

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

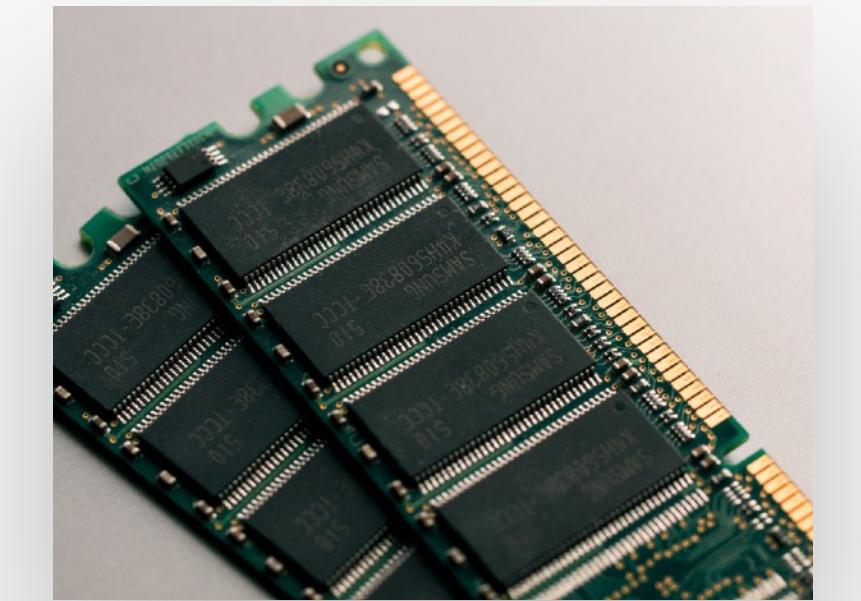
First-class functions

Dynamic

Single-threaded

Non-blocking event loop

👉 Any computer program needs resources:



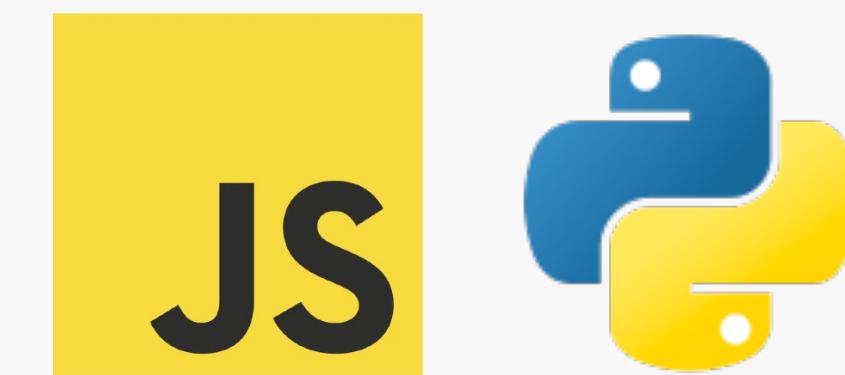
+



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

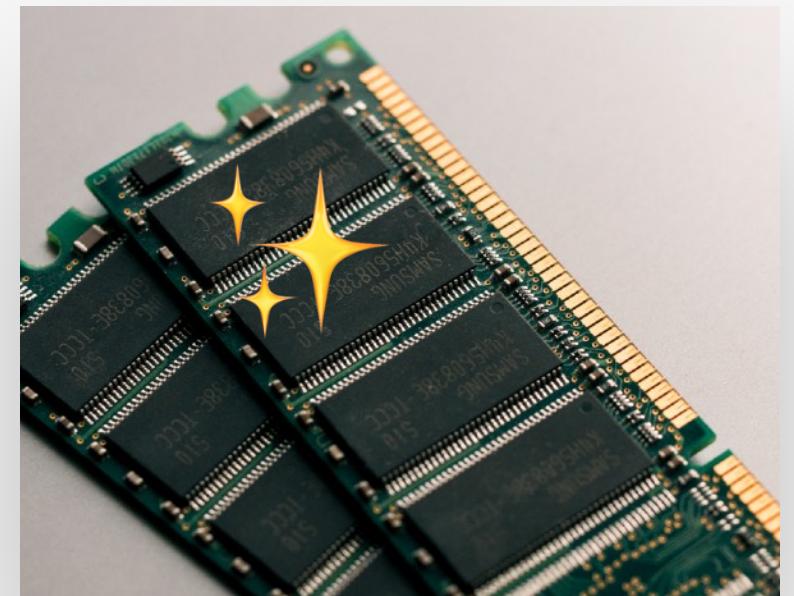
Prototype-based object-oriented

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
11101010111010100111010101010101010101010101011100111010
0111010110101010010101010101110101010101010101010101010
1110101001110101010111010101010101010101010101010101010
```

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

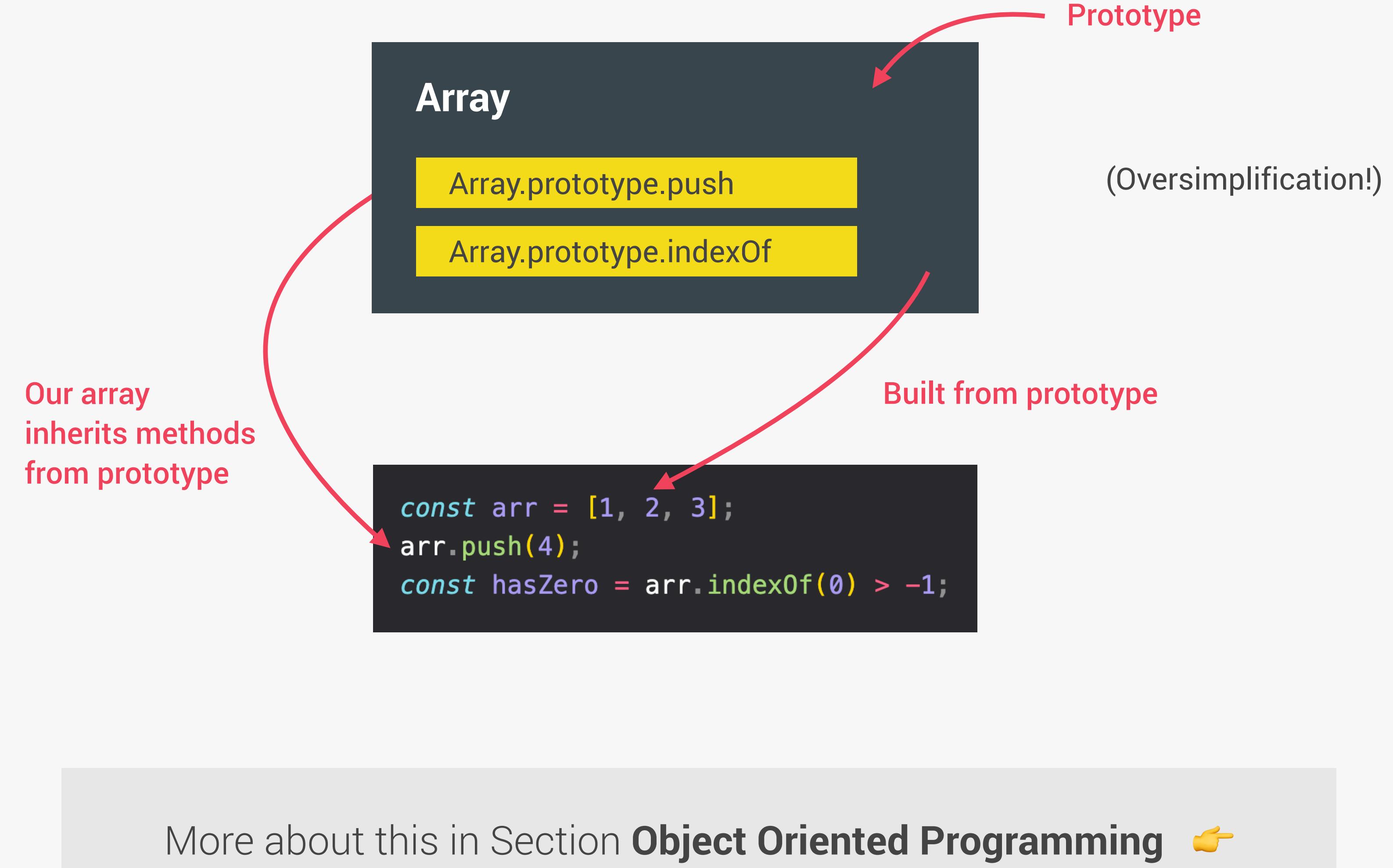
- 1 Procedural programming
- 2 Object-oriented programming (OOP)
- 3 Functional programming (FP)

👉 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

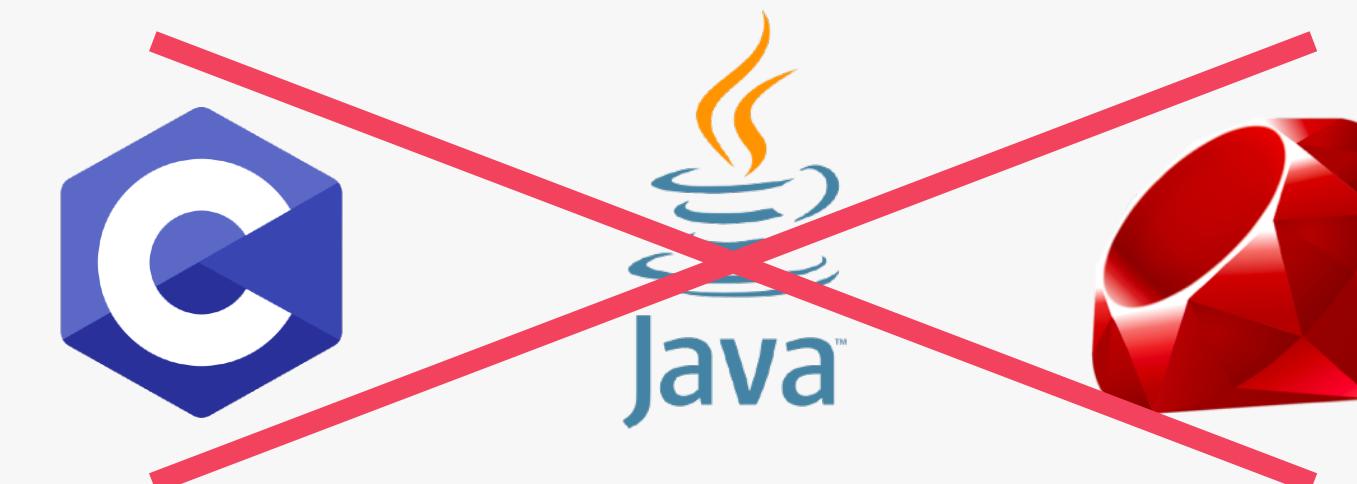
👉 **Dynamically-typed language:**

No data type definitions. Types becomes known at runtime

Data type of variable is automatically changed

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

We don't know the type of a var until it is run by the browser engine



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

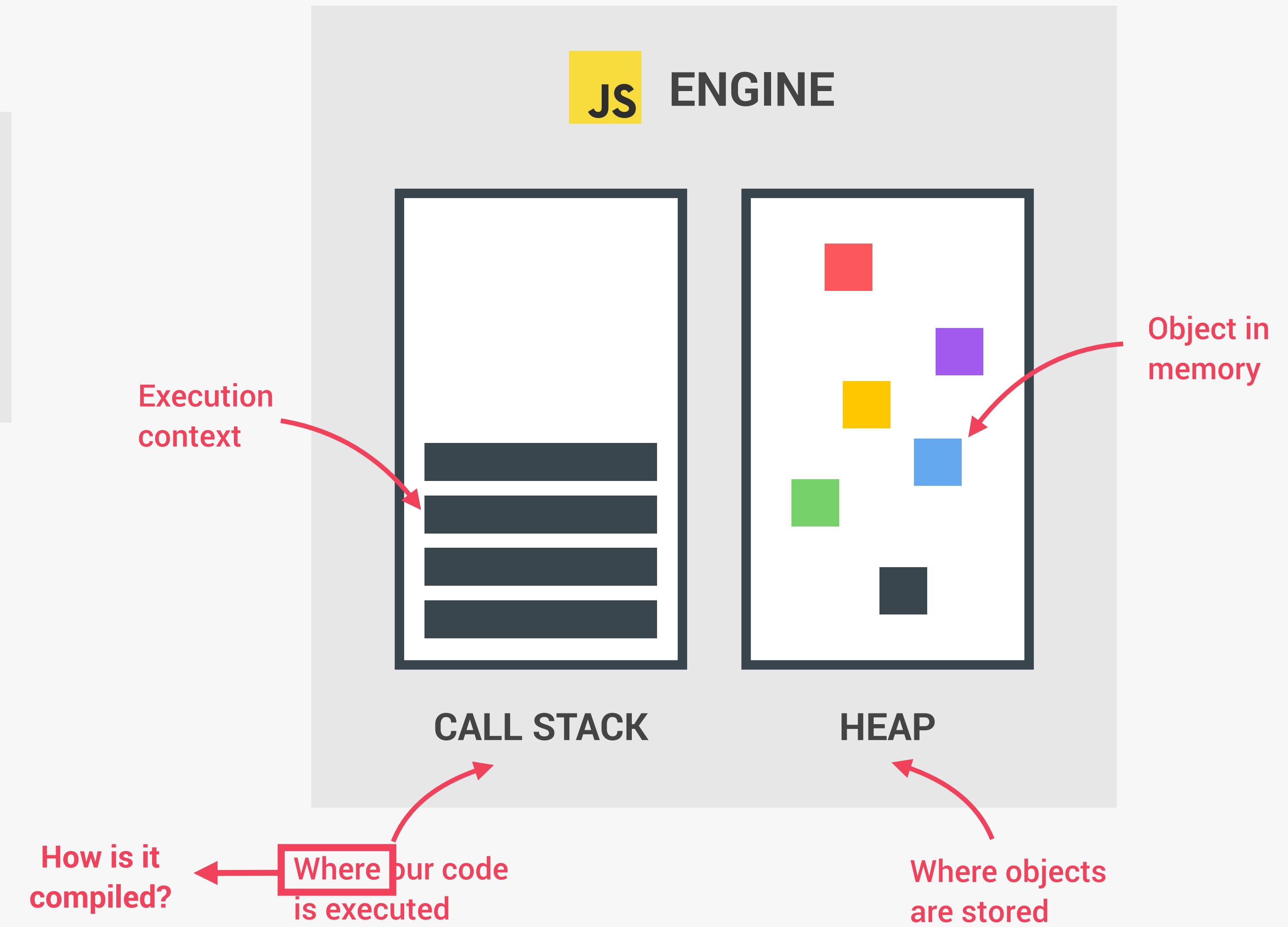
THE JAVASCRIPT ENGINE AND
RUNTIME

JS

WHAT IS A JAVASCRIPT ENGINE?



👉 Example: V8 Engine



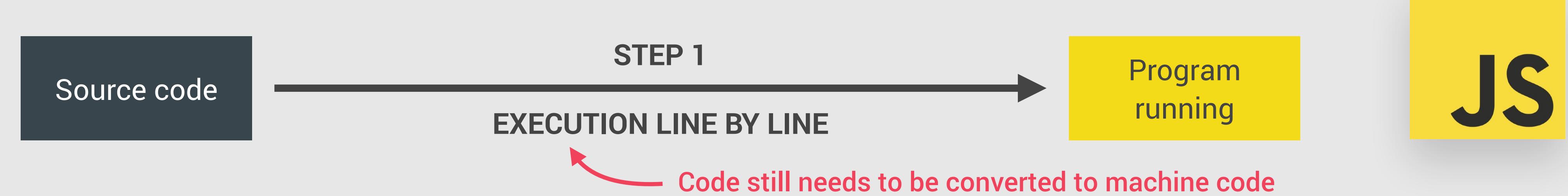
COMPUTER SCIENCE SIDENOTE: COMPIRATION VS. INTERPRETATION



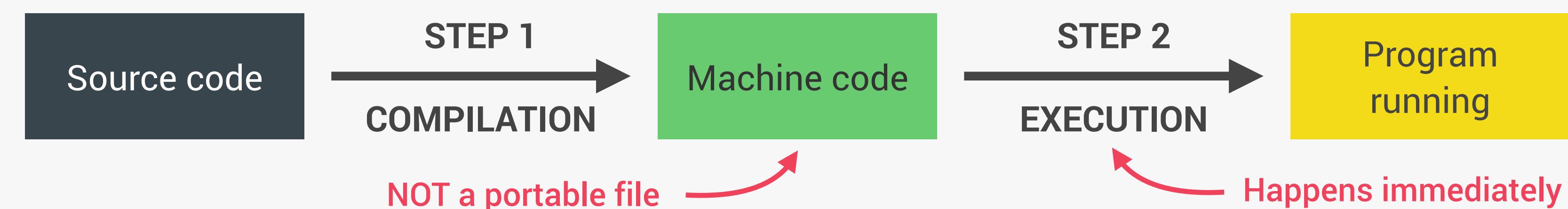
- 👉 **Compilation:** Entire code is converted into machine code at once, and written to a binary file that can be executed by a computer.



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

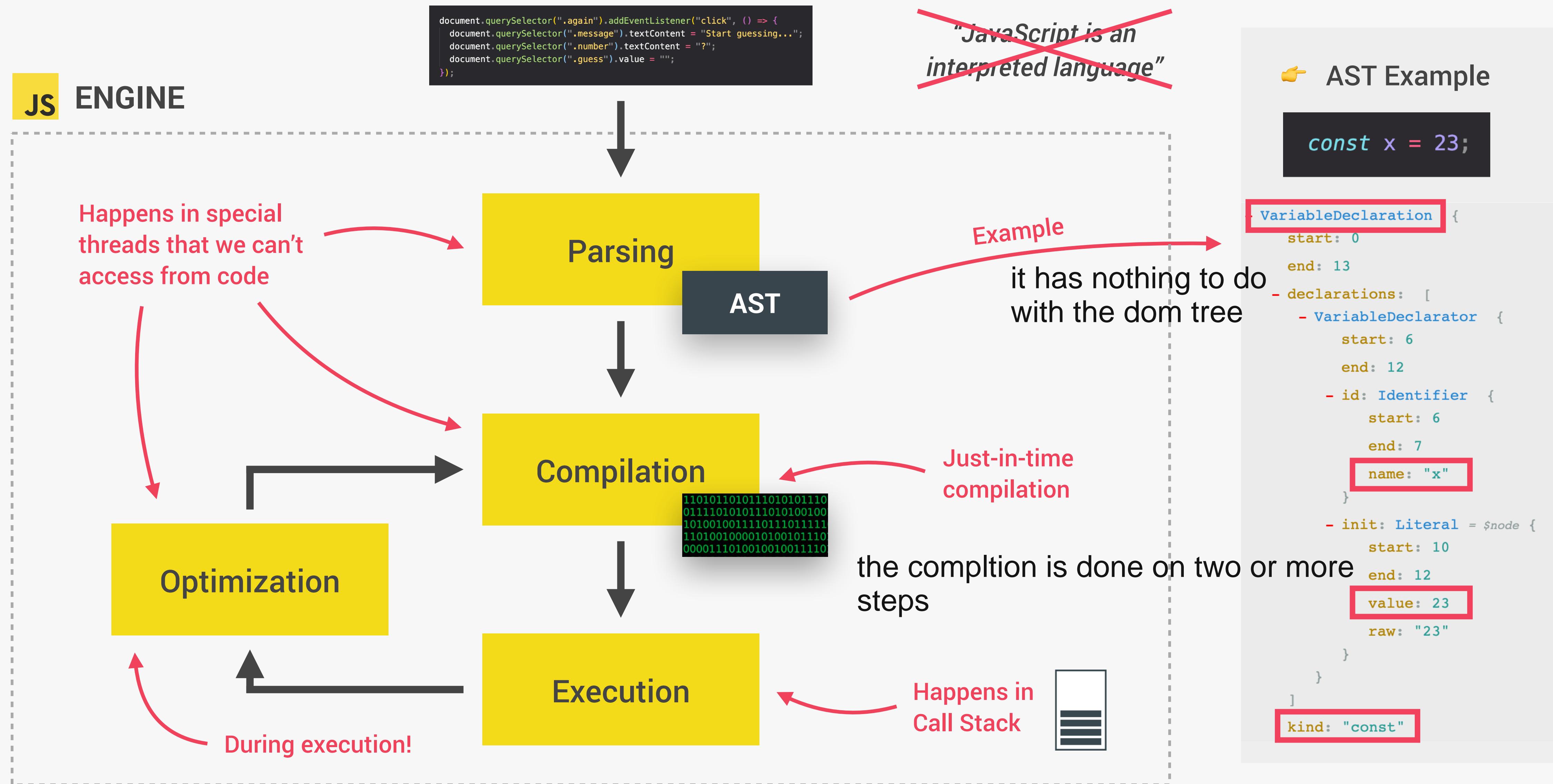


- 👉 **Just-in-time (JIT) compilation:** Entire code is converted into machine code at once, then executed immediately.

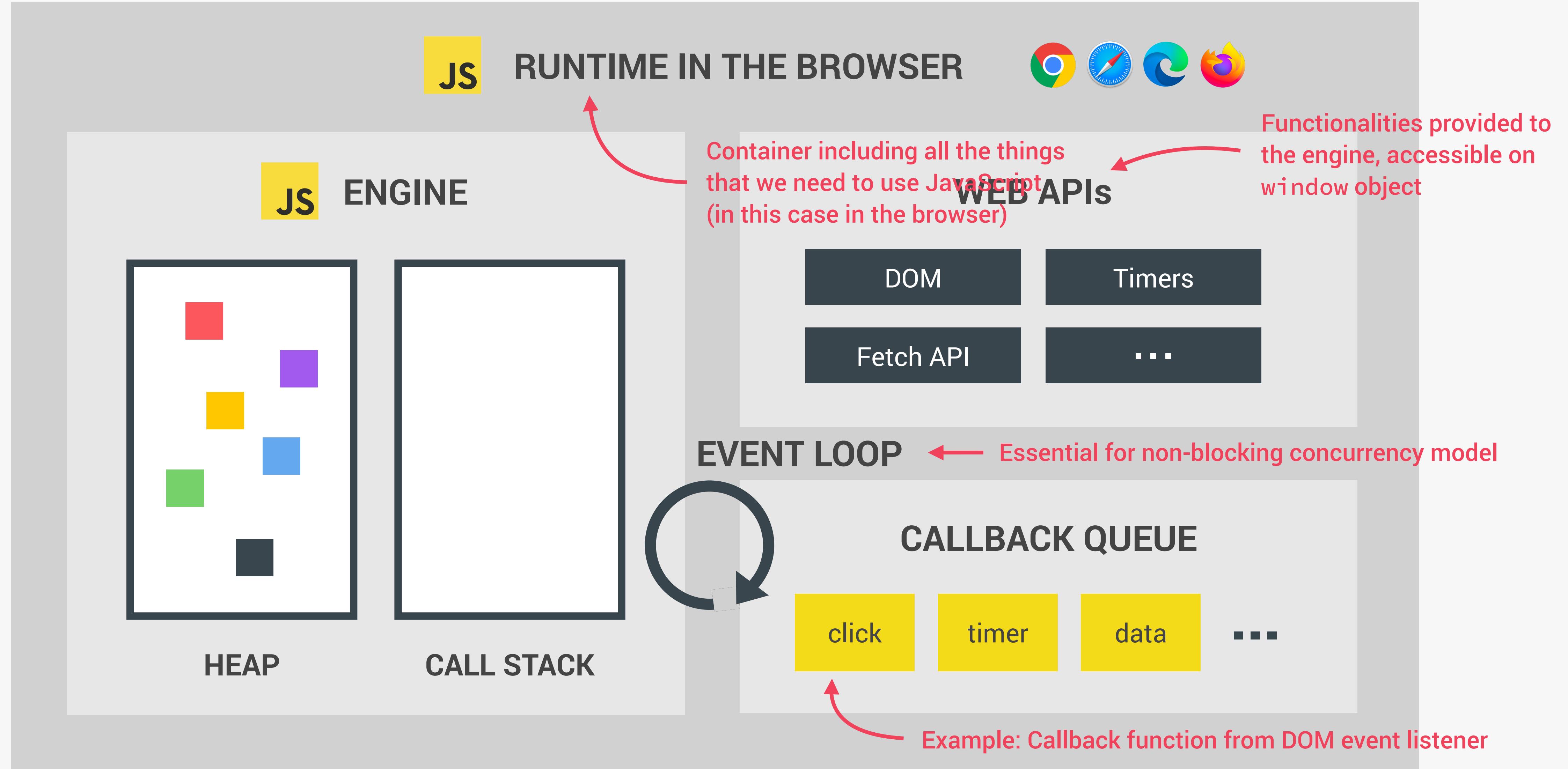


the difference between compilation and jit is that the second one executes right after it is done there is no need to am

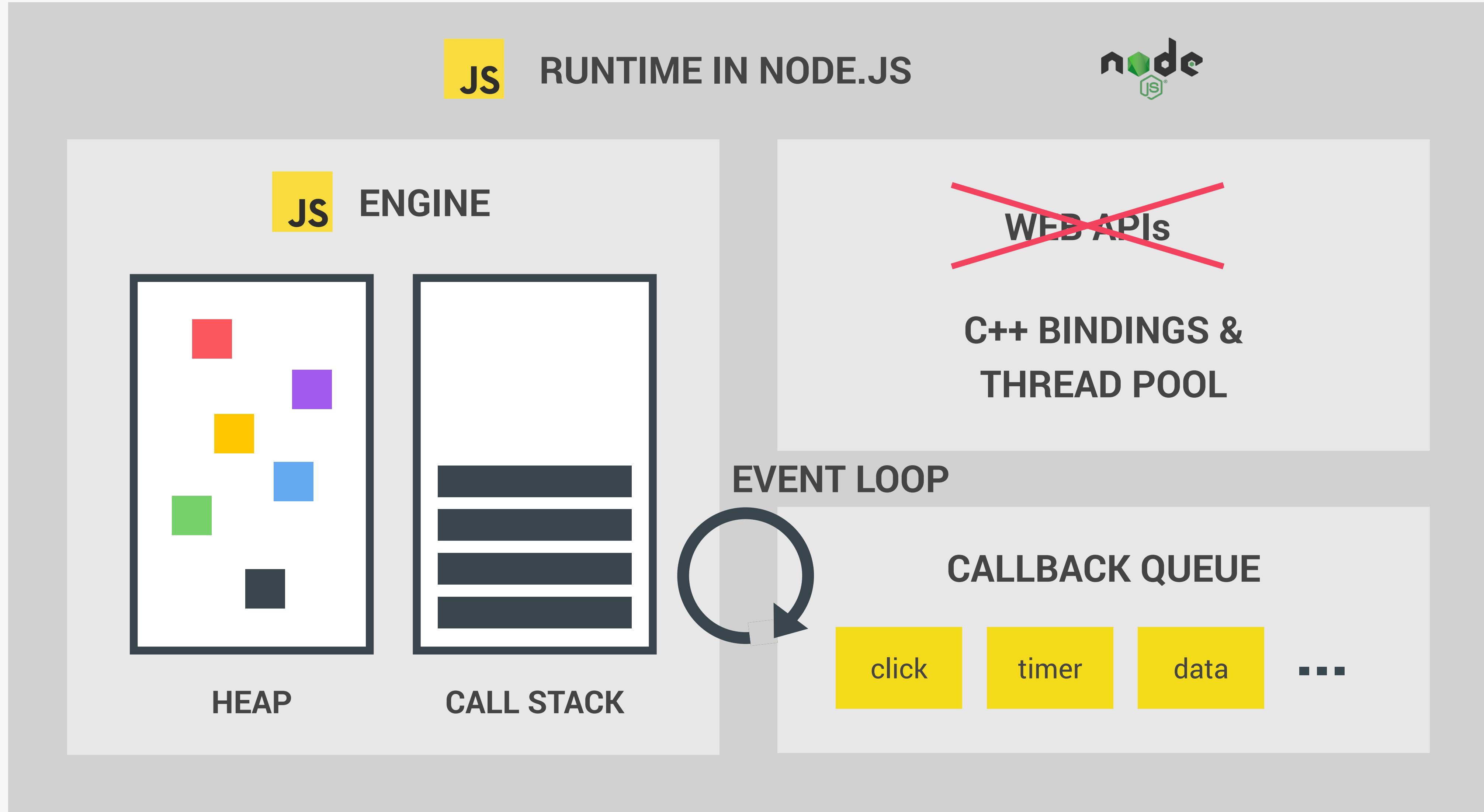
MODERN JUST-IN-TIME COMPIRATION OF JAVASCRIPT



THE BIGGER PICTURE: JAVASCRIPT RUNTIME



THE BIGGER PICTURE: JAVASCRIPT RUNTIME





JONAS.IO
SCHMEDTMANN

THE COMPLETE JAVASCRIPT COURSE

FROM ZERO TO EXPERT!



@JONASSCHMEDTMAN

SECTION

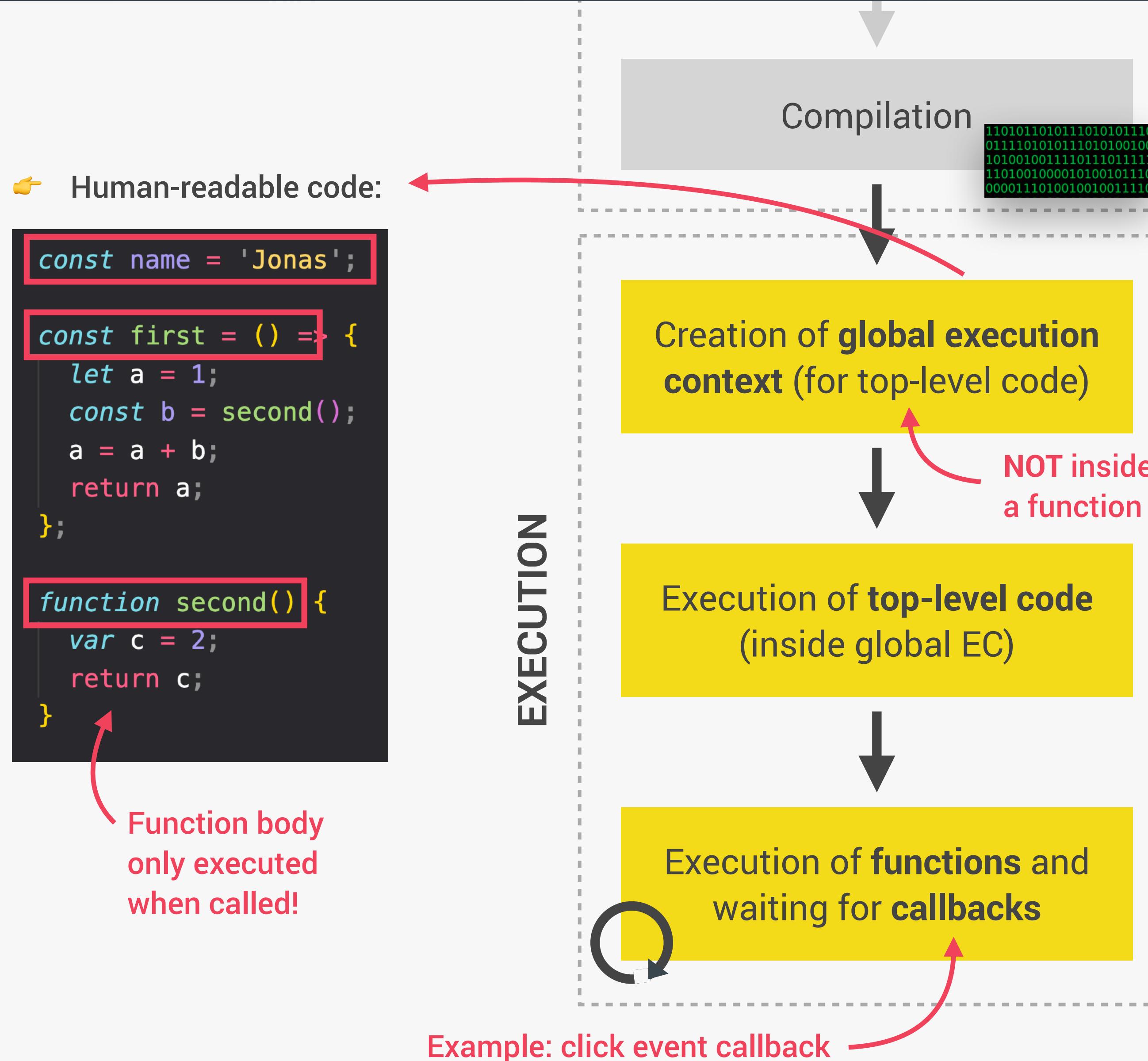
HOW JAVASCRIPT WORKS BEHIND THE
SCENES

LECTURE

EXECUTION CONTEXTS AND THE
CALL STACK

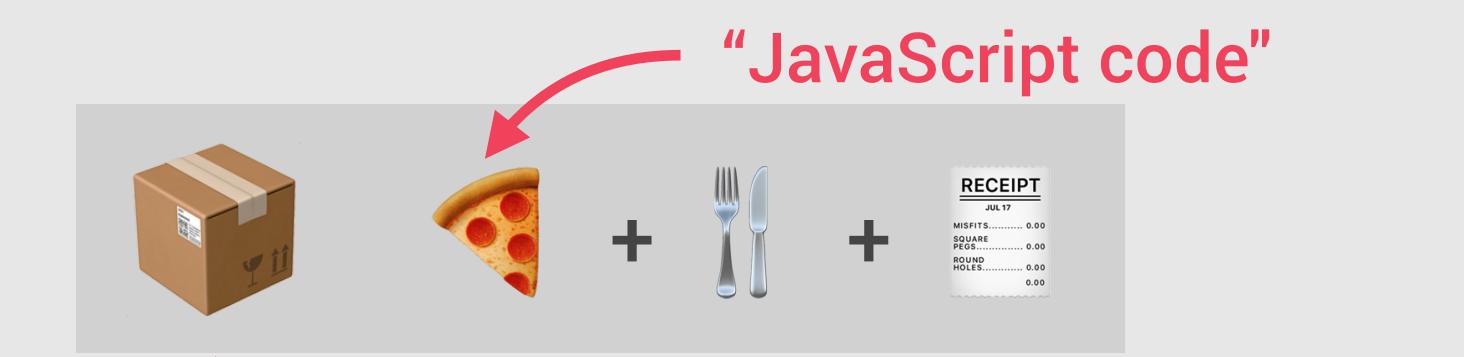
JS

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

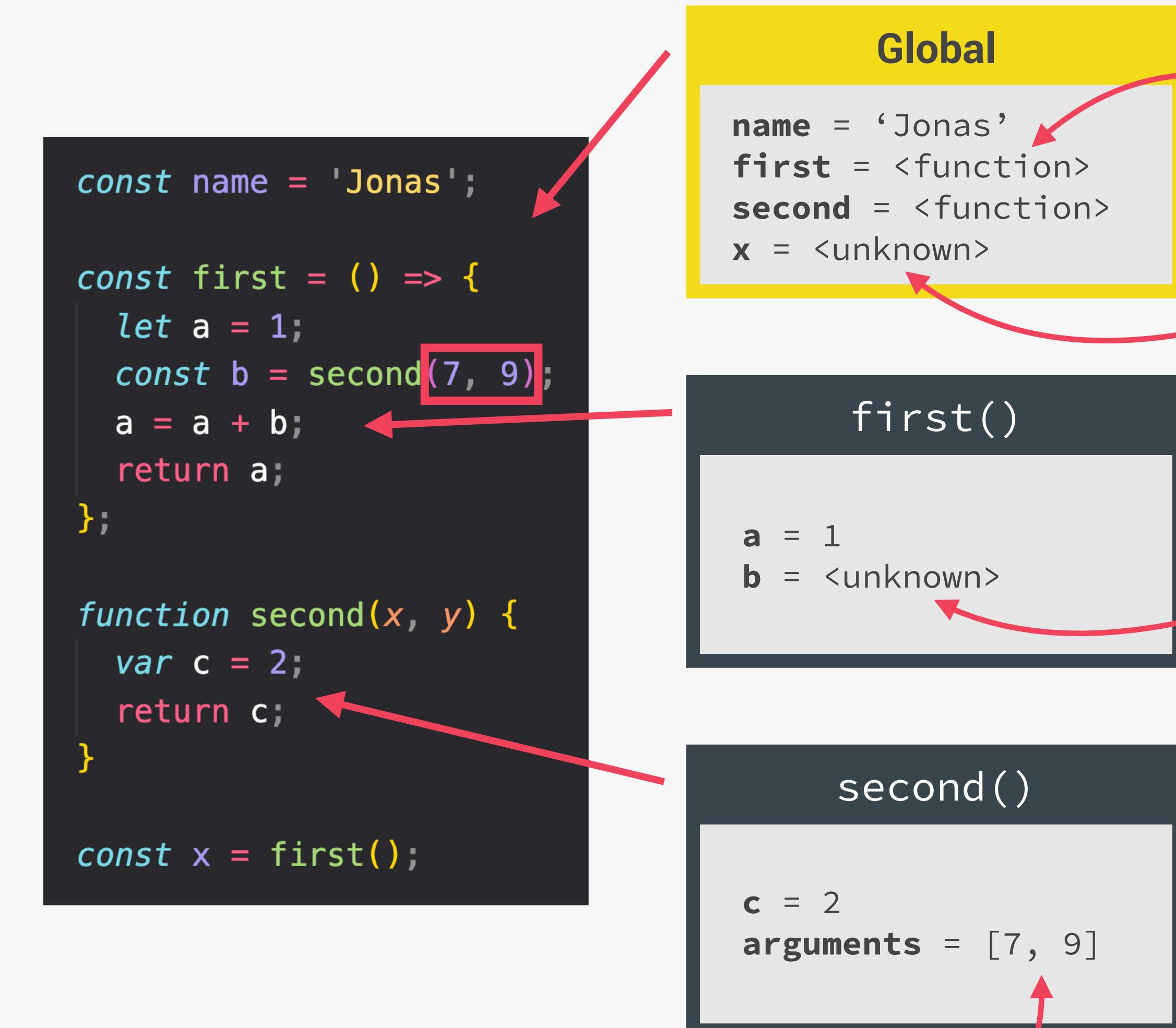
- 👉 let, const and var declarations
- 👉 Functions
- 👉 ~~arguments~~ object

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)

THE CALL STACK

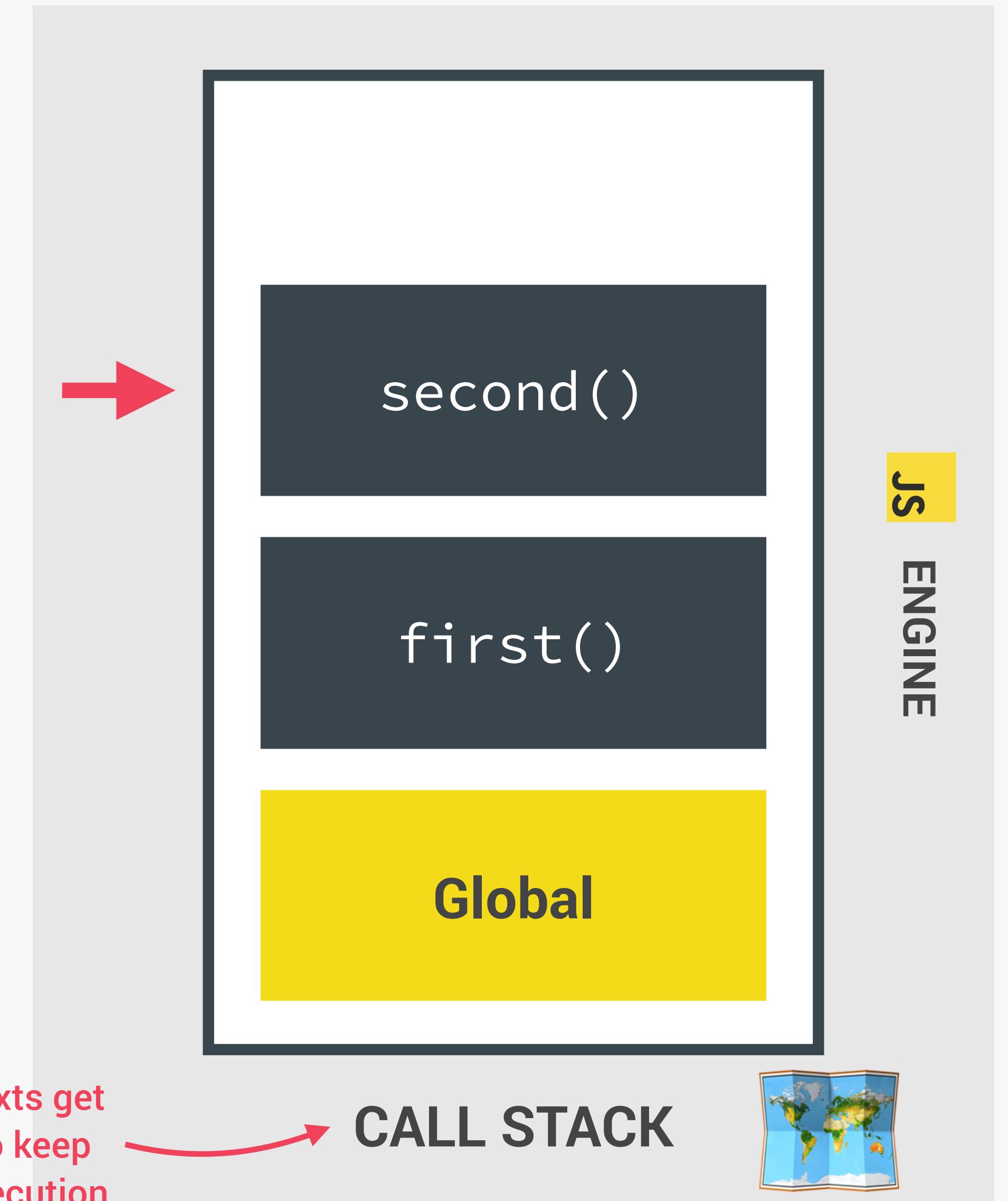
👉 Compiled code starts execution

```
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! not **var**
 - 👉 Functions are **also block scoped** (only in strict mode)
var is only func scope

THE SCOPE CHAIN

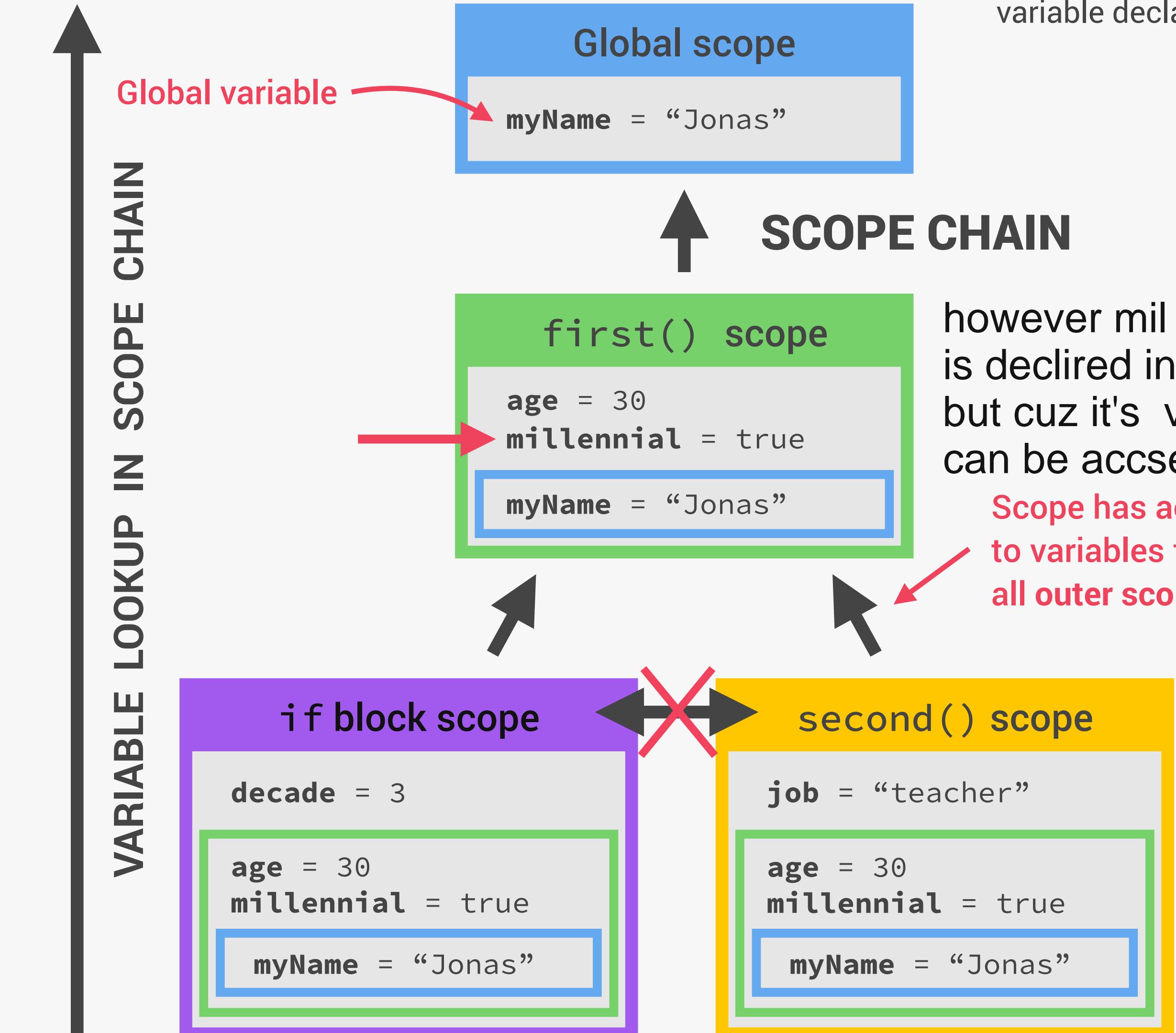
```
const myName = 'Jonas';

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();

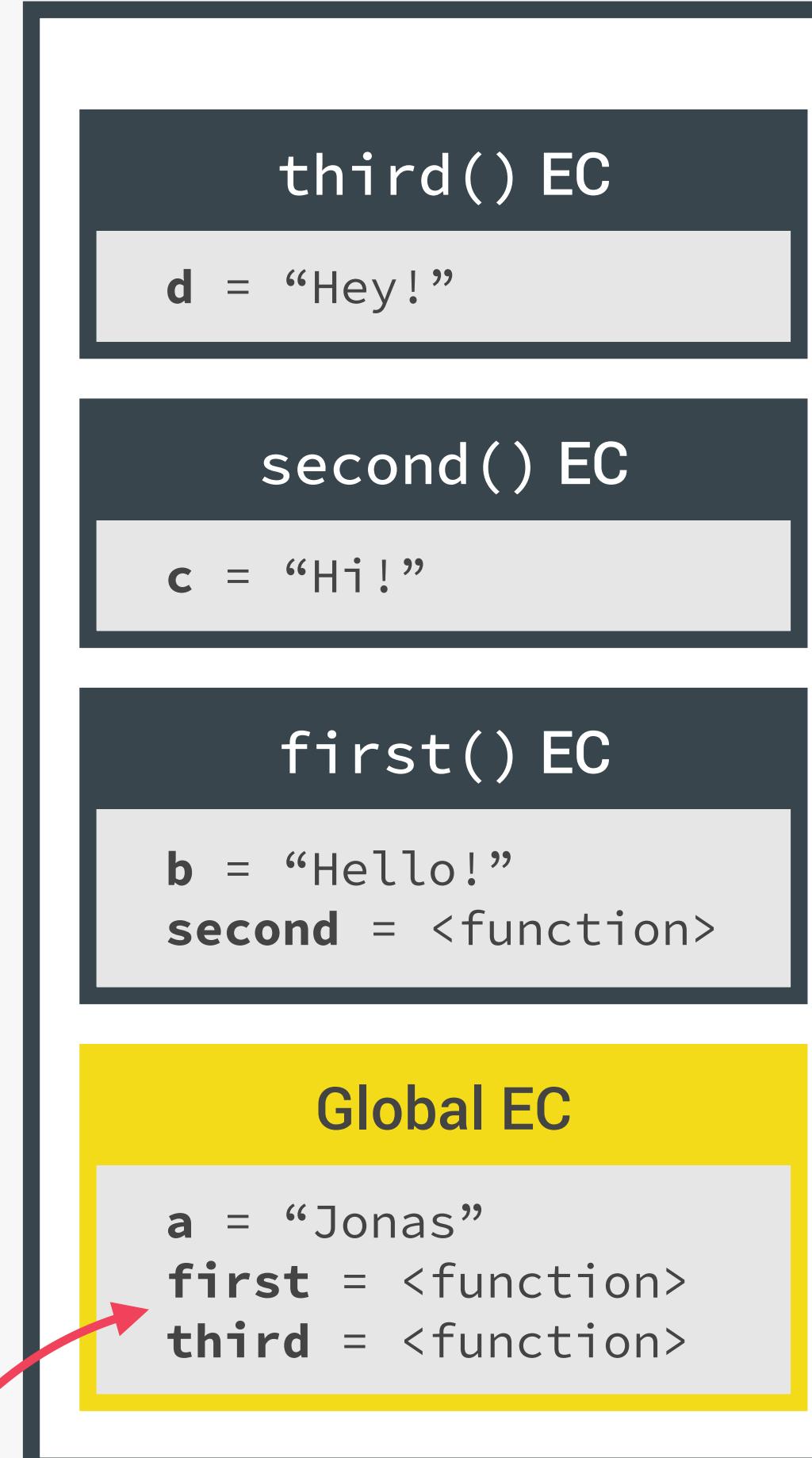
function first() {
  const b = 'Hello!';
  second();

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

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

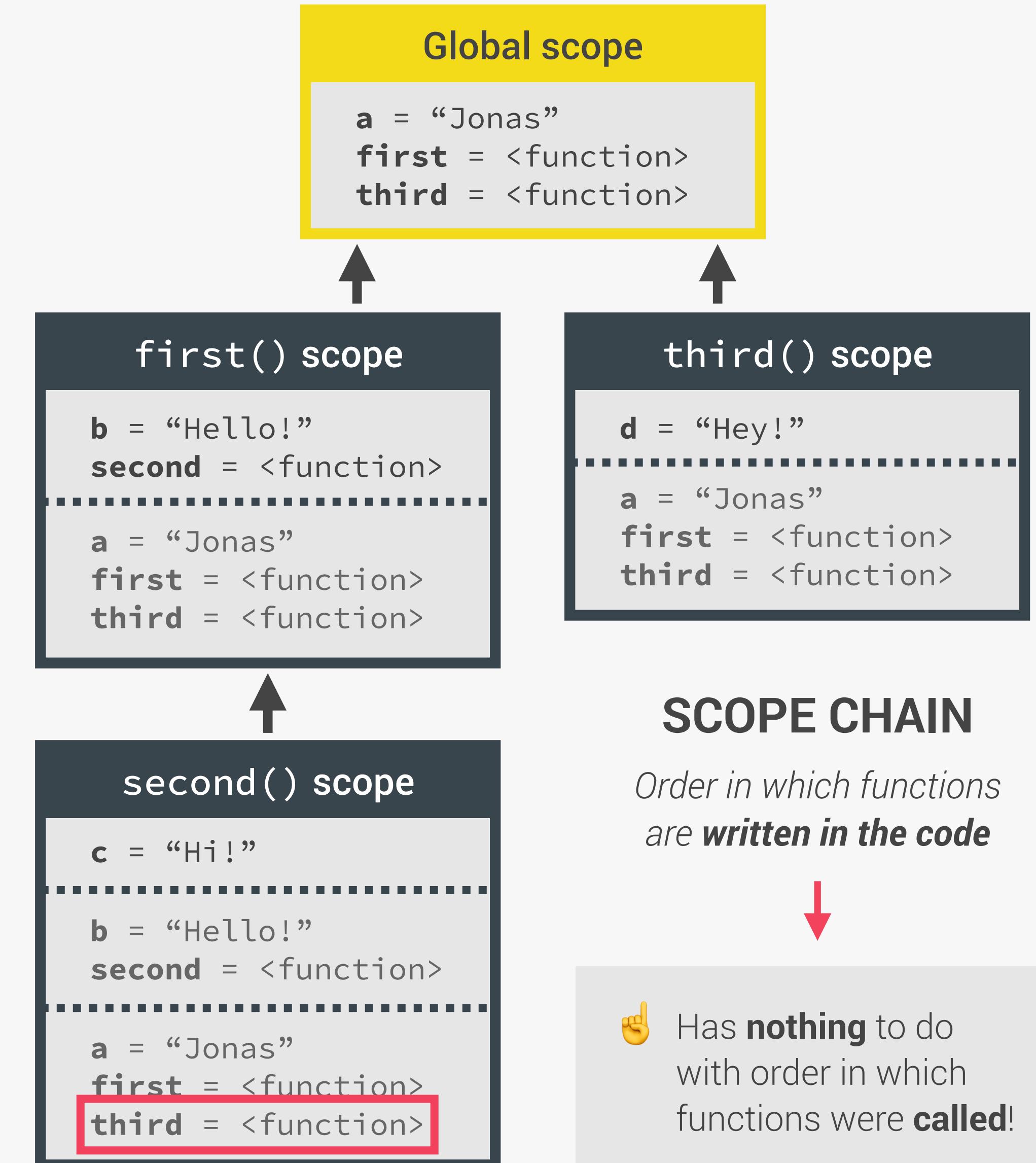
c and b can NOT be found
in third() scope!

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.

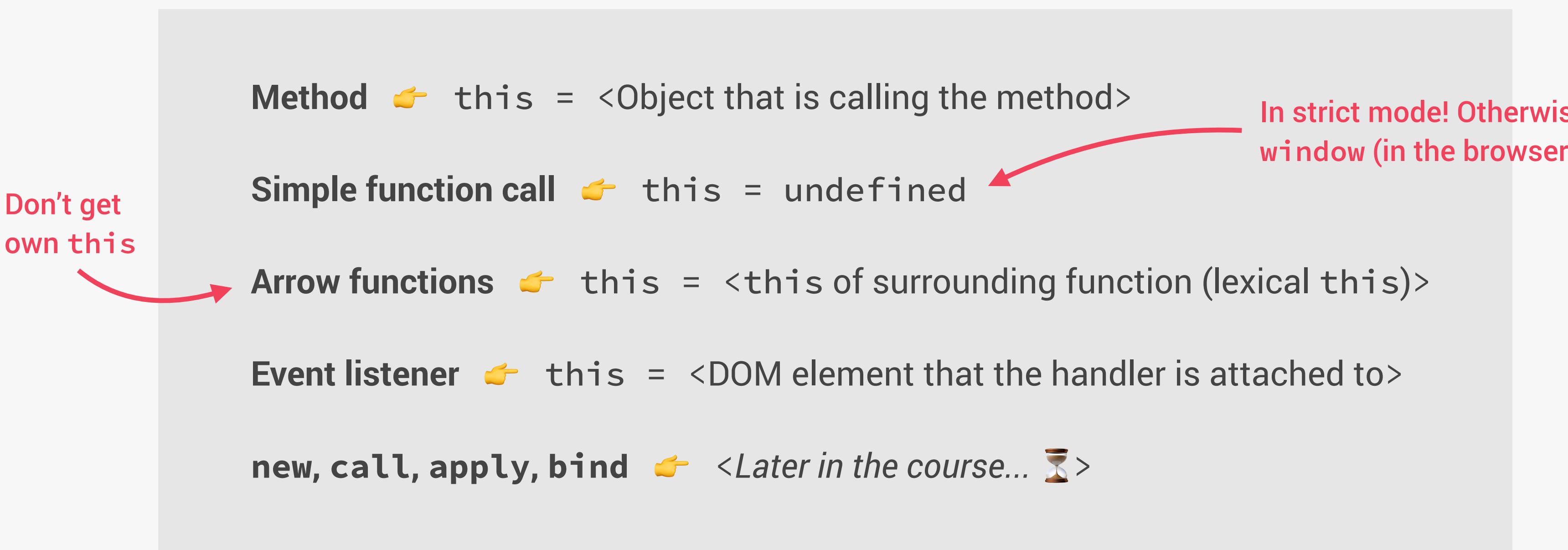
👉 **this** is **NOT** static. It depends on **how** the function is called, and its value is only assigned when the function **is actually called**.

EXECUTION CONTEXT

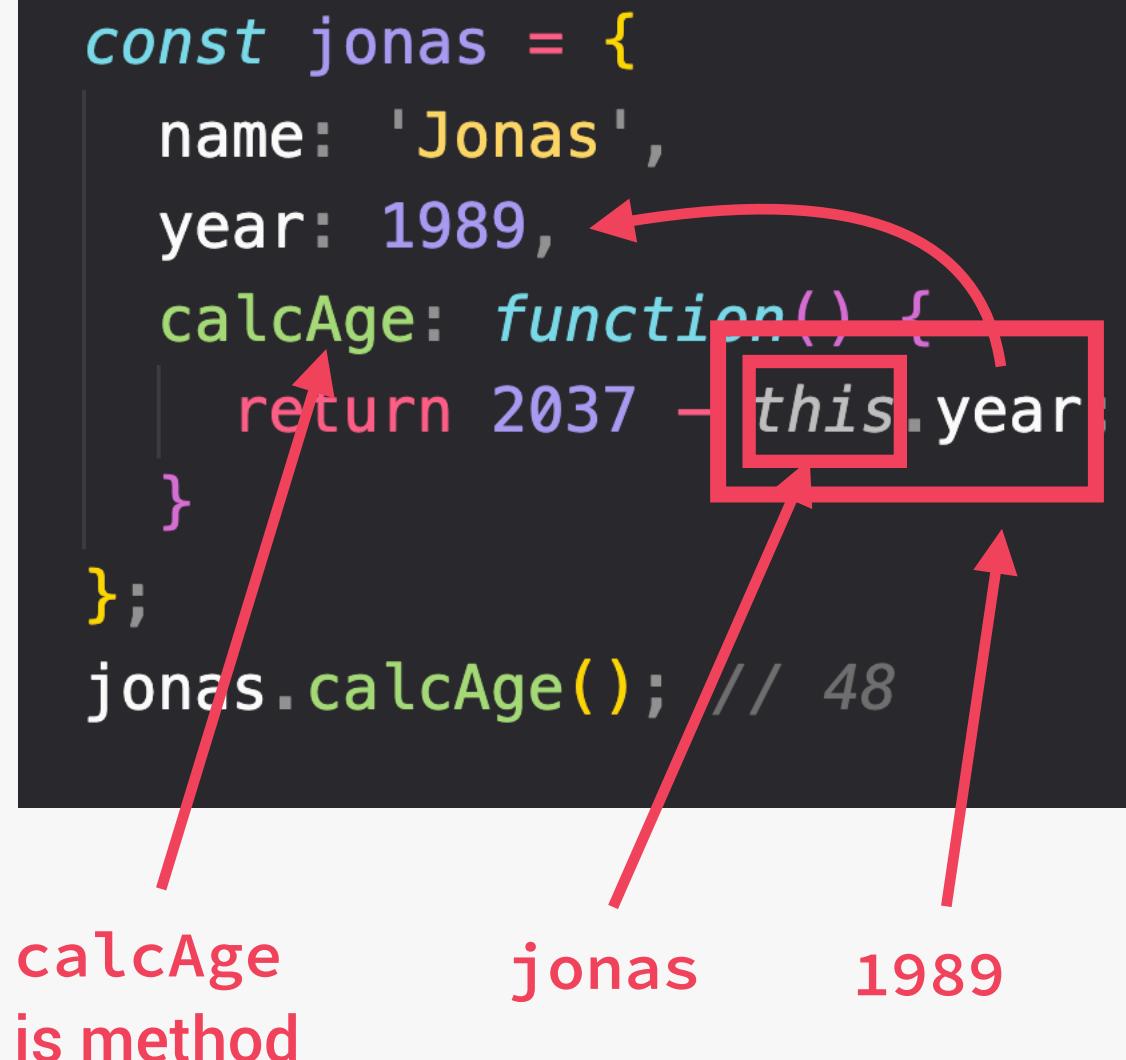
✓ Variable environment

✓ 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