ANECO ACADEMY

JS-
CALL_STACK_EXECUTIO
N_CONTEXT_THIS
KEYWORD_HOISTING

FIRST CLASS FUNCTION

👉 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!

DYNAMIC LANGUAGE

👉 **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";
```

SINGLE THREAD

- **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.

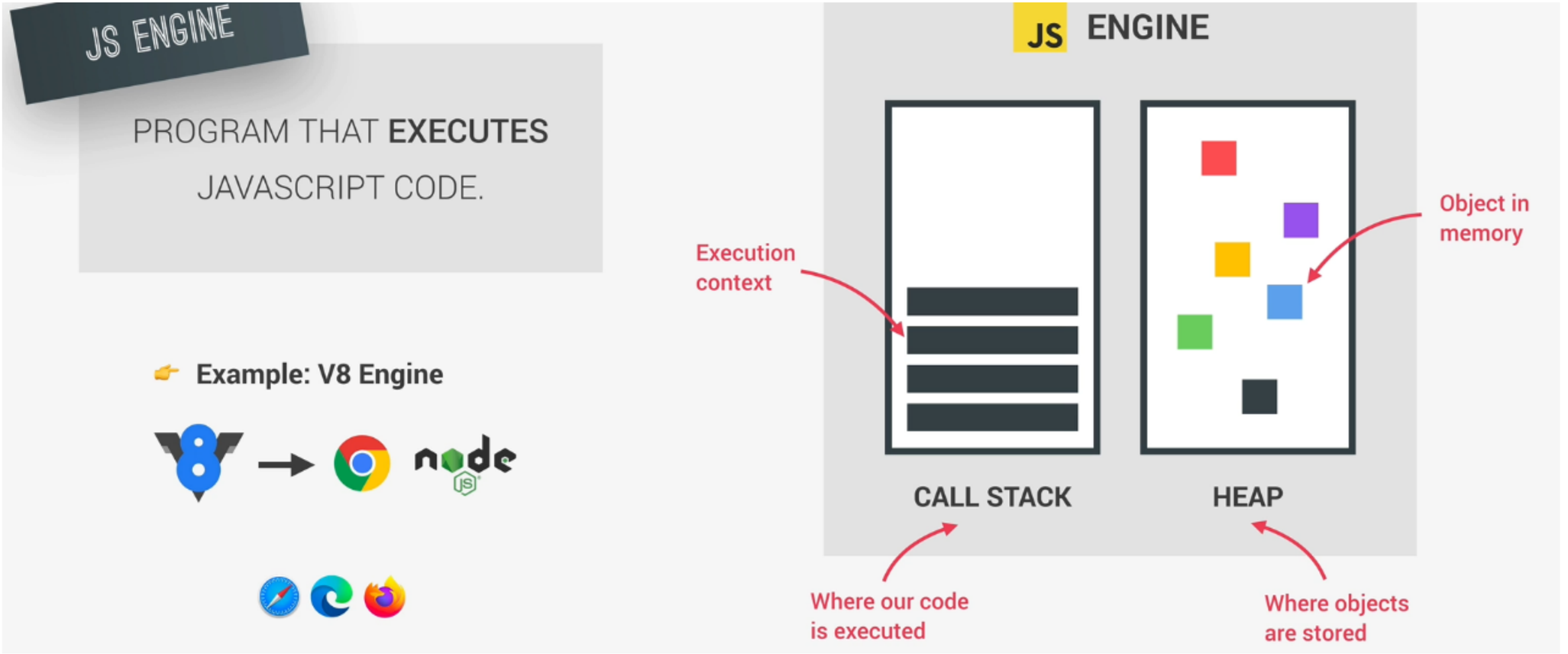
So what about a long-running task?

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

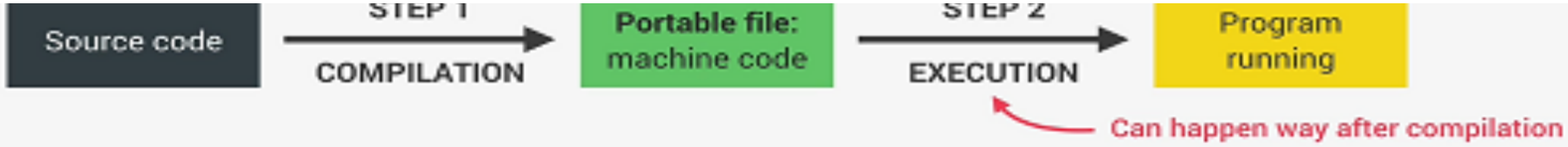
How do we achieve that?

- 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.

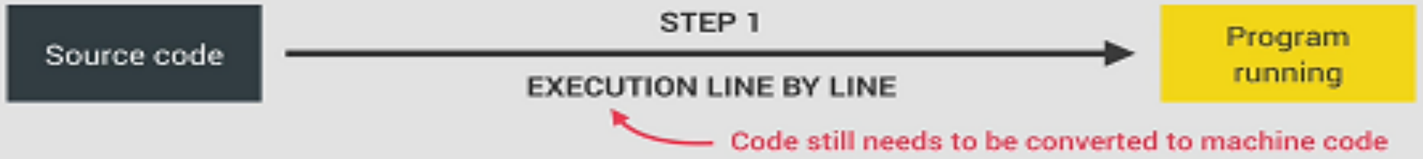
JAVASCRIPT ENGINE



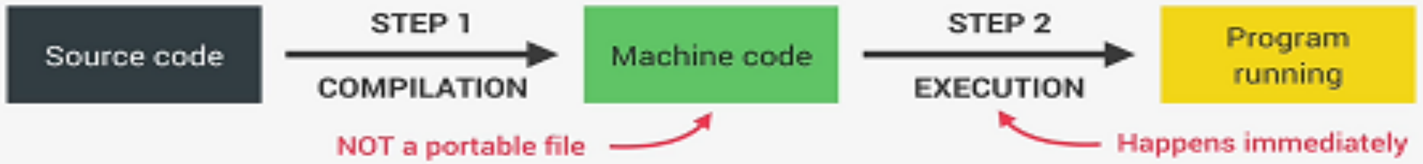
COMPILATION & INTERPREATION & JIT



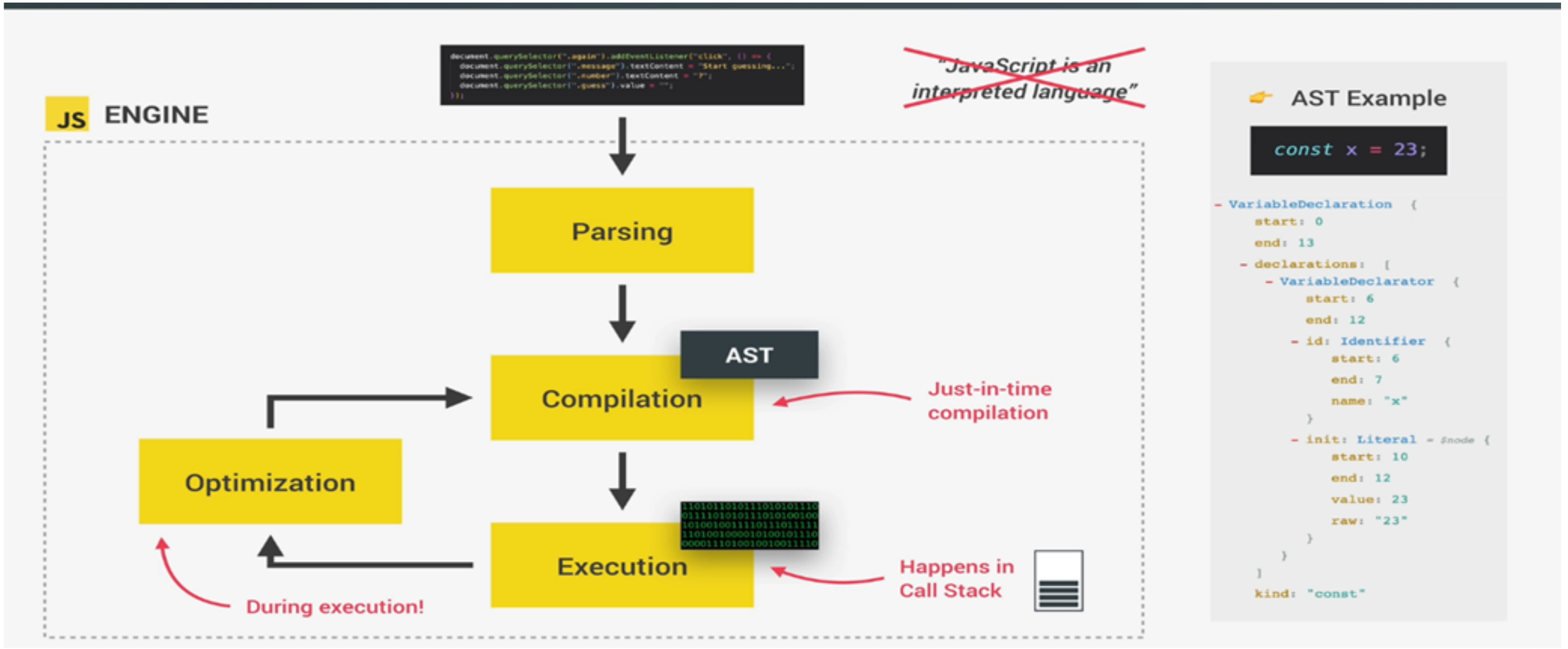
👉 **Interpretation:** Interpreter runs through the source code and executes it line by line.



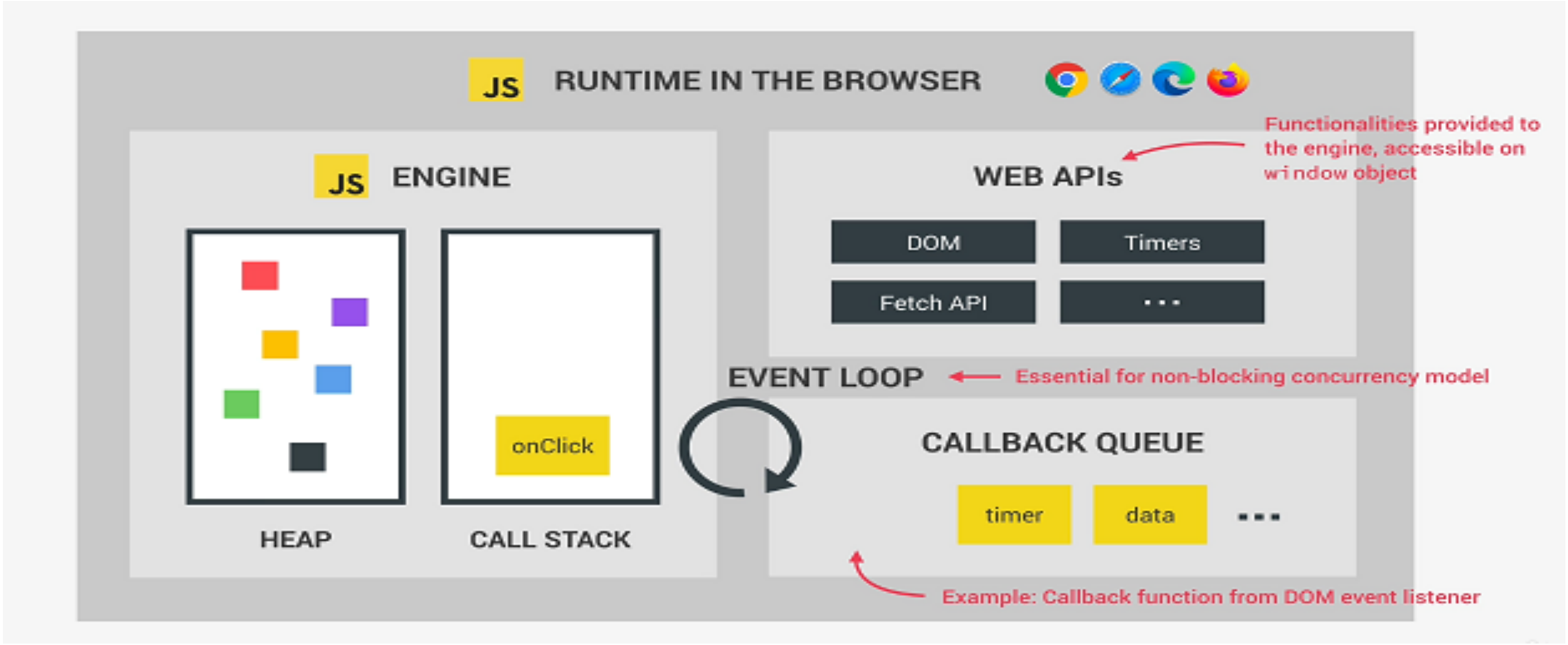
👉 **Just-in-time (JIT) compilation:** Entire code is converted into machine code at once, then executed immediately.



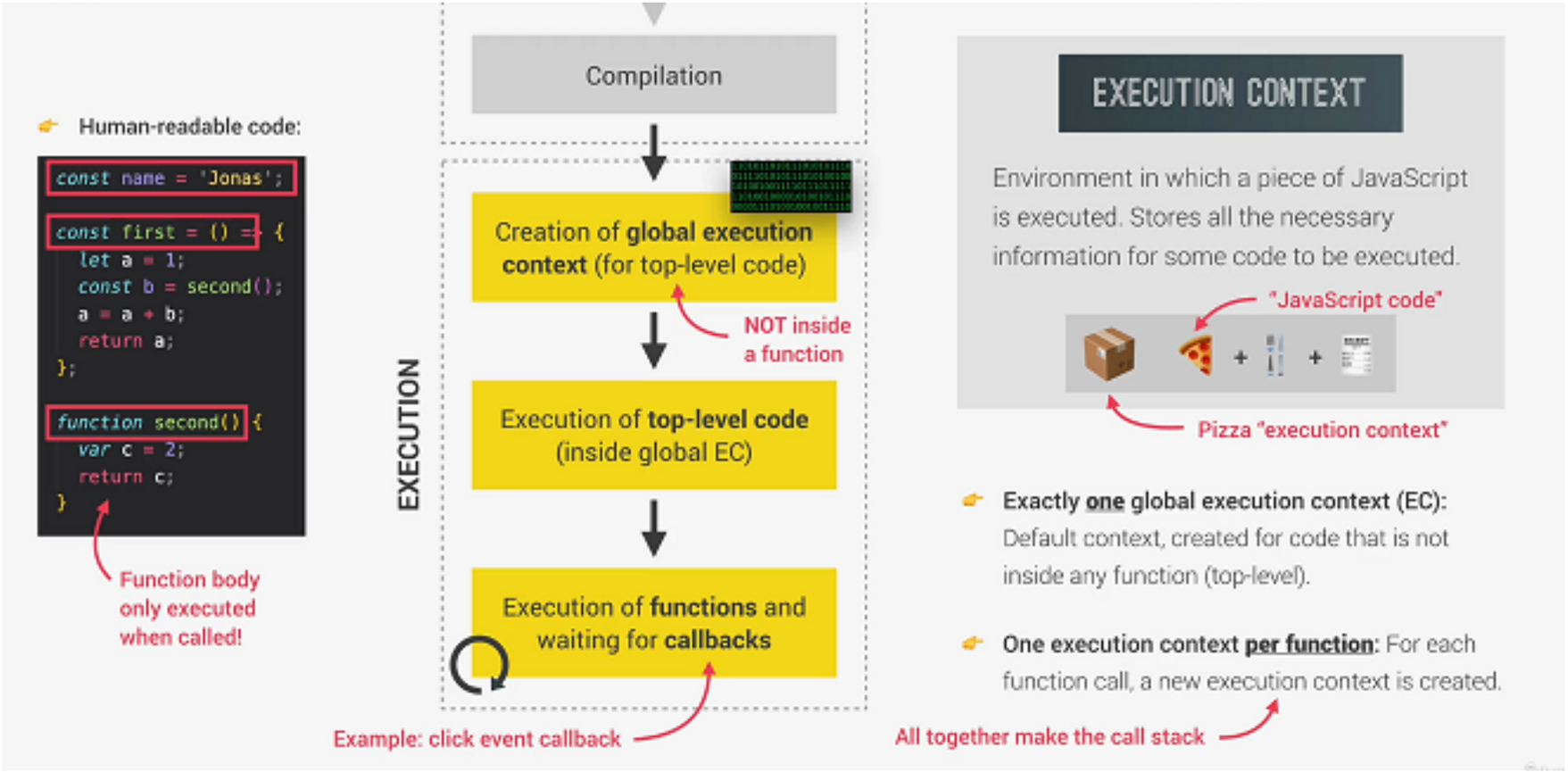
JAVASCRIPT ENGINE



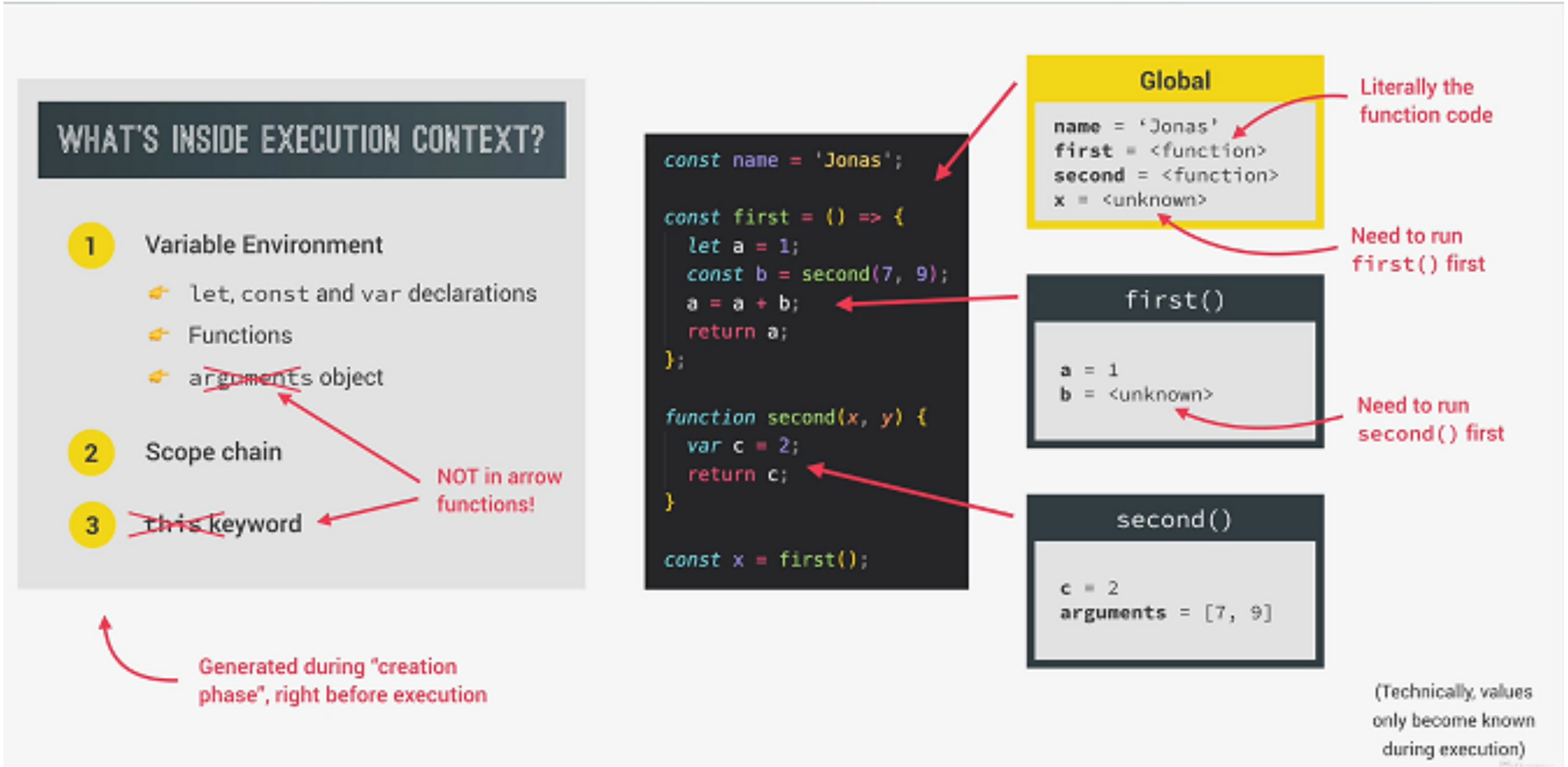
RUNTIME


JS RUNTIME IN THE BROWSER
JS ENGINE
HEAP
CALL STACK
onClick
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

EXECUTION CONTEXT



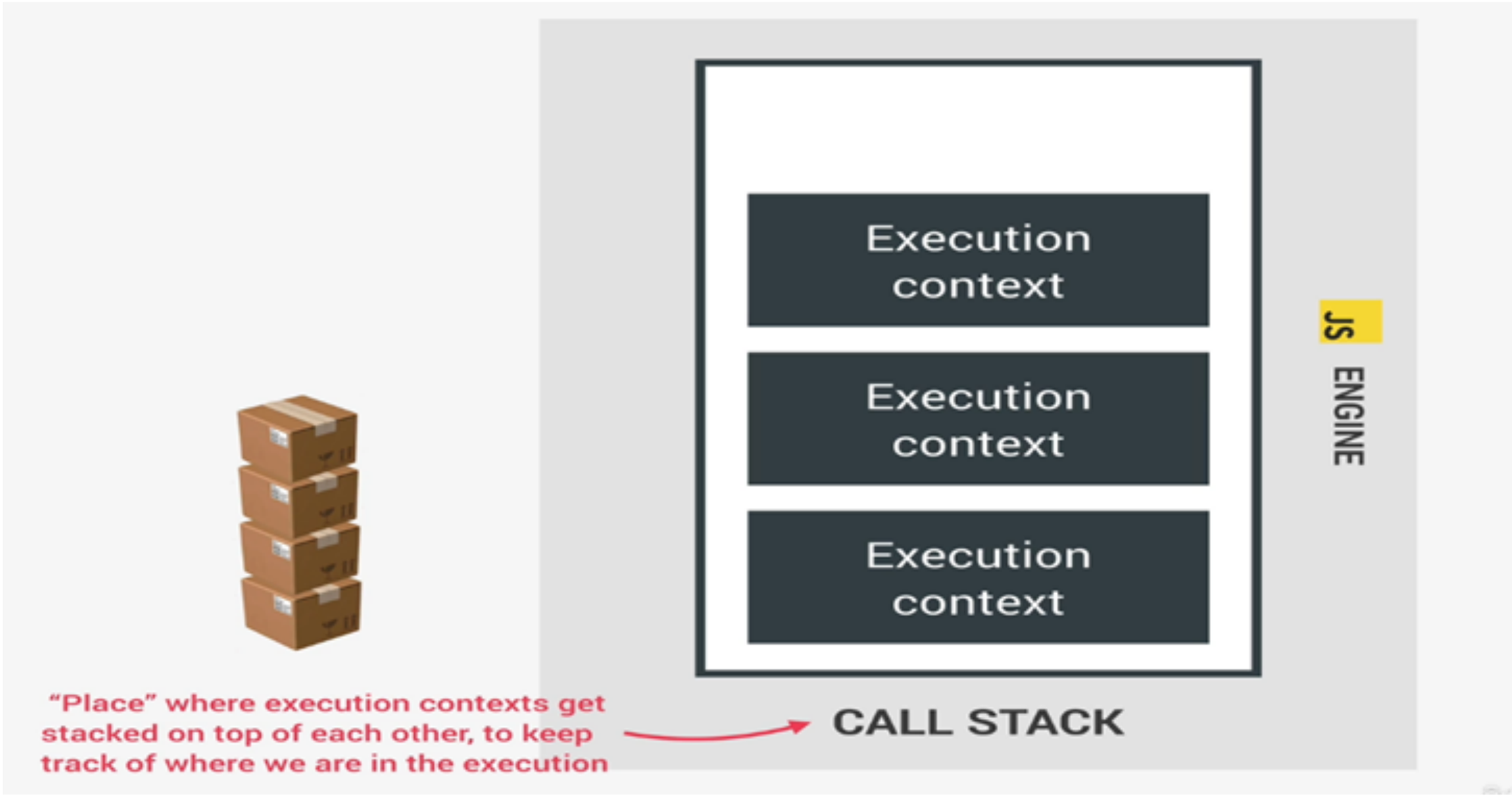
EXECUTION CONTEXT



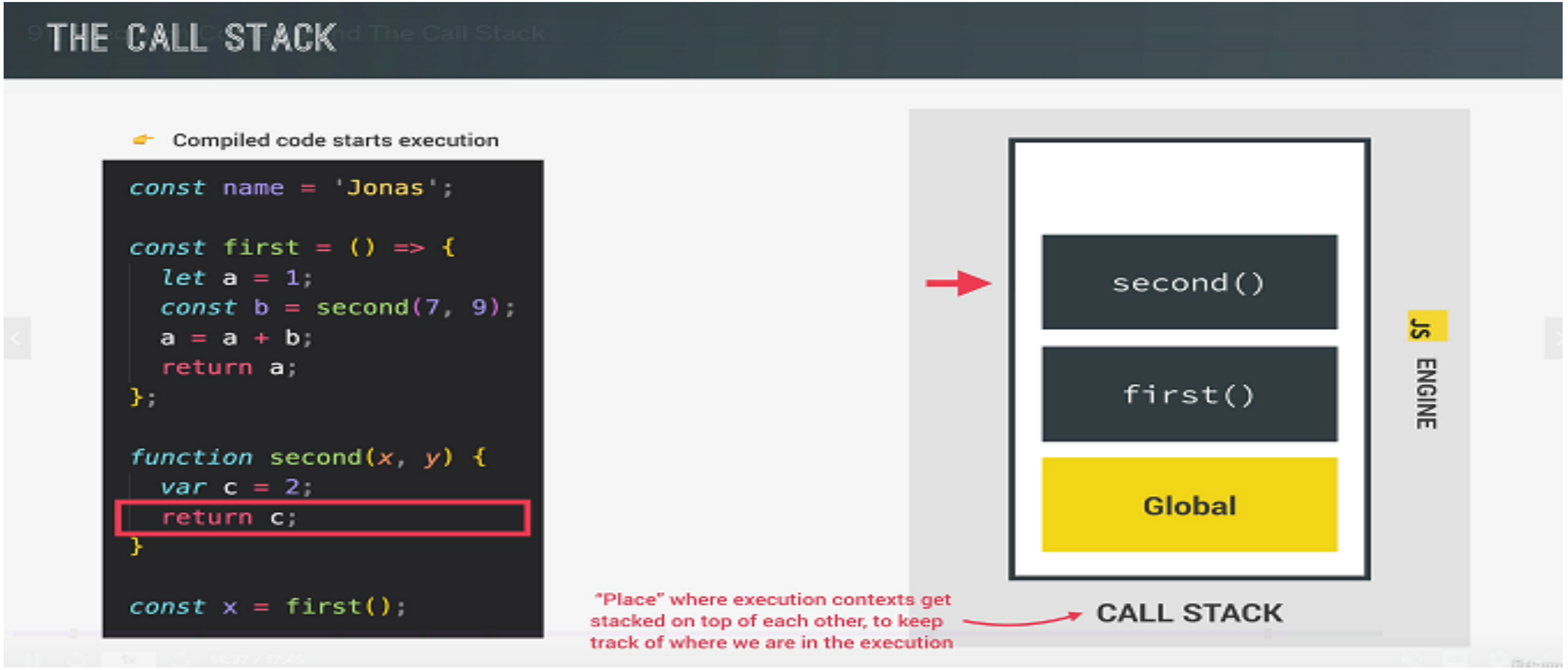
DIFFERENCE ABOUT NORMAL AND ARROW FUNCTION

```
<script>
    let obj={
        firstName:"akash",
        normalFunction:function(){
            console.log(this.firstName);
        },
        arrowFunction:()=>{
            console.log(this.firstName);
        }
    }
obj.normalFunction();
obj.arrowFunction();
</script>
```

EXECUTION CONTEXT



CALL STACK


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();
second()
first()
Global
JS ENGINE
"Place" where execution contexts get stacked on top of each other, to keep track of where we are in the execution
CALL STACK

CALL STACK

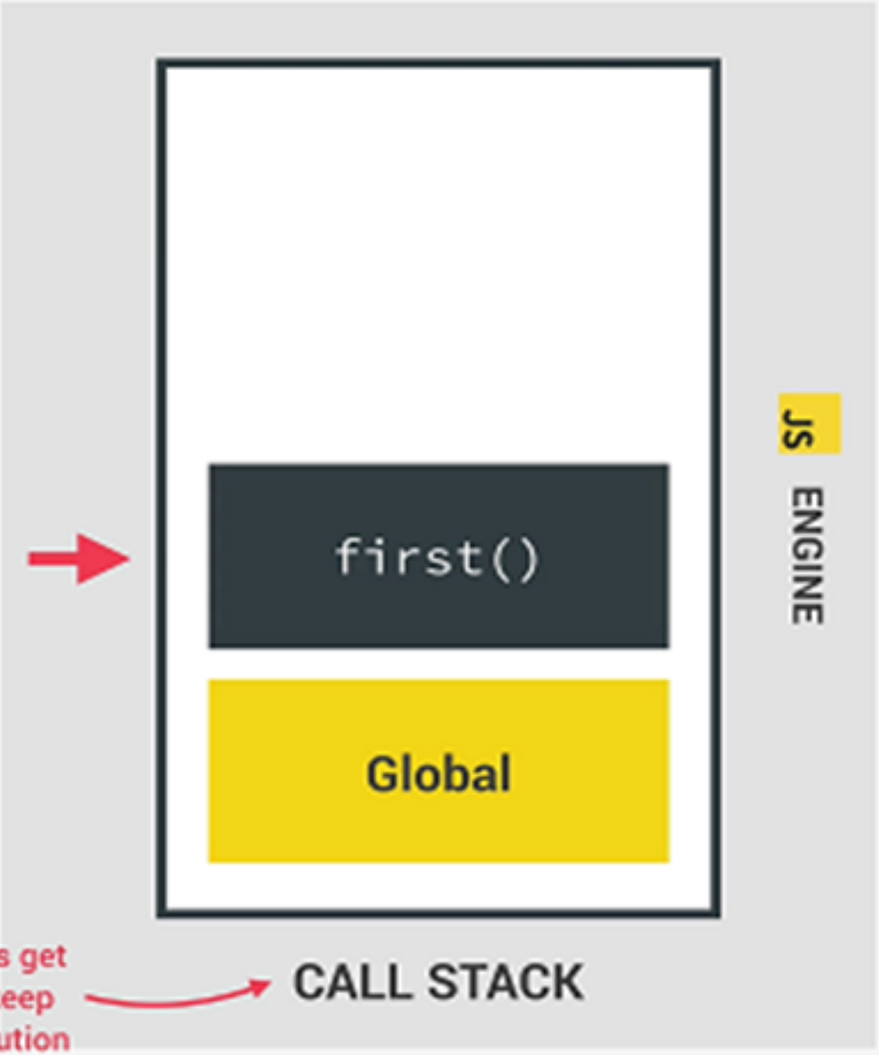
👉 Compiled code starts execution

```
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();
```



CALL STACK

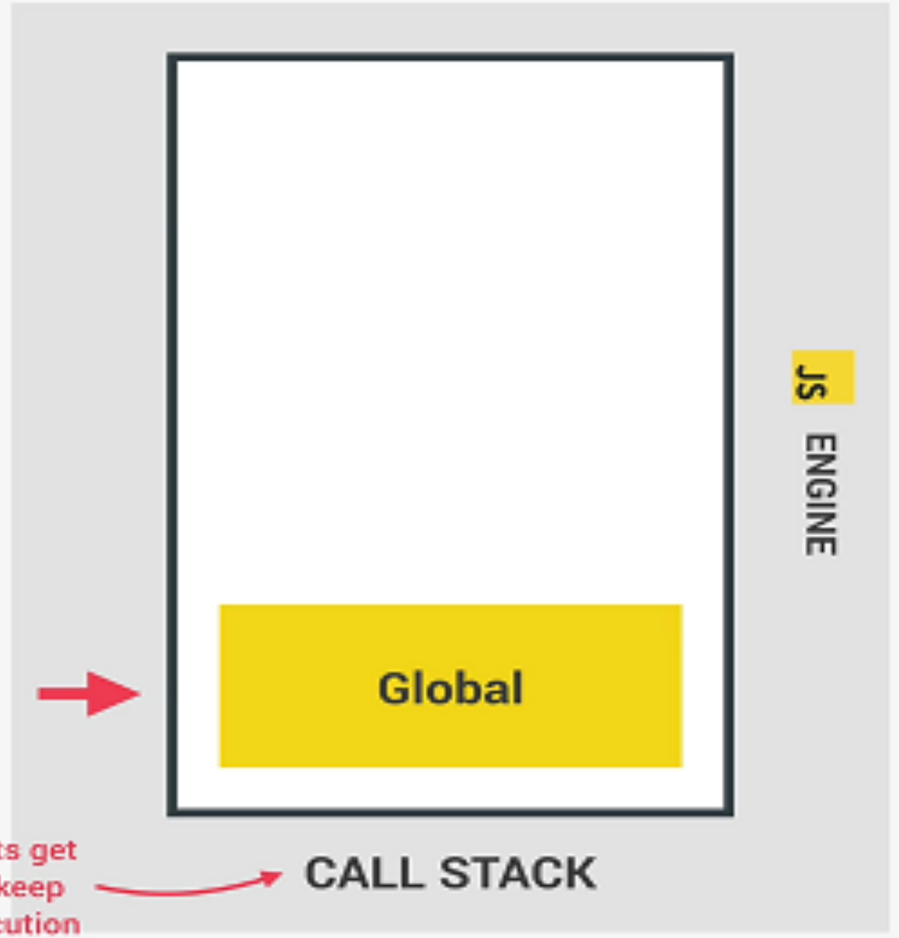
👉 Compiled code starts execution

```
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

CALL STACK

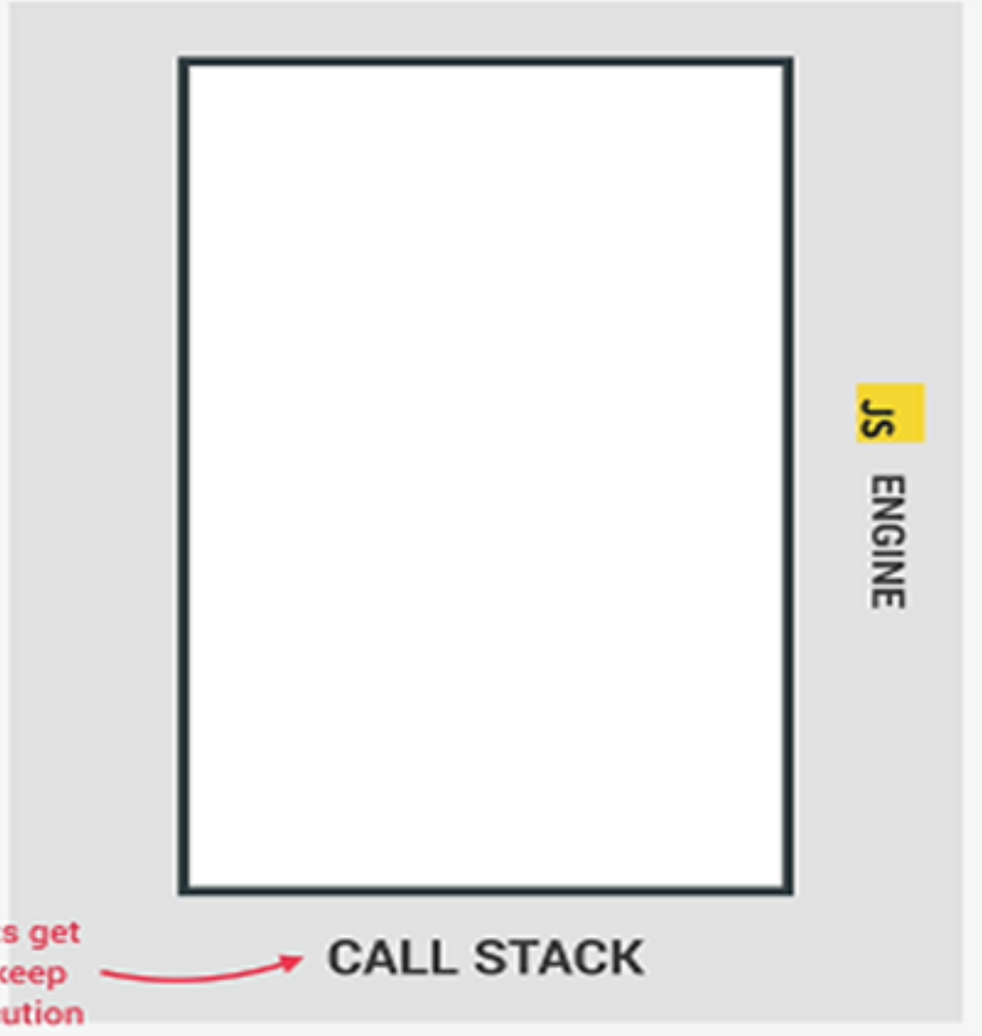
👉 Compiled code starts execution

```
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

CALL STACK - HOISTING

👉 **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	`<uninitialized>`, TDZ	Block
`function` expressions and arrows	🤷 Depends if using `var` or `let/const`		

In strict mode. Otherwise: function!

Technically, yes. But not in practice

Temporal Dead Zone

TDZ

```
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
```

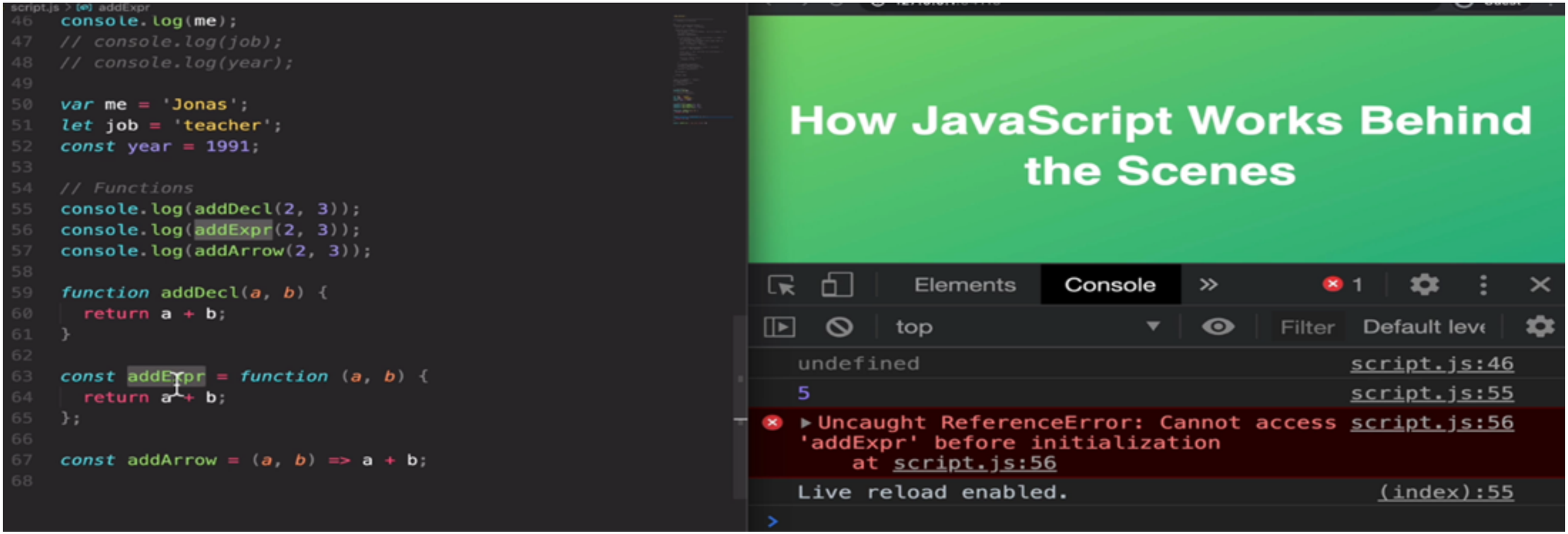
WHY HOISTING?

- Using functions before actual declaration;

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

HOISTING


script.js > addExpr
46 console.log(me);
47 // console.log(job);
48 // console.log(year);
49
50 var me = 'Jonas';
51 let job = 'teacher';
52 const year = 1991;
53
54 // Functions
55 console.log(addDecl(2, 3));
56 console.log(addExpr(2, 3));
57 console.log(addArrow(2, 3));
58
59 function addDecl(a, b) {
60 return a + b;
61 }
62
63 const addExpr = function (a, b) {
64 return a + b;
65 };
66
67 const addArrow = (a, b) => a + b;
68
How JavaScript Works Behind the Scenes
Elements
Console
1
top
Filter
Default leve
undefined
script.js:46
5
script.js:55
Uncaught ReferenceError: Cannot access 'addExpr' before initialization
at script.js:56
script.js:56
Live reload enabled.
(index):55

HOISTING

👉 **`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 👉 `this` = <Object that is calling the method>

Simple function call 👉 `this` = `undefined`

In strict mode! Otherwise: `window` (in the browser)

Don't get own `this`

Arrow functions 👉 `this` = <`this` of surrounding function (lexical `this`)>

Event listener 👉 `this` = <DOM element that the handler is attached to>

`new`, `call`, `apply`, `bind` 👉 <*Later in the course...* ⏳>

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

👉 Method example:

```
const jonas = {
  name: 'Jonas',
  year: 1989,
  calcAge: function() {
    return 2037 - this.year;
  }
};
jonas.calcAge(); // 48
```

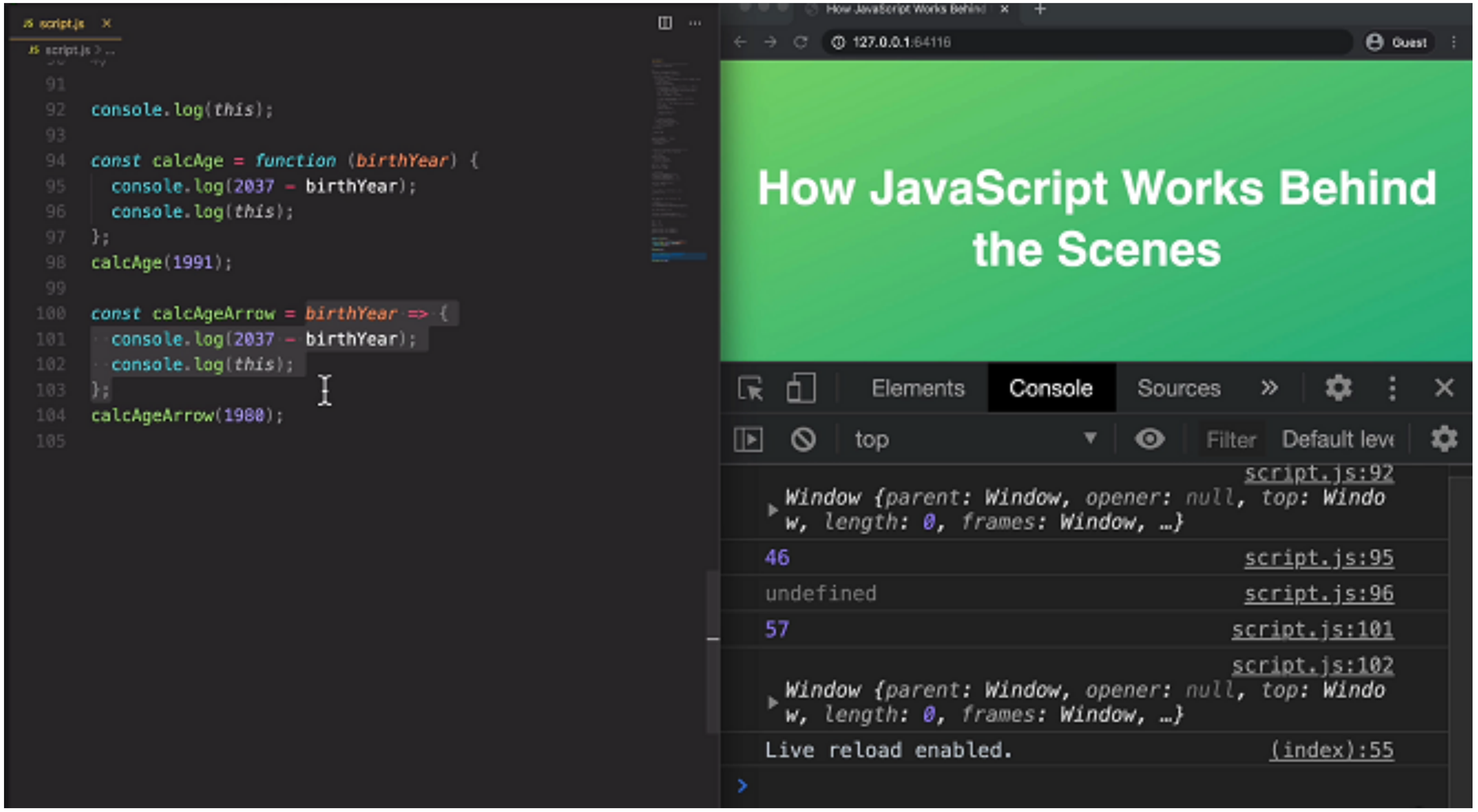
`calcAge` is method

`jonas`

`1989`

Way better than using `jonas.year`!

HOISTING


script.js
91
92 console.log(this);
93
94 const calcAge = function (birthYear) {
95 console.log(2037 - birthYear);
96 console.log(this);
97 };
98 calcAge(1991);
99
100 const calcAgeArrow = birthYear => {
101 console.log(2037 - birthYear);
102 console.log(this);
103 };
104 calcAgeArrow(1980);
105
127.0.0.1:64116
Guest
How JavaScript Works Behind the Scenes
Elements
Console
Sources
top
Filter
Default levels
script.js:92
Window {parent: Window, opener: null, top: Window, length: 0, frames: Window, …}
46
script.js:95
undefined
script.js:96
57
script.js:101
script.js:102
Window {parent: Window, opener: null, top: Window, length: 0, frames: Window, …}
Live reload enabled.
(index):55

HOISTING

```
<script>
var akash = {
  name: 'Akash',
  showTasks: function() {
    var _this = this;
    function display(){
        console.log(_this.name);
    }
    display();
  }
};
akash.showTasks();
</script>
```

```
<script>
var akash = {
  name: 'Akash',
  showTasks: function() {
    const display=()=>{
        console.log(this.name);
    }
    display();
  }
};
akash.showTasks();
</script>
```