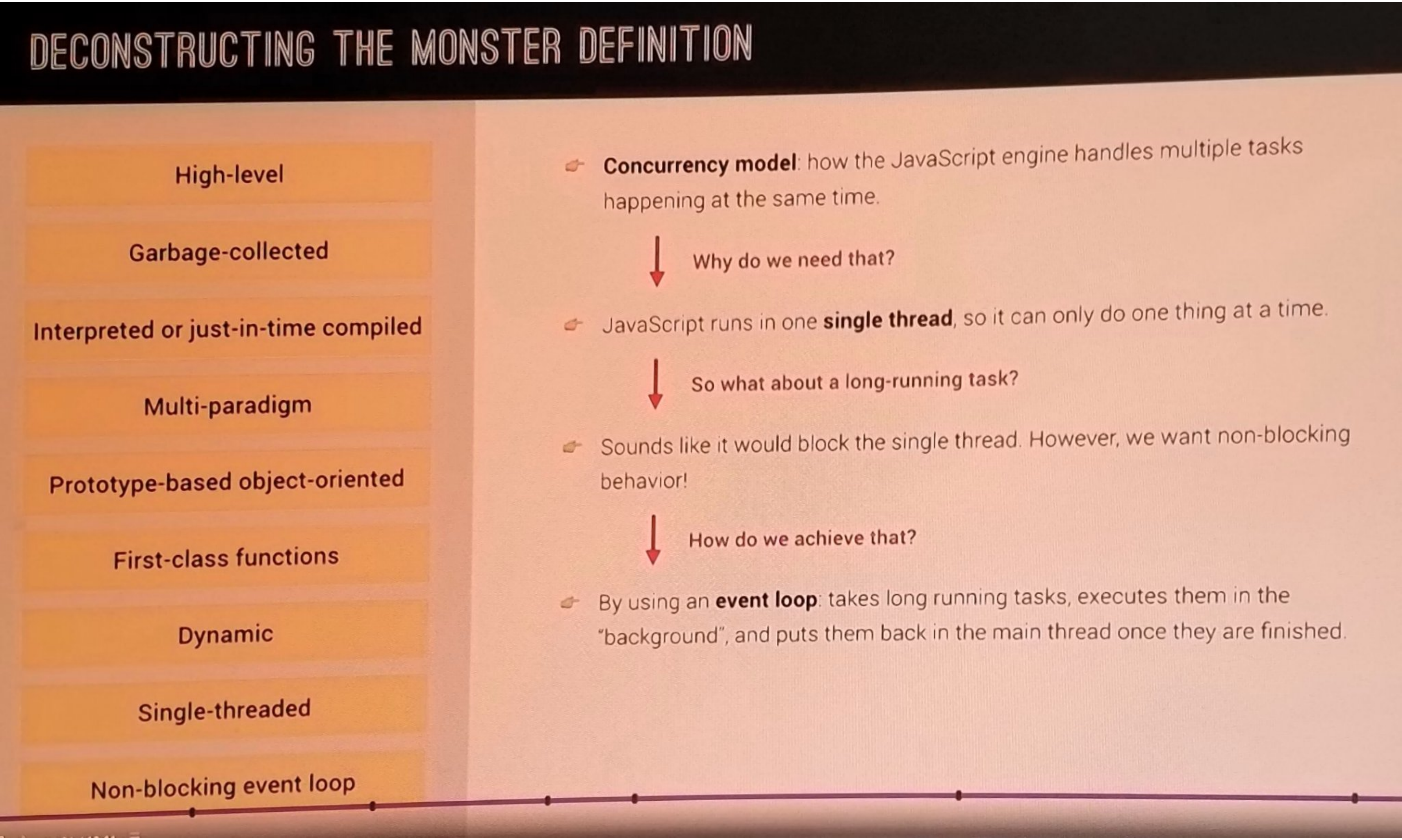


DECONSTRUCTING THE MONSTER DEFINITION

- 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?**

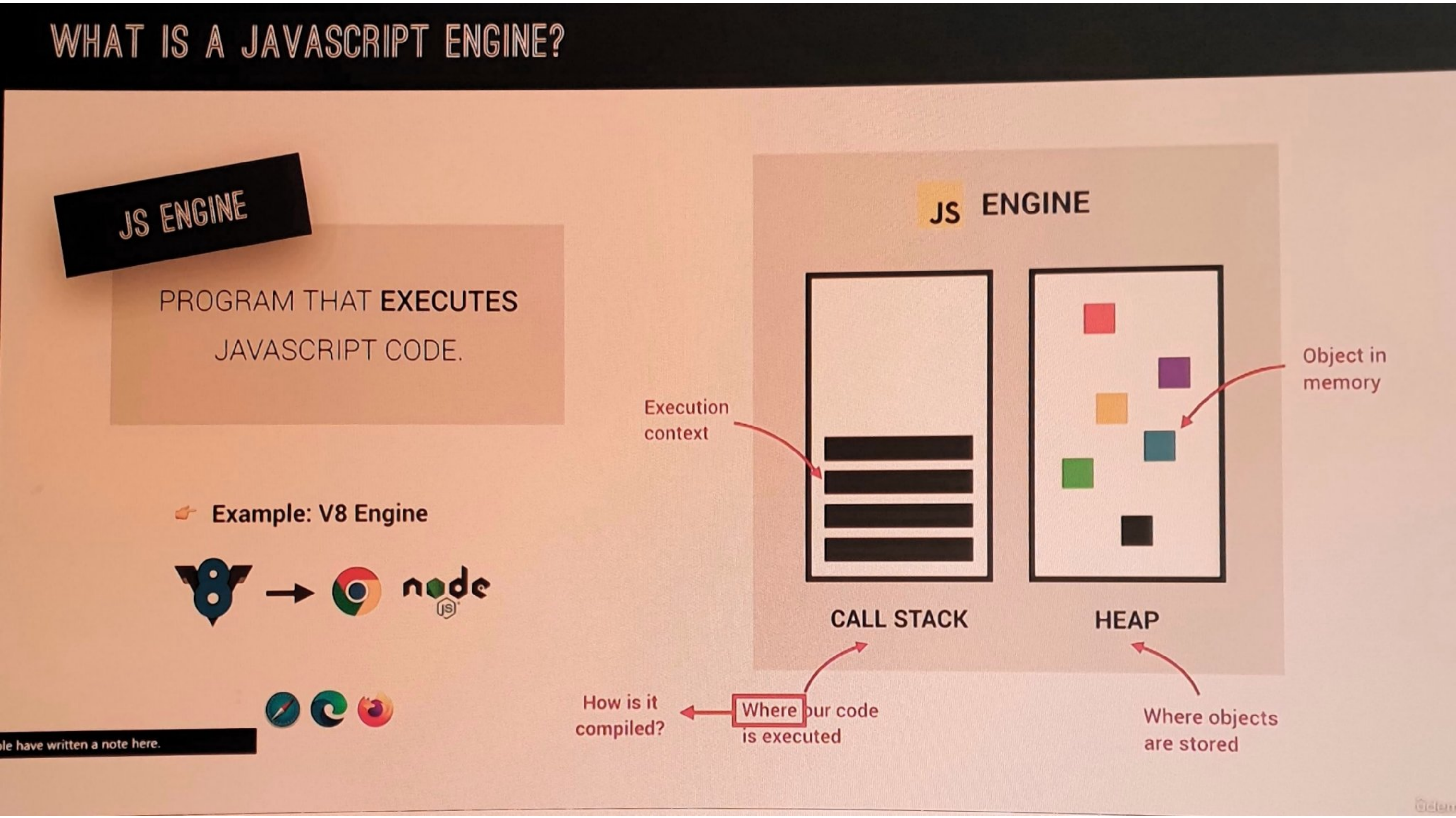
👉 JavaScript runs in one **single thread**, so it can only do one thing at a time.

↓ **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?**

👉 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.



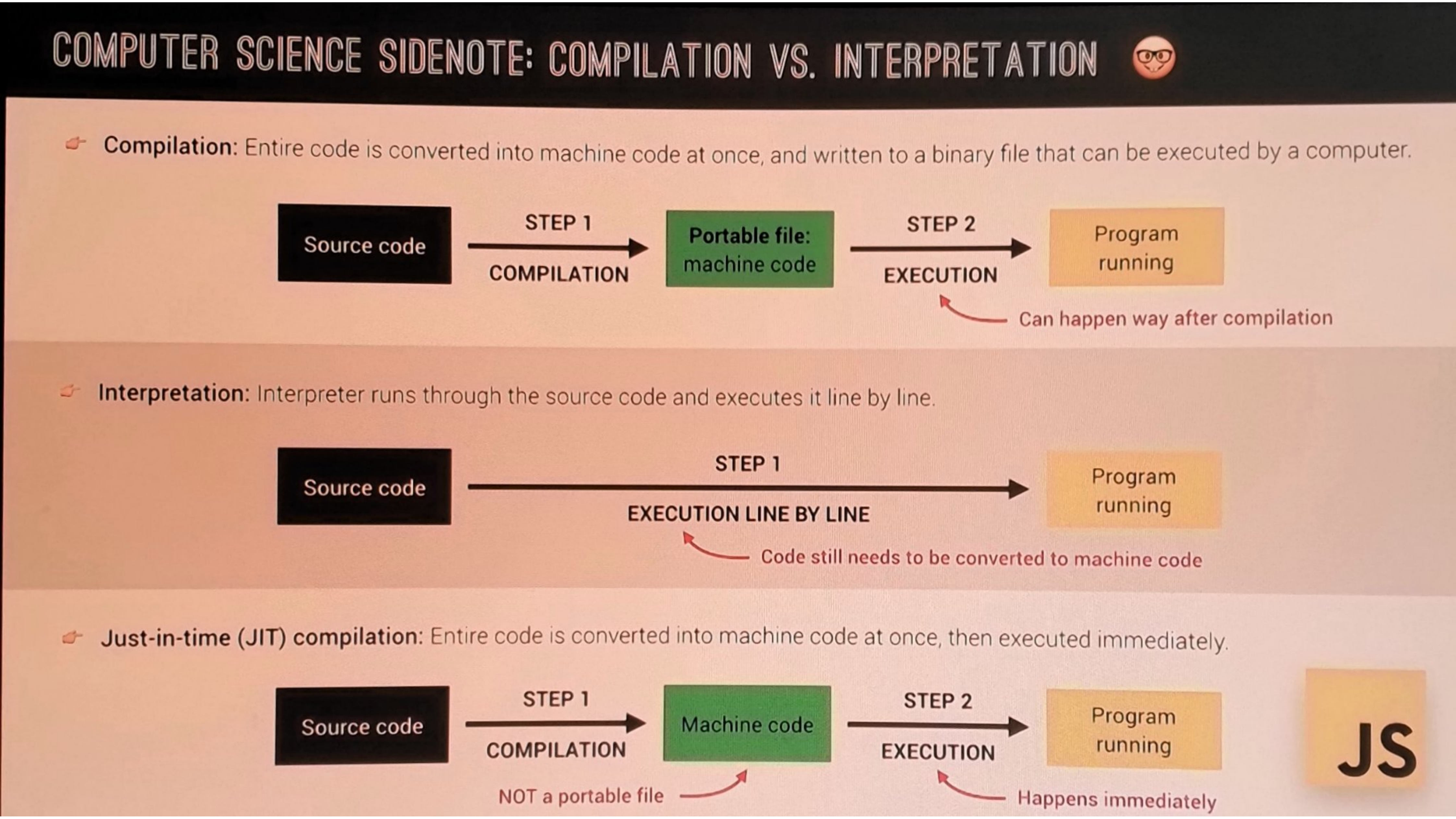
WHAT IS A JAVASCRIPT ENGINE?

JS ENGINE

PROGRAM THAT **EXECUTES** JAVASCRIPT CODE.

Example: V8 Engine

ple have written a note here.



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.

$$\begin{tikzcd}\boxed{\text{Source code}} \arrow[r, "\text{STEP 1}", "\text{COMPILATION}"'] & \boxed{\text{Portable file: machine code}} \arrow[r, "\text{STEP 2}", "\text{EXECUTION}"'] & \boxed{\text{Program running}}\end{tikzcd}$$

Can happen way after compilation

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

$$\begin{tikzcd}\boxed{\text{Source code}} \arrow[r, "\text{STEP 1}", "\text{EXECUTION LINE BY LINE}"'] & \boxed{\text{Program running}}\end{tikzcd}$$

Code still needs to be converted to machine code

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

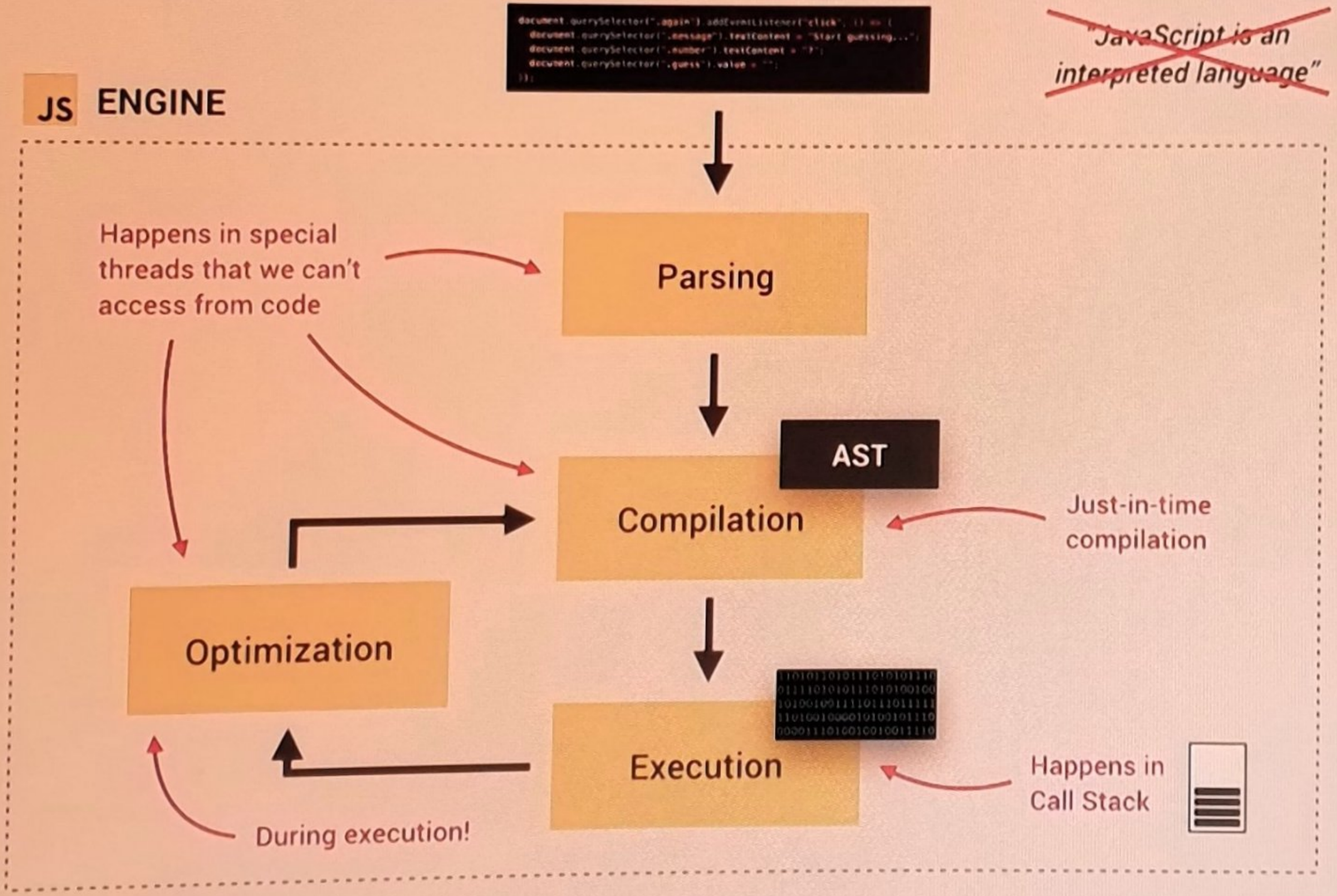
$$\begin{tikzcd}\boxed{\text{Source code}} \arrow[r, "\text{STEP 1}", "\text{COMPILATION}"'] & \boxed{\text{Machine code}} \arrow[r, "\text{STEP 2}", "\text{EXECUTION}"'] & \boxed{\text{Program running}}\end{tikzcd}$$

NOT a portable file

Happens immediately

JS

MODERN JUST-IN-TIME COMPILATION OF JAVASCRIPT



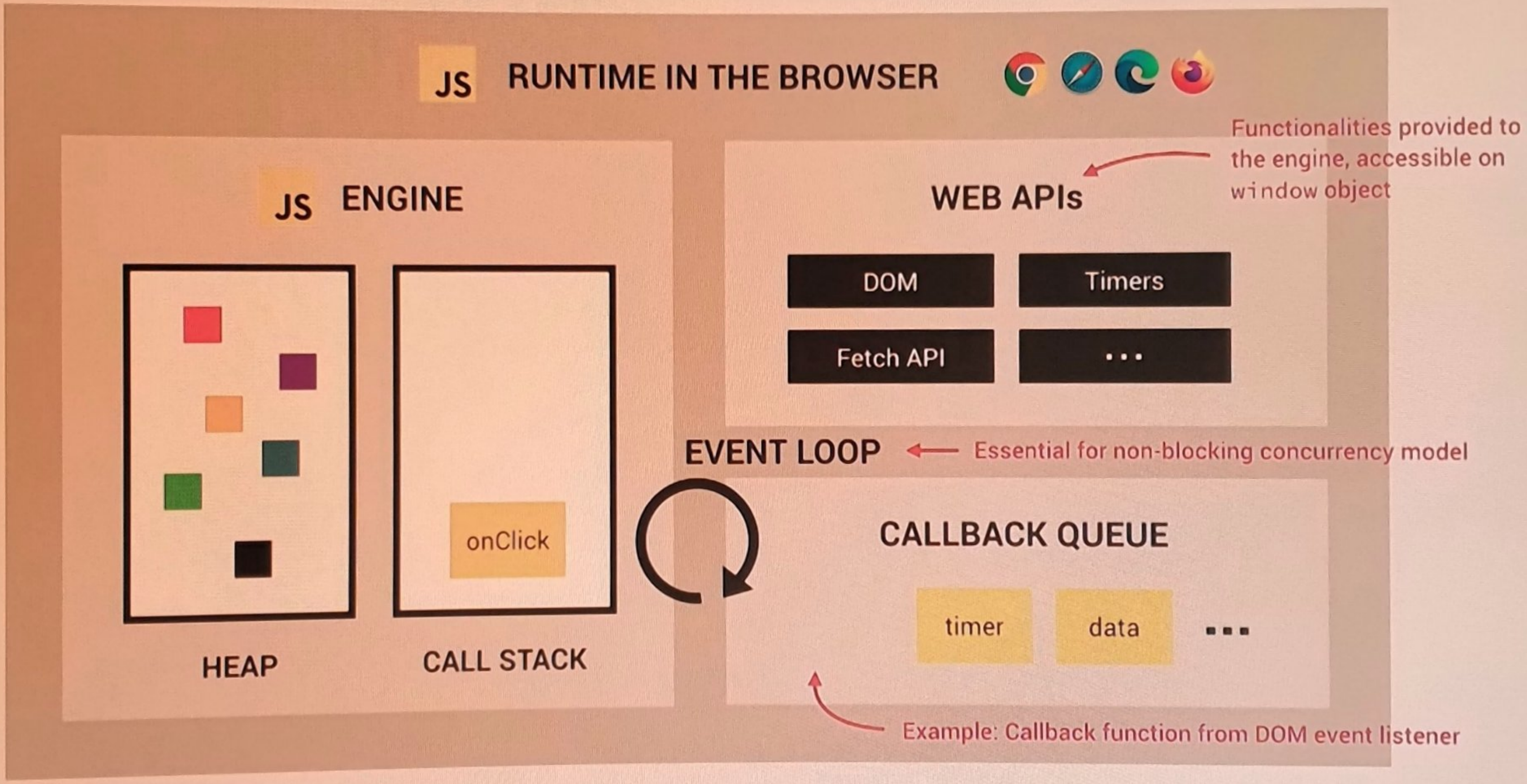
```
document.querySelector(".again").addEventListener("click", () => {
  document.querySelector(".message").textContent = "Start guessing...";
  document.querySelector(".number").textContent = "?";
  document.querySelector(".guess").value = "";
});
```

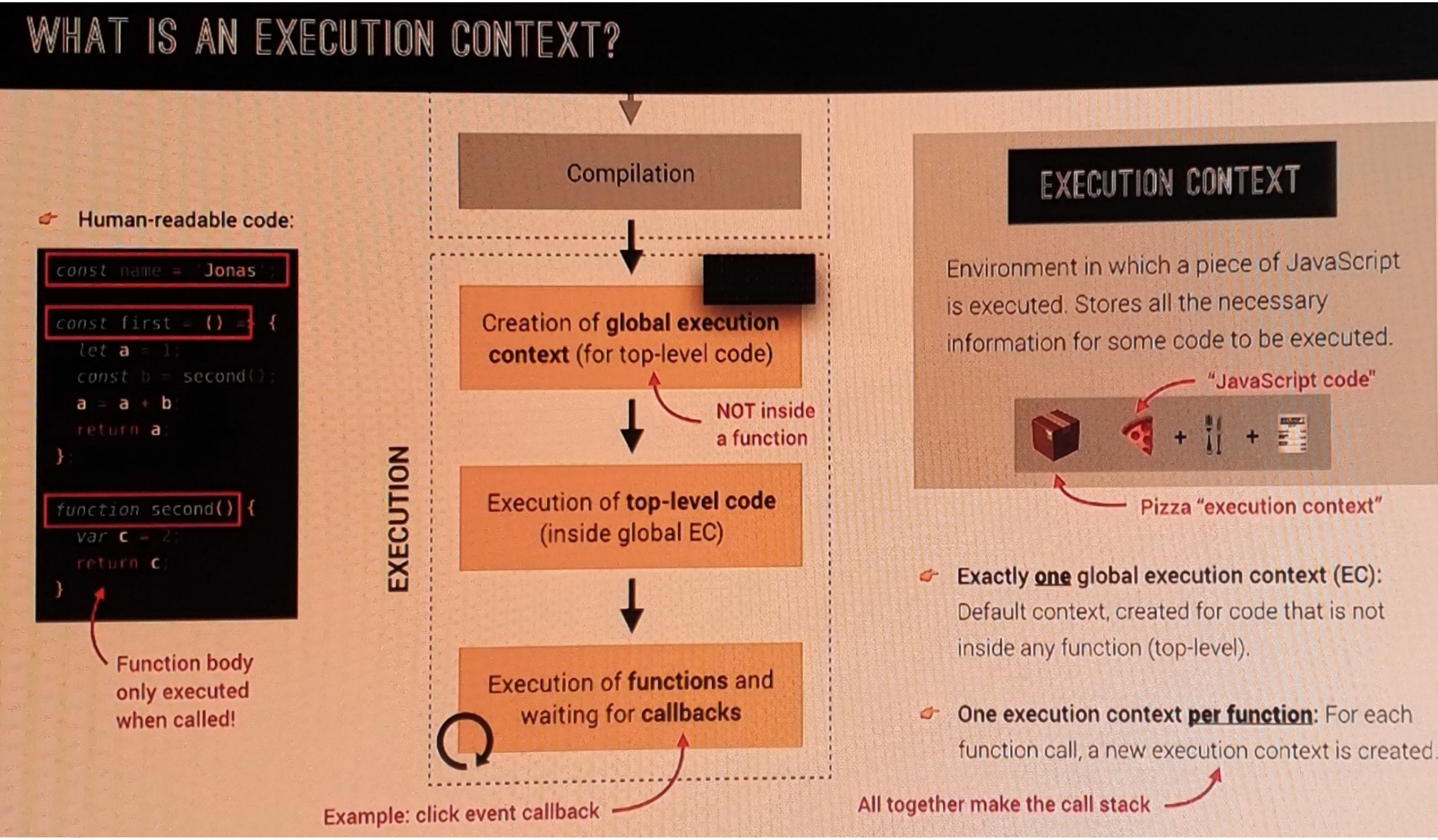
👉 AST Example

```
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


JS RUNTIME IN THE BROWSER
JS ENGINE
HEAP
onClick
CALL STACK
WEB APIs
Functionalities provided to the engine, accessible on window object
DOM
Timers
Fetch API
...
EVENT LOOP
Essential for non-blocking concurrency model
CALLBACK QUEUE
timer
data
...
Example: Callback function from DOM event listener



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

[redacted]

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

NOT inside a function

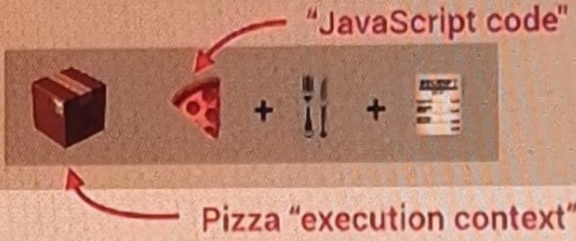
Execution of **top-level code** (inside global EC)

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

Example: click event callback

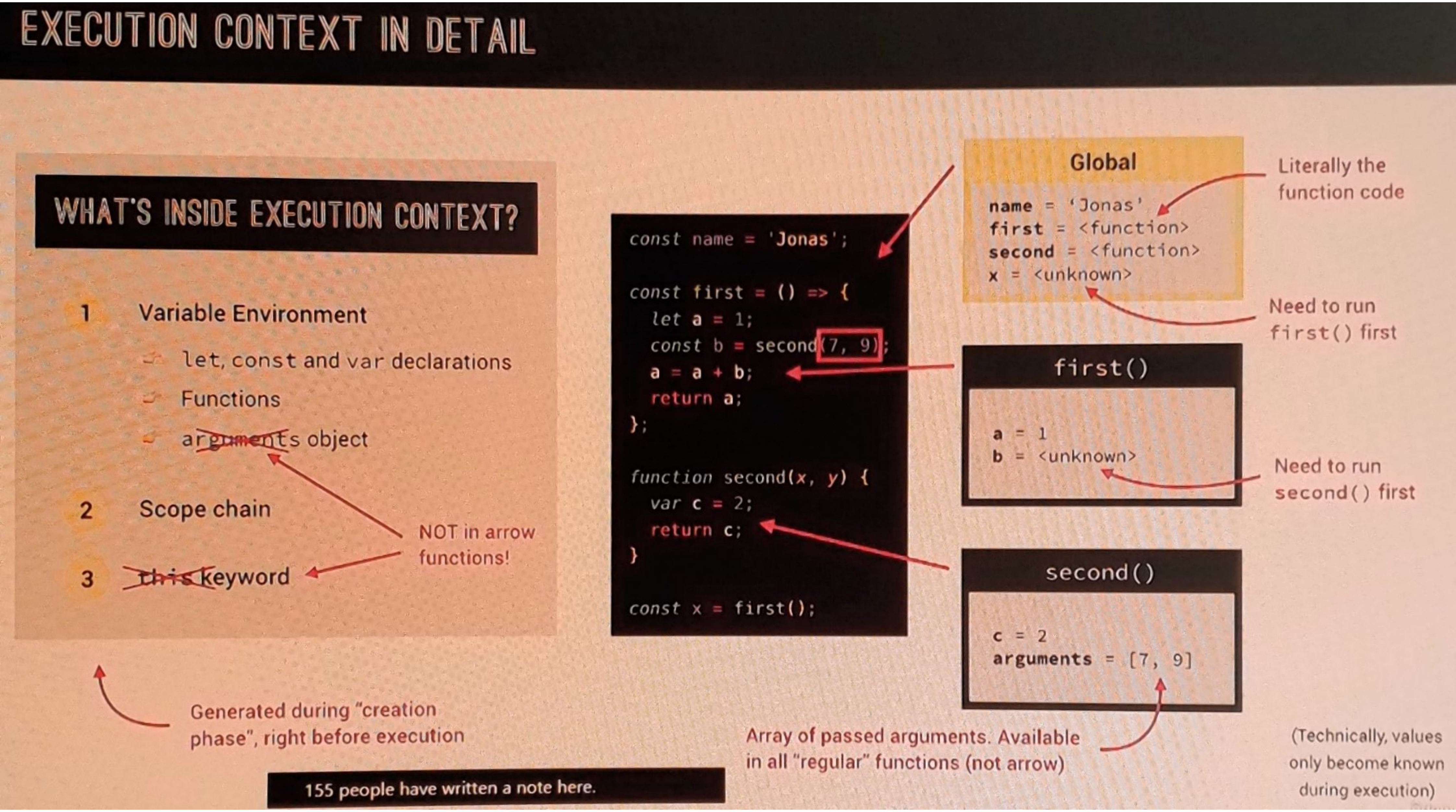
EXECUTION CONTEXT

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



- **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
   - ar~~guments~~s 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

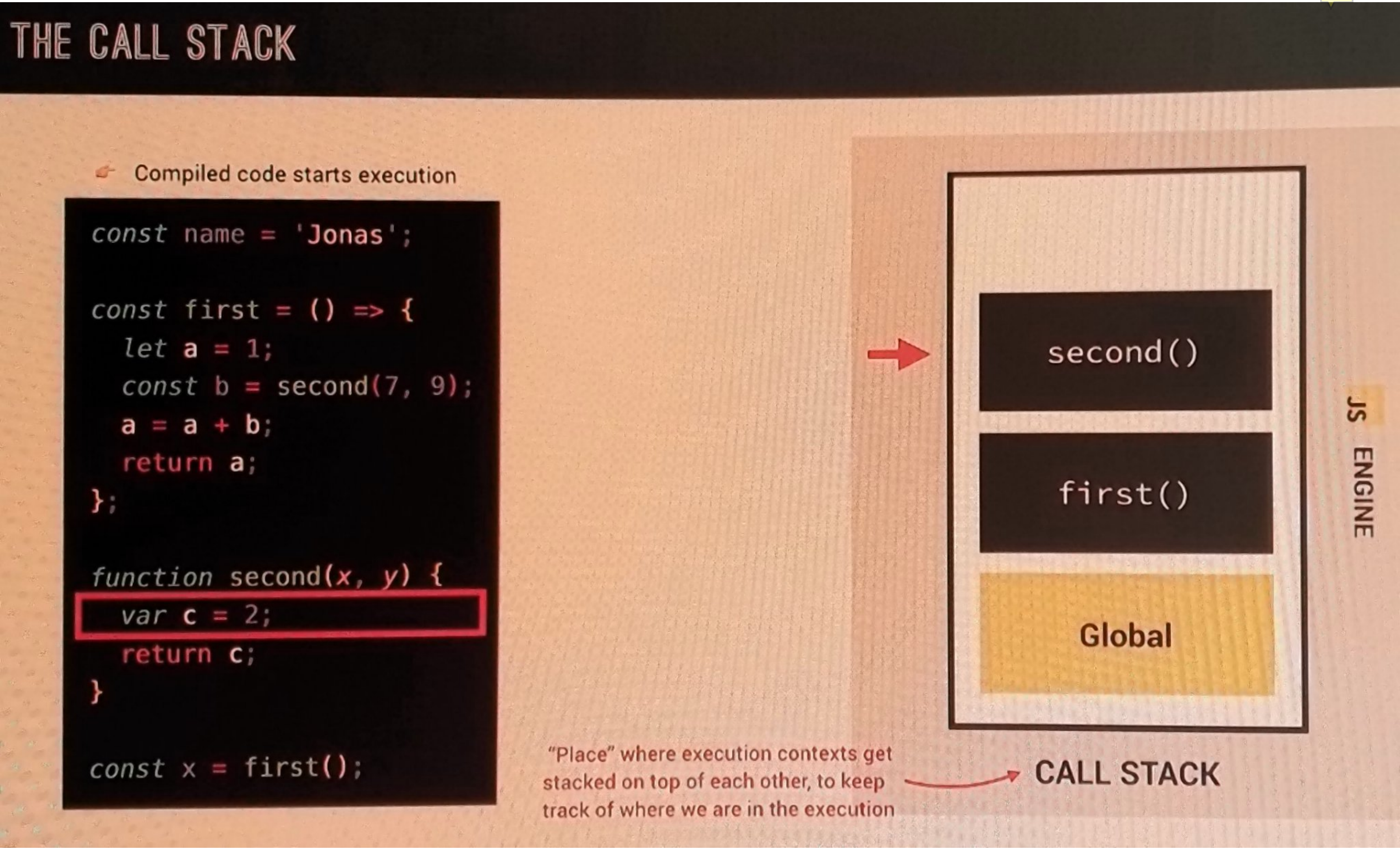
second()

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

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

(Technically, values only become known during execution)

155 people have written a note here.



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