JavaScript Fundamentals

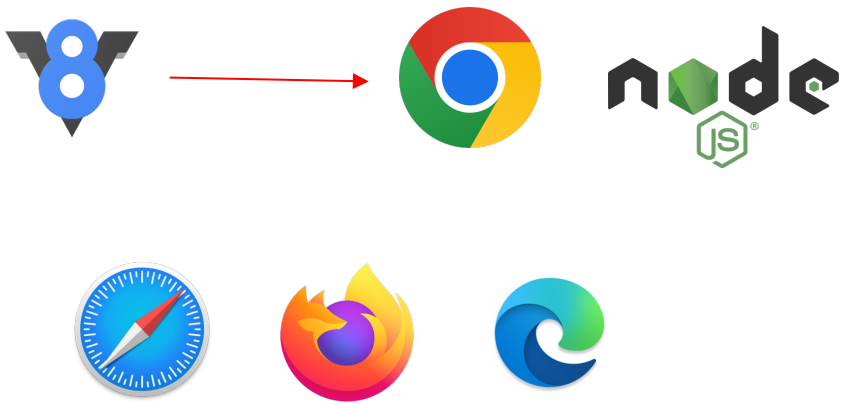
Instructor: Bui Binh Giang
buibinhgiang@vanlanguni.vn

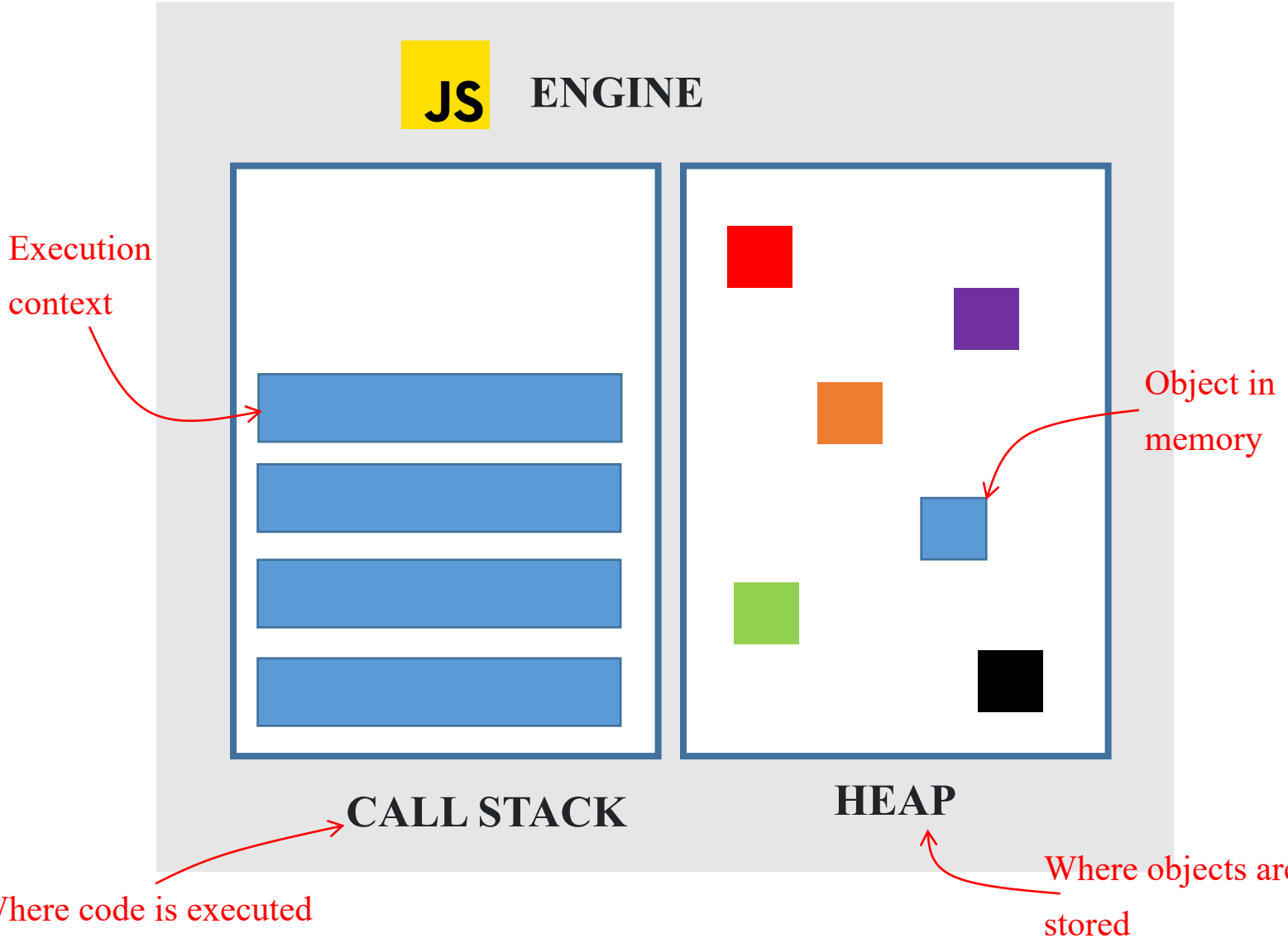
JAVASCRIPT RUNTIME

JavaScript engine

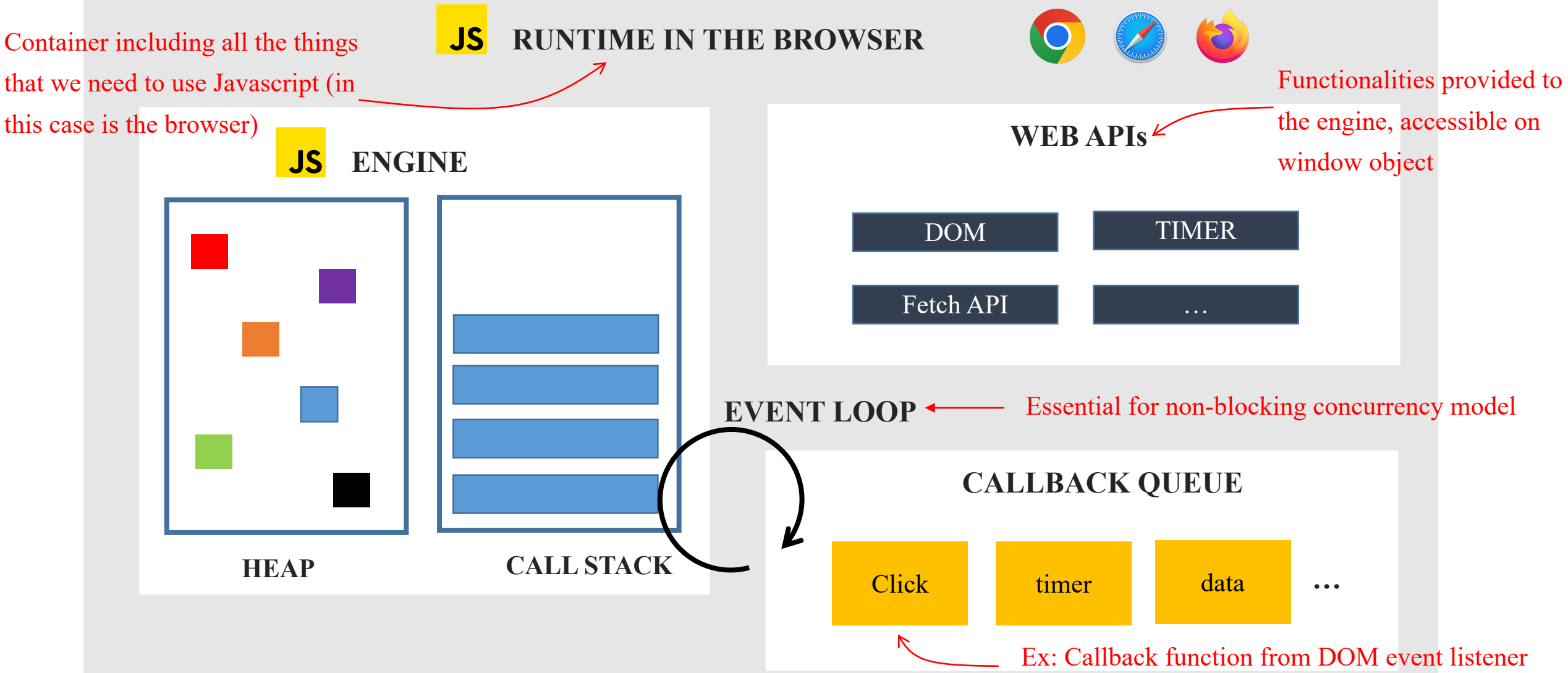
JS engine is a program that executes Javascript code.

Every browser has its own JS engine.



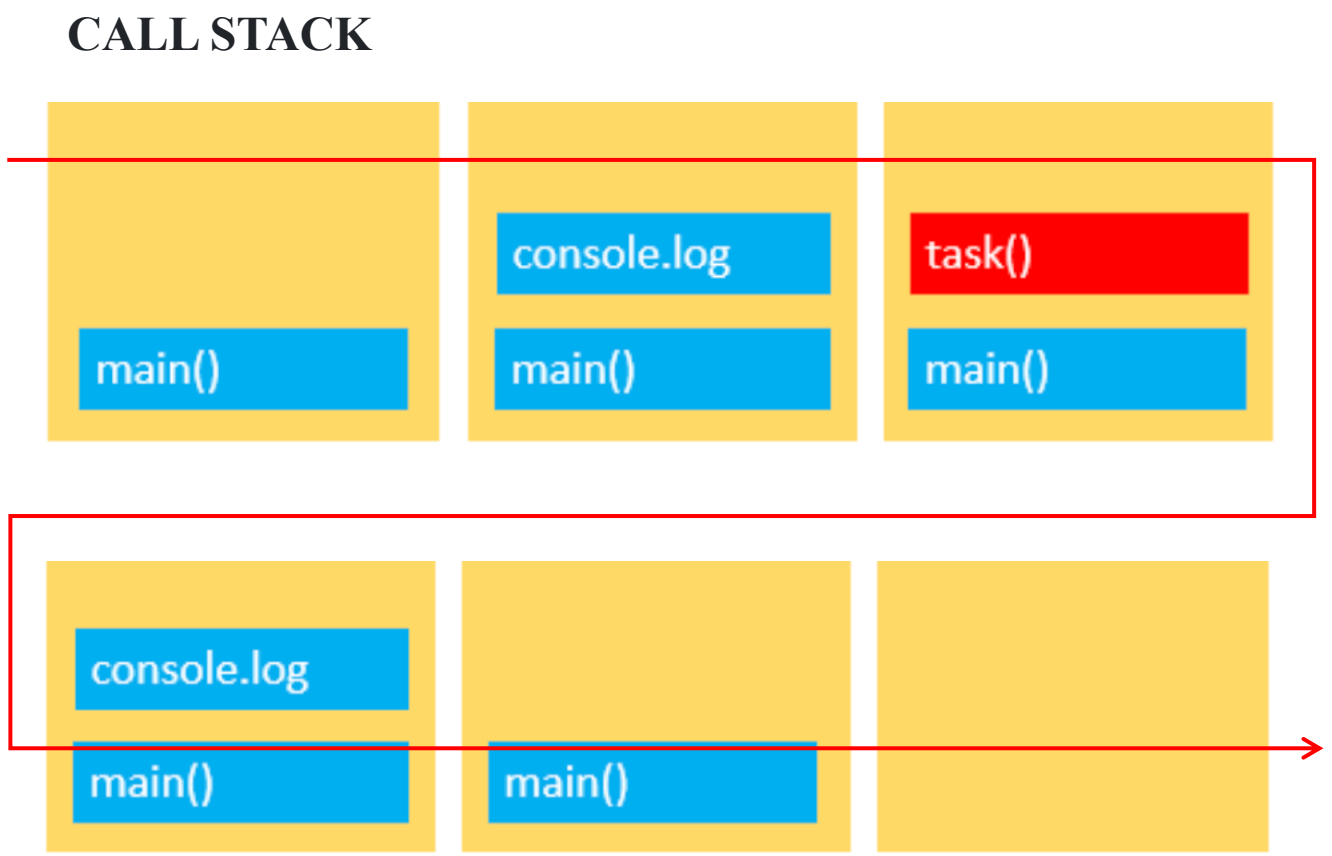


JavaScript Runtime



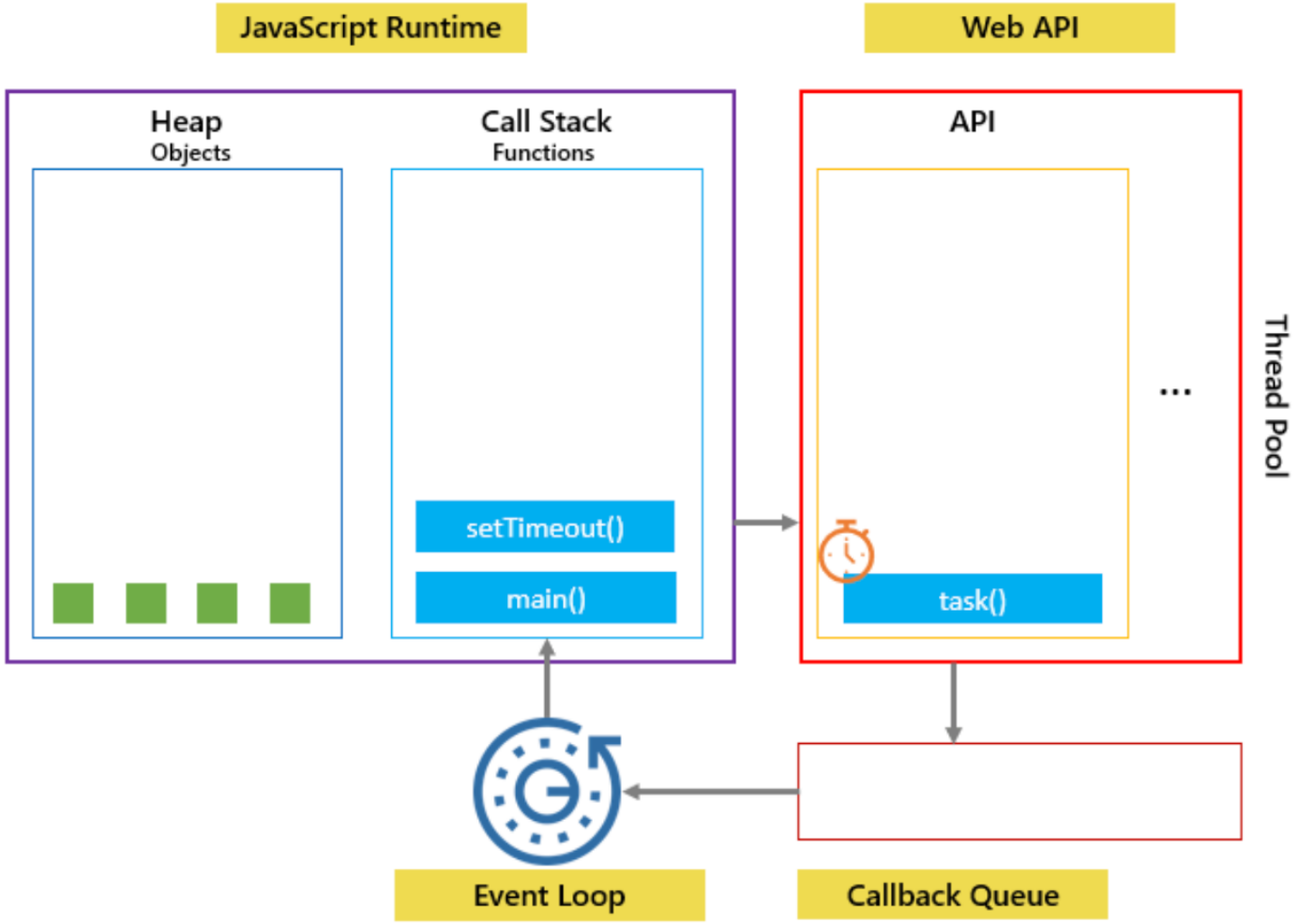
JavaScript Event Loop

```
function task() {
    console.log('Downloading a file');
    // emulate time consuming task
    let n = 3000000000;
    while (n > 0){
        n--;
    }
    console.log('download Done!!!');
}

console.log('Start script...');
task();
console.log('Done!');
```

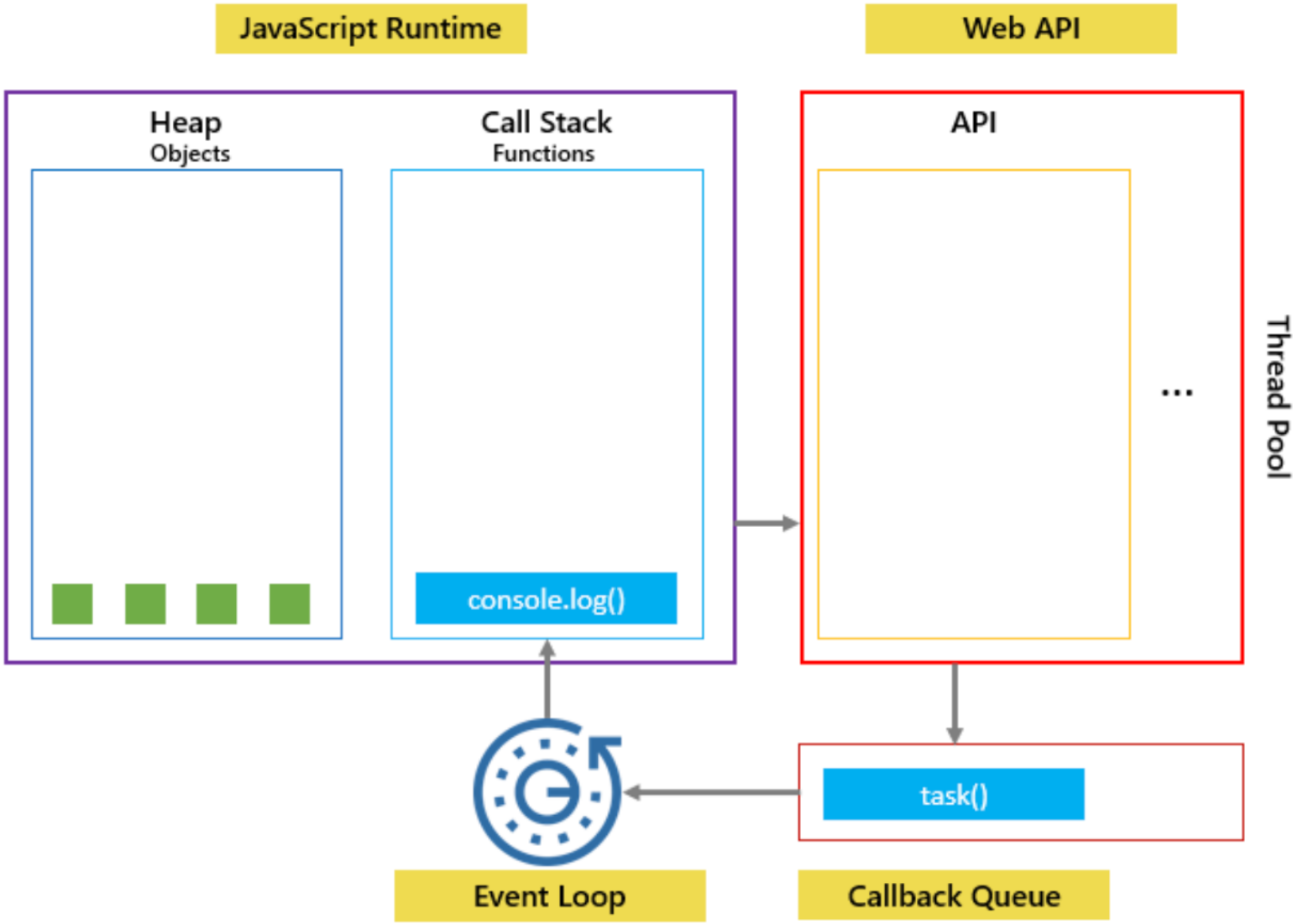


JavaScript Event Loop

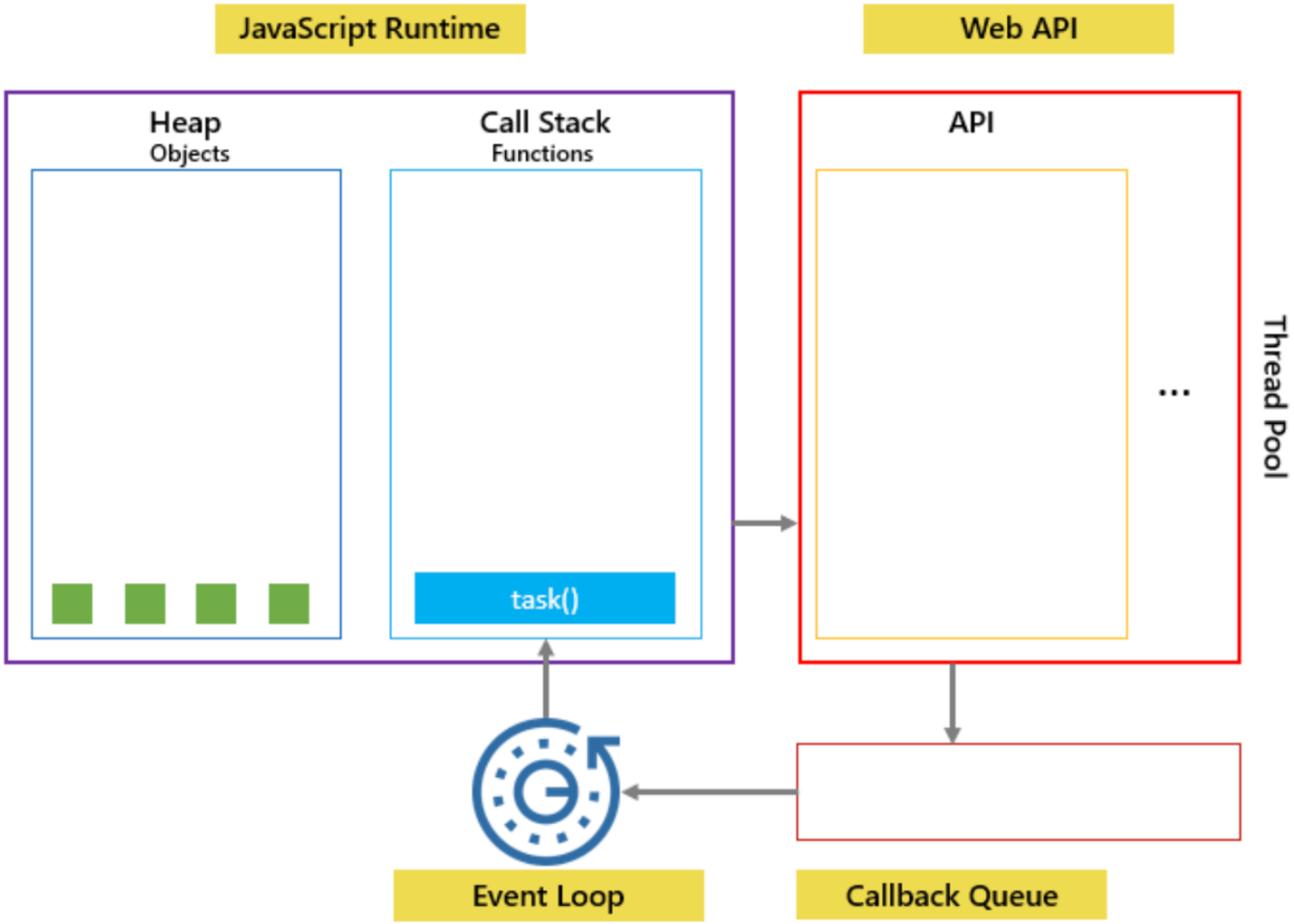
```
function task() {
    console.log('Downloading a file');
    // emulate time consuming task
    let n = 3000000000;
    while (n > 0){
        n--;
    }
    console.log('download Done!!!');
}

console.log('Start script...');
setTimeout( handler: () => {
    task();
}, timeout: 1000);
console.log('Done!');
```



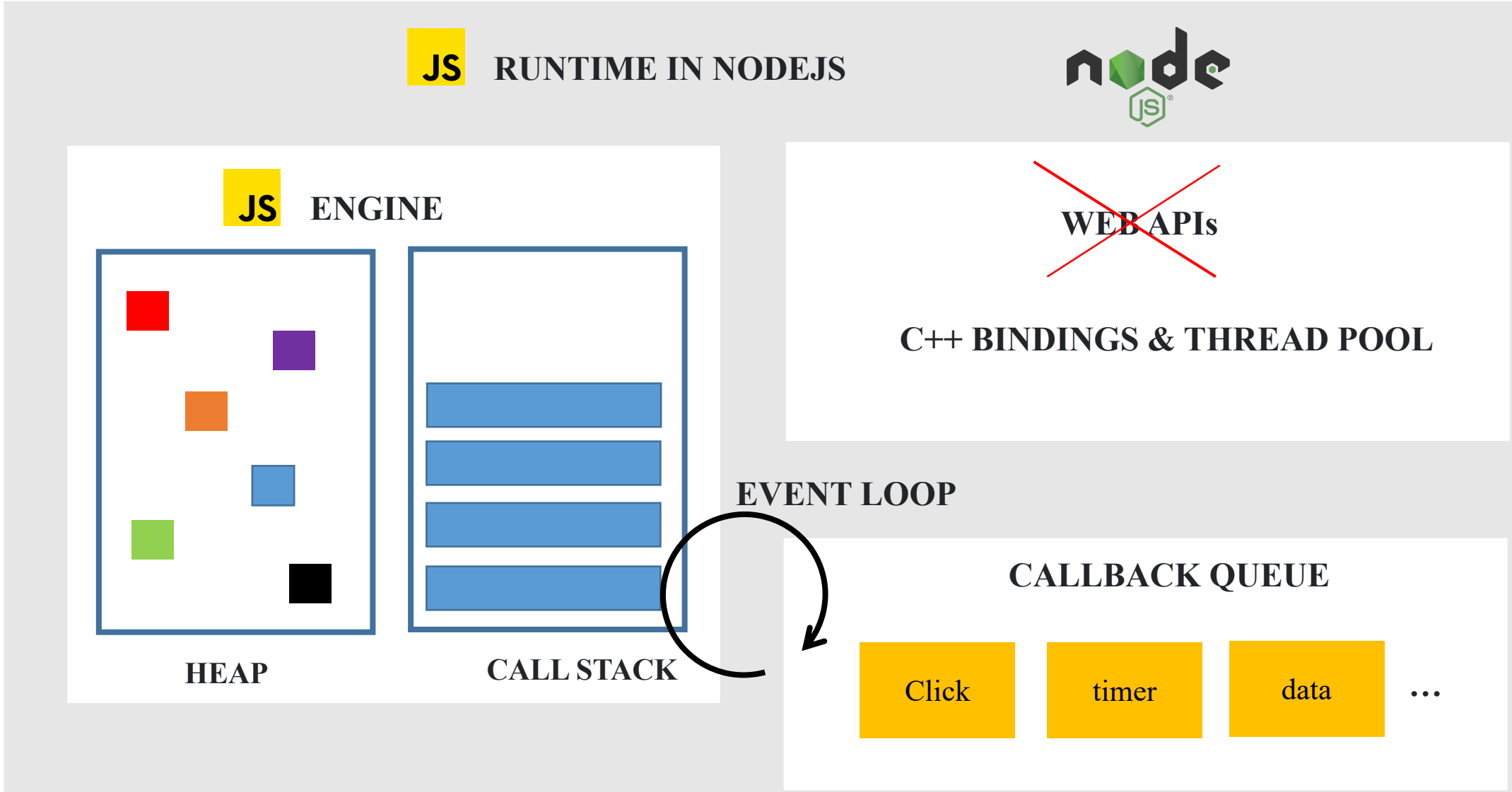
JavaScript Event Loop



JavaScript Event Loop



JavaScript Runtime



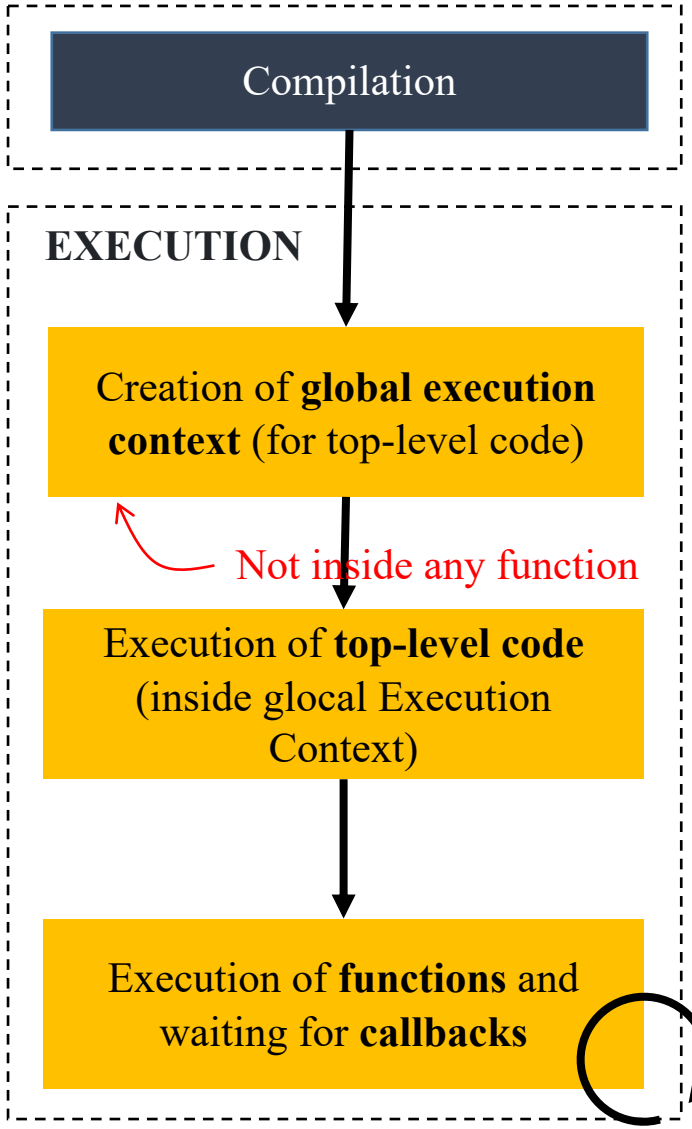
Since we don't have browser, so we don't have web apis

JAVASCRIPT EXECUTION CONTEXT

JavaScript Execution context

```
const name = 'Alex';

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

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

function body only executed when called



Execution context is environment in which a piece of Javascript is executed.

- **Global execution context** is default context, created for code that is not inside any function (**top-level code**)
- For each **function** call, a **new execution context** is created

→ **All execution context together make the call stack**

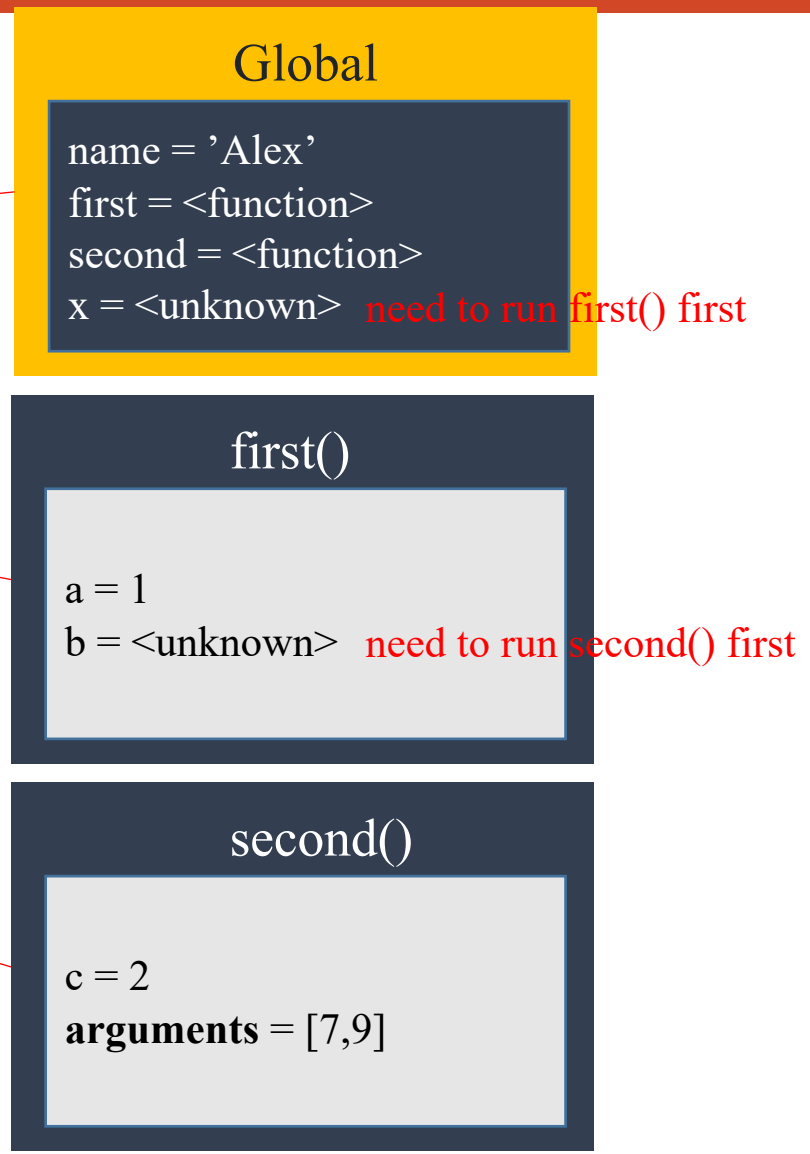
JavaScript Execution context

What's inside execution context?

- Variable Environment
  - let, const and var declarations
  - Functions
  - arguments object
- Scope chain (references to variables outside of the current function)
- **this** keyword

Generated during "creation phase", right before execution

```
const name = 'Alex';

const first = function() {
  let a = 1;
  const b = second( x: 7, y: 9);
  a = a + b;
  return a;
}

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

const x = first();
```



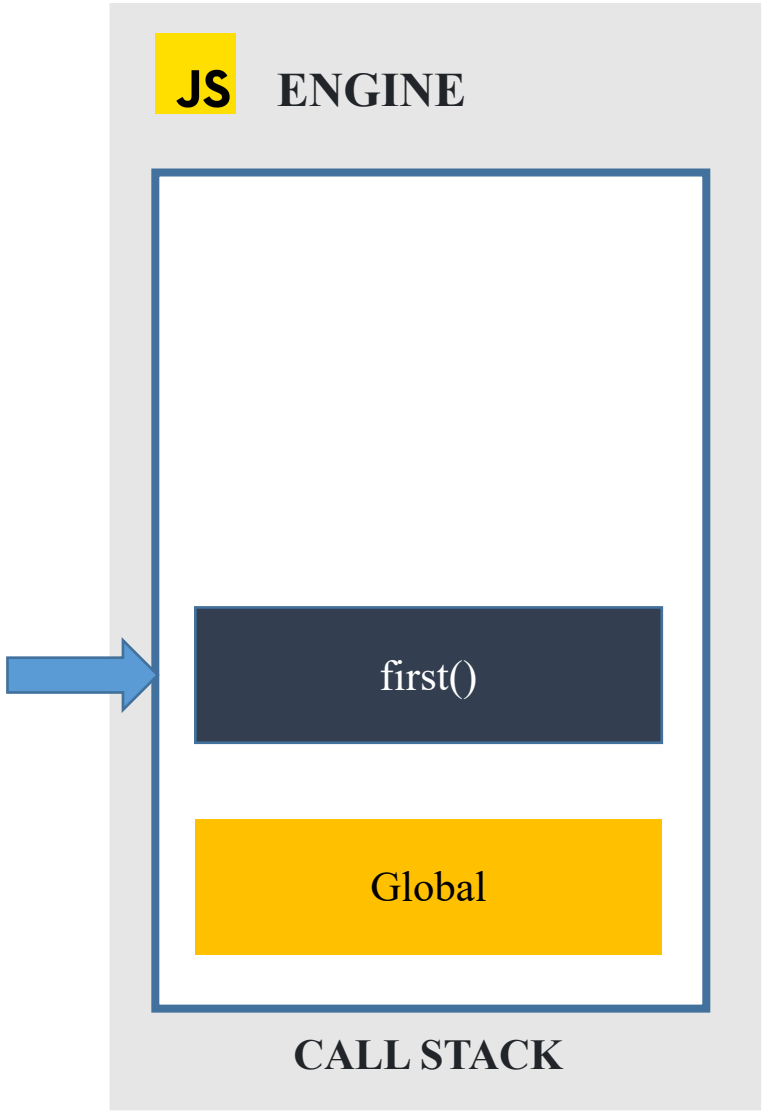
JavaScript Call Stack

```
const name = 'Alex';

const first = function() {
  let a = 1;
  const b = second( x: 7, y: 9);
  a = a + b;
  return a;
}

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

const x = first();
```

Call stack is place where execution contexts get stacked on top of each other, to keep track of where we are in the execution

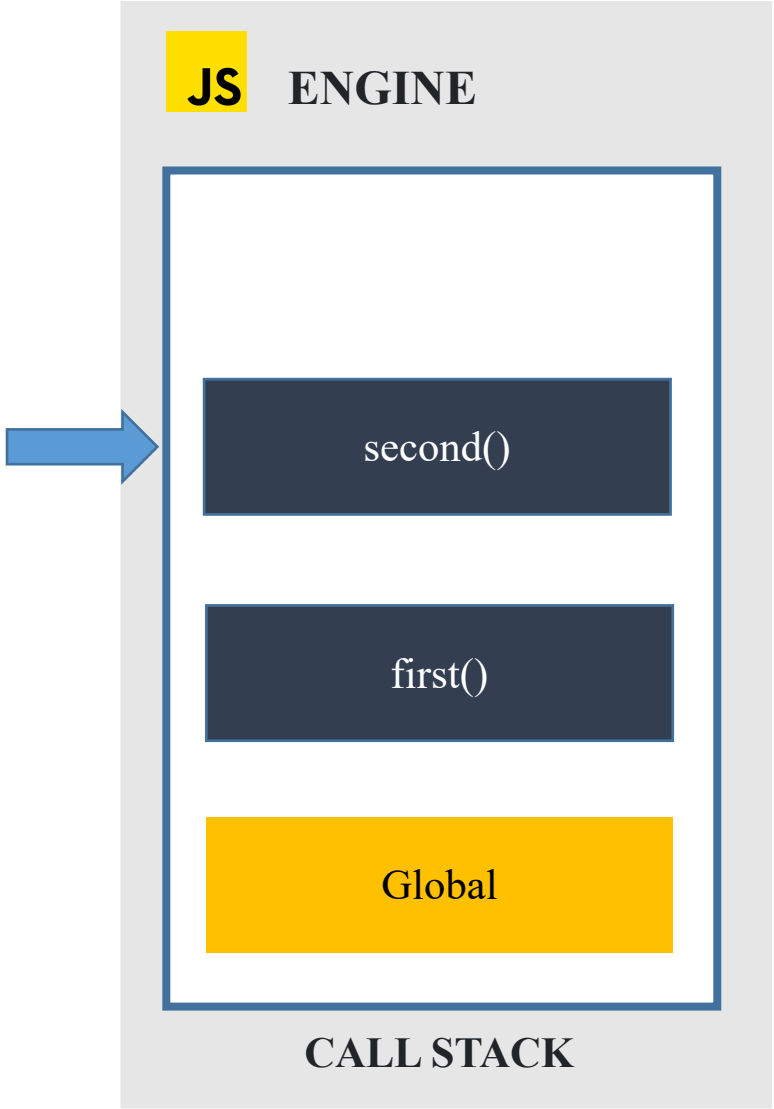
JavaScript Call Stack

```
const name = 'Alex';

const first = function() {
  let a = 1;
  const b = second( x: 7, y: 9);
  a = a + b;
  return a;
}

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

const x = first();
```



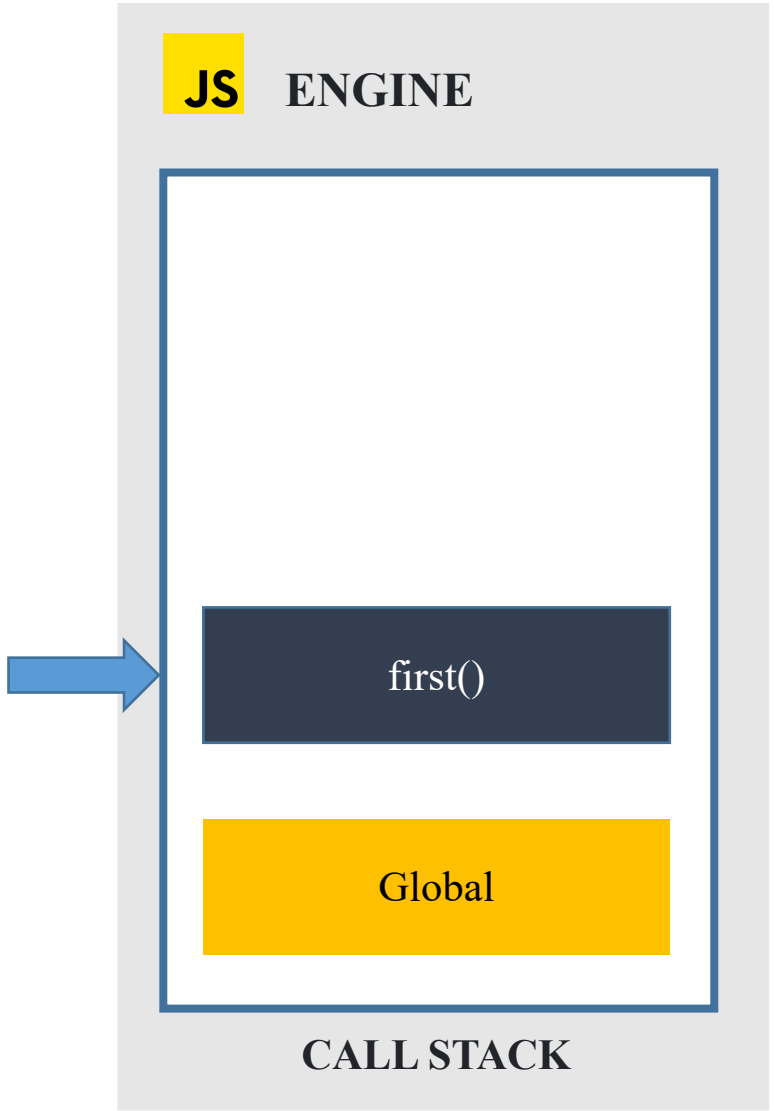
JavaScript Call Stack

```
const name = 'Alex';

const first = function() {
  let a = 1;
  const b = second( x: 7, y: 9);
  a = a + b;
  return a;
}

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

const x = first();
```



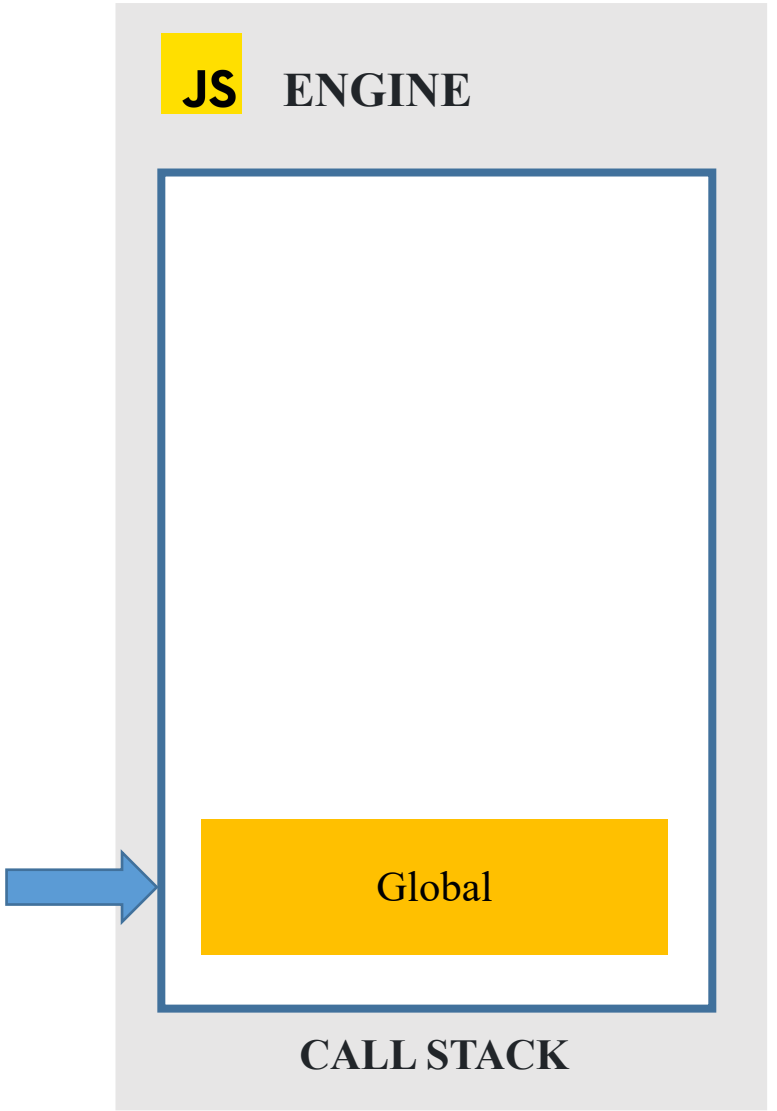
JavaScript Call Stack

```
const name = 'Alex';

const first = function() {
  let a = 1;
  const b = second( x: 7, y: 9);
  a = a + b;
  return a;
}

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

const x = first();
```



JavaScript Call Stack

```
const name = 'Alex';

const first = function() {
  let a = 1;
  const b = second( x: 7, y: 9);
  a = a + b;
  return a;
}

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

const x = first();
```



JAVASCRIPT SCOPE AND THE SCOPE CHAIN

JavaScript Scope and Scope Chain

- **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**.

JavaScript Scope and Scope Chain

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!

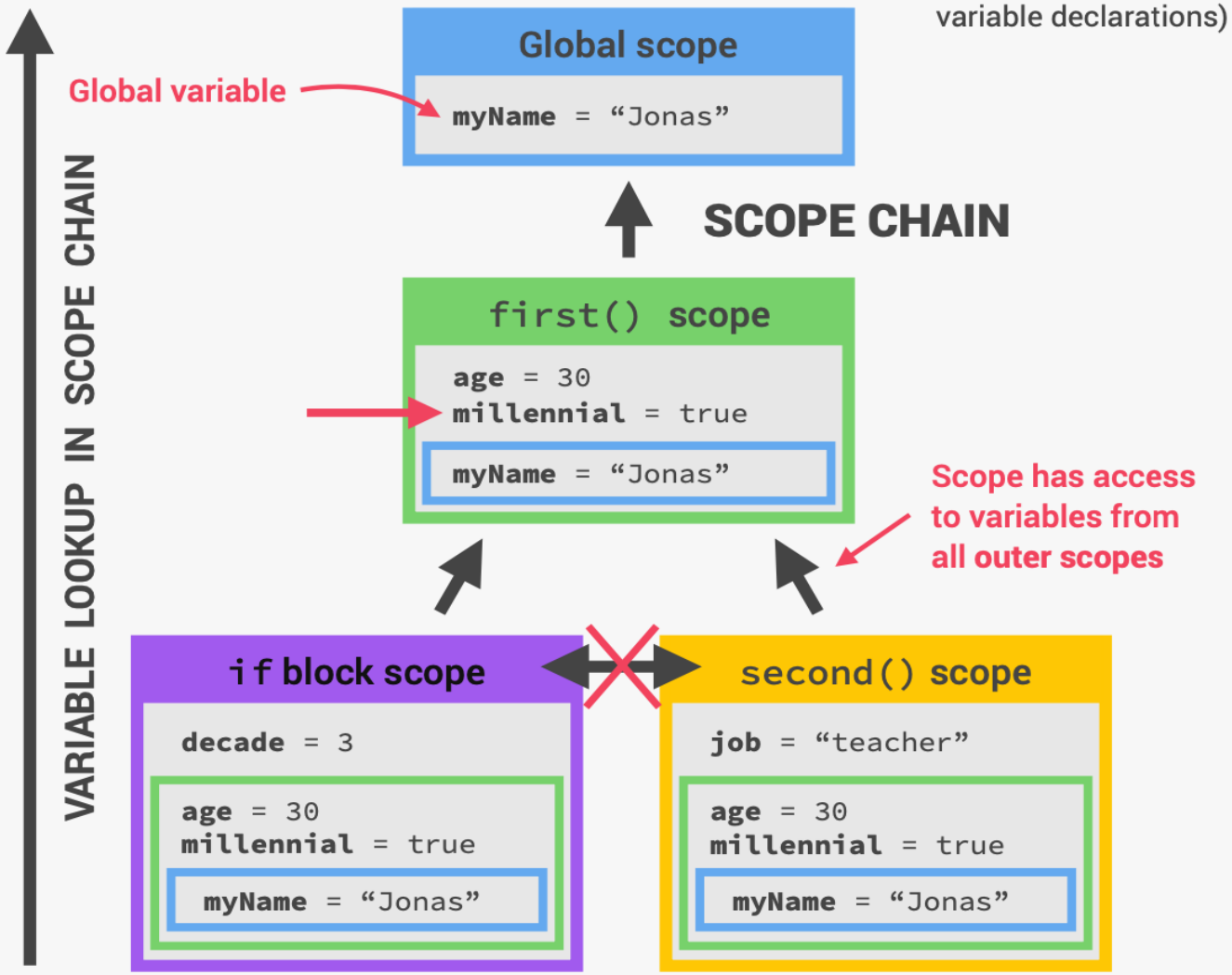
JavaScript Scope and Scope Chain

```
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

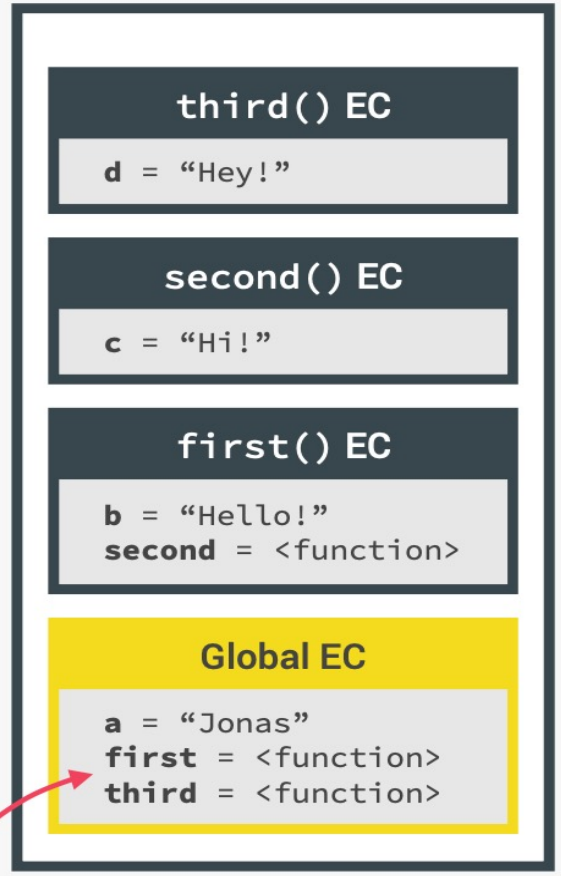


JavaScript Scope and Scope Chain

```
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
}
```

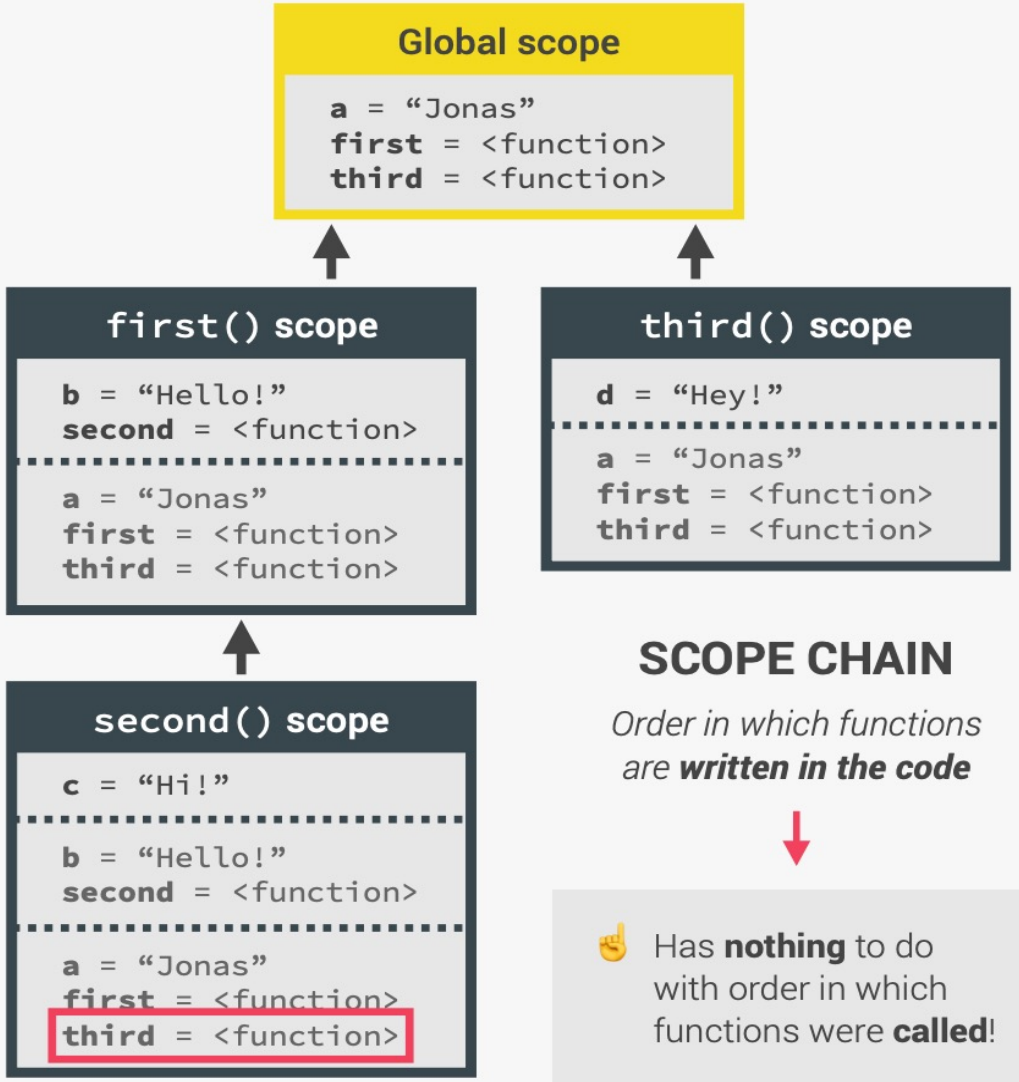
c and b can NOT be found in third() scope!

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**!

JavaScript Scope and Scope Chain

- 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: **global scope**, **function scope**, **and block scope**;
- 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**;
- An outer scope will never, ever have access to the variables of an inner scope ;

HOISTING IN JAVASCRIPT

Hoisting In JavaScript

- **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**

HOISTED		INITIAL VALUE
Function declarations	YES	Actual function
var variables	YES	undefined
let and const variables	NO	<uninitialized>, TDZ

Temporal Dead Zone

```
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;**
- 👉 `var` hoisting is just a byproduct.

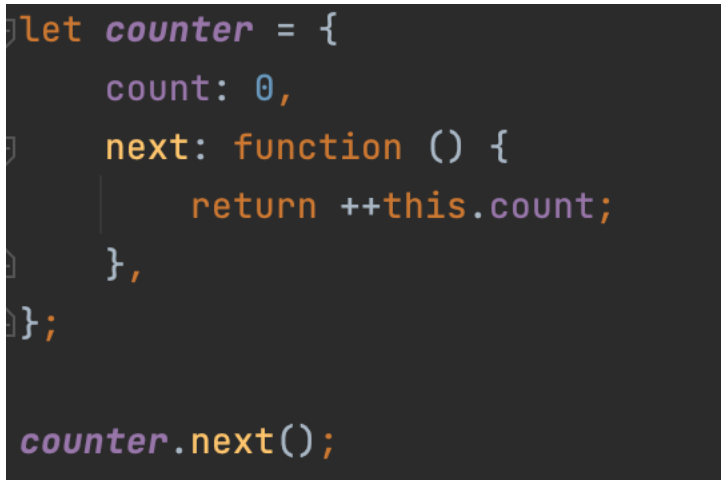
WHY TDZ?

- 👉 **Makes it easier to avoid and catch errors:** accessing variables before declaration is bad practice and should be avoided;
- 👉 **Makes `const` variables actually work**

JAVASCRIPT “THIS” KEYWORD

JavaScript **this** Keyword

The **this** references the object that is currently calling the function.



JavaScript **this** Keyword – Global context

In the **global context**, the **this** references the **global object**, which is the **window** object on the web browser or **global** object on Node.js.

```
console.log(this === window) //true
this.color = 'red';
console.log(window.color); //'red'
```

JavaScript **this** Keyword – Function context

Simple function invocation:

In the non-strict mode, the this references the **global** object when the function is called

```
function show() {
   console.log(this === window); // true
}

show();
```

Method invocation:

When you call a method of an object, JavaScript sets **this** to the object that **owns** the method

```
let car = {
    brand: 'Honda',
    getBrand: function () {
        return this.brand;
    }
}

console.log(car.getBrand()); // Honda
```

JavaScript **this** Keyword – Function context

Constructor invocation:

```
function Car(brand) {
    this.brand = brand;
}

Car.prototype.getBrand = function () {
    return this.brand;
}

let car = new Car('Honda');
console.log(car.getBrand());
```

JavaScript **this** Keyword – Function context

Indirect Invocation:

The **Function** type has two methods: **call**() and **apply**() . These methods allow you to set the **this** value when calling a function

```
function getBrand(prefix) {
    console.log(prefix + this.brand);
}

let honda = {
    brand: 'Honda'
};
let audi = {
    brand: 'Audi'
};
```

```
getBrand.call(honda, "It's a ");
getBrand.call(audi, "It's an ");
```

```
getBrand.apply(honda, ["It's a "]);
getBrand.apply(audi, ["It's an "]);
```