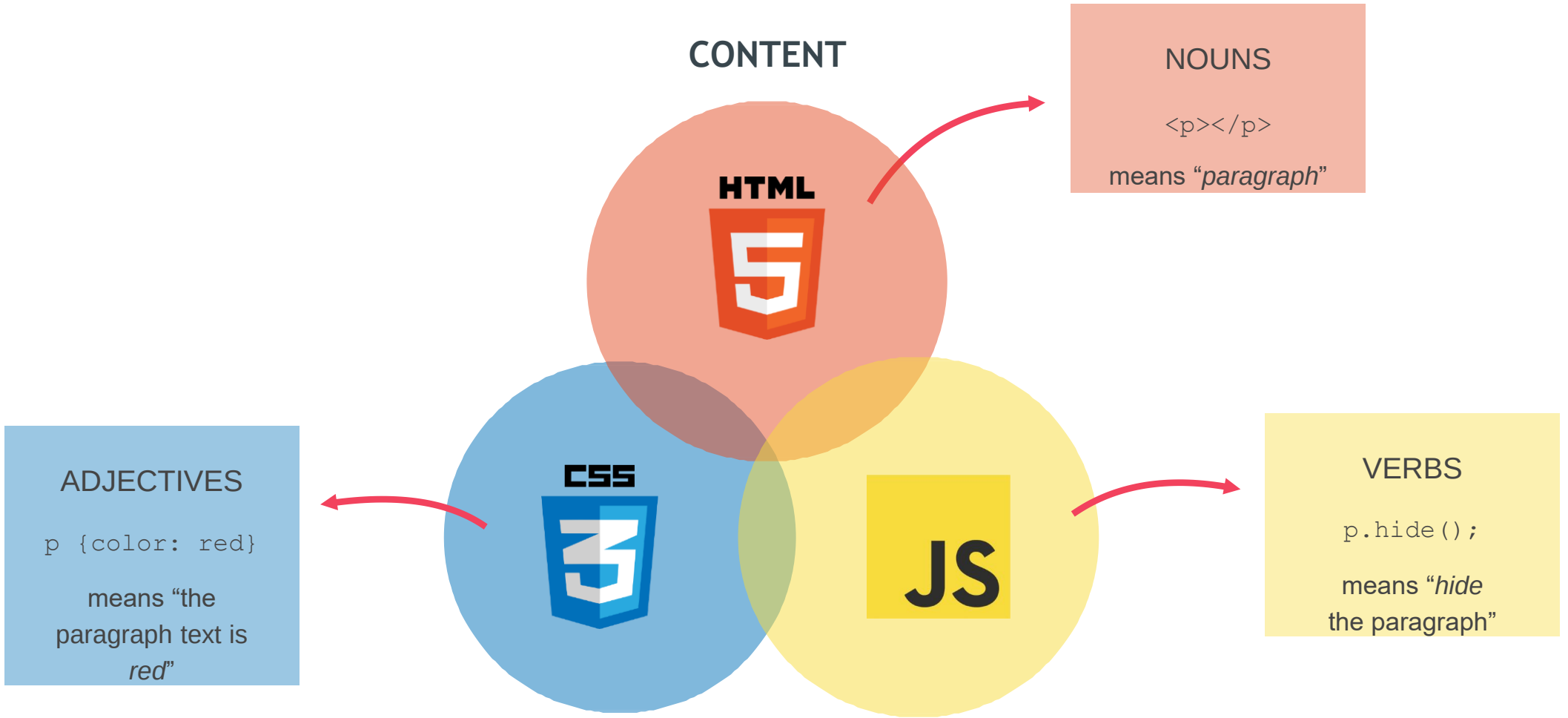
JAVASCRIPT IS A HIGH-LEVEL, OBJECT-ORIENTED, MULTI-PARADIGM PROGRAMMING 🤯 LANGUAGE.

Based on objects, for storing most kinds of data

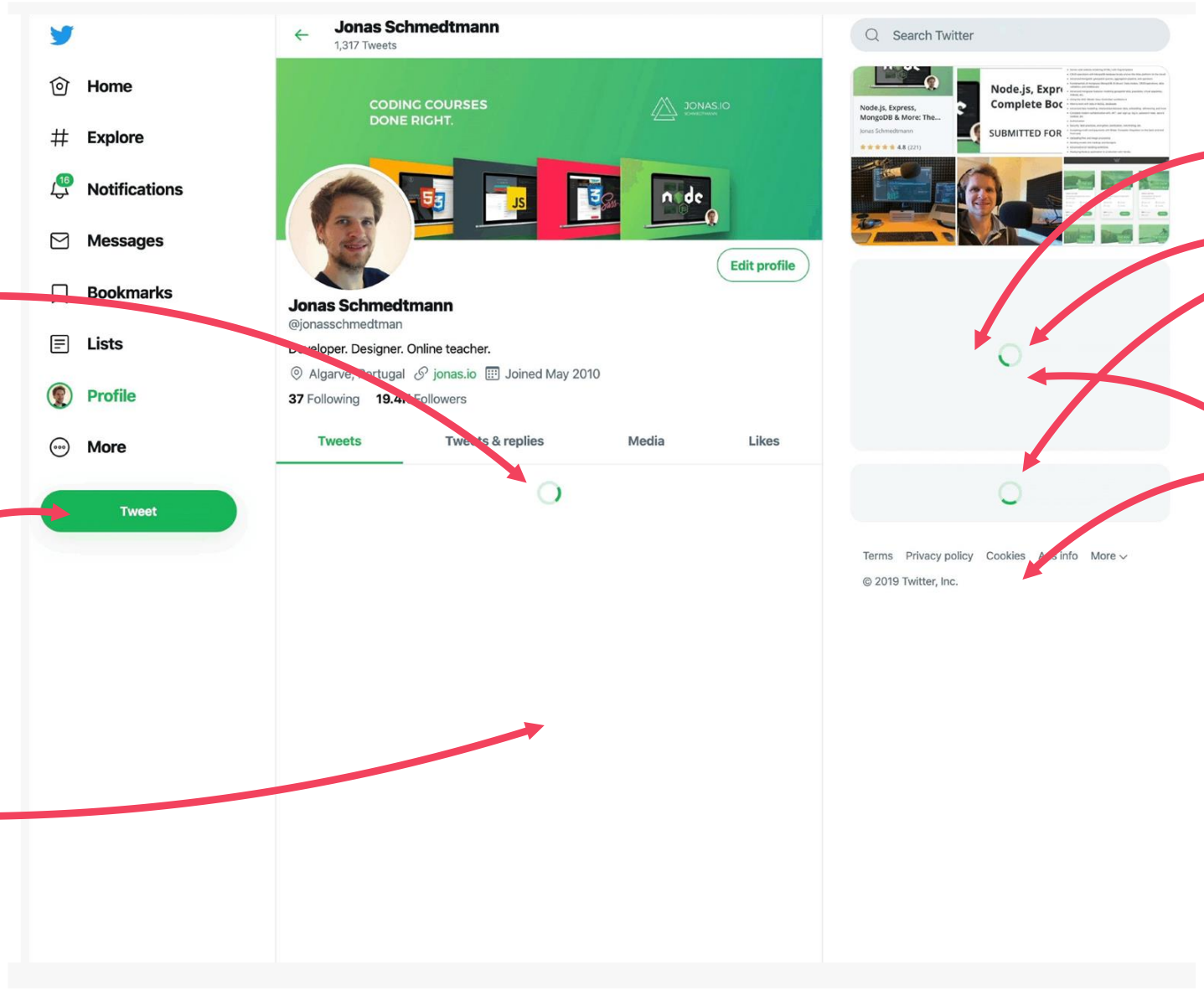
We can use different styles of programming

Instruct computer to *do* things

THE 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...)

Front-end apps

Dynamic effects and web applications in the browser 😍

THIS COURSE

100% based on JavaScript. They might go away, **but JavaScript won't**!

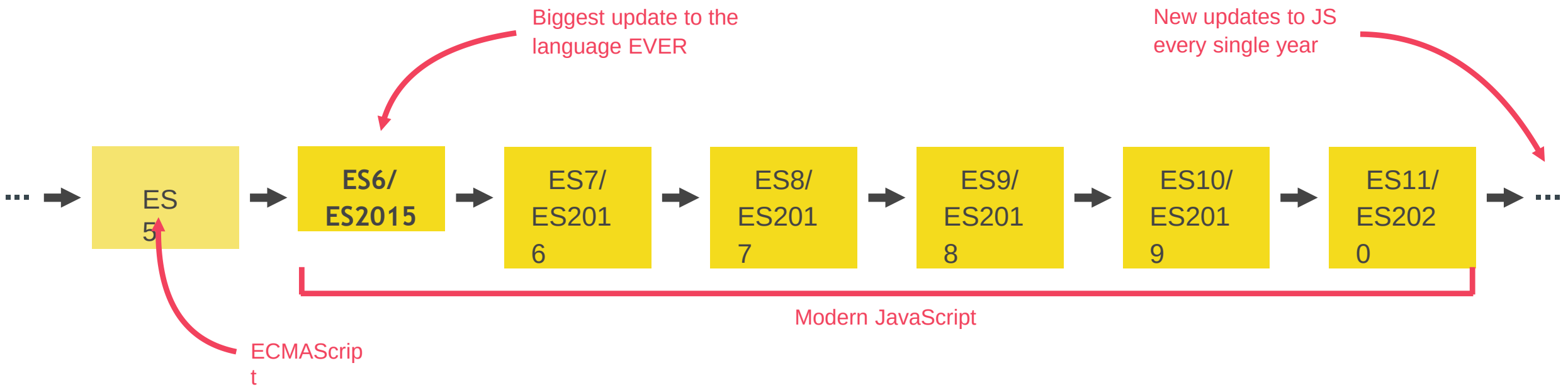
Back-end apps

Web applications on web servers

Native mobile applications

Native desktop applications

JAVASCRIPT RELEASES... (MORE ABOUT THIS LATER)



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

🥳 Let's finally get started!

THE COMPLETE
JAVASCRIPT COURSE

FROM ZERO TO EXPERT!

SECTION

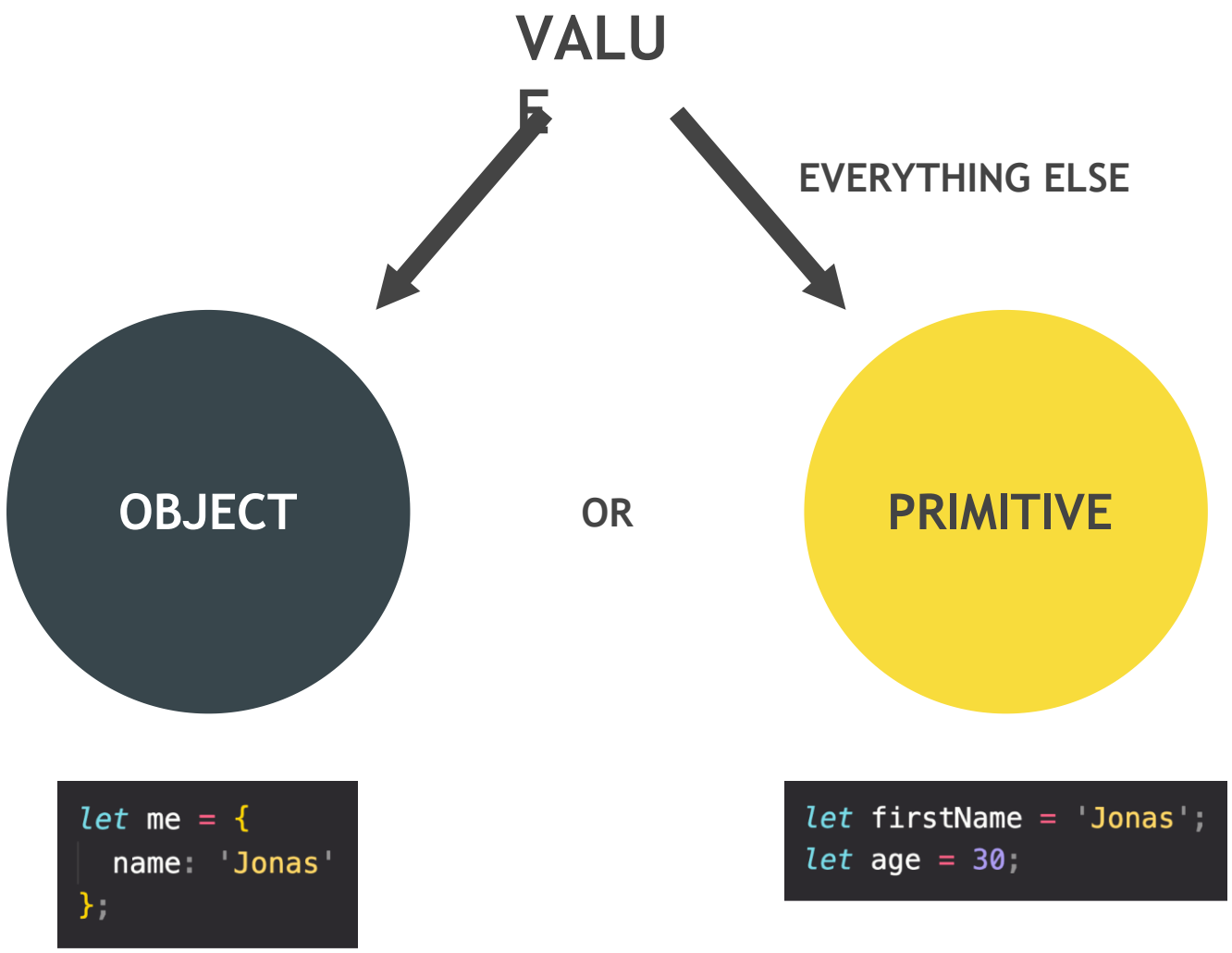
JAVASCRIPT
FUNDAMENTALS – PART 1

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;
```

3. Undefined: Value taken by a variable that is not yet defined (‘empty value’)

```
let children;
```

4. Null: Also means ‘empty value’

5. Symbol (ES2015): Value that is unique and cannot be changed *[Not useful for now]*

6. 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!

THE COMPLETE
JAVASCRIPT COURSE

FROM ZERO TO EXPERT!

SECTION

JAVASCRIPT
FUNDAMENTALS – PART 1

LECTURE

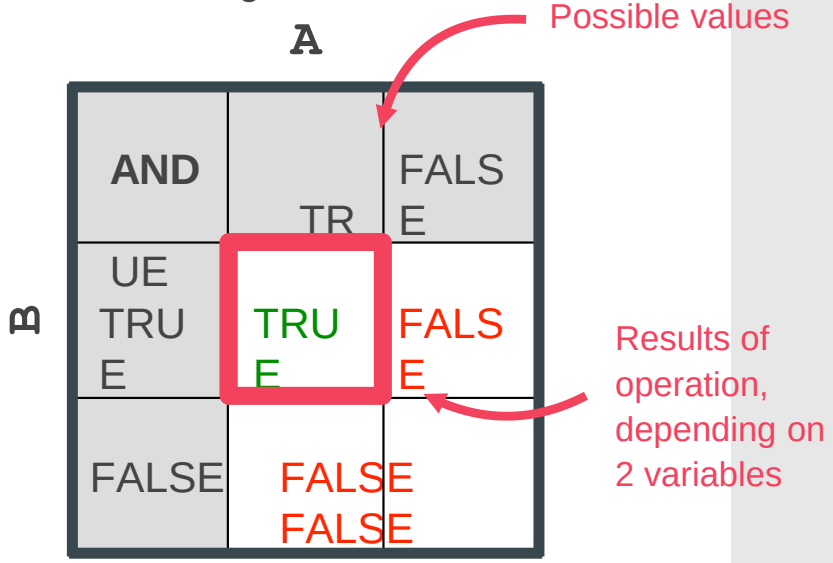
BOOLEAN LOGIC

JS

BASIC BOOLEAN LOGIC: THE AND, OR & NOT OPERATORS

A AND B

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



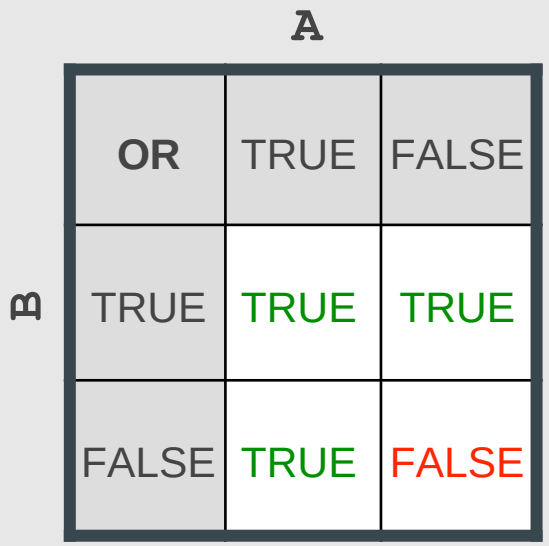
👇

`true` when **ALL** are `true`

No matter how many variables

A OR B

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



👇

`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

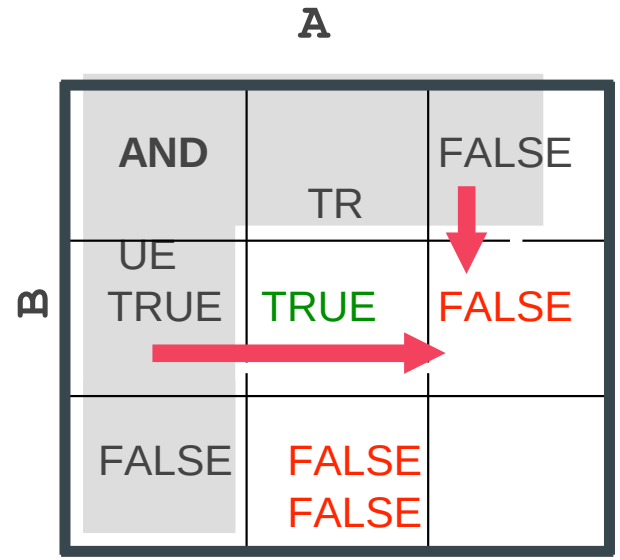
BOOLEAN VARIABLES

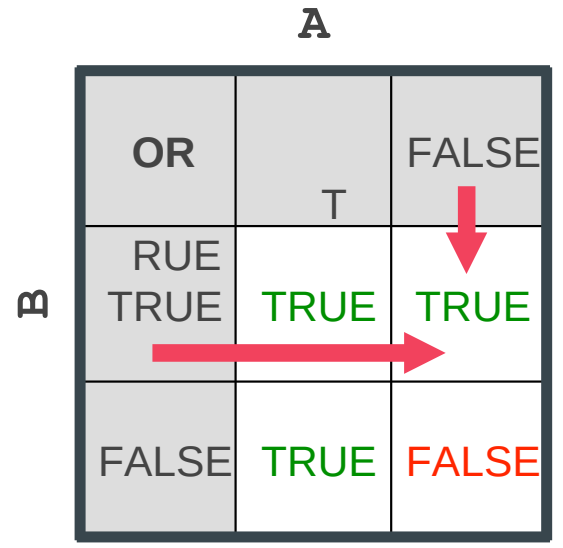
```
age = 16
```

- A: Age is greater or equal 20 → `false`
- B: Age is less than 30 → `true`

LET'S USE OPERATORS!

- `!A` → `true`
  - `false`
- `A AND B` → `false`
  - `false` `true`
- `A OR B` → `true`
  - `false` `true`
- `!A AND B` → `true`
  - `true` `true`
- `A OR B` → `false`
  - `false` `false`





THE COMPLETE
JAVASCRIPT COURSE

FROM ZERO TO EXPERT!

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 / … / ES208

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.

JS

JAVASCRIPT FUNDAMENTALS – PART 2

THE COMPLETE JAVASCRIPT COURSE

FROM ZERO TO EXPERT!

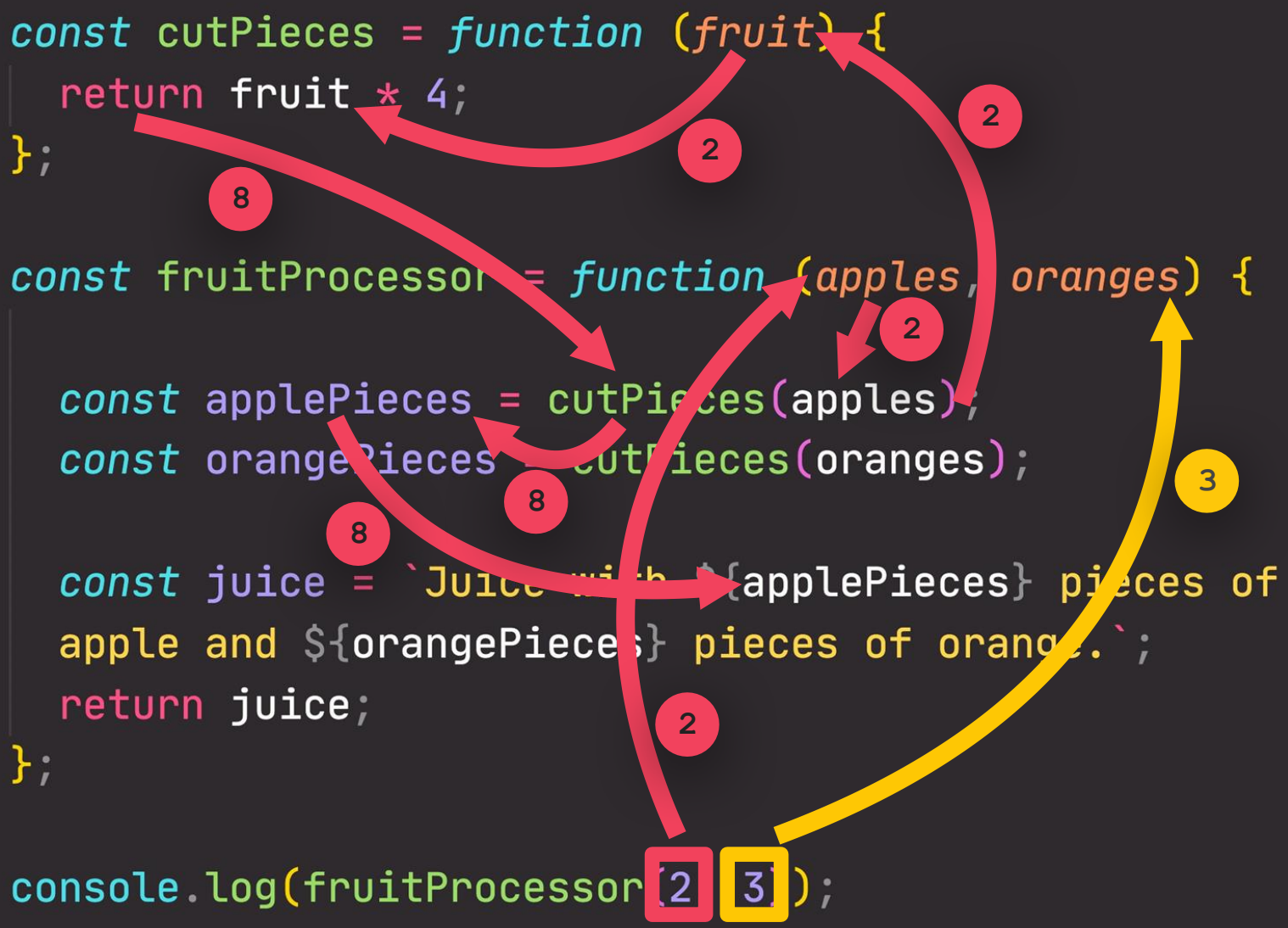
SECTION

JAVASCRIPT FUNDAMENTALS – PART 2

LECTURE

FUNCTIONS CALLING OTHER FUNCTIONS

CALLING A FUNCTION INSIDE A FUNCTION: DATA FLOW



THE COMPLETE
JAVASCRIPT COURSE

FROM ZERO TO EXPERT!

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 expression**

Essentially a function *value* stored in a variable

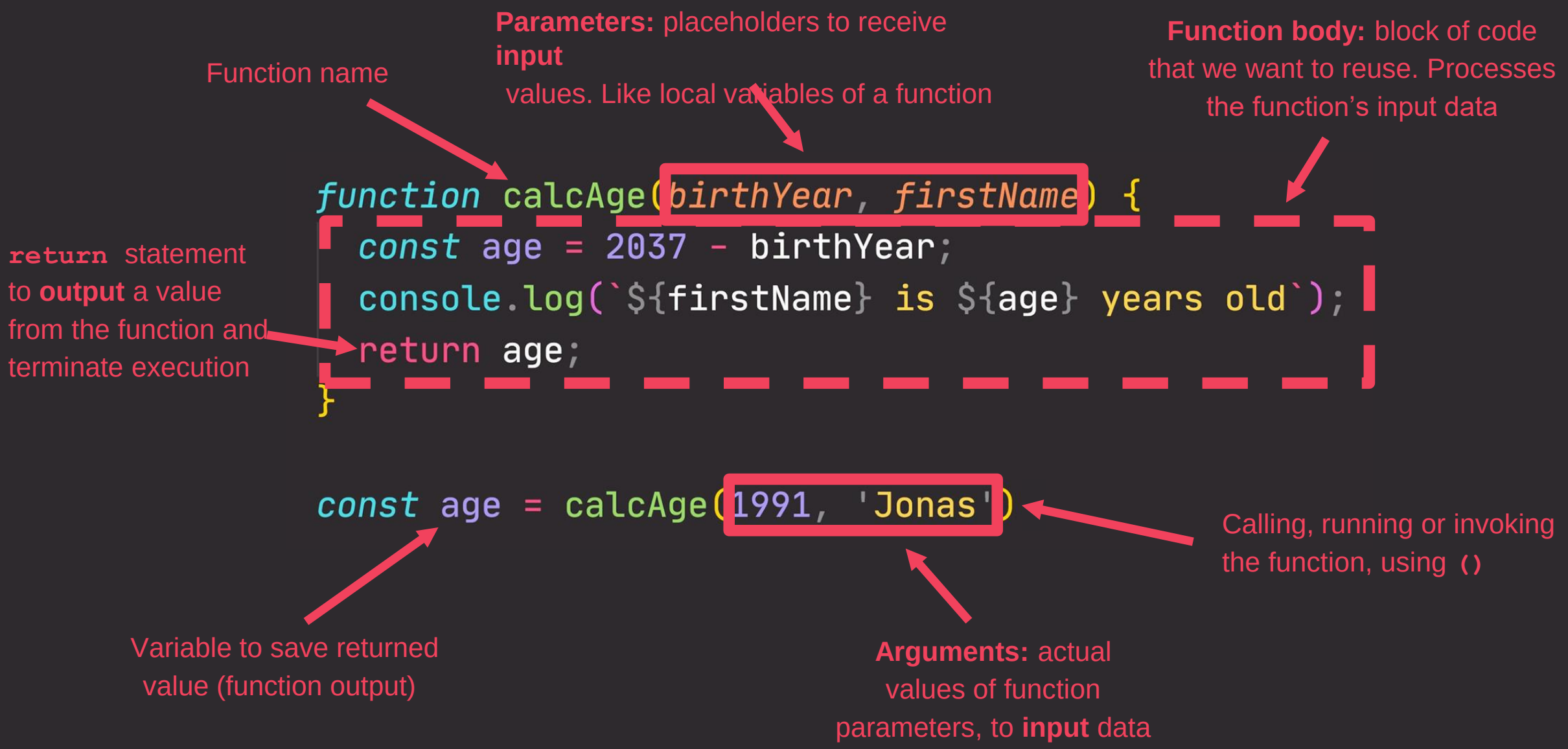
👉 **Arrow function**

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

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

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

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

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

FUNCTIONS REVIEW: ANATOMY OF A FUNCTION



JS

DEVELOPER SKILLS & EDITOR SETUP

THE COMPLETE JAVASCRIPT COURSE

FROM ZERO TO EXPERT!

SECTION

DEVELOPER SKILLS & EDITOR SETUP

LECTURE

LEARNING HOW TO CODE

JS

HOW TO FAIL 🤦 AT LEARNING HOW TO CODE

JS

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

JS

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

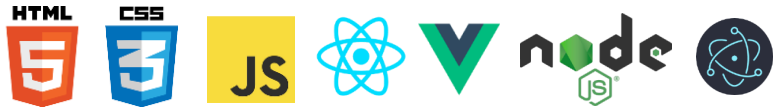
⬇ **FIX**

👍 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!

codewars

HOW TO SUCCEED AT LEARNING HOW TO CODE

JS

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

⬇ FIX

👍 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

⬇ FIX

👍 **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

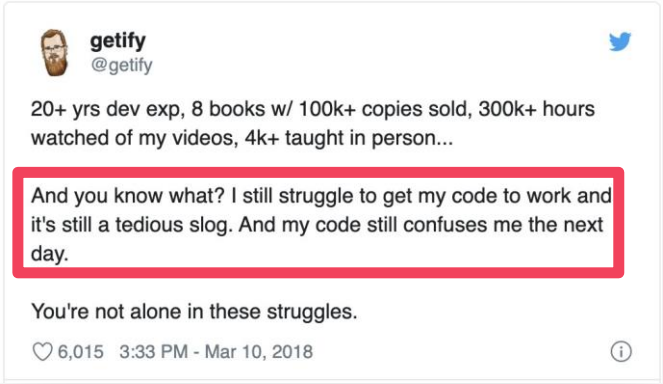
👍 You can always refactor code later

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

⬇ FIX

👍 Embrace the fact that **you will never you know everything**

👍 Just focus on what you need to achieve your goal!



HOW TO SUCCEED AT LEARNING HOW TO CODE

💥 He was **learning in isolation**

⬇ **FIX**

👍 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

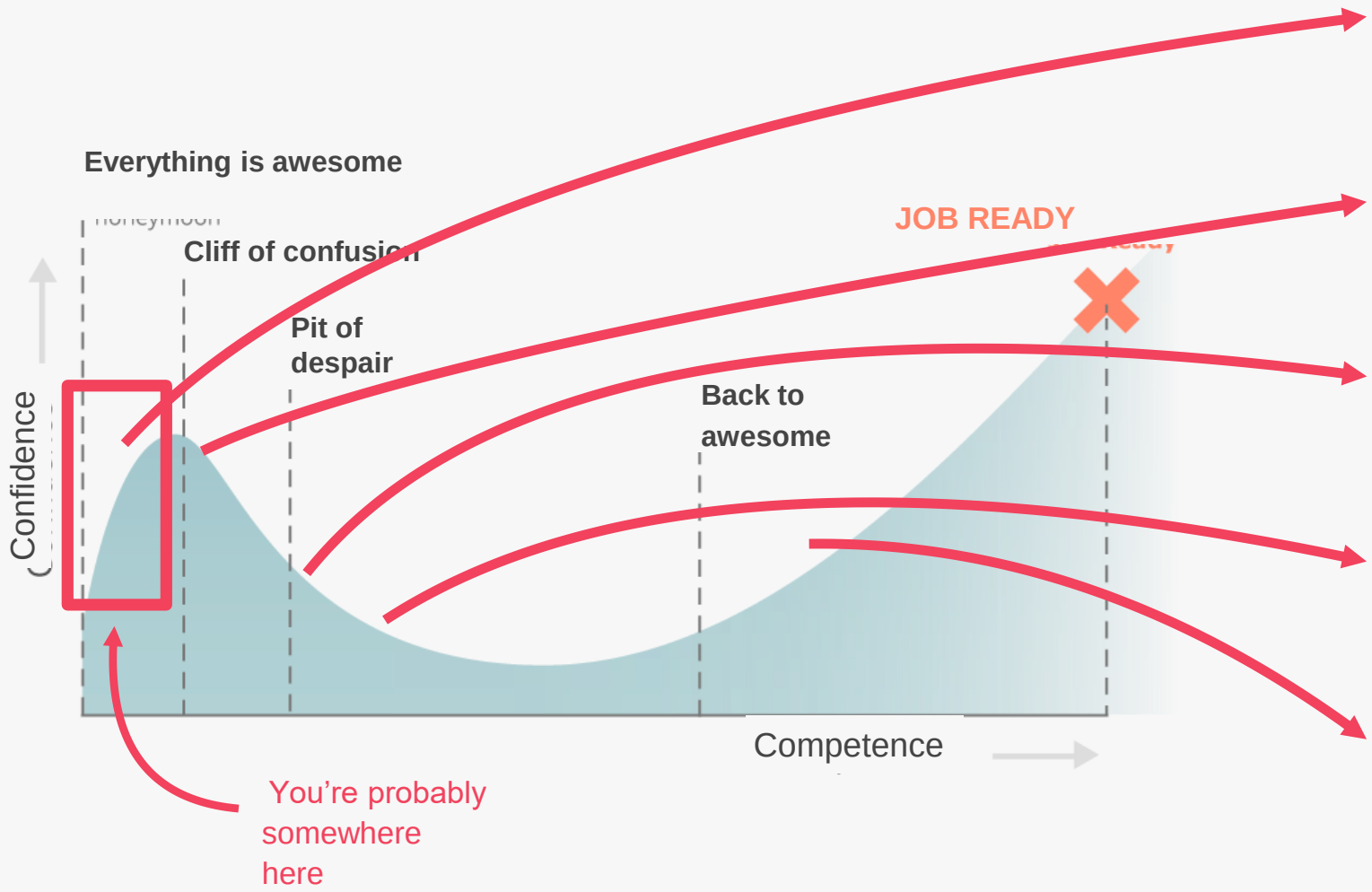
⬇ **FIX**

👍 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!



Study courses: understand code, take challenges and notes

↓

Stay motivated! Keep writing lots of code on your own, no matter how bad

↓

Learn with other people, devs and beginners, and share progress

↓

Keep challenging yourself, run into lots of problems, and fix them

↓

Round up your skillset with best practices and tools (git, testing, …)

↓

JOB READY
(But the learning never stops 😅
)

Source: https://www.thinkful.com/blog/why-learning-to-code-is-so-damn-hard/

THE COMPLETE JAVASCRIPT COURSE

FROM ZERO TO EXPERT!

SECTION

DEVELOPER SKILLS & EDITOR SETUP

LECTURE

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

HOW TO FAIL AT SOLVING PROBLEMS 🤦

JS

WHENEVER JOHN ENCOUNTERS A PROBLEM:

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

John can code now 😄

- 💥 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 e
- 👍 Use my **4-step framework** to solve any problem

➡ **NEXT SLIDE**

4 STEPS TO SOLVE ANY PROBLEM

JS

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 whatever that reverses 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?

Google stackoverflow MDN web docs moz://a

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
```

THE COMPLETE JAVASCRIPT COURSE

FROM ZERO TO EXPERT!

SECTION

DEVELOPER SKILLS & EDITOR SETUP

LECTURE

DEBUGGING (FIXING ERRORS)

JS

WHAT IS A SOFTWARE BUG?

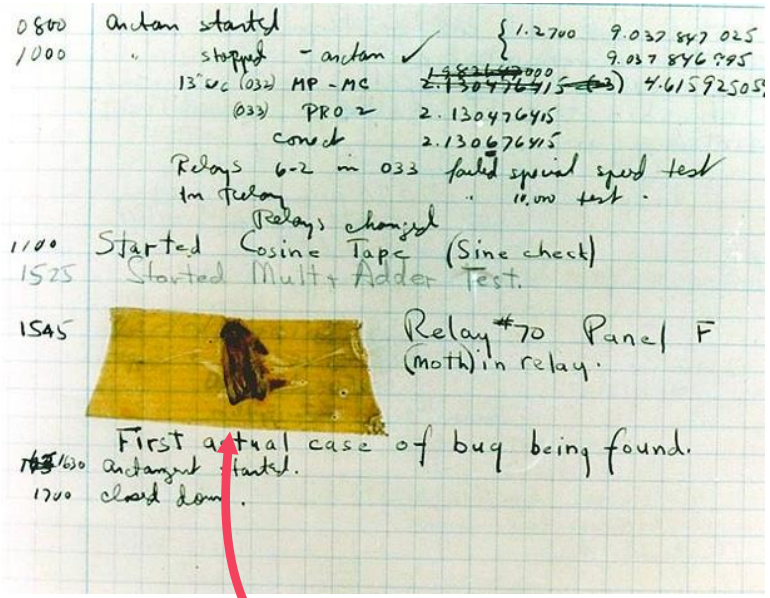
- 👉 **Software bug:** Defect or problem in a computer program. Basically, any **unexpected or unintended behavior** of a computer program is a software bug.
- 👉 Bugs are **completely normal** in software development!

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

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

```
[5, 1, 7, 3]
```

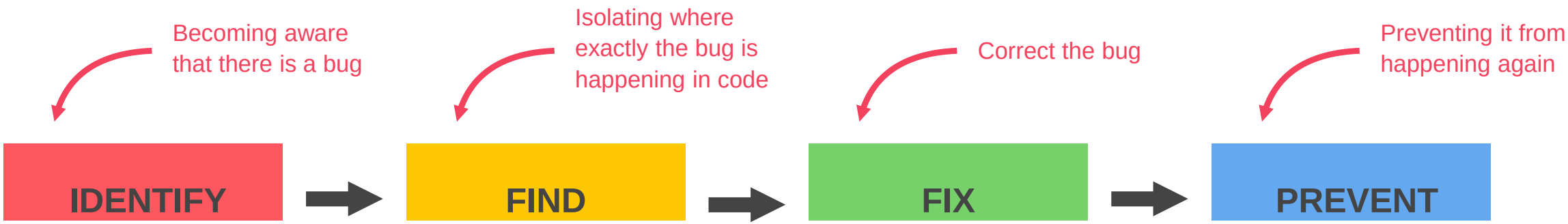
Unexpected result: the array is scrambled, **NOT** reversed. So there is a **bug** in the `reverse` function 🐛



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

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

THE DEBUGGING PROCESS



- 👉 During development
- 👉 Testing software
- 👉 User reports during production
- 👉 Context: browsers, users, etc.

- 👉 Developer console (*simple* code)
- 👉 Debugger (*complex* code)

- 👉 Replace wrong solution with new correct solution

- 👉 Searching for the same bug in similar code
- 👉 Writing tests using testing software

JS

JAVASCRIPT IN THE BROWSER: DOM AND EVENTS FUNDAMENTALS

THE COMPLETE JAVASCRIPT COURSE

FROM ZERO TO EXPERT!

SECTION

JAVASCRIPT IN THE BROWSER: DOM AND EVENTS FUNDAMENTALS

LECTURE

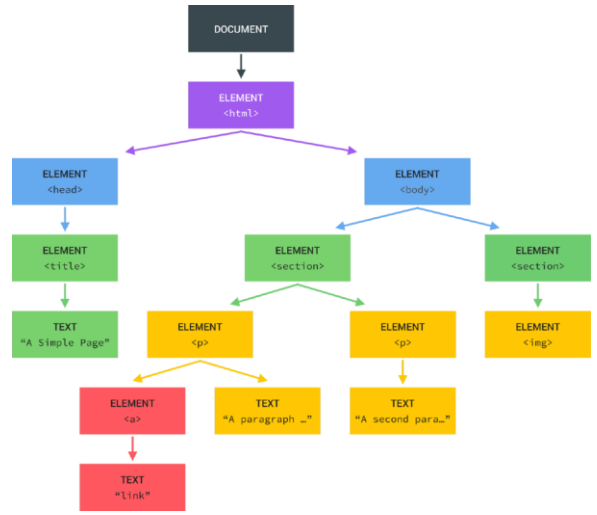
WHAT'S THE DOM AND DOM MANIPULATION

JS

WHAT IS THE DOM?

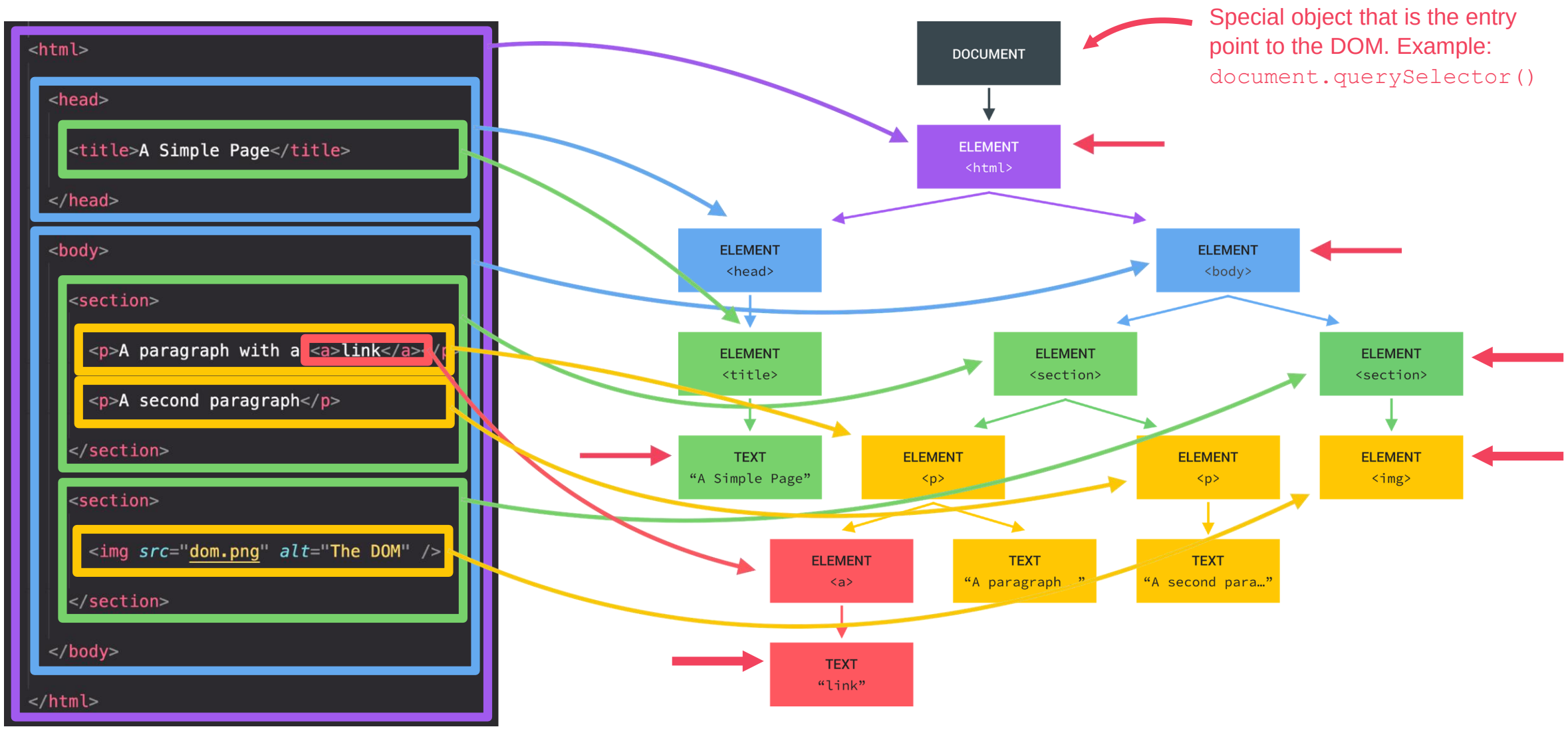
DOCUMENT OBJECT MODEL: STRUCTURED
REPRESENTATION OF HTML DOCUMENTS.
DOM
SCRIPT TO ACCESS HTML
ELEMENTS AND STYLES TO MANIPULATE THEM.

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

👎

NOT PART OF

JS

ecma INTERNATIONAL

For example
`document.querySelector()`

API: Application Programming Interface

WEB APIs

DOM Methods and Properties

Timers

Fetch

👍

CAN INTERACT WITH

JS

JS

HOW JAVASCRIPT WORKS BEHIND THE SCENES

THE COMPLETE JAVASCRIPT COURSE

FROM ZERO TO EXPERT!

SECTION

HOW JAVASCRIPT WORKS BEHIND THE SCENES

LECTURE

AN HIGH-LEVEL OVERVIEW OF JAVASCRIPT

JS

WHAT IS JAVASCRIPT: REVISITED

JAVASCRIPT

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

WHAT IS JAVASCRIPT: REVISITED

JAVASCRIPT

FIRST-CLASS FUNCTIONS

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

DECONSTRUCTING THE MONSTER DEFINITION

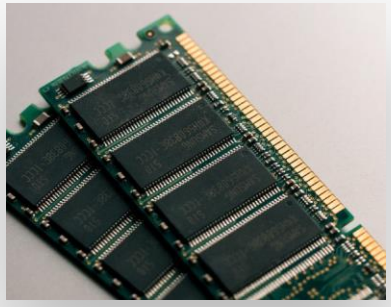
JS

- 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

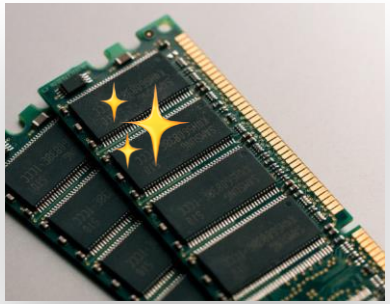
Developer does **NOT** have to worry, everything happens automatically

DECONSTRUCTING THE MONSTER

DEFINITION

JS

- 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

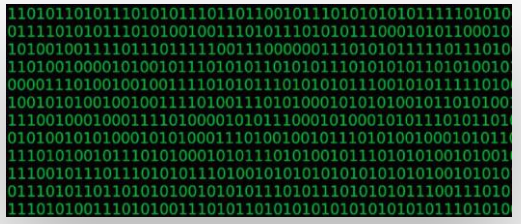
DECONSTRUCTING THE MONSTER DEFINITION

- 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



Happens inside the JavaScript engine

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

DECONSTRUCTING THE MONSTER

DEFINITION

JS

- 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**

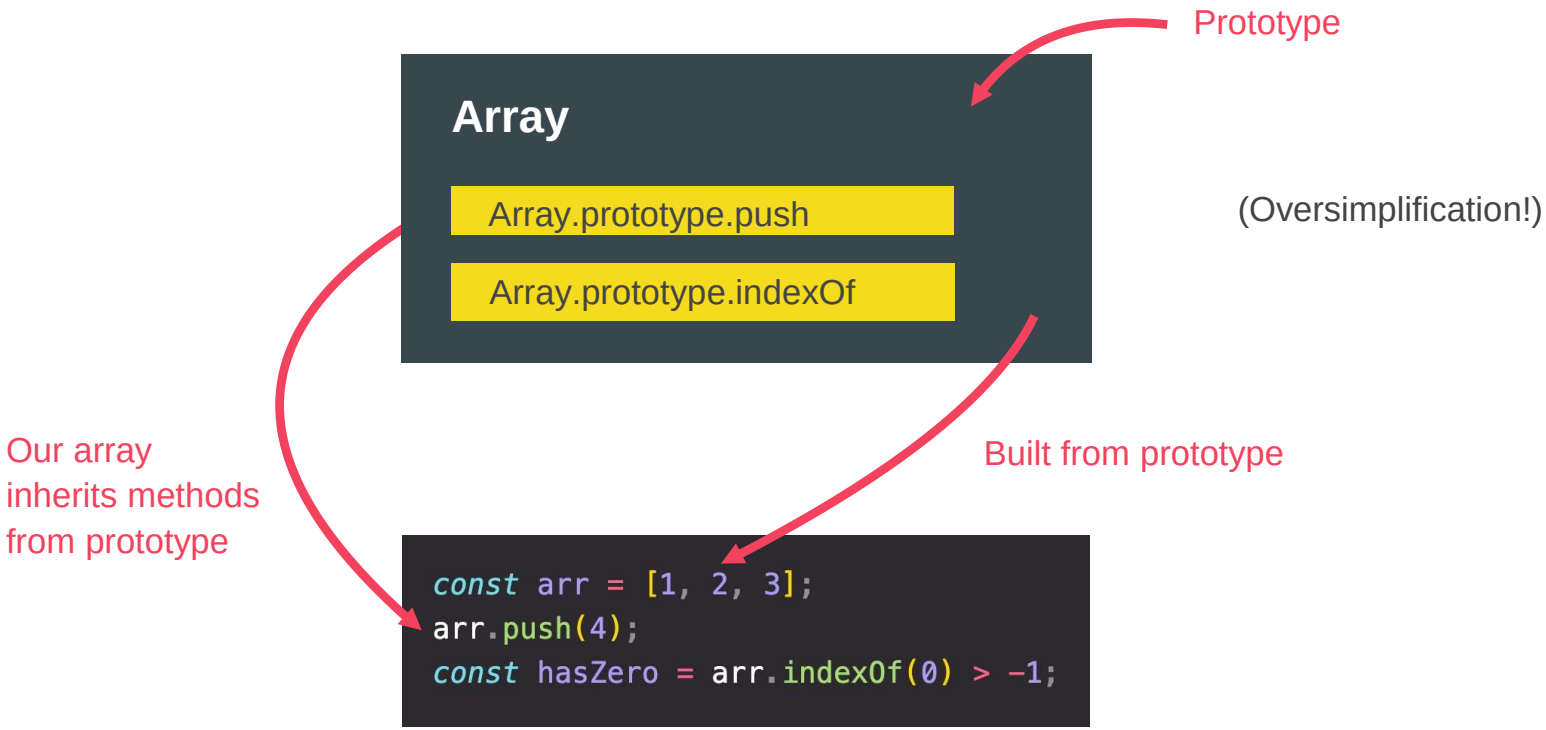
More about this later in **Multiple Sections** 👉

DECONSTRUCTING THE MONSTER

DEFINITION

JS

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



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

DECONSTRUCTING THE MONSTER

DEFINITION

JS

- 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";
```

TS

DECONSTRUCTING THE MONSTER DEFINITION

JS

- 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!

(Oversimplification!)

⬇ How do we achieve that?

👉 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** 👇

THE COMPLETE JAVASCRIPT COURSE

FROM ZERO TO EXPERT!

SECTION

HOW JAVASCRIPT WORKS BEHIND THE SCENES

LECTURE

THE JAVASCRIPT ENGINE AND RUNTIME

WHAT IS A JAVASCRIPT ENGINE?

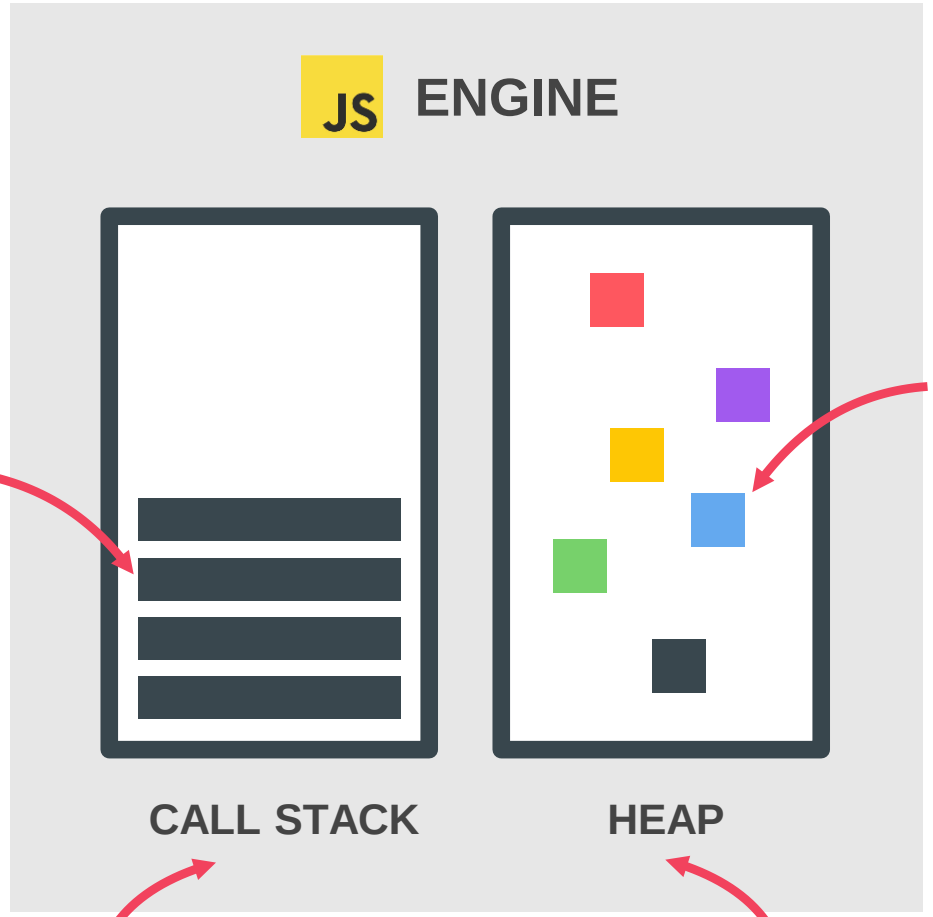
JS ENGINE

PROGRAM THAT EXECUTES

JAVASCRIPT CODE.

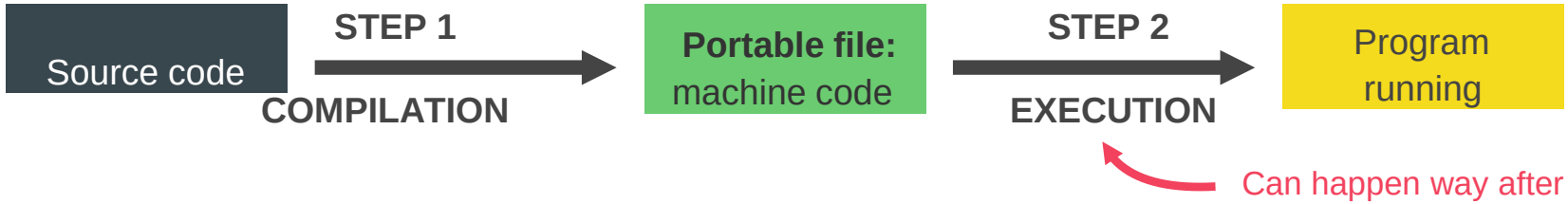
👉 **Example: V8 Engine**

How is it compiled?

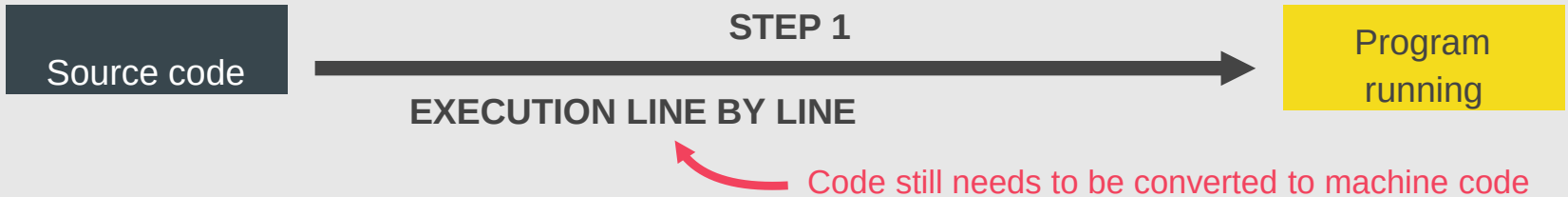


COMPUTER SCIENCE SIDENOTE: COMPILATION VS. INTERPRETATION

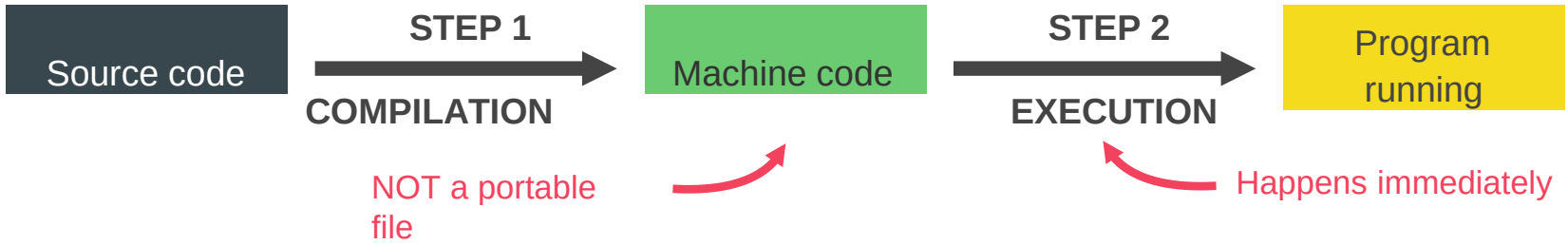
👉 Compilation: Entire code is converted into machine code at once, and written to a binary file that can be executed by a computer.



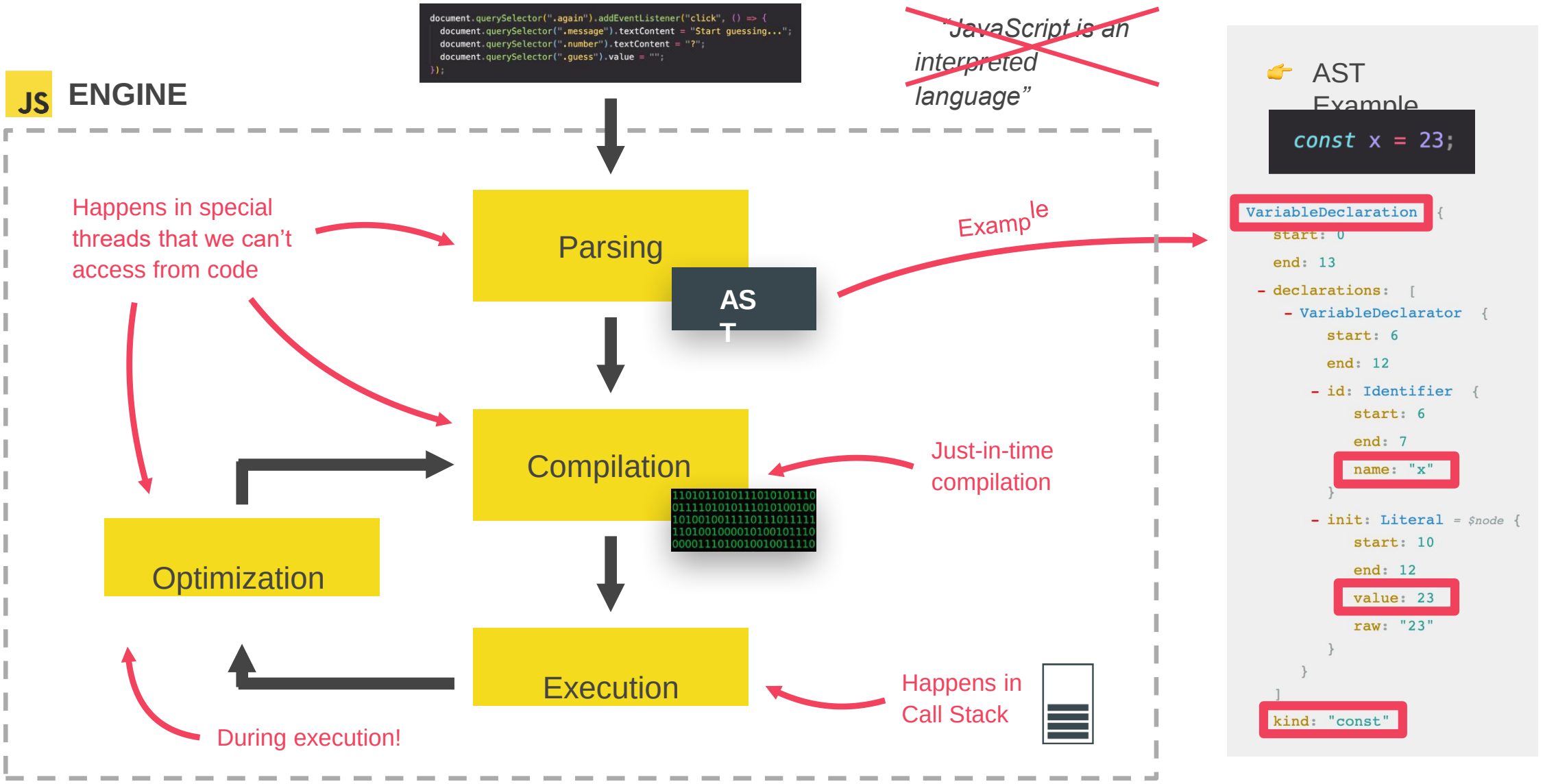
👉 Interpretation: Interpreter runs through the source code and executes it line by line.



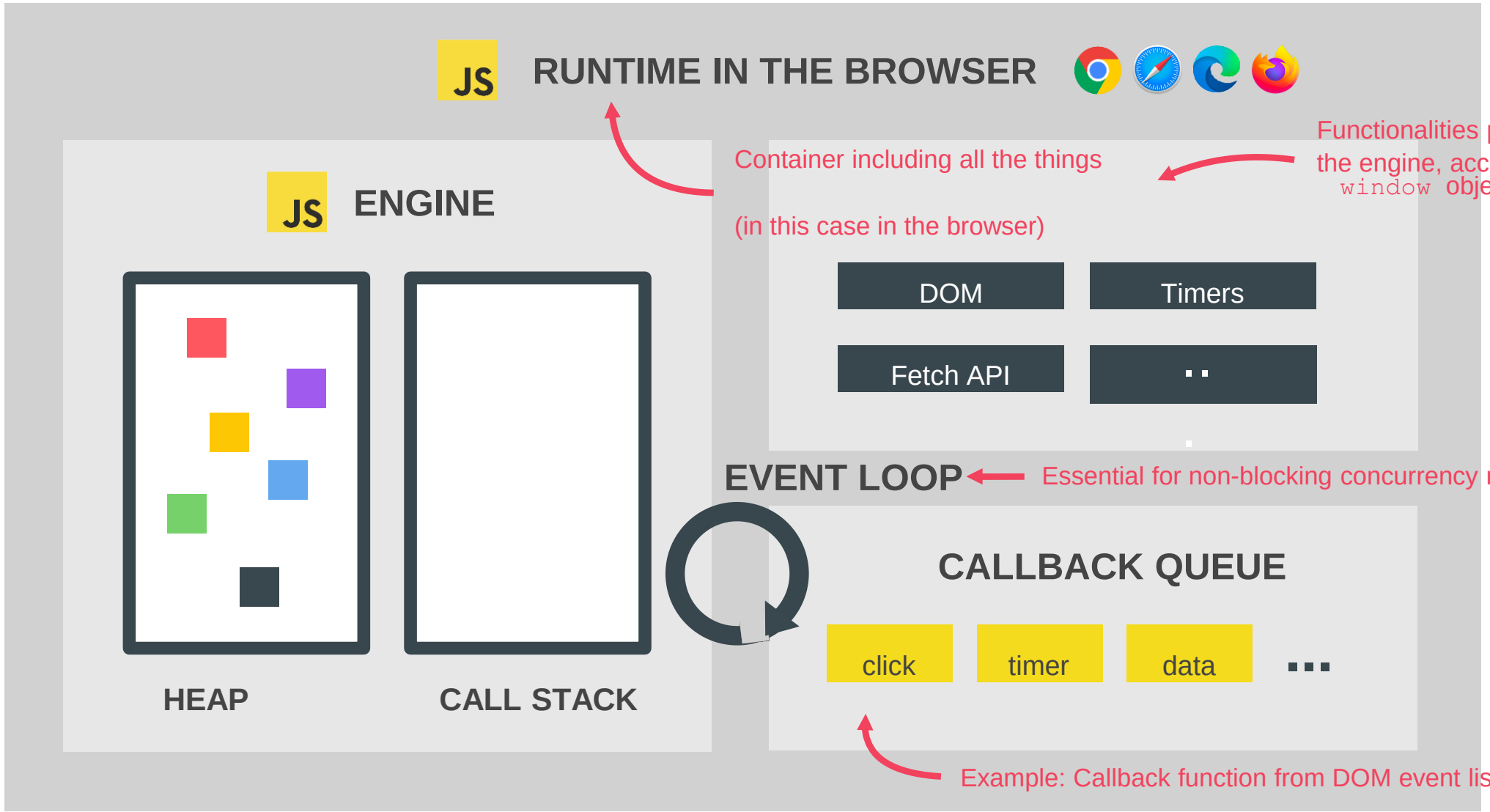
👉 Just-in-time (JIT) compilation: Entire code is converted into machine code at once, then executed immediately.



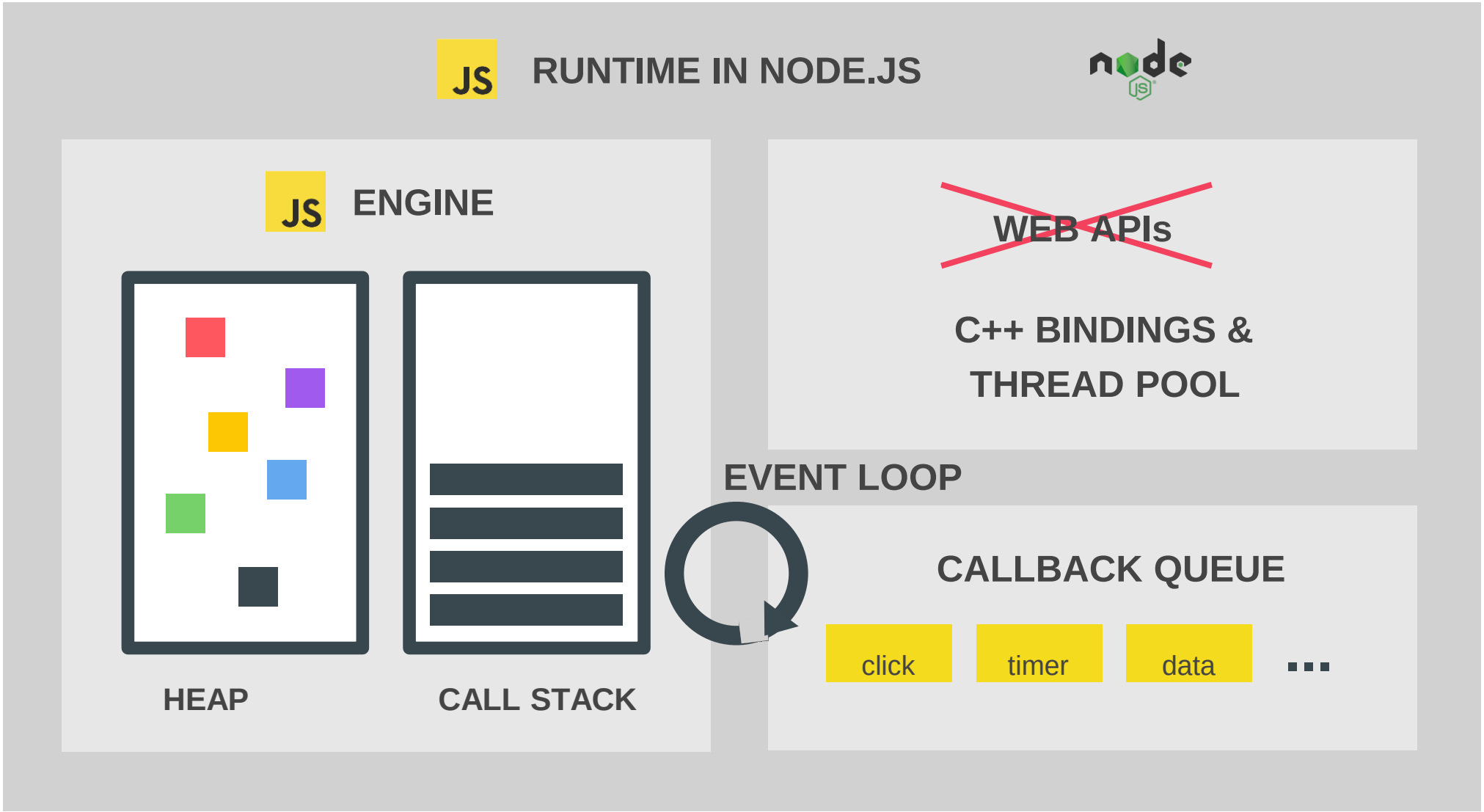
MODERN JUST-IN-TIME COMPILATION OF JAVASCRIPT



THE BIGGER PICTURE: JAVASCRIPT RUNTIME



THE BIGGER PICTURE: JAVASCRIPT RUNTIME



THE COMPLETE JAVASCRIPT COURSE

FROM ZERO TO EXPERT!

SECTION

HOW JAVASCRIPT WORKS BEHIND THE SCENES

LECTURE

EXECUTION CONTEXTS AND THE CALL STACK

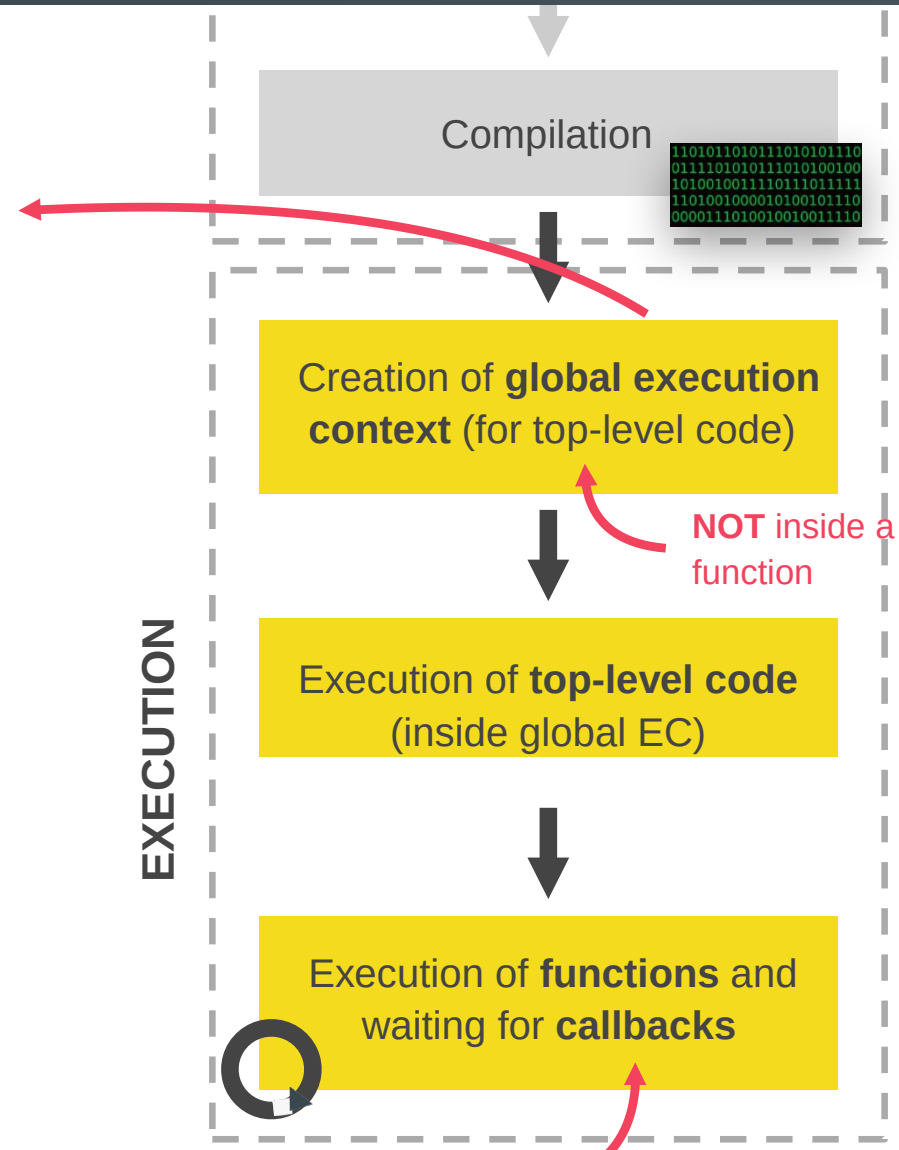
JS

WHAT IS AN EXECUTION CONTEXT?

👉 Human-readable code:

```
const name = 'Jonas';

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

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

Function body only executed when called!



EXECUTION CONTEXT

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

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]
```

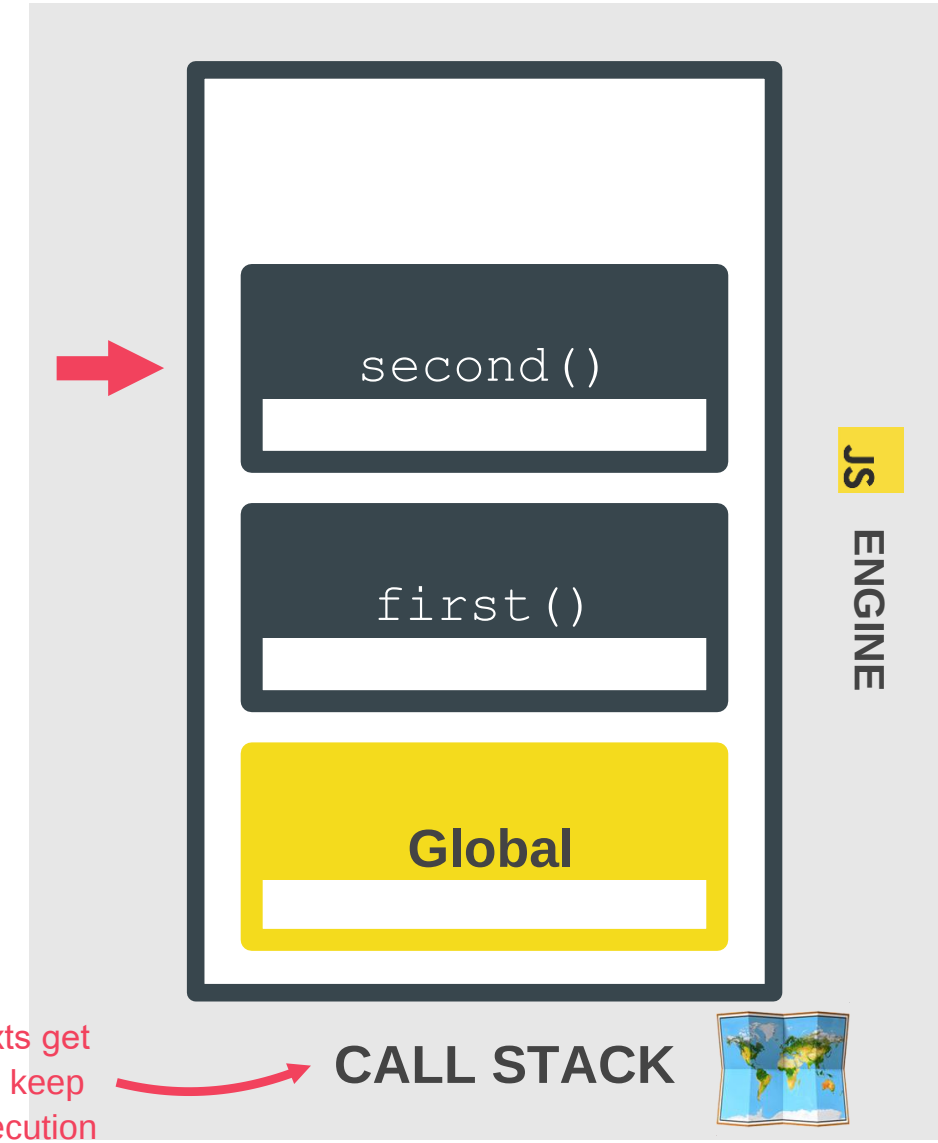
Array of passed arguments. Available in all "regular" functions (not arrow)

(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

THE COMPLETE JAVASCRIPT COURSE

FROM ZERO TO EXPERT!

SECTION

HOW JAVASCRIPT WORKS BEHIND THE SCENES

LECTURE

SCOPE AND THE SCOPE CHAIN

JS

SCOPING AND SCOPE IN JAVASCRIPT: CONCEPTS

EXECUTION CONTEXT

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

SCOPE CONCEPTS

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

THE 3 TYPES OF SCOPE

GLOBAL SCOPE

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

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

FUNCTION SCOPE

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

BLOCK SCOPE (ES6)

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

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

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

THE SCOPE CHAIN

```
const myName = 'Jonas';

function first() {
  const age = 30;

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

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

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

  second();
}

first();
```

let and const are **block-scoped**

var is **function-scoped**

Variables not in current scope

(Considering only variable declarations)

VARIABLE LOOKUP IN SCOPE CHAIN

Global variable

Global scope
- myName = "Jonas"

SCOPE CHAIN

first() scope
- age = 30
- millennial = true
- myName = "Jonas"

Scope has access to variables from all **outer scopes**

if block scope
- decade = 3
- age = 30
- millennial = true
- myName = "Jonas"

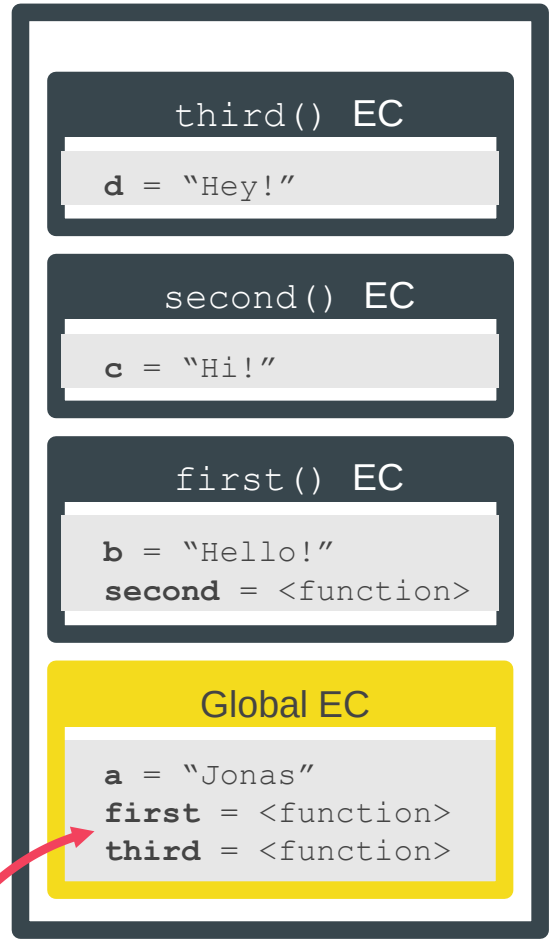
second() scope
- job = "teacher"
- age = 30
- millennial = true
- myName = "Jonas"

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
}
```

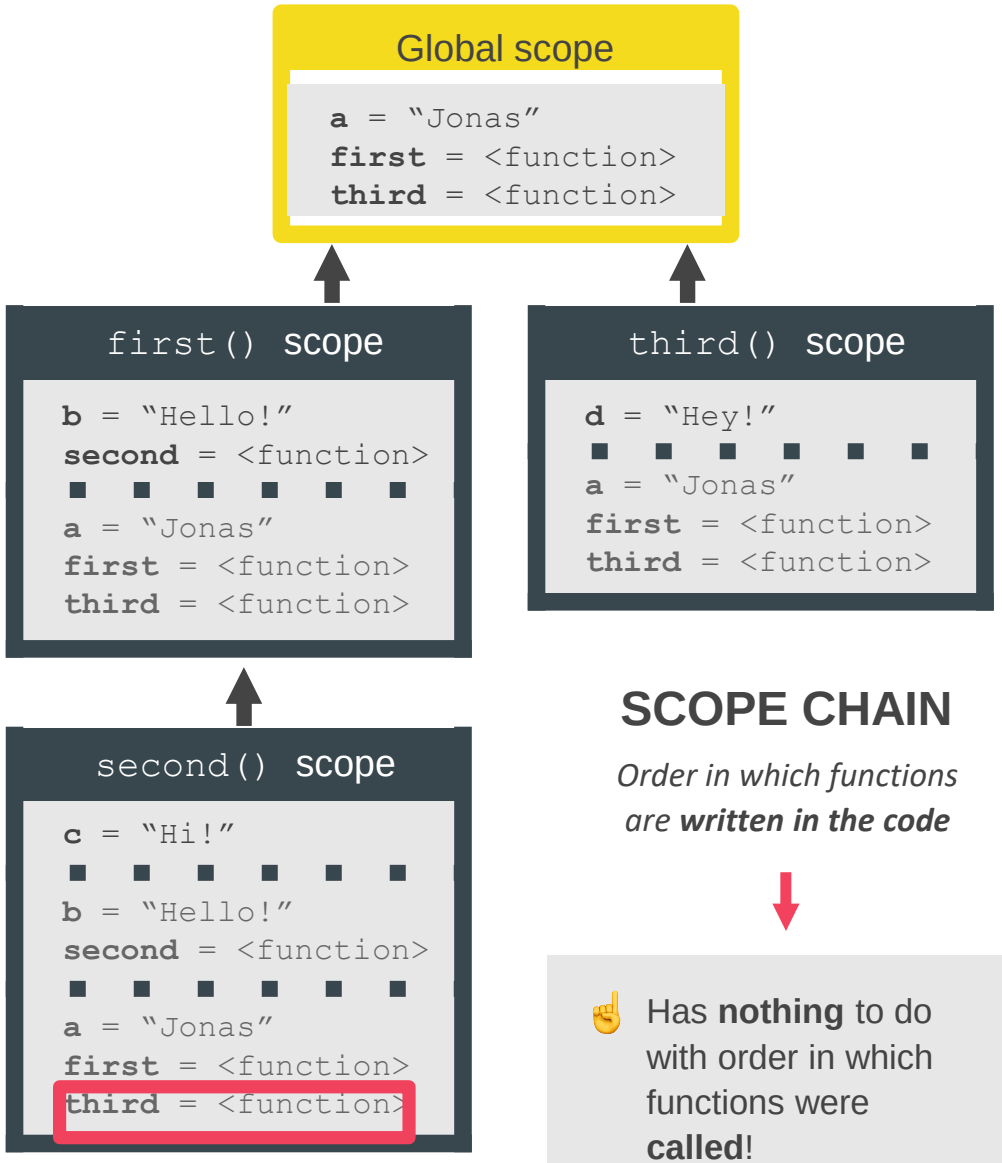
`c` and `b` can NOT be found in `third()` scope!

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;
- 👉 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 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 f 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!

THE COMPLETE JAVASCRIPT COURSE

FROM ZERO TO EXPERT!

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”.

EXECUTION CONTEXT

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

⬇ ***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.

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

In strict mode. Otherwise: function!

Technically, yes. But not in practice

Temporal Dead Zone

TEMPORAL DEAD ZONE, LET AND CONST

```
const myName = 'Jonas';

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

TEMPORAL DEAD ZONE FOR `job` VARIABLE

👉 Different kinds of error messages:

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

WHY HOISTING?

- 👉 Using functions before actual declaration;
- 👉 `var` hoisting is just a by product.

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

THE COMPLETE JAVASCRIPT COURSE

FROM ZERO TO EXPERT!

SECTION

HOW JAVASCRIPT WORKS BEHIND THE SCENES

LECTURE

THE THIS KEYWORD

JS

HOW THE THIS KEYWORD WORKS

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

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

EXECUTION CONTEXT

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

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

Simple function call 👉 `this = undefined`

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

Don't get own **`this`**

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

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

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

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

👉 Method example:

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

`calcAge` is method

`jonas`

`1989`

Way better than using `jonas.year`!

THE COMPLETE JAVASCRIPT COURSE

FROM ZERO TO EXPERT!

GOT QUESTIONS? FEEDBACK?

THE COMPLETE JAVASCRIPT COURSE

FROM ZERO TO EXPERT!

SECTION

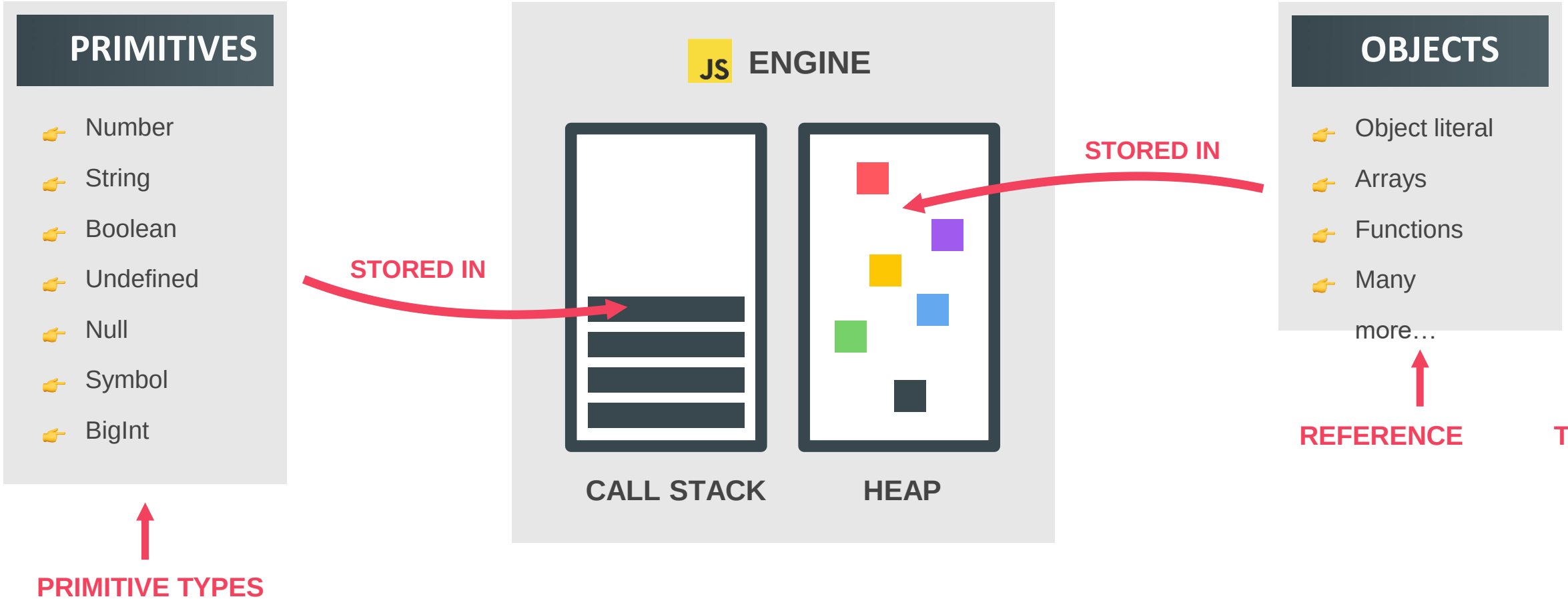
HOW JAVASCRIPT WORKS BEHIND THE SCENES

LECTURE

PRIMITIVES VS. OBJECTS (PRIMITIVE VS. REFERENCE TYPES)

JS

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

JS

DATA STRUCTURES, MODERN OPERATORS AND STRINGS

THE COMPLETE JAVASCRIPT COURSE

FROM ZERO TO EXPERT!

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)
1. From the UI: Data input from the user or data written in DOM (e.g tasks in todo app)
1. From external sources: Data fetched for example from web API (e.g. recipe objects)

Application Programming Interface

👉 JSON data format example

Collection of data

Data structure

SIMPLE LIST?

KEY/VALUE PAIRS?

Arrays or Sets

Objects or Maps

Keys allow us to *describe* values

OTHER BUILT-IN:

- 👉 WeakMap
- 👉 WeakSet

NON-BUILT IN:

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

ARRAYS VS. SETS AND OBJECTS VS. MAPS

ARRAYS VS. SETS

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

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

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

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

OBJECTS VS. MAPS

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

- 👉 More “traditional” key/value store (“abused” objects)
- 👉 Easier to write and access values with `.` and `[]`
- 👉 Use when you need to include functions (methods)
- 👉 Use when working with JSON (can convert to map)

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

- 👉 Better performance
- 👉 Keys can have any data
- 👉 type Easy to iterate
- 👉 Easy to compute size
- 👉 Use when you simply need to map key to values
- 👉 Use when you need keys that are not strings

JS

A CLOSER LOOK AT FUNCTIONS

THE COMPLETE JAVASCRIPT COURSE

FROM ZERO TO EXPERT!

SECTION

A CLOSER LOOK AT FUNCTIONS

LECTURE

FIRST-CLASS AND HIGHER-ORDER FUNCTIONS

JS

FIRST-CLASS VS. HIGHER-ORDER FUNCTIONS

FIRST-CLASS FUNCTIONS

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

- 👉 Store functions in variables or properties:

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

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

- 👉 Pass functions as arguments to OTHER functions:

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

- 👉 Return functions FROM functions
- 👉 Call methods on functions:

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

HIGHER-ORDER FUNCTIONS

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

1 Function that receives another function

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

Higher-order function

Callback function

✅📞💬

2 Function that returns new function

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

Higher-order function

Returned function

THE COMPLETE JAVASCRIPT COURSE

FROM ZERO TO EXPERT!

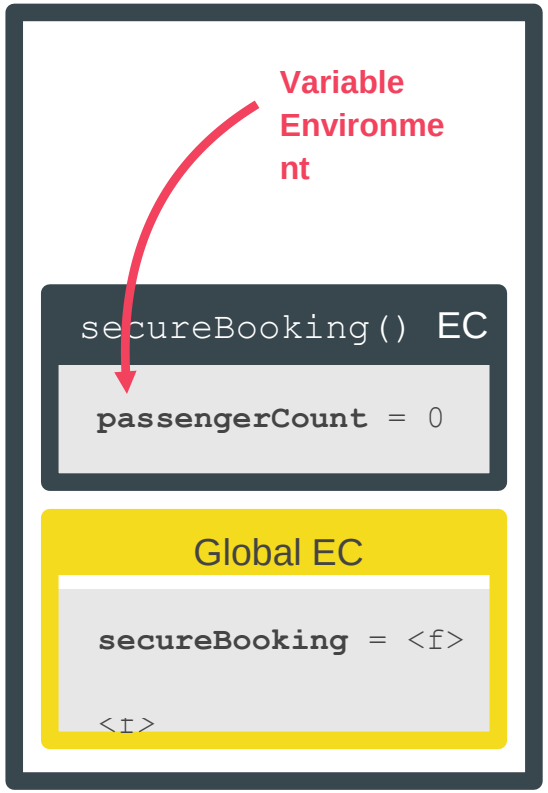
SECTION

A CLOSER LOOK AT FUNCTIONS

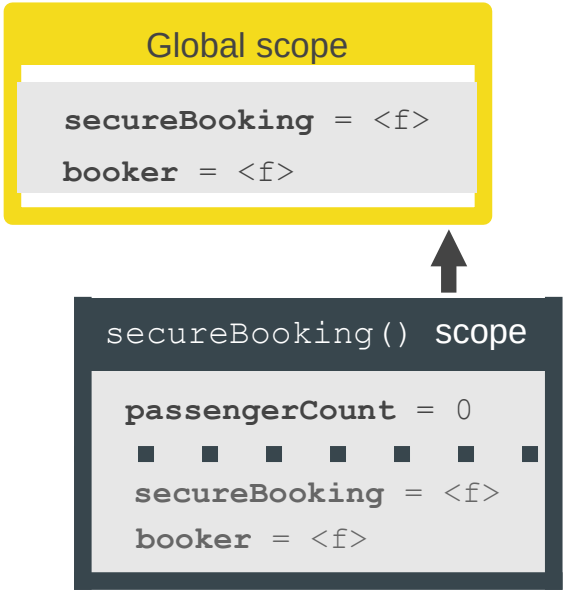
LECTURE

CLOSURES

“CREATING” A CLOSURE




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

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

const booker = secureBooking();
```



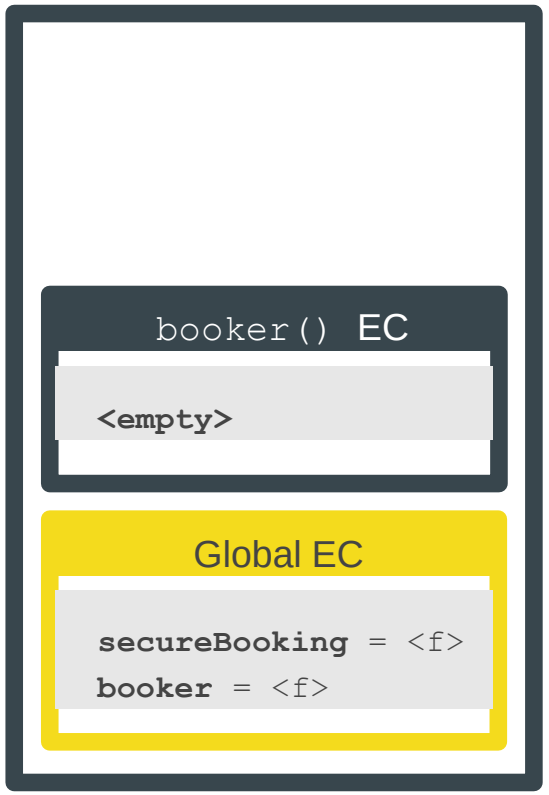
CALL STACK

Order in which functions were *called*

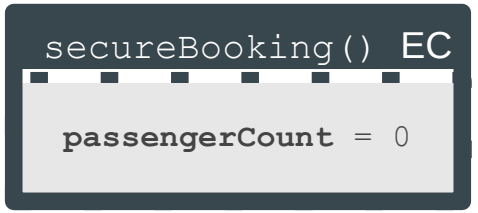
Order in which functions are ***written in the code***

SCOPE CHAIN

UNDERSTANDING CLOSURES



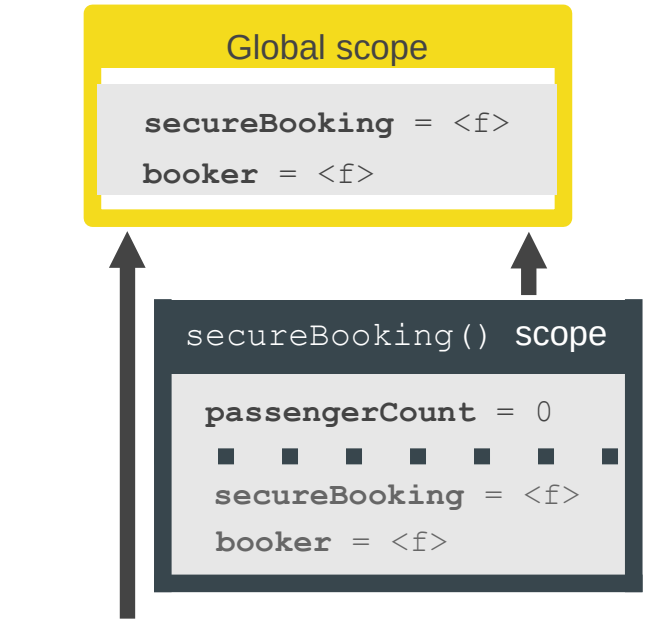
CALL STACK



Variable Environment of Execution Context in which `booker` was created

```
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

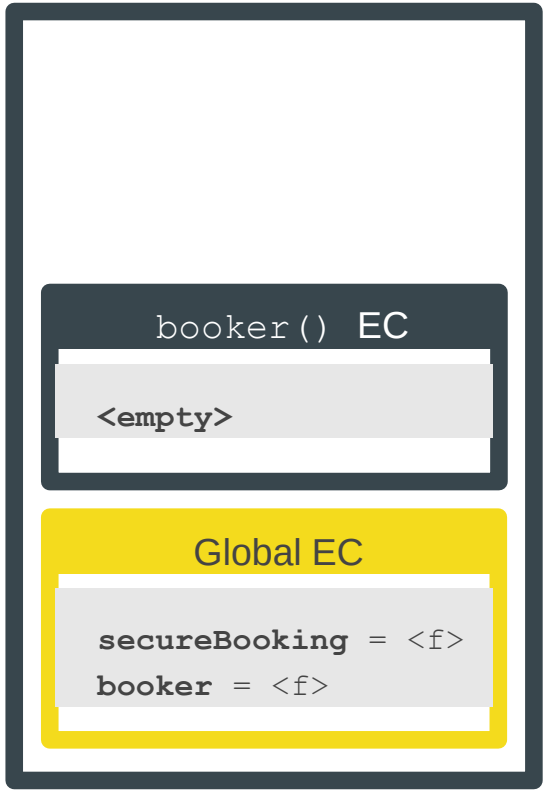


How to access `passengerCount` ?

SCOPE CHAIN

UNDERSTANDING CLOSURES

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



CALL STACK

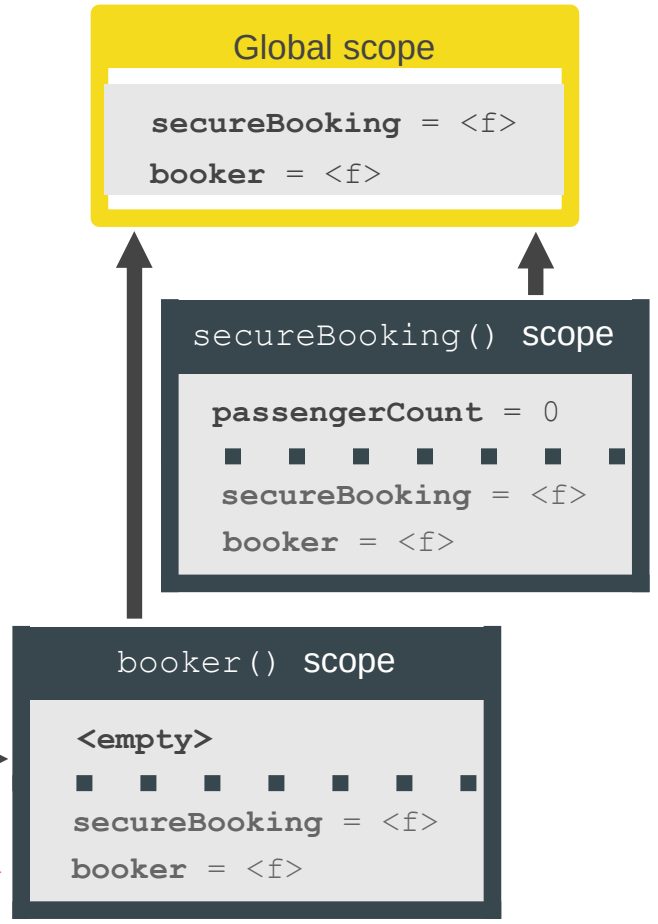
```
const secureBooking = function () {
  let passengerCount = 0;

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

const booker = secureBooking();

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

This is the function

passengerCount = 2

(**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;

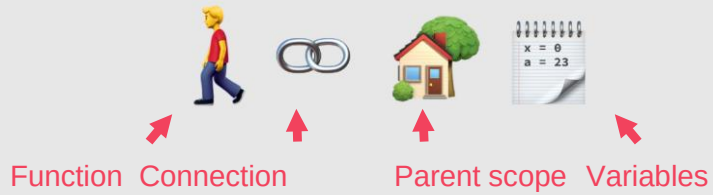
Less formal

👉 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.

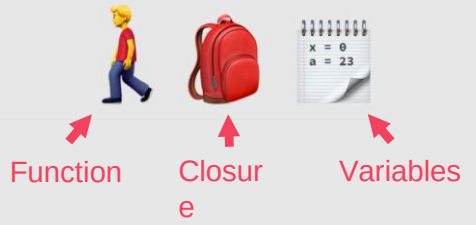
Less formal

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

Less formal



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



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

JS

WORKING WITH ARRAYS

THE COMPLETE JAVASCRIPT COURSE

FROM ZERO TO EXPERT!

SECTION

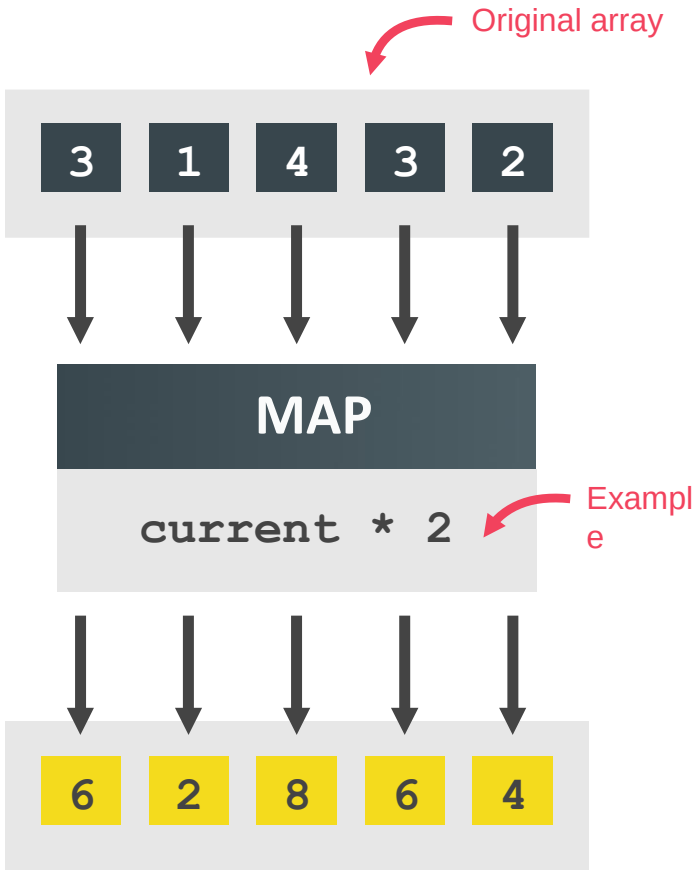
WORKING WITH ARRAYS

LECTURE

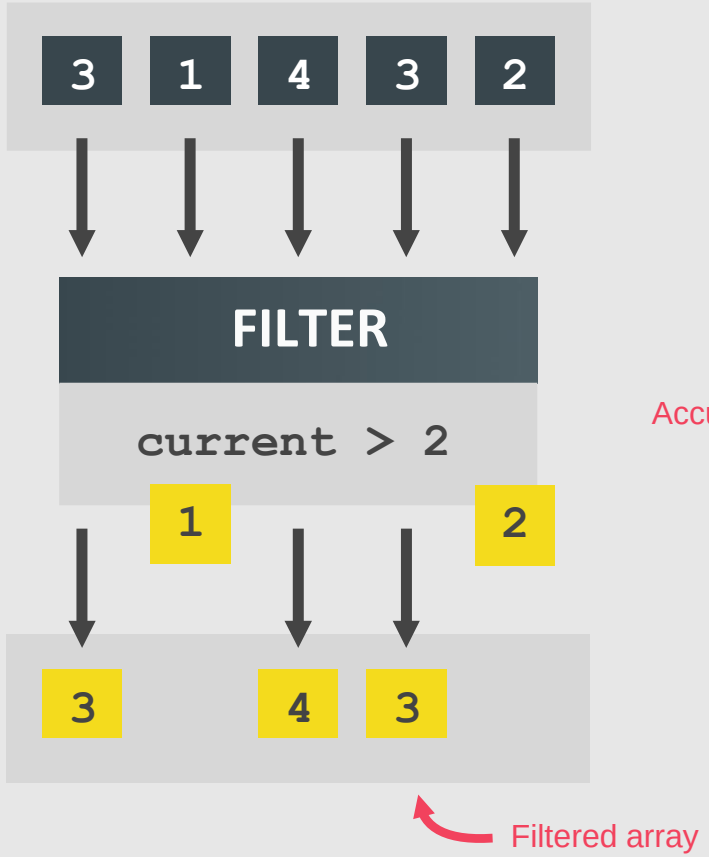
DATA TRANSFORMATIONS: MAP, FILTER, REDUCE

JS

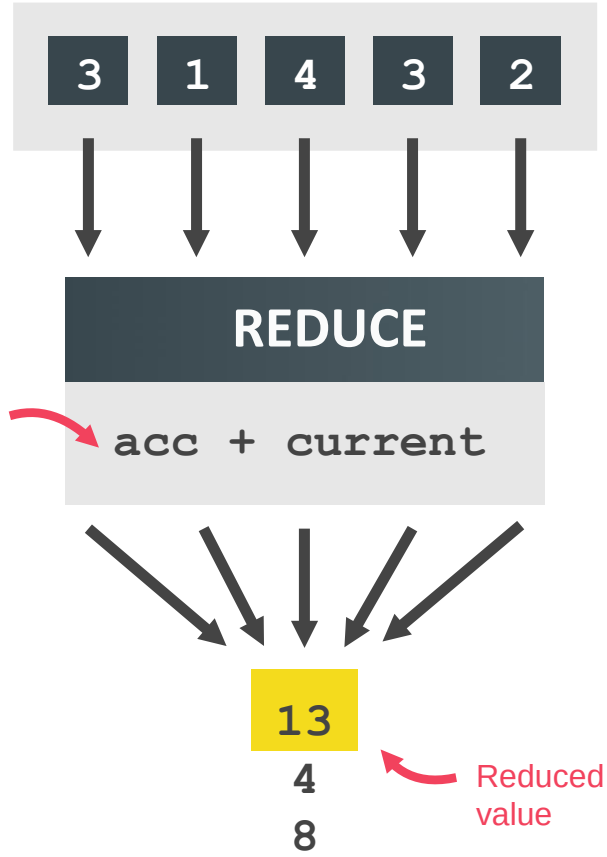
DATA TRANSFORMATIONS WITH MAP, FILTER AND REDUCE



👉 `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)

THE COMPLETE JAVASCRIPT COURSE

FROM ZERO TO EXPERT!

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)

JS

ADVANCED DOM AND EVENTS

THE COMPLETE JAVASCRIPT COURSE

FROM ZERO TO EXPERT!

SECTION

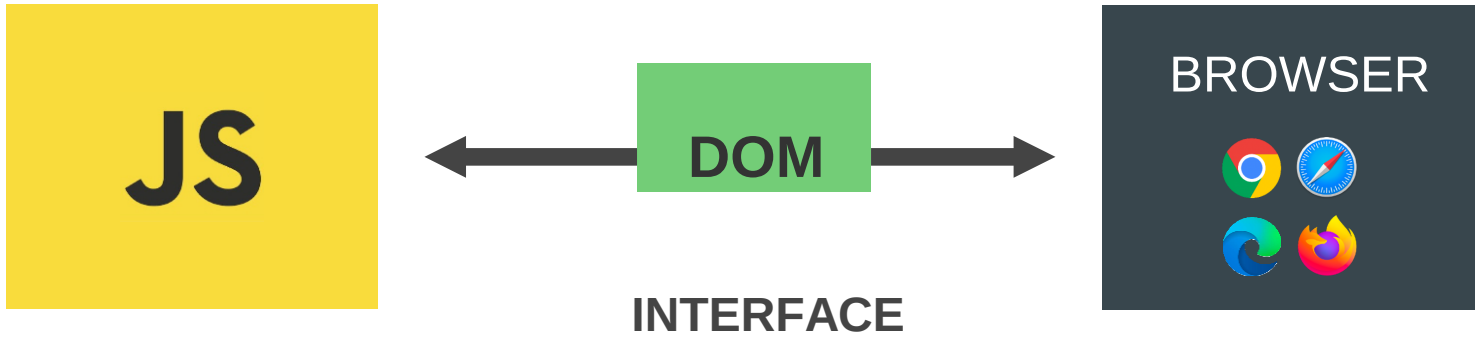
ADVANCED DOM AND EVENTS

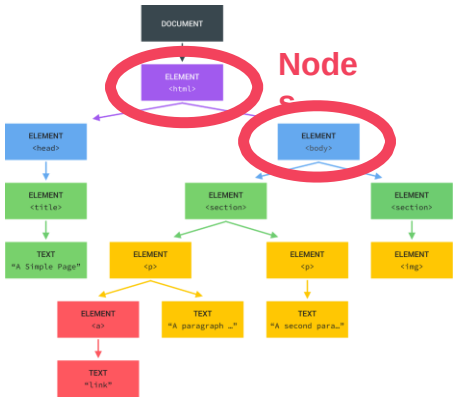
LECTURE

HOW THE DOM REALLY WORKS

JS

REVIEW: WHAT IS THE DOM?



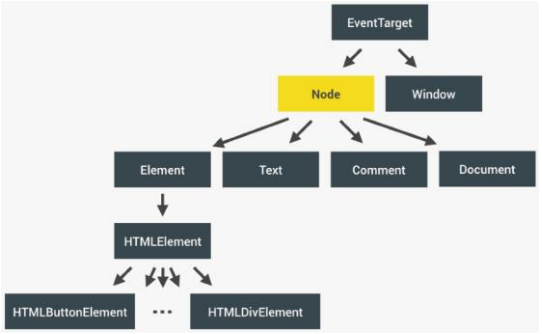


DOM tree

- 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

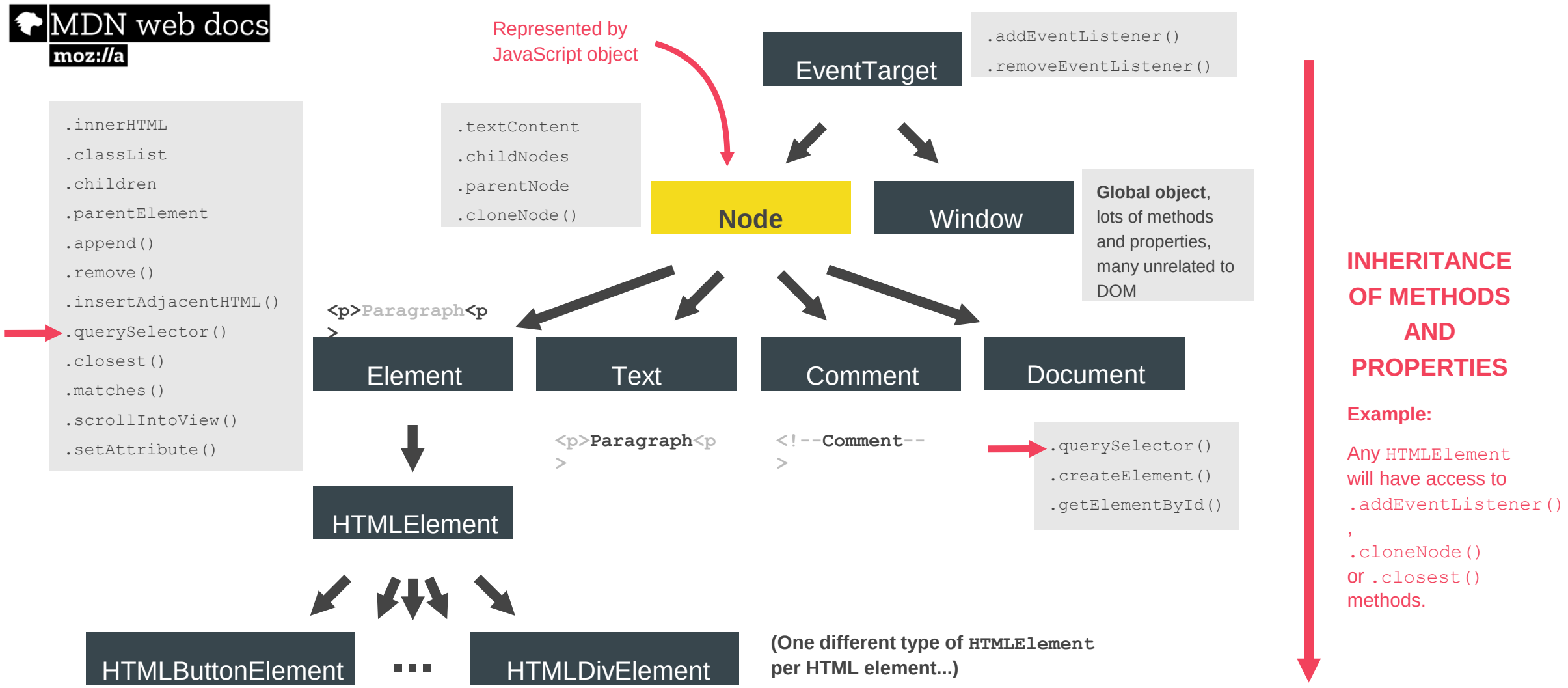
Application Programming Interface

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



"Types" of DOM objects (next slide)

HOW THE DOM API IS ORGANIZED BEHIND THE SCENES



THE COMPLETE JAVASCRIPT COURSE

FROM ZERO TO EXPERT!

SECTION

ADVANCED DOM AND EVENTS

LECTURE

EVENT PROPAGATION: BUBBLING AND CAPTURING

JS

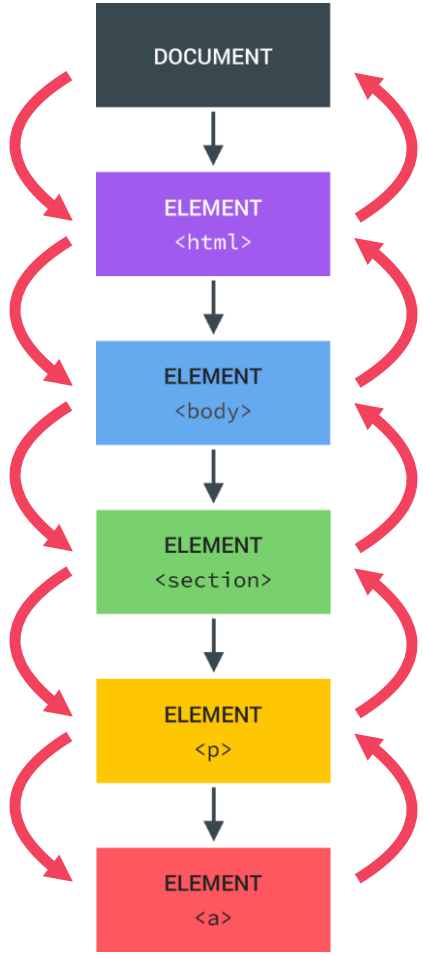
BUBBLING AND CAPTURING

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

Click event

1

CAPTURING 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 😀

(THIS DOES NOT HAPPEN ON *ALL* EVENTS)

2 TARGET PHASE

THE COMPLETE JAVASCRIPT COURSE

FROM ZERO TO EXPERT!

SECTION

ADVANCED DOM AND EVENTS

LECTURE

EFFICIENT SCRIPT LOADING: DEFER AND ASYNC

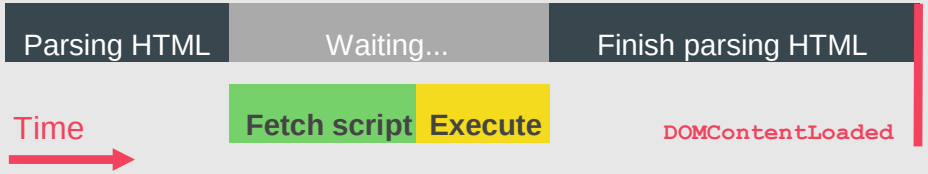
DEFER AND ASYNC SCRIPT LOADING

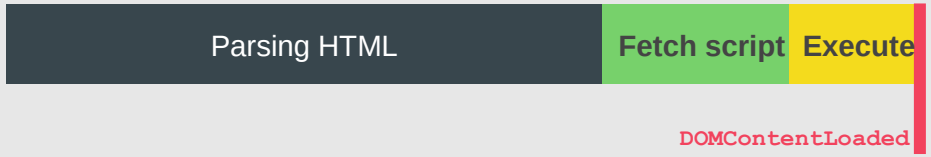
HEAD

BODY END

REGULAR

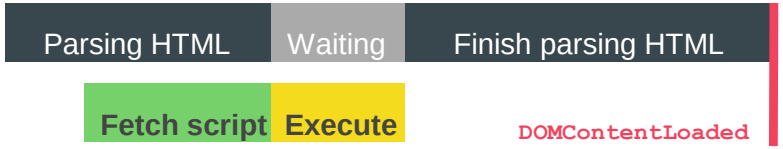
```
<script src="script.js">
```





ASYNC

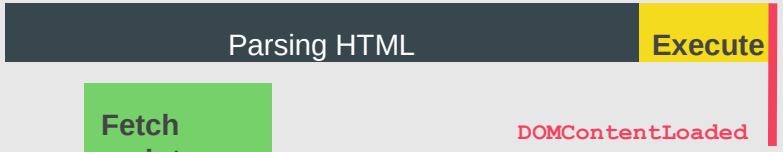
```
<script async src="script.js">
```



👉 Makes no sense 🤷

DEFER

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



👉 Makes no sense 🤷

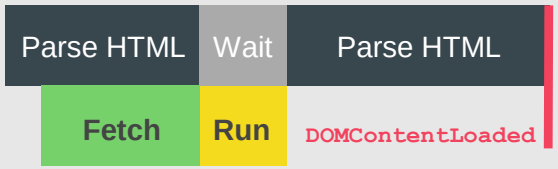
REGULAR VS. ASYNC VS. DEFER

END OF BODY

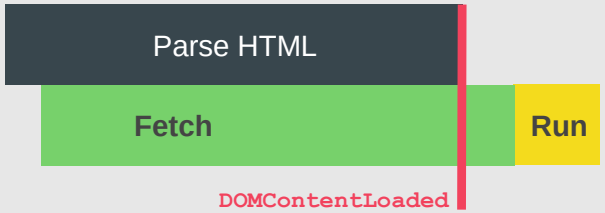
- Scripts are fetched and executed ***after the HTML is completely parsed***
- **Use if you need to support old browsers**

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

ASYNC IN HEAD

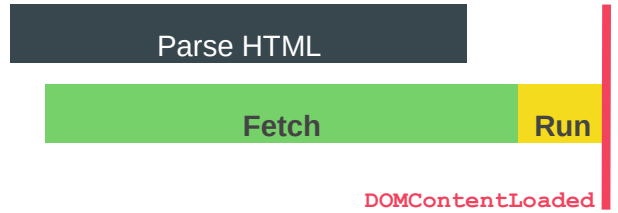


- Scripts are fetched ***asynchronously*** and executed ***immediately***
- Usually the `DOMContentLoaded` event waits for ***all*** scripts to execute, except for `async` scripts. So, `DOMContentLoaded` does ***not*** wait for an `async` script
- Scripts ***not*** guaranteed to execute in order
- **Use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



DEFER IN HEAD

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



JS

OBJECT ORIENTED PROGRAMMING (OOP) WITH JAVASCRIPT

THE COMPLETE JAVASCRIPT COURSE

FROM ZERO TO EXPERT!

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

Data

```
const user = {
  user: 'jonas',
  password: 'dk23s',

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

Behaviour

- Object-oriented programming (OOP) is a programming paradigm based on the concept of objects;
- We use objects to **model** (describe) real-world or abstract features;
- OOP was developed with the goal of **organizing** code, to make it **more flexible and easier to maintain** (avoid “spaghetti code”).

Objects may contain data (properties) and code (methods). objects, we pack **data and the corresponding behavior** in

E.g. user or todo list item

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

E.g. HTML component or data structure

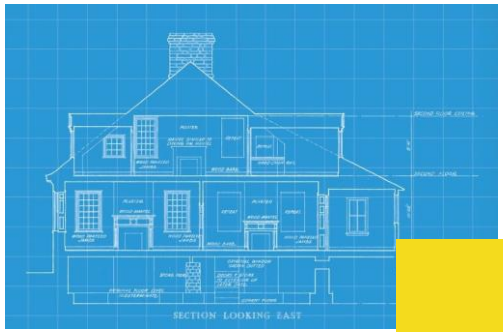
Objects are **building blocks** of applications, and **interact** w

another; Interactions happen through a **public interface** (A

methods that the code

outside of the object can access and use to communicate w object;

CLASSES AND INSTANCES (TRADITIONAL OOP)



Like a blueprint from which we can create **new objects**

CLASS

```
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

new User('jonas')

new User('mary')

new User('steven')

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
  }
}
```

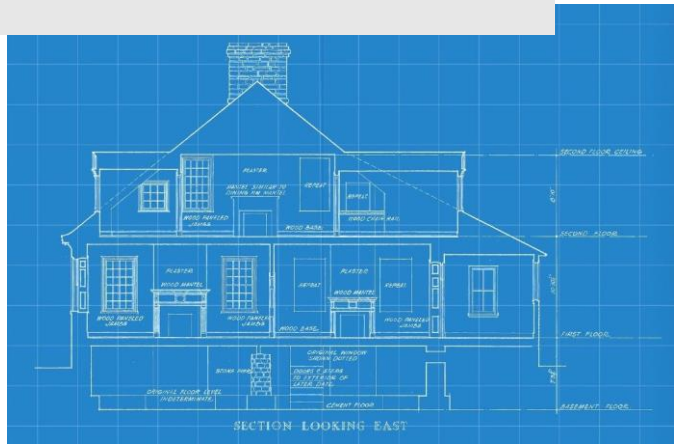
☝ Conceptual overview: it works a bit **differently** in JavaScript. Still important to understand!

THE 4 FUNDAMENTAL OOP PRINCIPLES

- Abstraction
- Encapsulation
- Inheritance
- Polymorphism

The 4 fundamental principles of Object-Oriented Programming

🤔 *“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() {}
}
```

Real phone

Do we *really need* all these low-level details?

Abstracted phone

```
Phone {
  charge
  volume

  homeBtn() {}
  volumeBtn() {}
  screen() {}
}
```

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)

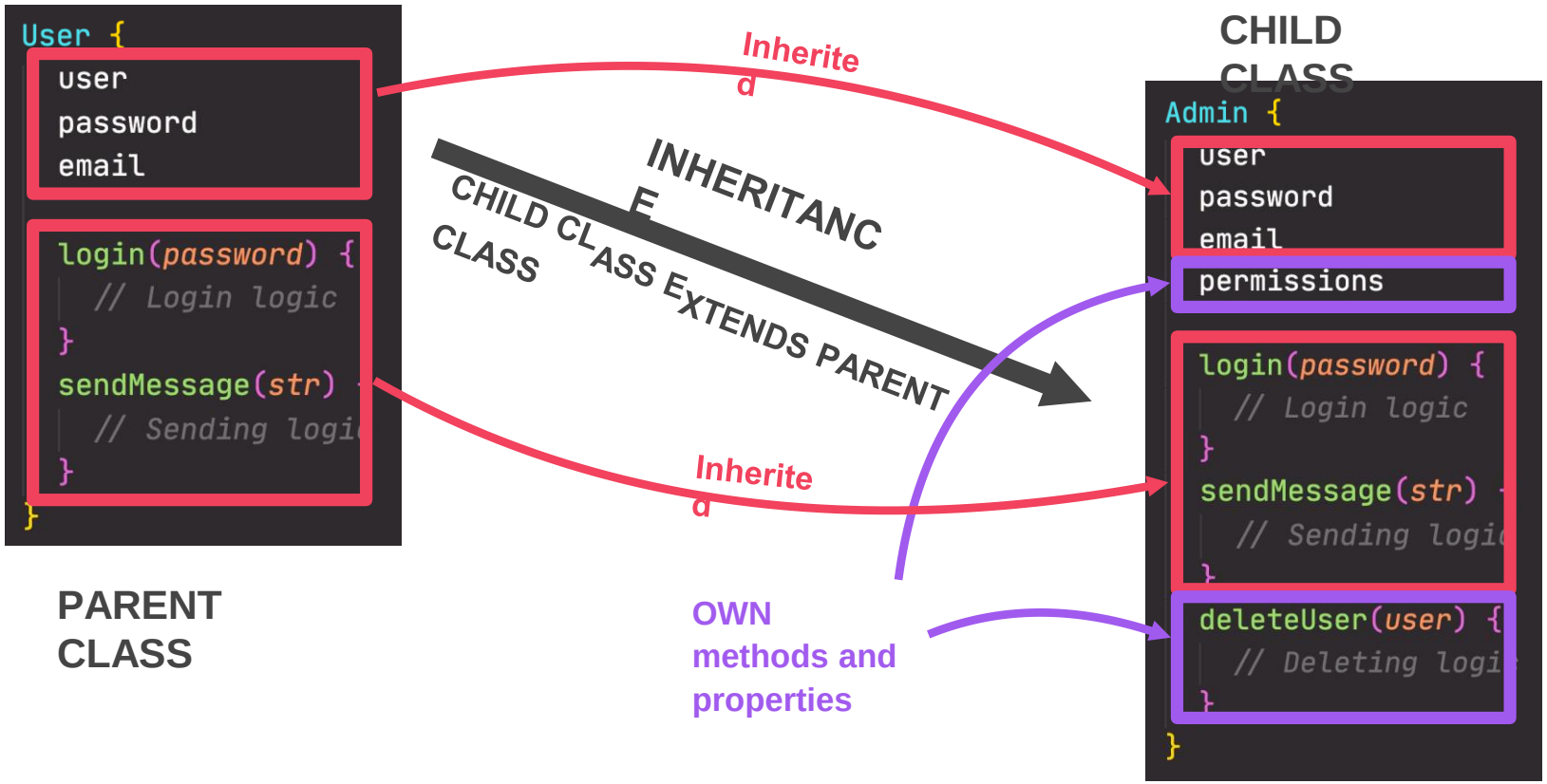
WHY

- 👉 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

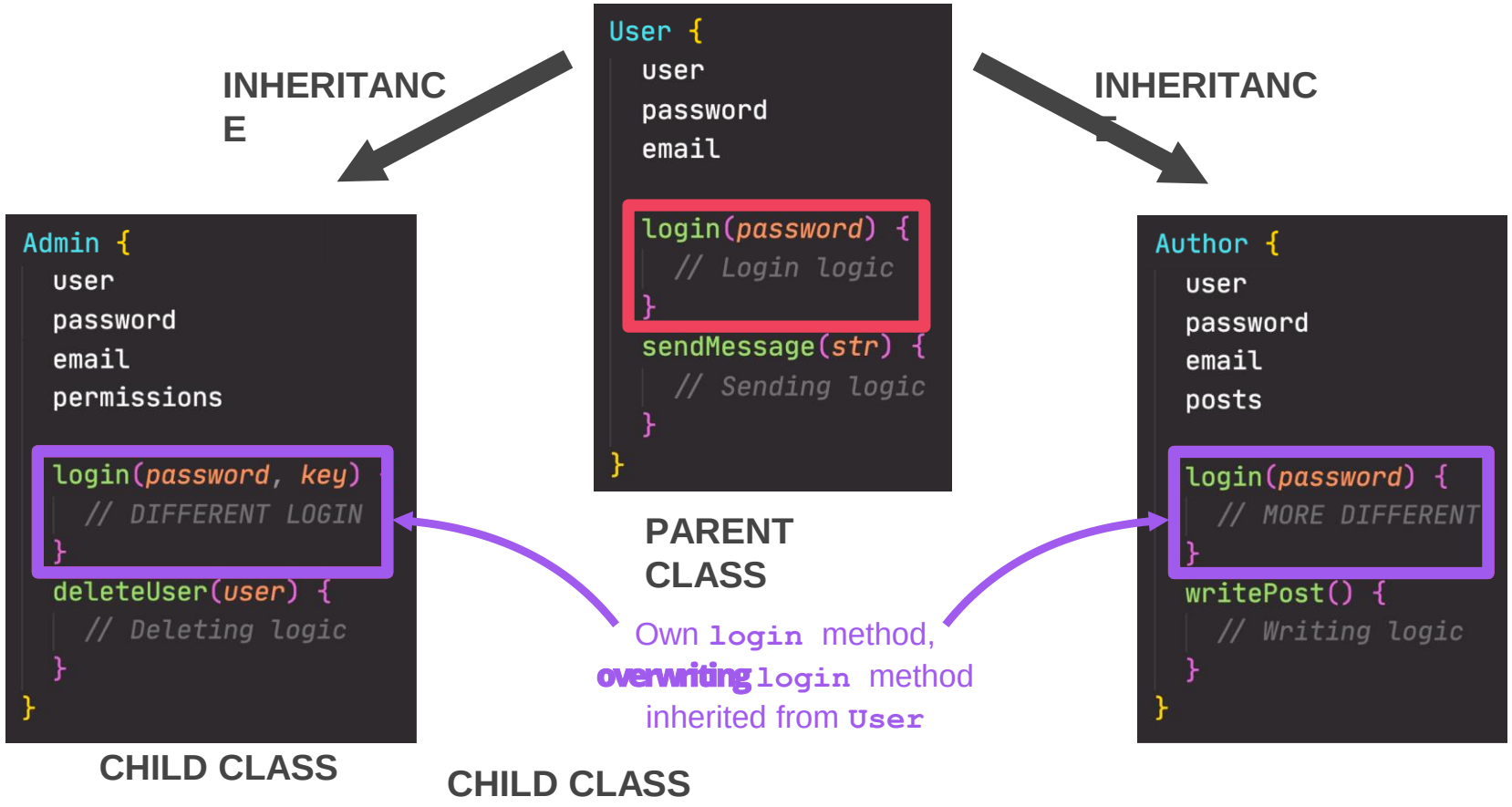
- Abstraction
- Encapsulation
- Inheritance
- Polymorphism



👉 **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 that that, but enough for our purposes].

THE COMPLETE JAVASCRIPT COURSE

FROM ZERO TO EXPERT!

SECTION

OBJECT ORIENTED PROGRAMMING (OOP) WITH JAVASCRIPT

LECTURE

OOP IN JAVASCRIPT

JS

OOP IN JAVASCRIPT: PROTOTYPES

"CLASSICAL OOP": CLASSES

Class

INSTANTIATION

Instance

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

OOP IN JS: PROTOTYPES

Prototype

Contains methods

PROTOTYPAL INHERITANCE/ DELEGATION

Object

Can access methods

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

- **Example: Array**

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

MDN web docs
moz://a

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

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

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

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

3 WAYS OF IMPLEMENTING PROTOTYPAL INHERITANCE IN JS

🤔 *"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.

THE COMPLETE JAVASCRIPT COURSE

FROM ZERO TO EXPERT!

SECTION

OBJECT ORIENTED PROGRAMMING (OOP) WITH JAVASCRIPT

LECTURE

PROTOTYPAL INHERITANCE AND THE PROTOTYPE CHAIN

JS

HOW PROTOTYPAL INHERITANCE / DELEGATION WORKS

Constructor function
`[Person()]`

`.prototype` →
← `.constructor`

Prototype
`[Person.prototype]`
`calcAge: function`

NOT of `Person`, but objects **created by** `Person`

```
const Person = function(name, birthYear) {
  this.name = name;
  this.birthYear = birthYear;
};
```

PROTOTYPE CHAIN

PROTOTYPAL INHERITANCE / DELEGATION

`.__proto__` →

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

jonas.calcAge(); // 47
```

Can't find `calcAge` here!

Object
`[jonas]`

`name: 'Jonas'`

`birthYear: 1990`

`__proto__ : Person.prototype`

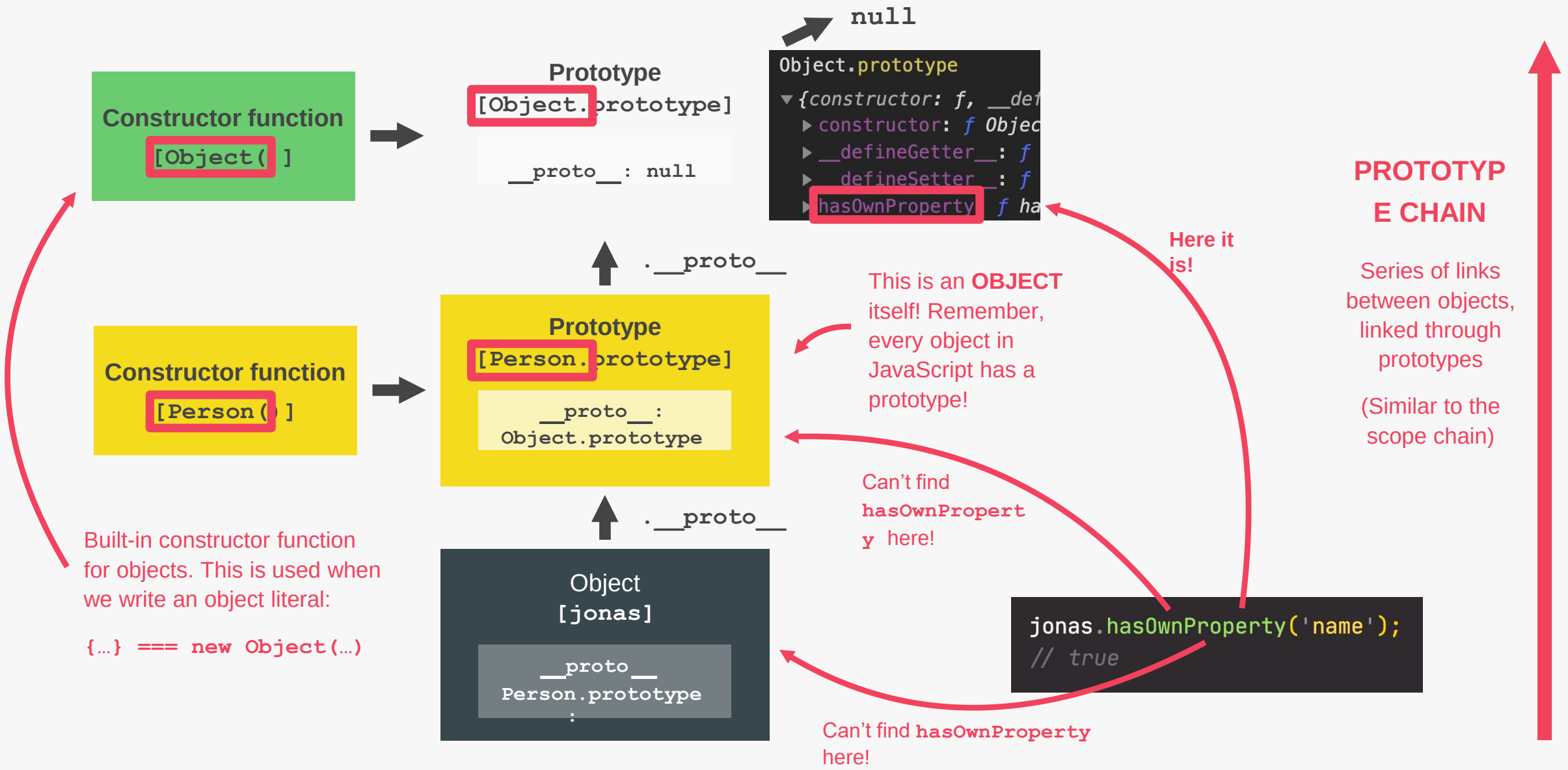
Always points to an object's prototype

☝ This is how it works with **function constructors** and **ES6 classes**

🆕 The new operator:

1. An empty object is created
2. `this` keyword in constructor function call is set to the new object
3. **The new object is linked (`__proto__` property) to the constructor function's prototype property**
4. The new object is returned from the constructor function call

THE PROTOTYPE CHAIN



THE COMPLETE JAVASCRIPT COURSE

FROM ZERO TO EXPERT!

SECTION

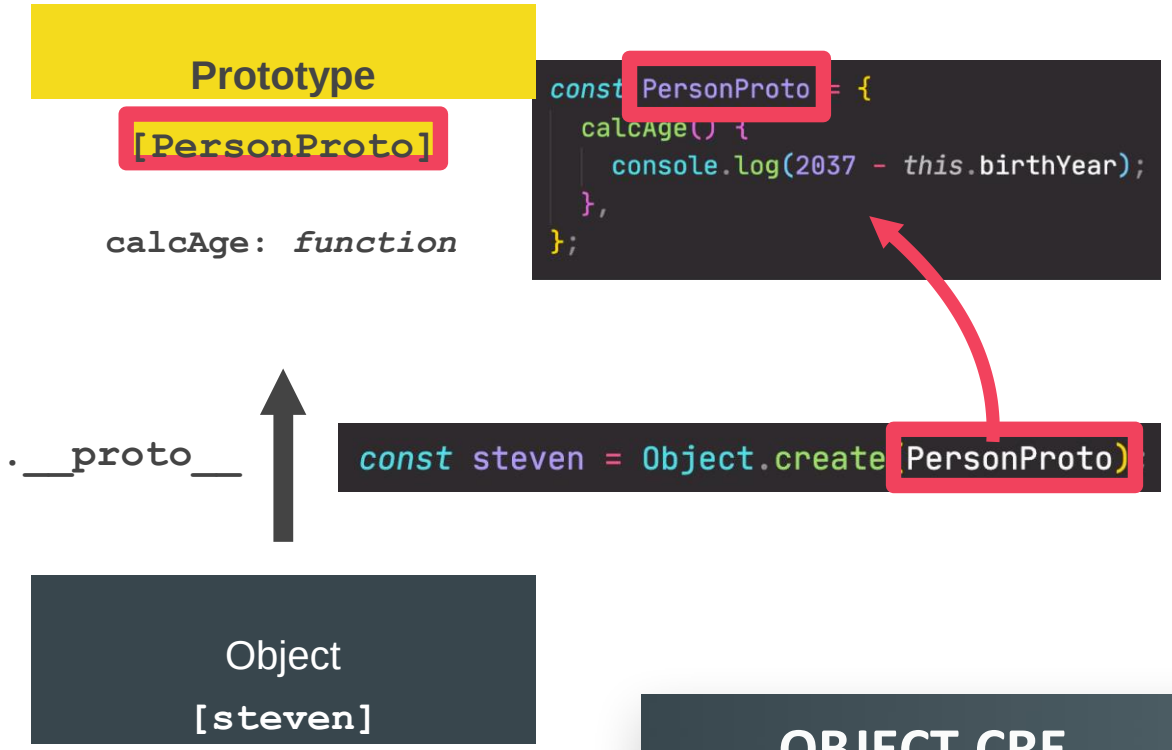
OBJECT ORIENTED PROGRAMMING (OOP) WITH JAVASCRIPT

LECTURE

OBJECT.CREATE

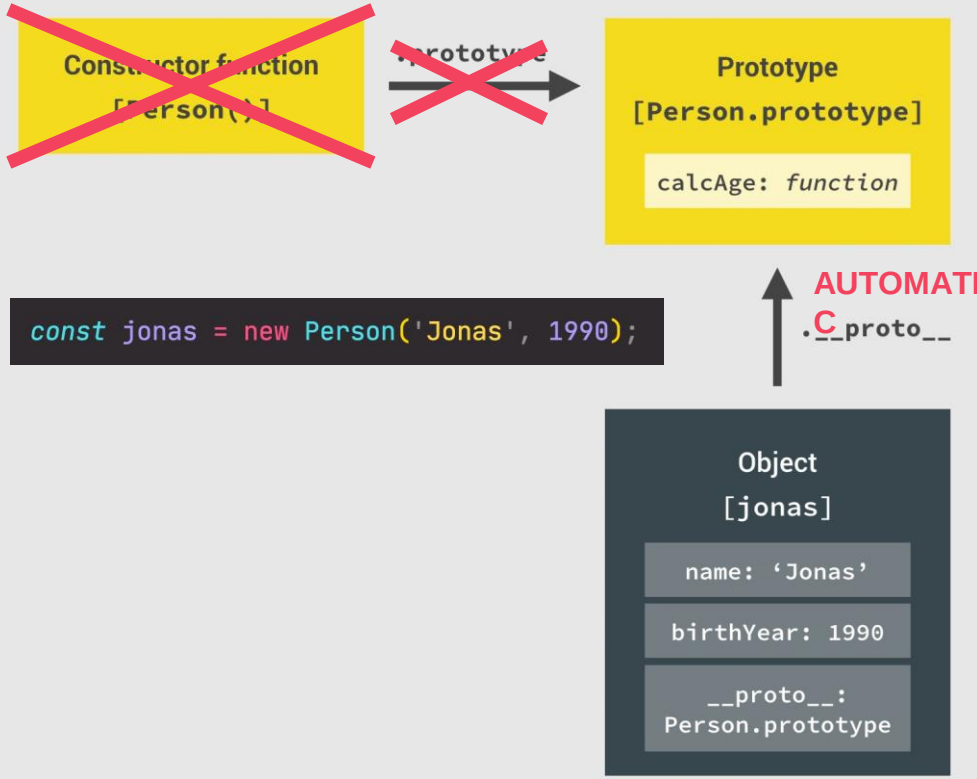
JS

HOW OBJECT CREATE WORKS



OBJECT.CRE
ATE

CONSTRUCTOR FUNCTIONS



THE COMPLETE JAVASCRIPT COURSE

FROM ZERO TO EXPERT!

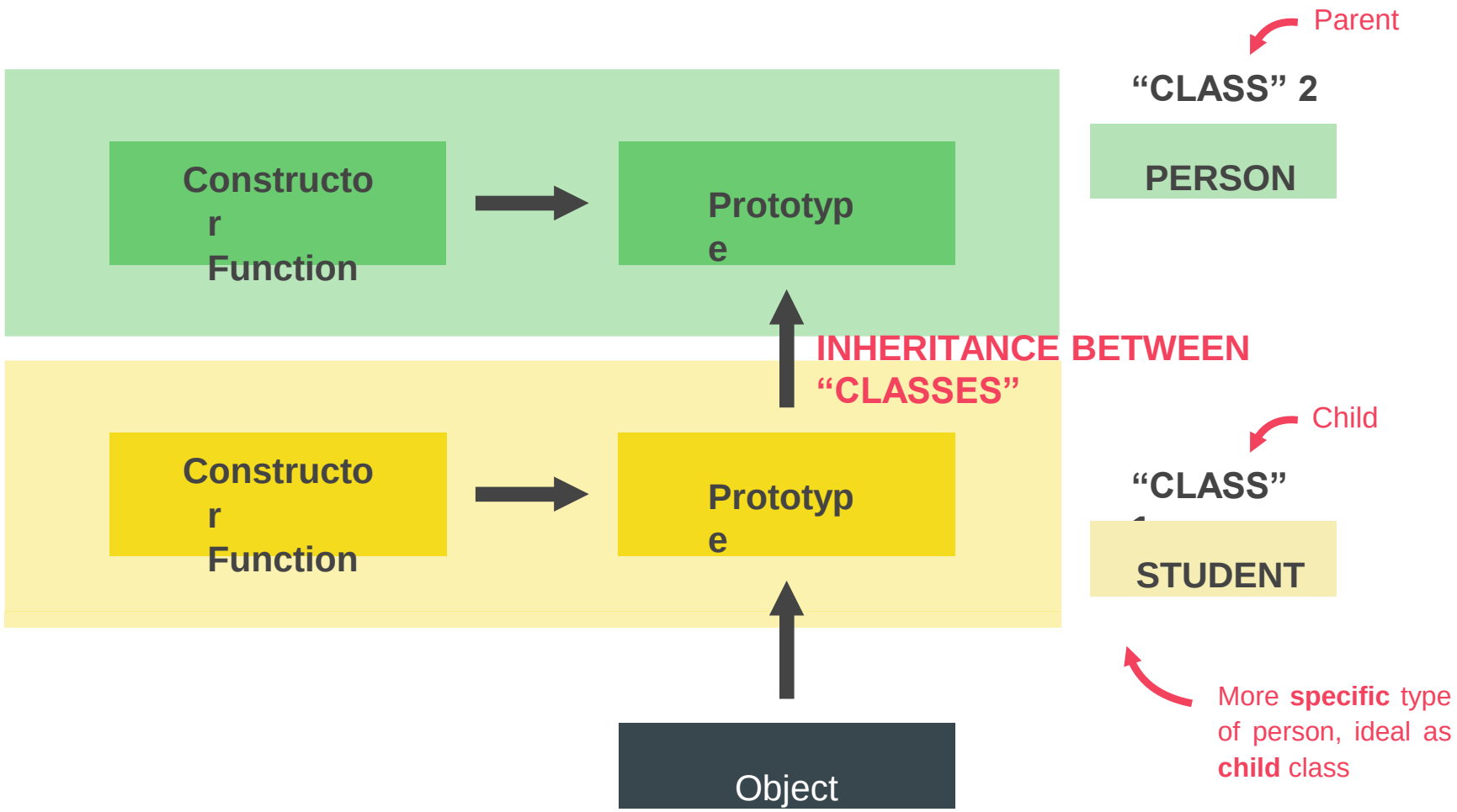
SECTION

OBJECT ORIENTED PROGRAMMING (OOP) WITH JAVASCRIPT

LECTURE

INHERITANCE BETWEEN "CLASSES": CONSTRUCTOR FUNCTIONS

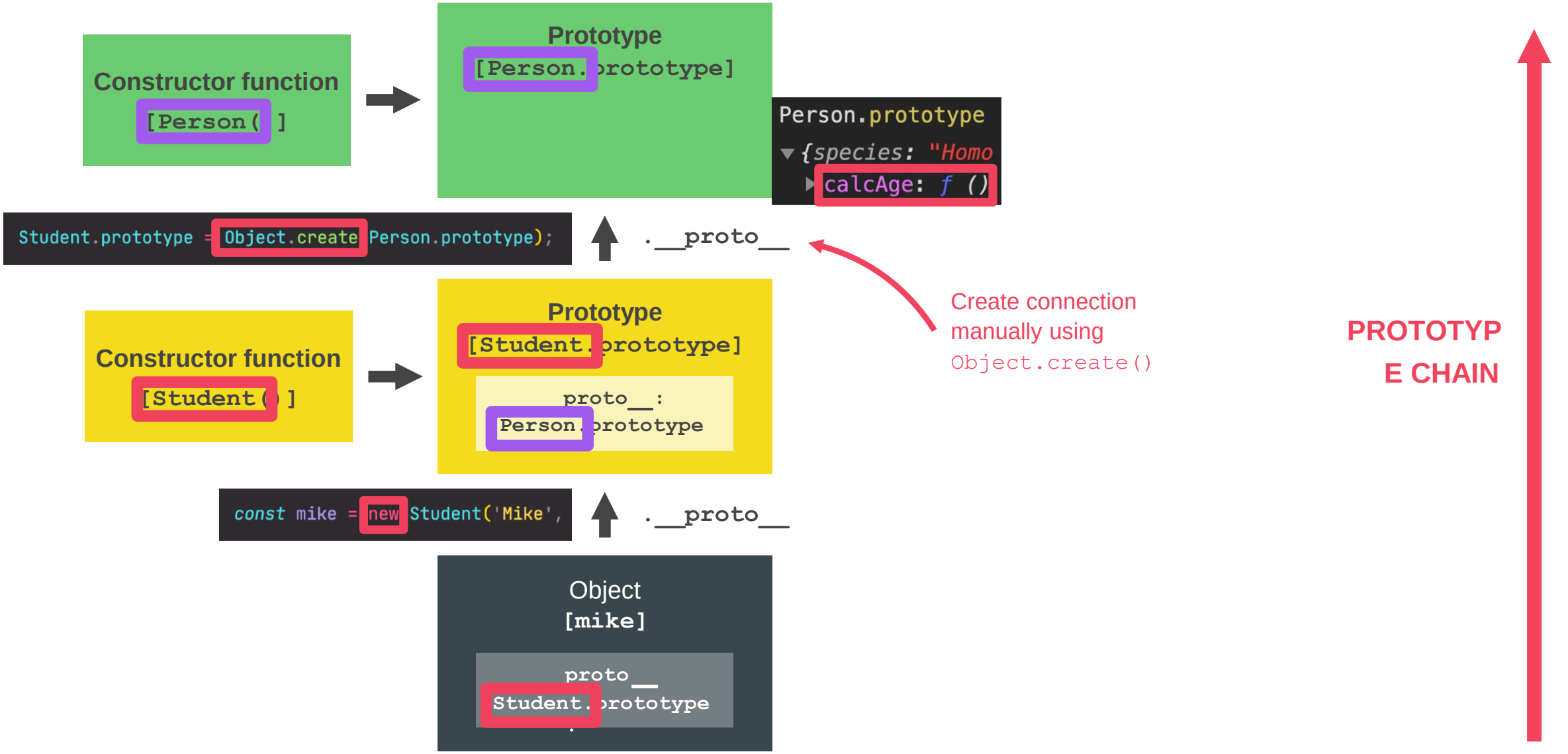
INHERITANCE BETWEEN “CLASSES”



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

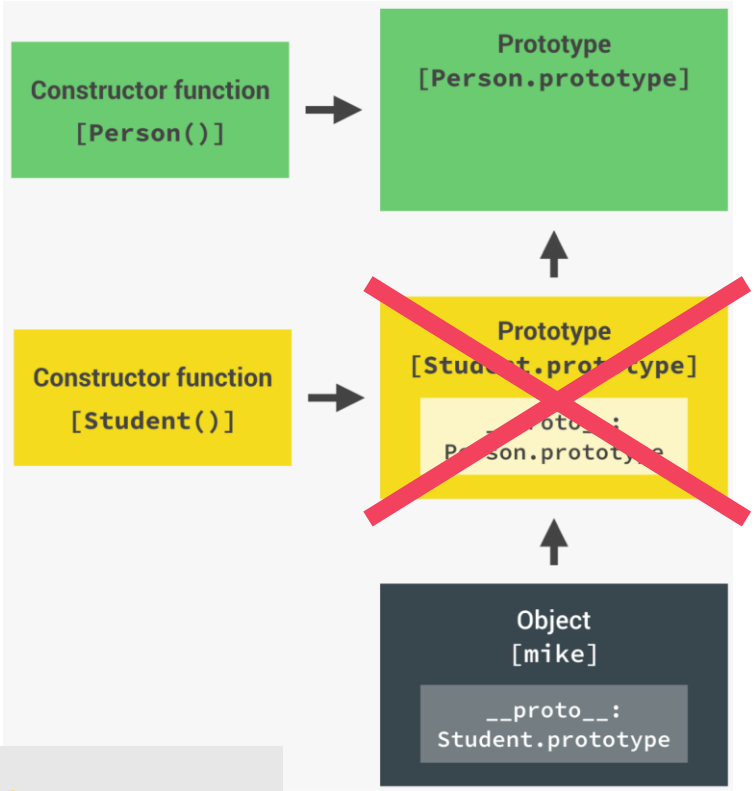
☝ Using class terminology here to make it easier to understand.

INHERITANCE BETWEEN “CLASSES”

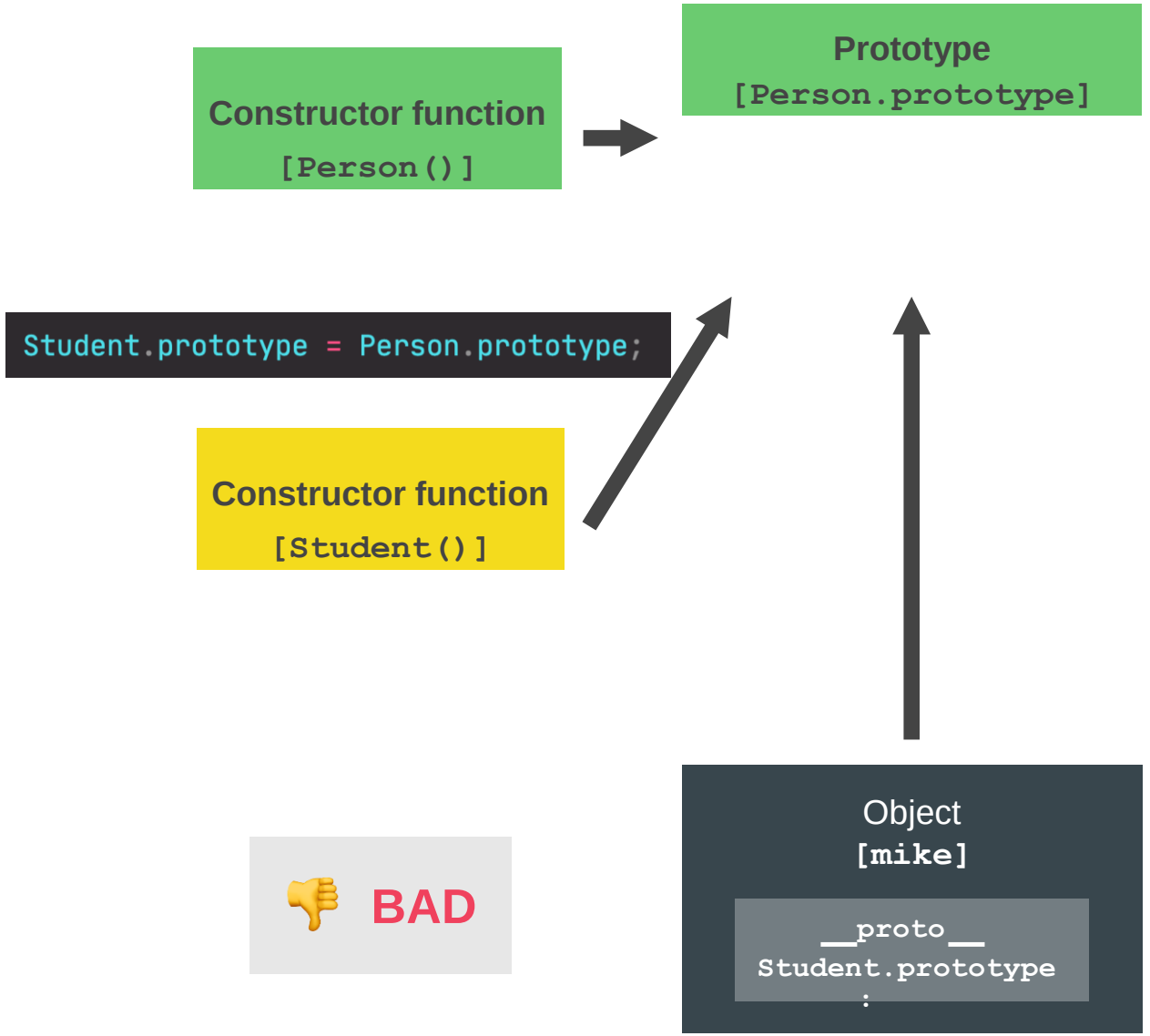


INHERITANCE BETWEEN “CLASSES”

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

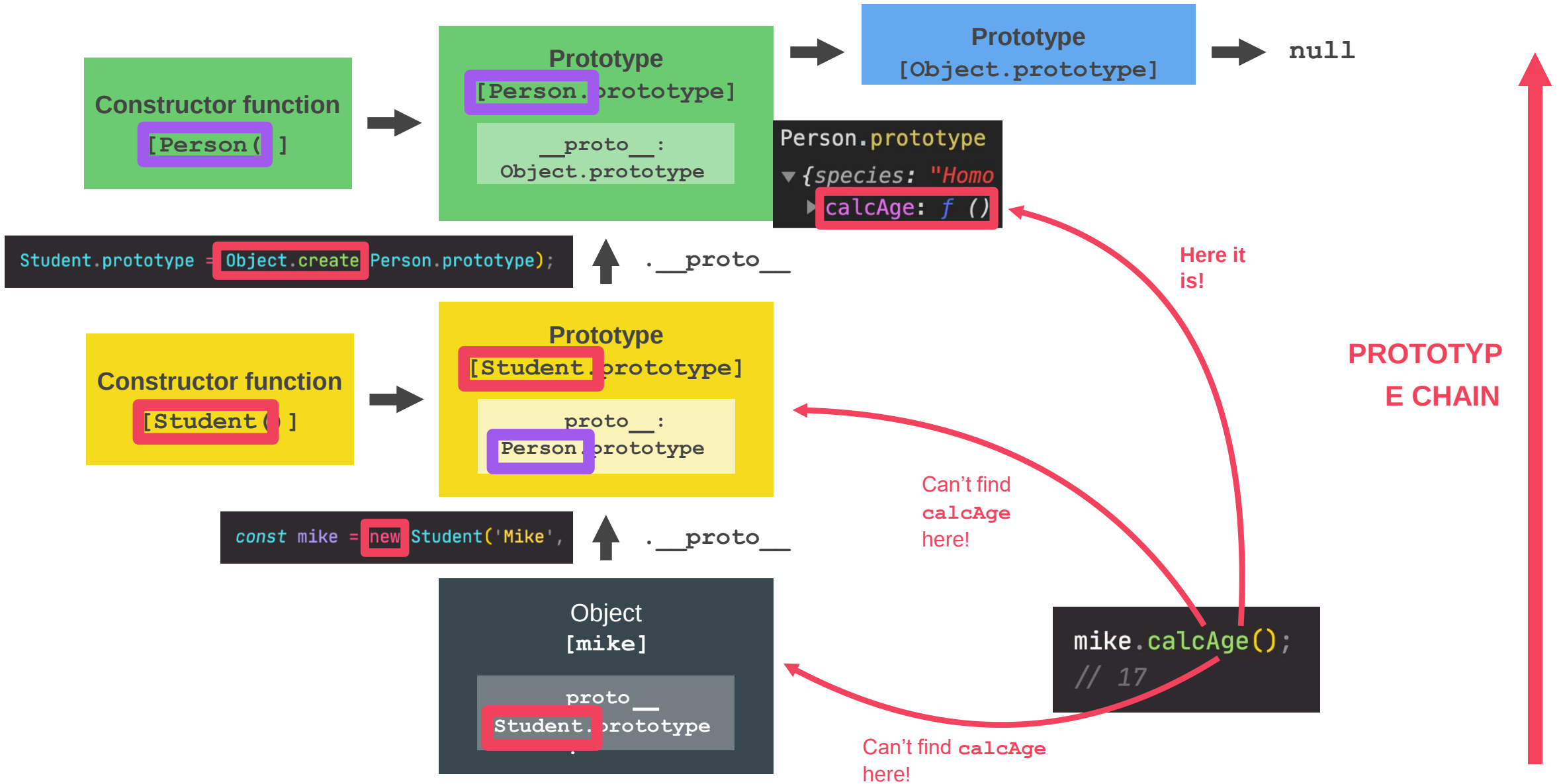


👍 GOOD



👎 BAD

INHERITANCE BETWEEN “CLASSES”



THE COMPLETE JAVASCRIPT COURSE

FROM ZERO TO EXPERT!

SECTION

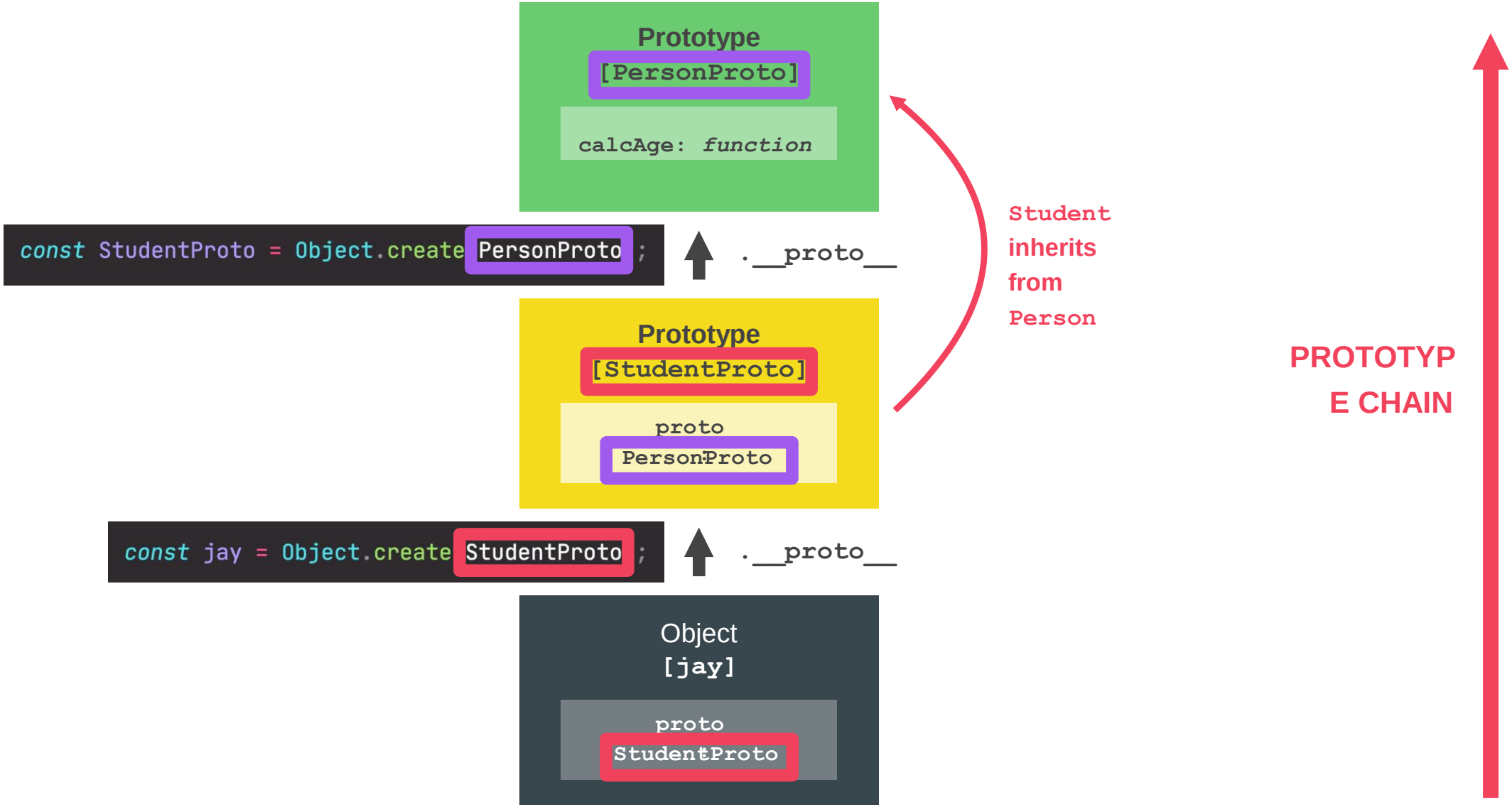
OBJECT ORIENTED PROGRAMMING (OOP) WITH JAVASCRIPT

LECTURE

INHERITANCE BETWEEN "CLASSES": OBJECT.CREATE

JS

INHERITANCE BETWEEN “CLASSES”: OBJECT.CREATE



THE COMPLETE JAVASCRIPT COURSE

FROM ZERO TO EXPERT!

SECTION

OBJECT ORIENTED PROGRAMMING (OOP) WITH JAVASCRIPT

LECTURE

ES6 CLASSES SUMMARY

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**

JS

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

THE COMPLETE JAVASCRIPT COURSE

FROM ZERO TO EXPERT!

SECTION

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

LECTURE

HOW TO PLAN A WEB PROJECT

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

1. USER STORIES

mapty

👉 **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? **What?** **Why?**

Example: user, admin, etc.

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

mapty

USER STORY	FEATURES
Map where user clicks to add new workout (best way to get location coordinates)	👉
1 Log my running workouts with location, distance, time, pace and steps/minute	👉
Geolocation to display map at current location (more user friendly)	👉
2 Log my cycling workouts with location, distance, time, speed and elevation gain	👉 Form to input distance, time, speed, elevation gain
Form to input distance, time, pace, steps/minute	
3 See all my workouts at a glance	👉 Display all workouts in a list
4 See my workouts on a map	👉 Display all workouts on the map
5 See all my workouts when I leave the app and come back later	👉 Store workout data in the browser using local storage API 👉 On page load, read the saved data from local storage and display

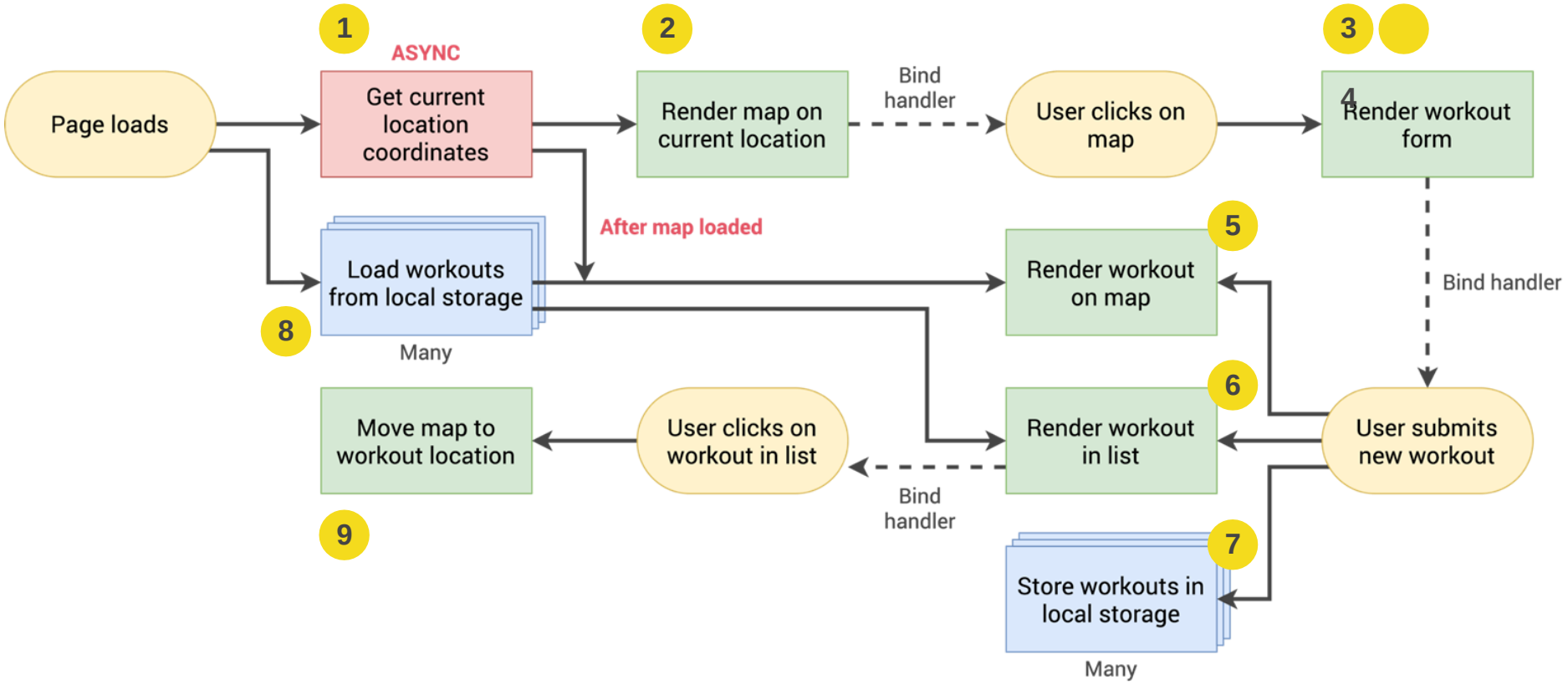
3. FLOWCHART

mapty

FEATURES

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

Added later



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

FOR NOW, LET'S
JUST START 😍
CODING

THE COMPLETE JAVASCRIPT COURSE

FROM ZERO TO EXPERT!

SECTION

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

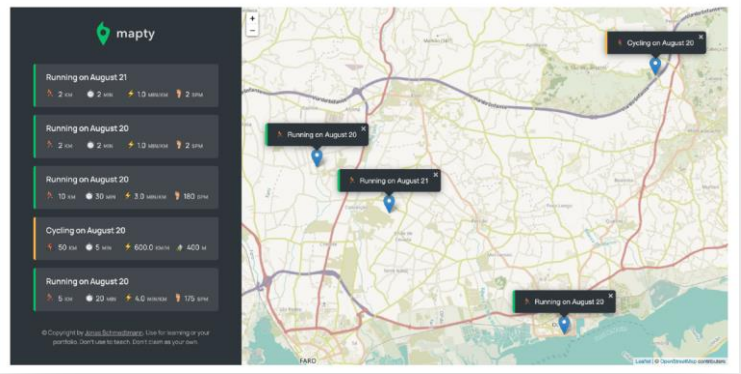
LECTURE

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];



JS

ASYNCHRONOUS JAVASCRIPT: PROMISES, ASYNC/AWAIT AND AJAX

THE COMPLETE JAVASCRIPT COURSE

FROM ZERO TO EXPERT!

SECTION

ASYNCHRONOUS JAVASCRIPT: PROMISES, ASYNC/AWAIT AND AJAX

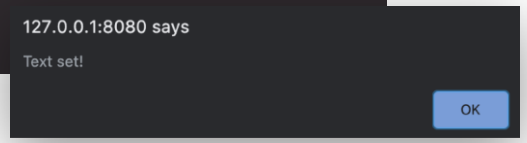
LECTURE

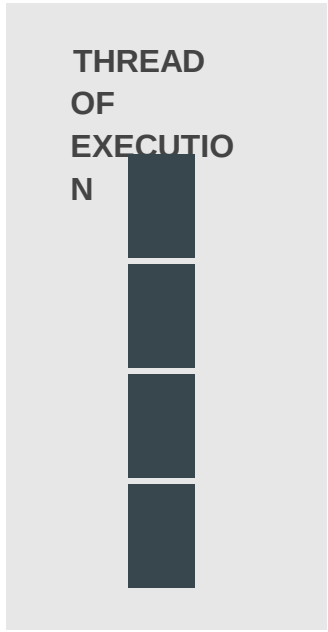
ASYNCHRONOUS JAVASCRIPT, AJAX AND APIS

JS

BLOCKING

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





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

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.

JS

Asynchrono
us

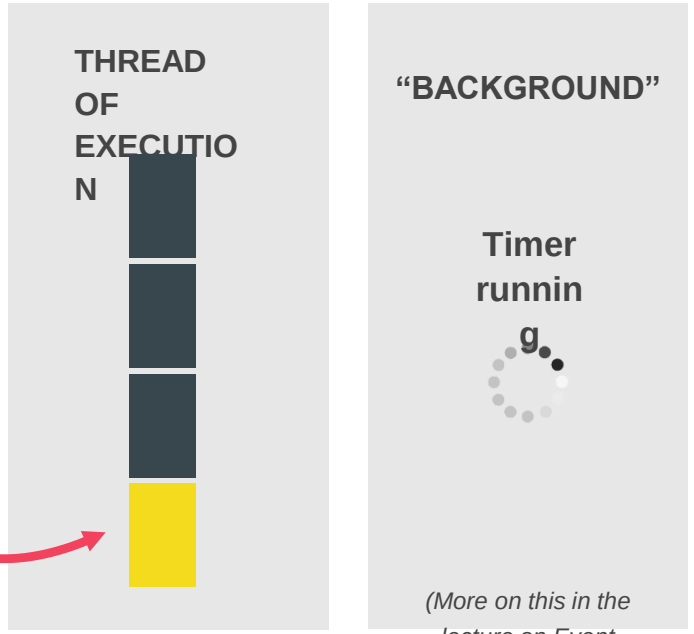
```
const p = document.querySelector('.p');
setTimeout(function () {
  p.textContent = 'My name is Jonas!';
}, 5000);
p.style.color = 'red';
```

CALLBACK WILL RUN AFTER TIMER

👉 **Example:** Timer with callback

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

Callback does NOT automatically make code asynchronous!



Executed after all other code

(More on this in the lecture on Event Loop)

ASYNCHRON OUS

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

Asynchrono
us

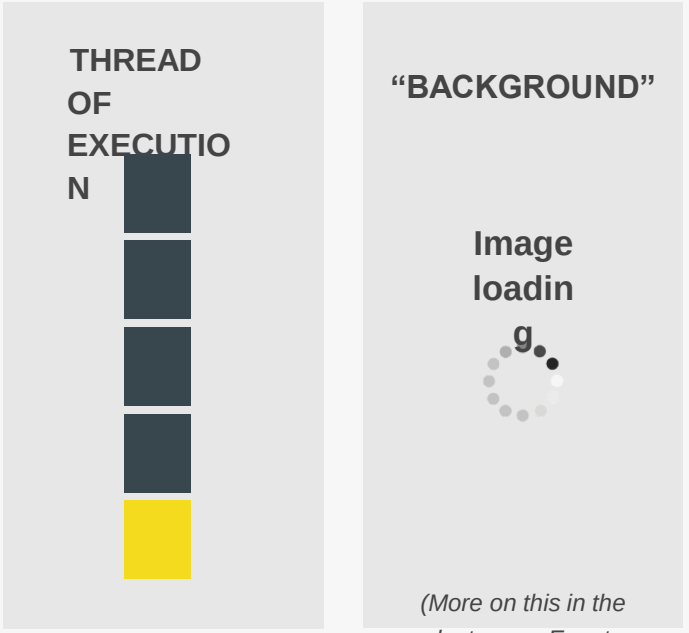
```
const img = document.querySelector('.dog');
img.src = 'dog.jpg';
img.addEventListener('load', function () {
  img.classList.add('fadeIn');
});
p.style.width = '300px';
```

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!



(More on this in the lecture on Event Loop)

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;

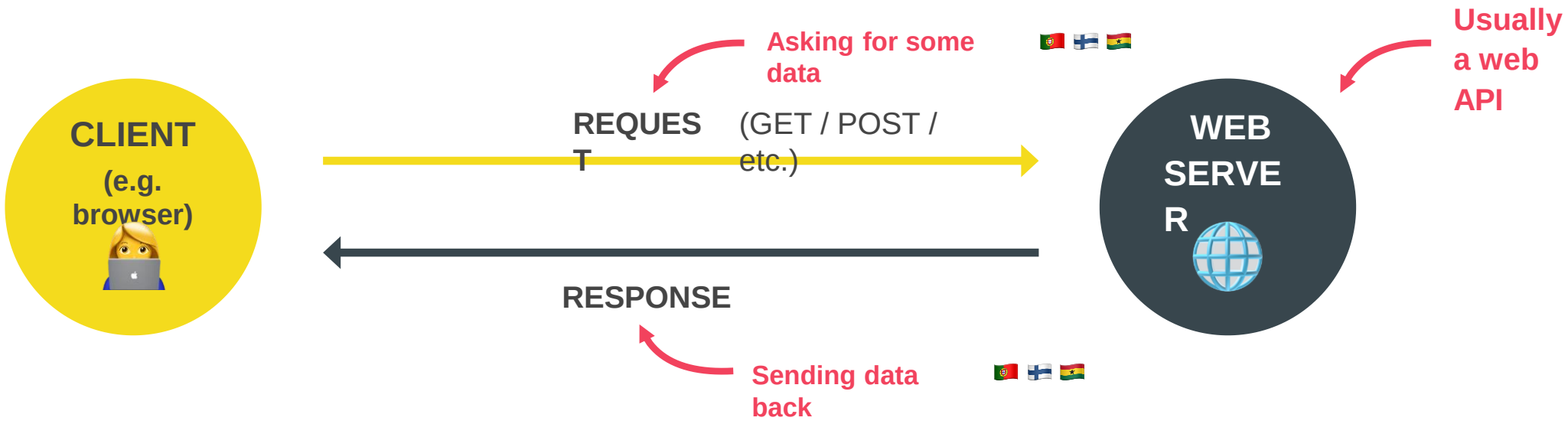
👉 Callback functions alone do **NOT** make code asynchronous!

WHAT ARE AJAX CALLS?

JS

AJAX

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



WHAT IS AN API?

JS

API

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

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

Just "API"

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

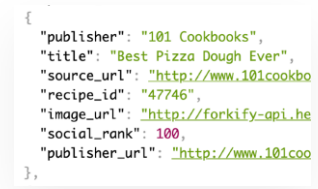
AJA

~~XML~~

~~XML data format~~

JSON data format

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...*

✈️ 🌍

THE COMPLETE JAVASCRIPT COURSE

FROM ZERO TO EXPERT!

SECTION

ASYNCHRONOUS JAVASCRIPT: PROMISES, ASYNC/AWAIT AND AJAX

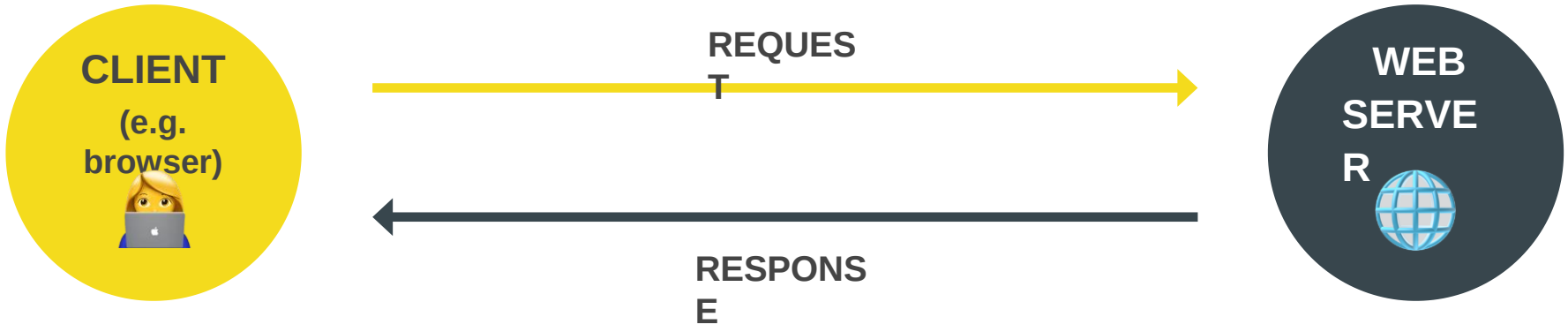
LECTURE

HOW THE WEB WORKS: REQUESTS AND RESPONSES

WHAT HAPPENS WHEN WE ACCESS A WEB SERVER

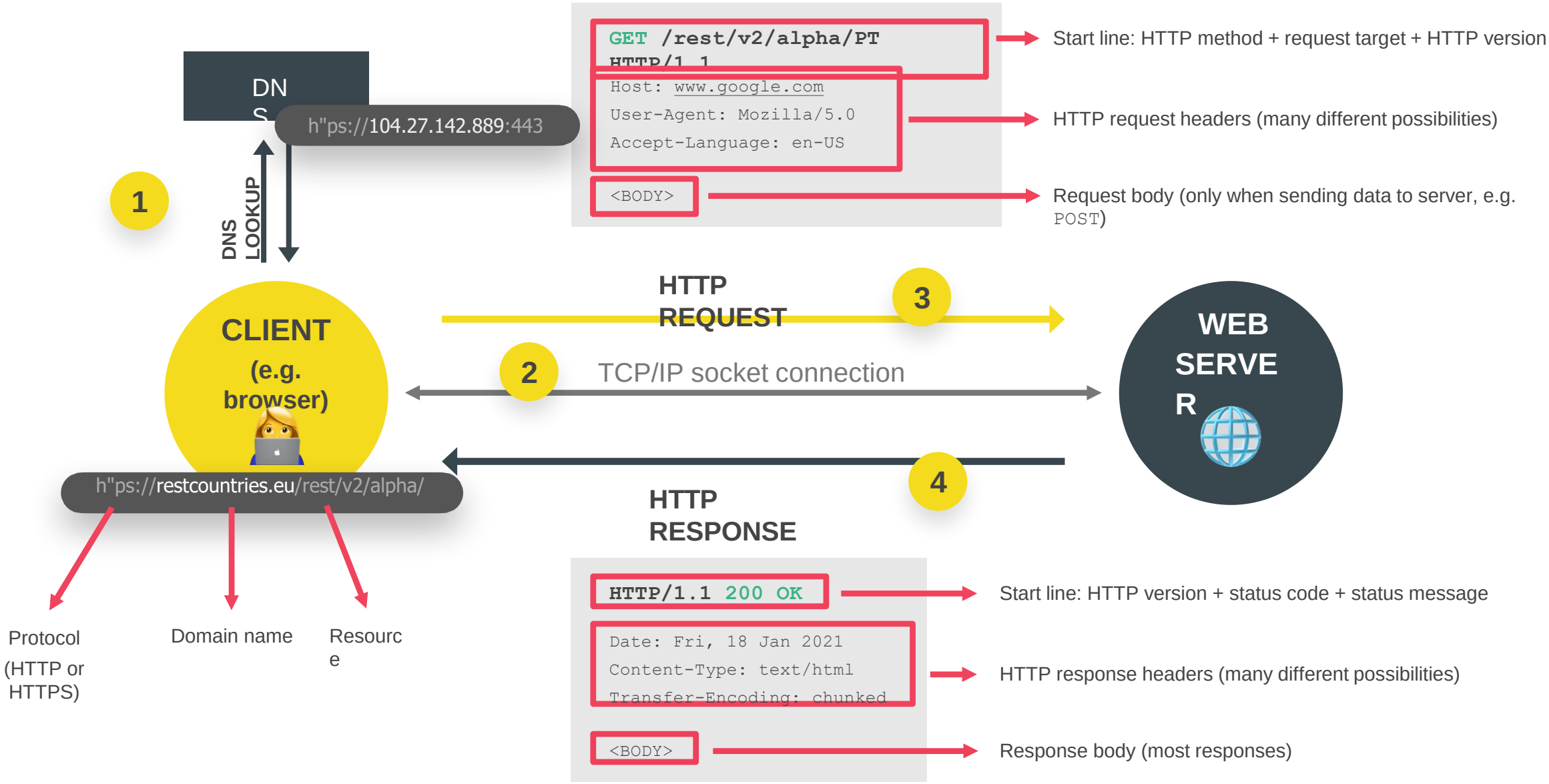
JS

👉 **Request-response model** or **Client-server architecture**



WHAT HAPPENS WHEN WE ACCESS A WEB SERVER

JS



THE COMPLETE
JAVASCRIPT COURSE

FROM ZERO TO EXPERT!

SECTION

ASYNCHRONOUS JAVASCRIPT:
PROMISES, ASYNC/AWAIT AND AJAX

LECTURE

PROMISES AND THE FETCH API

JS

WHAT ARE PROMISES?

PROMISE

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

↓ Less formal

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

↓ Less formal

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

Example: Response from AJAX call

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

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

Promise that I will receive money if I guess correct outcome

🤞 I buy lottery ticket (promise) right now

↓

🔮 Lottery draw happens asynchronously

↓

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

THE PROMISE LIFECYCLE

Before the future value is available

Asynchronous task **has finished**

PENDING

ASYNC TASK

SETTLED

FULFILLED

REJECTED

Success! The value is now **available**

An **error** happened

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

E.g. Fetch API returns promise

BUILD PROMISE

CONSUME PROMISE

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

THE COMPLETE JAVASCRIPT COURSE

FROM ZERO TO EXPERT!

SECTION

ASYNCHRONOUS JAVASCRIPT: PROMISES, ASYNC/AWAIT AND AJAX

LECTURE

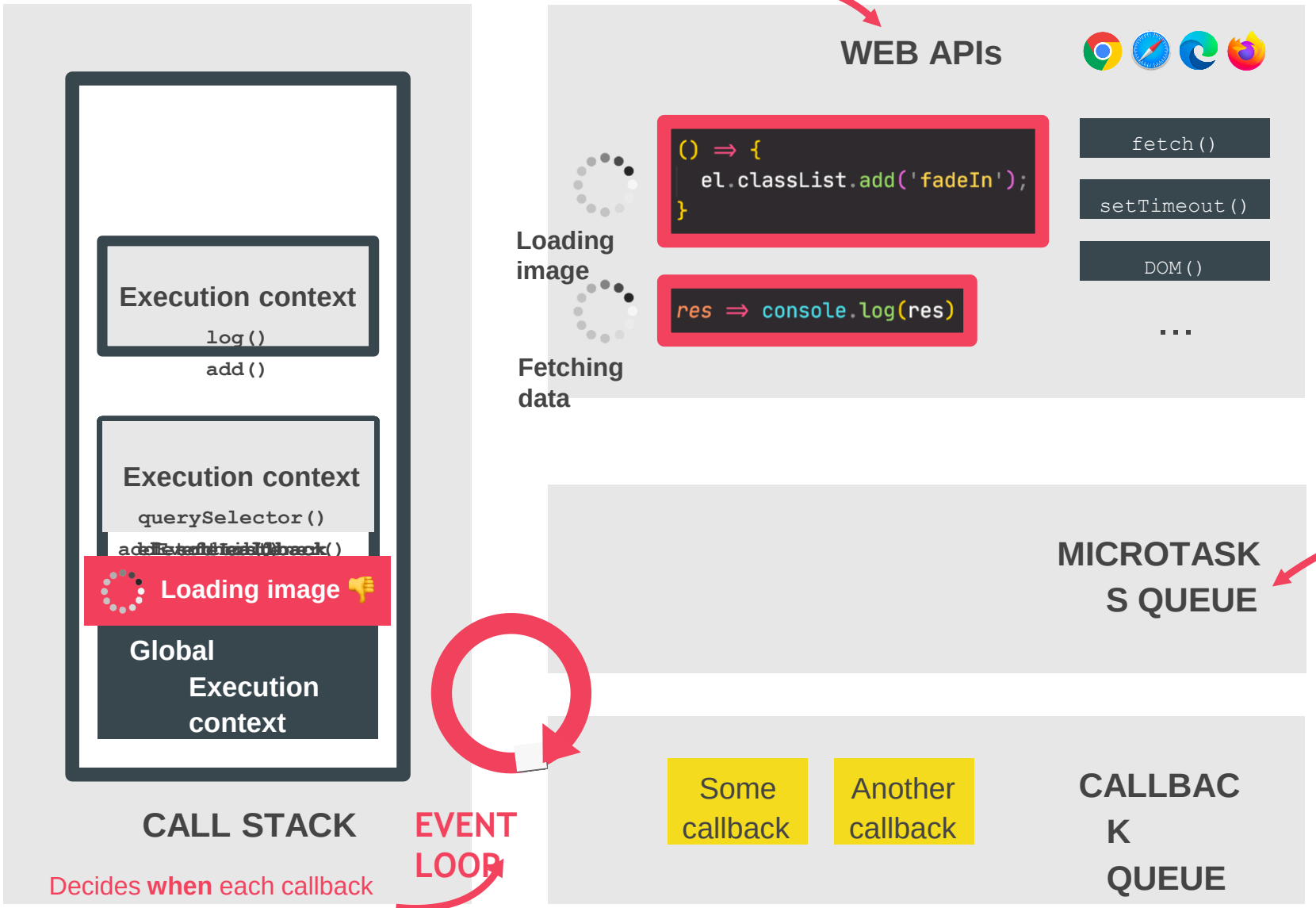
ASYNCHRONOUS BEHIND THE SCENES: THE EVENT LOOP

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




```
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?

JS

MODERN JAVASCRIPT DEVELOPMENT: MODULES AND TOOLING

THE COMPLETE JAVASCRIPT COURSE

FROM ZERO TO EXPERT!

SECTION

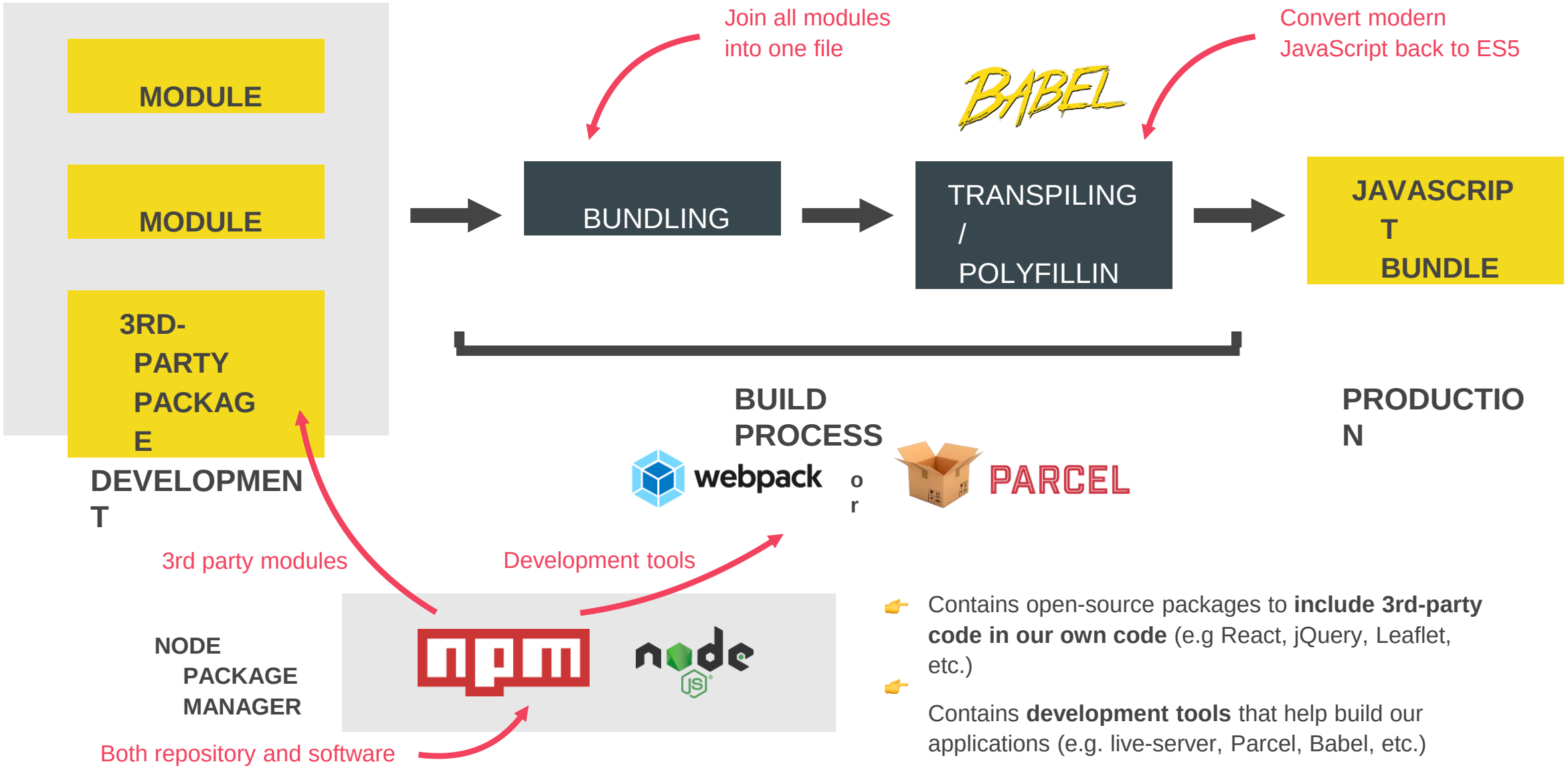
MODERN JAVASCRIPT DEVELOPMENT: MODULES AND TOOLING

LECTURE

AN OVERVIEW OF MODERN JAVASCRIPT DEVELOPMENT

JS

MODERN JAVASCRIPT DEVELOPMENT



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

THE COMPLETE JAVASCRIPT COURSE

FROM ZERO TO EXPERT!

SECTION

MODERN JAVASCRIPT DEVELOPMENT: MODULES AND TOOLING

LECTURE

AN OVERVIEW OF MODULES IN JAVASCRIPT

JS

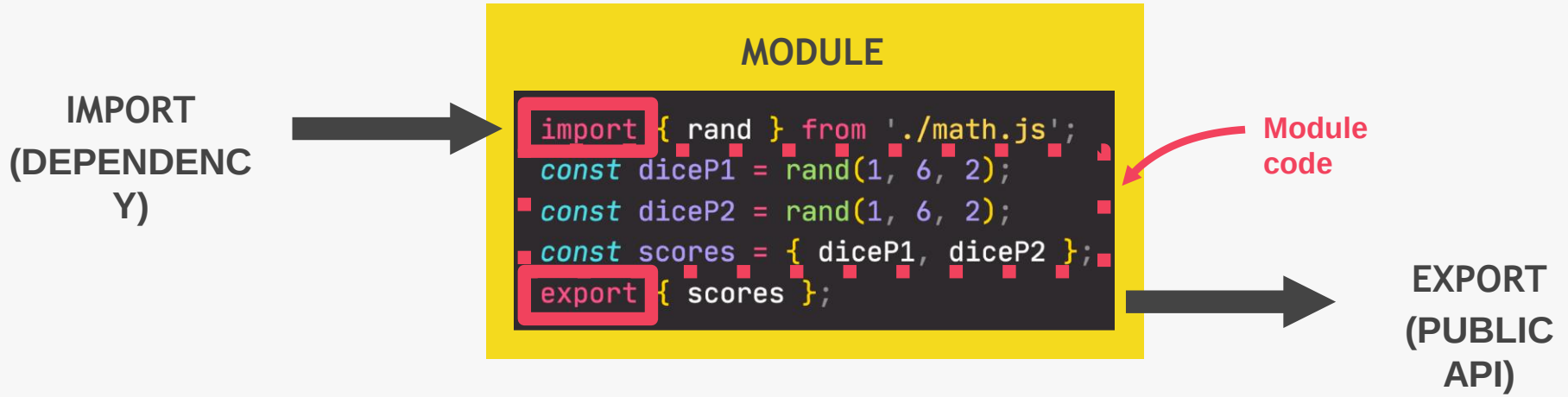
AN OVERVIEW OF MODULES

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.



NATIVE JAVASCRIPT (ES6) MODULES

ES6 MODULES

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

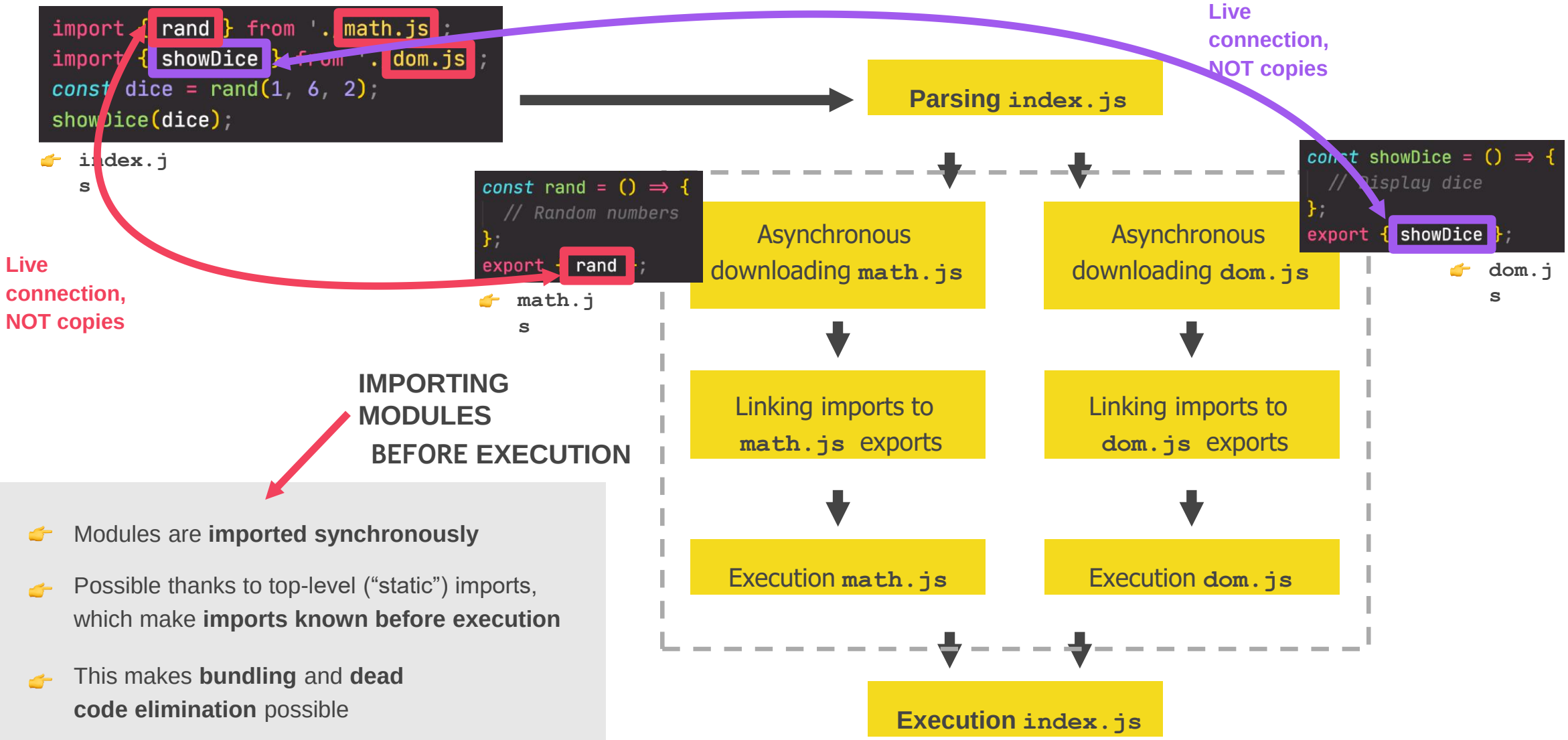
```
import { rand } from './math.js';
const diceP1 = rand(1, 6, 2);
const diceP2 = rand(1, 6, 2);
const scores = { diceP1, diceP2 };
export { scores };
```

`import` and `export` syntax

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

☝️ **Need to happen at top-level Imports are hoisted!**

HOW ES6 MODULES ARE IMPORTED



IMPORTING MODULES BEFORE EXECUTION

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

THE COMPLETE JAVASCRIPT COURSE

FROM ZERO TO EXPERT!

SECTION

MODERN JAVASCRIPT DEVELOPMENT: MODULES AND TOOLING

LECTURE

REVIEW: WRITING CLEAN AND MODERN JAVASCRIPT

JS

REVIEW: MODERN AND CLEAN CODE

READABLE CODE

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

GENERAL

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

FUNCTIONS

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

OOP

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

REVIEW: MODERN AND CLEAN CODE

AVOID NESTED CODE

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

ASYNCHRONOUS CODE

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

THE COMPLETE JAVASCRIPT COURSE

FROM ZERO TO EXPERT!

SECTION

MODERN JAVASCRIPT DEVELOPMENT: MODULES AND TOOLING

LECTURE

DECLARATIVE AND FUNCTIONAL JAVASCRIPT PRINCIPLES

IMPERATIVE VS. DECLARATIVE CODE

Two fundamentally different ways of writing code (paradigms)

IMPERATIVE

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

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

DECLARATIVE

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

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

FUNCTIONAL PROGRAMMING PRINCIPLES

Declarative programming paradigm

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

Side effect: Modification (mutation) of any data **outside** of the function (mutating external variables, logging to console, writing to DOM, etc.)

Pure function: Function without side effects. Does not depend on external variables. **Given the same inputs, always returns the same outputs**.

Immutability: State (data) is **never** modified! Instead, state is **copied** and the copy is mutated and returned.

Examples:

FUNCTIONAL PROGRAMMING TECHNIQUES

Avoid 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

JS

FORKIFY APP: BUILDING A MODERN APPLICATION

THE COMPLETE JAVASCRIPT COURSE

FROM ZERO TO EXPERT!

SECTION

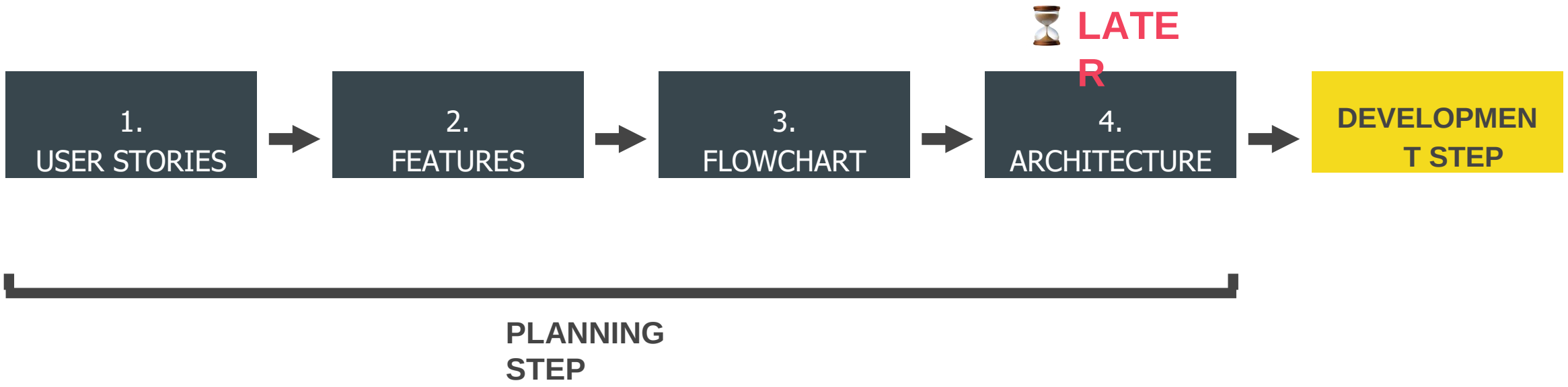
FORKIFY APP: BUILDING A MODERN APPLICATION

LECTURE

PROJECT OVERVIEW AND PLANNING

JS

PROJECT PLANNING



1. USER STORIES

forkify

- 👉 **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

forkify

USER STORIES	FEATURES
Search functionality: input field to send request to API with searched keywords Display results with pagination	👉
1 Search for recipes	👉
Display recipe with cooking time, servings and ingredients	👉
2 Update the number of servings	👉 Change servings functionality: update all ingredients according to current number of servings
3 Bookmark recipes	👉 Bookmarking functionality: display list of all bookmarked recipes
4 Create my own recipes	👉 User can upload own recipes 👉 User recipes will automatically be bookmarked 👉 User can only see their own recipes, not recipes from other users
5 See my bookmarks and own recipes when I leave the app and come back later	👉 Store bookmark data in the browser using local storage 👉 On page load, read saved bookmarks from local storage and display

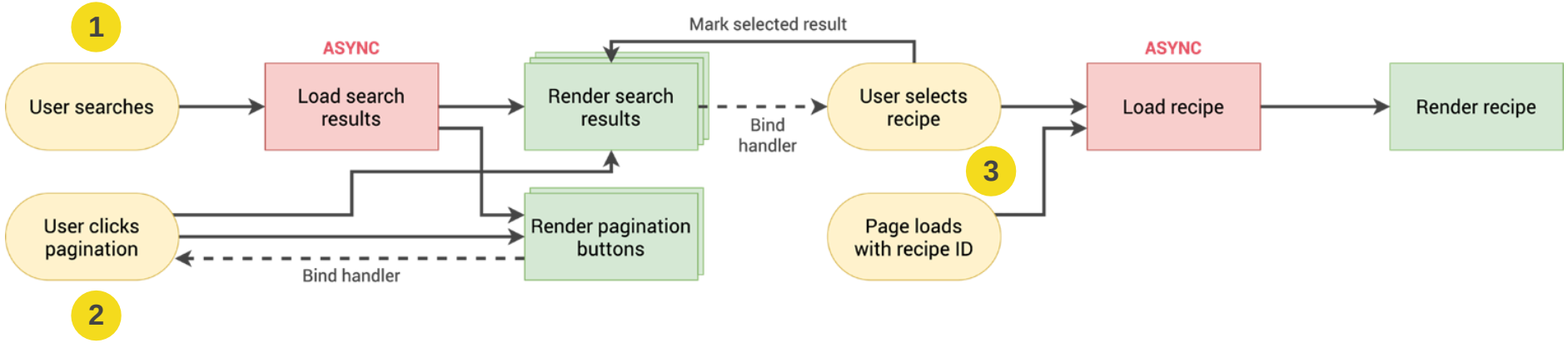
3. FLOWCHART (PART 1)

forkify

FEATURES

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

Other features later



THE COMPLETE JAVASCRIPT COURSE

FROM ZERO TO EXPERT!

SECTION

FORKIFY APP: BUILDING A MODERN APPLICATION

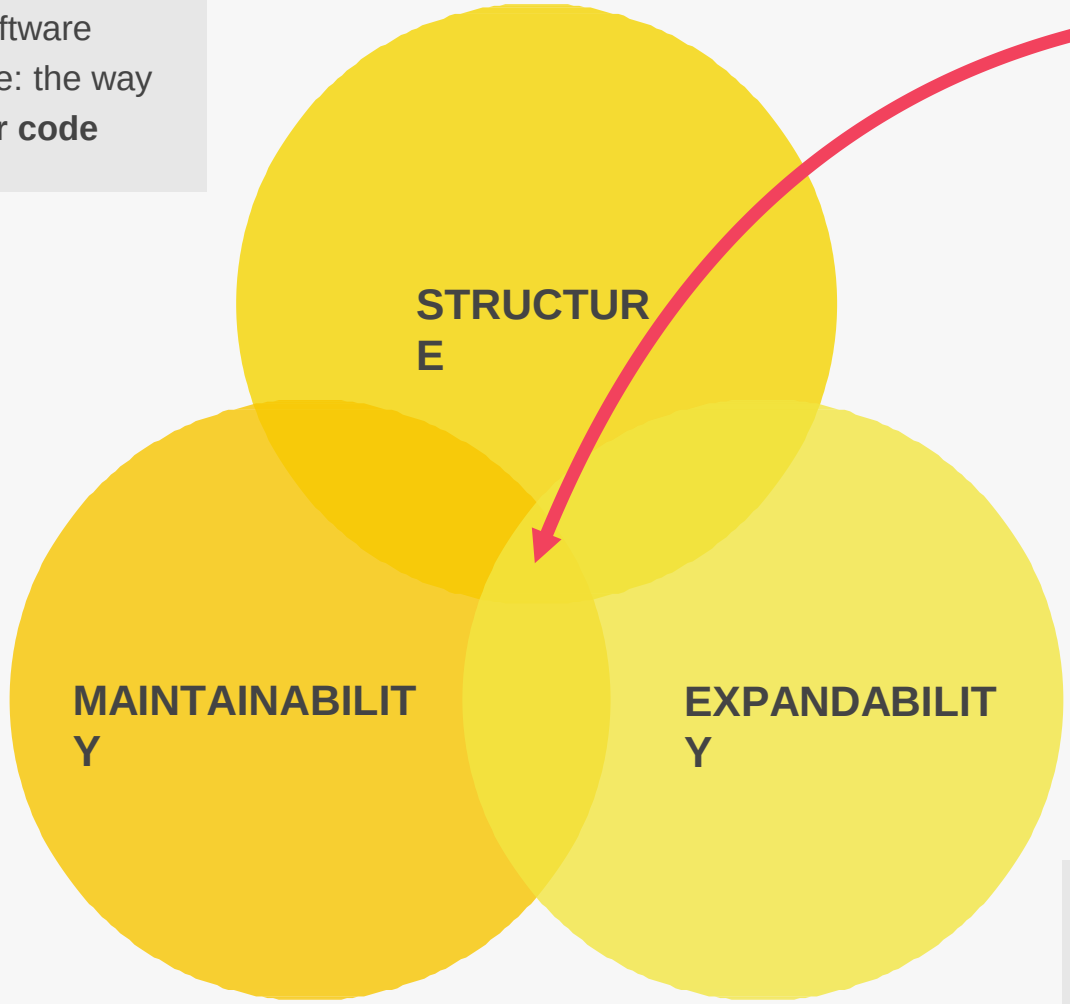
LECTURE

THE MVC ARCHITECTURE

JS

WHY WORRY ABOUT ARCHITECTURE?

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



The perfect architecture

👉 We can create our own architecture (Mapty project)

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

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

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

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

COMPONENTS OF ANY ARCHITECTURE

BUSINESS LOGIC

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

STATE

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

HTTP LIBRARY

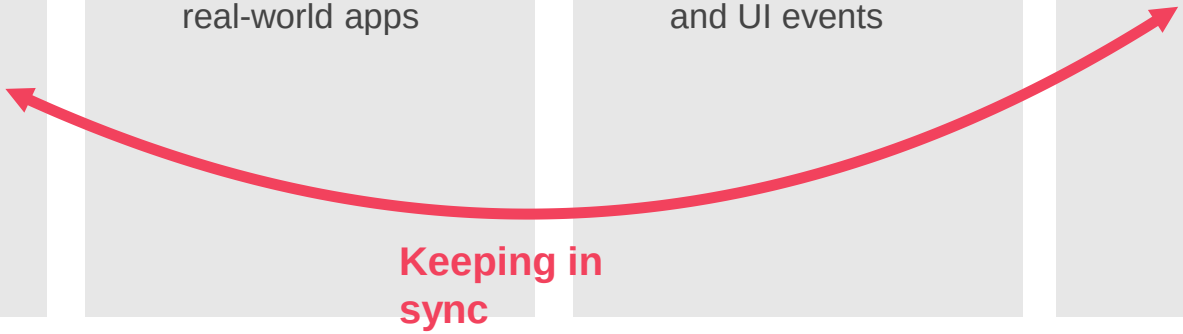
- Responsible for making and receiving AJAX requests
- Optional but almost always necessary in real-world apps

APPLICATION LOGIC (ROUTER)

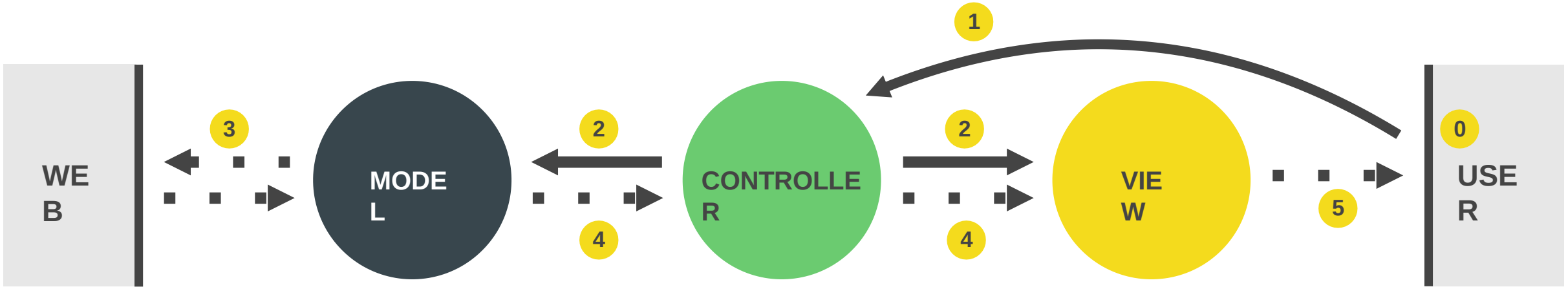
- Code that is only concerned about the **implementation of application itself**;
- Handles navigation and UI events

PRESENTATION LOGIC (UI LAYER)

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



THE MODEL-VIEW-CONTROLLER (MVC) ARCHITECTURE



BUSINESS LOGIC

STATE

HTTP LIBRARY

APPLICATION LOGIC

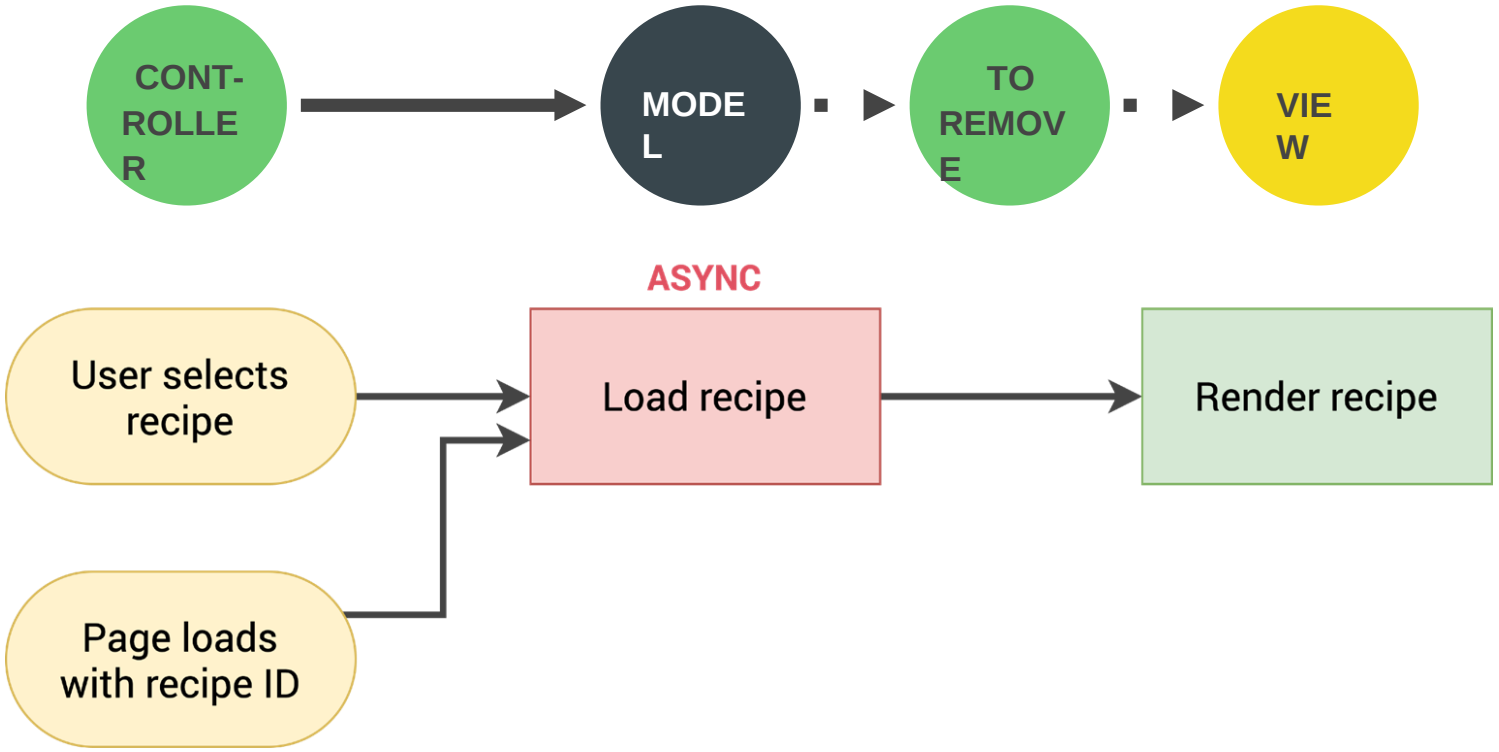
- 👉 Bridge between model and views (which don't know about one another)
- 👉 Handles UI events and **dispatches tasks to model and view**

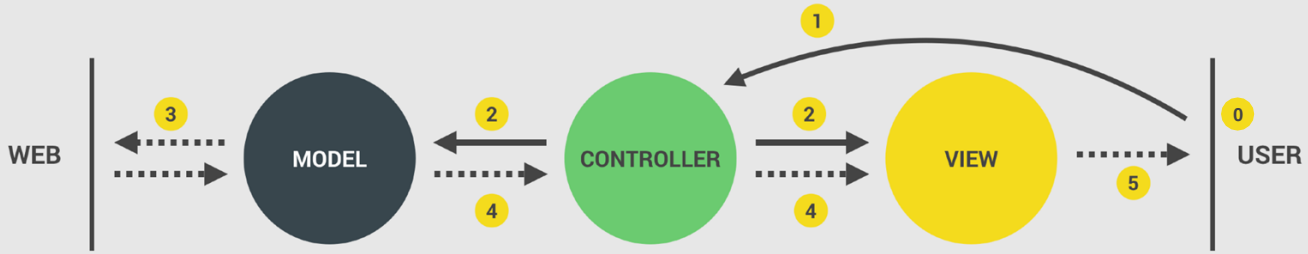
PRESENTATION LOGIC

→ Connected by function call and import

▪ ▪ ▶ Data flow

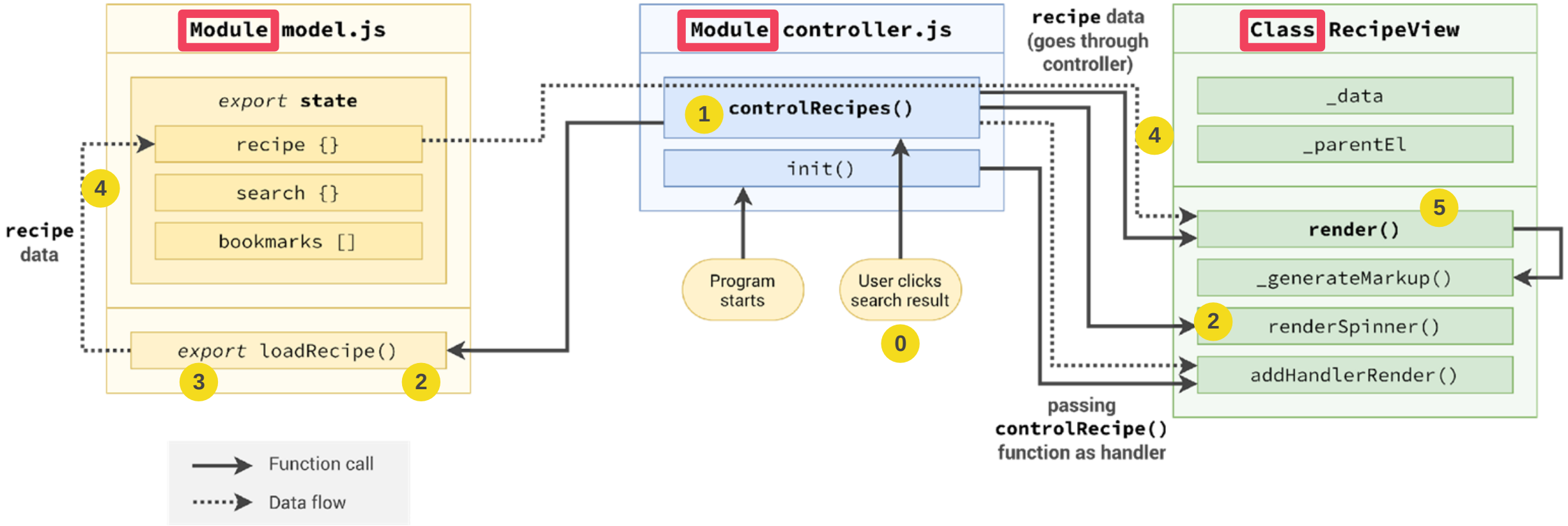
MODEL, VIEW AND CONTROLLER IN FORKIFY (RECIPE DISPLAY ONLY)

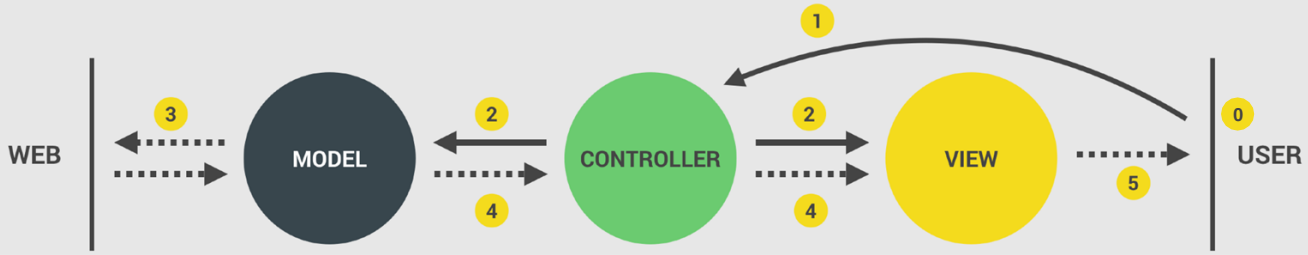




MVC IMPLEMENTATION (RECIPE DISPLAY ONLY)

forkify





THE COMPLETE JAVASCRIPT COURSE

FROM ZERO TO EXPERT!

SECTION

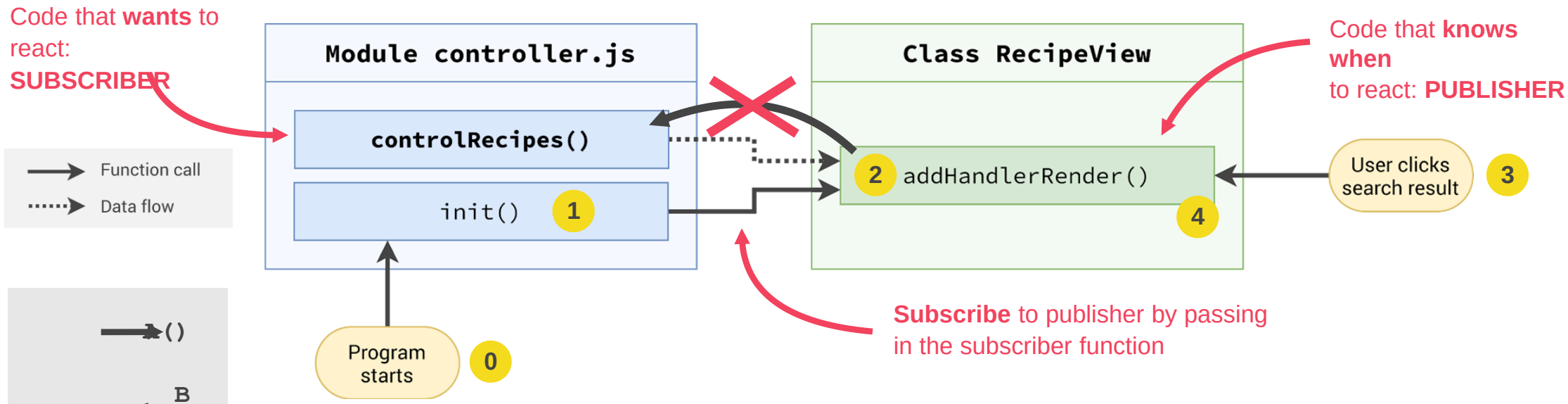
FORKIFY APP: BUILDING A MODERN APPLICATION

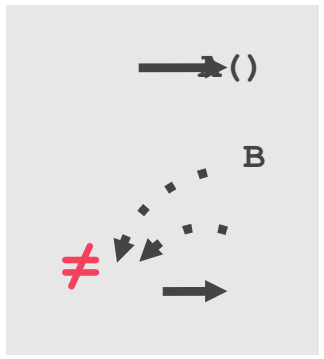
LECTURE

EVENT HANDLERS IN MVC: PUBLISHER-SUBSCRIBER PATTERN

JS

EVENT HANDLING IN MVC: PUBLISHER-SUBSCRIBER PATTERN





B

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

C

A(X)

- 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)

THE COMPLETE JAVASCRIPT COURSE

FROM ZERO TO EXPERT!

SECTION

FORKIFY APP: BUILDING A MODERN APPLICATION

LECTURE

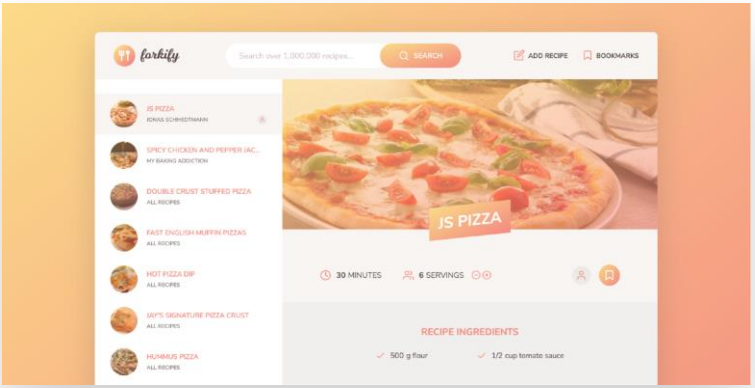
WRAPPING UP: FINAL CONSIDERATIONS

JS

IMPROVEMENT AND FEATURE IDEAS: CHALLENGES

forkify

- Display **number of pages** between the pagination buttons;
- Ability to **sort** search results by duration or number of ingredients; Perform **ingredient validation** in view, before submitting the form;
- **Improve recipe ingredient input**: separate in multiple fields and allow more than 6 ingredients;
- **Shopping list feature**: button on recipe to add ingredients to a list;
- **Weekly meal planning feature**: assign recipes to the next 7 days and show on a weekly calendar;
- **Get nutrition data** on each ingredient from spoonacular API (https://spoonacular.com/food-api) and calculate total calories of recipe.



END