

# Javascript-Behind the science

# 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

## 👉 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 THREADED AND 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.

# JAVASCRIPT ENGINE

JS ENGINE

PROGRAM THAT **EXECUTES**  
JAVASCRIPT CODE.

👉 Example: V8 Engine



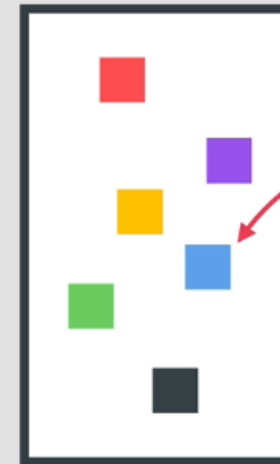
JS ENGINE

Execution  
context



CALL STACK

Where our code  
is executed



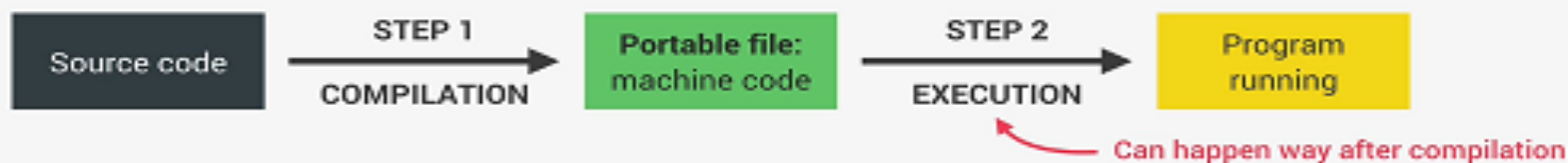
HEAP

Where objects  
are stored

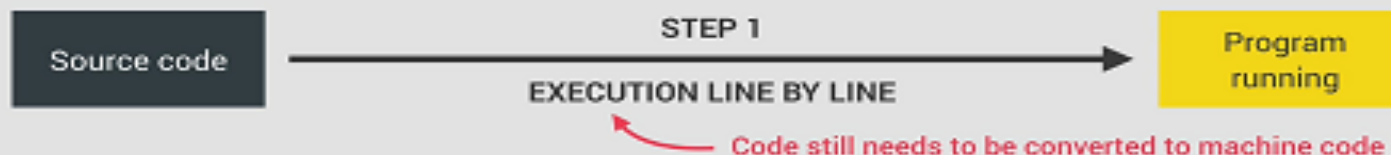
Object in  
memory

# COMPILE AND INTERPRETATION

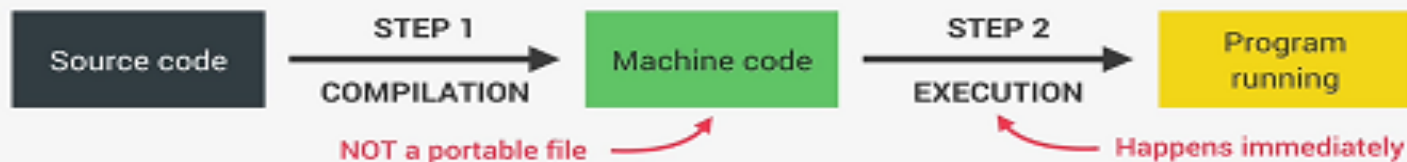
👉 **Compilation:** Entire code is converted into machine code at once, and written to a binary file that can be executed by a computer.



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

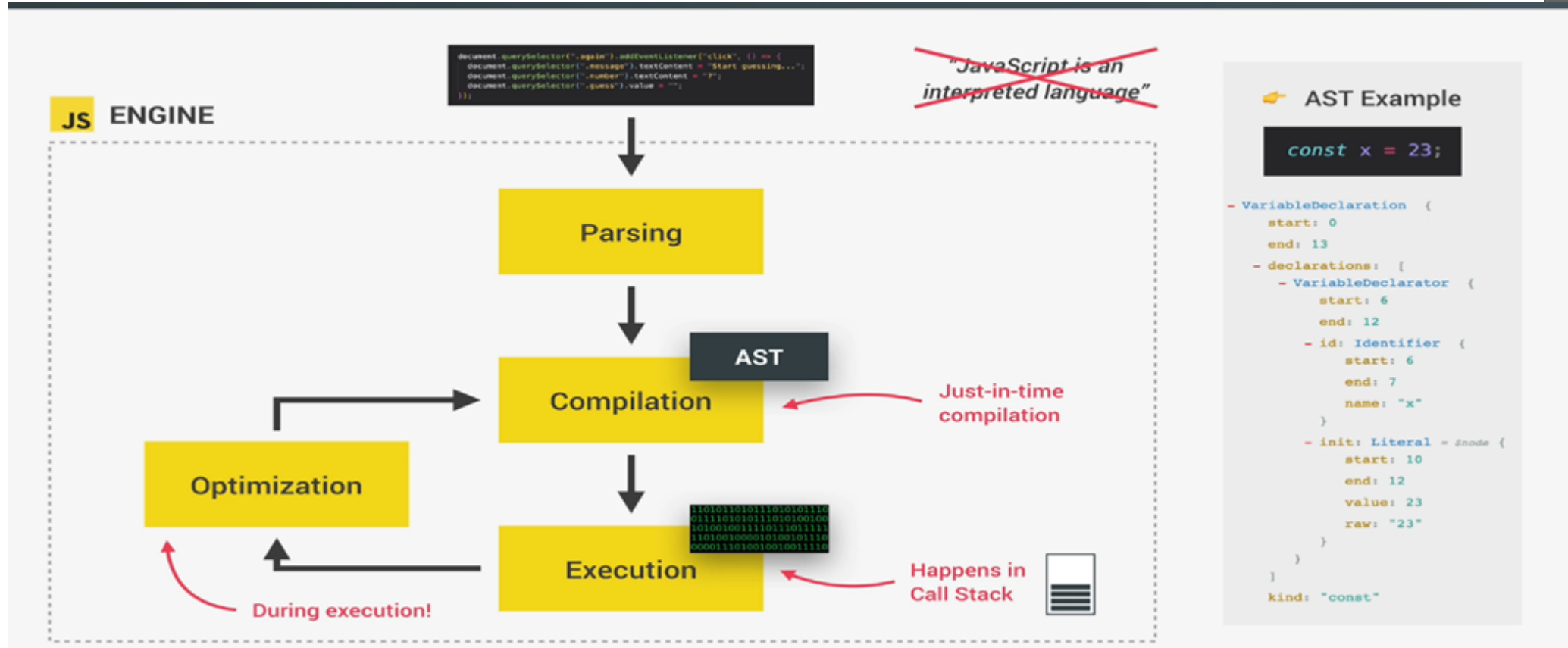


👉 **Just-in-time (JIT) compilation:** Entire code is converted into machine code at once, then executed immediately.

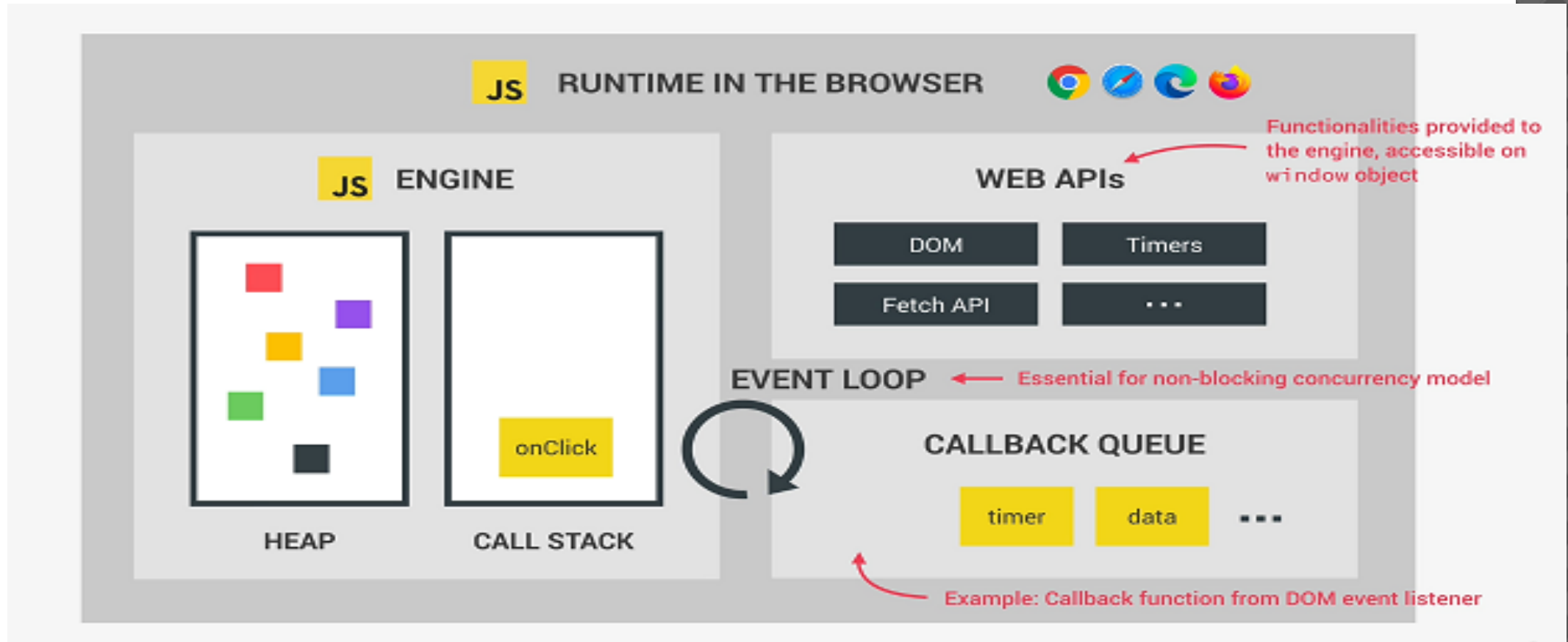


JS

# COMPILATION OF JAVASCRIPT

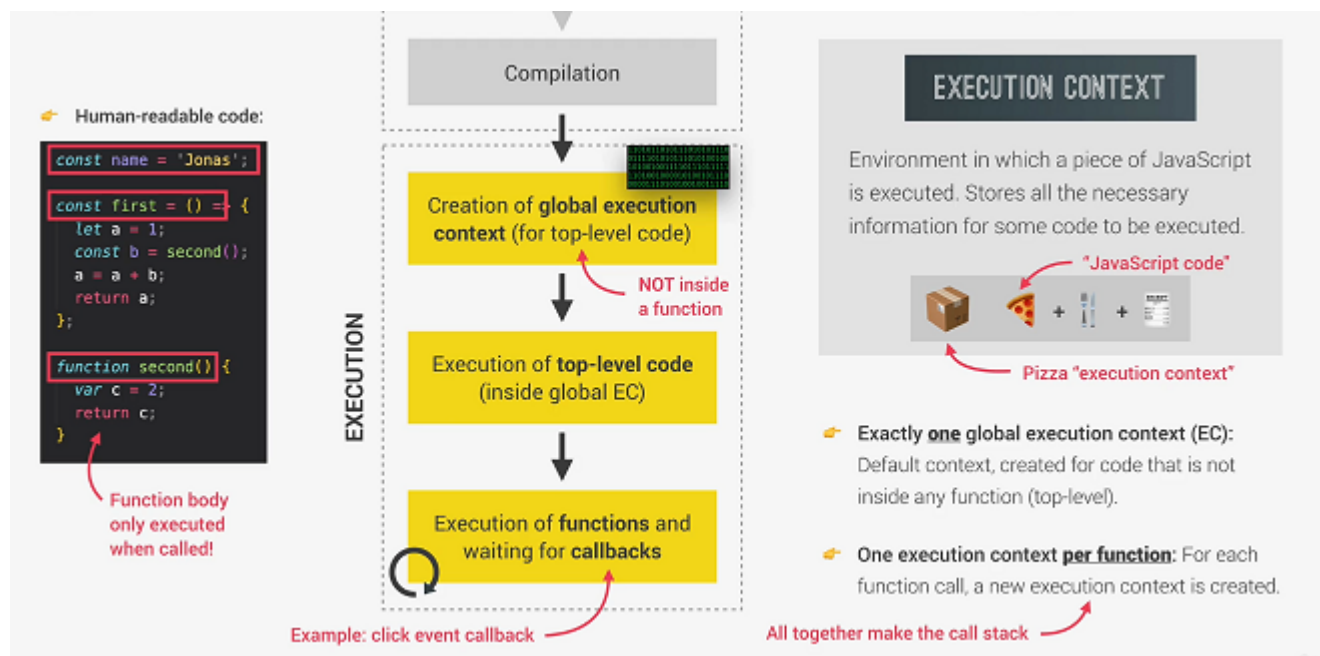


# JAVASCRIPT RUNTIME

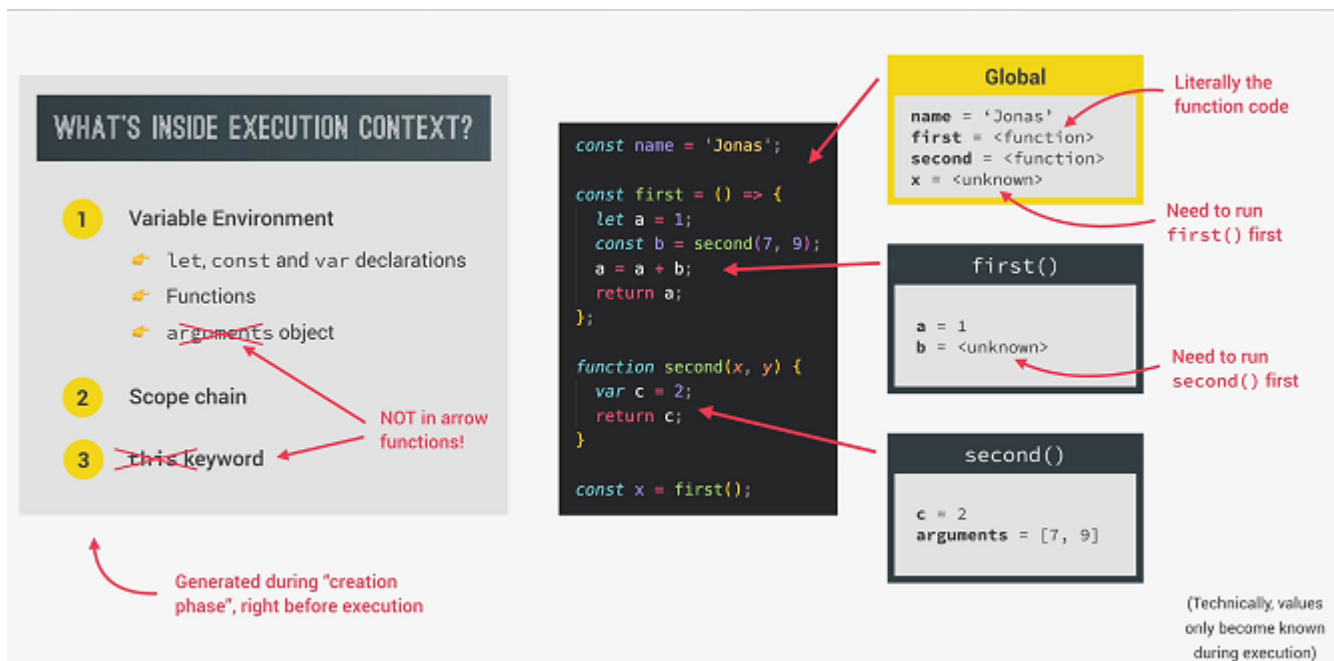




# EXECUTION CONTEXT



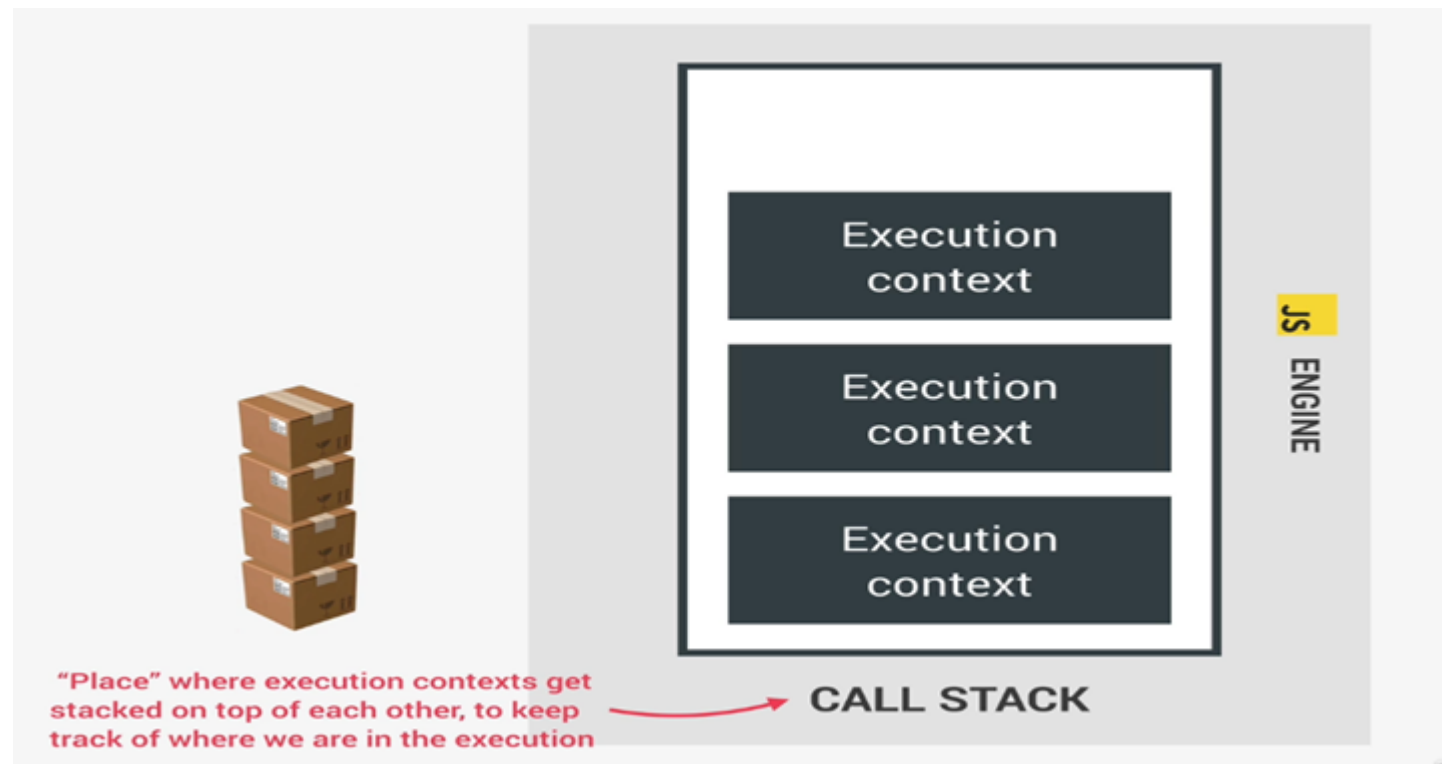
# EXECUTION CONTEXT



# NORMAL FUNCTION 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



# EXECUTION CONTEXT : CALL STACK - 1

## 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();
```

JS ENGINE

CALL STACK

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

# EXECUTION CONTEXT : CALL STACK - 2

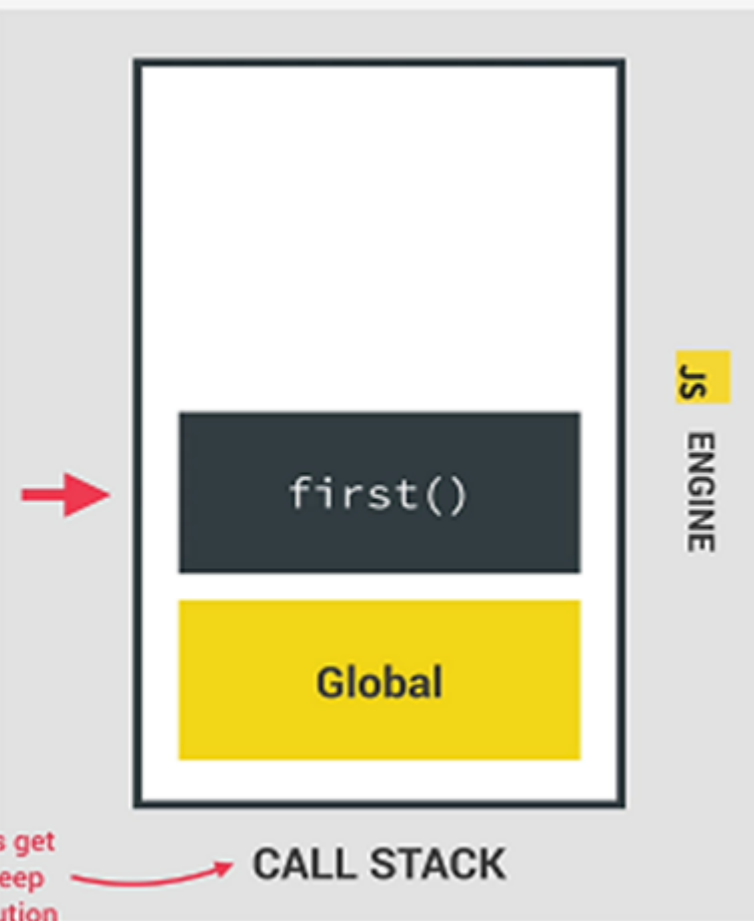
👉 Compiled code starts execution

```
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

# EXECUTION CONTEXT : CALL STACK - 3

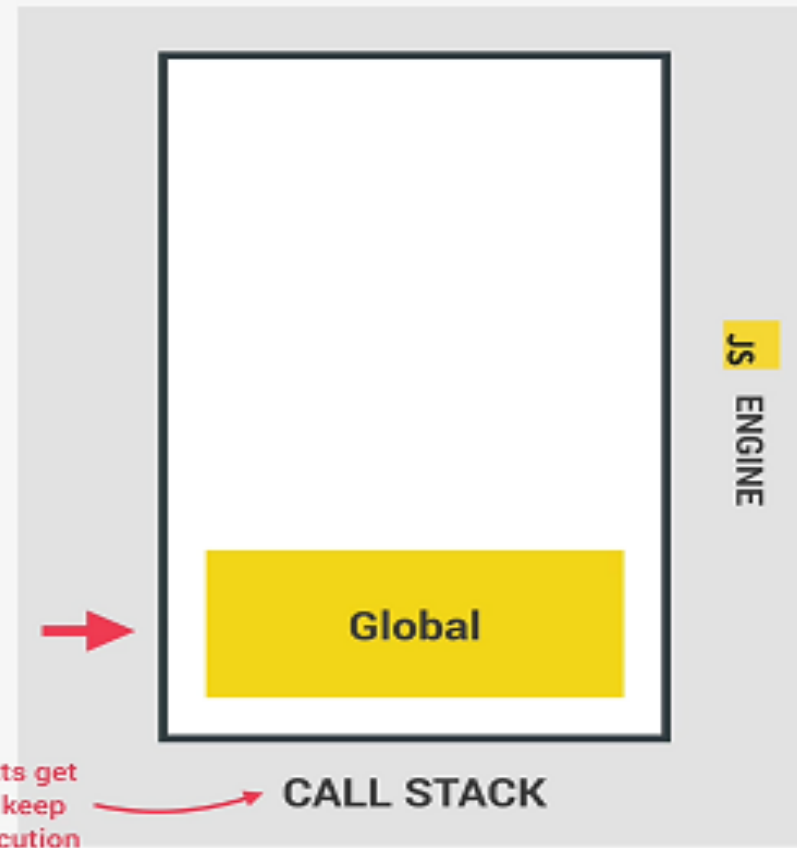
⚡ Compiled code starts execution

```
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**

# EXECUTION CONTEXT : CALL STACK - 4

