WHAT IS JAVASCRIPT: REVISITED

JAVASCRIPT

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

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

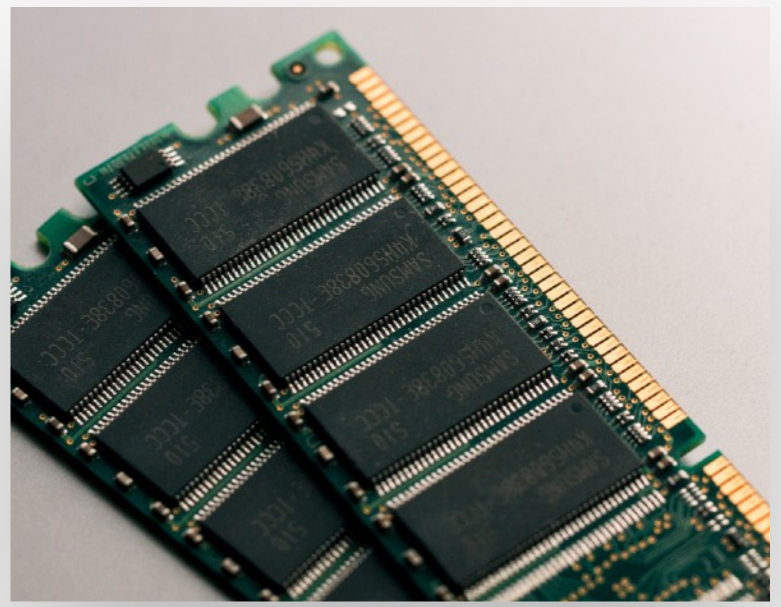
DECONSTRUCTING THE MONSTER DEFINITION

- 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

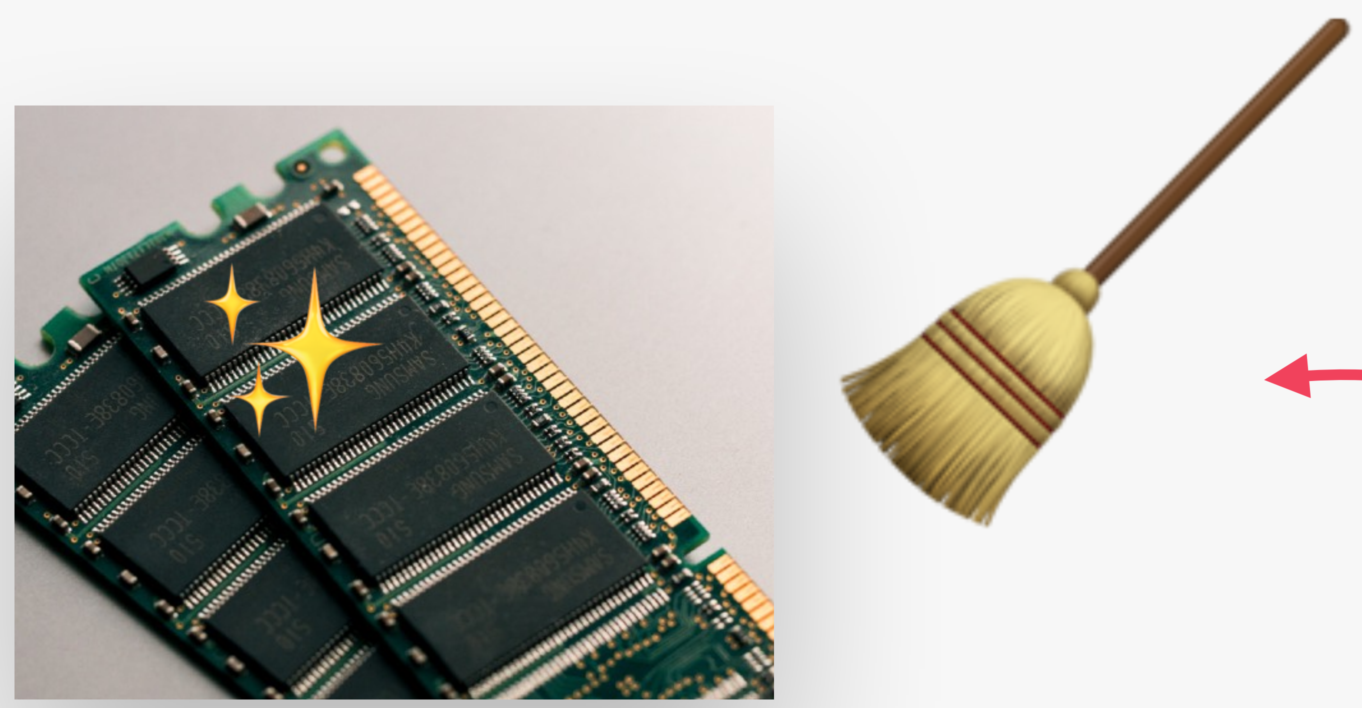
Developer has to manage resources **manually**

HIGH-LEVEL

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

DECONSTRUCTING THE MONSTER DEFINITION

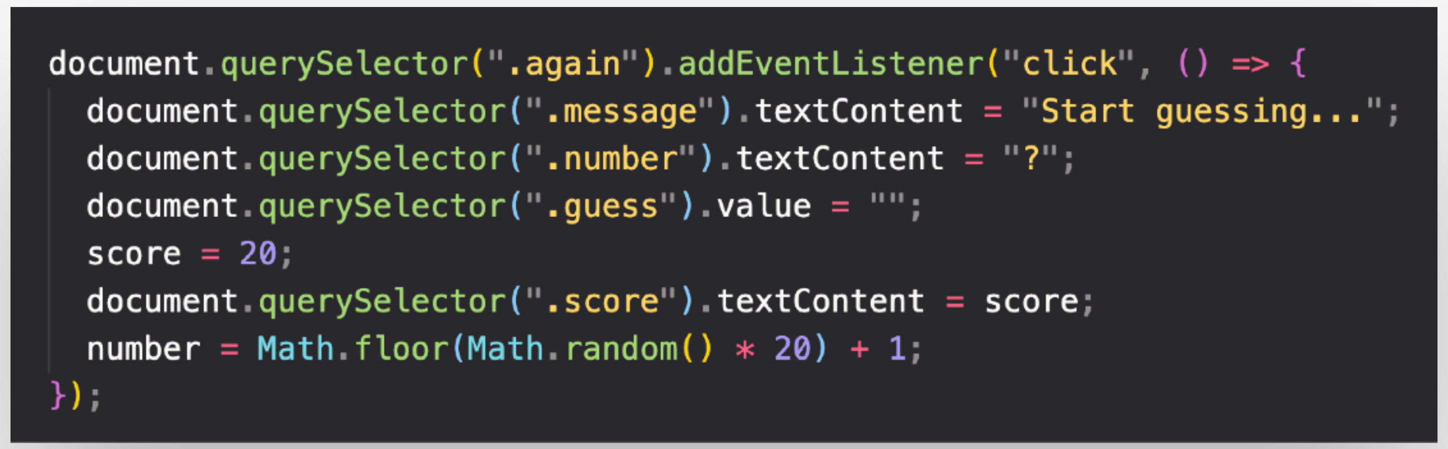
- High-level
- Garbage-collected
- Interpreted or just-in-time compiled
- Multi-paradigm
- Prototype-based object-oriented
- First-class functions
- Dynamic
- Single-threaded
- Non-blocking event loop



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 = COMPILING

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

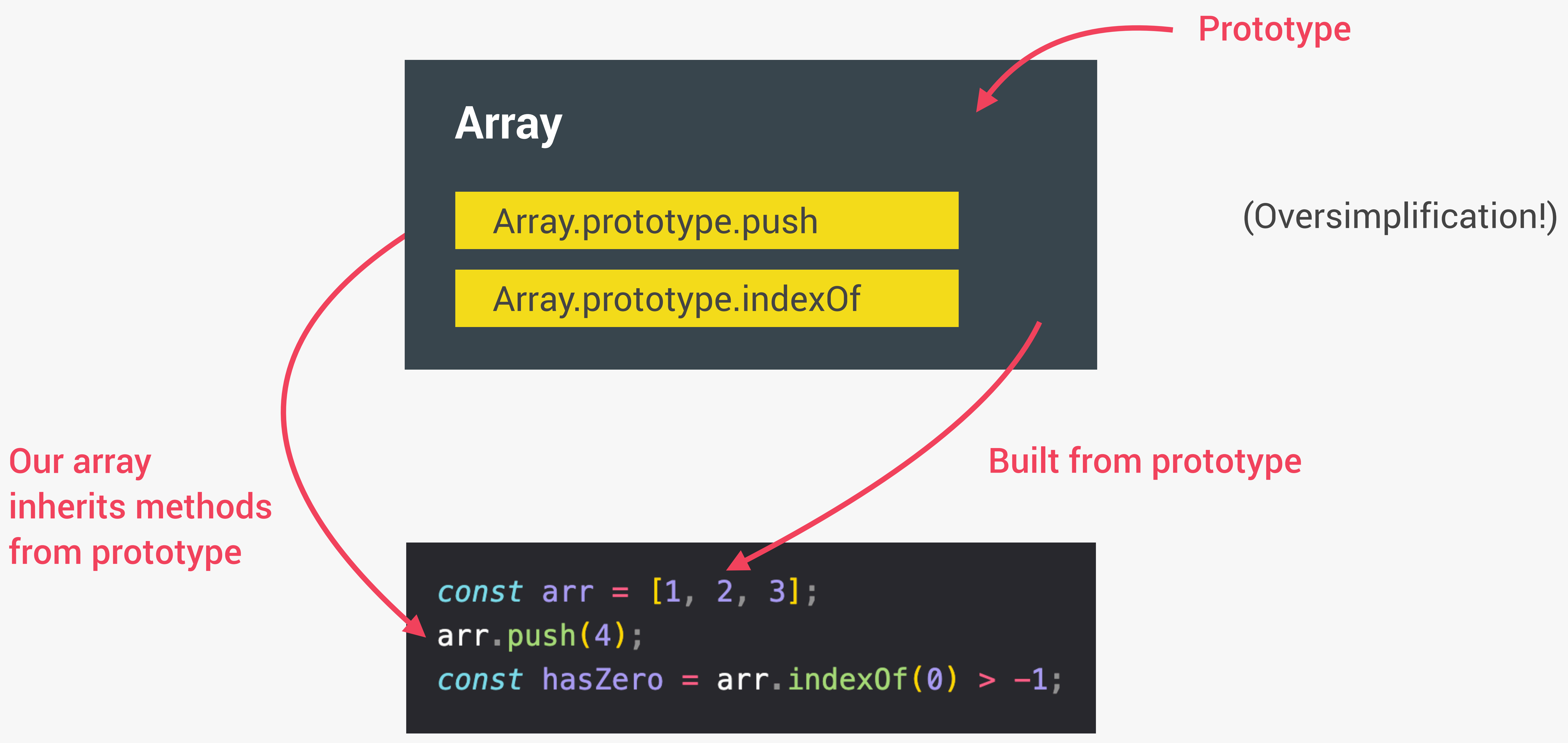
1. Procedural programming
2. Object-oriented programming (OOP)
3. Functional programming (FP)

☝️ **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



```
const arr = [1, 2, 3];
arr.push(4);
const hasZero = arr.indexOf(0) > -1;
```

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

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.

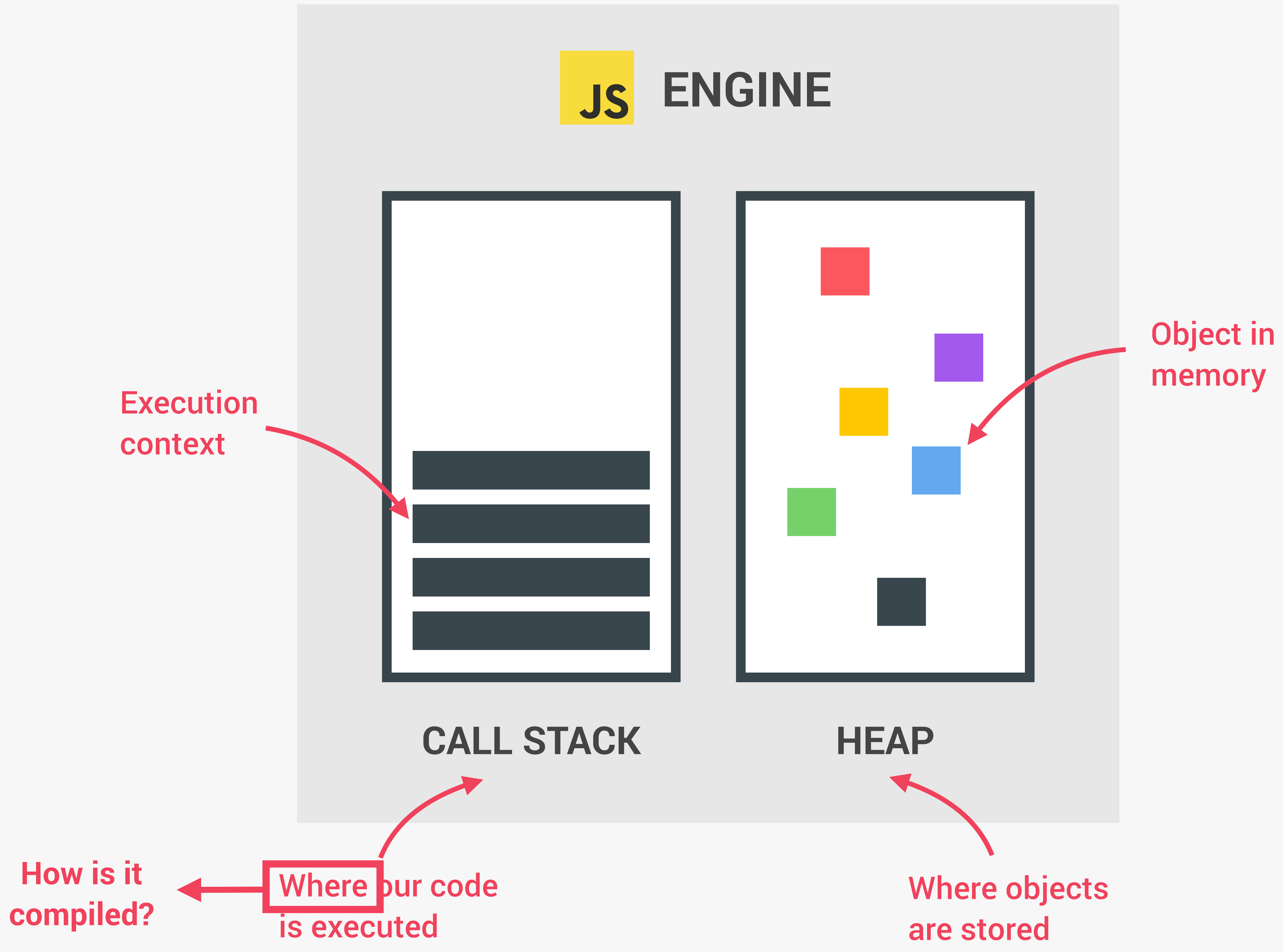
More about this **Later in this Section** 👇

WHAT IS A JAVASCRIPT ENGINE?

JS ENGINE

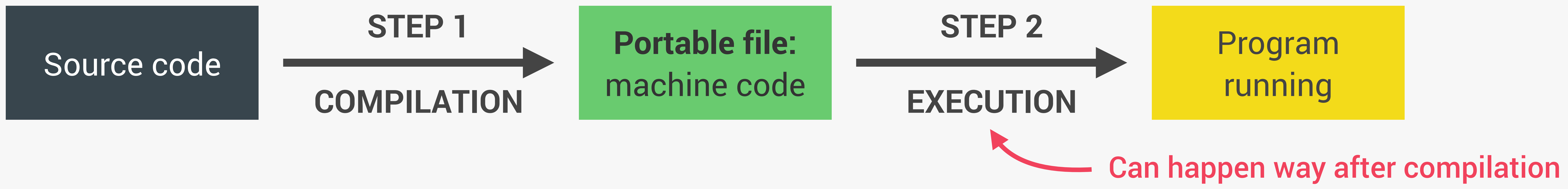
PROGRAM THAT **EXECUTES** JAVASCRIPT CODE.

👉 **Example: V8 Engine**

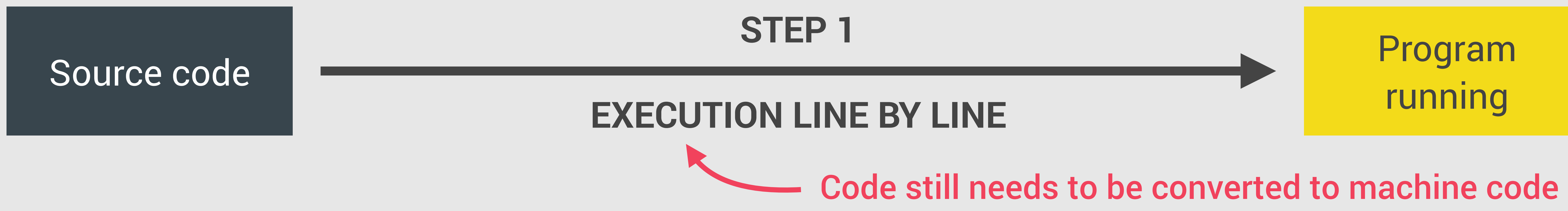


COMPUTER SCIENCE SIDENOTE: COMPILATION VS. INTERPRETATION 🤓

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

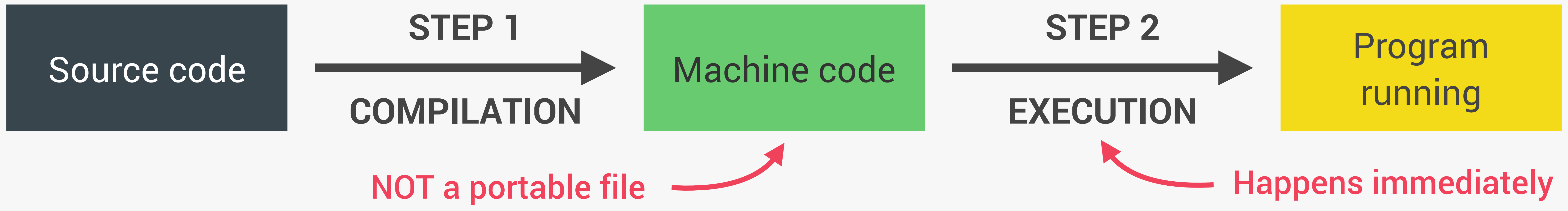


- **Interpretation:** Interpreter runs through the source code and executes it line by line.

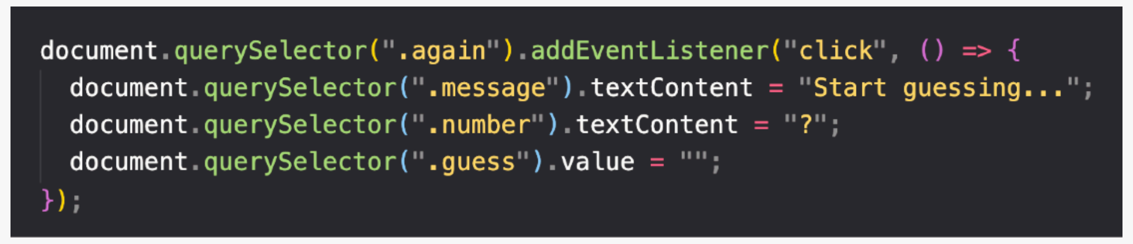


JS

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

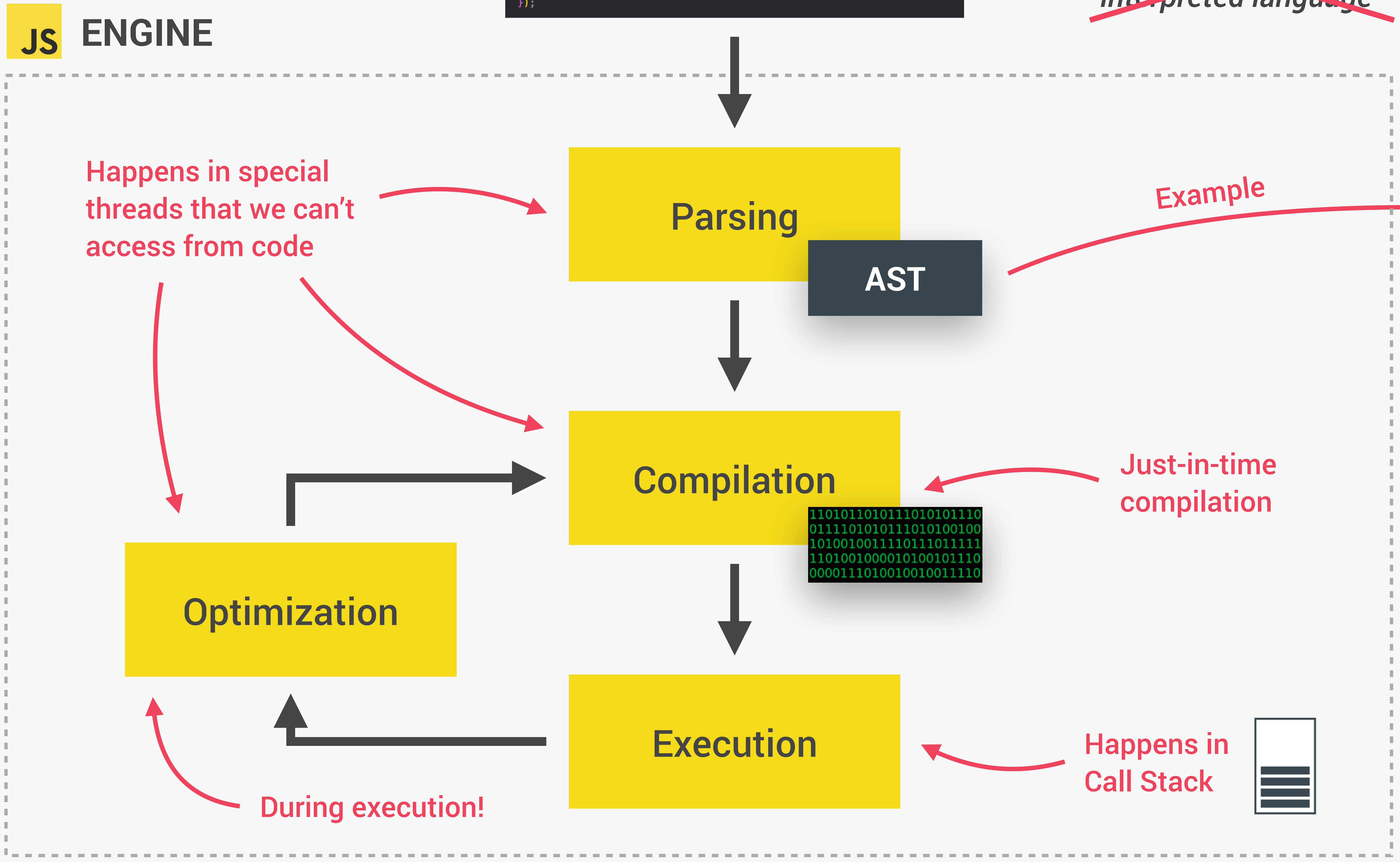


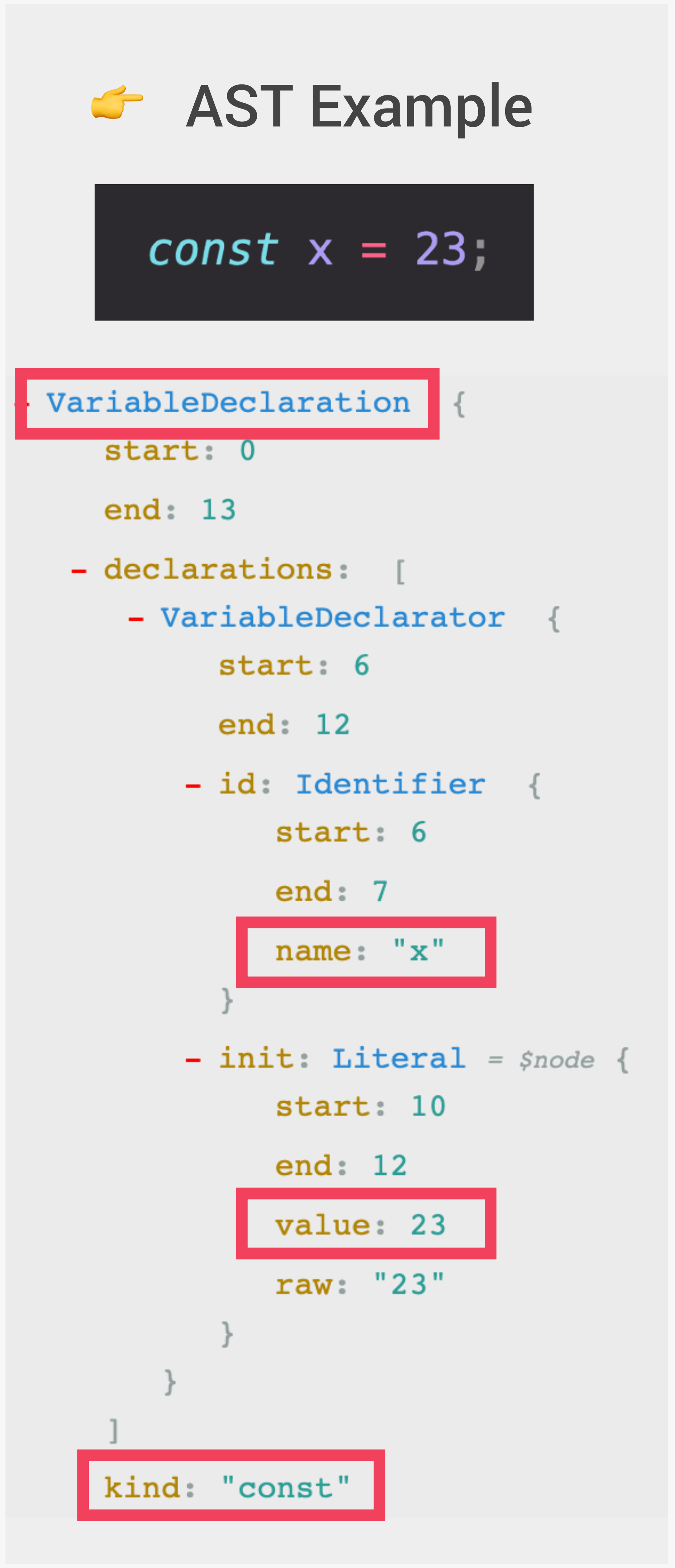
MODERN JUST-IN-TIME COMPILATION OF JAVASCRIPT



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

~~"JavaScript is an interpreted language"~~



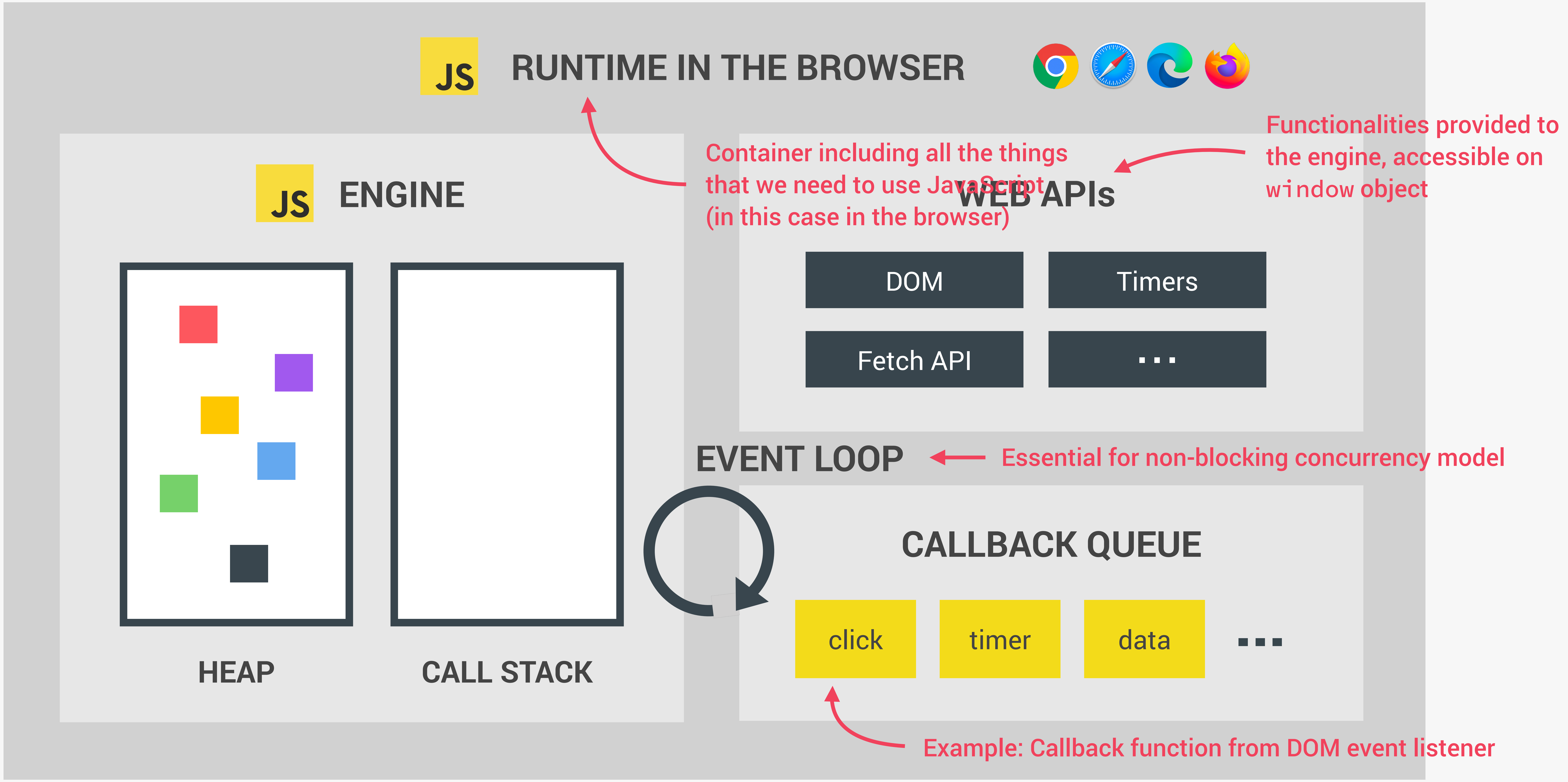


AST Example

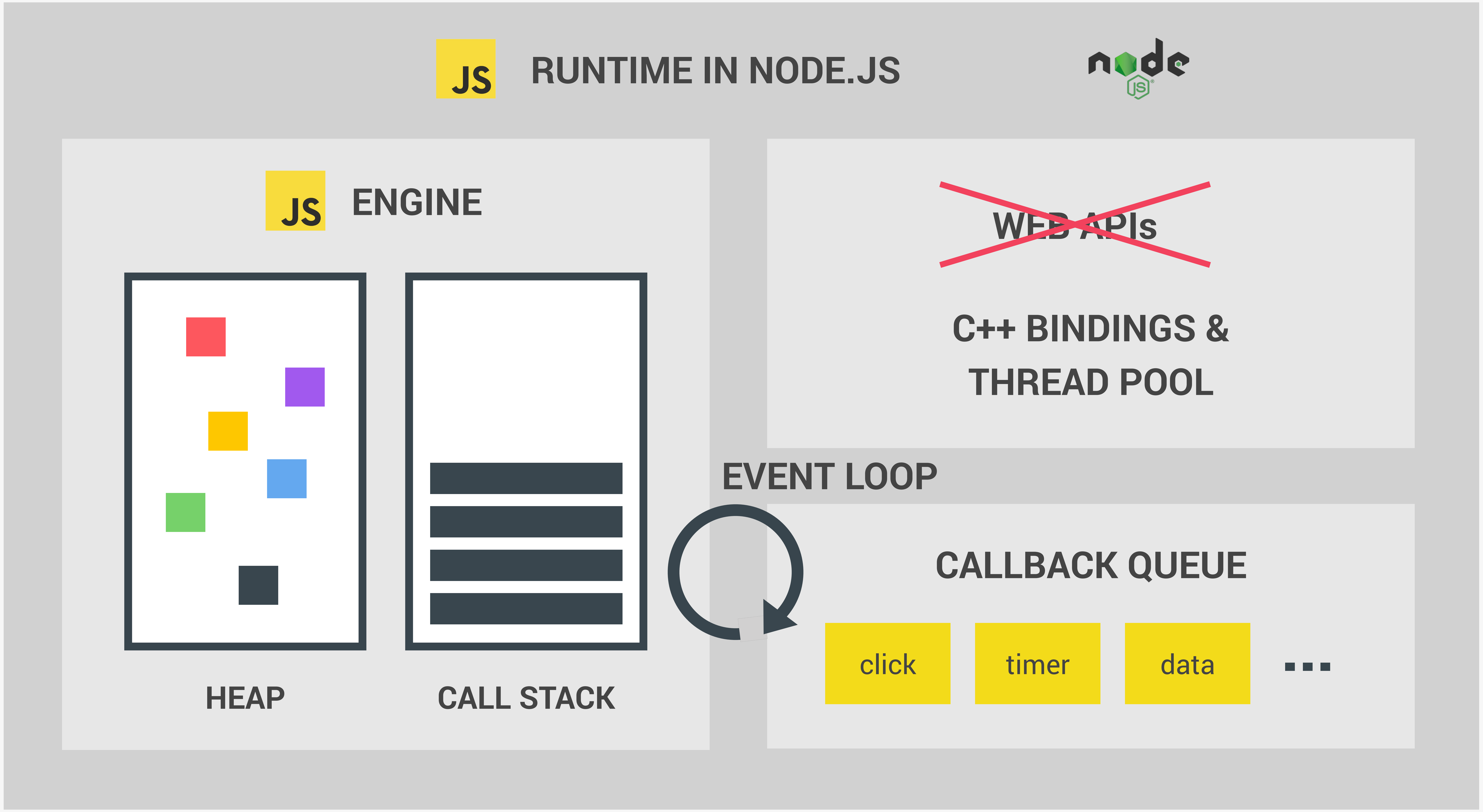
```
const x = 23;
```

```
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

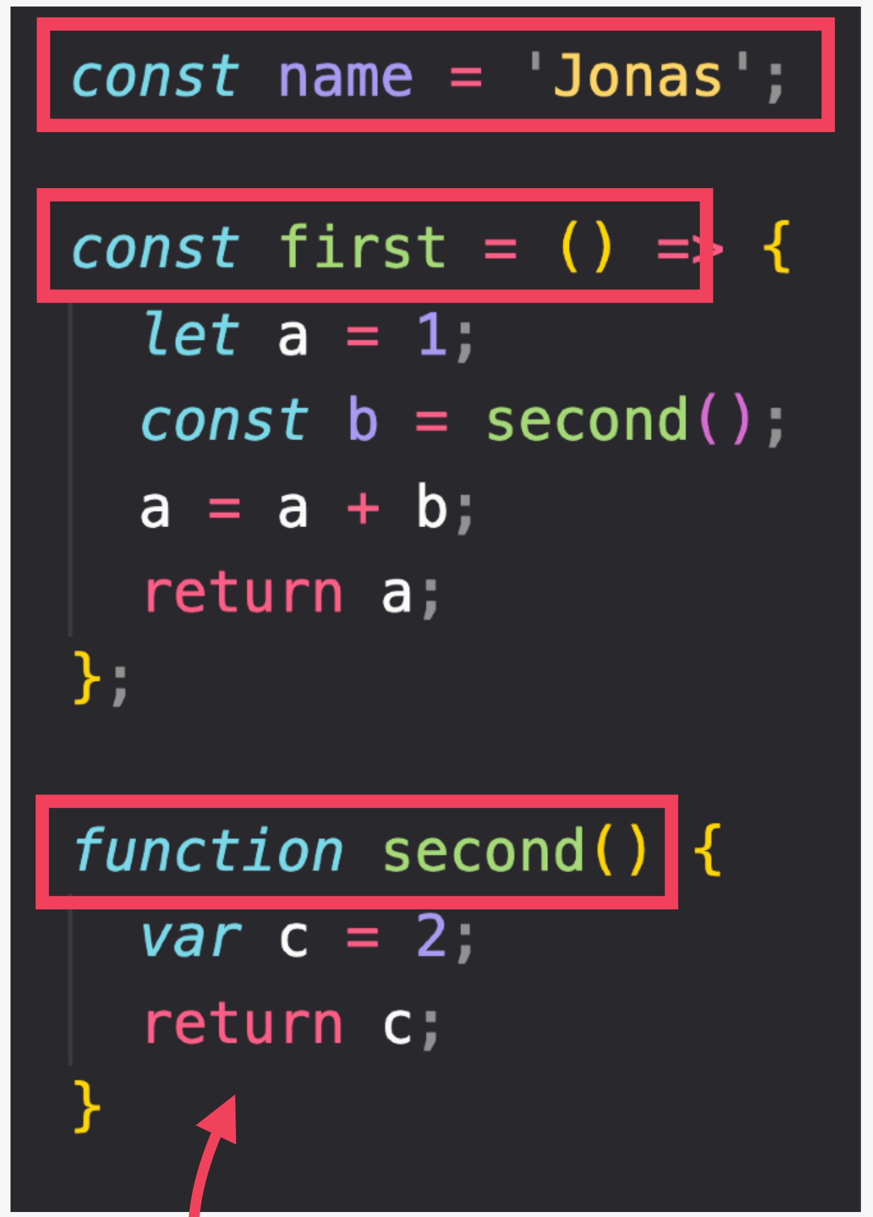

JS RUNTIME IN THE BROWSER
Container including all the things that we need to use JavaScript (in this case in the browser)
JS ENGINE
HEAP
CALL STACK
WEB APIs
Functionalities provided to the engine, accessible on window object
DOM
Timers
Fetch API
...
EVENT LOOP
Essential for non-blocking concurrency model
CALLBACK QUEUE
click
timer
data
...
Example: Callback function from DOM event listener

THE BIGGER PICTURE: JAVASCRIPT RUNTIME



WHAT IS AN EXECUTION CONTEXT?

👉 Human-readable code:

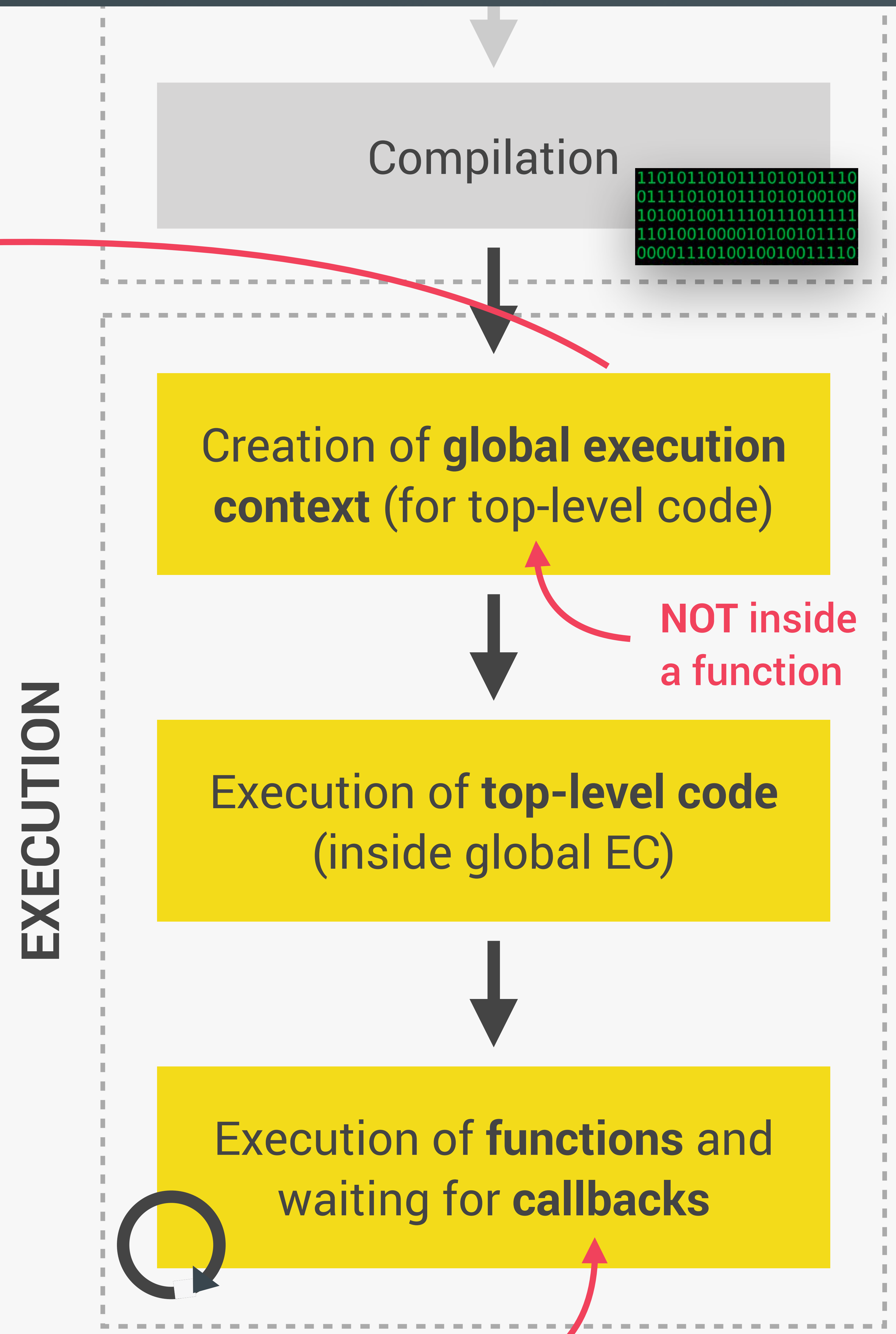


```
const name = 'Jonas';

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!



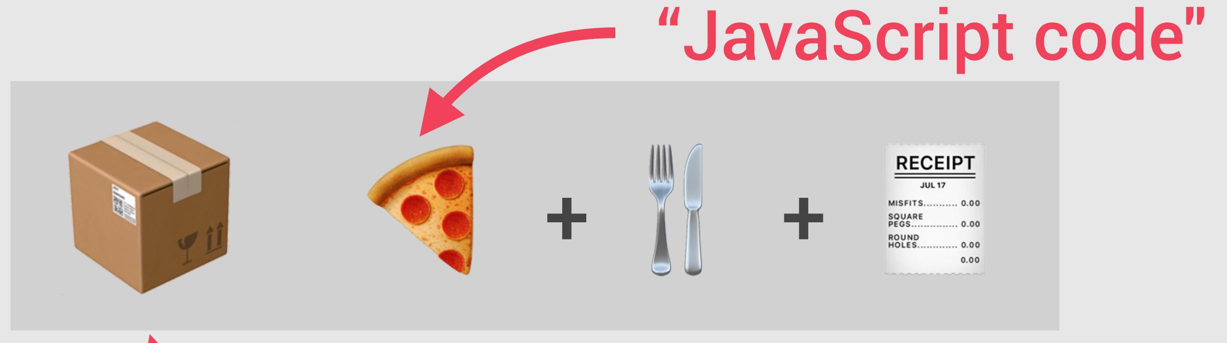
Compilation
EXECUTION
Creation of **global execution context** (for top-level code)
Execution of **top-level code** (inside global EC)
Execution of **functions** and waiting for **callbacks**

NOT inside a function

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.



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

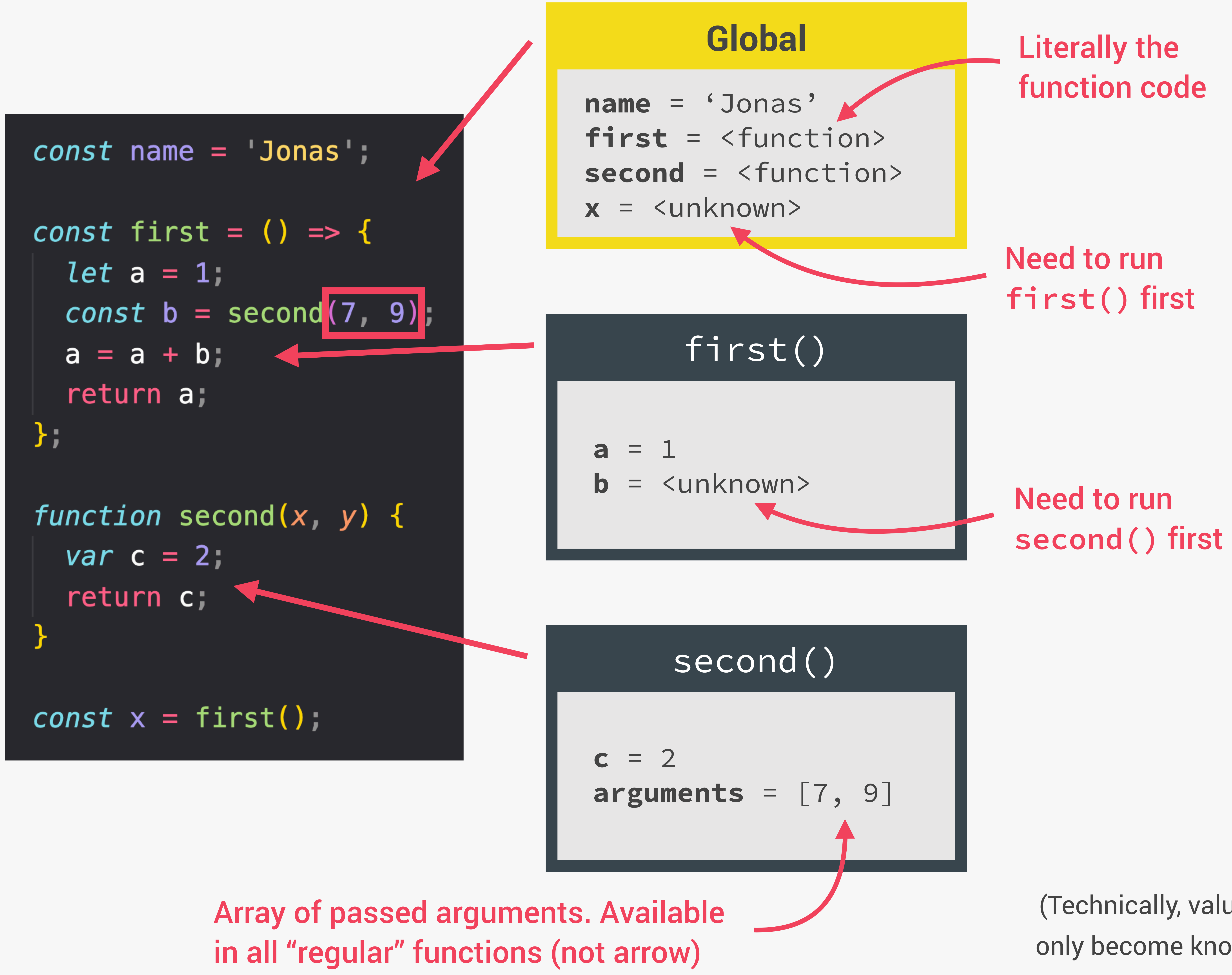
EXECUTION CONTEXT IN DETAIL

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

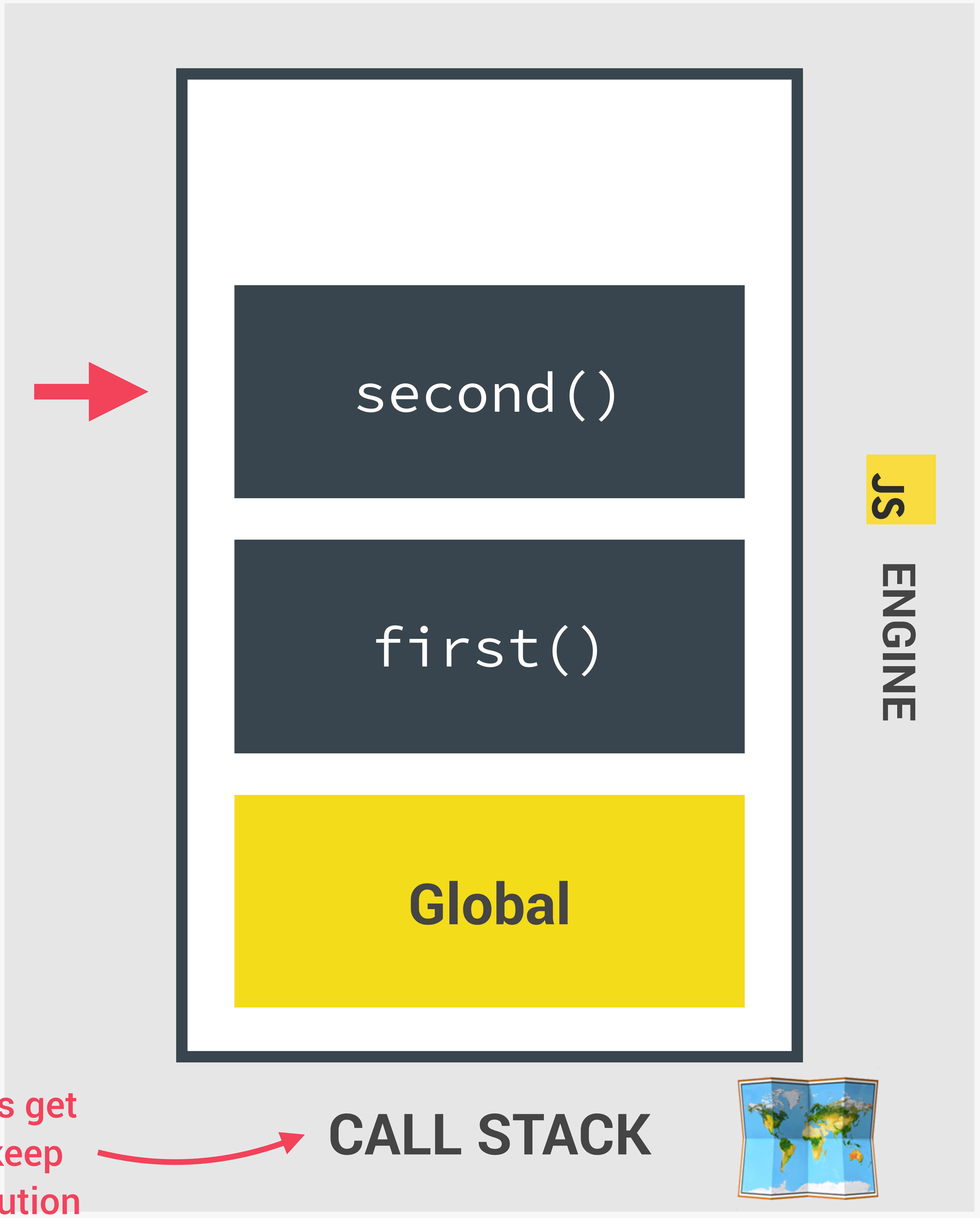
👉 Compiled code starts execution

```
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