THE COMPLETE JAVASCRIPT COURSE

FROM ZERO TO EXPERT!

SECTION

HOW JAVASCRIPT WORKS BEHIND THE SCENES

LECTURE

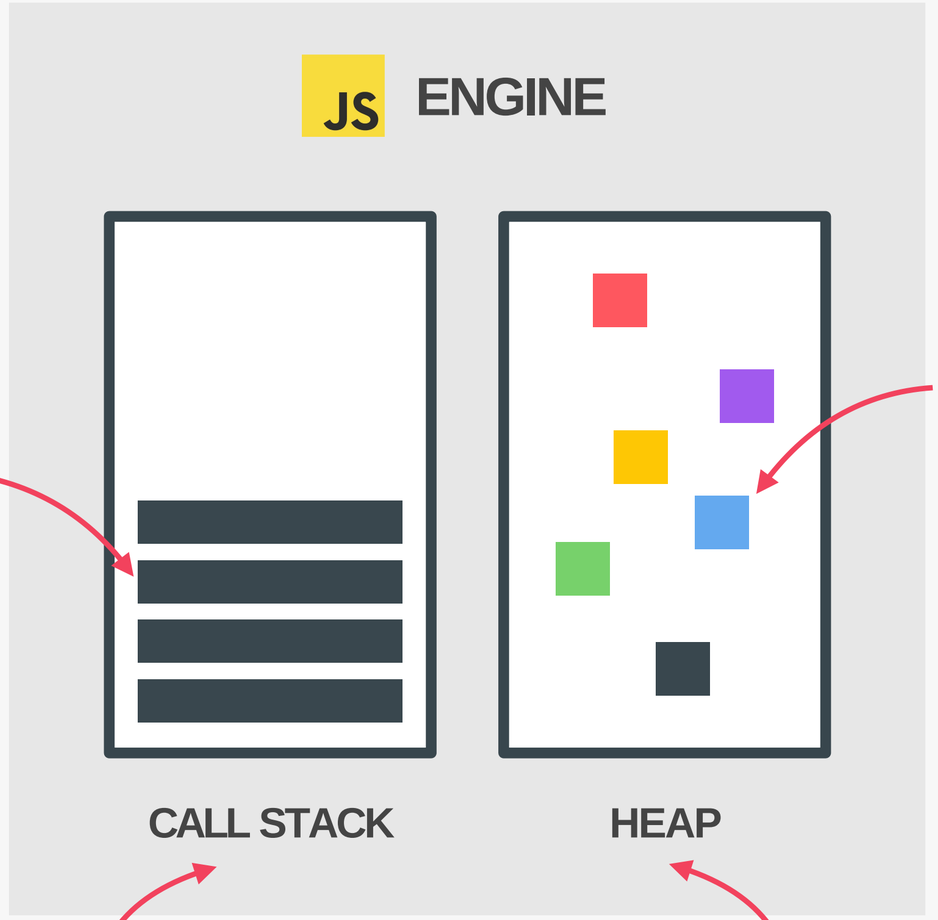
THE JAVASCRIPT ENGINE AND RUNTIME

WHAT IS A JAVASCRIPT ENGINE?

JS ENGINE

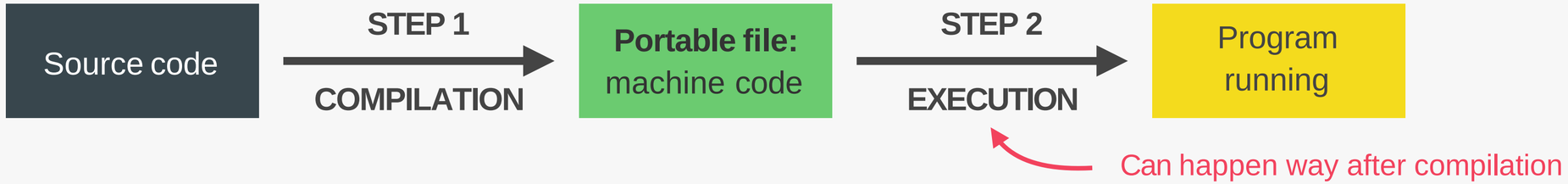
PROGRAM THAT EXECUTES JAVASCRIPT CODE.

👉 **Example: V8 Engine**

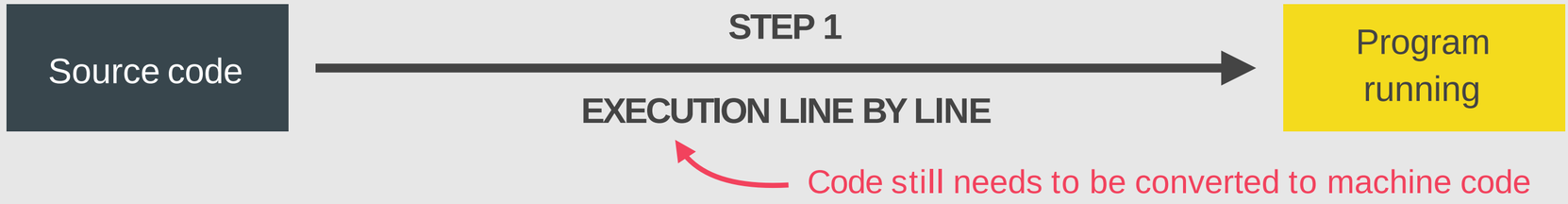


COMPUTER SCIENCE SIDENOTE: COMPILATION VS. INTERPRETATION

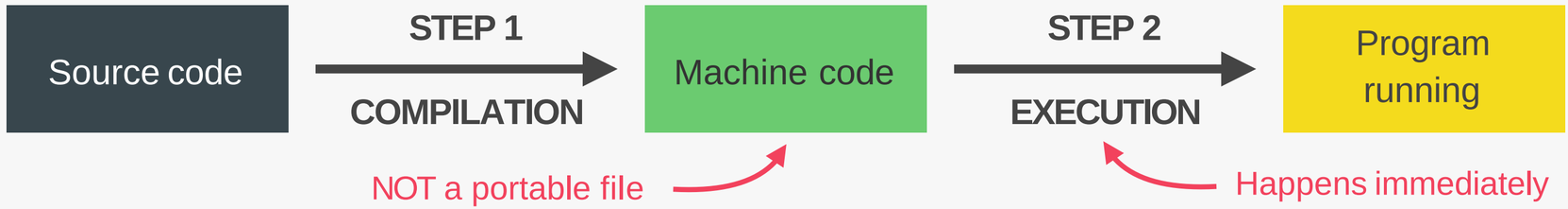
👉 Compilation: Entire code is converted into machine code at once, and written to a binary file that can be executed by a computer.



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



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



MODERN JUST-IN-TIME COMPILATION OF JAVASCRIPT

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

~~"JavaScript is an interpreted language"~~

JS ENGINE

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

Parsing

AST

Example

Compilation

Just-in-time compilation

Optimization

Execution

Happens in Call Stack

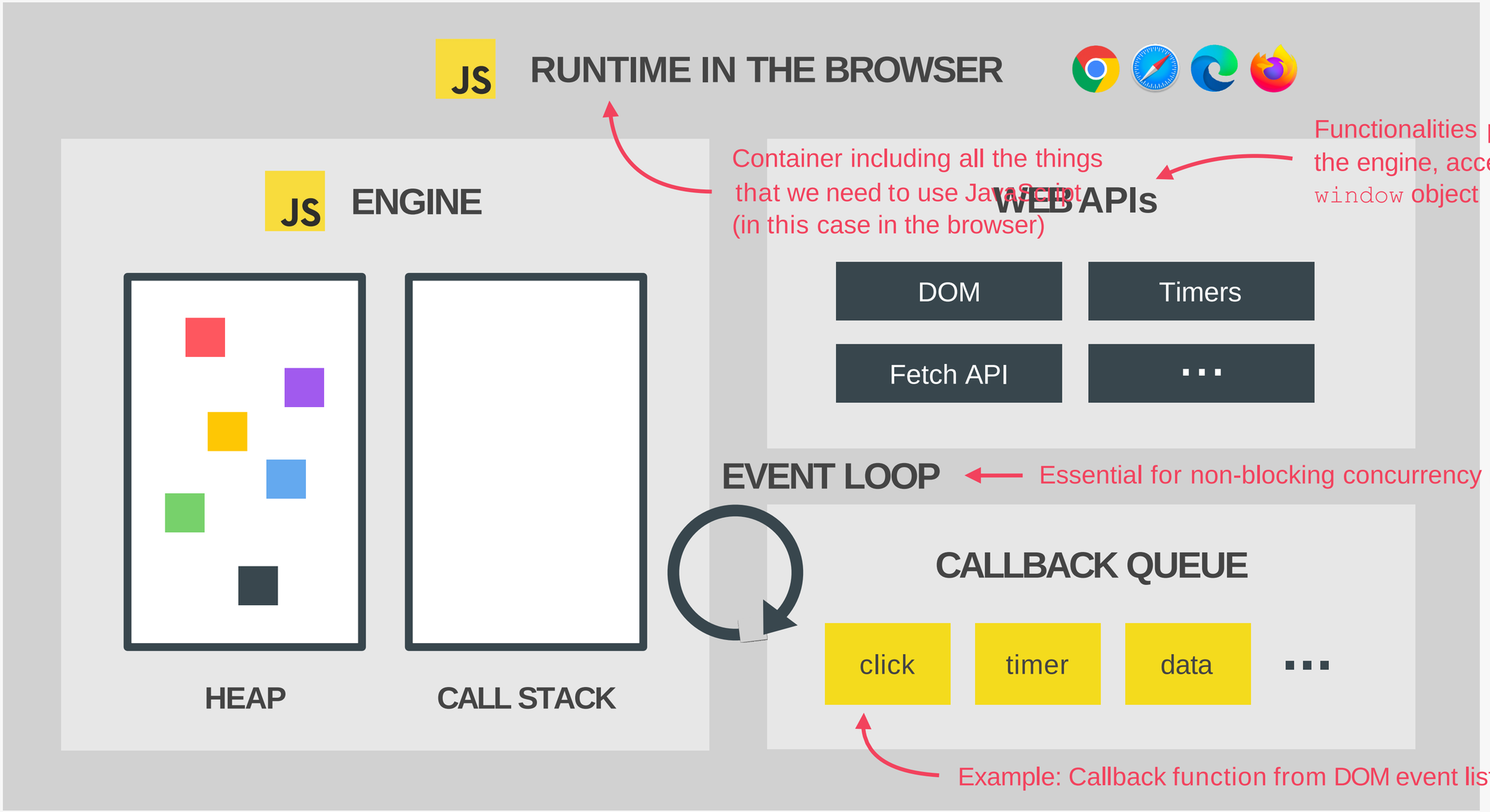
During execution!

👉 AST Example

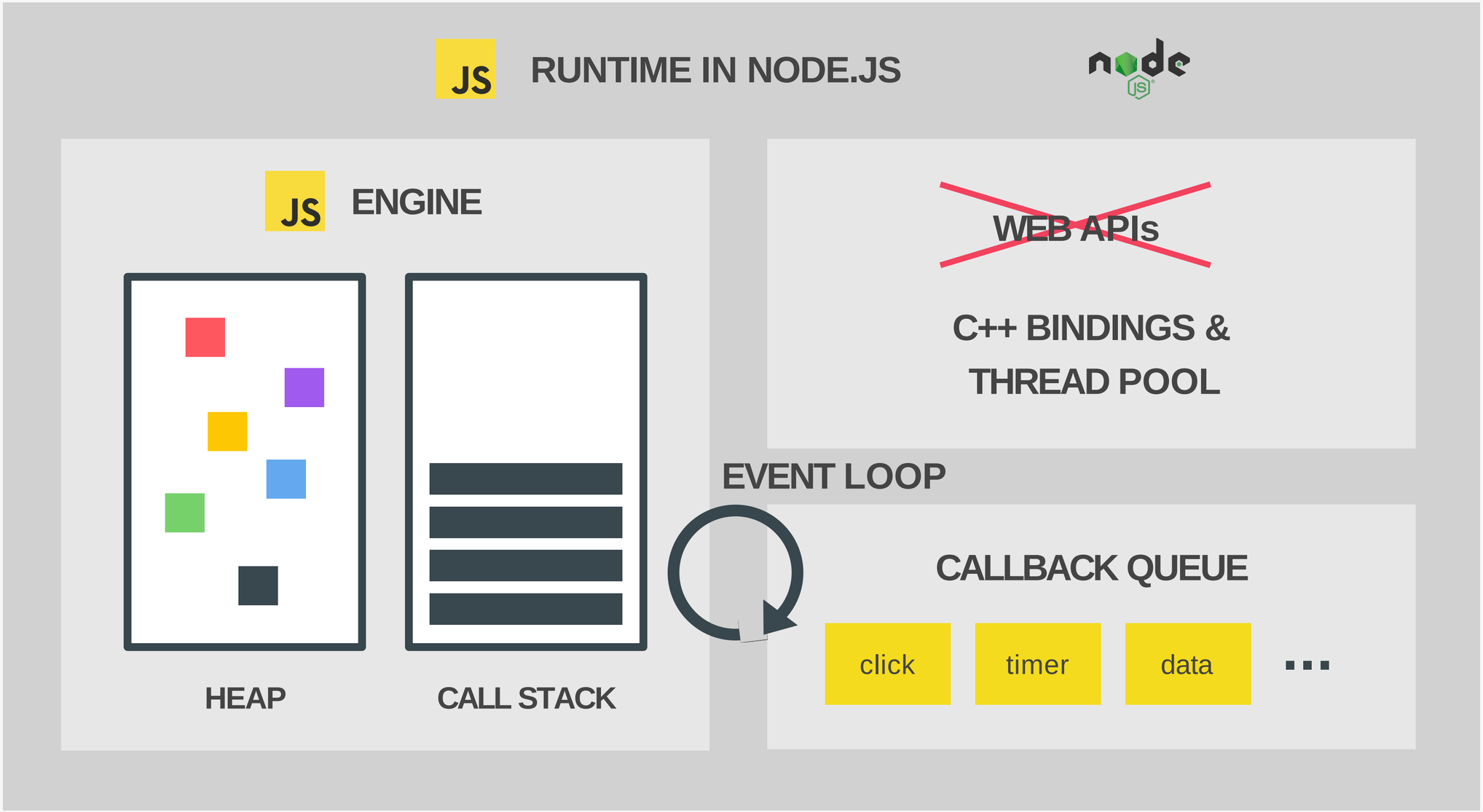
```
const x = 23;
```

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

THE BIGGER PICTURE: JAVASCRIPT RUNTIME



THE BIGGER PICTURE: JAVASCRIPT RUNTIME



THE COMPLETE JAVASCRIPT COURSE

FROM ZERO TO EXPERT!

SECTION

HOW JAVASCRIPT WORKS BEHIND THE SCENES

LECTURE

EXECUTION CONTEXTS AND THE CALL STACK

WHAT IS AN EXECUTION CONTEXT?

👉 Human-readable code:

```
const name = 'Jonas';

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

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

Function body only executed when called!

Compilation

EXECUTION

Creation of **global execution context** (for top-level code)

NOT inside a function

Execution of **top-level code** (inside global EC)

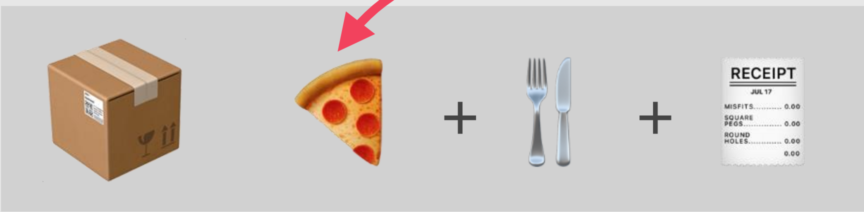
Execution of **functions** and waiting for **callbacks**

Example: click event callback

EXECUTION CONTEXT

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

“JavaScript code"



Pizza “execution context”

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

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

All together make the call stack

EXECUTION CONTEXT IN DETAIL

WHAT'S INSIDE EXECUTION CONTEXT?

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

NOT in arrow functions!

Generated during "creation phase", right before execution

```
const name = 'Jonas';

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

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

const x = first();
```

Global

```
name = 'Jonas'
first = <function>
second = <function>
x = <unknown>
```

Literally the function code

Need to run **`first()`** first

first()

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

Need to run **`second()`** first

second()

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

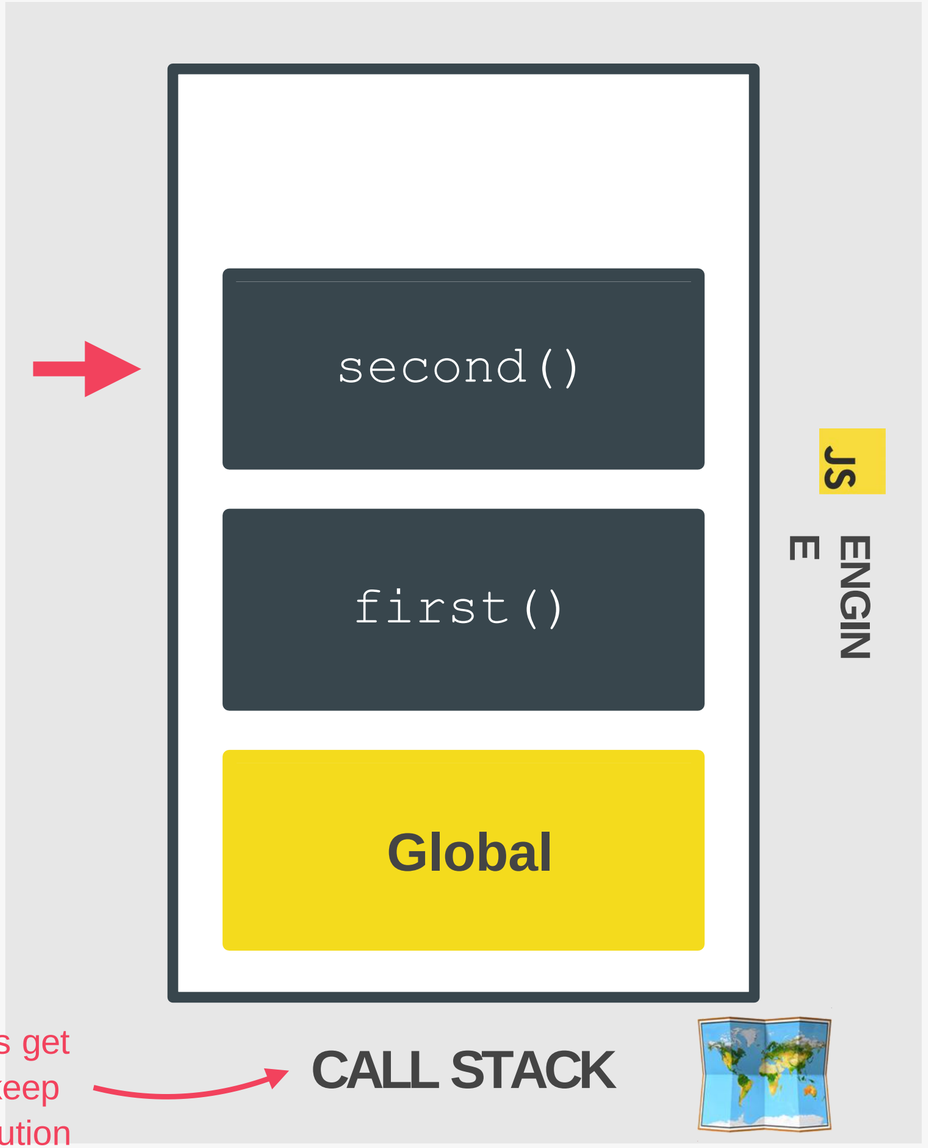
Array of passed arguments. Available in all "regular" functions (not arrow)

(Technically, values only become known during execution)

THE CALL STACK

👉 Compiled code starts execution

```
const name = 'Jonas';

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

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

const x = first();
```



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

THE COMPLETE JAVASCRIPT COURSE

FROM ZERO TO EXPERT!

SECTION

HOW JAVASCRIPT WORKS BEHIND THE SCENES

LECTURE

SCOPE AND THE SCOPE CHAIN

SCOPING AND SCOPE IN JAVASCRIPT: CONCEPTS

EXECUTION CONTEXT

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

SCOPE CONCEPTS

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

THE 3 TYPES OF SCOPE

GLOBAL SCOPE

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

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

FUNCTION SCOPE

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

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

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

BLOCK SCOPE (ES6)

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

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

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

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

THE SCOPE CHAIN

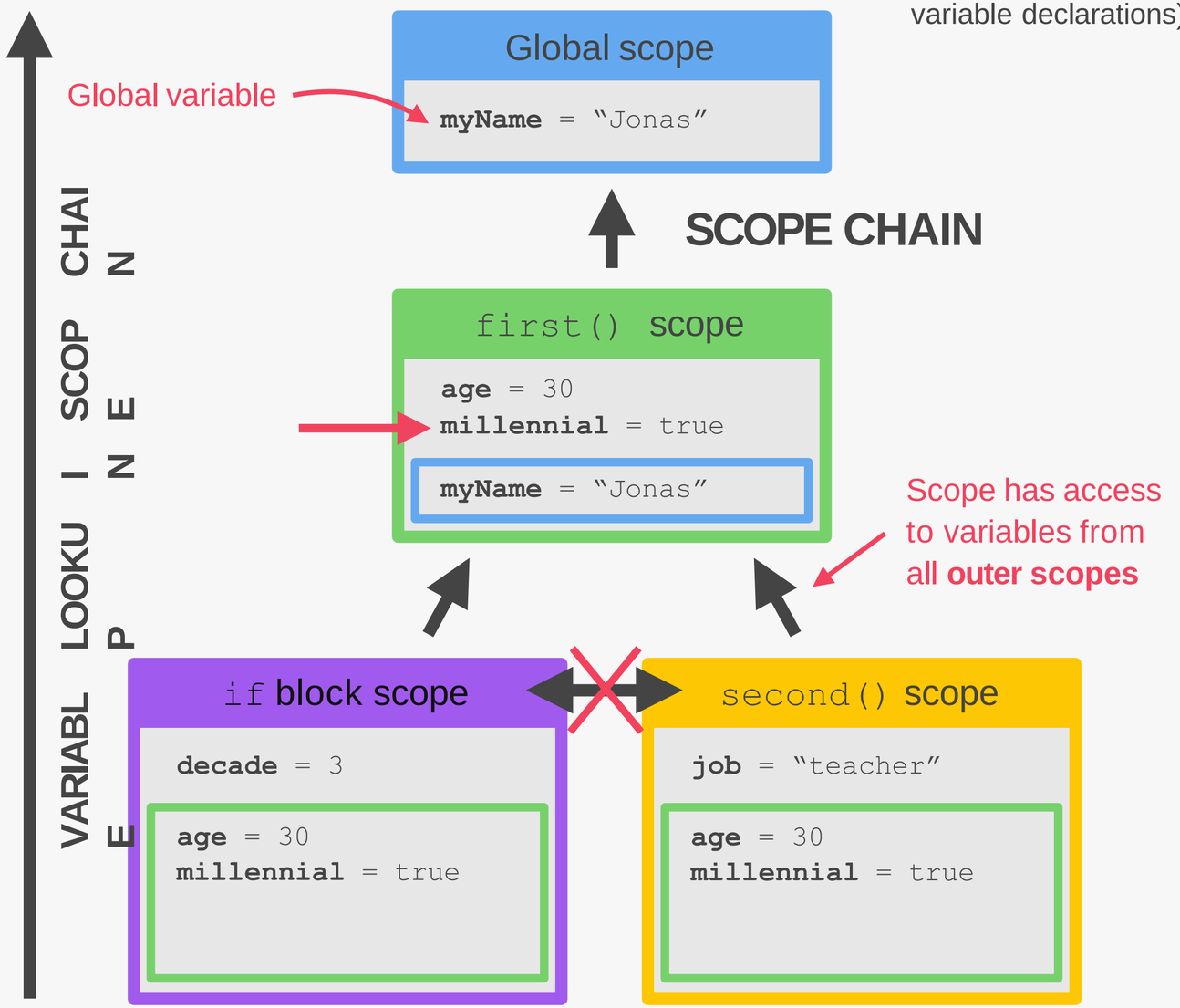
```
const myName = 'Jonas';

function first() {
  const age = 30;

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

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

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

  second();
}

first();
```

let and const are **block-scoped**

var is **function-scoped**

Variables not in current scope

(Considering only variable declarations)

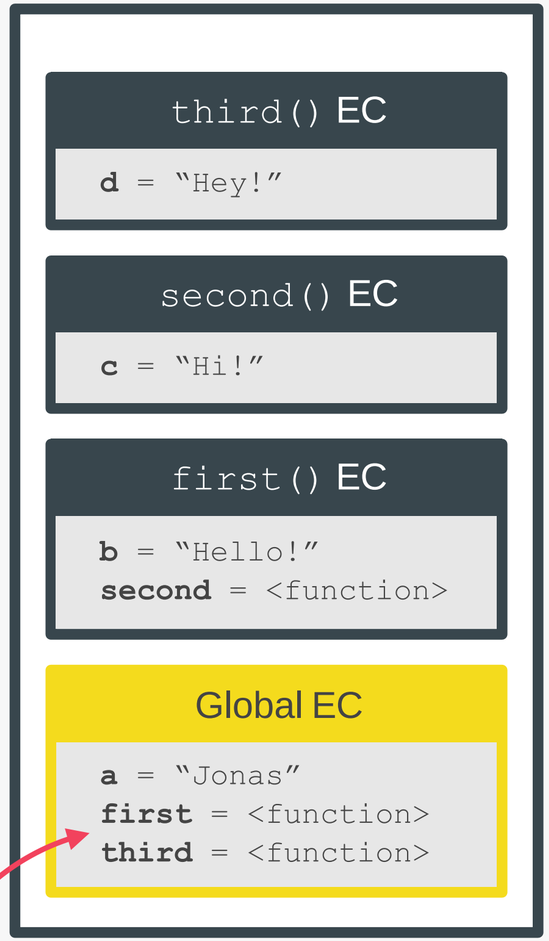


SCOPE CHAIN VS. CALL STACK

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

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

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

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

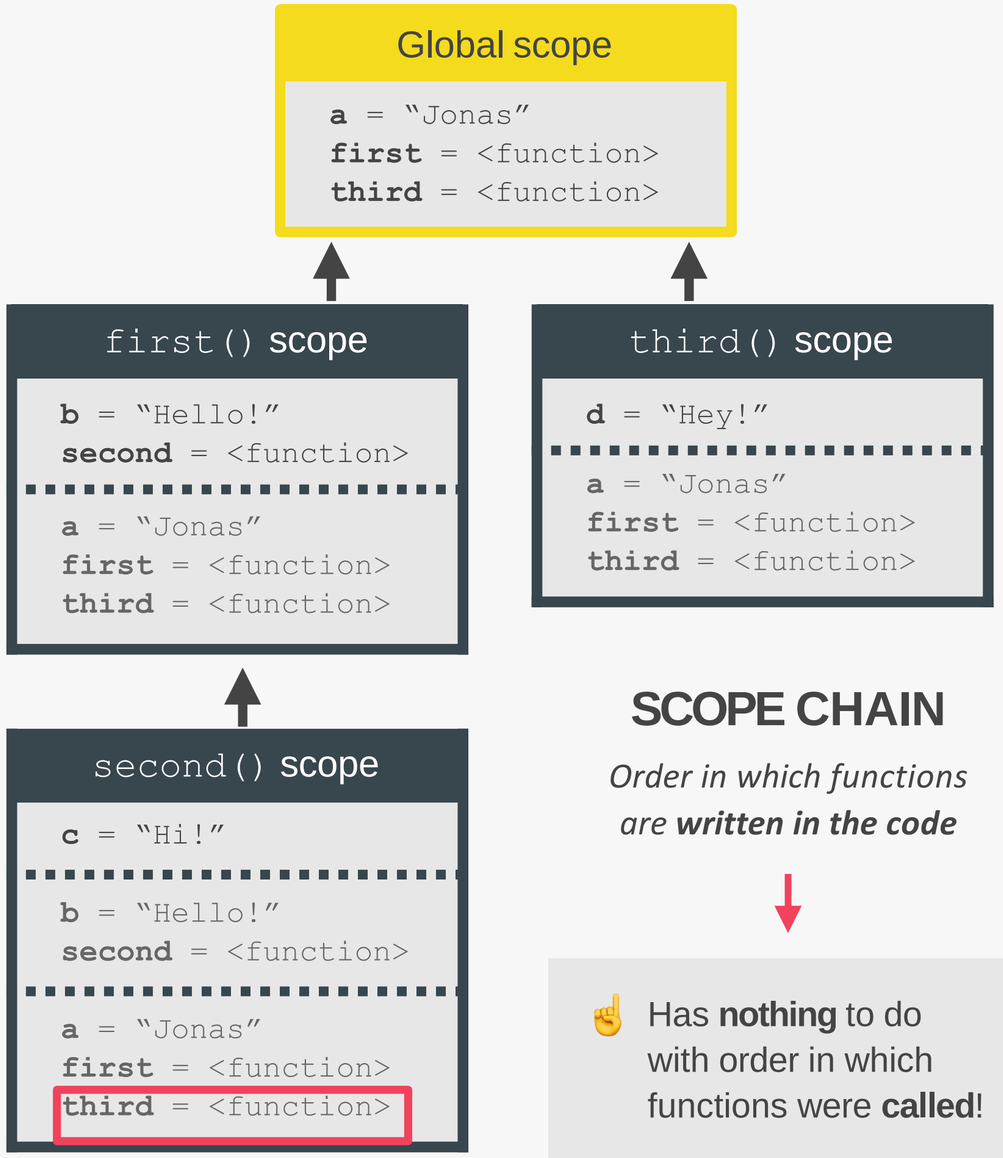
`c` and `b` can NOT be found in `third()` scope!

Variable environment (VE)



CALL STACK

*Order in which functions were **called***



SCOPE CHAIN

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

☝ Has **nothing** to do with order in which functions were **called**!

SUMMARY 🥳

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