



JONAS.IO  
SCHMEDTMANN

Subscribe here

# THE COMPLETE JAVASCRIPT COURSE

FROM ZERO TO EXPERT!



@JONASSCHMEDTMAN

Follow me here

# SLIDES FOR THEORY LECTURES

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

JS



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

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



WELCOME, WELCOME,  
WELCOME!



JONAS.IO  
SCHMEDTMANN

# THE COMPLETE JAVASCRIPT COURSE

FROM ZERO TO EXPERT!



@JONASSCHMEDTMAN

SECTION

WELCOME, WELCOME, WELCOME!

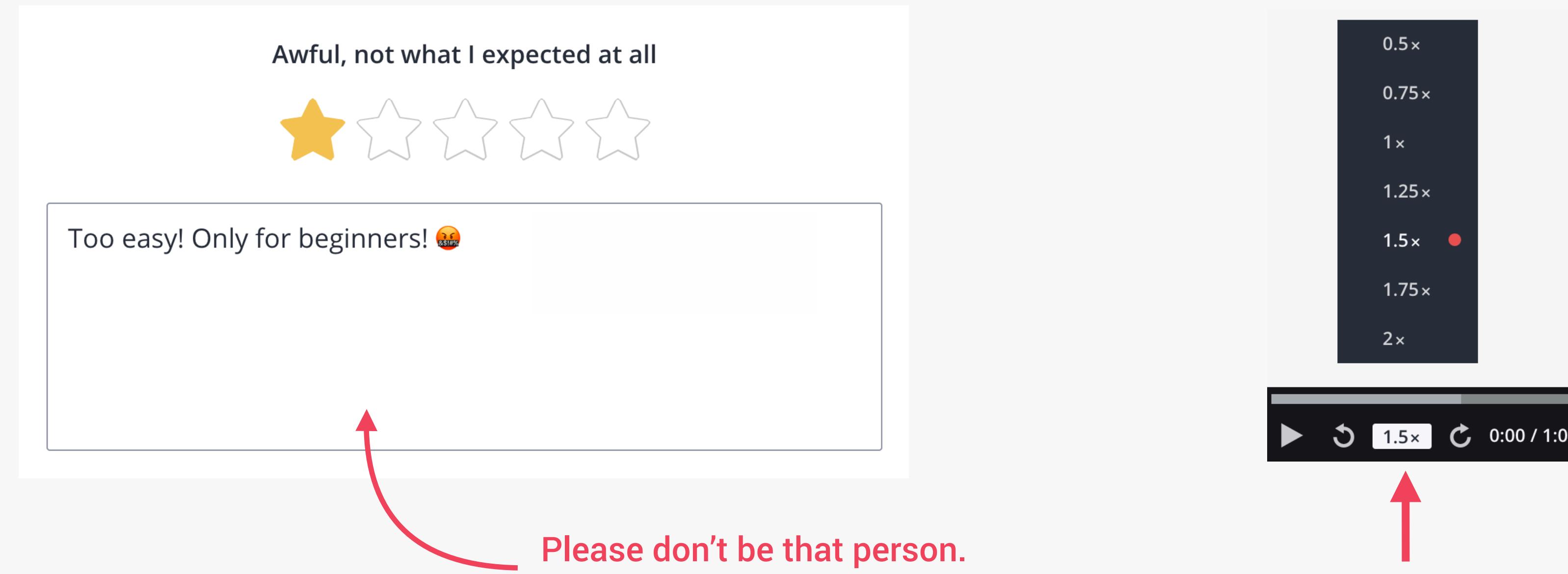
LECTURE

WATCH BEFORE YOU START!

JS

# SOME QUICK CONSIDERATIONS BEFORE WE START...

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



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

# SOME QUICK CONSIDERATIONS BEFORE WE START...



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



# SOME QUICK CONSIDERATIONS BEFORE WE START...



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



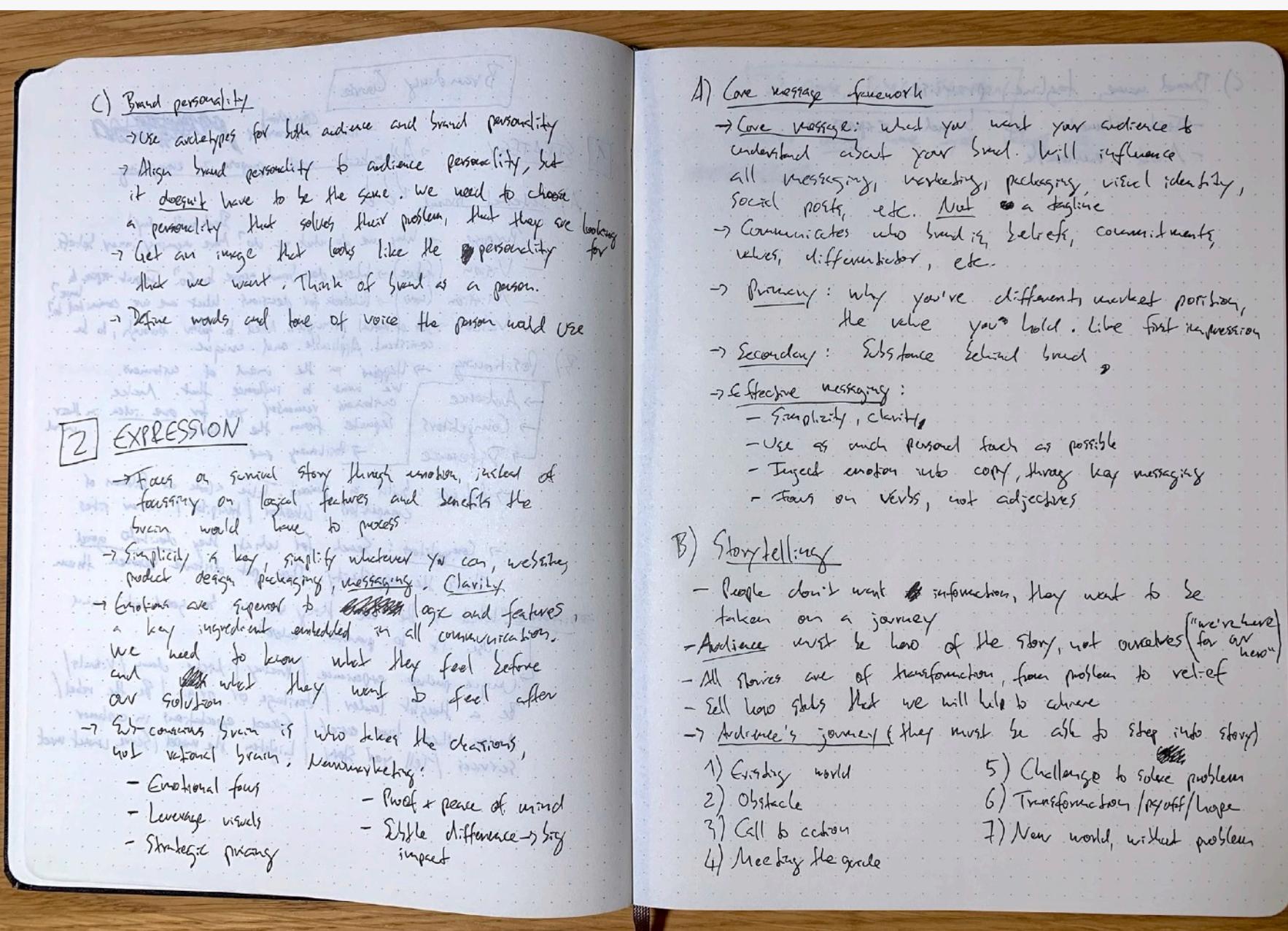
Watch for this sign!

PAUSE THE VIDEO  
FOR CHALLENGE

# SOME QUICK CONSIDERATIONS BEFORE WE START...



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



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

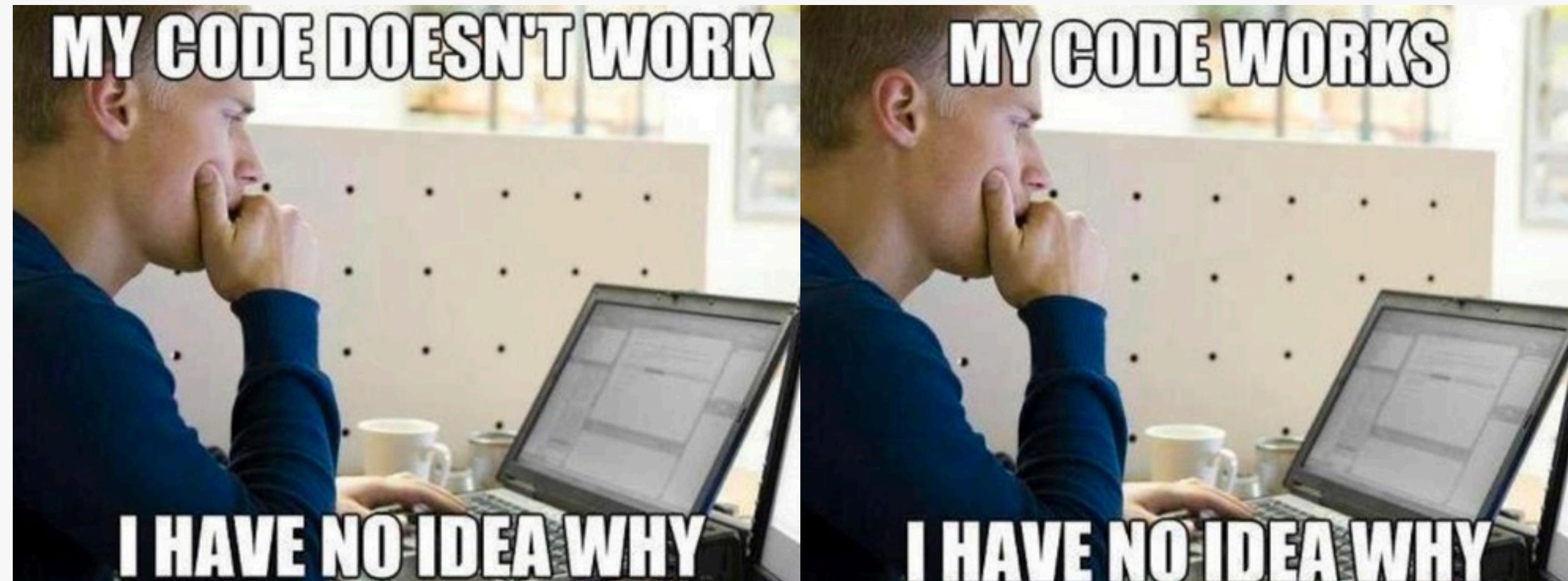
# SOME QUICK CONSIDERATIONS BEFORE WE START...

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



# SOME QUICK CONSIDERATIONS BEFORE WE START...

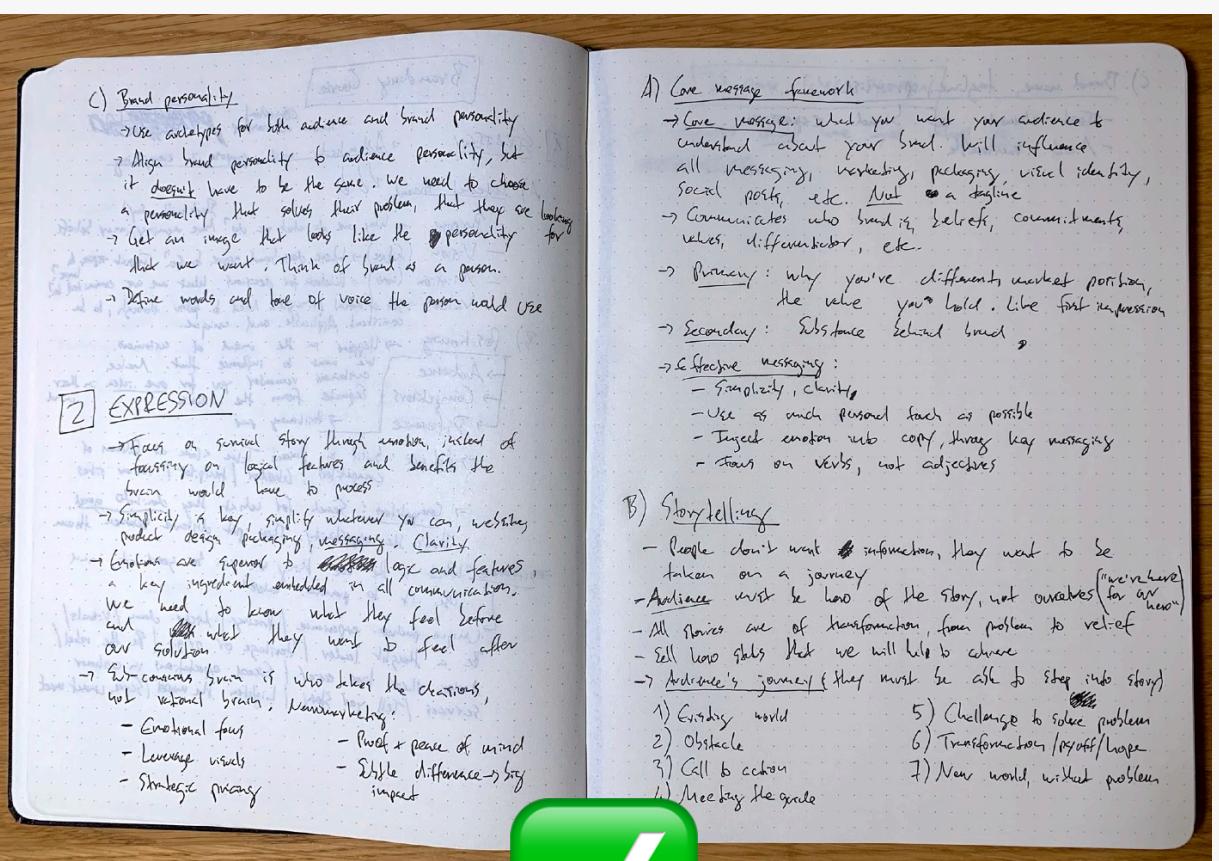
😊 In the first sections of the course, don't bother understanding WHY things work the way they do in JavaScript. Also, don't stress about efficient code, or fast code, or clean code. While learning, we just want to make things WORK. We will understand the WHY later in the course.



# SOME QUICK CONSIDERATIONS BEFORE WE START...



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

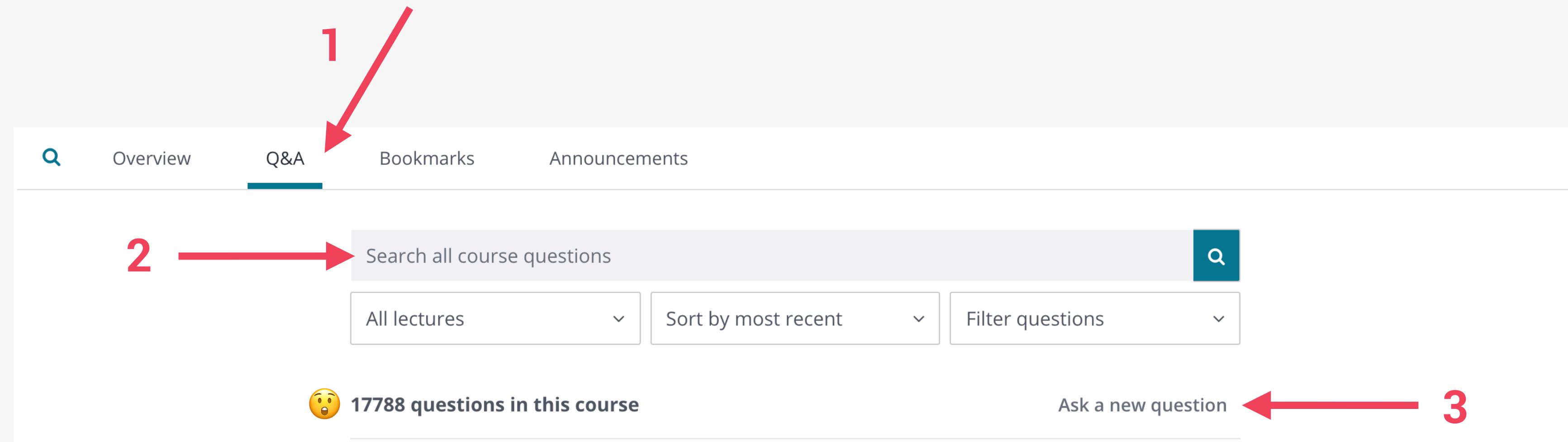


A screenshot of a mobile game titled "Guess My Number!". The title is at the top center in a large, bold, black font. Below it is a horizontal input bar with a white rectangular input field containing the number "3". To the left of the input field is a button labeled "Again!" and to the right is a button labeled "(Between 1 and 20)".



# SOME QUICK CONSIDERATIONS BEFORE WE START...

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

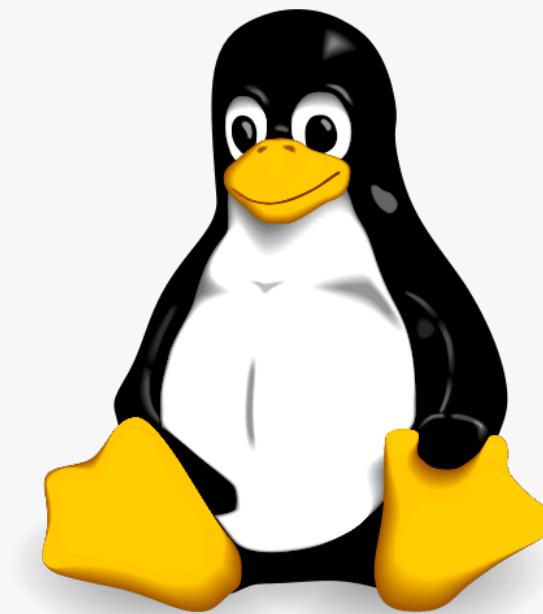
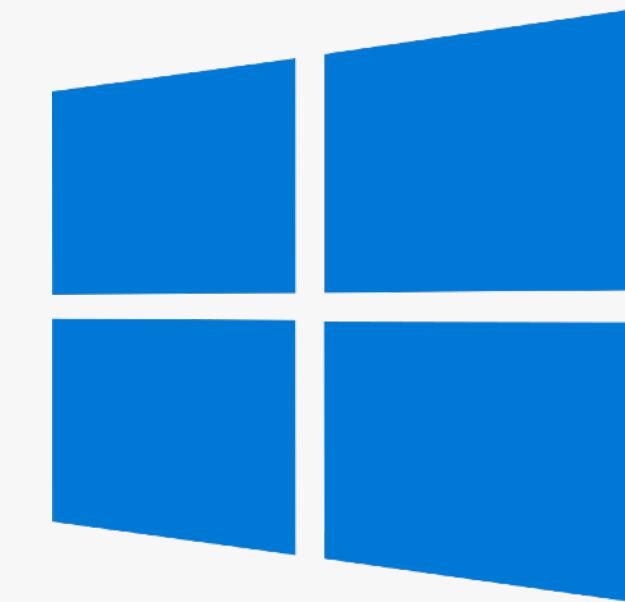


The screenshot shows a user interface for a course's Q&A section. At the top, there are navigation links: Overview, Q&A (which is highlighted with a teal underline), Bookmarks, and Announcements. Below this is a search bar with the placeholder "Search all course questions" and a magnifying glass icon. Underneath the search bar are three dropdown menus: "All lectures", "Sort by most recent", and "Filter questions". At the bottom left, there is a statistic showing "17788 questions in this course" next to a surprised face emoji. On the bottom right, there is a button labeled "Ask a new question". Red numbers 1, 2, and 3 are overlaid on the image to guide the user: 1 points to the "Q&A" link, 2 points to the search bar, and 3 points to the "Ask a new question" button.

# SOME QUICK CONSIDERATIONS BEFORE WE START...

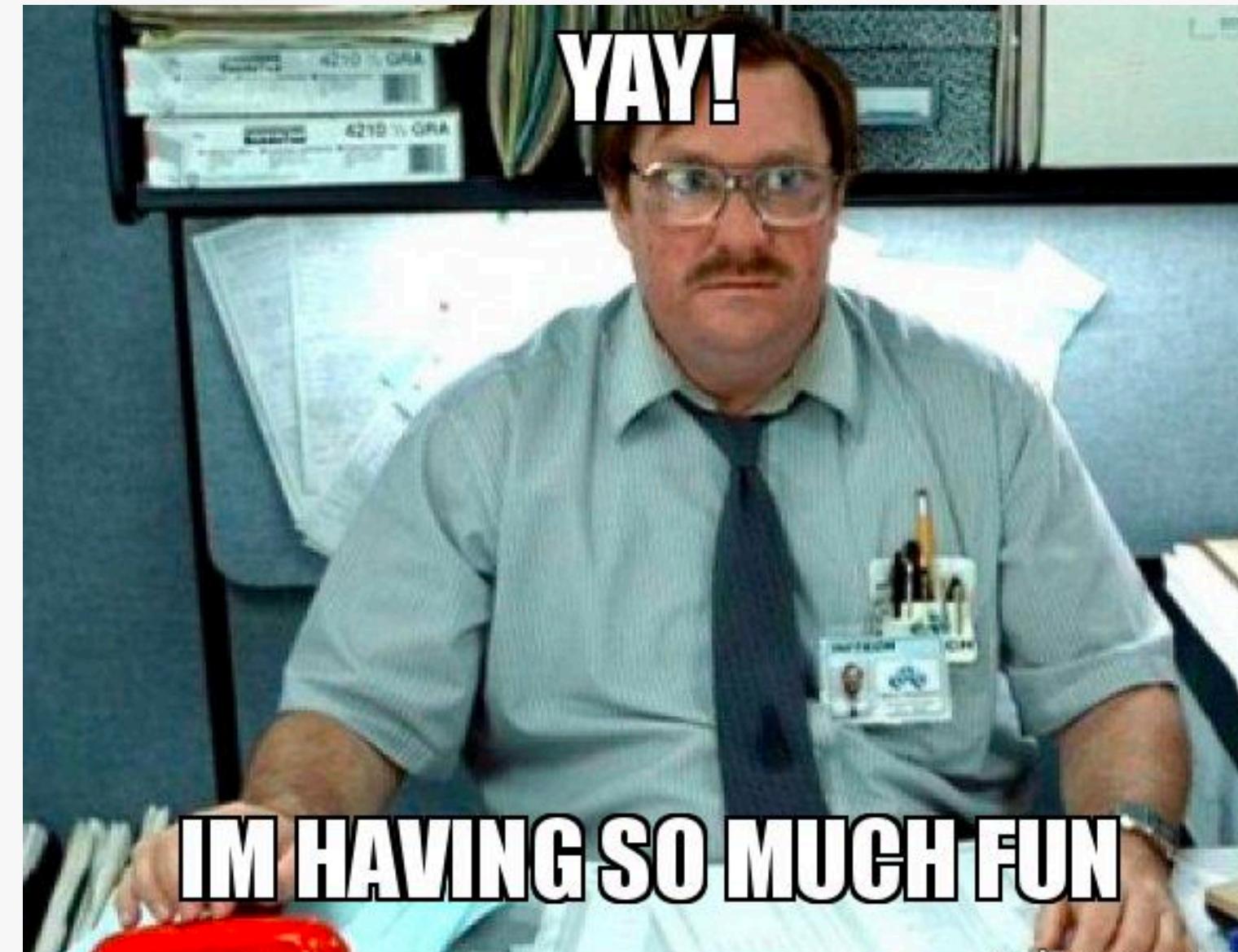


I recorded this course on a Mac, but everything works the exact same way on Windows or Linux. If something doesn't work on your computer, it's NOT because you're using a different OS.



# SOME QUICK CONSIDERATIONS BEFORE WE START...

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



And I mean **REAL** fun 😊



# JAVASCRIPT FUNDAMENTALS – PART 1



JONAS.IO  
SCHMEDTMANN

# THE COMPLETE JAVASCRIPT COURSE

FROM ZERO TO EXPERT!



@JONASSCHMEDTMAN

SECTION

JAVASCRIPT FUNDAMENTALS - PART 1

LECTURE

A BRIEF INTRODUCTION TO  
JAVASCRIPT

JS

# WHAT IS JAVASCRIPT?

JAVASCRIPT

Based on objects, for  
storing most kinds of data

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

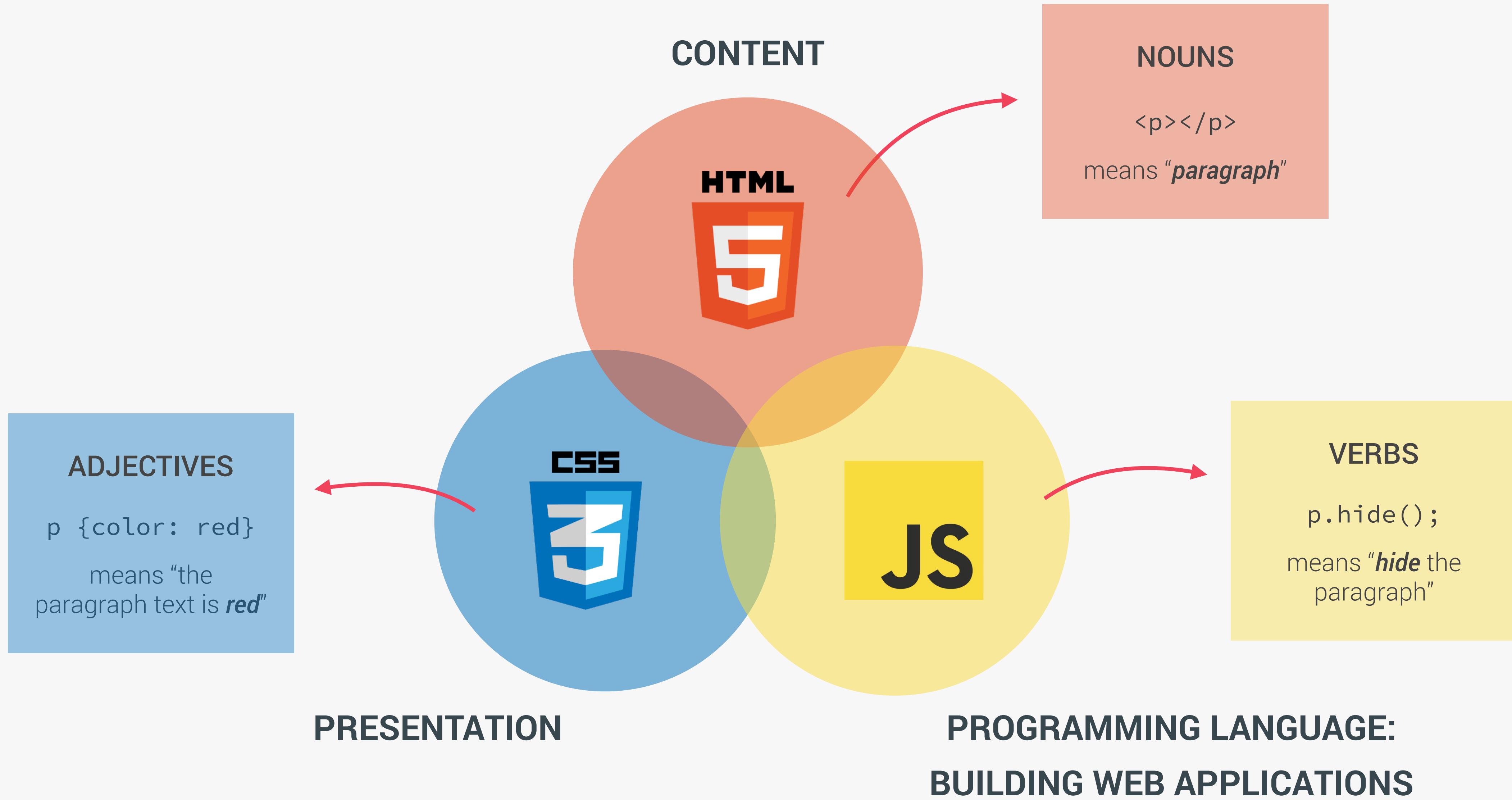
Instruct computer to *do things*

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

We can use different styles  
of programming



# THE ROLE OF JAVASCRIPT IN WEB DEVELOPMENT



# EXAMPLE OF DYNAMIC EFFECTS / WEB APPLICATION

The image shows a screenshot of the Twitter mobile application interface. On the left, a sidebar menu includes Home, Explore, Notifications (16), Messages, Bookmarks, Lists, Profile (highlighted in green), and More. A red arrow points from the 'Profile' section to a green 'Tweet' button at the bottom of the sidebar. Another red arrow points from the 'Profile' section to a green circular loading indicator on the main profile page. On the right, the main profile page for 'Jonas Schmedtmann' (@jonasschmedtmann) is displayed. It features a banner for 'CODING COURSES DONE RIGHT.', a profile picture, and a bio: 'Developer. Designer. Online teacher.' Below the bio are statistics: 37 Following and 19.4K Followers. At the bottom of the profile page, there are tabs for Tweets, Tweets & replies, Media, and Likes. A red arrow points from the 'Tweets' tab to a green circular loading indicator. In the background, a search bar shows 'Search Twitter' and a card for 'Node.js, Express, MongoDB & More: The Complete Bootcamp'. Red arrows point from the top of this card to the top of the sidebar and the bottom of the sidebar. At the very bottom of the screen, there are links for Terms, Privacy policy, Cookies, Ads info, and More.

Show spinner + loading data in the background

Show tweet box after clicking

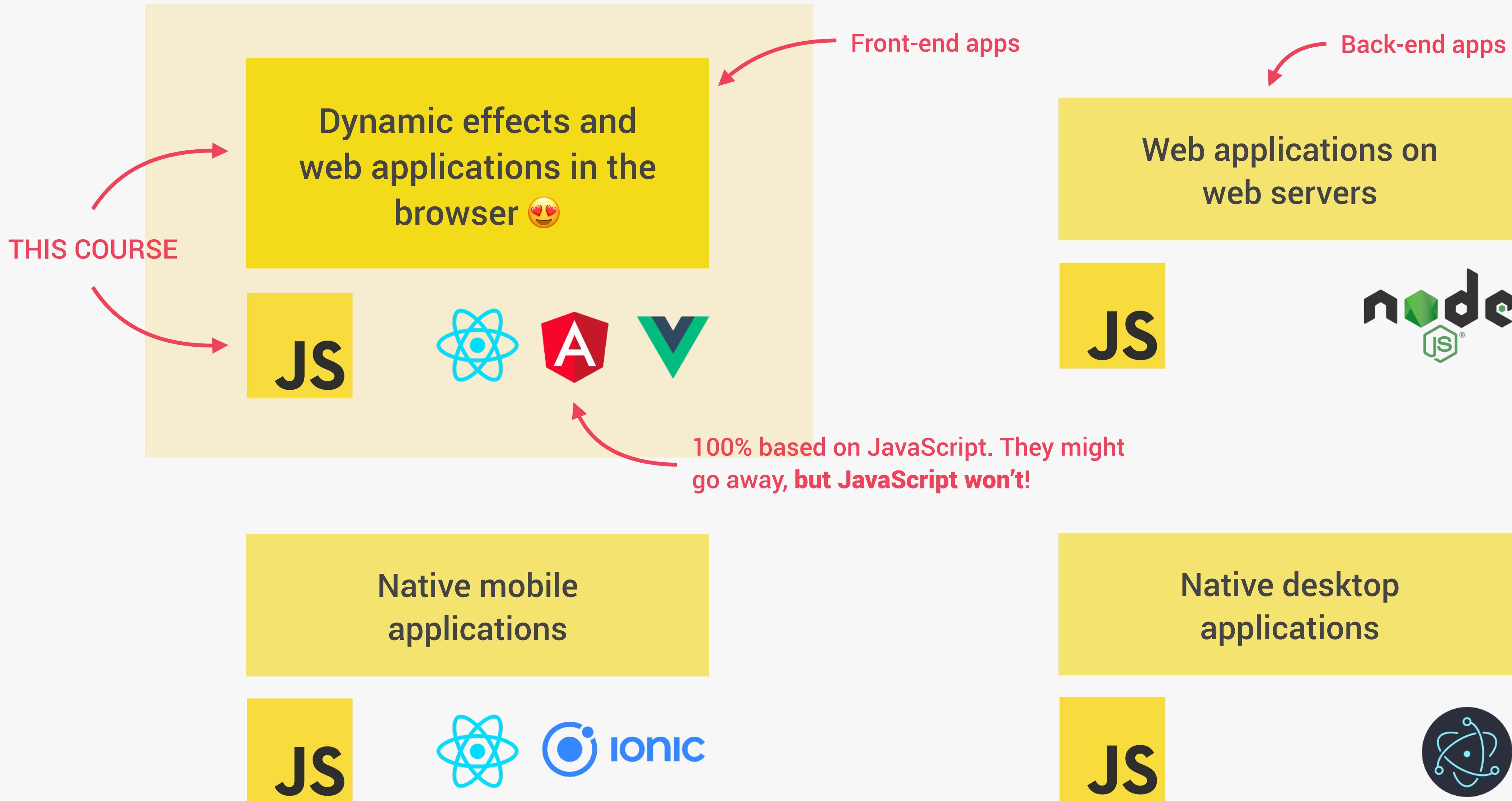
Show tweets after loading data

Display user info on hover

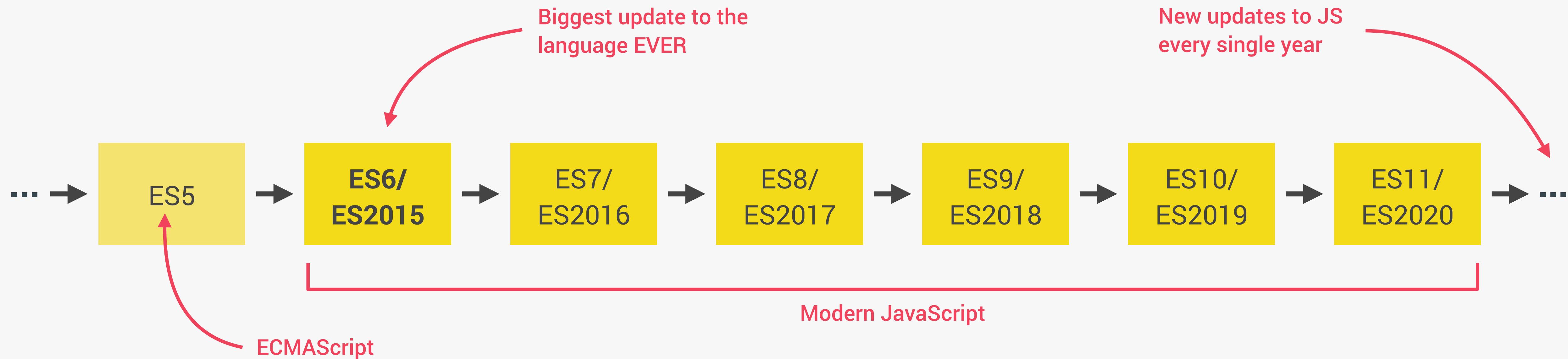
Show spinner + loading data in the background

Show data after loading

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



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



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



Let's finally get started!





JONAS.IO  
SCHMEDTMANN

# THE COMPLETE JAVASCRIPT COURSE

FROM ZERO TO EXPERT!



@JONASSCHMEDTMAN

SECTION

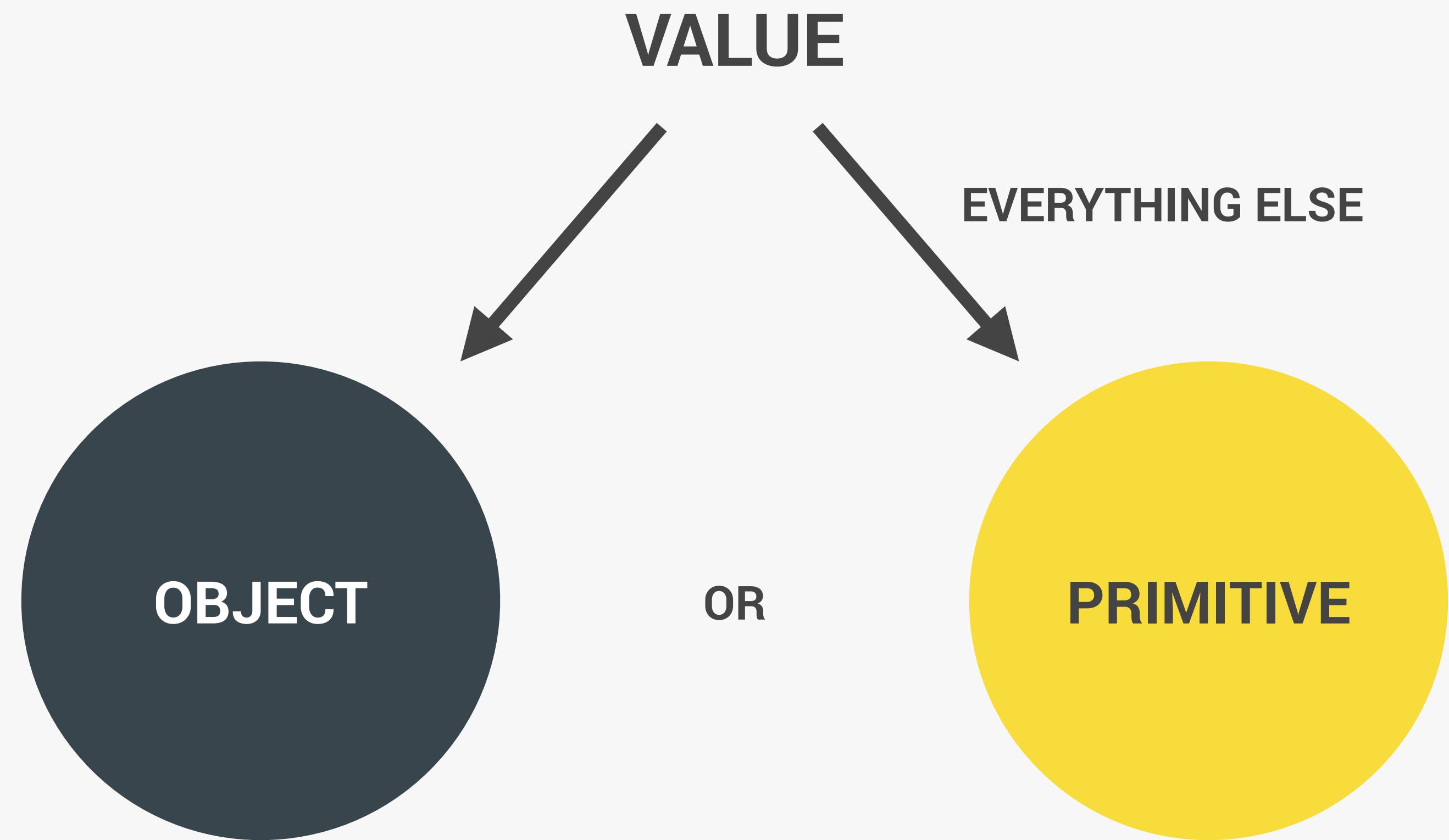
JAVASCRIPT FUNDAMENTALS - PART 1

LECTURE

DATA TYPES

JS

# OBJECTS AND PRIMITIVES



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

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

# THE 7 PRIMITIVE DATA TYPES

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

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

Value has type, NOT variable!





JONAS.IO  
SCHMEDTMANN

# THE COMPLETE JAVASCRIPT COURSE

FROM ZERO TO EXPERT!



@JONASSCHMEDTMAN

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"

A OR B

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

NOT A, NOT B



Possible values

		A
AND		TRUE FALSE
B	TRUE	TRUE FALSE
	FALSE	FALSE FALSE

Results of operation, depending on 2 variables

true when **ALL** are true

No matter how many variables

A

OR		TRUE FALSE
B	TRUE	TRUE TRUE
	FALSE	TRUE FALSE

true when **ONE** is true

Inverts **true/false** value

👉 EXAMPLE:

A: Sarah has a driver's license

B: Sarah has good vision

Boolean variables that can be either TRUE or FALSE

# AN EXAMPLE



## BOOLEAN VARIABLES

- 👉 A: Age is greater or equal 20
- 👉 B: Age is less than 30

false

true

age = 16

		A	B	
		AND	TRUE	FALSE
A	TRUE	TRUE	FALSE	
	FALSE	FALSE	FALSE	

## LET'S USE OPERATORS!

- 👉 !A

true

false

- 👉 A AND B

false true

false

- 👉 A OR B

false true

true

- 👉 !A AND B

true true

true

- 👉 A OR !B

false false

false

		A	B	
		OR	TRUE	FALSE
A	TRUE	TRUE	TRUE	
	FALSE	TRUE	FALSE	





JONAS.IO  
SCHMEDTMANN

# THE COMPLETE JAVASCRIPT COURSE

FROM ZERO TO EXPERT!



@JONASSCHMEDTMAN

SECTION

JAVASCRIPT FUNDAMENTALS - PART 1

LECTURE

JAVASCRIPT RELEASES: ES5, ES6+  
AND ESNEXT

JS

# A BRIEF HISTORY OF JAVASCRIPT

1995

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



1996

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

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



1997

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



2009

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

2015

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

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

2016 – ∞

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

# BACKWARDS COMPATIBILITY: DON'T BREAK THE WEB!

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

1997



BACKWARDS  
COMPATIBLE

Modern JavaScript  
Engine

2020

DON'T BREAK THE WEB!

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

Modern JavaScript  
Engine

2020

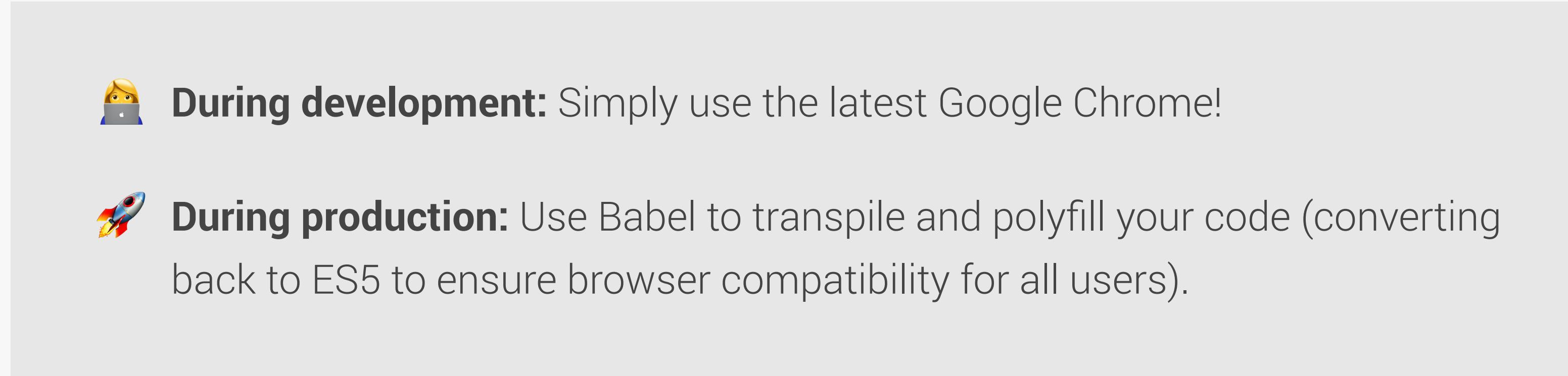
NOT FORWARD  
COMPATIBLE



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

2089

# HOW TO USE MODERN JAVASCRIPT TODAY


Feature name	Current browser	Trunk	Babel 6.2	Babel 7	Browser	Edge	Firefox ESR	Firefox Dev	IE 11	IE 12	IE 13	IE 14	IE 15	IE 16	IE 17	IE 18	IE 19	IE 20	IE 21	IE 22	IE 23	IE 24	IE 25	IE 26	IE 27	IE 28	IE 29	IE 30	IE 31	IE 32	IE 33	IE 34	IE 35	IE 36	IE 37	IE 38	IE 39	IE 40	IE 41	IE 42	IE 43	IE 44	IE 45	IE 46	IE 47	IE 48	IE 49	IE 50	IE 51	IE 52	IE 53	IE 54	IE 55	IE 56	IE 57	IE 58	IE 59	IE 60	IE 61	IE 62	IE 63	IE 64	IE 65	IE 66	IE 67	IE 68	IE 69	IE 70	IE 71	IE 72	IE 73	IE 74	IE 75	IE 76	IE 77	IE 78	IE 79	IE 80	IE 81	IE 82	IE 83	IE 84	IE 85	IE 86	IE 87	IE 88	IE 89	IE 90	IE 91	IE 92	IE 93	IE 94	IE 95	IE 96	IE 97	IE 98	IE 99	IE 100	IE 101	IE 102	IE 103	IE 104	IE 105	IE 106	IE 107	IE 108	IE 109	IE 110	IE 111	IE 112	IE 113	IE 114	IE 115	IE 116	IE 117	IE 118	IE 119	IE 120	IE 121	IE 122	IE 123	IE 124	IE 125	IE 126	IE 127	IE 128	IE 129	IE 130	IE 131	IE 132	IE 133	IE 134	IE 135	IE 136	IE 137	IE 138	IE 139	IE 140	IE 141	IE 142	IE 143	IE 144	IE 145	IE 146	IE 147	IE 148	IE 149	IE 150	IE 151	IE 152	IE 153	IE 154	IE 155	IE 156	IE 157	IE 158	IE 159	IE 160	IE 161	IE 162	IE 163	IE 164	IE 165	IE 166	IE 167	IE 168	IE 169	IE 170	IE 171	IE 172	IE 173	IE 174	IE 175	IE 176	IE 177	IE 178	IE 179	IE 180	IE 181	IE 182	IE 183	IE 184	IE 185	IE 186	IE 187	IE 188	IE 189	IE 190	IE 191	IE 192	IE 193	IE 194	IE 195	IE 196	IE 197	IE 198	IE 199	IE 200	IE 201	IE 202	IE 203	IE 204	IE 205	IE 206	IE 207	IE 208	IE 209	IE 210	IE 211	IE 212	IE 213	IE 214	IE 215	IE 216	IE 217	IE 218	IE 219	IE 220	IE 221	IE 222	IE 223	IE 224	IE 225	IE 226	IE 227	IE 228	IE 229	IE 230	IE 231	IE 232	IE 233	IE 234	IE 235	IE 236	IE 237	IE 238	IE 239	IE 240	IE 241	IE 242	IE 243	IE 244	IE 245	IE 246	IE 247	IE 248	IE 249	IE 250	IE 251	IE 252	IE 253	IE 254	IE 255	IE 256	IE 257	IE 258	IE 259	IE 260	IE 261	IE 262	IE 263	IE 264	IE 265	IE 266	IE 267	IE 268	IE 269	IE 270	IE 271	IE 272	IE 273	IE 274	IE 275	IE 276	IE 277	IE 278	IE 279	IE 280	IE 281	IE 282	IE 283	IE 284	IE 285	IE 286	IE 287	IE 288	IE 289	IE 290	IE 291	IE 292	IE 293	IE 294	IE 295	IE 296	IE 297	IE 298	IE 299	IE 300	IE 301	IE 302	IE 303	IE 304	IE 305	IE 306	IE 307	IE 308	IE 309	IE 310	IE 311	IE 312	IE 313	IE 314	IE 315	IE 316	IE 317	IE 318	IE 319	IE 320	IE 321	IE 322	IE 323	IE 324	IE 325	IE 326	IE 327	IE 328	IE 329	IE 330	IE 331	IE 332	IE 333	IE 334	IE 335	IE 336	IE 337	IE 338	IE 339	IE 340	IE 341	IE 342	IE 343	IE 344	IE 345	IE 346	IE 347	IE 348	IE 349	IE 350	IE 351	IE 352	IE 353	IE 354	IE 355	IE 356	IE 357	IE 358	IE 359	IE 360	IE 361	IE 362	IE 363	IE 364	IE 365	IE 366	IE 367	IE 368	IE 369	IE 370	IE 371	IE 372	IE 373	IE 374	IE 375	IE 376	IE 377	IE 378	IE 379	IE 380	IE 381	IE 382	IE 383	IE 384	IE 385	IE 386	IE 387	IE 388	IE 389	IE 390	IE 391	IE 392	IE 393	IE 394	IE 395	IE 396	IE 397	IE 398	IE 399	IE 400	IE 401	IE 402	IE 403	IE 404	IE 405	IE 406	IE 407	IE 408	IE 409	IE 410	IE 411	IE 412	IE 413	IE 414	IE 415	IE 416	IE 417	IE 418	IE 419	IE 420	IE 421	IE 422	IE 423	IE 424	IE 425	IE 426	IE 427	IE 428	IE 429	IE 430	IE 431	IE 432	IE 433	IE 434	IE 435	IE 436	IE 437	IE 438	IE 439	IE 440	IE 441	IE 442	IE 443	IE 444	IE 445	IE 446	IE 447	IE 448	IE 449	IE 450	IE 451	IE 452	IE 453	IE 454	IE 455	IE 456	IE 457	IE 458	IE 459	IE 460	IE 461	IE 462	IE 463	IE 464	IE 465	IE 466	IE 467	IE 468	IE 469	IE 470	IE 471	IE 472	IE 473	IE 474	IE 475	IE 476	IE 477	IE 478	IE 479	IE 480	IE 481	IE 482	IE 483	IE 484	IE 485	IE 486	IE 487	IE 488	IE 489	IE 490	IE 491	IE 492	IE 493	IE 494	IE 495	IE 496	IE 497	IE 498	IE 499	IE 500	IE 501	IE 502	IE 503	IE 504	IE 505	IE 506	IE 507	IE 508	IE 509	IE 510	IE 511	IE 512	IE 513	IE 514	IE 515	IE 516	IE 517	IE 518	IE 519	IE 520	IE 521	IE 522	IE 523	IE 524	IE 525	IE 526	IE 527	IE 528	IE 529	IE 530	IE 531	IE 532	IE 533	IE 534	IE 535	IE 536	IE 537	IE 538	IE 539	IE 540	IE 541	IE 542	IE 543	IE 544	IE 545	IE 546	IE 547	IE 548	IE 549	IE 550	IE 551	IE 552	IE 553	IE 554	IE 555	IE 556	IE 557	IE 558	IE 559	IE 560	IE 561	IE 562	IE 563	IE 564	IE 565	IE 566	IE 567	IE 568	IE 569	IE 570	IE 571	IE 572	IE 573	IE 574	IE 575	IE 576	IE 577	IE 578	IE 579	IE 580	IE 581	IE 582	IE 583	IE 584	IE 585	IE 586	IE 587	IE 588	IE 589	IE 590	IE 591	IE 592	IE 593	IE 594	IE 595	IE 596	IE 597	IE 598	IE 599	IE 600	IE 601	IE 602	IE 603	IE 604	IE 605	IE 606	IE 607	IE 608	IE 609	IE 610	IE 611	IE 612	IE 613	IE 614	IE 615	IE 616	IE 617	IE 618	IE 619	IE 620	IE 621	IE 622	IE 623	IE 624	IE 625	IE 626	IE 627	IE 628	IE 629	IE 630	IE 631	IE 632	IE 633	IE 634	IE 635	IE 636	IE 637	IE 638	IE 639	IE 640	IE 641	IE 642	IE 643	IE 644	IE 645	IE 646	IE 647	IE 648	IE 649	IE 650	IE 651	IE 652	IE 653	IE 654	IE 655	IE 656	IE 657	IE 658	IE 659	IE 660	IE 661	IE 662	IE 663	IE 664	IE 665	IE 666	IE 667	IE 668	IE 669	IE 670	IE 671	IE 672	IE 673	IE 674	IE 675	IE 676	IE 677	IE 678	IE 679	IE 680	IE 681	IE 682	IE 683	IE 684	IE 685	IE 686	IE 687	IE 688	IE 689	IE 690	IE 691	IE 692	IE 693	IE 694	IE 695	IE 696	IE 697	IE 698	IE 699	IE 700	IE 701	IE 702	IE 703	IE 704	IE 705	IE 706	IE 707	IE 708	IE 709	IE 710	IE 711	IE 712	IE 713	IE 714	IE 715	IE 716	IE 717	IE 718	IE 719	IE 720	IE 721	IE 722	IE 723	IE 724	IE 725	IE 726	IE 727	IE 728	IE 729	IE 730	IE 731	IE 732	IE 733	IE 734	IE 735	IE 736	IE 737	IE 738	IE 739	IE 740	IE 741	IE 742	IE 743	IE 744	IE 745	IE 746	IE 747	IE 748	IE 749	IE 750	IE 751	IE 752	IE 753	IE 754	IE 755	IE 756	IE 757	IE 758	IE 759	IE 760	IE 761	IE 762	IE 763	IE 764	IE 765	IE 766	IE 767	IE 768	IE 769	IE 770	IE 771	

# MODERN JAVASCRIPT FROM THE BEGINNING



Learn **modern JavaScript from the beginning!**



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

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

- 👉 You will better understand how JavaScript actually works;
- 👉 Many tutorials and code you find online today are still in ES5;
- 👉 When working on old codebases, these will be written in ES5.



# JAVASCRIPT FUNDAMENTALS – PART 2



JONAS.IO  
SCHMEDTMANN

# THE COMPLETE JAVASCRIPT COURSE

FROM ZERO TO EXPERT!



@JONASSCHMEDTMAN

SECTION

JAVASCRIPT FUNDAMENTALS - PART 2

LECTURE

FUNCTIONS CALLING OTHER  
FUNCTIONS

JS

# CALLING A FUNCTION INSIDE A FUNCTION: DATA FLOW

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

The diagram illustrates the data flow in the `fruitProcessor` function. It starts with two arguments: `apples` (value 8) and `oranges` (value 2). These values are passed to the `cutPieces` function, which returns `applePieces` (value 8) and `orangePieces` (value 2). These intermediate values are then used in the template string for `juice`. Finally, the `juice` string is returned by the `fruitProcessor` function and logged to the console.





JONAS.IO  
SCHMEDTMANN

# THE COMPLETE JAVASCRIPT COURSE

FROM ZERO TO EXPERT!



@JONASSCHMEDTMAN

SECTION

JAVASCRIPT FUNDAMENTALS - PART 2

LECTURE

REVIEWING FUNCTIONS

JS

# FUNCTIONS REVIEW: 3 DIFFERENT FUNCTION TYPES

## 👉 Function declaration

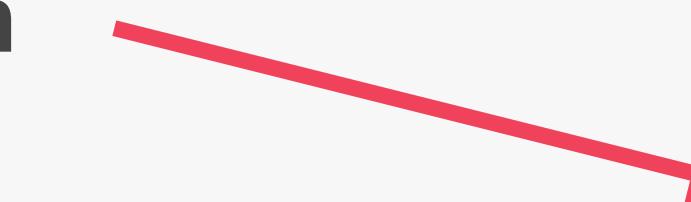
Function that can be used before it's declared

## 👉 Function 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

Function name

Parameters: placeholders to receive input values. Like local variables of a function

Function body: block of code that we want to reuse. Processes the function's input data

return statement to output a value from the function and terminate execution

```
function calcAge(birthYear, firstName) {  
  const age = 2037 - birthYear;  
  console.log(`${firstName} is ${age} years old`);  
  return age;  
}
```

Variable to save returned value (function output)

Arguments: actual values of function parameters, to input data

Calling, running or invoking the function, using ()



# DEVELOPER SKILLS & EDITOR SETUP



JONAS.IO  
SCHMEDTMANN

# THE COMPLETE JAVASCRIPT COURSE

FROM ZERO TO EXPERT!



@JONASSCHMEDTMAN

SECTION

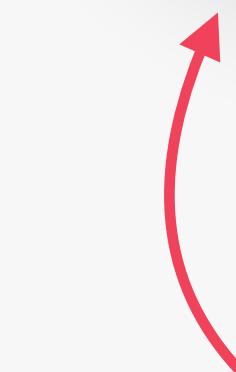
DEVELOPER SKILLS & EDITOR SETUP

LECTURE

LEARNING HOW TO CODE

JS

# HOW TO FAIL 🤦 AT LEARNING HOW TO CODE



John  
(not actually...)

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

# HOW TO SUCCEED AT LEARNING HOW TO CODE

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

↓ **FIX**

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

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

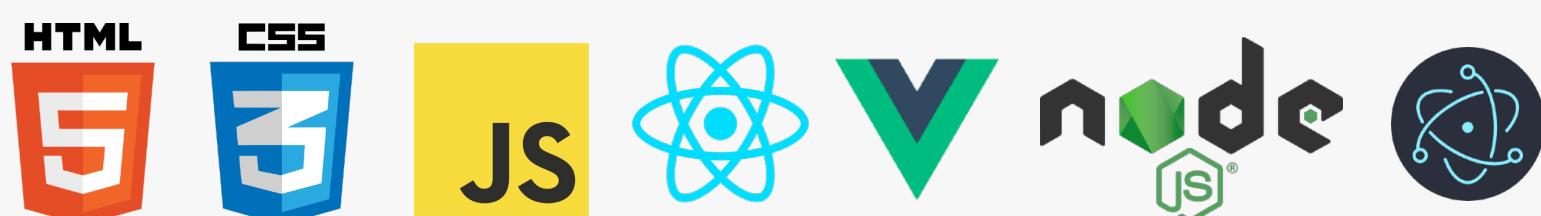
↓ **FIX**

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

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

↓ **FIX**

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



PAUSE THE VIDEO  
FOR CHALLENGE

codewars 

# HOW TO SUCCEED AT LEARNING HOW TO CODE

💥 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 know everything**

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



getify  
@getify

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

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

You're not alone in these struggles.

1,601 3:33 PM - Mar 10, 2018



# 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!







JONAS.IO  
SCHMEDTMANN

# THE COMPLETE JAVASCRIPT COURSE

FROM ZERO TO EXPERT!



@JONASSCHMEDTMAN

SECTION

DEVELOPER SKILLS & EDITOR SETUP

LECTURE

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

JS

# HOW TO FAIL 🤦 AT SOLVING PROBLEMS



John can  
code now 😊

## WHENEVER JOHN ENCOUNTERS A PROBLEM:

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

↓ FIX

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

NEXT SLIDE →

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

# 4 STEPS TO SOLVE ANY PROBLEM

1

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

## EXAMPLE

Project Manager: “We need a function that reverses whatever we pass into it”

1

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

# 4 STEPS TO SOLVE ANY PROBLEM

1

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



2

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

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

## EXAMPLE

Project Manager: “We need a function that reverses whatever we pass into it”

# 4 STEPS TO SOLVE ANY PROBLEM

1

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



2

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



3

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

## EXAMPLE

Project Manager: “*We need a function that reverses whatever we pass into it*”

3

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



# 4 STEPS TO SOLVE ANY PROBLEM

1

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



2

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



3

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



4

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

## EXAMPLE

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

4

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

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

  return reversed value
```





JONAS.IO  
SCHMEDTMANN

# THE COMPLETE JAVASCRIPT COURSE

FROM ZERO TO EXPERT!



@JONASSCHMEDTMAN

SECTION

DEVELOPER SKILLS & EDITOR SETUP

LECTURE

DEBUGGING (FIXING ERRORS)

JS

# WHAT IS A SOFTWARE BUG?

- 👉 **Software bug:** Defect or problem in a computer program.  
Basically, any **unexpected or unintended behavior** of a computer program is a software bug.

- 👉 Bugs are **completely normal** in software development!

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

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



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



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

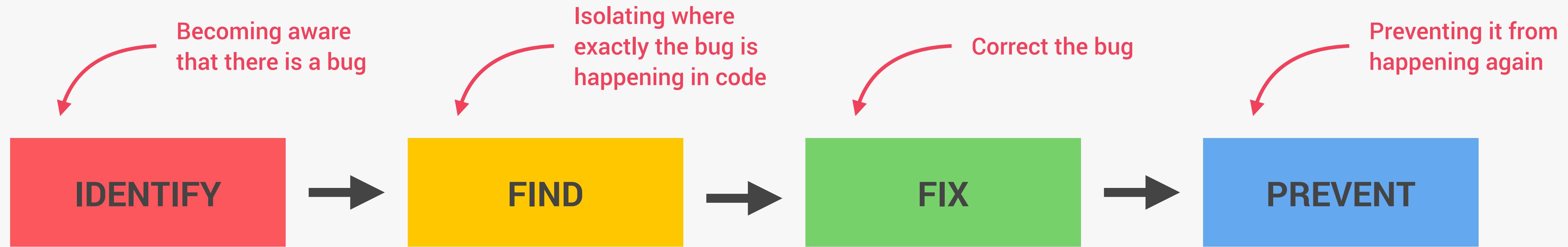
- 👉 Debugging: Process of finding, fixing and preventing bugs.

0800 Autan started  
1000 " stopped - autan ✓  
13" UC (032) MP - MC { 1.2700 9.037 847 025  
033 PRO 2 1.982 647 000 9.037 846 995  
2.130 476 415 4.615 925 055  
const 2.130 676 415  
Relays 6-2 in 033 failed special speed test  
in relay " 10.000 test .  
Relays changed  
1100 Started Cosine Tape (Sine check)  
1525 Started Multi Adder Test.  
1545 Relay #70 Panel F  
(moth) in relay.  
First actual case of bug being found.  
Autan started.  
closed down.



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

# 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



# JAVASCRIPT IN THE BROWSER: DOM AND EVENTS FUNDAMENTALS



# THE COMPLETE JAVASCRIPT COURSE

FROM ZERO TO EXPERT!



@JONASSCHMEDTMAN

SECTION

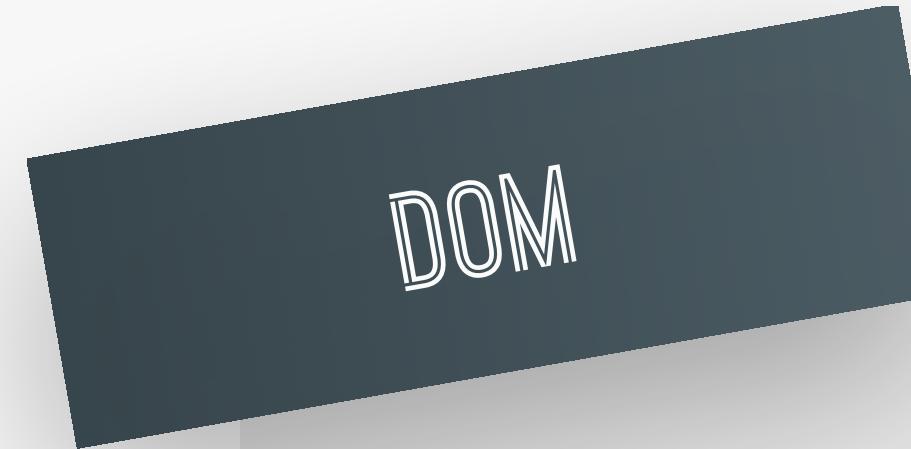
JAVASCRIPT IN THE BROWSER: DOM  
AND EVENTS FUNDAMENTALS

LECTURE

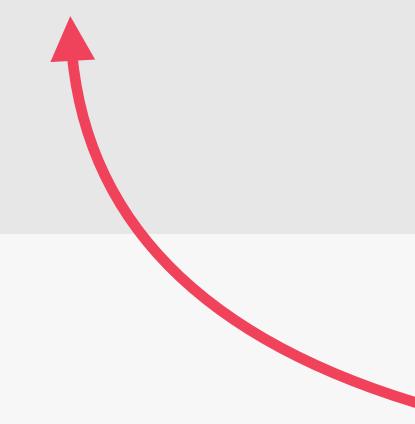
WHAT'S THE DOM AND DOM  
MANIPULATION

JS

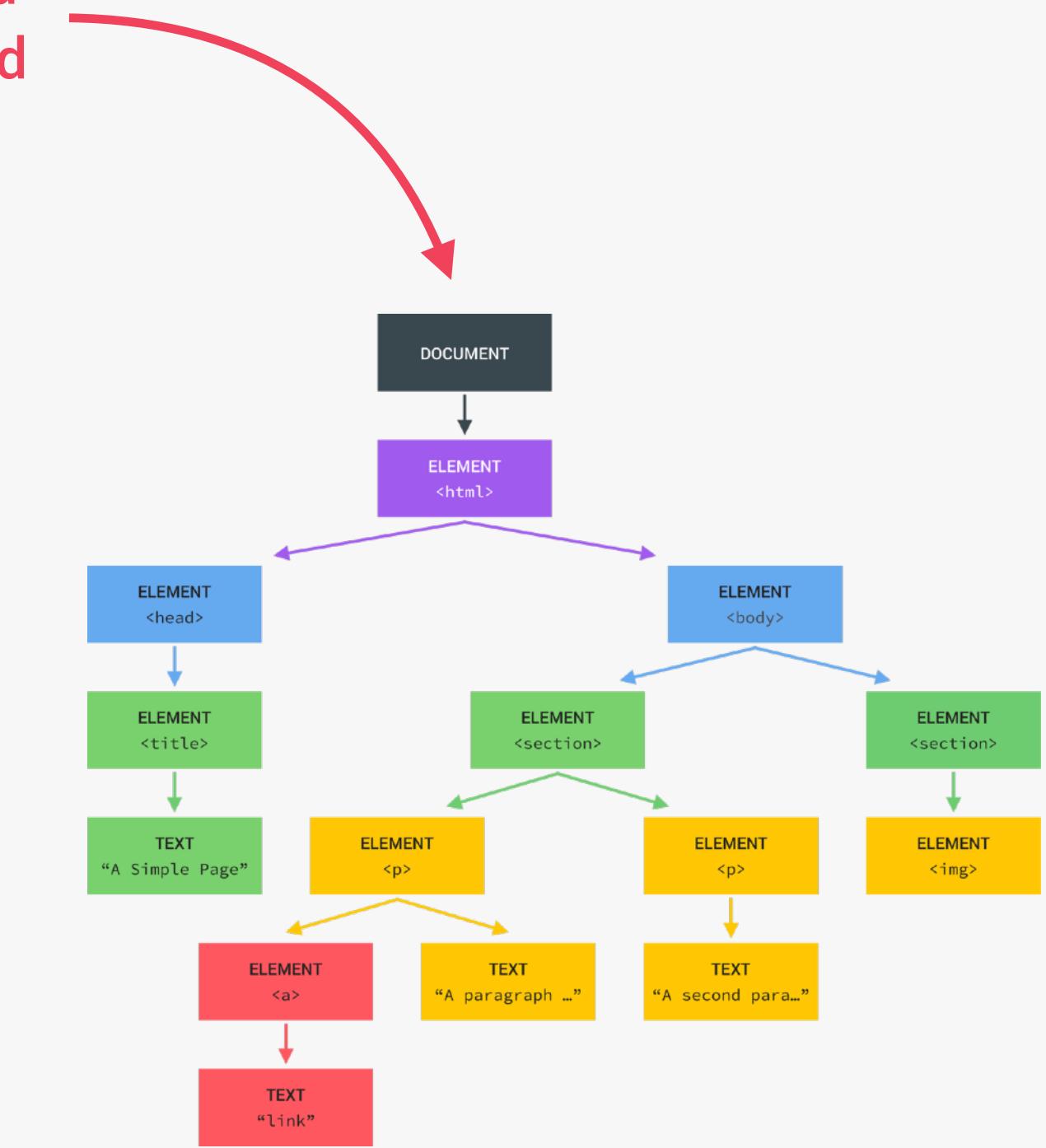
# WHAT IS THE DOM?



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

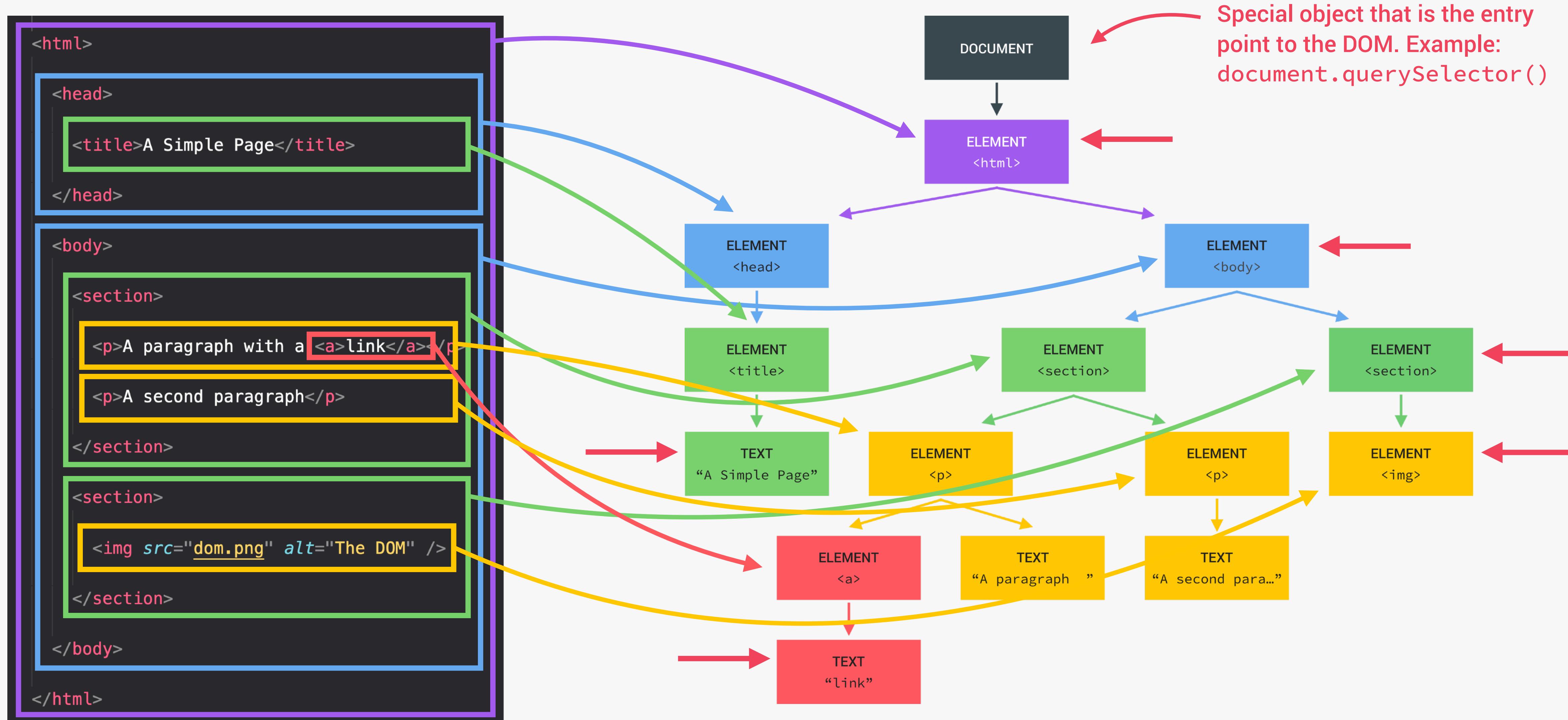


Tree structure, generated by browser on HTML load



Change text, HTML attributes, and even CSS styles

# THE DOM TREE STRUCTURE



# DOM != JAVASCRIPT



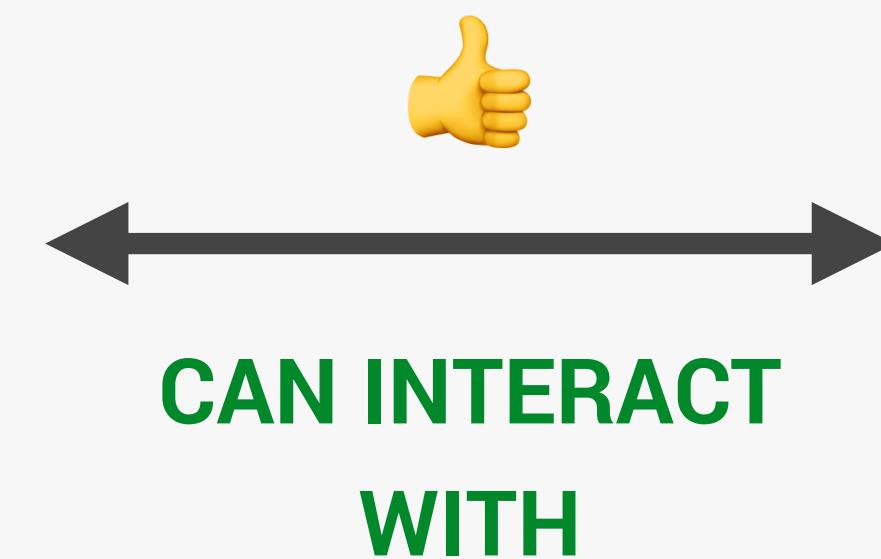
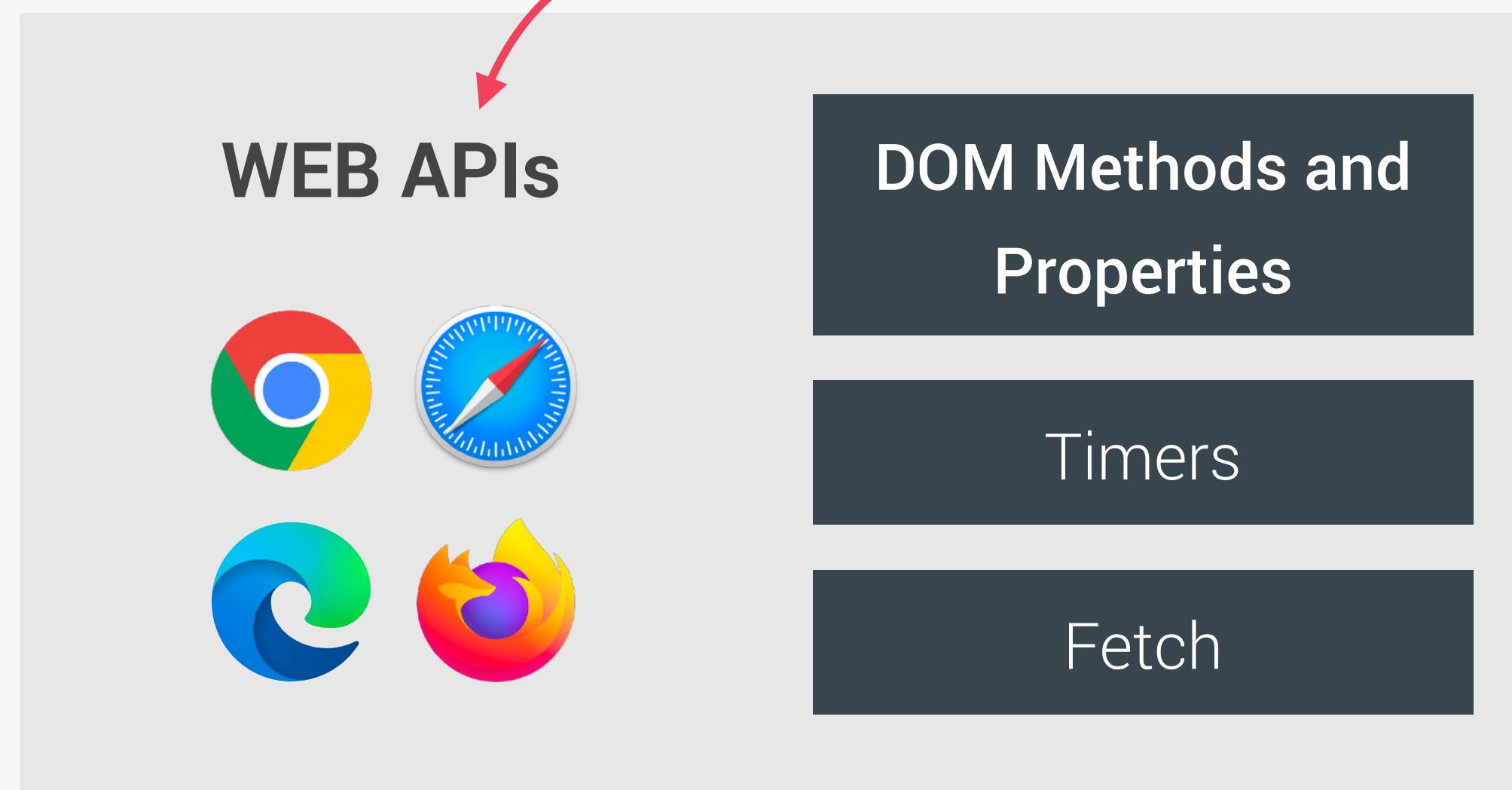
DOM Methods and  
Properties for DOM  
Manipulation



JS



For example  
`document.querySelector()`



JS



HOW JAVASCRIPT  
WORKS BEHIND THE  
SCENES



JONAS.IO  
SCHMEDTMANN

# THE COMPLETE JAVASCRIPT COURSE

FROM ZERO TO EXPERT!



@JONASSCHMEDTMAN

SECTION

HOW JAVASCRIPT WORKS BEHIND THE  
SCENES

LECTURE

AN HIGH-LEVEL OVERVIEW OF  
JAVASCRIPT

JS

# WHAT IS JAVASCRIPT: REVISITED

JAVASCRIPT

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

JS

# WHAT IS JAVASCRIPT: REVISITED

JAVASCRIPT

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



JS

# DECONSTRUCTING THE MONSTER DEFINITION

High-level

Garbage-collected

Interpreted or just-in-time compiled

Multi-paradigm

Prototype-based object-oriented

First-class functions

Dynamic

Single-threaded

Non-blocking event loop

# DECONSTRUCTING THE MONSTER DEFINITION

High-level

Garbage-collected

Interpreted or just-in-time compiled

Multi-paradigm

Prototype-based object-oriented

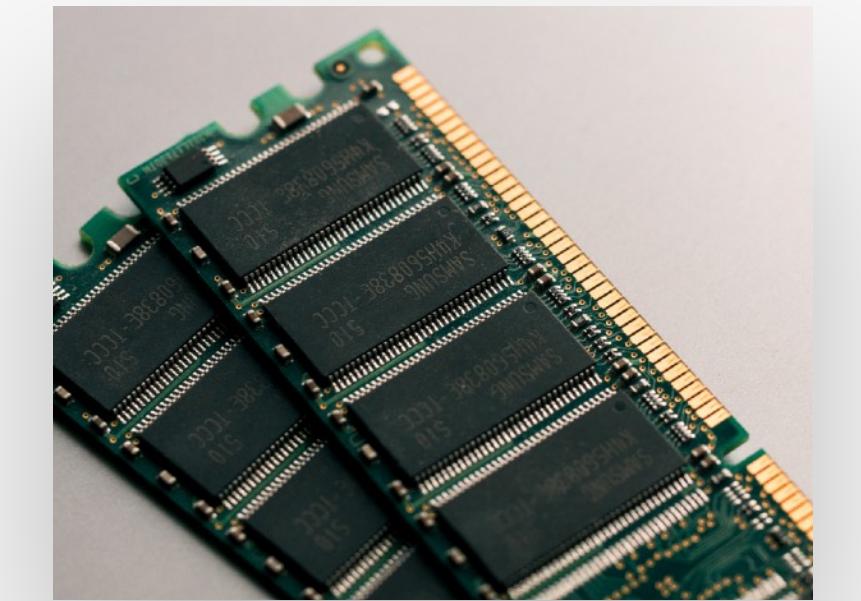
First-class functions

Dynamic

Single-threaded

Non-blocking event loop

👉 Any computer program needs resources:



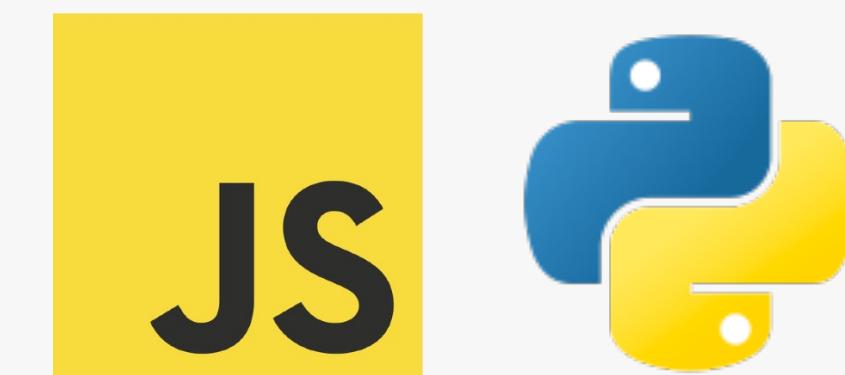
+



LOW-LEVEL



Developer has to manage  
resources manually



HIGH-LEVEL



Developer does NOT have  
to worry, everything  
happens automatically

# DECONSTRUCTING THE MONSTER DEFINITION

High-level

Garbage-collected

Interpreted or just-in-time compiled

Multi-paradigm

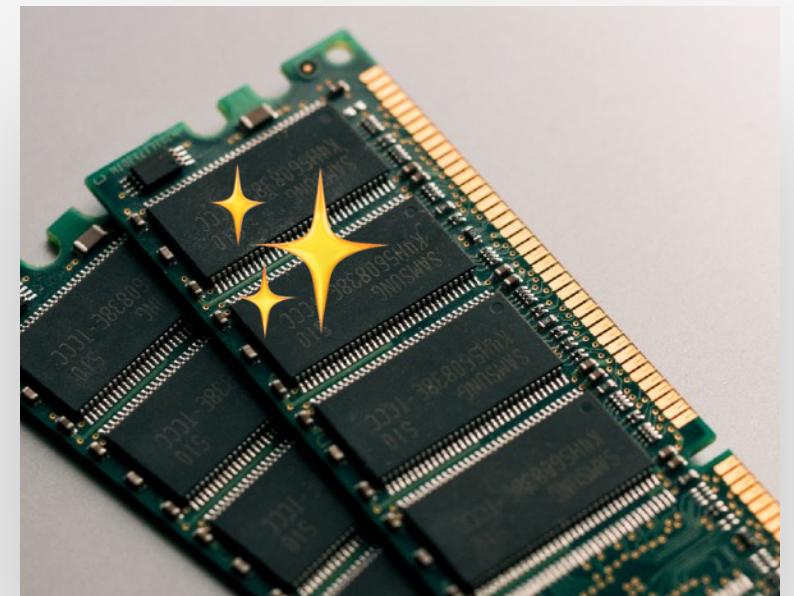
Prototype-based object-oriented

First-class functions

Dynamic

Single-threaded

Non-blocking event loop



Cleaning the memory  
so we don't have to

# DECONSTRUCTING THE MONSTER DEFINITION

High-level

Garbage-collected

Interpreted or just-in-time compiled

Multi-paradigm

Prototype-based object-oriented

First-class functions

Dynamic

Single-threaded

Non-blocking event loop

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

Abstraction over  
0s and 1s

CONVERT TO MACHINE CODE = COMPILED

```
11010110101110101011101101100101110101010111101010
01111010101110101001001110101110101011100010101100010
1010010011101110111100111000001110101011110111010
110100100001010010111010101101010111010101101010010
00001110100100100111101010111010101110010101111010
100101010010011110100111010010101010010101001011010100
100101010010001111010000101011100010100010101110101101
1110010001000111101000101011100010100010101110101101
010100101010001010100011101001001011101010010001010110
11101010010111010100010101110101010101010101010101001
1110101001011101010001010111010101010101010101010101001
011101010111010101000101011101010101010101011100111010
1110101001110101000111010101010101010101010101010101010
```

Happens inside the  
JavaScript engine

More about this **Later in this Section** 

# DECONSTRUCTING THE MONSTER DEFINITION

High-level

Garbage-collected

Interpreted or just-in-time compiled

Multi-paradigm

Prototype-based object-oriented

First-class functions

Dynamic

Single-threaded

Non-blocking event loop

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

The one we've been  
using so far

1

Procedural programming

2

Object-oriented programming (OOP)

3

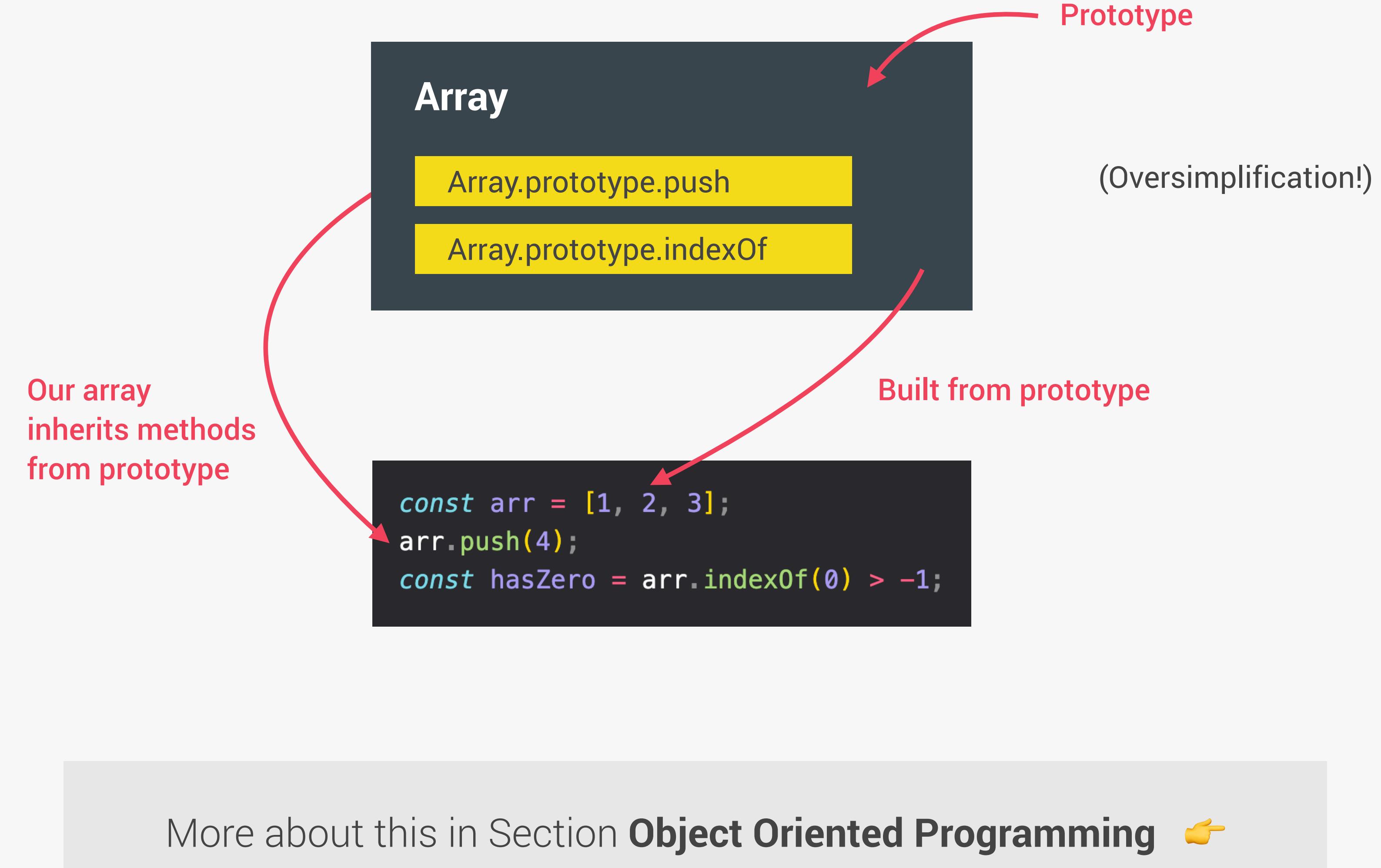
Functional programming (FP)

👉 Imperative vs.  
👉 Declarative

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

# DECONSTRUCTING THE MONSTER DEFINITION

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



# DECONSTRUCTING THE MONSTER DEFINITION

High-level

Garbage-collected

Interpreted or just-in-time compiled

Multi-paradigm

Prototype-based object-oriented

First-class functions

Dynamic

Single-threaded

Non-blocking event loop

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

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

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

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

# DECONSTRUCTING THE MONSTER DEFINITION

High-level

Garbage-collected

Interpreted or just-in-time compiled

Multi-paradigm

Prototype-based object-oriented

First-class functions

Dynamic

Single-threaded

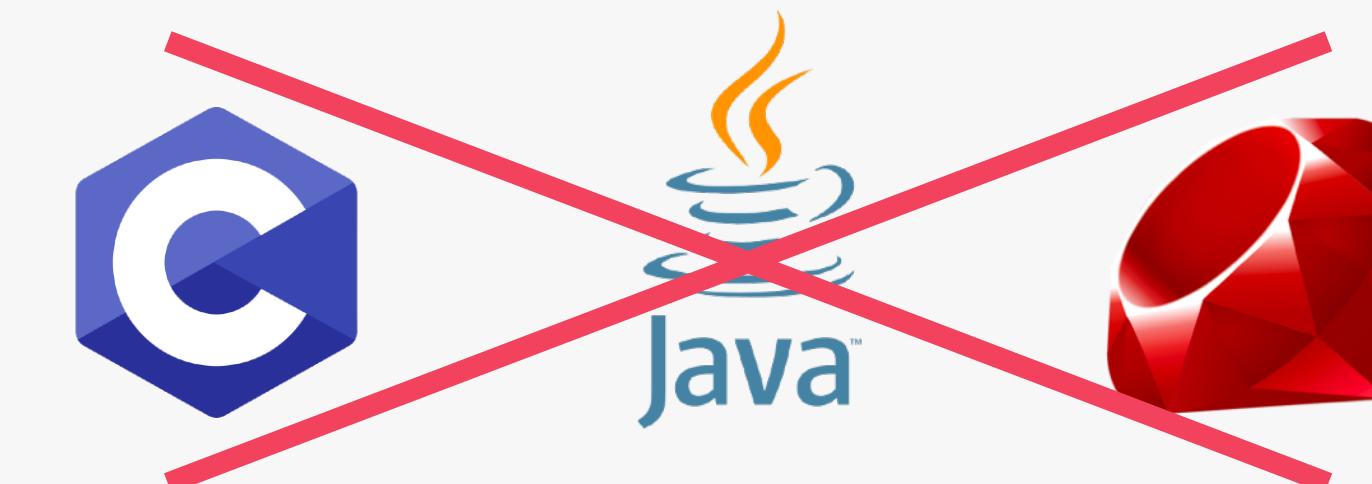
Non-blocking event loop

👉 **Dynamically-typed language:**

No data type definitions. Types becomes known at runtime

Data type of variable is automatically changed

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



TS

# DECONSTRUCTING THE MONSTER DEFINITION

High-level

Garbage-collected

Interpreted or just-in-time compiled

Multi-paradigm

Prototype-based object-oriented

First-class functions

Dynamic

Single-threaded

Non-blocking event loop

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



Why do we need that?

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



So what about a long-running task?

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



How do we achieve that?

(Oversimplification!)

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

More about this **Later in this Section**







JONAS.IO  
SCHMEDTMANN

# THE COMPLETE JAVASCRIPT COURSE

FROM ZERO TO EXPERT!



@JONASSCHMEDTMAN

SECTION

HOW JAVASCRIPT WORKS BEHIND THE  
SCENES

LECTURE

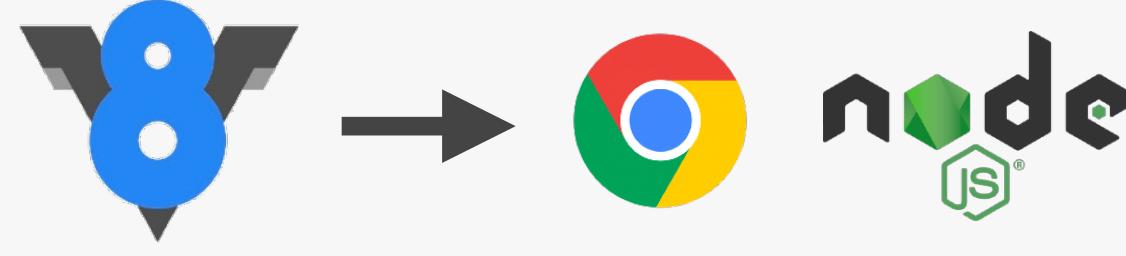
THE JAVASCRIPT ENGINE AND  
RUNTIME

JS

# WHAT IS A JAVASCRIPT ENGINE?

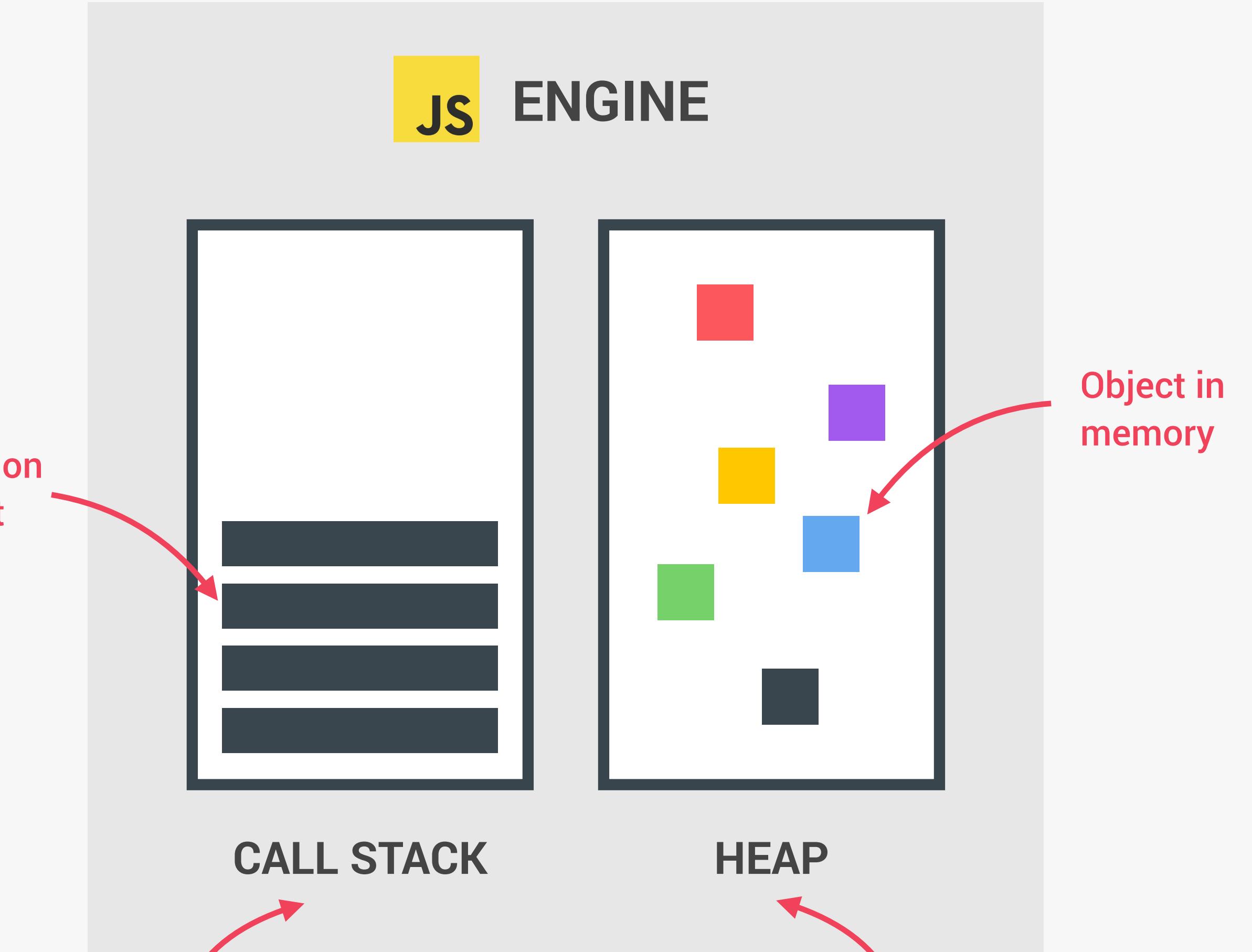


👉 Example: V8 Engine



How is it compiled?

Where our code  
is executed



Where objects  
are stored

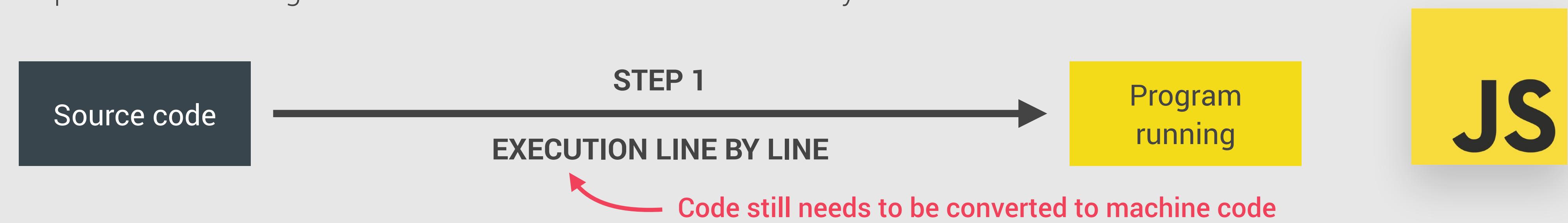
# COMPUTER SCIENCE SIDENOTE: COMPIRATION VS. INTERPRETATION



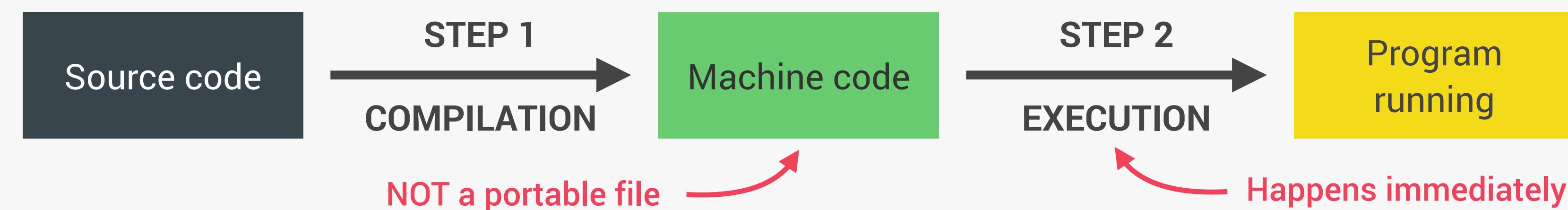
- 👉 **Compilation:** Entire code is converted into machine code at once, and written to a binary file that can be executed by a computer.



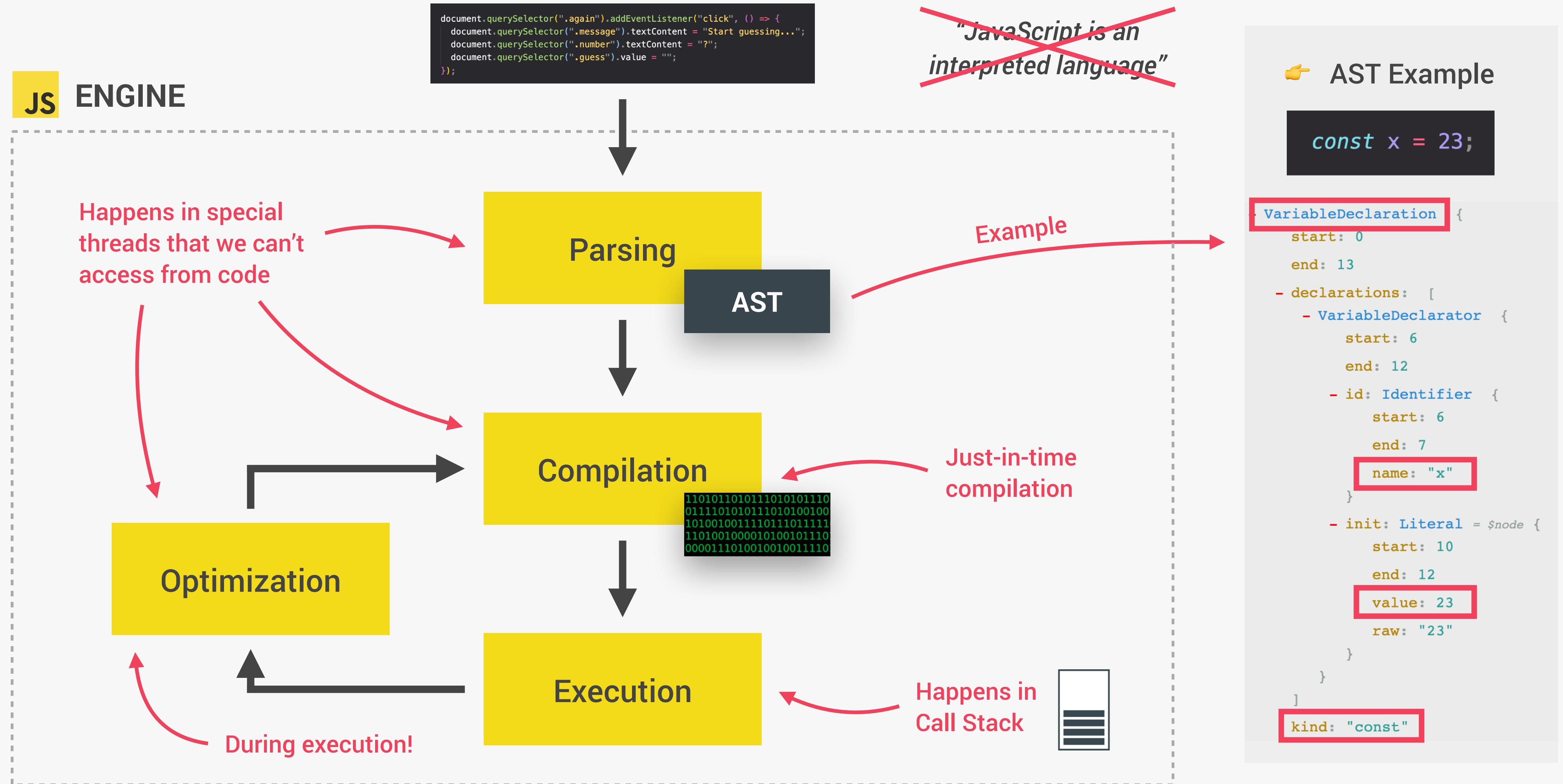
- 👉 **Interpretation:** Interpreter runs through the source code and executes it line by line.



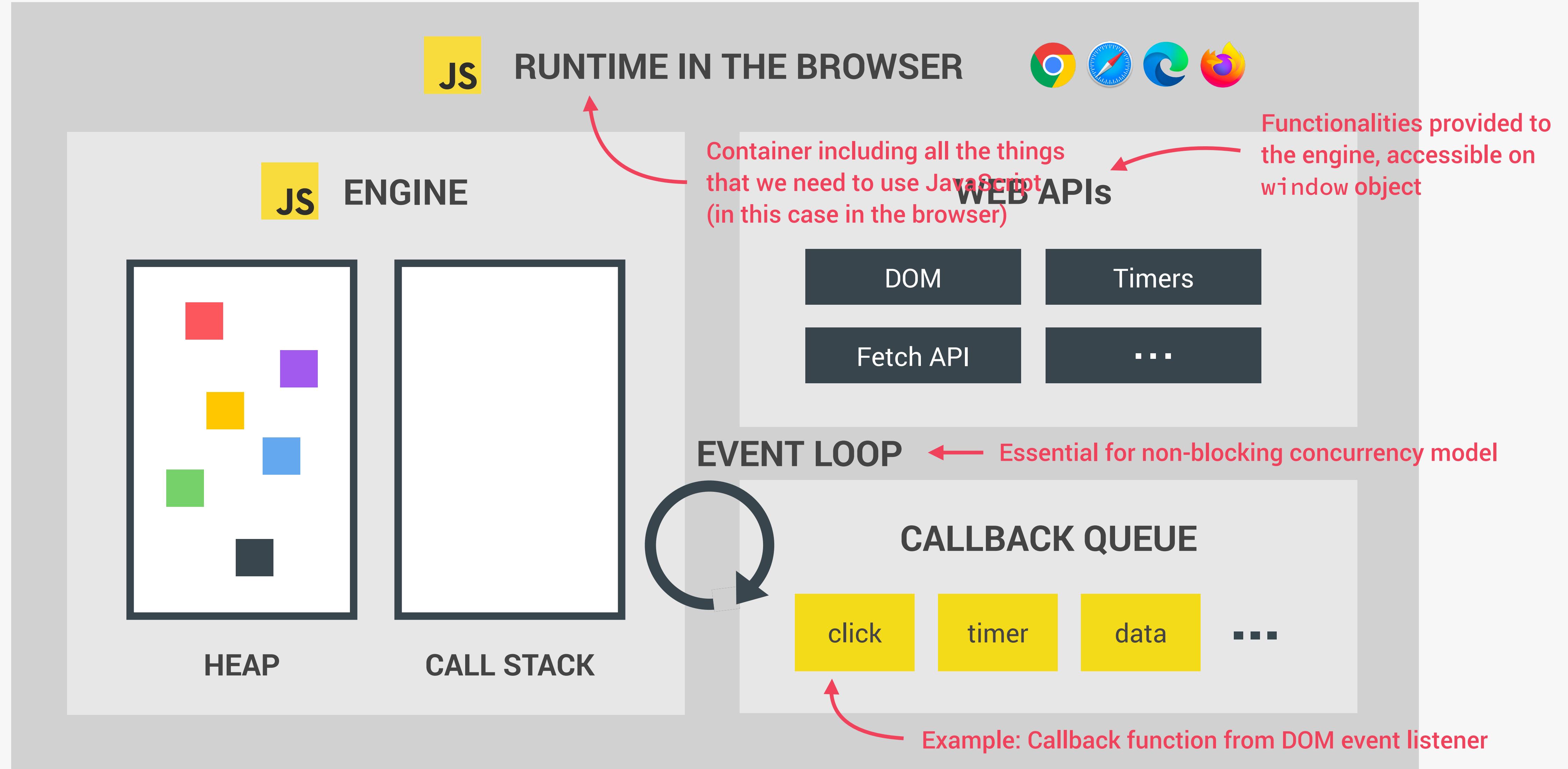
- 👉 **Just-in-time (JIT) compilation:** Entire code is converted into machine code at once, then executed immediately.



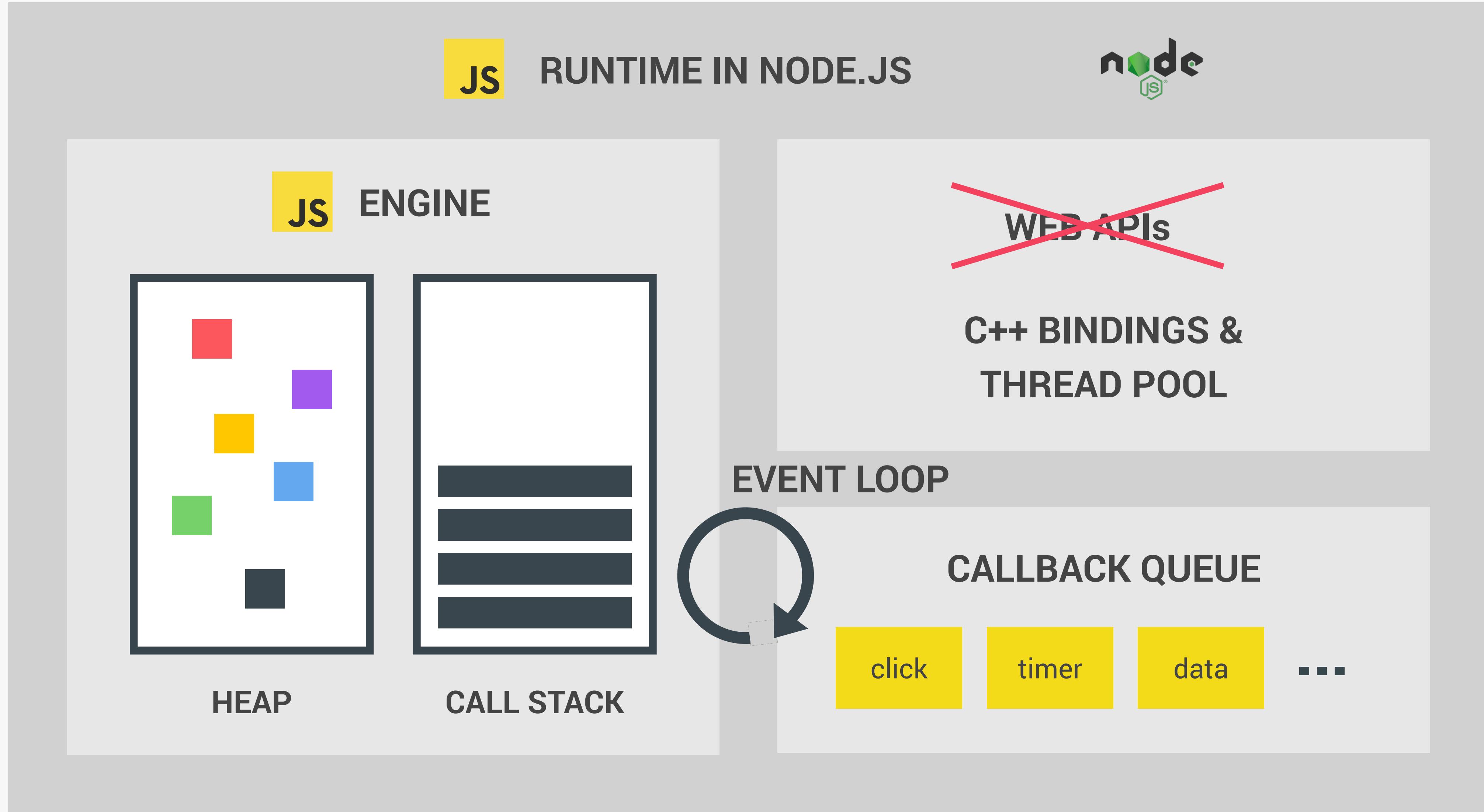
# MODERN JUST-IN-TIME COMPIRATION OF JAVASCRIPT



# THE BIGGER PICTURE: JAVASCRIPT RUNTIME



# THE BIGGER PICTURE: JAVASCRIPT RUNTIME







JONAS.IO  
SCHMEDTMANN

# THE COMPLETE JAVASCRIPT COURSE

FROM ZERO TO EXPERT!



@JONASSCHMEDTMAN

SECTION

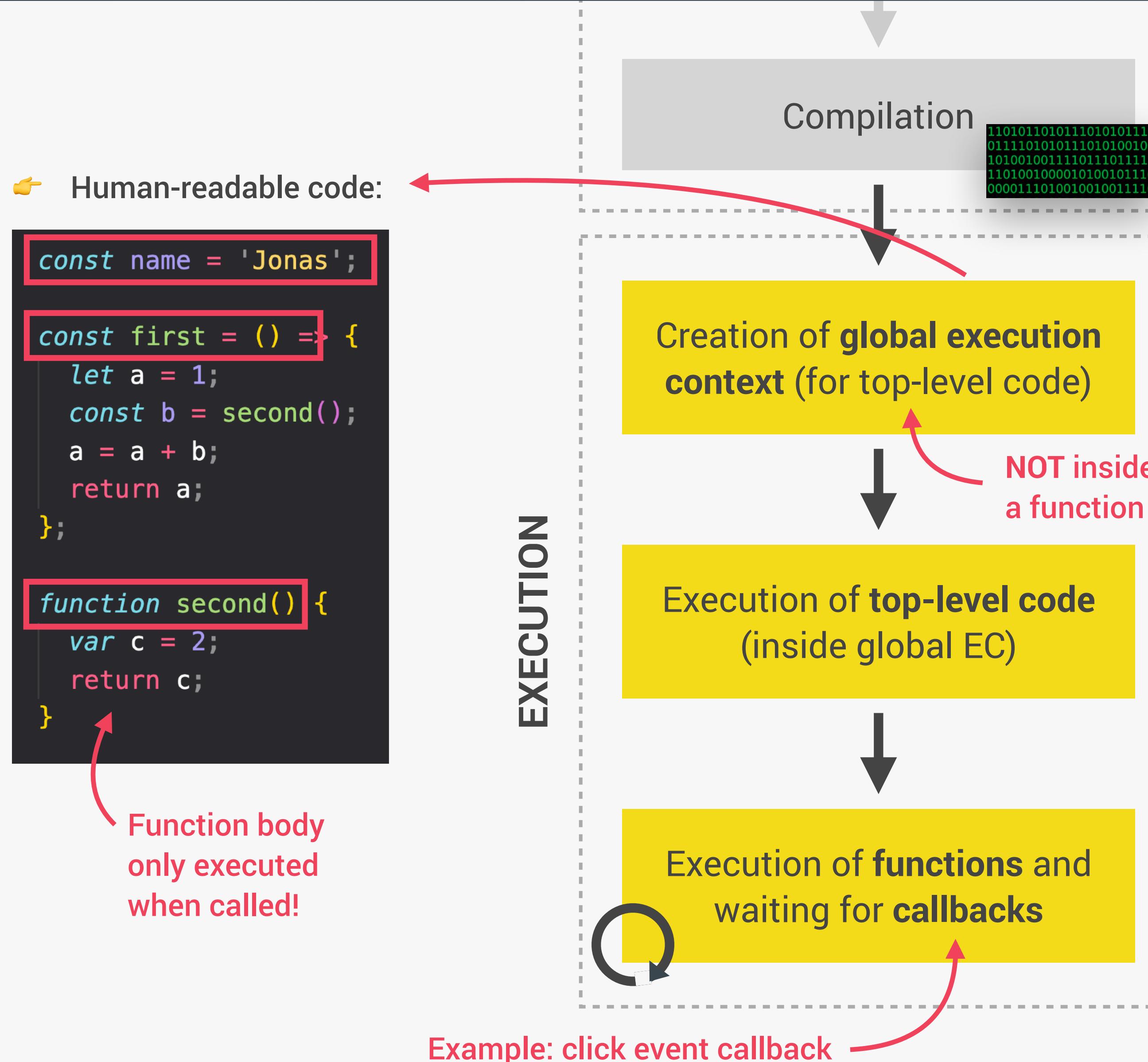
HOW JAVASCRIPT WORKS BEHIND THE  
SCENES

LECTURE

EXECUTION CONTEXTS AND THE  
CALL STACK

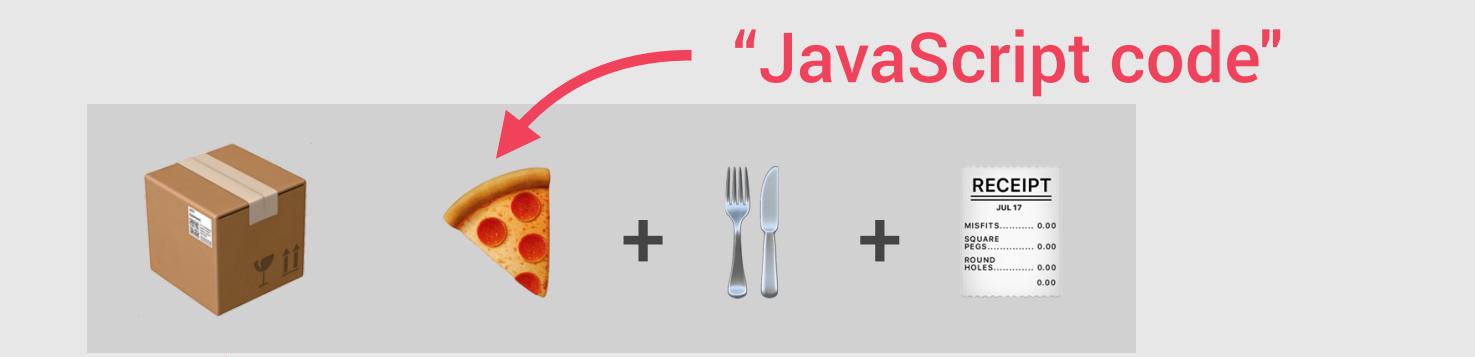
JS

# WHAT IS AN EXECUTION CONTEXT?



## EXECUTION CONTEXT

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



- 👉 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

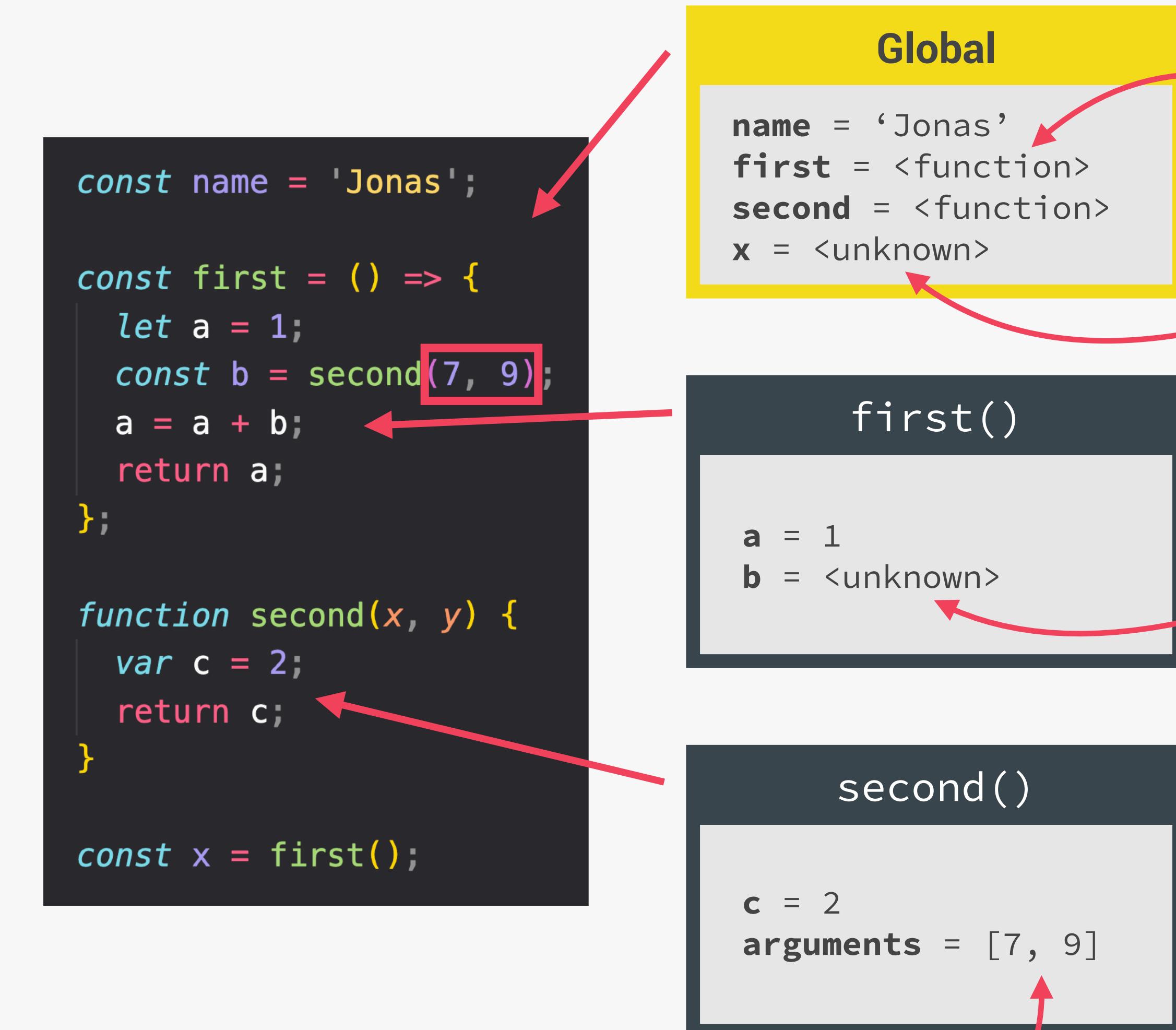
- 👉 let, const and var declarations
- 👉 Functions
- 👉 ~~arguments~~ object

### 2 Scope chain

### 3 ~~this~~ keyword

NOT in arrow functions!

Generated during “creation phase”, right before execution



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

(Technically, values only become known during execution)

Literally the function code

Need to run first() first

Need to run second() first

# THE CALL STACK

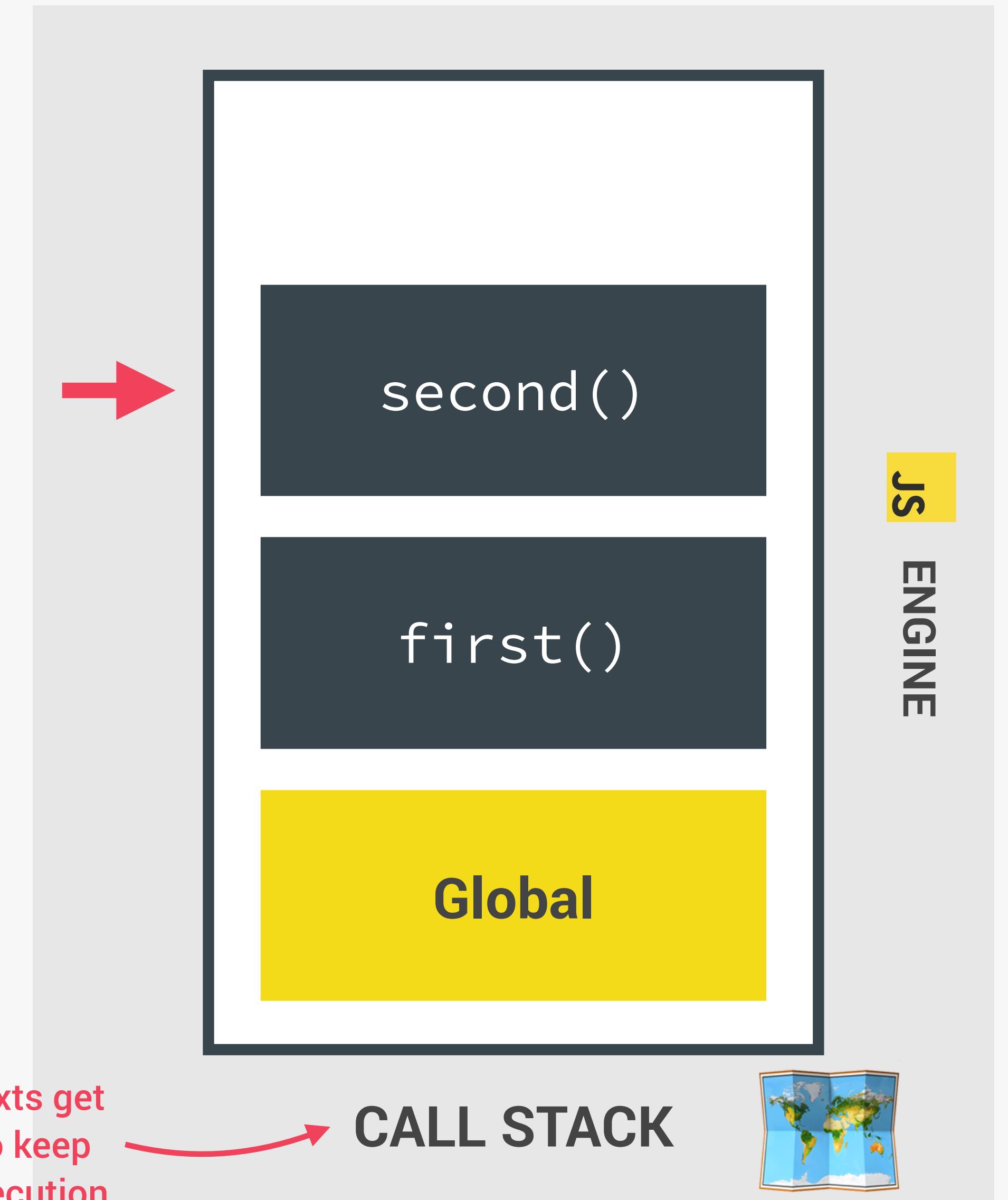
👉 Compiled code starts execution

```
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();
```







JONAS.IO  
SCHMEDTMANN

# THE COMPLETE JAVASCRIPT COURSE

FROM ZERO TO EXPERT!



@JONASSCHMEDTMAN

SECTION

HOW JAVASCRIPT WORKS BEHIND THE  
SCENES

LECTURE

SCOPE AND THE SCOPE CHAIN

JS

# SCOPING AND SCOPE IN JAVASCRIPT: CONCEPTS

## SCOPE CONCEPTS

### EXECUTION CONTEXT

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

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

# THE 3 TYPES OF SCOPE

## GLOBAL SCOPE

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

## FUNCTION SCOPE

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

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

## BLOCK SCOPE (ES6)

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

} ← Example: if block, for loop block, etc.

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

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

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

- 👉 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';

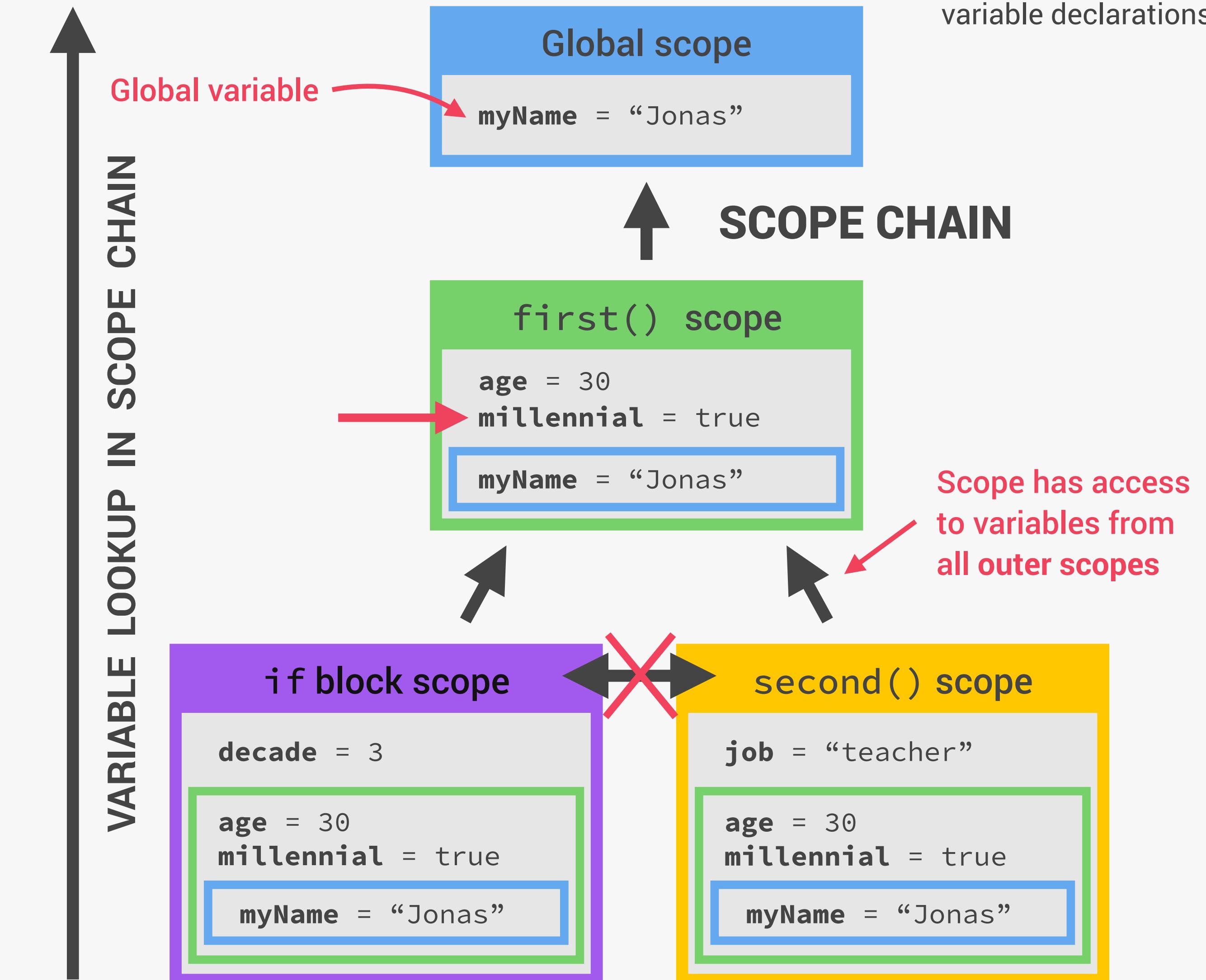
function first() {
  const age = 30;
  let and const are block-scoped
  if (age >= 30) { // true
    const decade = 3;
    var millennial = true;
  }
  var is function-scoped
  function second() {
    const job = 'teacher';
    console.log(`[myName] is a ${age}-old ${job}`);
    // Jonas is a 30-old teacher
  }
  second();
}

first();
```

Variables not in current scope

let and const are **block-scoped**

var is **function-scoped**



# SCOPE CHAIN VS. CALL STACK

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

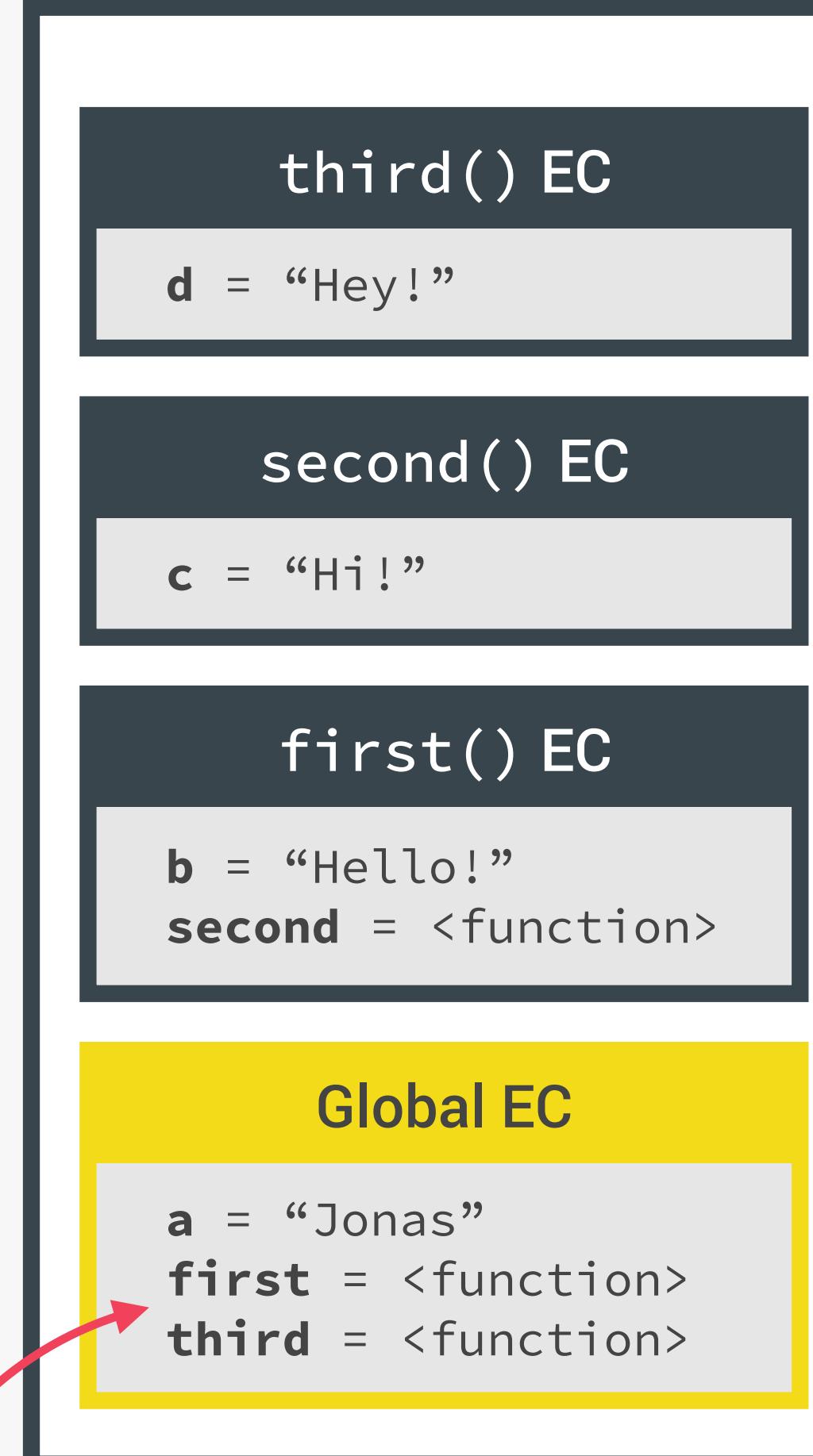
function first() {
  const b = 'Hello!';
  second();

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

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

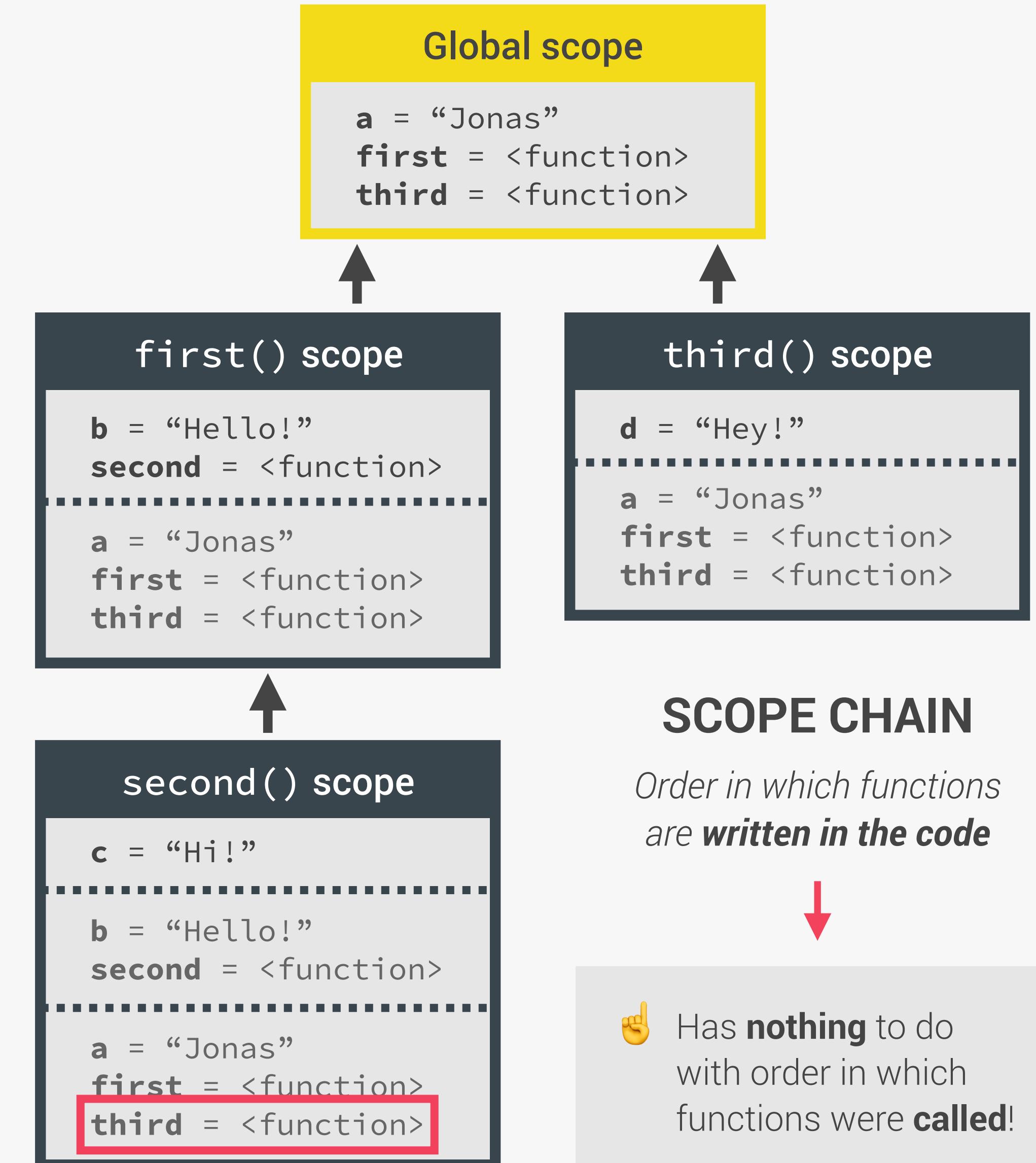
c and b can NOT be found  
in third() scope!

Variable  
environment (VE)



## CALL STACK

Order in which  
functions were **called**



# SUMMARY



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





JONAS.IO  
SCHMEDTMANN

# THE COMPLETE JAVASCRIPT COURSE

FROM ZERO TO EXPERT!



@JONASSCHMEDTMAN

SECTION

HOW JAVASCRIPT WORKS BEHIND THE  
SCENES

LECTURE

VARIABLE ENVIRONMENT: HOISTING  
AND THE TDZ

JS

# HOISTING IN JAVASCRIPT

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

↓ **BEHIND THE SCENES**

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

## EXECUTION CONTEXT

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

	HOISTED?	INITIAL VALUE	SCOPE
function declarations	✓ YES	Actual function	Block
var variables	✓ YES	undefined	Function
let and const variables	✗ NO Technically, yes. But not in practice	<uninitialized>, TDZ	Block
function expressions and arrows	✗ Depends if using var or let/const		Temporal Dead Zone

# TEMPORAL DEAD ZONE, LET AND CONST

```
const myName = 'Jonas';

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

## TEMPORAL DEAD ZONE FOR **job** VARIABLE

- 👉 Different kinds of error messages:

ReferenceError: Cannot access 'job' before initialization

ReferenceError: x is not defined

## WHY HOISTING?

- 👉 Using functions before actual declaration;
- 👉 var hoisting is just a byproduct.

## WHY TDZ?

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





JONAS.IO  
SCHMEDTMANN

# THE COMPLETE JAVASCRIPT COURSE

FROM ZERO TO EXPERT!



@JONASSCHMEDTMAN

SECTION

HOW JAVASCRIPT WORKS BEHIND THE  
SCENES

LECTURE

THE THIS KEYWORD

JS

# HOW THE THIS KEYWORD WORKS

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

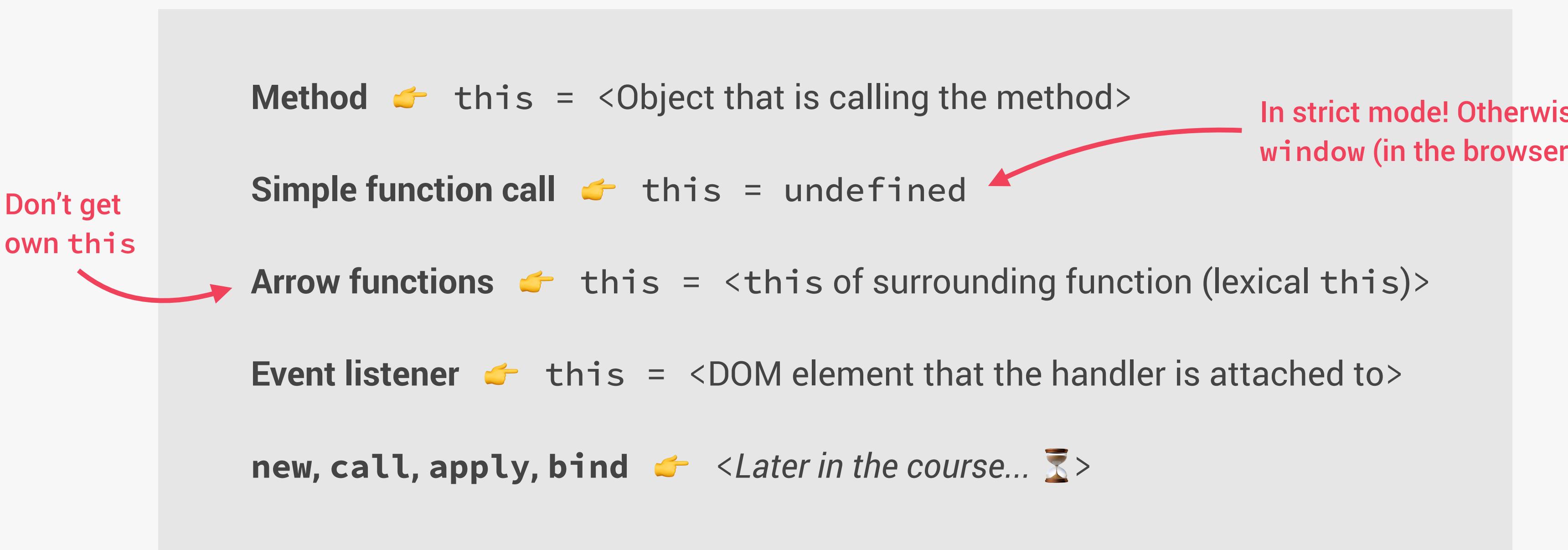
👉 **this** is **NOT** static. It depends on **how** the function is called, and its value is only assigned when the function **is actually called**.

## EXECUTION CONTEXT

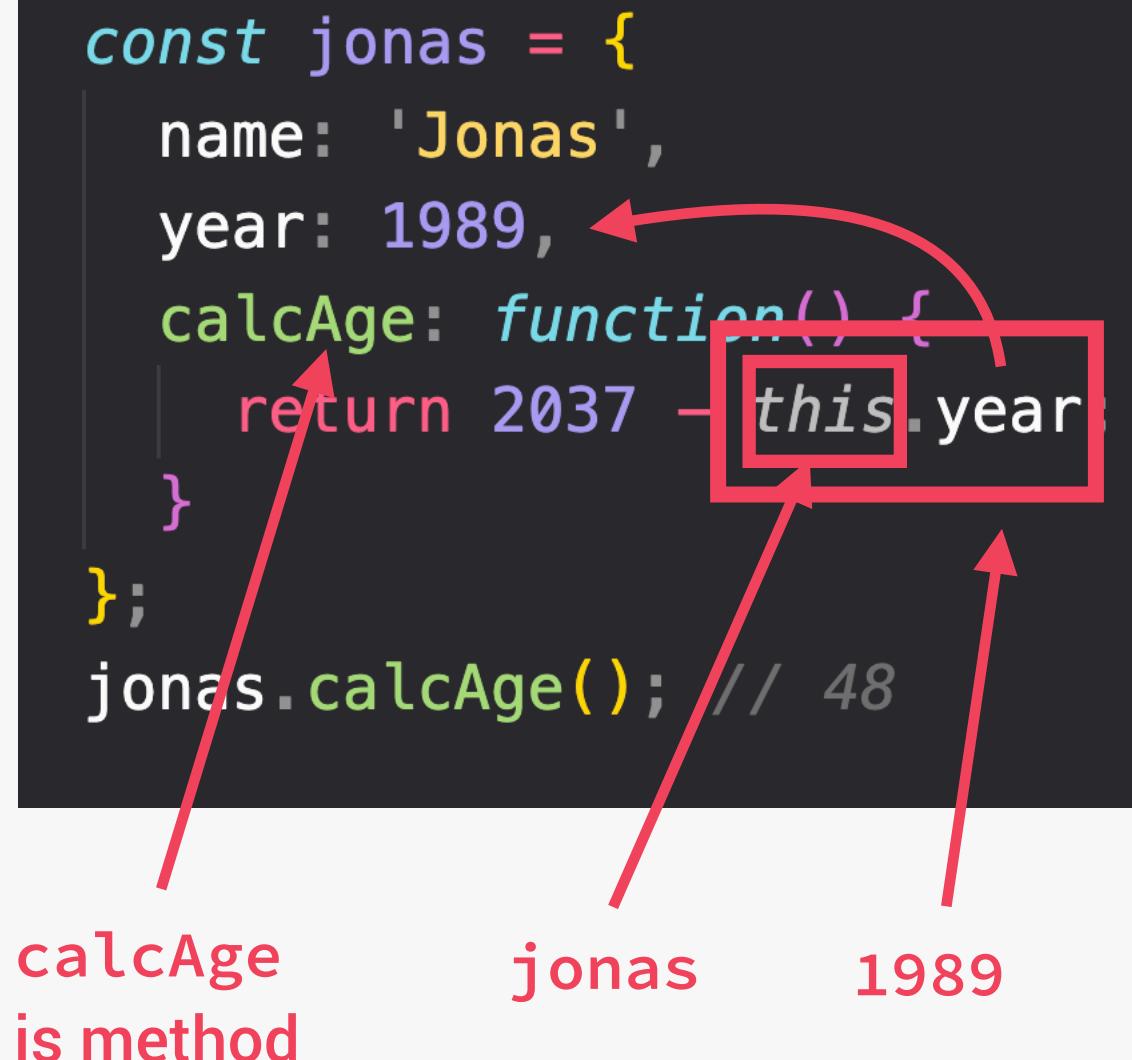
✓ Variable environment

✓ Scope chain

👉 **this keyword**



## 👉 Method example:



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

Way better than using  
**jonas.year!**



JONAS.IO  
SCHMEDTMANN

# THE COMPLETE JAVASCRIPT COURSE

FROM ZERO TO EXPERT!



@JONASSCHMEDTMAN

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

JS





# THE COMPLETE JAVASCRIPT COURSE

FROM ZERO TO EXPERT!



@JONASSCHMEDTMAN

SECTION

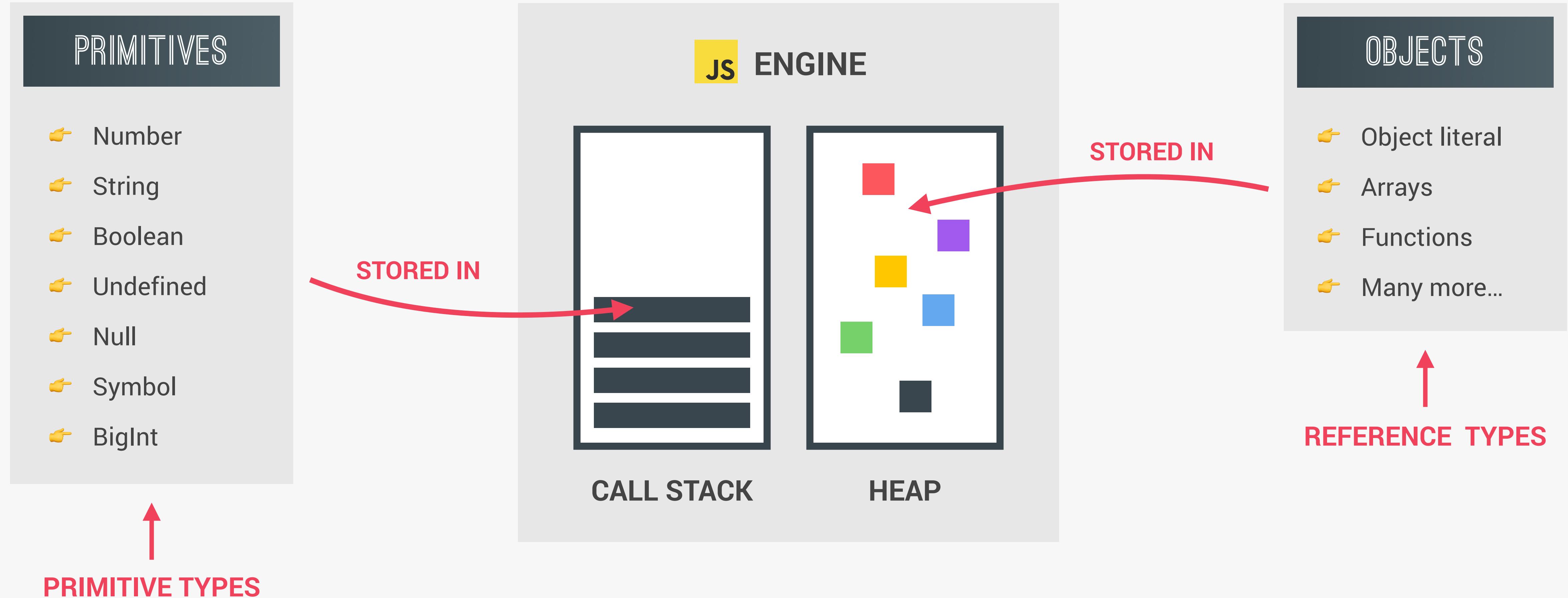
HOW JAVASCRIPT WORKS BEHIND THE  
SCENES

LECTURE

PRIMITIVES VS. OBJECTS (PRIMITIVE  
VS. REFERENCE TYPES)

JS

# REVIEW: PRIMITIVES, OBJECTS AND THE JAVASCRIPT ENGINE



# PRIMITIVE VS. REFERENCE VALUES

👉 Primitive values example:

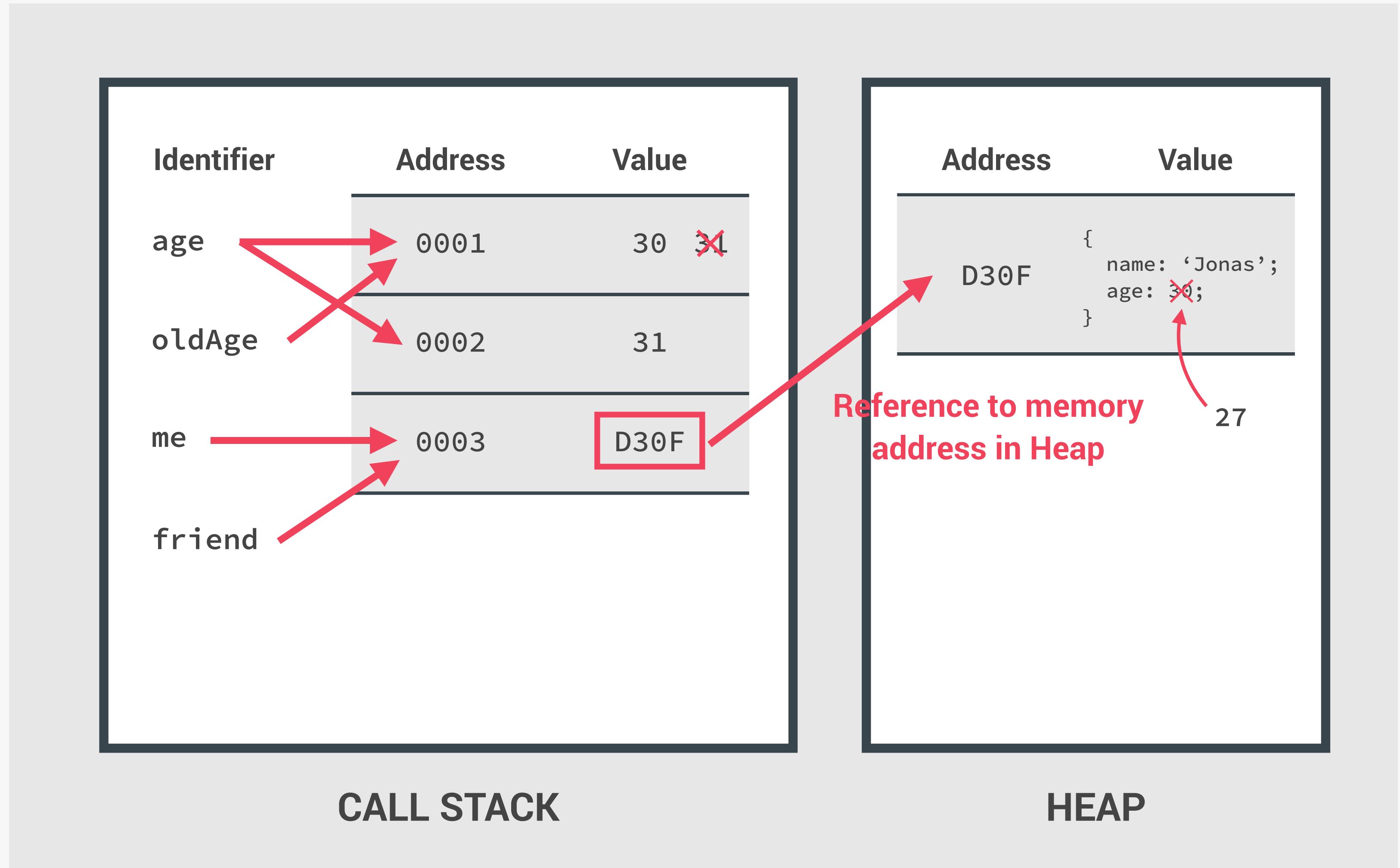
```
let age = 30;
let oldAge = age;
age = 31;
console.log(age); // 31
console.log(oldAge); // 30
```

👉 Reference values example:

```
const me = {
  name: 'Jonas'      No problem, because
  age: 30            we're NOT changing the
};                   value at address 0003!
const friend = me;
friend.age = 27;

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

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



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



1

**Prototypal Inheritance** ➡ Object Oriented Programming (OOP) With JavaScript

2

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

3

**How the DOM Really Works** ➡ Advanced DOM and Events



DATA STRUCTURES,  
MODERN OPERATORS  
AND STRINGS



# THE COMPLETE JAVASCRIPT COURSE

FROM ZERO TO EXPERT!



@JONASSCHMEDTMAN

SECTION

DATA STRUCTURES, MODERN  
OPERATORS AND STRINGS

LECTURE

SUMMARY: WHICH DATA STRUCTURE  
TO USE?

JS

# DATA STRUCTURES OVERVIEW

## SOURCES OF DATA

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



Collection of data



Data structure

SIMPLE LIST?

Arrays or Sets

KEY/VALUE PAIRS?

Objects or Maps

## OTHER BUILT-IN:

- 👉 WeakMap
- 👉 WeakSet

## NON-BUILT IN:

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

Application  
Programming  
Interface

Keys allow us to  
describe values

The diagram shows a JSON object structure with three levels of nesting. The first level is an object with keys "count" and "recipes". The "recipes" key points to an array containing three objects. Each object at this level has keys for publisher, title, source\_url, recipe\_id, image\_url, social\_rank, and publisher\_url. Red arrows point from the text labels to the corresponding JSON elements: "Object" points to the outermost {}, "Array" points to the [ ], and another "Object" points to one of the inner {} blocks.

```
{
  "count": 3,
  "recipes": [
    {
      "publisher": "101 Cookbooks",
      "title": "Best Pizza Dough Ever",
      "source_url": "http://www.101cookbooks.com/archiv",
      "recipe_id": "47746",
      "image_url": "http://forkify-api.herokuapp.com/im",
      "social_rank": 100,
      "publisher_url": "http://www.101cookbooks.com"
    },
    {
      "publisher": "The Pioneer Woman",
      "title": "Deep Dish Fruit Pizza",
      "source_url": "http://thepioneerwoman.com/cooking",
      "recipe_id": "46956",
      "image_url": "http://forkify-api.herokuapp.com/im",
      "social_rank": 100,
      "publisher_url": "http://thepioneerwoman.com"
    },
    {
      "publisher": "Closet Cooking",
      "title": "Pizza Dip",
      "source_url": "http://www.closetcooking.com/2011/",
      "recipe_id": "35477",
      "image_url": "http://forkify-api.herokuapp.com/im",
      "social_rank": 99.999999999994,
      "publisher_url": "http://closetcooking.com"
    }
  ]
}
```

👉 JSON data format example

# 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



# A CLOSER LOOK AT FUNCTIONS



# THE COMPLETE JAVASCRIPT COURSE

FROM ZERO TO EXPERT!



@JONASSCHMEDTMAN

SECTION

A CLOSER LOOK AT FUNCTIONS

LECTURE

FIRST-CLASS AND HIGHER-ORDER  
FUNCTIONS

JS

# FIRST-CLASS VS. HIGHER-ORDER FUNCTIONS

## FIRST-CLASS FUNCTIONS

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

- 👉 Store functions in variables or properties:

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

- 👉 Pass functions as arguments to OTHER functions:

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

- 👉 Return functions FROM functions

- 👉 Call methods on functions:

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

## HIGHER-ORDER FUNCTIONS

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

- 1 Function that receives another function

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

Higher-order  
function

Callback  
function



- 2 Function that returns new function

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

Higher-order  
function

Returned  
function





JONAS.IO  
SCHMEDTMANN

# THE COMPLETE JAVASCRIPT COURSE

FROM ZERO TO EXPERT!



@JONASSCHMEDTMAN

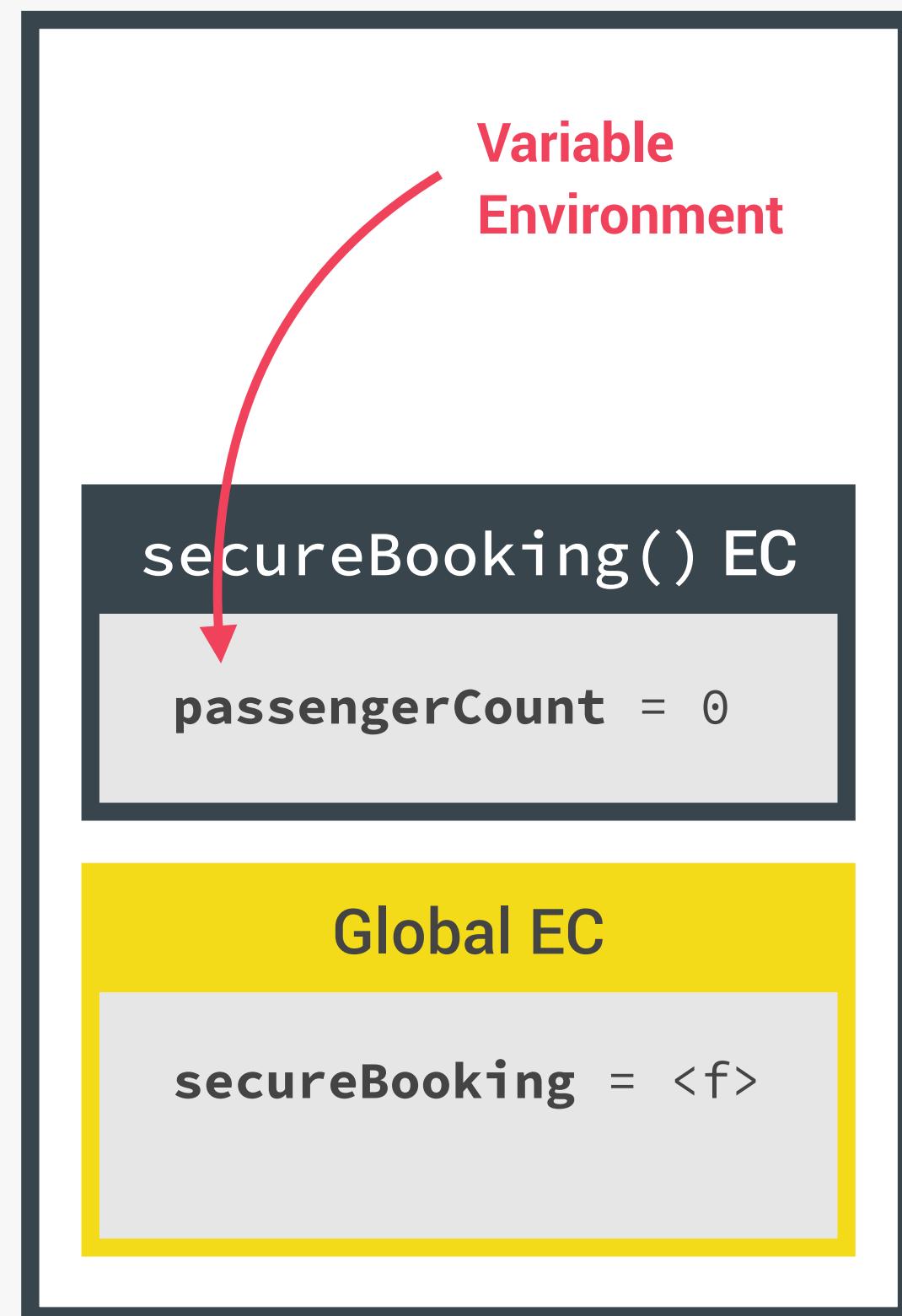
SECTION

A CLOSER LOOK AT FUNCTIONS

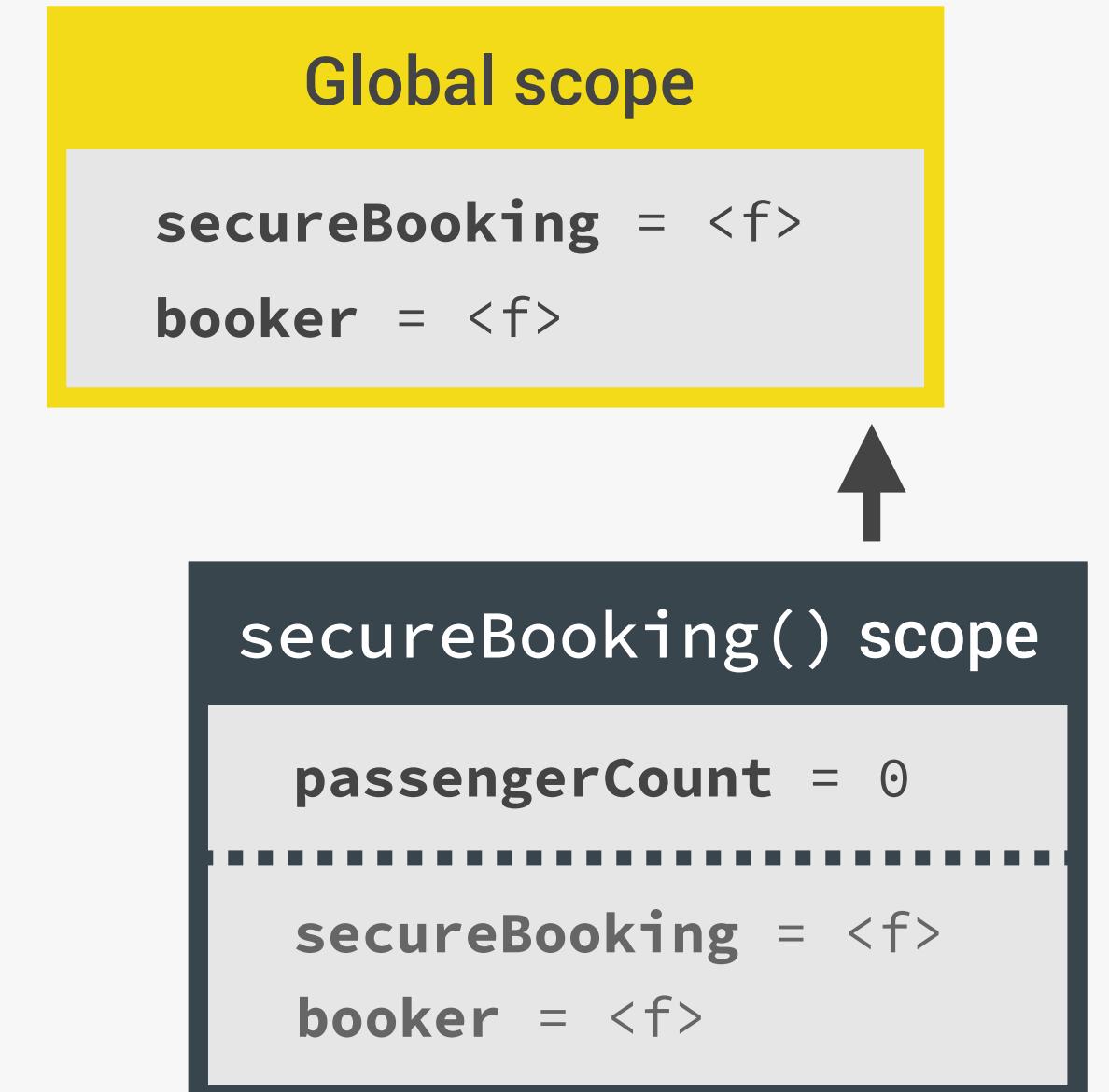
LECTURE  
CLOSURES

JS

# "CREATING" A CLOSURE



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



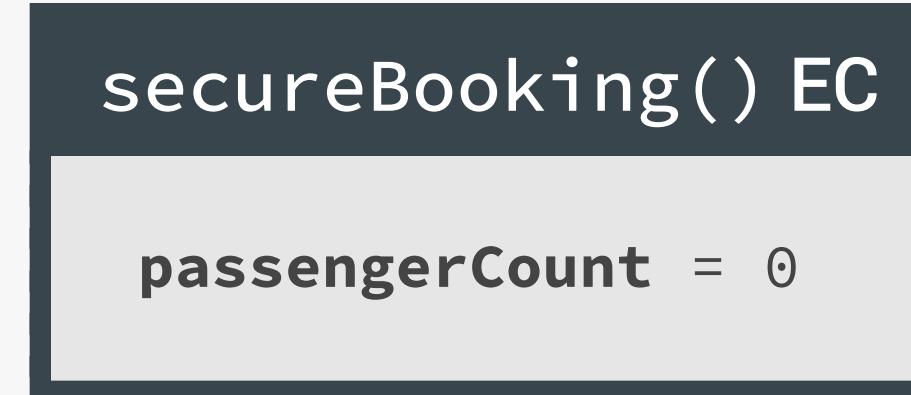
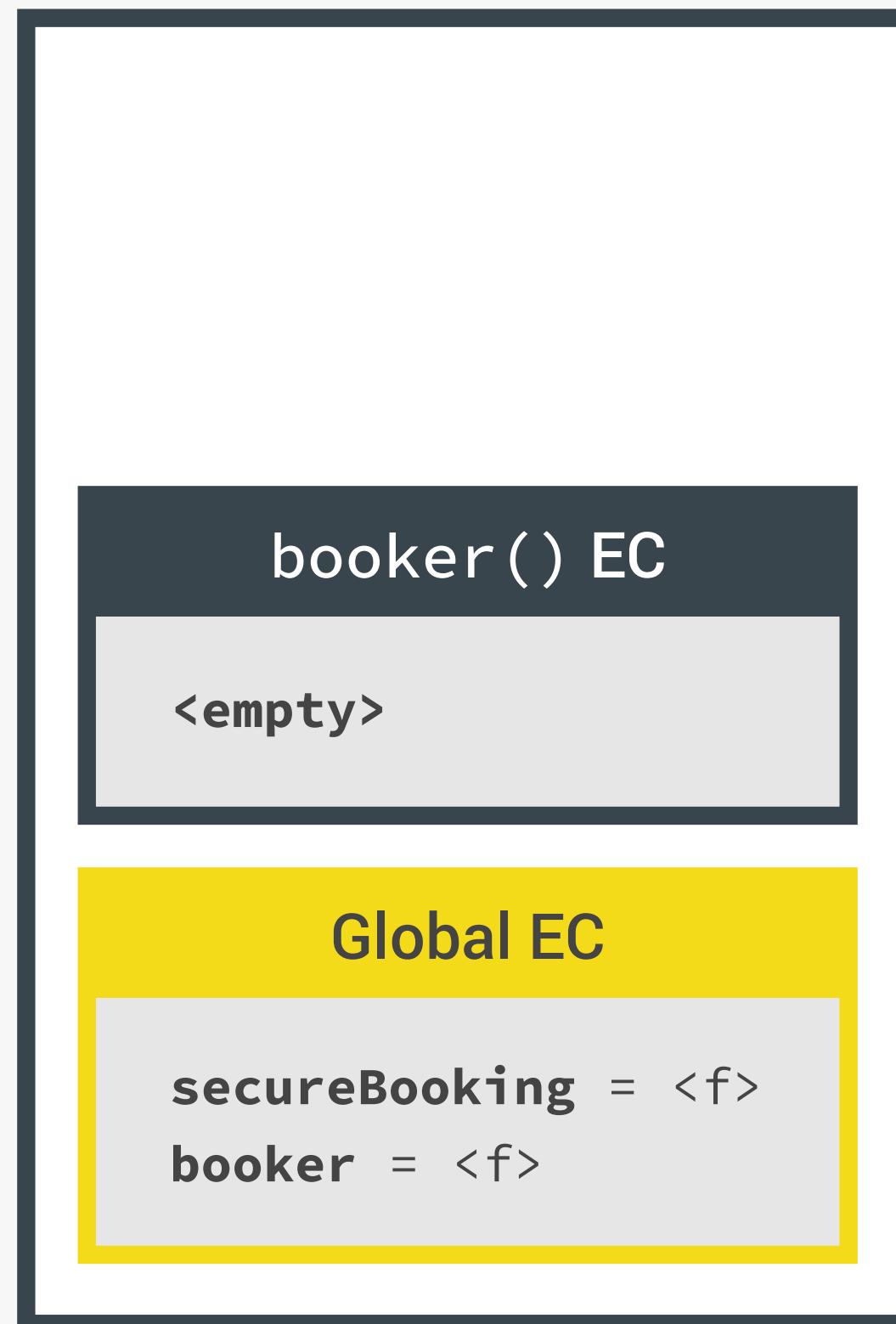
CALL STACK

Order in which  
functions were *called*

Order in which functions  
are *written in the code*

SCOPE CHAIN

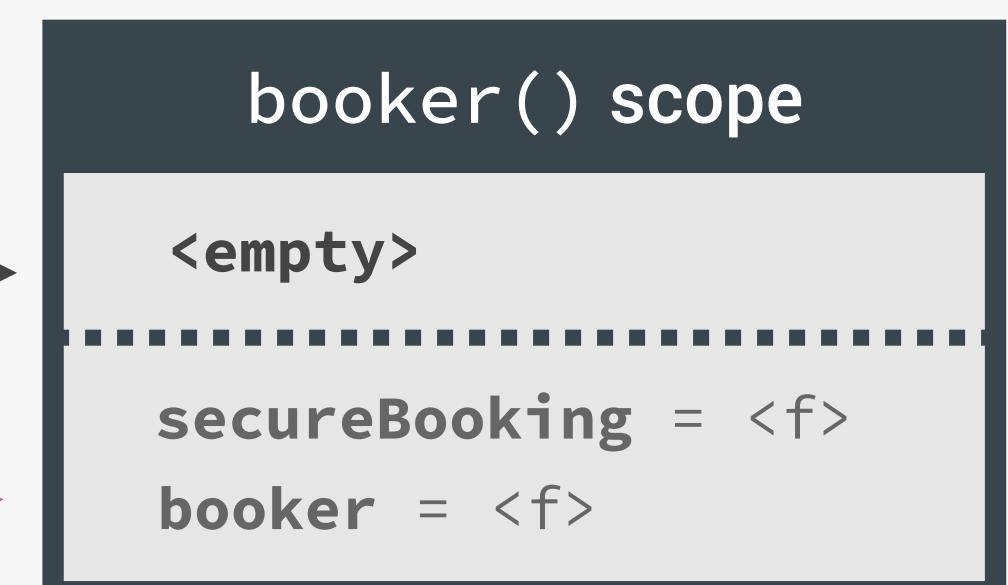
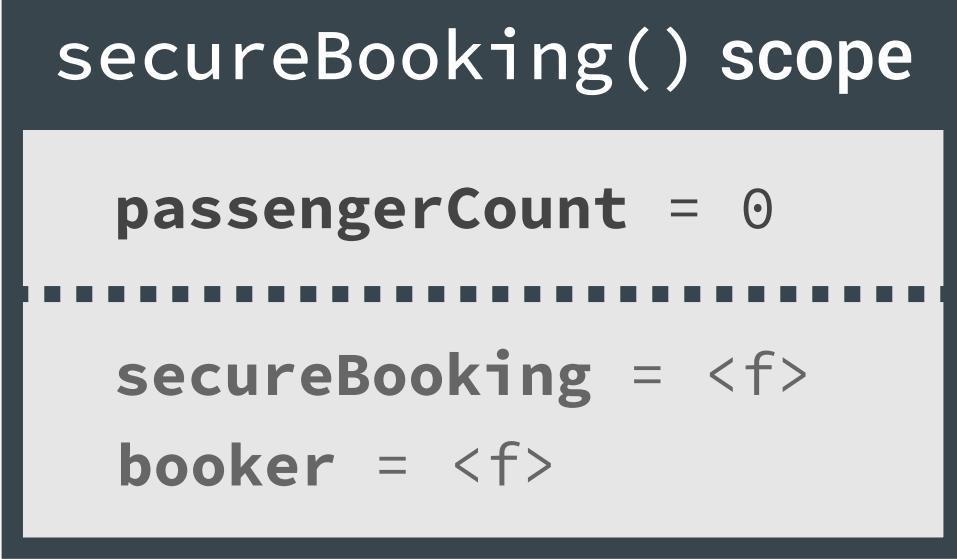
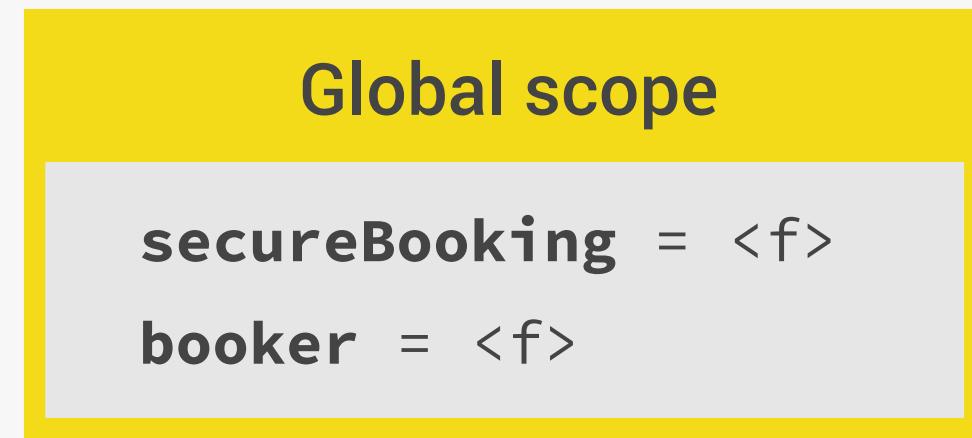
# UNDERSTANDING CLOSURES



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



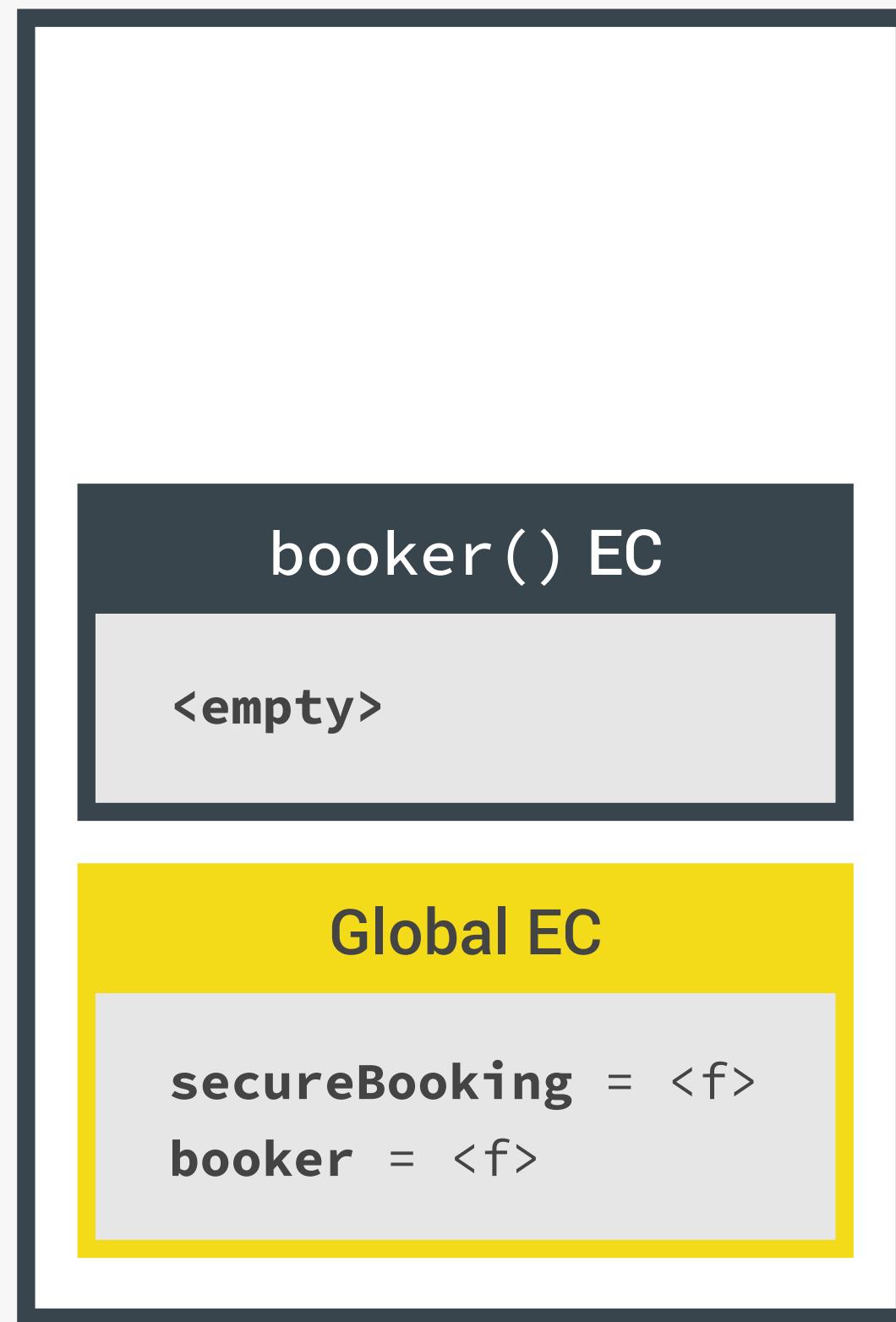
How to access  
passengerCount?

CALL STACK

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



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

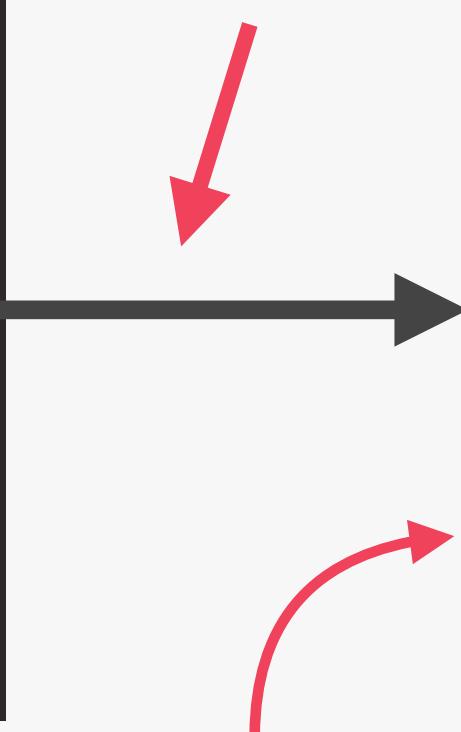
  return function () {
    passengerCount++;
    console.log(`#${passengerCount} passengers`);
  };
};

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

This is the function

The code defines a closure where the inner function `booker()` has access to the `passengerCount` variable from its outer scope. The first call to `booker()` logs "1 passengers" and increments the count to 1. The second call logs "2 passengers" and increments the count to 2.

(Priority over  
scope chain)  
**CLOSURE**



How to access  
passengerCount?

**CALL STACK**

**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;

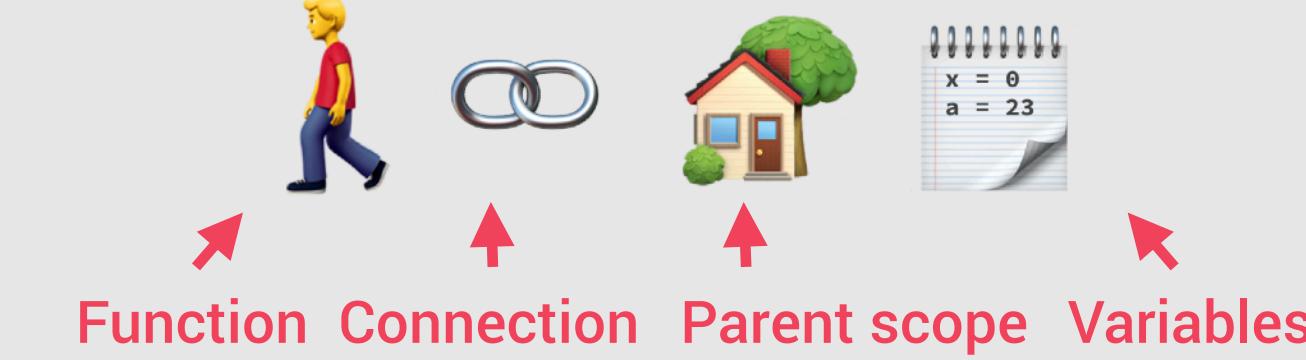
↓ Less formal

- 👉 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.



# WORKING WITH ARRAYS



# THE COMPLETE JAVASCRIPT COURSE

FROM ZERO TO EXPERT!



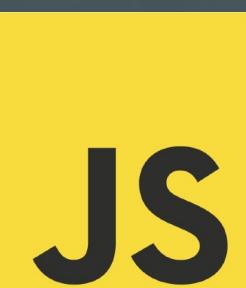
@JONASSCHMEDTMAN

SECTION

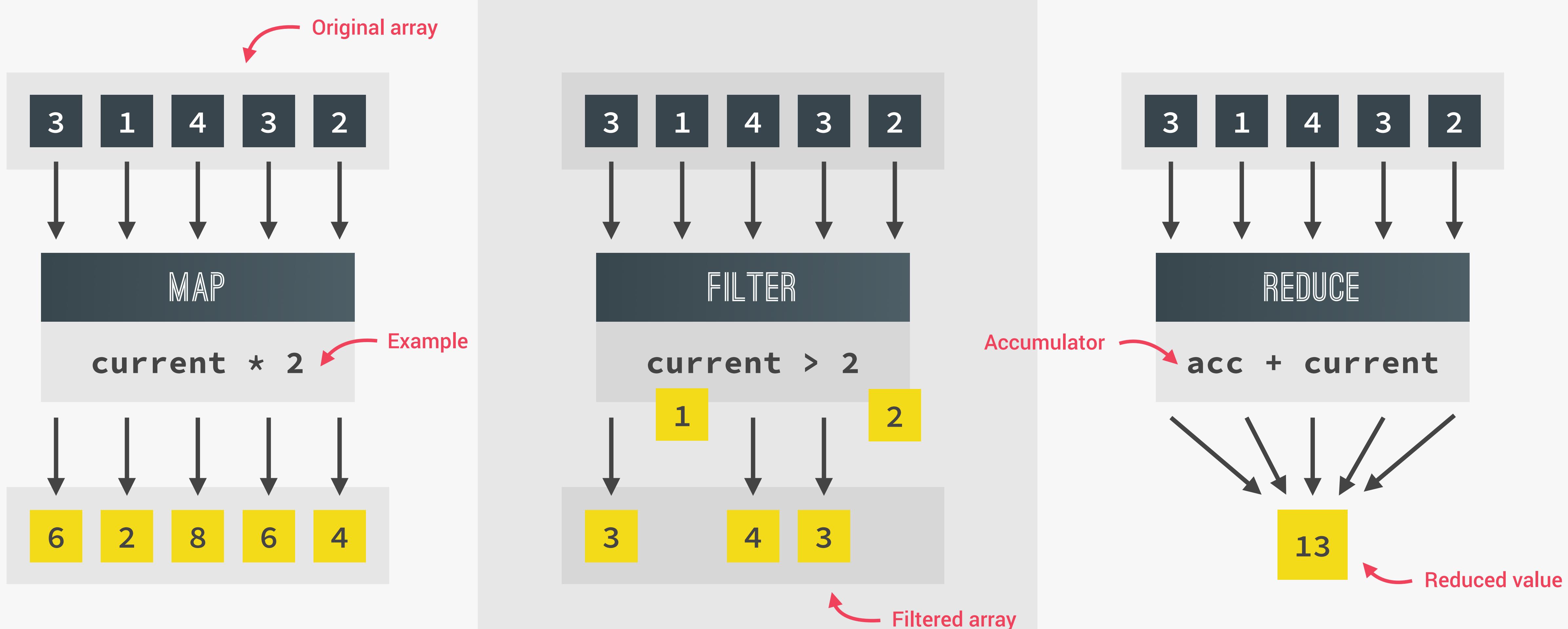
WORKING WITH ARRAYS

LECTURE

DATA TRANSFORMATIONS: MAP, FILTER,  
REDUCE



# 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!



@JONASSCHMEDTMAN

SECTION

WORKING WITH ARRAYS

LECTURE

SUMMARY: WHICH ARRAY METHOD TO  
USE?

JS

# WHICH ARRAY METHOD TO USE?



"I WANT..."

## To mutate original array

👉 Add to original:

`.push` (end)

`.unshift` (start)

👉 Remove from original:

`.pop` (end)

`.shift` (start)

`.splice` (any)

👉 Others:

`.reverse`

`.sort`

`.fill`

## A new array

👉 Computed from original:

`.map` (loop)

👉 Filtered using condition:

`.filter`

👉 Portion of original:

`.slice`

👉 Adding original to other:

`.concat`

👉 Flattening the original:

`.flat`

`.flatMap`

## An array index

👉 Based on value:

`.indexof`

👉 Based on test condition:

`.findIndex`

## An array element

👉 Based on test condition:

`.find`

## Know if array includes

👉 Based on value:

`.includes`

👉 Based on test condition:

`.some`

`.every`

## A new string

👉 Based on separator string:

`.join`

## To transform to value

👉 Based on accumulator:

`.reduce`

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

## To just loop array

👉 Based on callback:

`.forEach`

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



# ADVANCED DOM AND EVENTS



JONAS.IO  
SCHMEDTMANN

# THE COMPLETE JAVASCRIPT COURSE

FROM ZERO TO EXPERT!



@JONASSCHMEDTMAN

SECTION

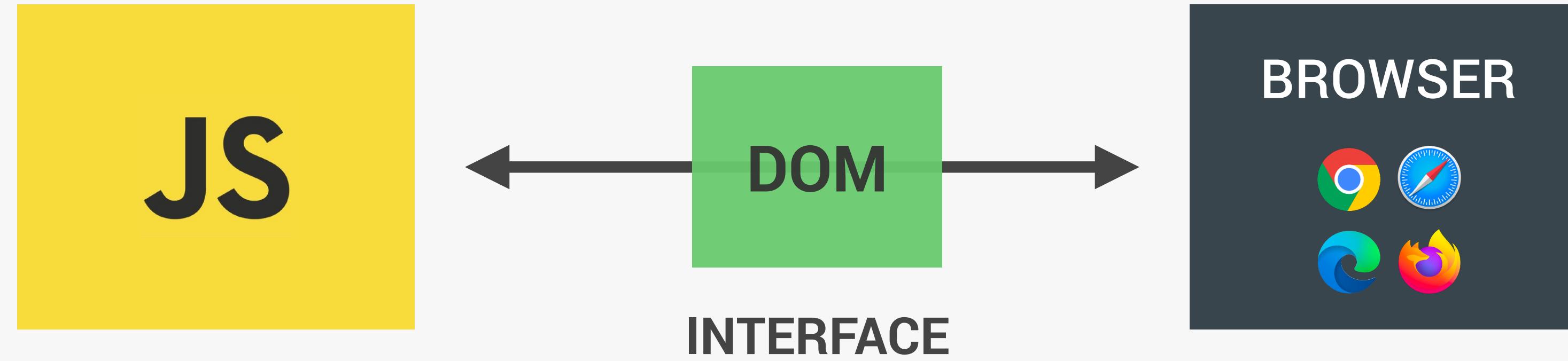
ADVANCED DOM AND EVENTS

LECTURE

HOW THE DOM REALLY WORKS

JS

# REVIEW: WHAT IS THE DOM?

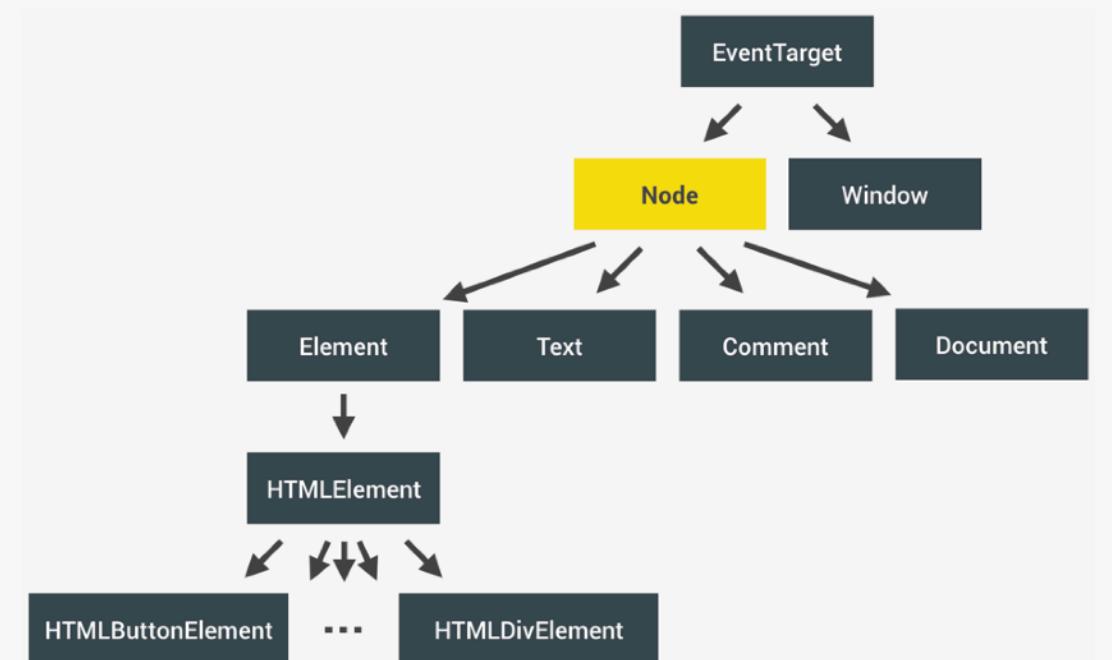


- 👉 Allows us to make JavaScript interact with the browser;
- 👉 We can write JavaScript to create, modify and delete HTML elements; set styles, classes and attributes; and listen and respond to events;
- 👉 DOM tree is generated from an HTML document, which we can then interact with;
- 👉 DOM is a very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

Application Programming Interface



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



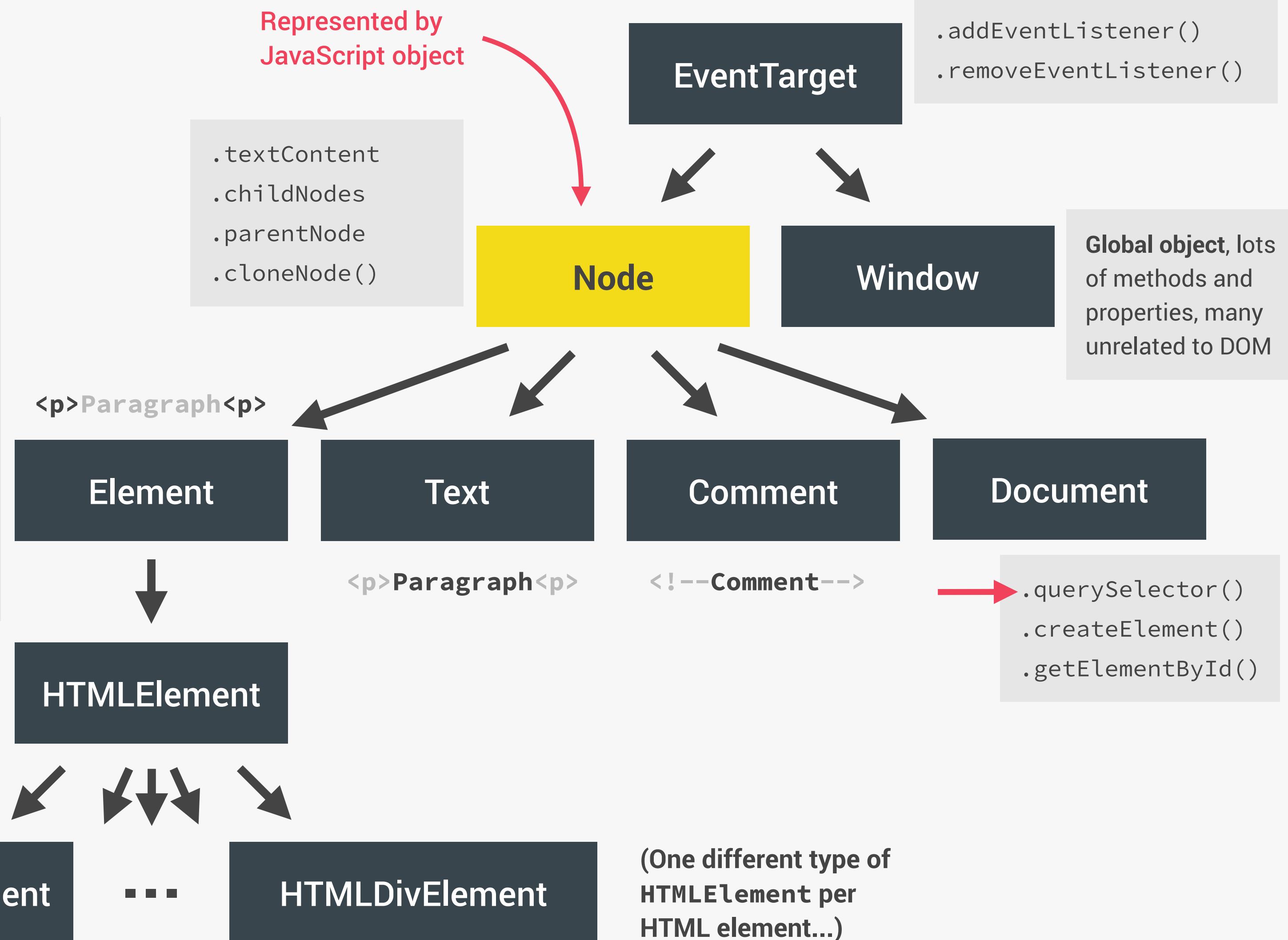
"Types" of  
DOM objects  
(next slide)

# HOW THE DOM API IS ORGANIZED BEHIND THE SCENES



.innerHTML  
.classList  
.children  
.parentElement  
.append()  
.remove()  
.insertAdjacentHTML()  
.querySelector()  
.closest()  
.matches()  
.scrollIntoView()  
.setAttribute()

→ .querySelector()



## INHERITANCE OF METHODS AND PROPERTIES

### Example:

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

(THIS IS NOT A DOM TREE)





# THE COMPLETE JAVASCRIPT COURSE

FROM ZERO TO EXPERT!



@JONASSCHMEDTMAN

SECTION

ADVANCED DOM AND EVENTS

LECTURE

EVENT PROPAGATION: BUBBLING AND  
CAPTURING

JS

# BUBBLING AND CAPTURING

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

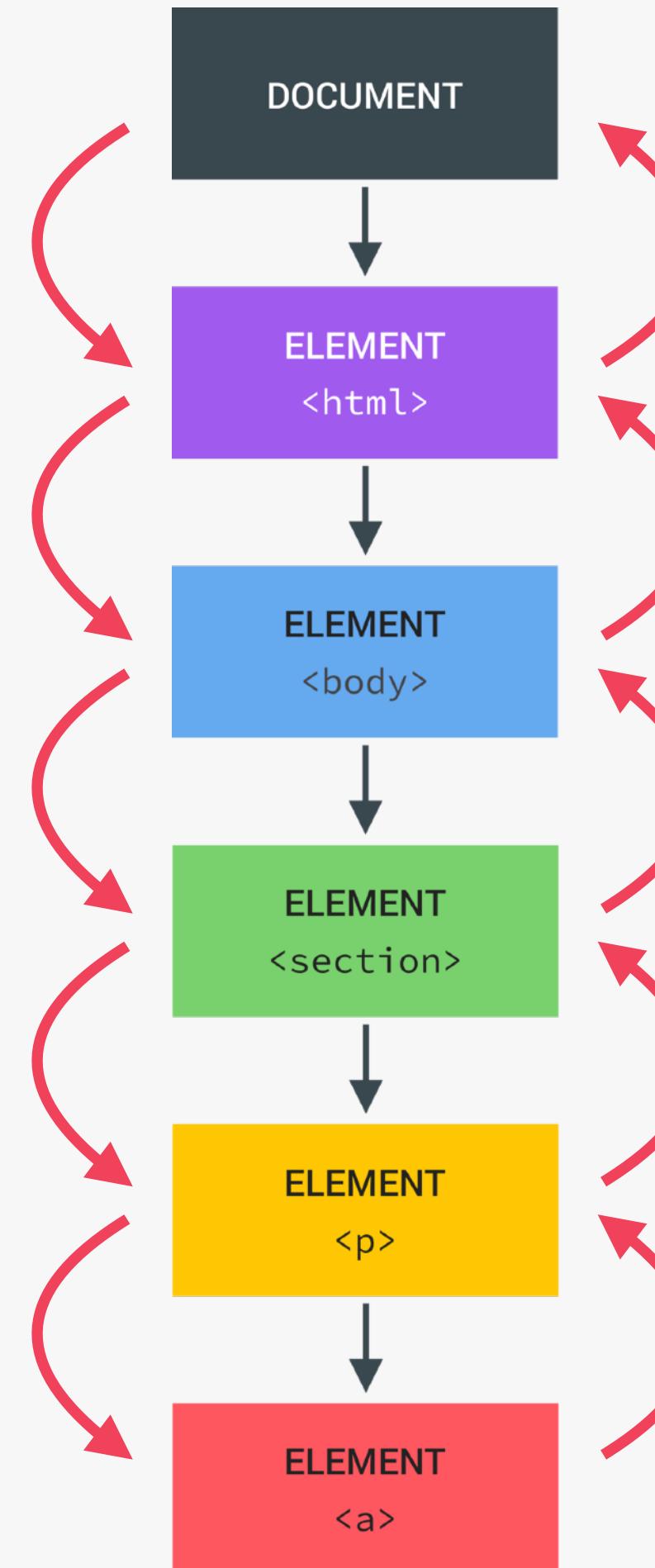
(THIS DOES NOT HAPPEN  
ON ALL EVENTS)

## 1 CAPTURING PHASE

Click event

1

## 2 TARGET PHASE



## 3 BUBBLING PHASE

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

```

127.0.0.1:8080 says  
You clicked me 😊

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

```

127.0.0.1:8080 says  
You clicked me 😊





JONAS.IO  
SCHMEDTMANN

# THE COMPLETE JAVASCRIPT COURSE

FROM ZERO TO EXPERT!



@JONASSCHMEDTMAN

SECTION

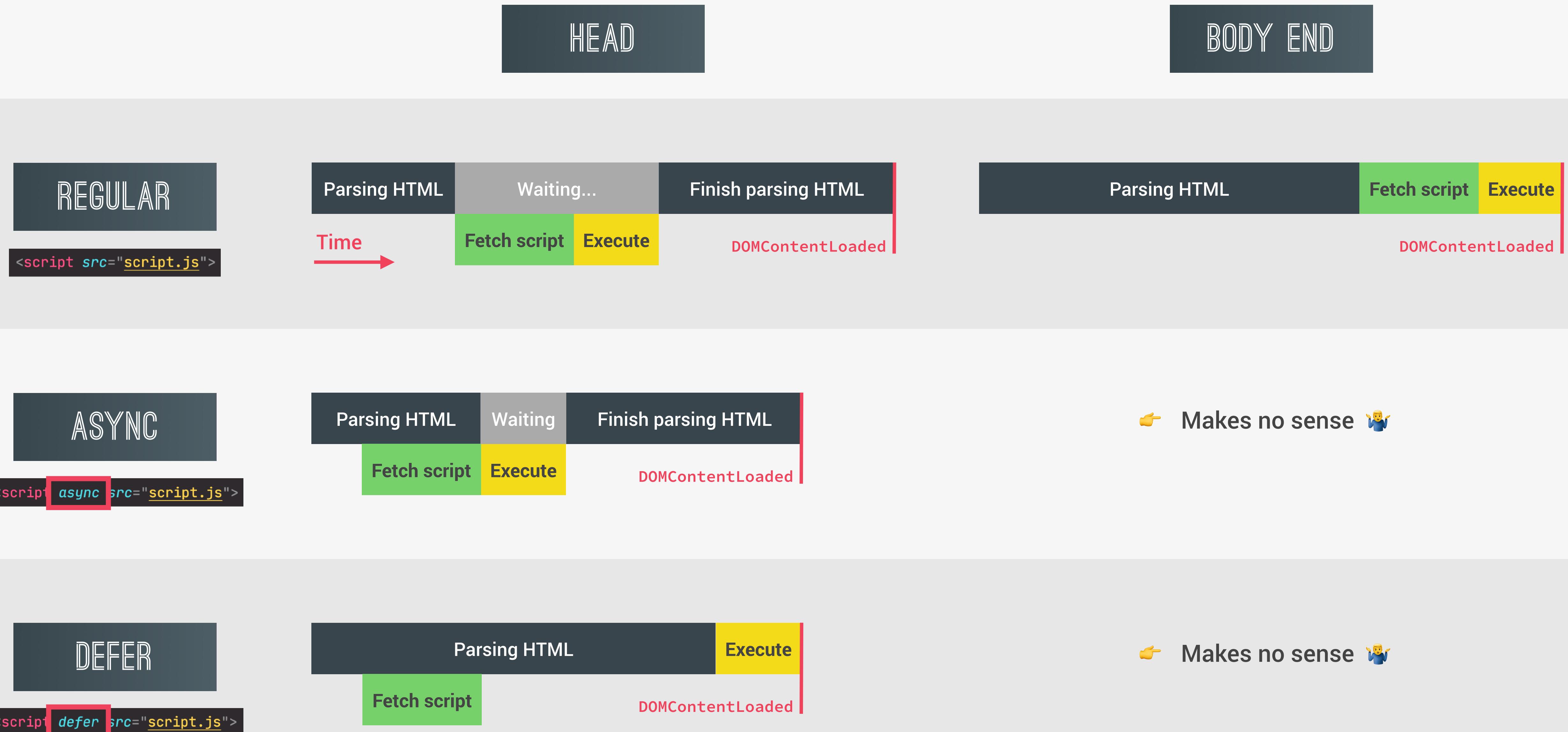
ADVANCED DOM AND EVENTS

LECTURE

EFFICIENT SCRIPT LOADING: DEFER  
AND ASYNC

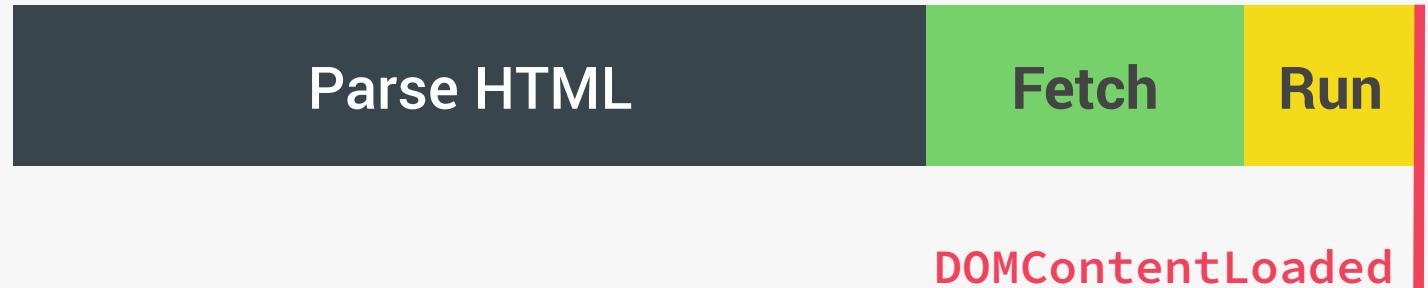
JS

# DEFER AND ASYNC SCRIPT LOADING



# REGULAR VS. ASYNC VS. DEFER

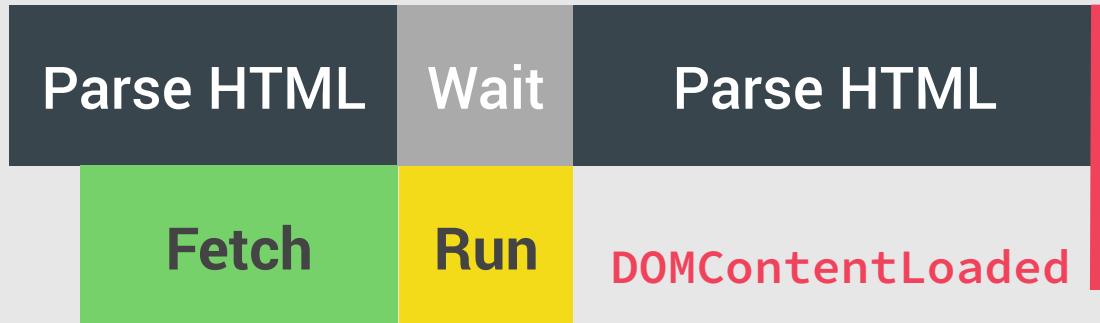
## END OF BODY



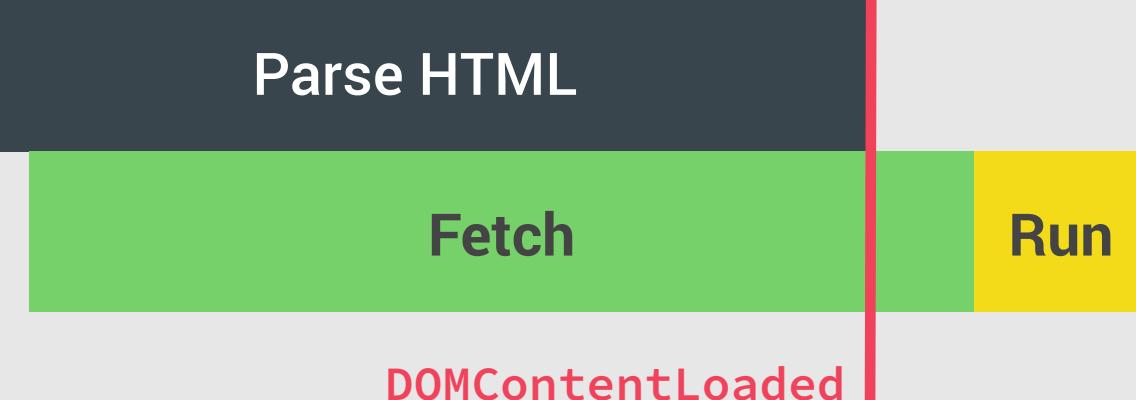
- 👉 Scripts are fetched and executed **after the HTML is completely parsed**
- 👉 **Use if you need to support old browsers**

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

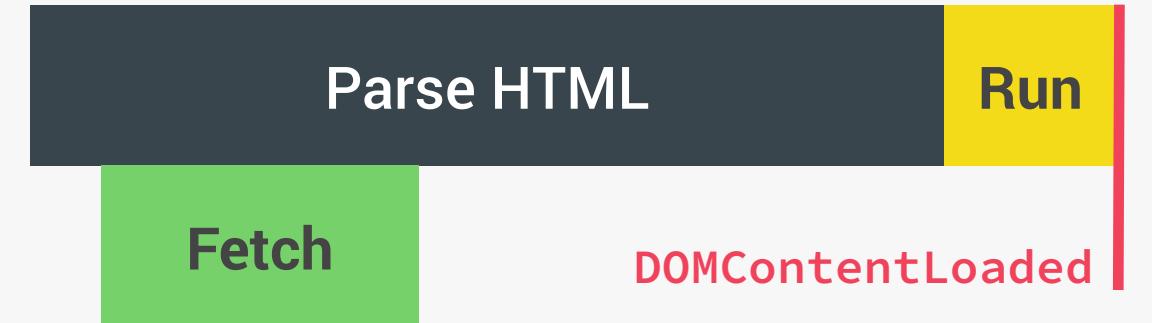
## ASYNC IN HEAD



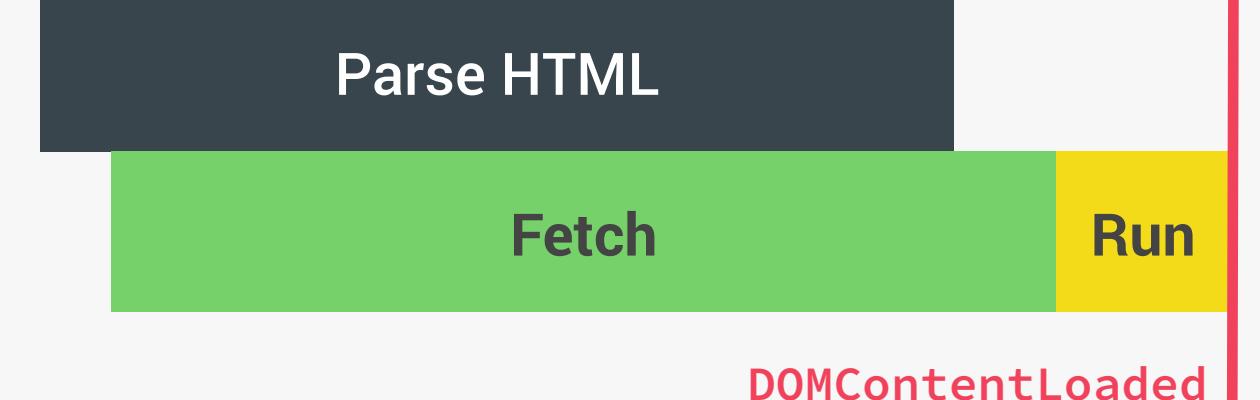
- 👉 Scripts are fetched **asynchronously** and executed **immediately**
- 👉 Usually the **DOMContentLoaded** event waits for **all** scripts to execute, except for `async` scripts. So, **DOMContentLoaded** does **not** wait for an `async` script
- 👉 Scripts **not** guaranteed to execute in order
- 👉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



## DEFER IN HEAD



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





OBJECT ORIENTED  
PROGRAMMING (OOP)  
WITH JAVASCRIPT



# THE COMPLETE JAVASCRIPT COURSE

FROM ZERO TO EXPERT!



@JONASSCHMEDTMAN

SECTION

OBJECT ORIENTED PROGRAMMING  
(OOP) WITH JAVASCRIPT

LECTURE

WHAT IS OBJECT-ORIENTED  
PROGRAMMING?

JS

# WHAT IS OBJECT-ORIENTED PROGRAMMING? (OOP)

OOP

Data

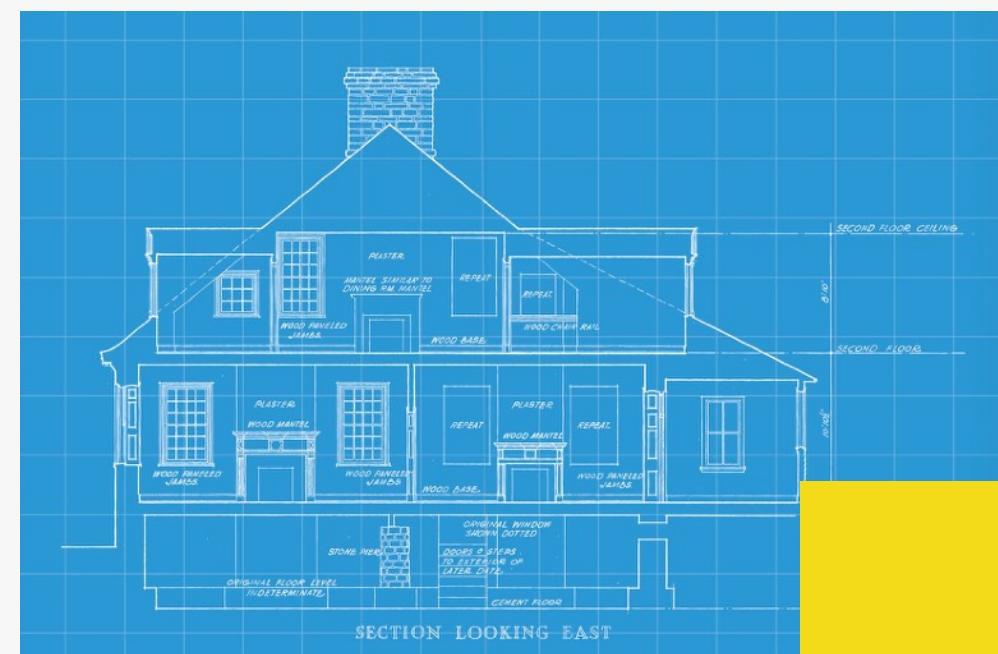
```
const user = {  
    user: 'jonas',  
    password: 'dk23s',  
  
    login(password) {  
        // Login logic  
    },  
    sendMessage(str) {  
        // Sending logic  
    }  
}
```

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;  
E.g. user or todo list item      E.g. HTML component or data structure
- 👉 Objects may contain data (properties) and code (methods). By using objects, we pack **data and the corresponding behavior** into one block;
- 👉 In OOP, objects are **self-contained** pieces/blocks of code;
- 👉 Objects are **building blocks** of applications, and **interact** with one another;
- 👉 Interactions happen through a **public interface** (API): methods that the code **outside** of the object can access and use to communicate with the object;
- 👉 OOP was developed with the goal of **organizing** code, to make it **more flexible** and easier to maintain (avoid “spaghetti code”).



# CLASSES AND INSTANCES (TRADITIONAL OOP)



## 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

Like a blueprint from  
which we can create  
new objects

## Instance



```
{  
  user = 'jonas'  
  password = 'dk23s'  
  email = 'hello@jonas.io'  
  
  login(password) {  
    // Login logic  
  }  
  sendMessage(str) {  
    // Sending logic  
  }  
}
```

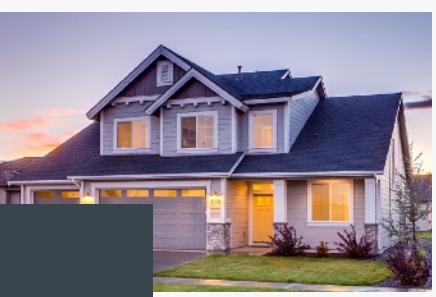
New object created from the class. Like a  
*real* house created from an *abstract* blueprint

## 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



Abstracted phone



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

Details have been abstracted away

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

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

# PRINCIPLE 2: ENCAPSULATION

Abstraction

Encapsulation

Inheritance

Polymorphism

NOT accessible from outside the class!

STILL accessible from within the class!

STILL accessible from within the class!

NOT accessible from outside the class!

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

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

WHY?

👉 Prevents external code from accidentally manipulating internal properties/state

👉 Allows to change internal implementation without the risk of breaking external code

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

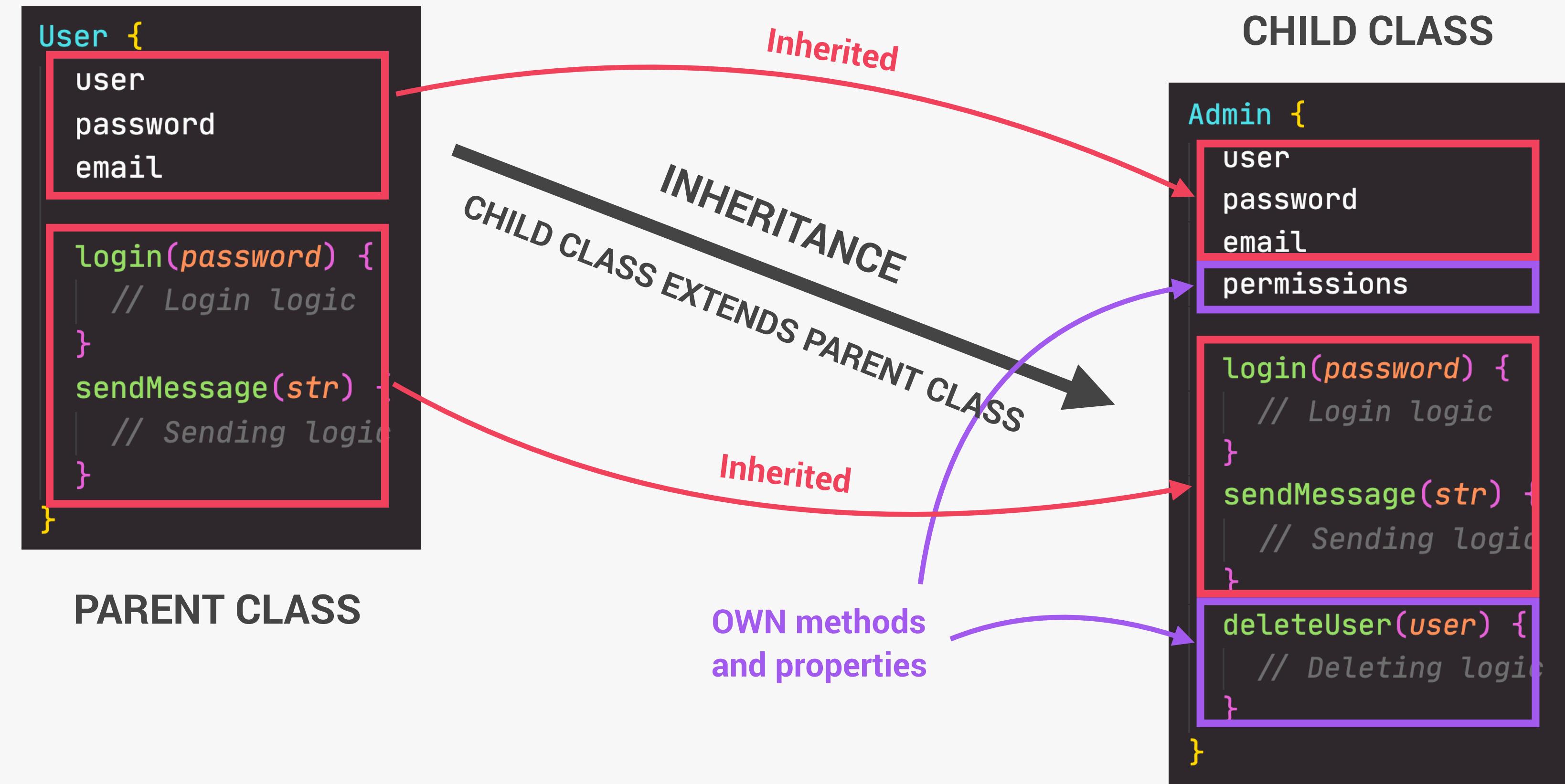
# PRINCIPLE 3: INHERITANCE

Abstraction

Encapsulation

Inheritance

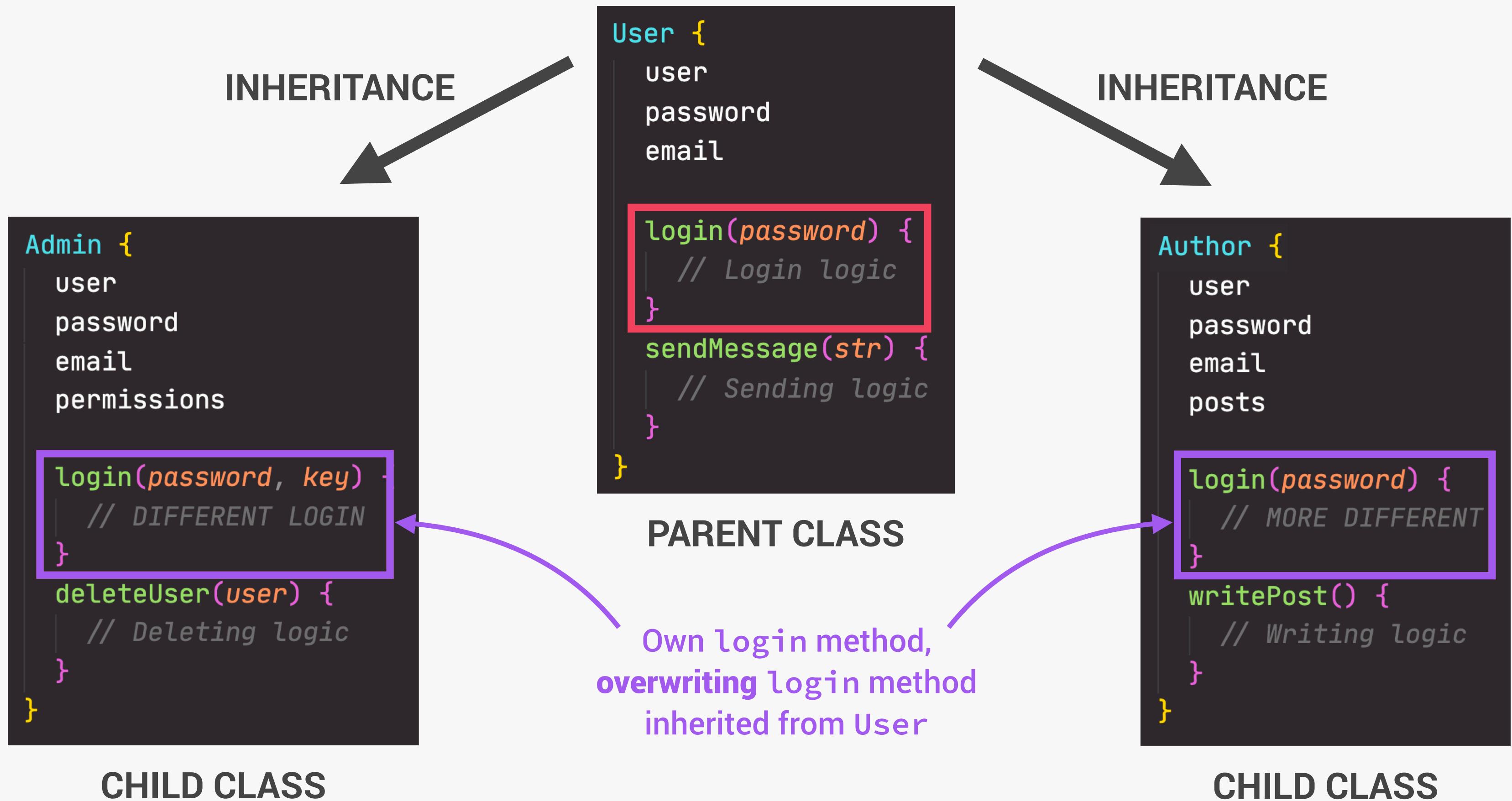
Polymorphism



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

# PRINCIPLE 4: POLYMORPHISM

Abstraction  
Encapsulation  
Inheritance  
Polymorphism



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





# THE COMPLETE JAVASCRIPT COURSE

FROM ZERO TO EXPERT!



@JONASSCHMEDTMAN

SECTION

OBJECT ORIENTED PROGRAMMING  
(OOP) WITH JAVASCRIPT

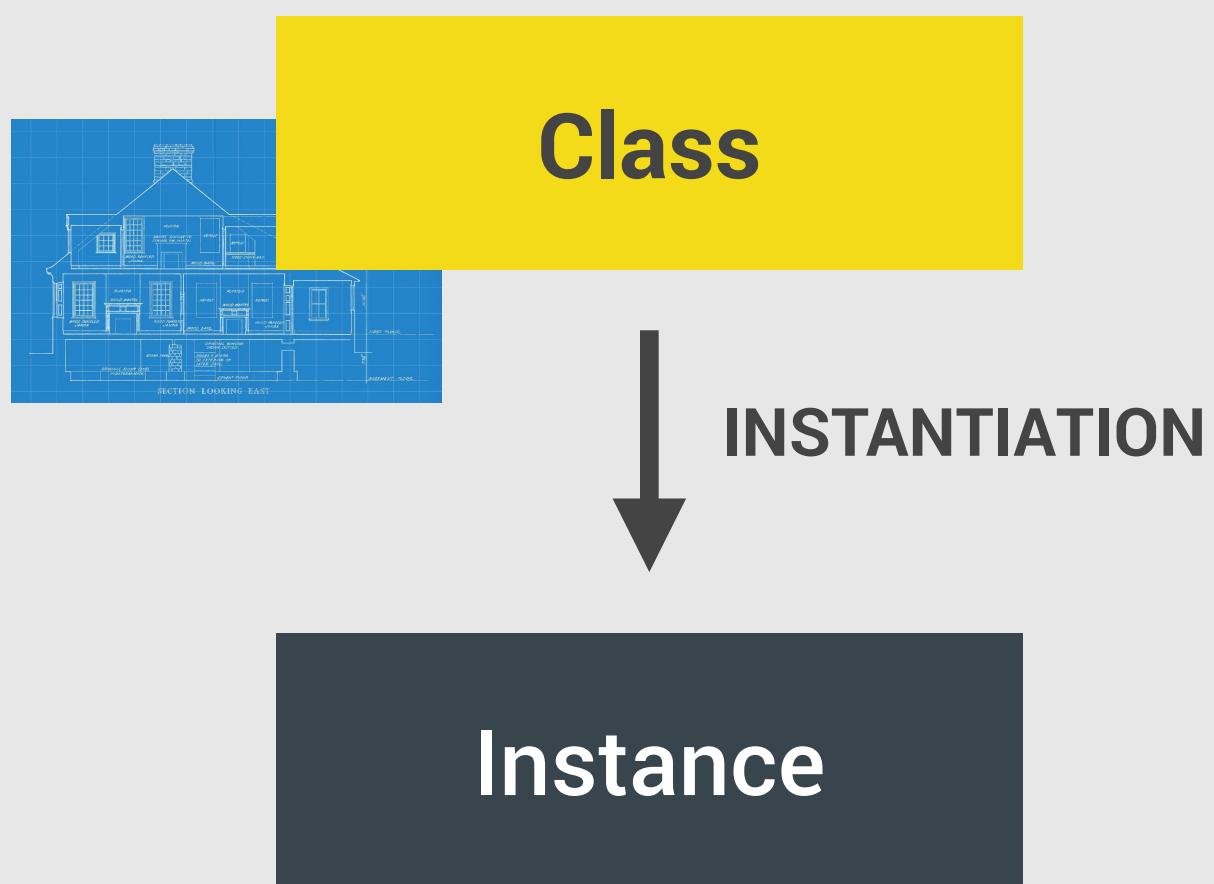
LECTURE

OOP IN JAVASCRIPT

JS

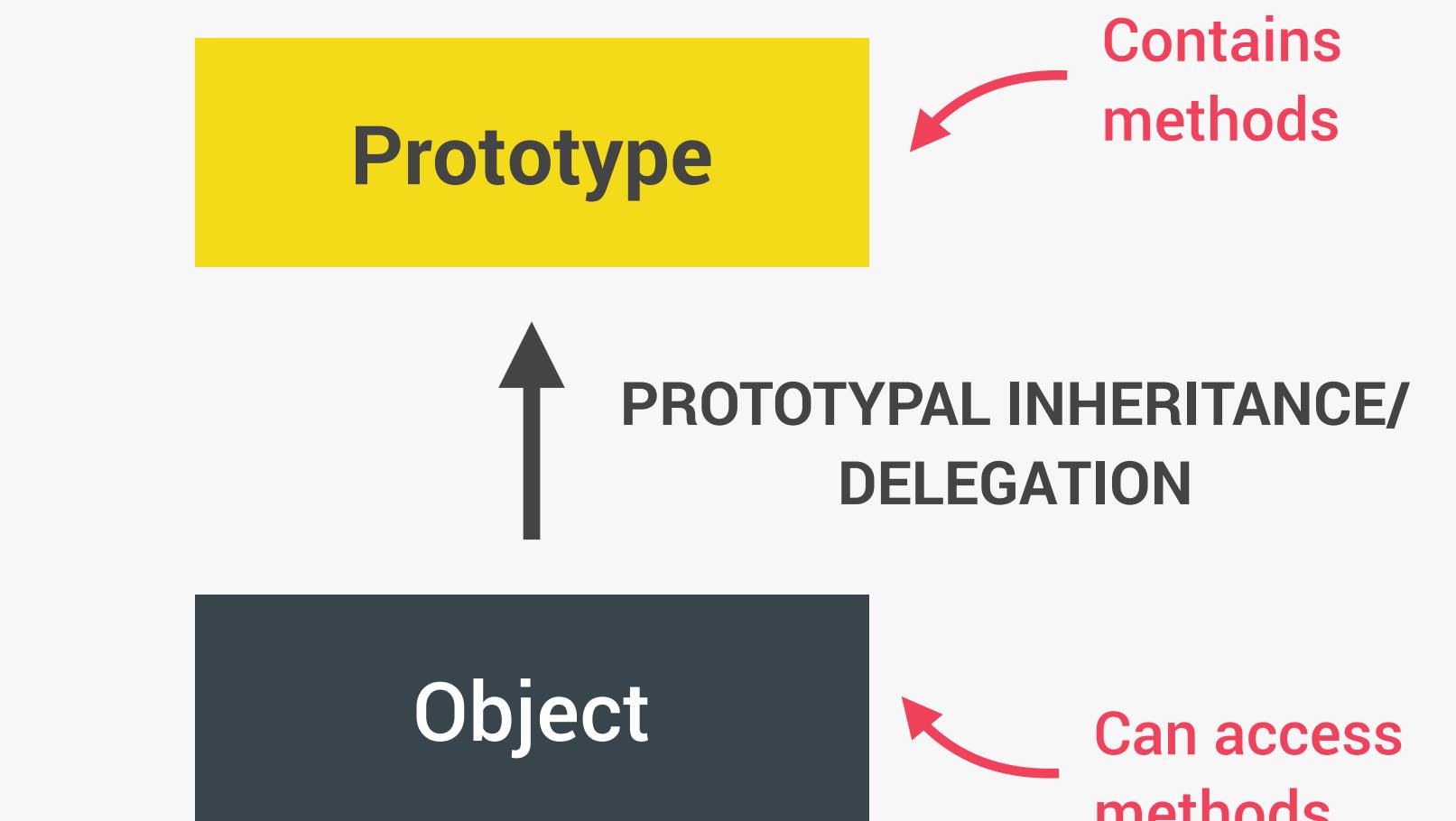
# OOP IN JAVASCRIPT: PROTOTYPES

## "CLASSICAL OOP": CLASSES



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

## OOP IN JS: PROTOTYPES



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

### 👉 Example: Array

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

MDN web docs  
moz://a

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

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

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

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

# 3 WAYS OF IMPLEMENTING PROTOTYPAL INHERITANCE IN JAVASCRIPT



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

👉 The 4 pillars of OOP are still valid!

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

1

## Constructor functions

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

2

## ES6 Classes

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

3

## `Object.create()`

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





# THE COMPLETE JAVASCRIPT COURSE

FROM ZERO TO EXPERT!



@JONASSCHMEDTMAN

SECTION

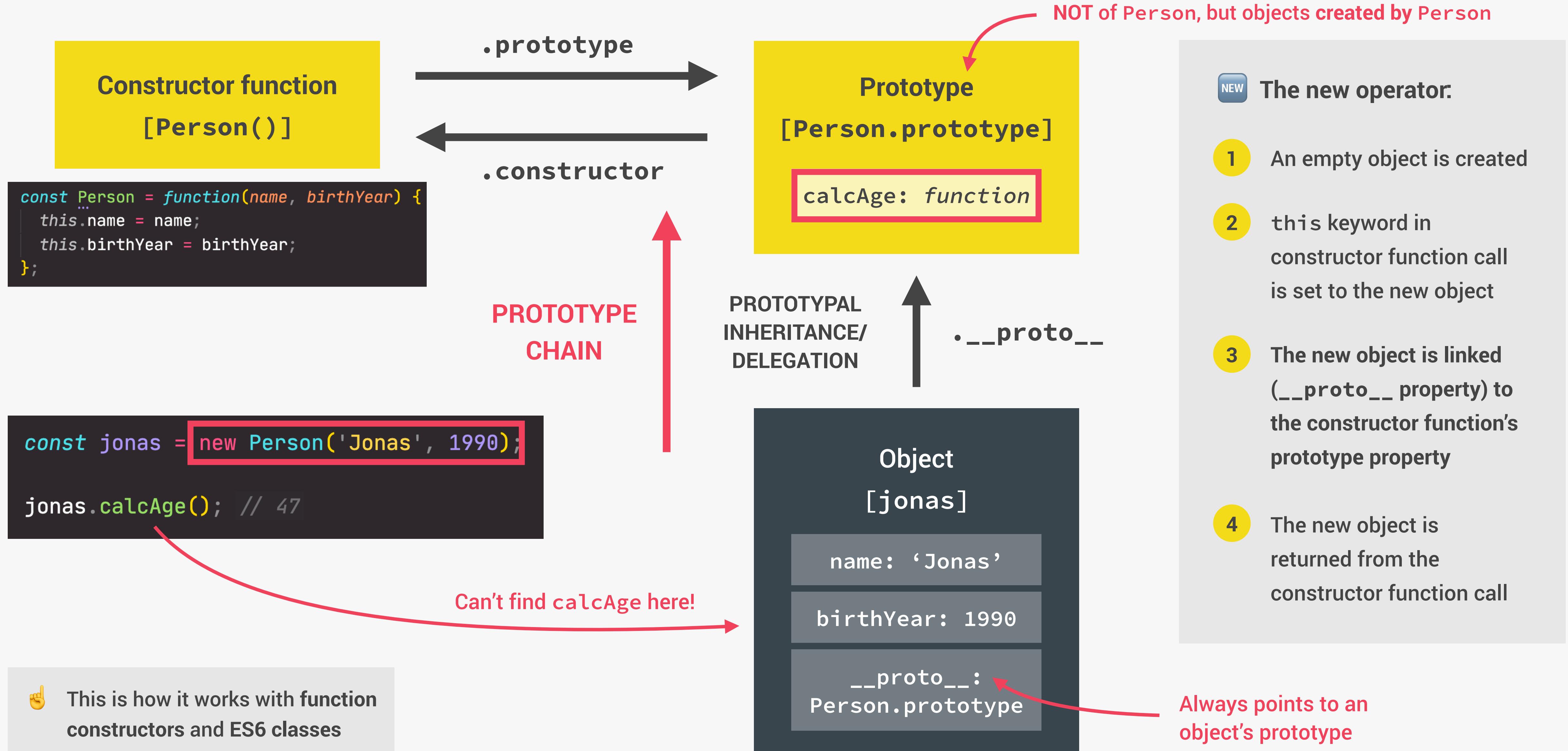
OBJECT ORIENTED PROGRAMMING  
(OOP) WITH JAVASCRIPT

LECTURE

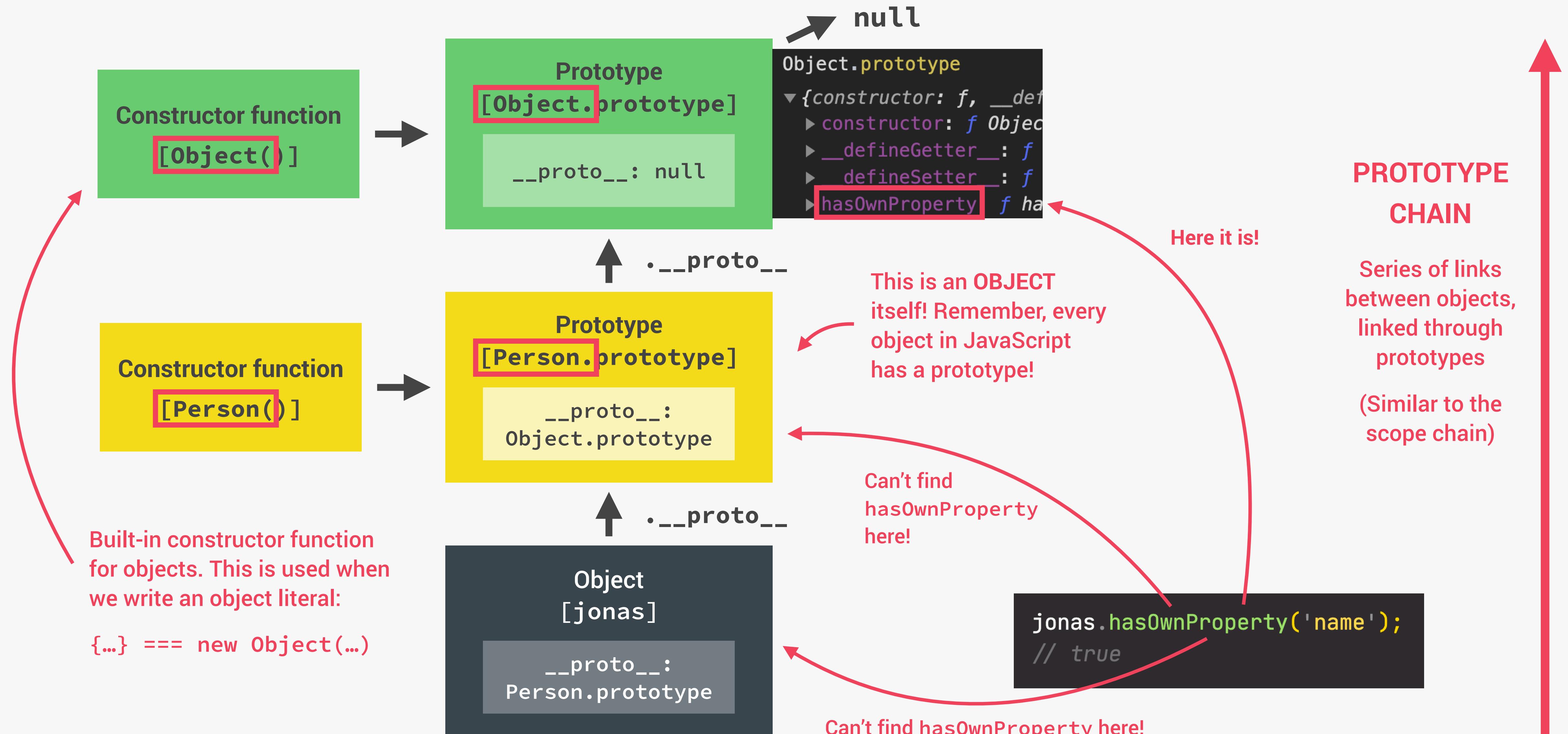
PROTOTYPAL INHERITANCE AND THE  
PROTOTYPE CHAIN

JS

# HOW PROTOTYPAL INHERITANCE / DELEGATION WORKS



# THE PROTOTYPE CHAIN







# THE COMPLETE JAVASCRIPT COURSE

FROM ZERO TO EXPERT!



@JONASSCHMEDTMAN

SECTION

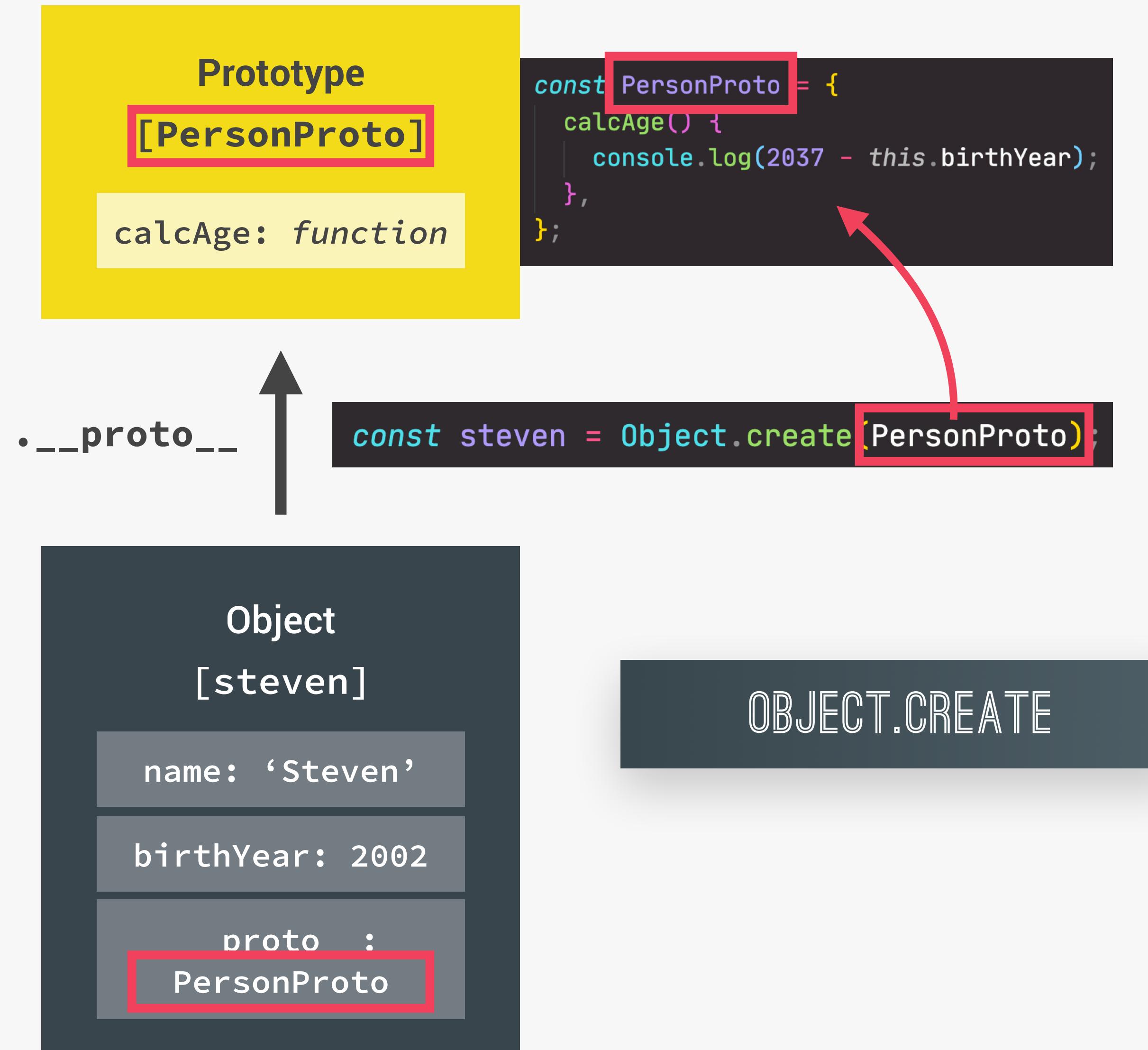
OBJECT ORIENTED PROGRAMMING  
(OOP) WITH JAVASCRIPT

LECTURE

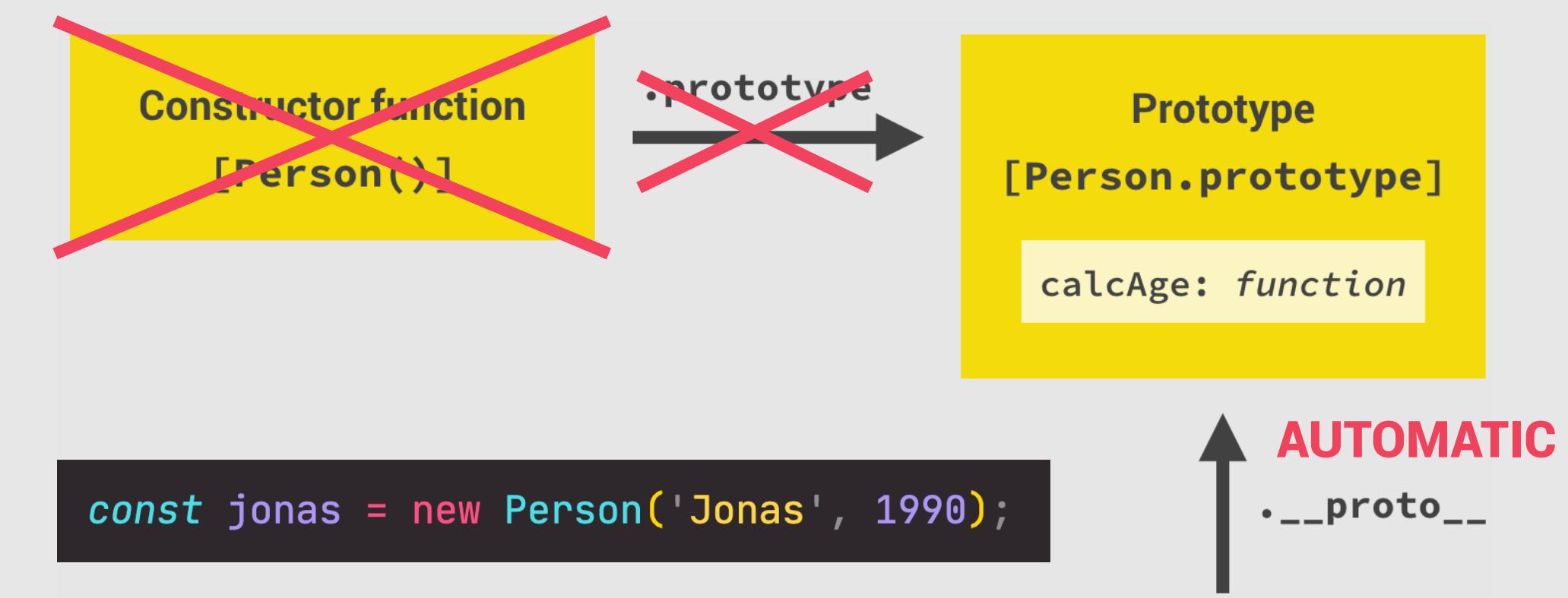
OBJECT.CREATE

JS

# HOW OBJECT.CREATE WORKS



## CONSTRUCTOR FUNCTIONS







# THE COMPLETE JAVASCRIPT COURSE

FROM ZERO TO EXPERT!



@JONASSCHMEDTMAN

SECTION

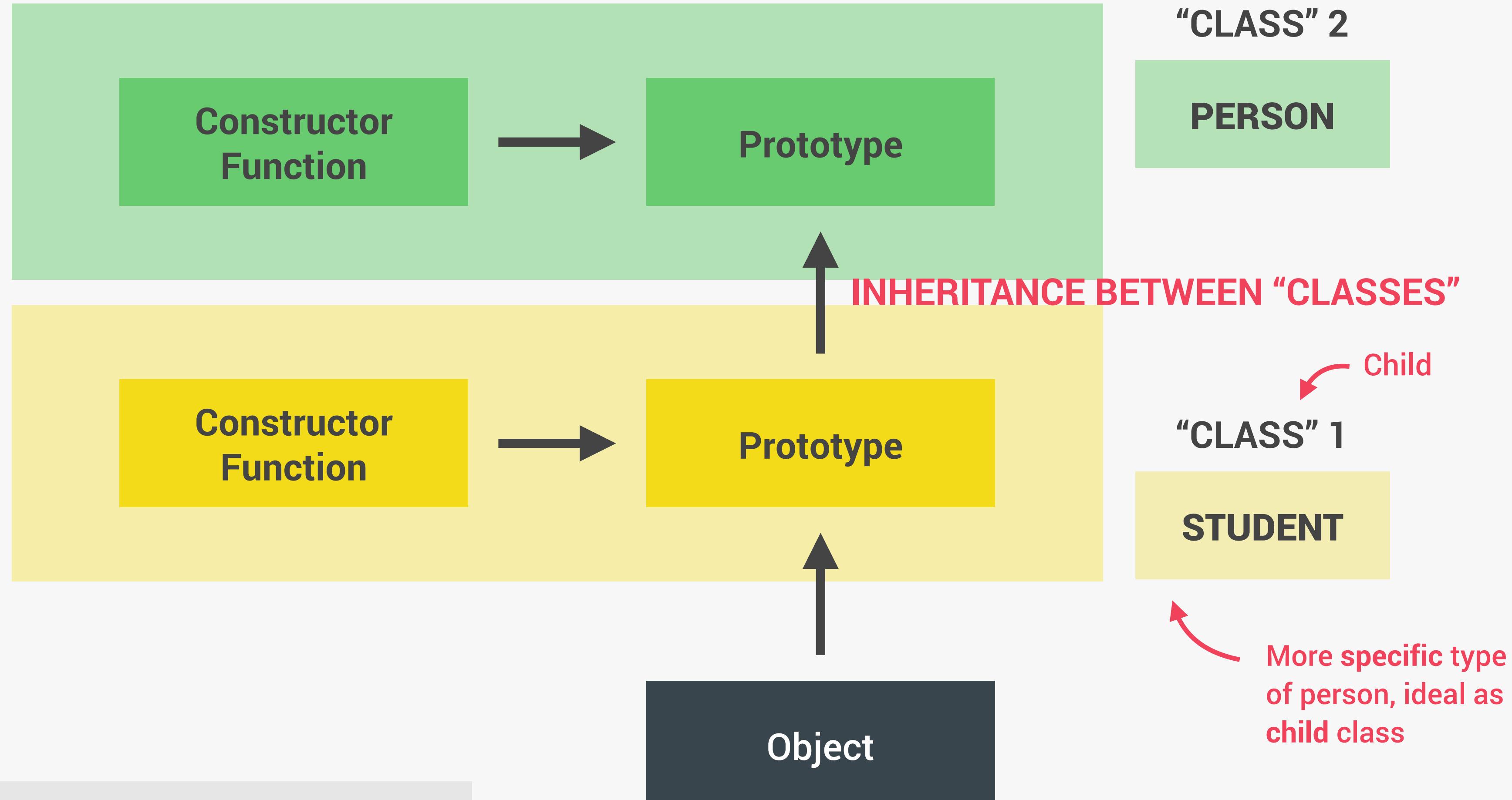
OBJECT ORIENTED PROGRAMMING  
(OOP) WITH JAVASCRIPT

LECTURE

INHERITANCE BETWEEN "CLASSES":  
CONSTRUCTOR FUNCTIONS

JS

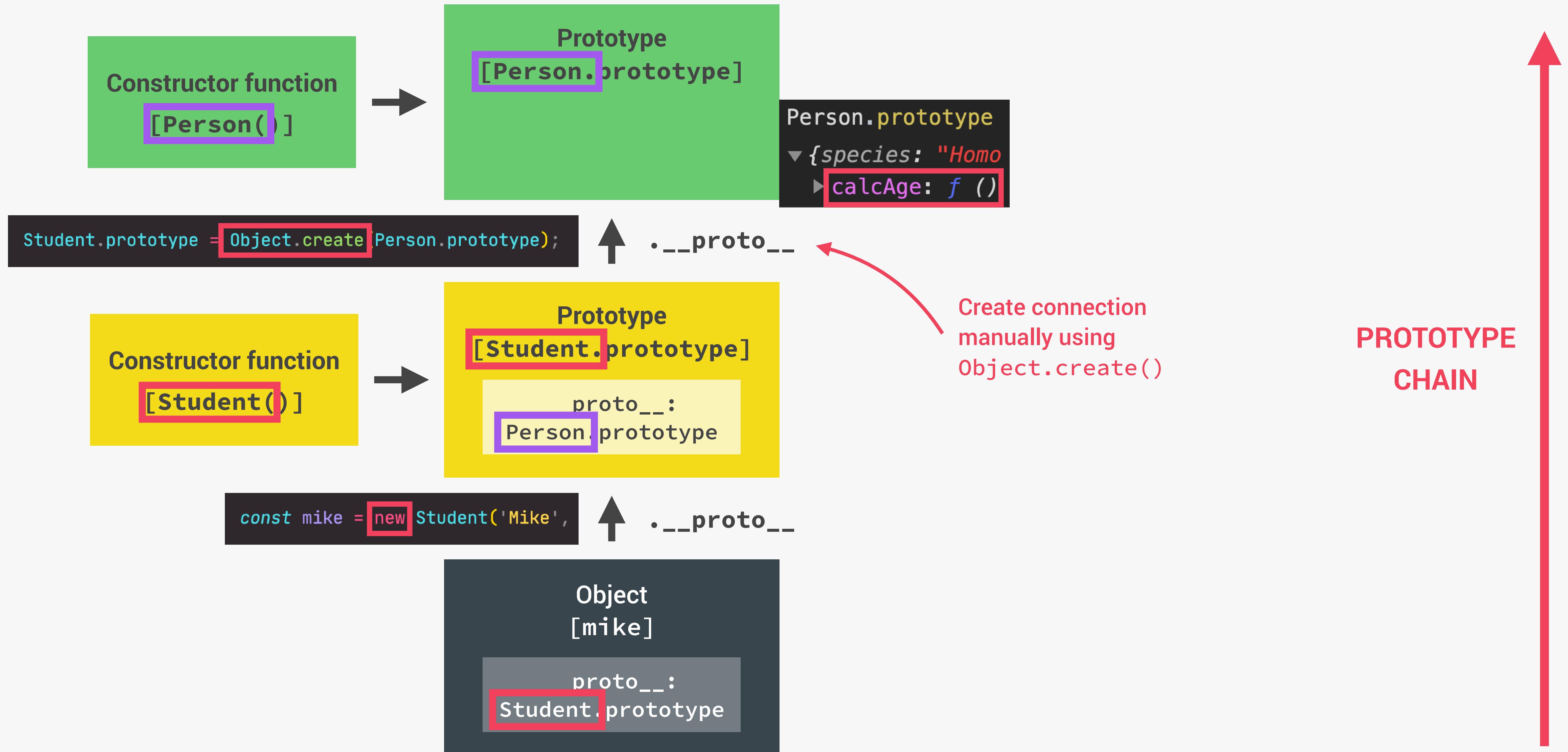
# INHERITANCE BETWEEN "CLASSES"



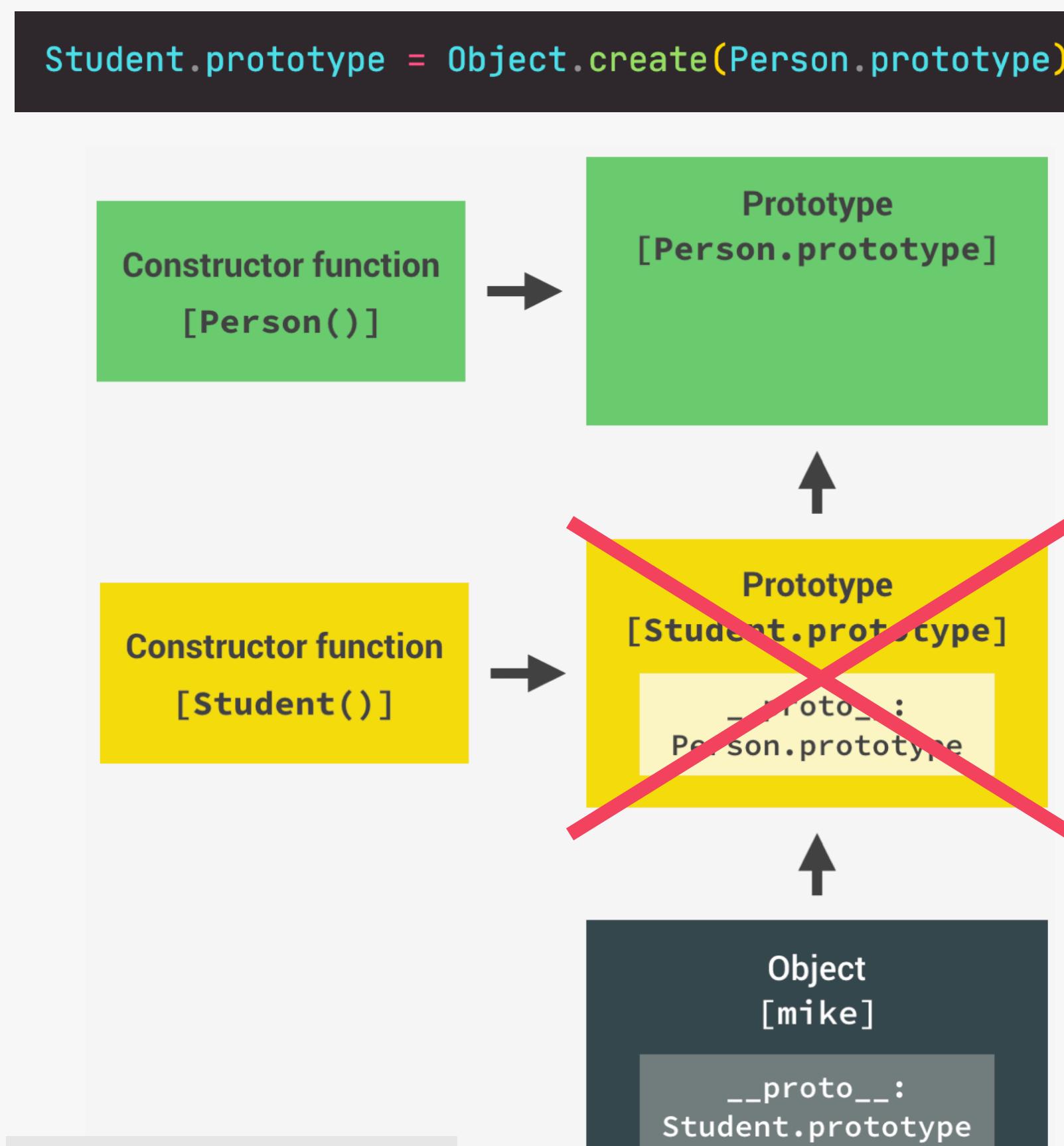
👉 Using class terminology here to make it easier to understand.

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

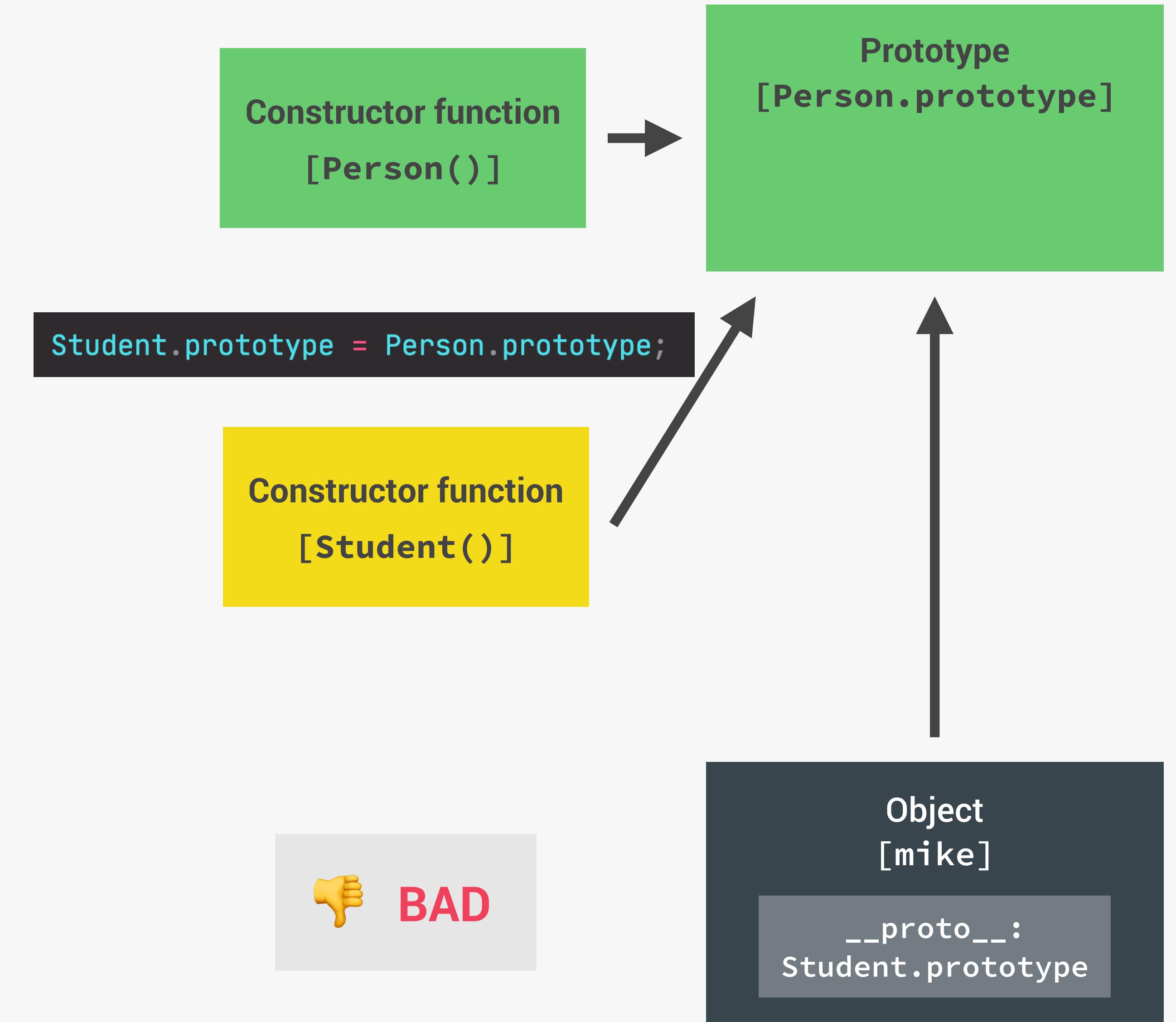
# INHERITANCE BETWEEN "CLASSES"



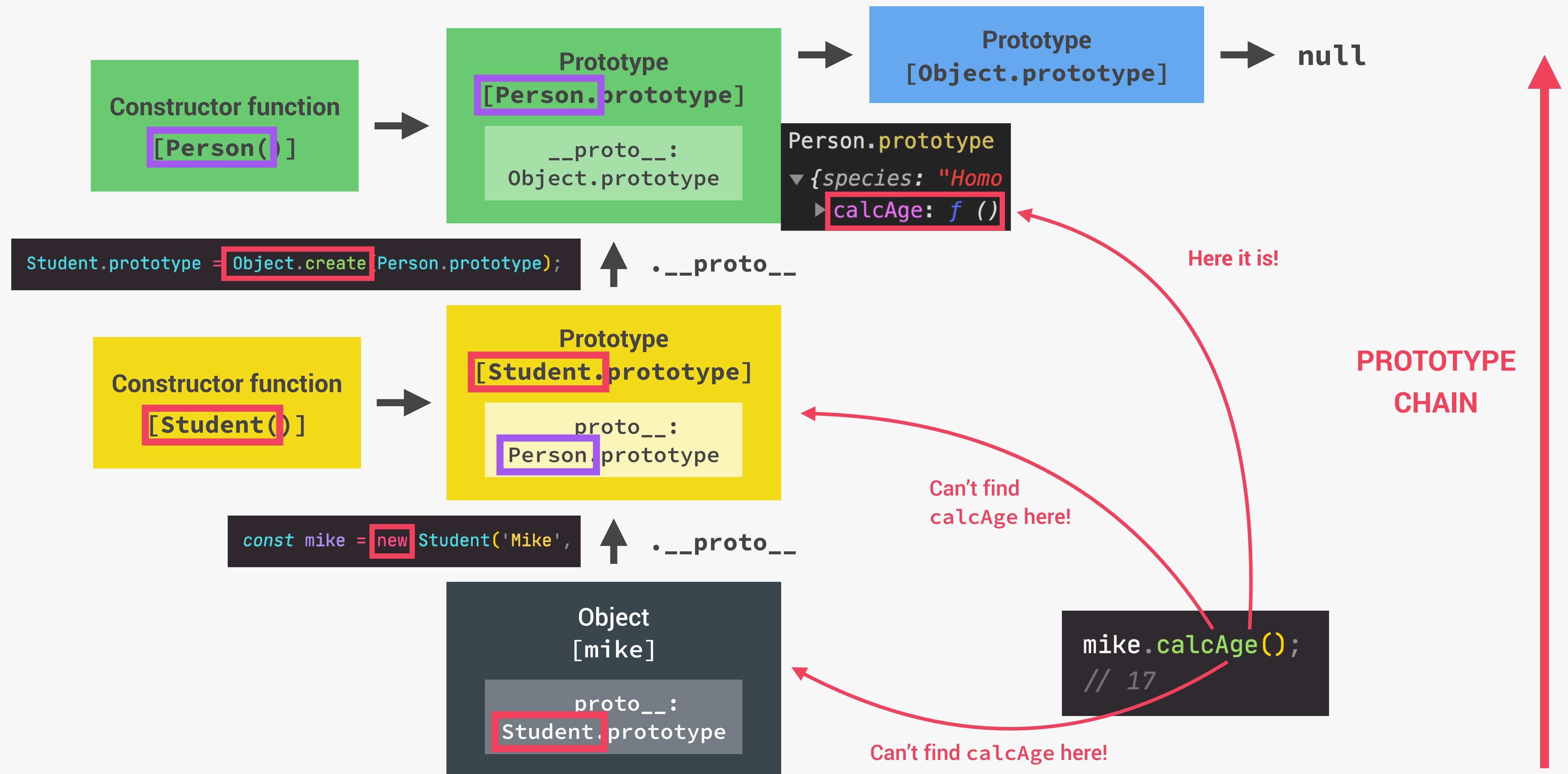
# INHERITANCE BETWEEN "CLASSES"



GOOD



# INHERITANCE BETWEEN "CLASSES"







# THE COMPLETE JAVASCRIPT COURSE

FROM ZERO TO EXPERT!



@JONASSCHMEDTMAN

SECTION

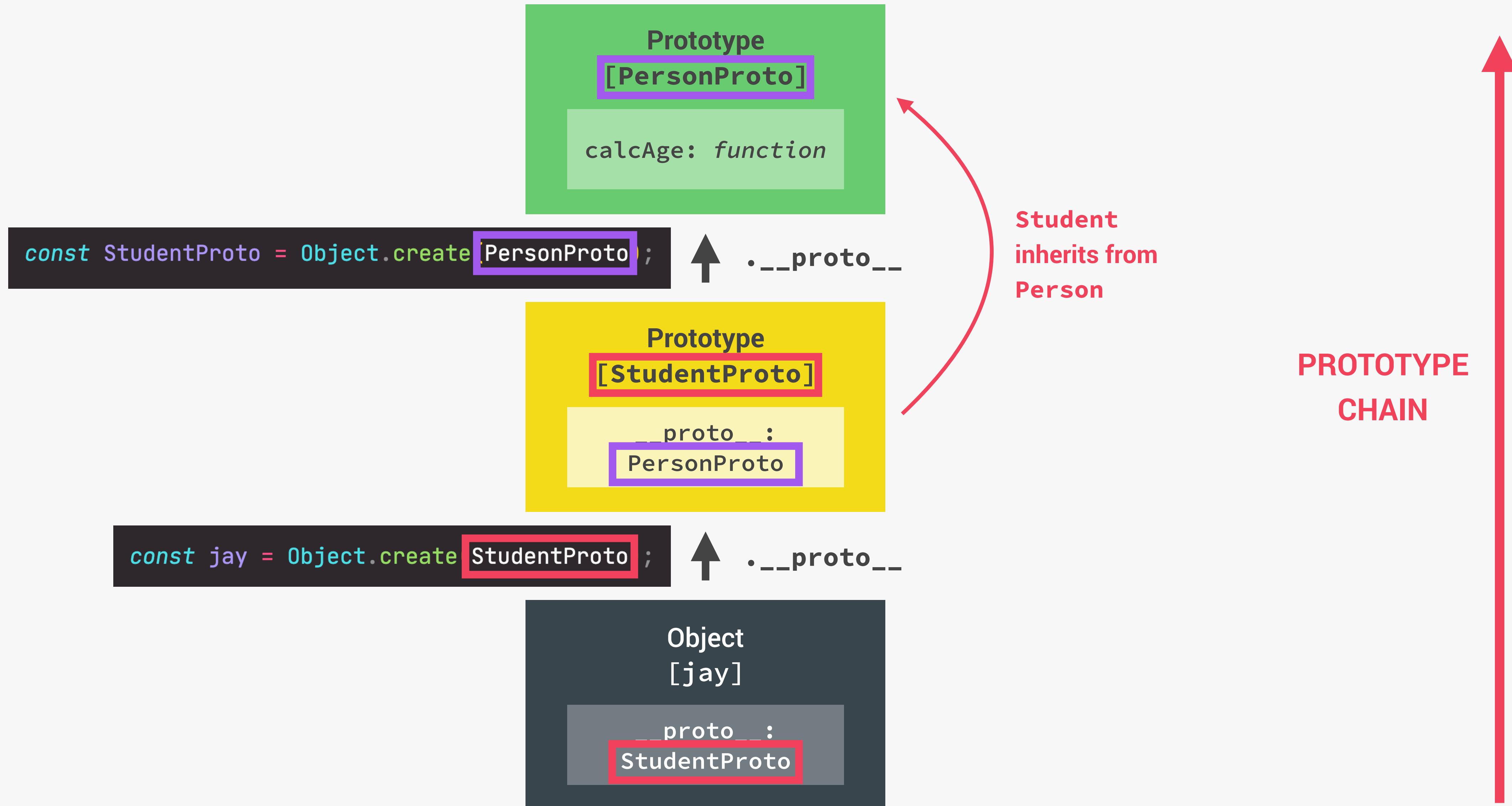
OBJECT ORIENTED PROGRAMMING  
(OOP) WITH JAVASCRIPT

LECTURE

INHERITANCE BETWEEN "CLASSES":  
OBJECT.CREATE

JS

# INHERITANCE BETWEEN "CLASSES": OBJECT.CREATE







# THE COMPLETE JAVASCRIPT COURSE

FROM ZERO TO EXPERT!



@JONASSCHMEDTMAN

SECTION

OBJECT ORIENTED PROGRAMMING  
(OOP) WITH JAVASCRIPT

LECTURE

ES6 CLASSES SUMMARY

JS

Public field (similar to property, available on created object)

Private fields (not accessible outside of class)

Static public field (available only on class)

Call to parent (super) class (necessary with extend). Needs to happen before accessing this

Instance property (available on created object)

Redefining private field

Public method

Referencing private field and method

Private method (⚠ Might not yet work in your browser. "Fake" alternative: \_ instead of #)

Getter method

Setter method (use \_ to set property with same name as method, and also add getter)

Static method (available only on class. Can not access instance properties nor methods, only static ones)

Creating new object with new operator

```
class Student extends Person {  
    university = 'University of Lisbon';  
    #studyHours = 0;  
    #course;  
    static numSubjects = 10;  
  
    constructor(fullName, birthYear, startYear, course) {  
        super(fullName, birthYear);  
  
        this.startYear = startYear;  
        this.#course = course;  
    }  
  
    introduce() {  
        console.log(`I study ${this.#course} at ${this.university}`);  
    }  
  
    study(h) {  
        this.#makeCoffe();  
        this.#studyHours += h;  
    }  
  
    #makeCoffe() {  
        return 'Here is a coffe for you ☕';  
    }  
  
    get testScore() {  
        return this._testScore;  
    }  
  
    set testScore(score) {  
        this._testScore = score ≤ 20 ? score : 0;  
    }  
  
    static printCurriculum() {  
        console.log(`There are ${this.numSubjects} subjects`);  
    }  
}  
  
const student = new Student('Jonas', 2020, 2037, 'Medicine');
```

Parent class

Inheritance between classes, automatically sets prototype

Child class

Constructor method, called by new operator. Mandatory in regular class, might be omitted in a child class

👉 Classes are just "syntactic sugar" over constructor functions

👉 Classes are not hoisted

👉 Classes are first-class citizens

👉 Class body is always executed in strict mode



**MAPTY APP. OOP,  
GEOLOCATION,  
EXTERNAL LIBRARIES,  
AND MORE!**



JONAS.IO  
SCHMEDTMANN

# THE COMPLETE JAVASCRIPT COURSE

FROM ZERO TO EXPERT!



@JONASSCHMEDTMAN

SECTION

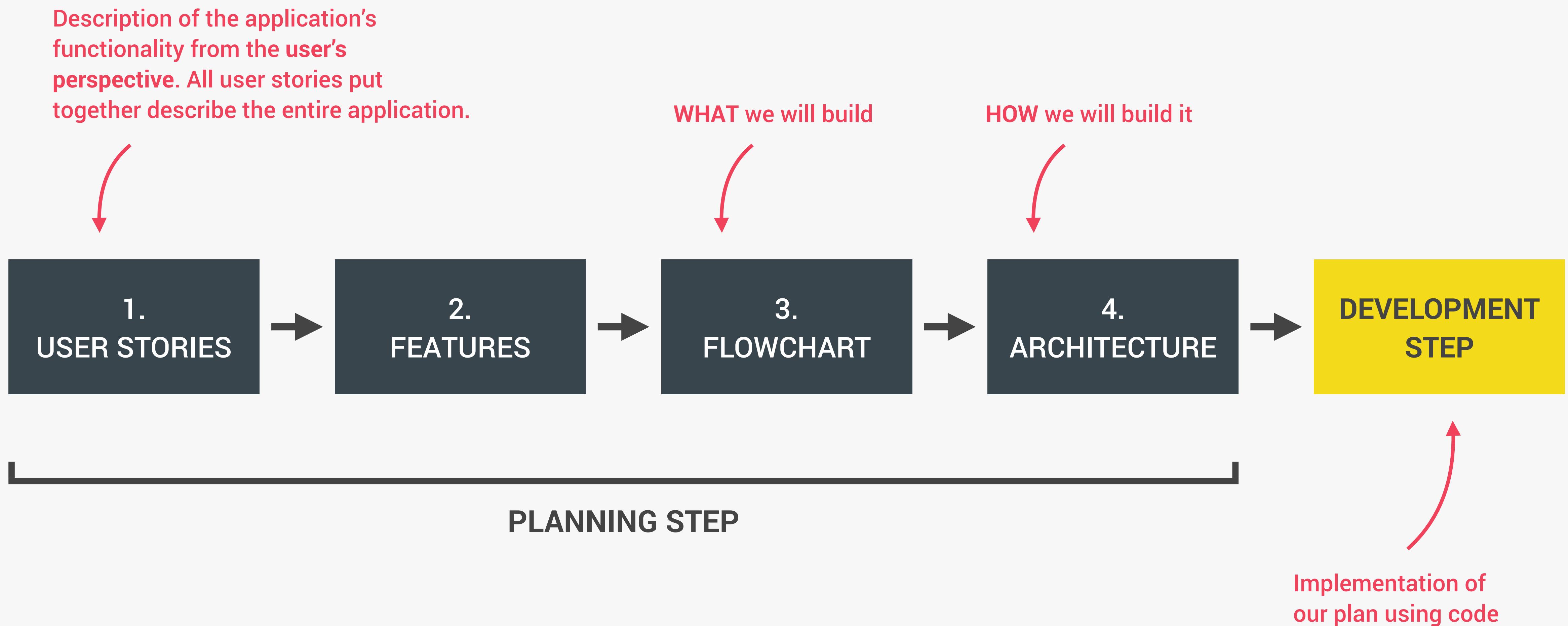
MAPTY APP: OOP, GEOLOCATION,  
EXTERNAL LIBRARIES, AND MORE!

LECTURE

HOW TO PLAN A WEB PROJECT

JS

# PROJECT PLANNING



# 1. USER STORIES



👉 **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



### USER STORIES



### FEATURES

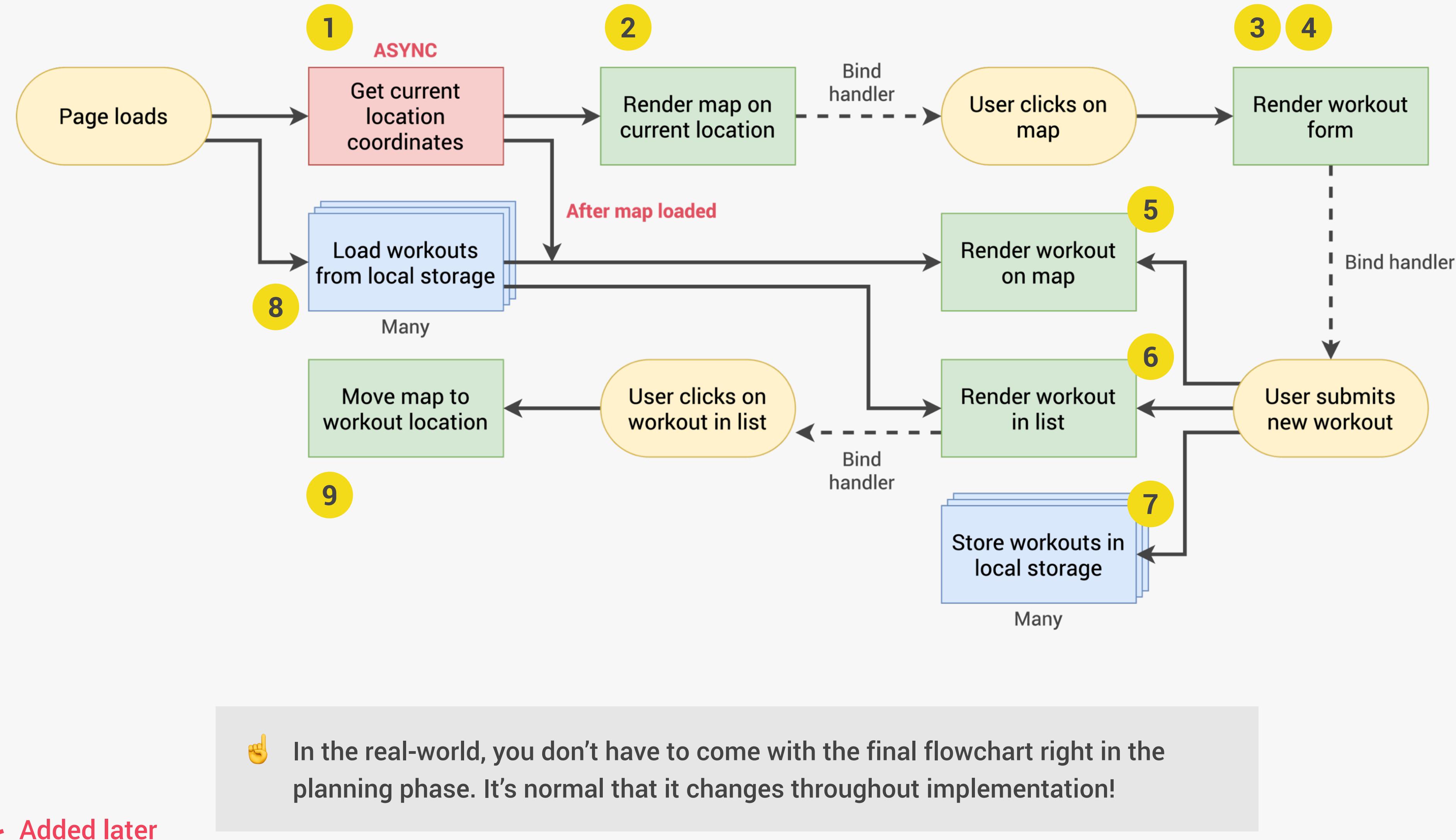
- |   |  |
|---|--|
| 1 Log my running workouts with location, distance, time, pace and steps/minute    | <ul style="list-style-type: none"><li>👉 Map where user clicks to add new workout (best way to get location coordinates)</li><li>👉 Geolocation to display map at current location (more user friendly)</li><li>👉 Form to input distance, time, pace, steps/minute</li></ul> |
| 2 Log my cycling workouts with location, distance, time, speed and elevation gain | <ul style="list-style-type: none"><li>👉 Form to input distance, time, speed, elevation gain</li></ul>  |
| 3 See all my workouts at a glance   | <ul style="list-style-type: none"><li>👉 Display all workouts in a list</li></ul>   |
| 4 See my workouts on a map  | <ul style="list-style-type: none"><li>👉 Display all workouts on the map</li></ul>  |
| 5 See all my workouts when I leave the app and come back later                    | <ul style="list-style-type: none"><li>👉 Store workout data in the browser using local storage API</li><li>👉 On page load, read the saved data from local storage and display</li></ul>   |

# 3. FLOWCHART



## FEATURES

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



FOR NOW, LET'S JUST  
START CODING 





# THE COMPLETE JAVASCRIPT COURSE

FROM ZERO TO EXPERT!



@JONASSCHMEDTMAN

SECTION

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

LECTURE

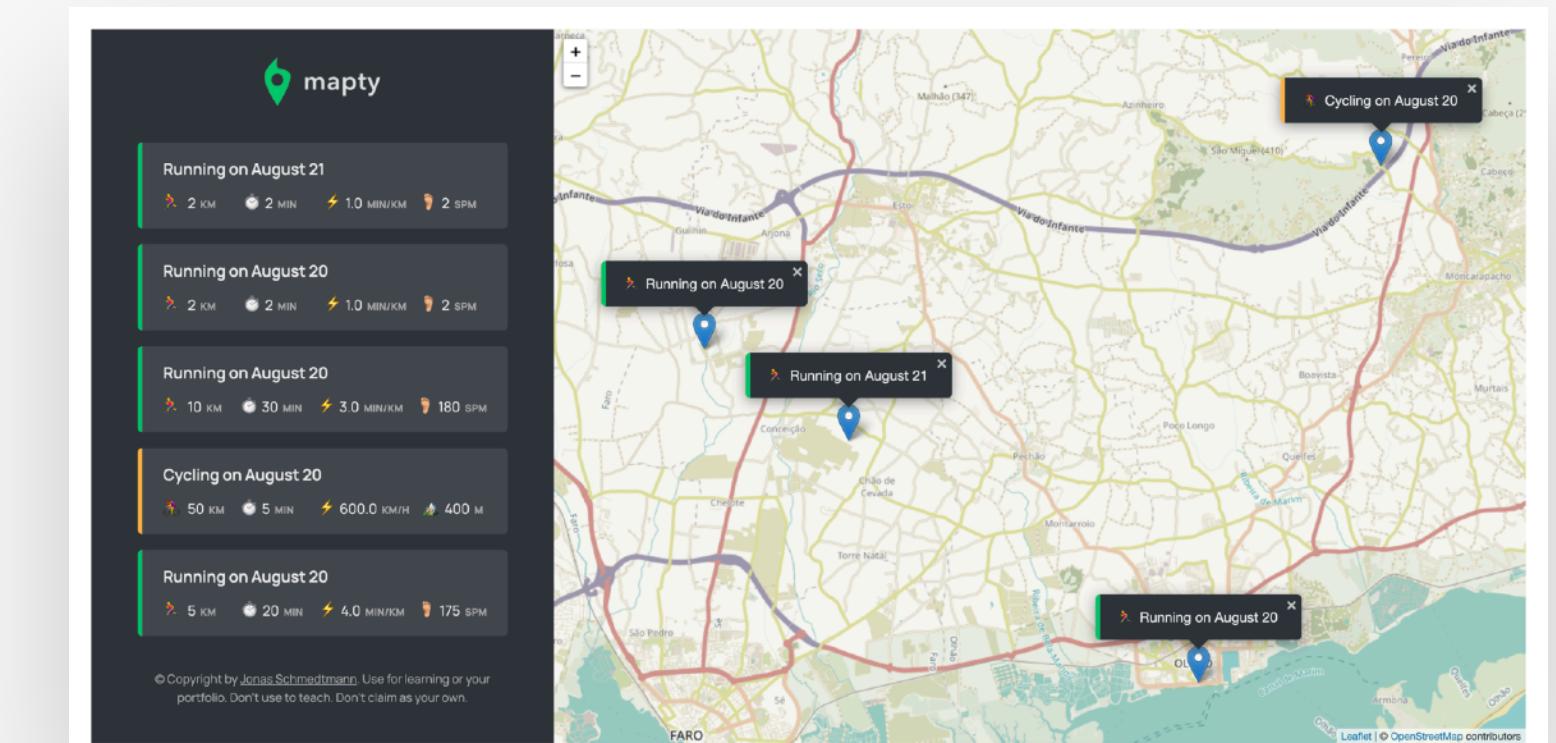
FINAL CONSIDERATIONS

JS

# 10 ADDITIONAL FEATURE IDEAS: CHALLENGES



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





# ASYNCHRONOUS JAVASCRIPT: PROMISES, ASYNC/ AWAIT AND AJAX



# THE COMPLETE JAVASCRIPT COURSE

FROM ZERO TO EXPERT!



@JONASSCHMEDTMAN

SECTION

ASYNCHRONOUS JAVASCRIPT:  
PROMISES, ASYNC/AWAIT AND AJAX

LECTURE

ASYNCHRONOUS JAVASCRIPT, AJAX  
AND APIs

JS

# SYNCHRONOUS CODE

BLOCKING



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

127.0.0.1:8080 says  
Text set!

OK

THREAD OF EXECUTION



SYNCHRONOUS

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

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

# ASYNCHRONOUS CODE

CALLBACK WILL  
RUN AFTER TIMER

Asynchronous

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

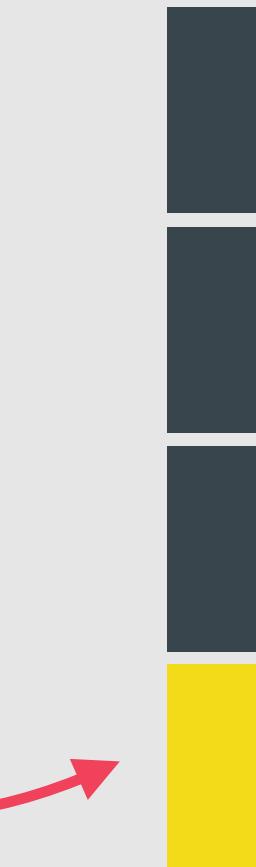
👉 Example: Timer with callback

Callback does NOT automatically  
make code asynchronous!

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

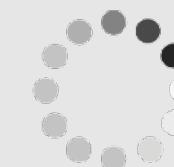
Executed after  
all other code

THREAD OF  
EXECUTION



"BACKGROUND"

Timer  
running



(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!

# ASYNCHRONOUS CODE

CALLBACK WILL RUN  
AFTER IMAGE LOADS

Asynchronous

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

👉 Example: Asynchronous image loading with event and callback

👉 Other examples: Geolocation API or AJAX calls

addEventListener does  
NOT automatically make  
code asynchronous!

ASYNCHRONOUS

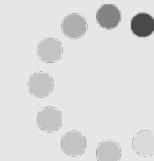
Coordinating behavior of a  
program over a period of time

THREAD OF  
EXECUTION



"BACKGROUND"

Image  
loading



(More on this in the  
lecture on Event Loop)

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

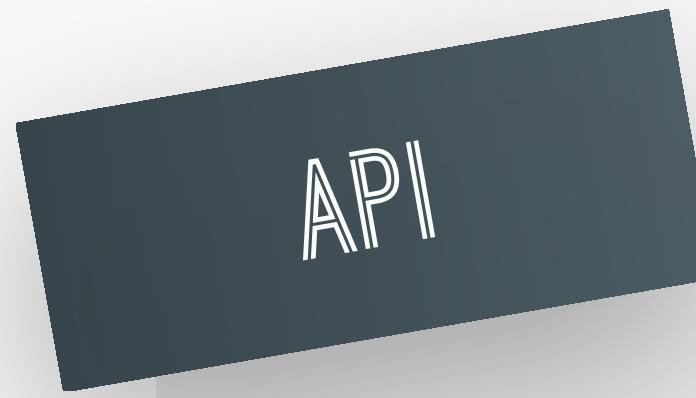
# WHAT ARE AJAX CALLS?

AJAX

**Asynchronous JavaScript And XML:** Allows us to communicate with remote web servers in an **asynchronous way**. With AJAX calls, we can **request data from web servers dynamically**.



# WHAT IS AN API?



- 👉 Application Programming Interface: Piece of software that can be used by another piece of software, in order to allow **applications to talk to each other**;

- 👉 There are many types of APIs in web development:

DOM API

Geolocation API

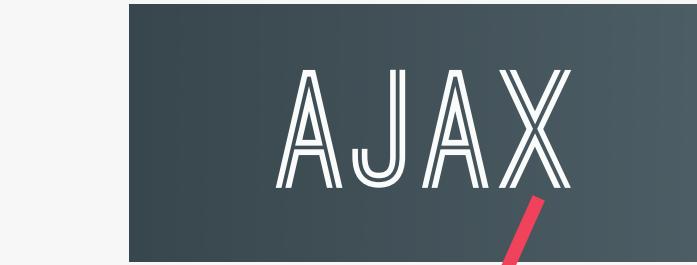
Own Class API

**“Online” API**

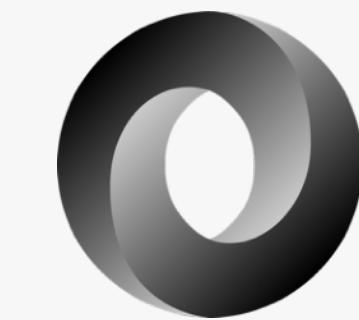
Just “API”

- 👉 **“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.



XML  
data  
format



JSON data  
format

{  
  "publisher": "101 Cookbooks",  
  "title": "Best Pizza Dough Ever",  
  "source\_url": "<http://www.101cookbo...>",  
  "recipe\_id": "47746",  
  "image\_url": "<http://forkify-api.he...>",  
  "social\_rank": 100,  
  "publisher\_url": "<http://www.101coo...>"  
},

Most popular  
API data format

There is an API for  
everything

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







# THE COMPLETE JAVASCRIPT COURSE

FROM ZERO TO EXPERT!



@JONASSCHMEDTMAN

SECTION

ASYNCHRONOUS JAVASCRIPT:  
PROMISES, ASYNC/AWAIT AND AJAX

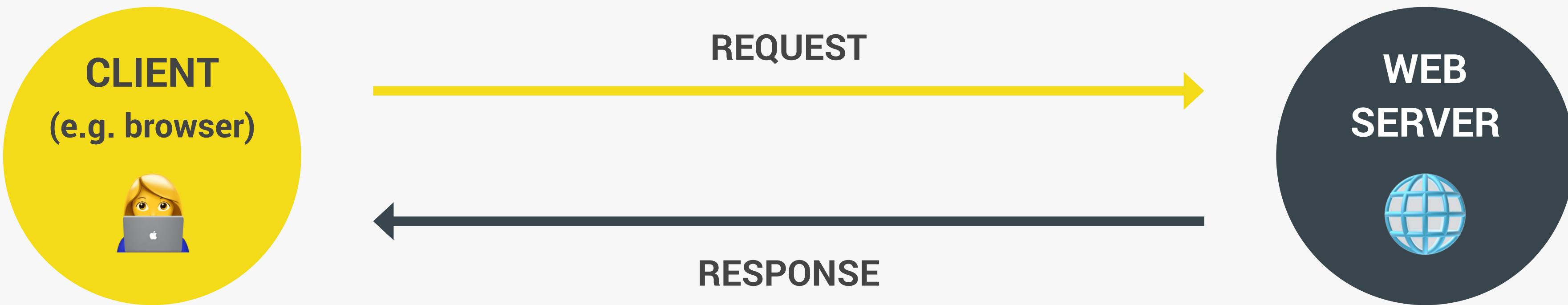
LECTURE

HOW THE WEB WORKS: REQUESTS  
AND RESPONSES

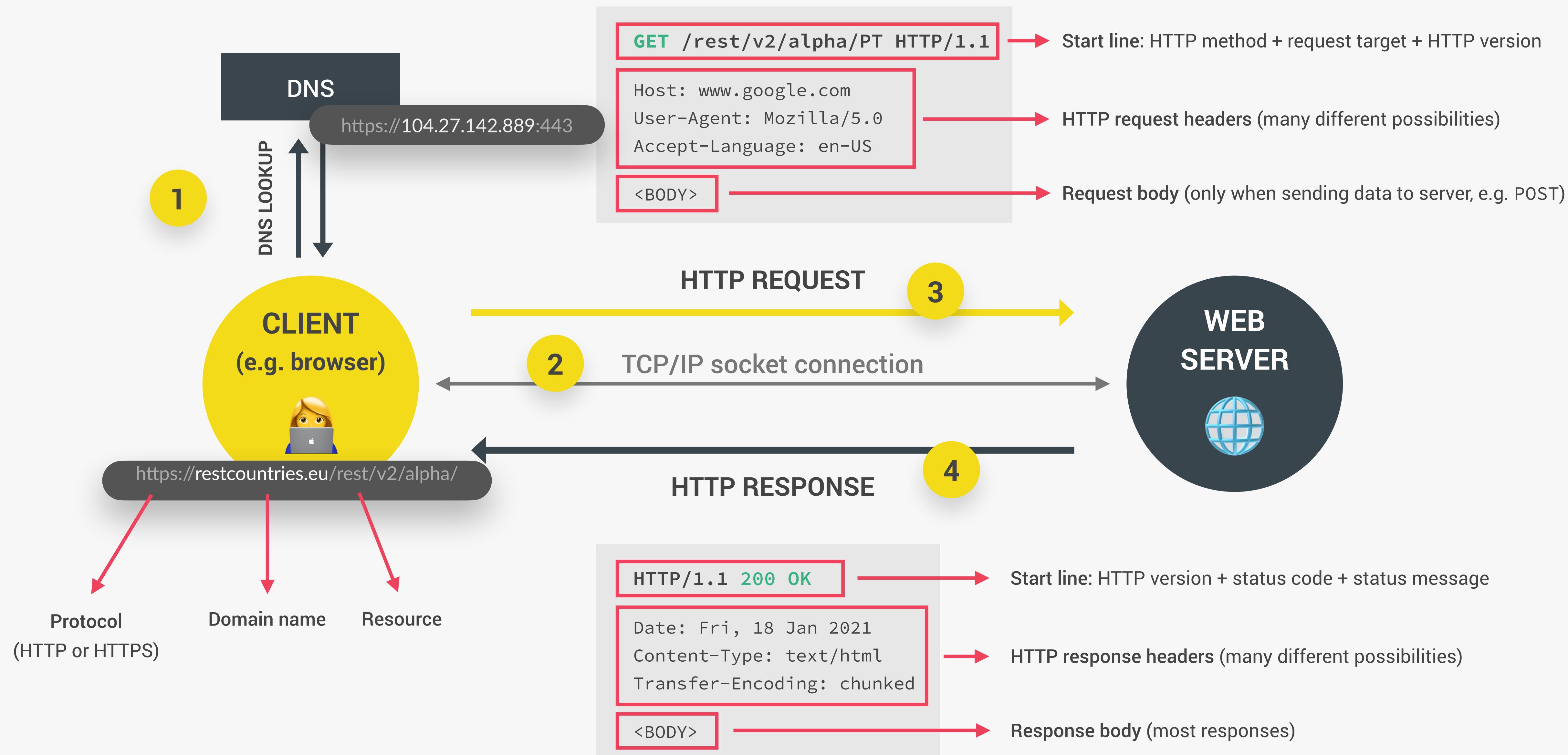
JS

# WHAT HAPPENS WHEN WE ACCESS A WEB SERVER

👉 Request-response model or Client-server architecture



# WHAT HAPPENS WHEN WE ACCESS A WEB SERVER







JONAS.IO  
SCHMEDTMANN

# THE COMPLETE JAVASCRIPT COURSE

FROM ZERO TO EXPERT!



@JONASSCHMEDTMAN

SECTION

ASYNCHRONOUS JAVASCRIPT:  
PROMISES, ASYNC/AWAIT AND AJAX

LECTURE

PROMISES AND THE FETCH API

JS

# WHAT ARE PROMISES?

## PROMISE

- 👉 **Promise:** An object that is used as a placeholder for the future result of an asynchronous operation.
  - ↓ Less formal
- 👉 **Promise:** A container for an asynchronously delivered value.
  - ↓ Less formal
- 👉 **Promise:** A container for a future value.
- 👉 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** 🎉

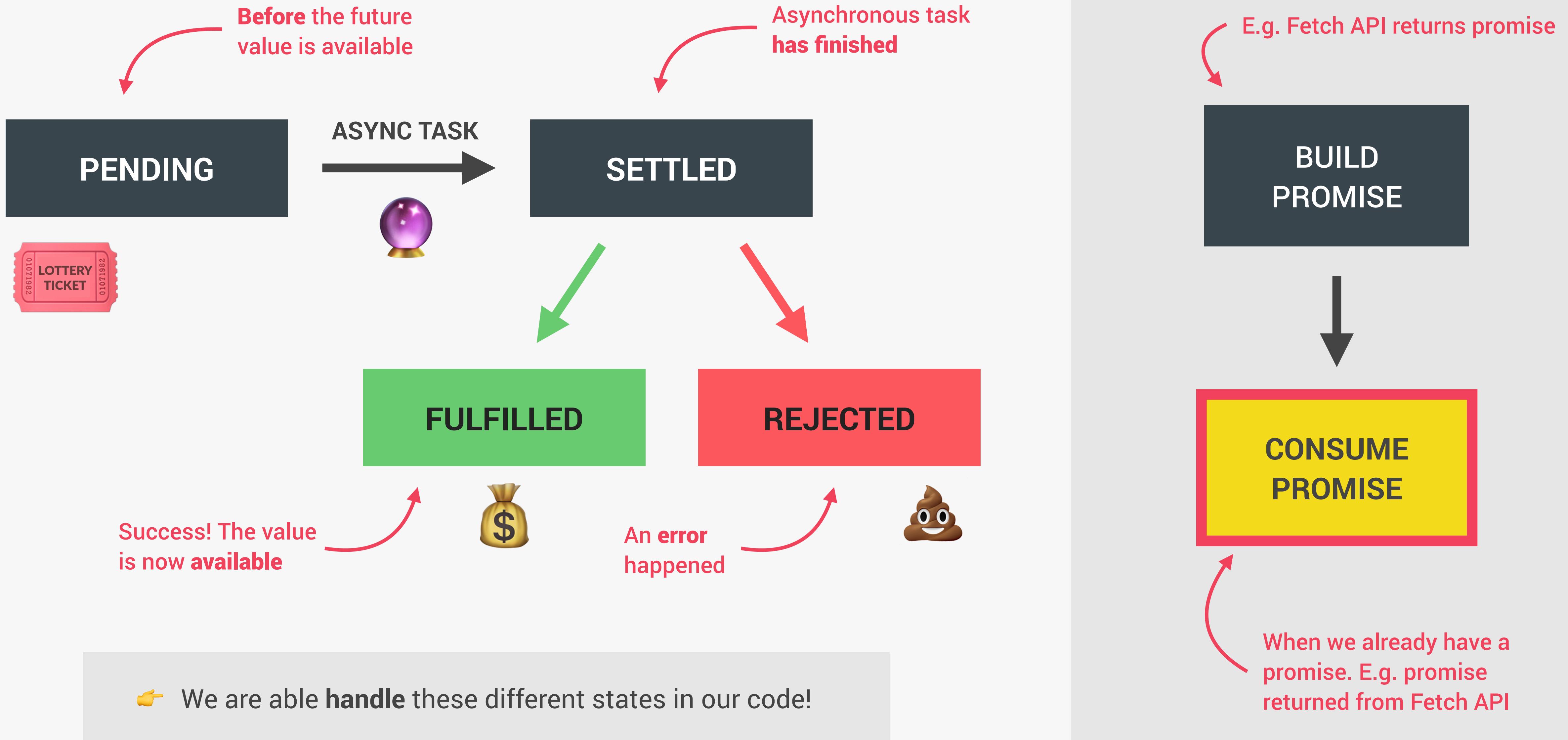
Example: Response from AJAX call



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







# THE COMPLETE JAVASCRIPT COURSE

FROM ZERO TO EXPERT!



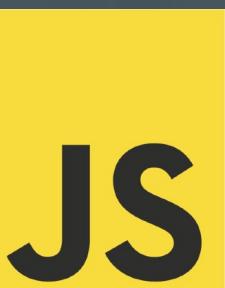
@JONASSCHMEDTMAN

SECTION

ASYNCHRONOUS JAVASCRIPT:  
PROMISES, ASYNC/AWAIT AND AJAX

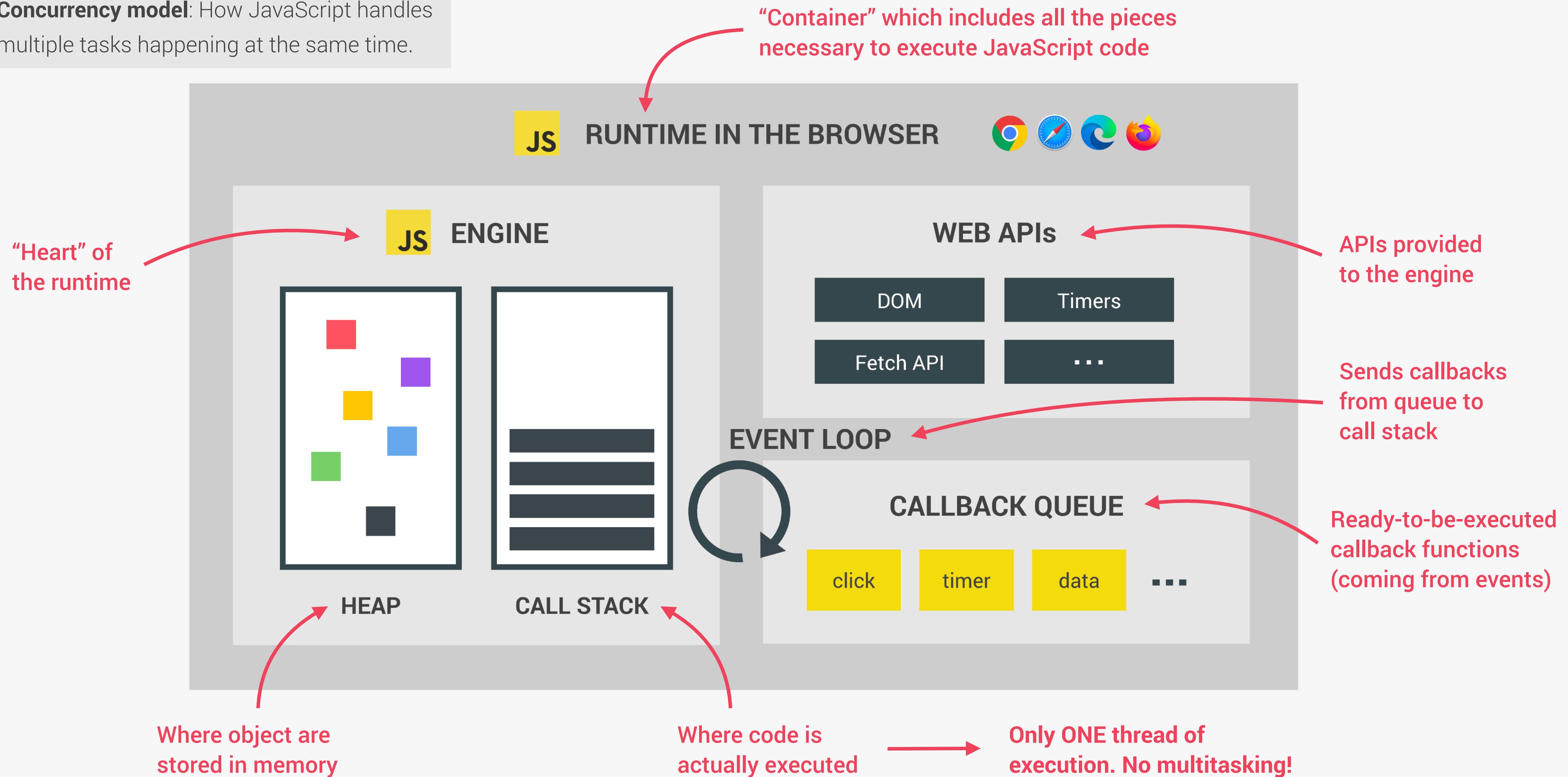
LECTURE

ASYNCHRONOUS BEHIND THE SCENES:  
THE EVENT LOOP

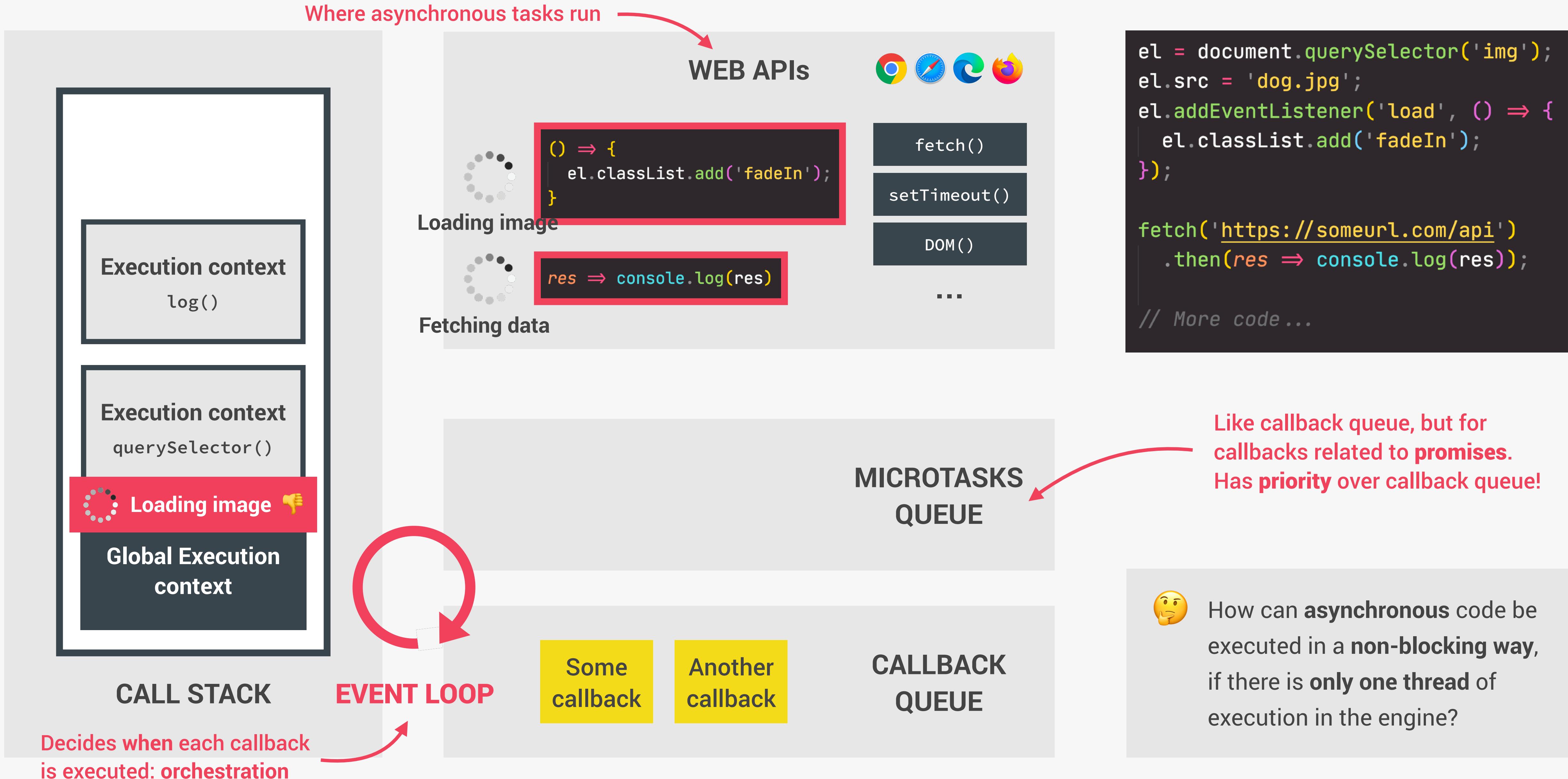


# REVIEW: JAVASCRIPT RUNTIME

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



# HOW ASYNCHRONOUS JAVASCRIPT WORKS BEHIND THE SCENES





MODERN JAVASCRIPT  
DEVELOPMENT:  
MODULES AND  
TOOLING



JONAS.IO  
SCHMEDTMANN

# THE COMPLETE JAVASCRIPT COURSE

FROM ZERO TO EXPERT!



@JONASSCHMEDTMAN

SECTION

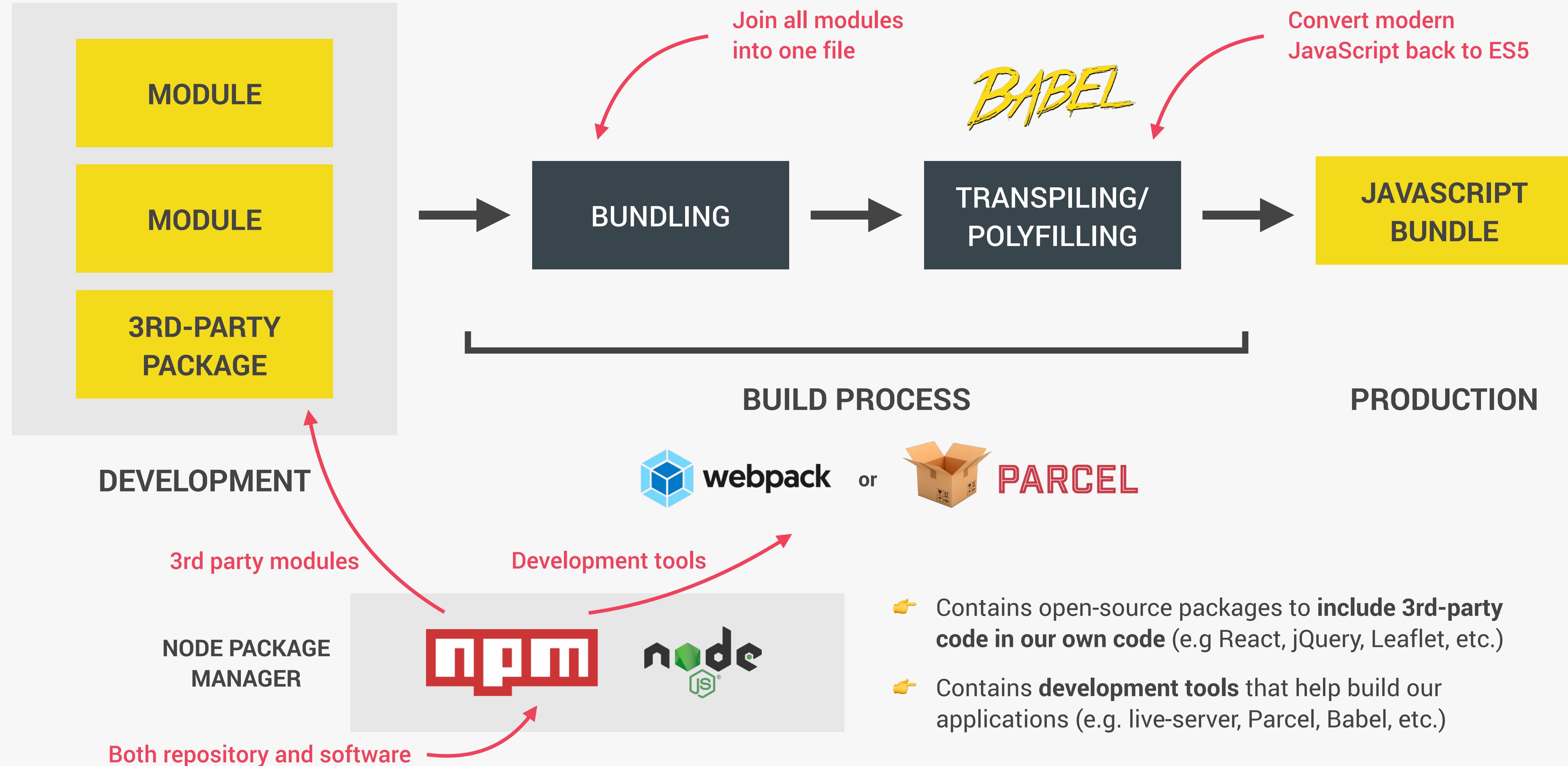
MODERN JAVASCRIPT DEVELOPMENT:  
MODULES AND TOOLING

LECTURE

AN OVERVIEW OF MODERN  
JAVASCRIPT DEVELOPMENT

JS

# MODERN JAVASCRIPT DEVELOPMENT







JONAS.IO  
SCHMEDTMANN

# THE COMPLETE JAVASCRIPT COURSE

FROM ZERO TO EXPERT!



@JONASSCHMEDTMAN

SECTION

MODERN JAVASCRIPT DEVELOPMENT:  
MODULES AND TOOLING

LECTURE

AN OVERVIEW OF MODULES IN  
JAVASCRIPT

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.

IMPORT  
(DEPENDENCY)



MODULE

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

Module code



EXPORT  
(PUBLIC API)

# NATIVE JAVASCRIPT (ES6) MODULES

## ES6 MODULES

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

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

**import and export syntax**

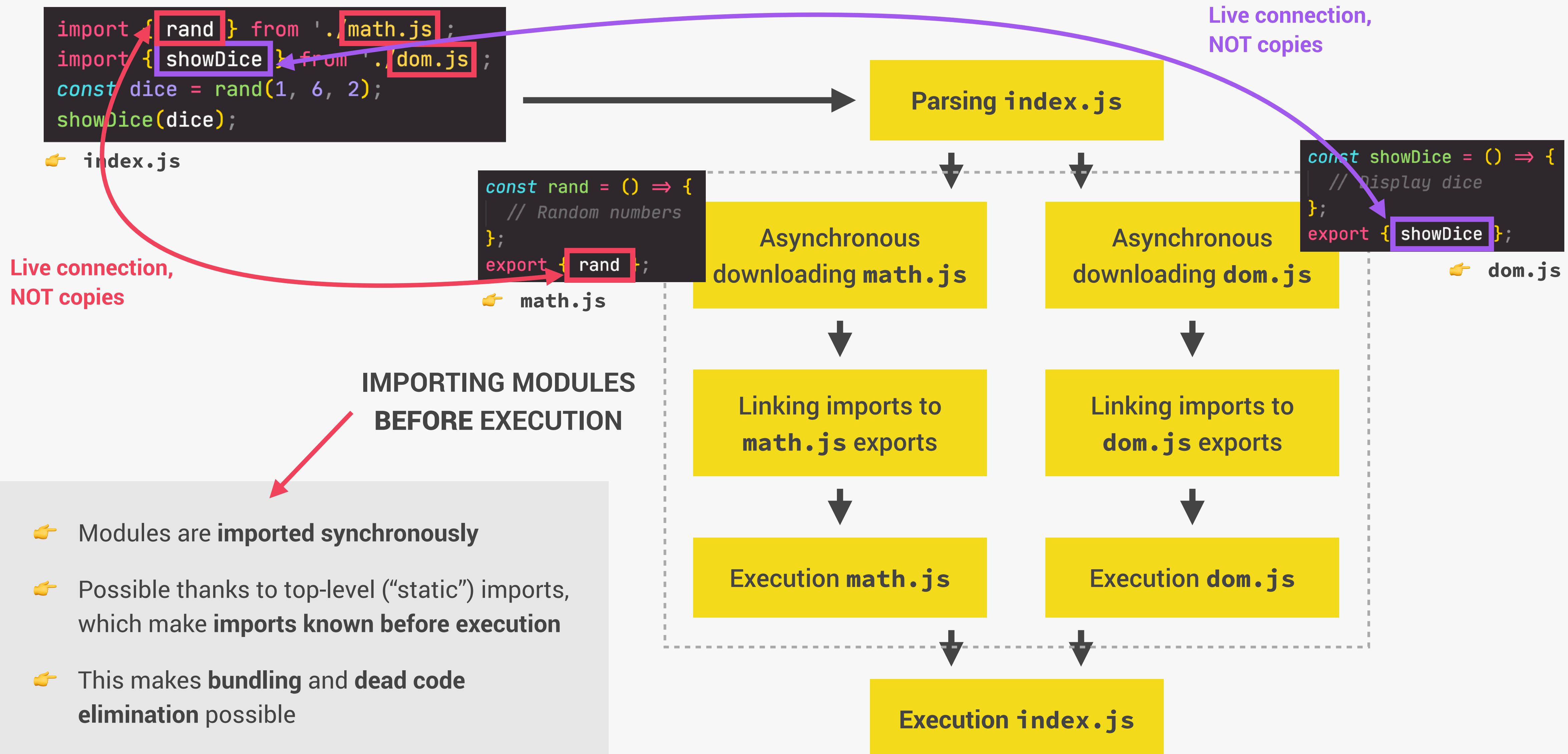
## ES6 MODULE

## 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







JONAS.IO  
SCHMEDTMANN

# THE COMPLETE JAVASCRIPT COURSE

FROM ZERO TO EXPERT!



@JONASSCHMEDTMAN

SECTION

MODERN JAVASCRIPT DEVELOPMENT:  
MODULES AND TOOLING

LECTURE

REVIEW: WRITING CLEAN AND  
MODERN JAVASCRIPT

JS

# REVIEW: MODERN AND CLEAN CODE

## READABLE CODE

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

## 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

## GENERAL

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

## OOP

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

# REVIEW: MODERN AND CLEAN CODE

## AVOID NESTED CODE

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

## ASYNCHRONOUS CODE

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





JONAS.IO  
SCHMEDTMANN

# THE COMPLETE JAVASCRIPT COURSE

FROM ZERO TO EXPERT!



@JONASSCHMEDTMAN

SECTION

MODERN JAVASCRIPT DEVELOPMENT:  
MODULES AND TOOLING

LECTURE

DECLARATIVE AND FUNCTIONAL  
JAVASCRIPT PRINCIPLES

JS

# IMPERATIVE VS. DECLARATIVE CODE

Two fundamentally different ways  
of writing code (paradigms)

IMPERATIVE

DECLARATIVE

- 👉 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
- 👉 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

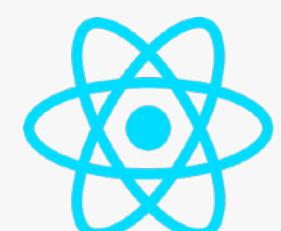
```
const arr = [2, 4, 6, 8];
const doubled = [];
for (let i = 0; i < arr.length; i++)
  doubled[i] = arr[i] * 2;
```

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

# FUNCTIONAL PROGRAMMING PRINCIPLES

## FUNCTIONAL PROGRAMMING

- 👉 **Declarative** programming paradigm
- 👉 Based on the idea of writing software by combining many **pure functions**, avoiding **side effects** and **mutating** data
- 👉 **Side effect:** Modification (mutation) of any data **outside** of the function (mutating external variables, logging to console, writing to DOM, etc.)
- 👉 **Pure function:** Function without side effects. Does not depend on external variables. **Given the same inputs, always returns the same outputs.**
- 👉 **Immutability:** State (data) is **never** modified! Instead, state is **copied** and the copy is mutated and returned.
- 👉 Examples:



React



Redux

## FUNCTIONAL PROGRAMMING TECHNIQUES

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

## DECLARATIVE SYNTAX

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



FORKIFY APP.  
BUILDING A MODERN  
APPLICATION



JONAS.IO  
SCHMEDTMANN

# THE COMPLETE JAVASCRIPT COURSE

FROM ZERO TO EXPERT!



@JONASSCHMEDTMAN

SECTION

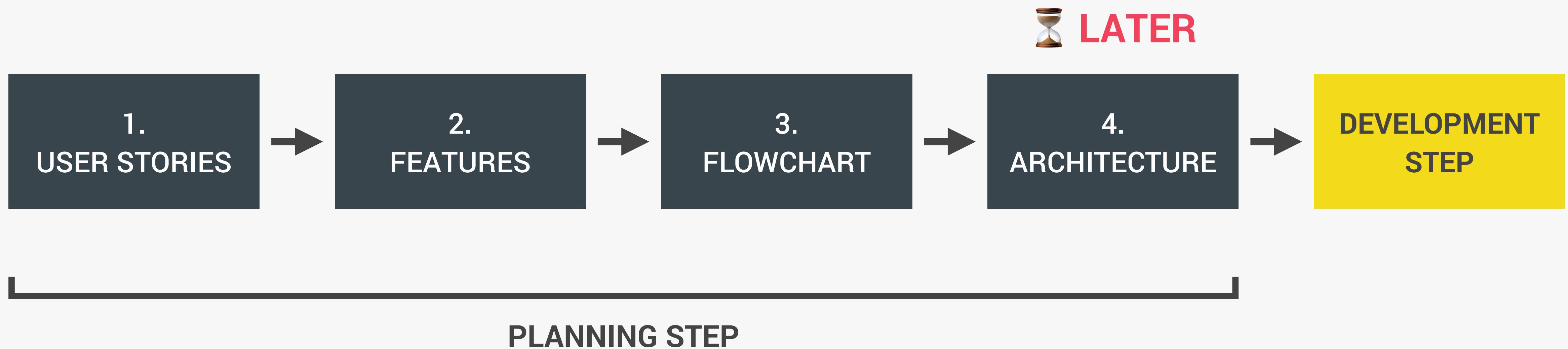
FORKIFY APP: BUILDING A MODERN  
APPLICATION

LECTURE

PROJECT OVERVIEW AND PLANNING

JS

# PROJECT PLANNING



# 1. USER STORIES



- 👉 **User story:** Description of the application's functionality from the user's perspective.
- 👉 **Common format:** As a *[type of user]*, I want *[an action]* so that *[a benefit]*

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

## 2. FEATURES



### USER STORIES

### FEATURES

1 Search for recipes

- 👉 Search functionality: input field to send request to API with searched keywords

2 Update the number of servings

- 👉 Display results with pagination

- 👉 Display recipe with cooking time, servings and ingredients

3 Bookmark recipes

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

4 Create my own recipes

- 👉 Bookmarking functionality: display list of all bookmarked recipes

5 See my bookmarks and own recipes  
when I leave the app and come back later

- 👉 User can upload own recipes

- 👉 User recipes will automatically be bookmarked

- 👉 User can only see their own recipes, not recipes from other users

- 👉 Store bookmark data in the browser using local storage

- 👉 On page load, read saved bookmarks from local storage and display

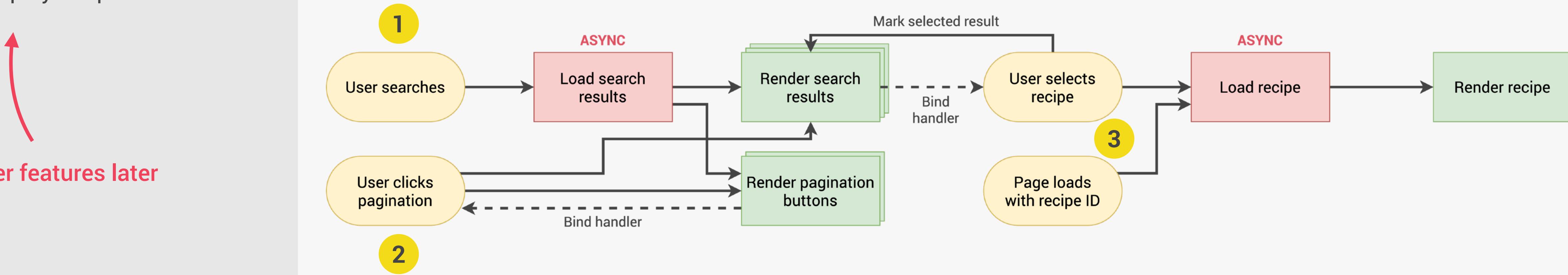
# 3. FLOWCHART (PART 1)



## FEATURES

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

Other features later







# THE COMPLETE JAVASCRIPT COURSE

FROM ZERO TO EXPERT!



@JONASSCHMEDTMAN

SECTION

FORKIFY APP: BUILDING A MODERN  
APPLICATION

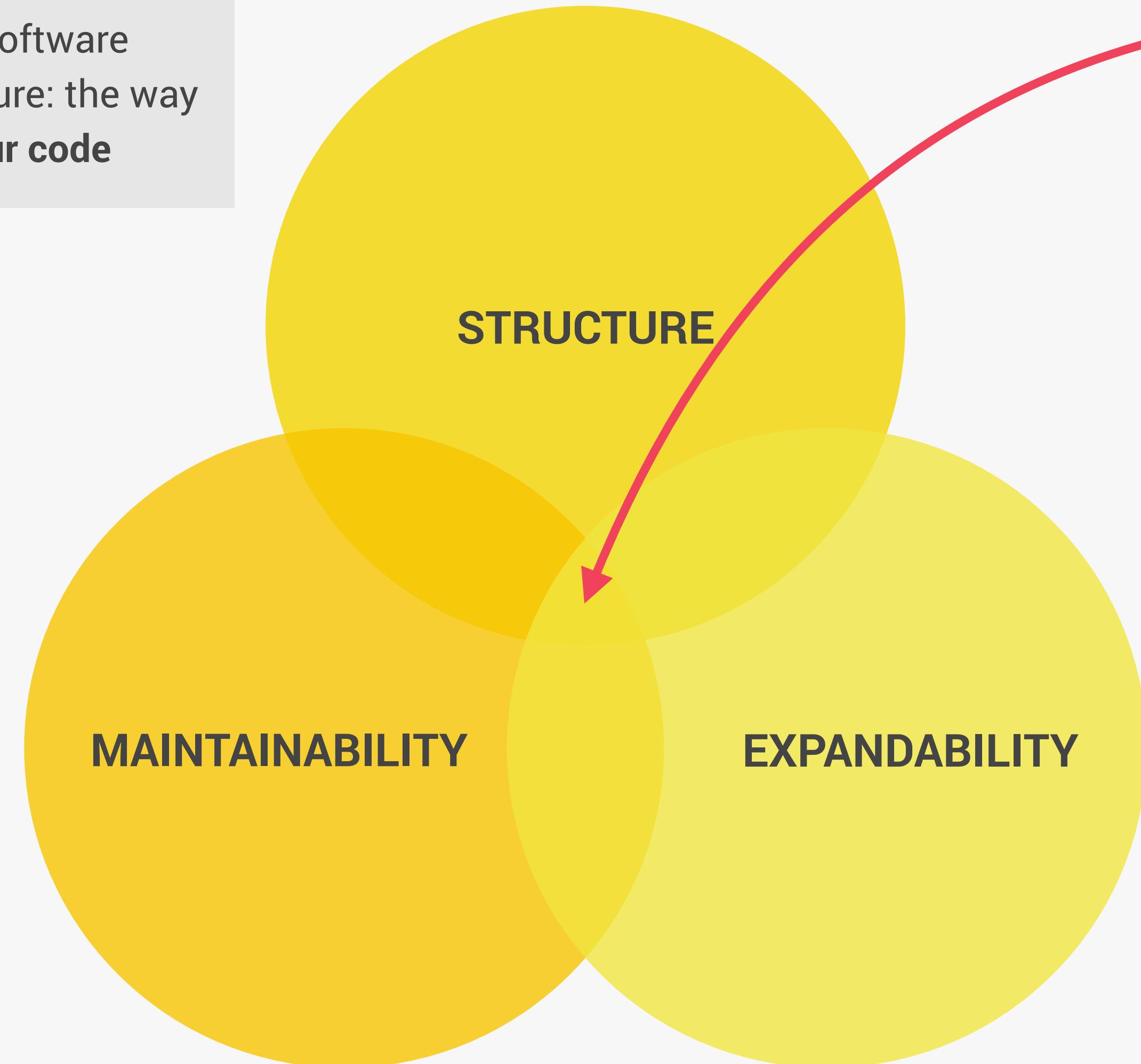
LECTURE

THE MVC ARCHITECTURE

JS

# WHY WORRY ABOUT ARCHITECTURE?

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



**The perfect architecture**

👉 We can create our own architecture (**Marty 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**

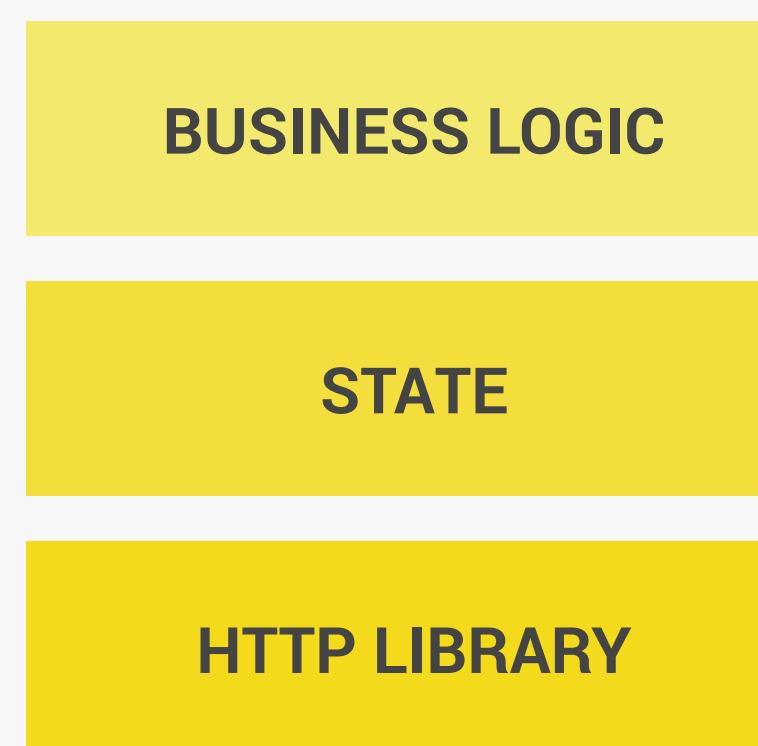
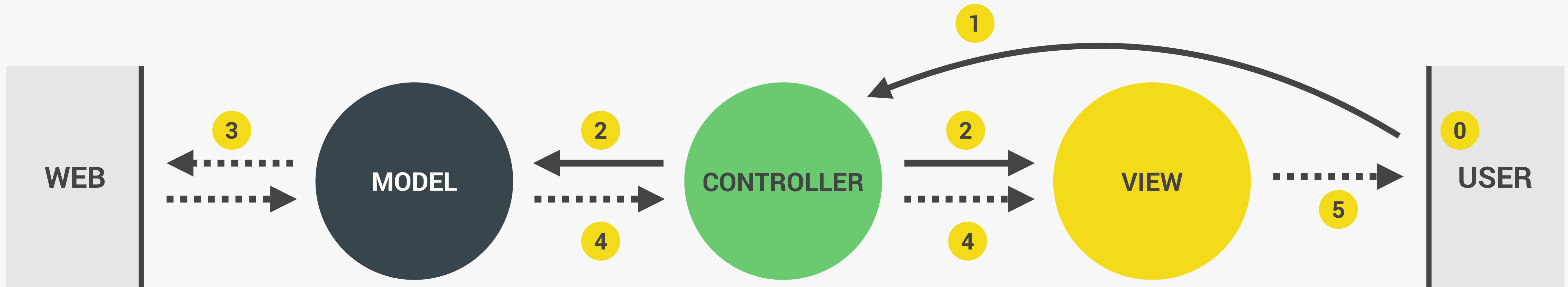
👉 We also need to be able to easily **add new features**

# COMPONENTS OF ANY ARCHITECTURE

BUSINESS LOGIC	STATE	HTTP LIBRARY	APPLICATION LOGIC (ROUTER)	PRESENTATION LOGIC (UI LAYER)
<ul style="list-style-type: none"><li>👉 Code that <b>solves the actual business problem</b>;</li><li>👉 Directly related to what business does and what it needs;</li><li>👉 <b>Example:</b> sending messages, storing transactions, calculating taxes, ...</li></ul>	<ul style="list-style-type: none"><li>👉 Essentially <b>stores all the data</b> about the application</li><li>👉 Should be the “single source of truth”</li><li>👉 UI should be kept in sync with the state</li><li>👉 State libraries exist</li></ul>  	<ul style="list-style-type: none"><li>👉 Responsible for making and receiving AJAX requests</li><li>👉 Optional but almost always necessary in real-world apps</li></ul>	<ul style="list-style-type: none"><li>👉 Code that is only concerned about the <b>implementation of application itself</b>;</li><li>👉 Handles navigation and UI events</li></ul>	<ul style="list-style-type: none"><li>👉 Code that is concerned about the <b>visible part</b> of the application</li><li>👉 Essentially displays application state</li></ul>

Keeping in sync

# THE MODEL-VIEW-CONTROLLER (MVC) ARCHITECTURE

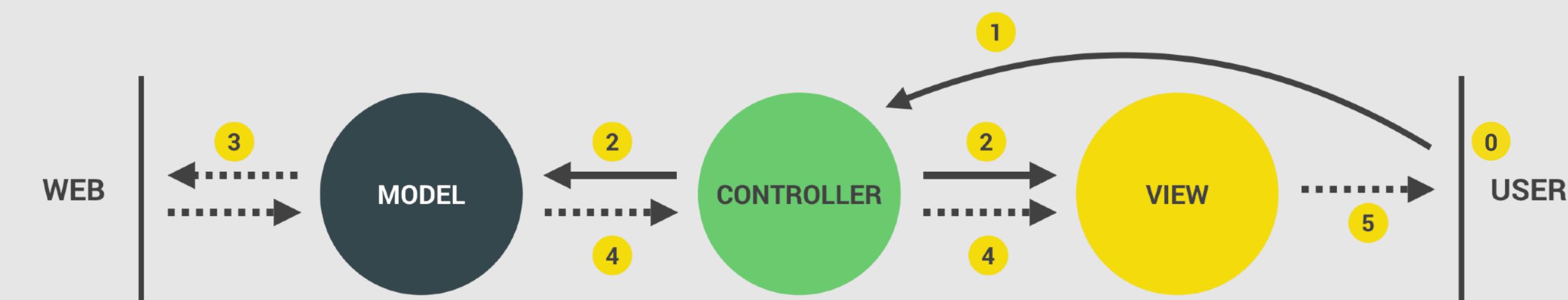
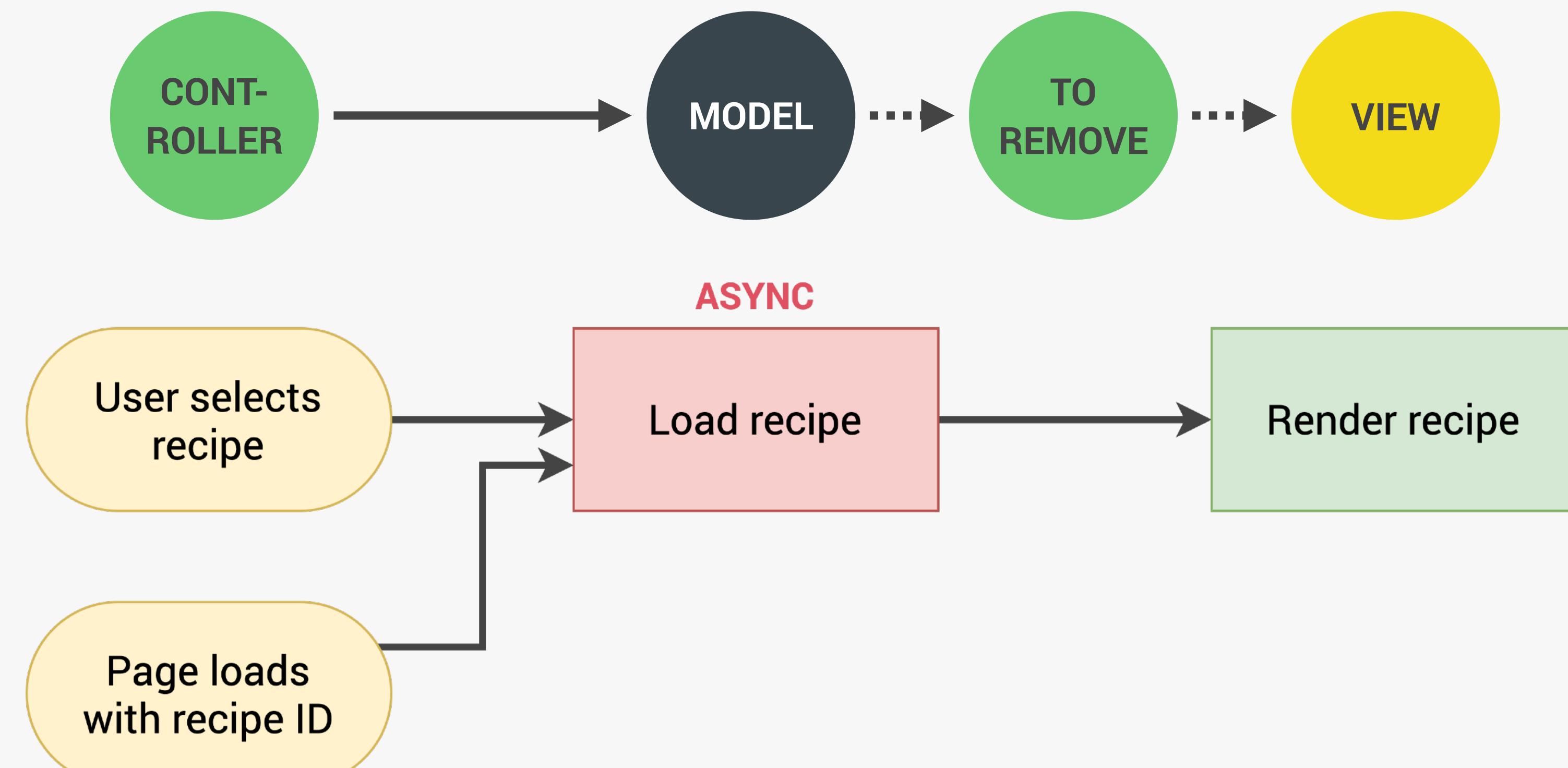


- 👉 Bridge between model and views (which don't know about one another)
- 👉 Handles UI events and **dispatches tasks to model and view**

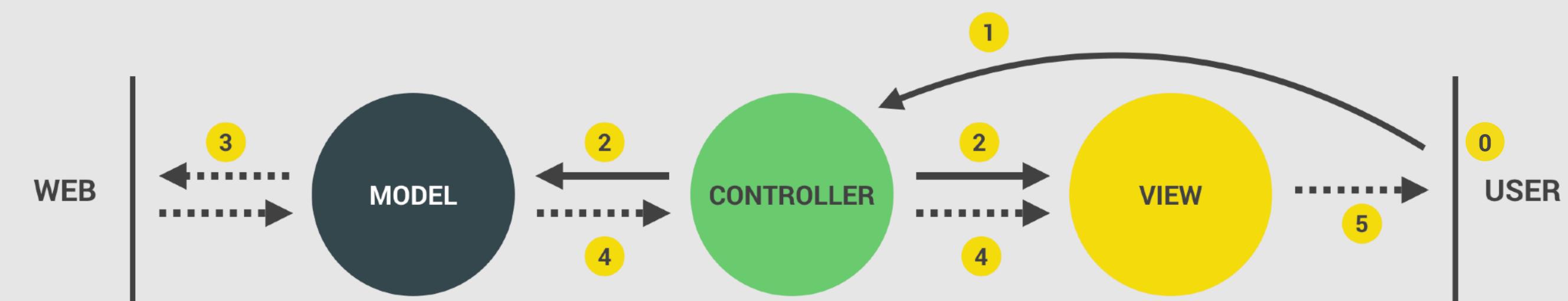
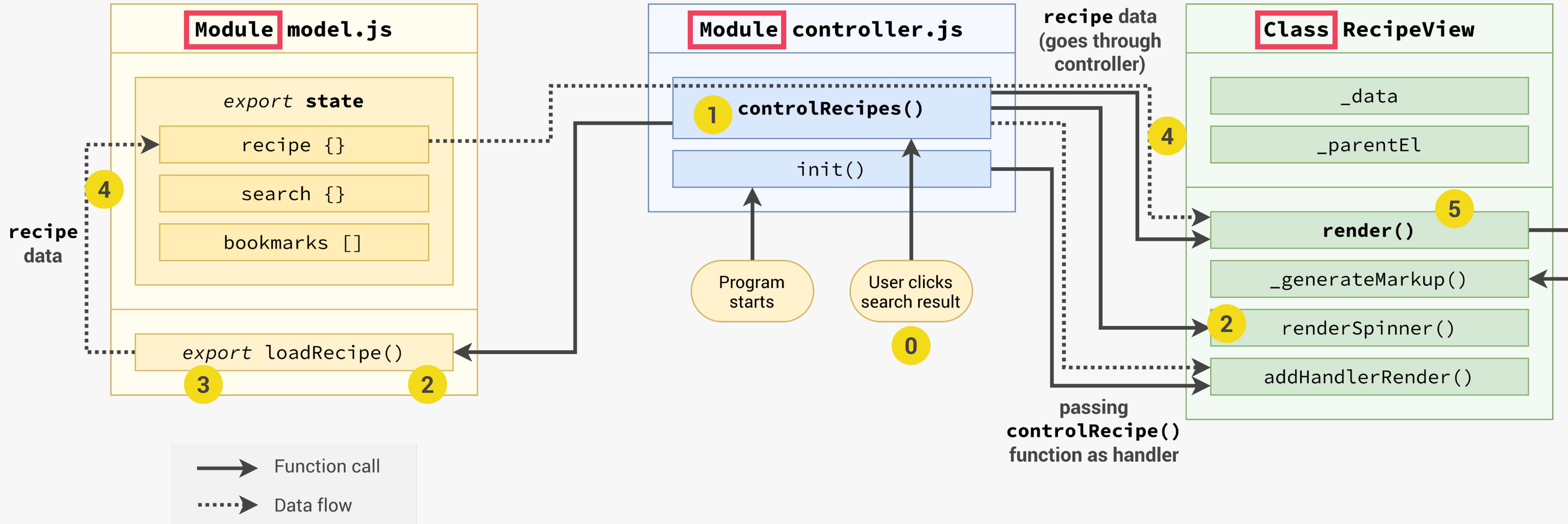


→ Connected by function call and import  
→ Data flow

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



# MVC IMPLEMENTATION (RECIPE DISPLAY ONLY)







# THE COMPLETE JAVASCRIPT COURSE

FROM ZERO TO EXPERT!



@JONASSCHMEDTMAN

SECTION

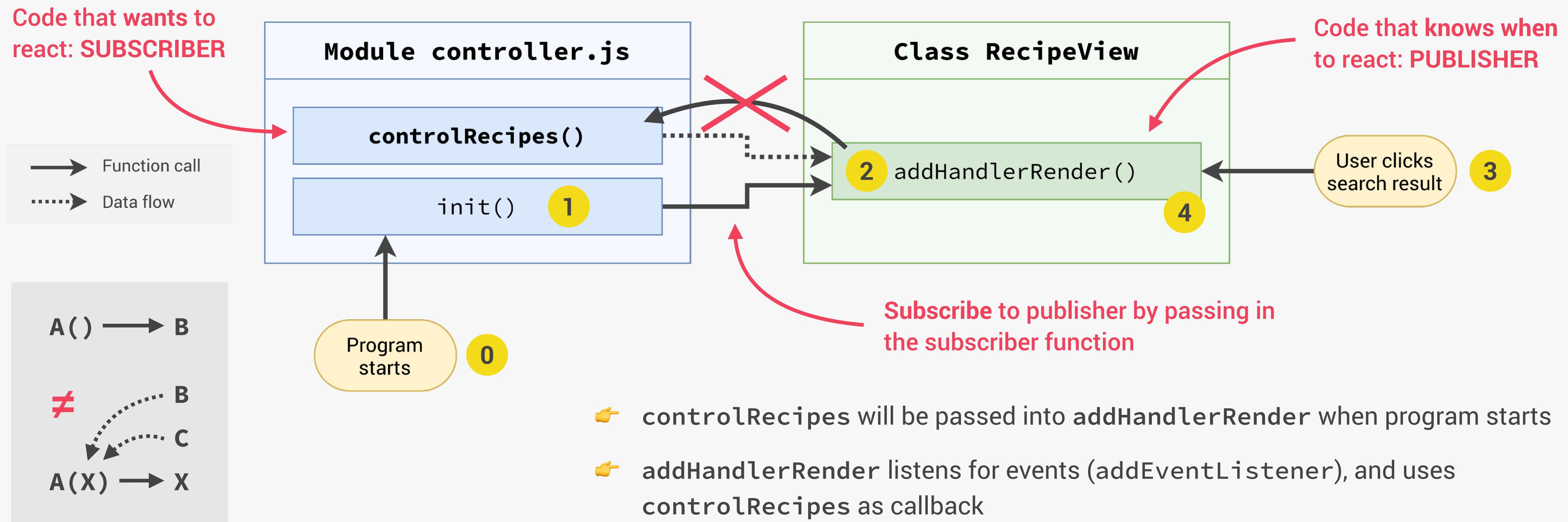
FORKIFY APP: BUILDING A MODERN  
APPLICATION

LECTURE

EVENT HANDLERS IN MVC:  
PUBLISHER-SUBSCRIBER PATTERN

JS

# EVENT HANDLING IN MVC: PUBLISHER-SUBSCRIBER PATTERN



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





# THE COMPLETE JAVASCRIPT COURSE

FROM ZERO TO EXPERT!



@JONASSCHMEDTMAN

SECTION

FORKIFY APP: BUILDING A MODERN  
APPLICATION

LECTURE

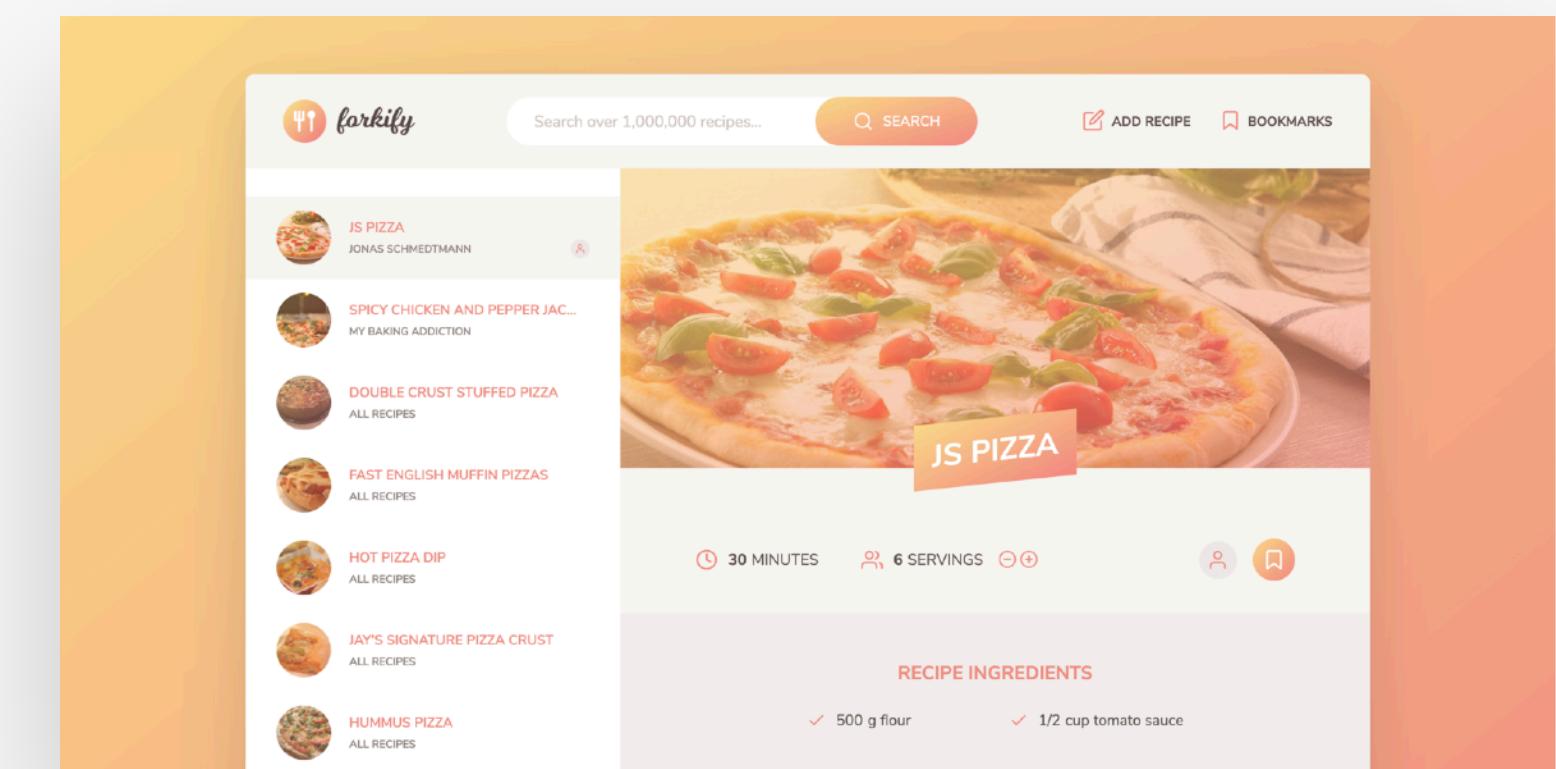
WRAPPING UP: FINAL  
CONSIDERATIONS

JS

# IMPROVEMENT AND FEATURE IDEAS: CHALLENGES 😎



- 👉 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**