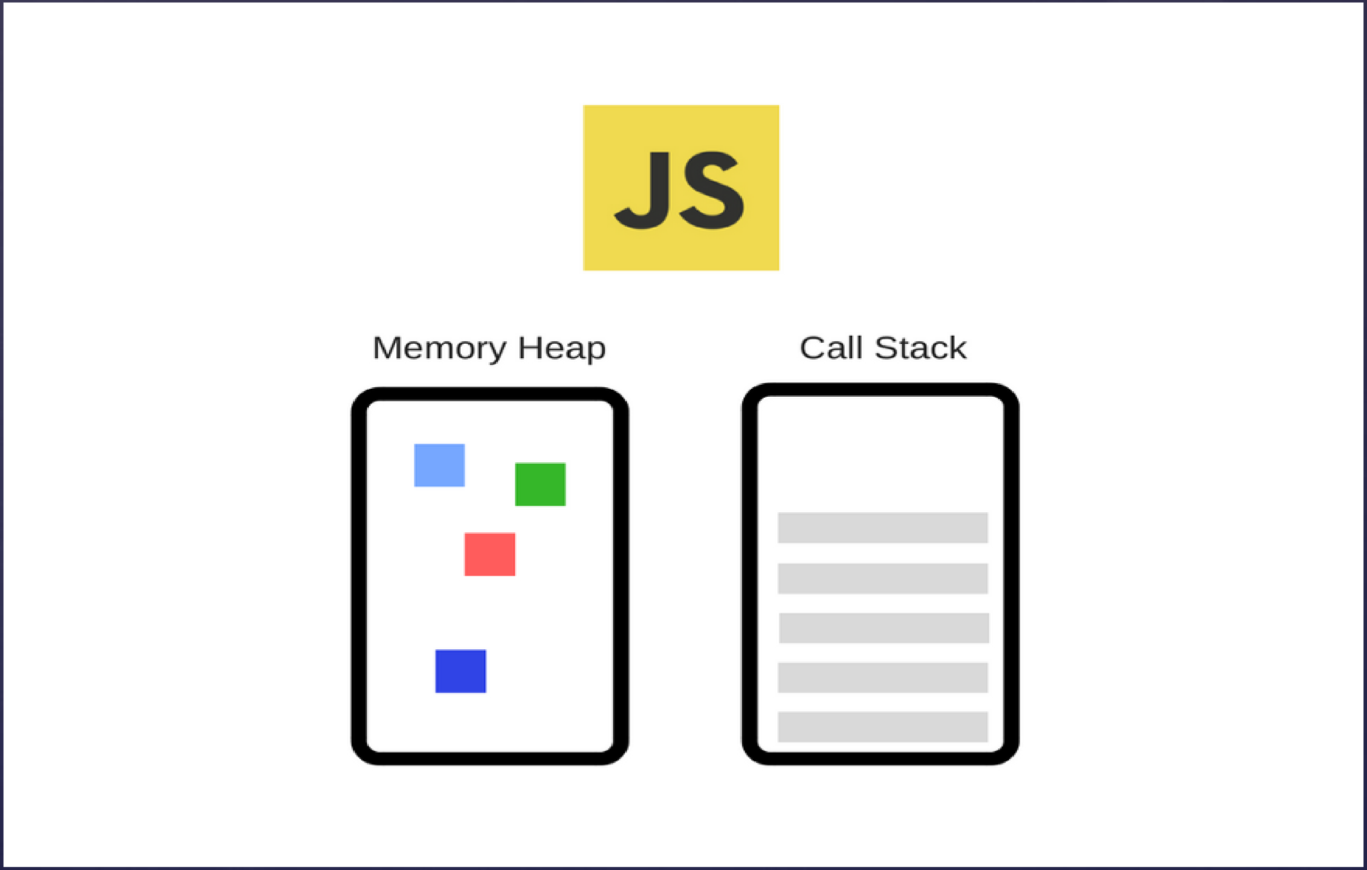
How js works behind the scene ?

- How browsers understand the code (Js engines).
  - Stack Vs Heap
- Compilation Vs Interpretation Vs JIT Compilation
- The whole process to convert code to 0's and 1's
  - Parser ?
  - AST ?
- Js Runtime
  - Engine + Web APIs + CallbackQueue
- Execution context and callstack
- Scope and Scope chain

Js Engine : heap + stack

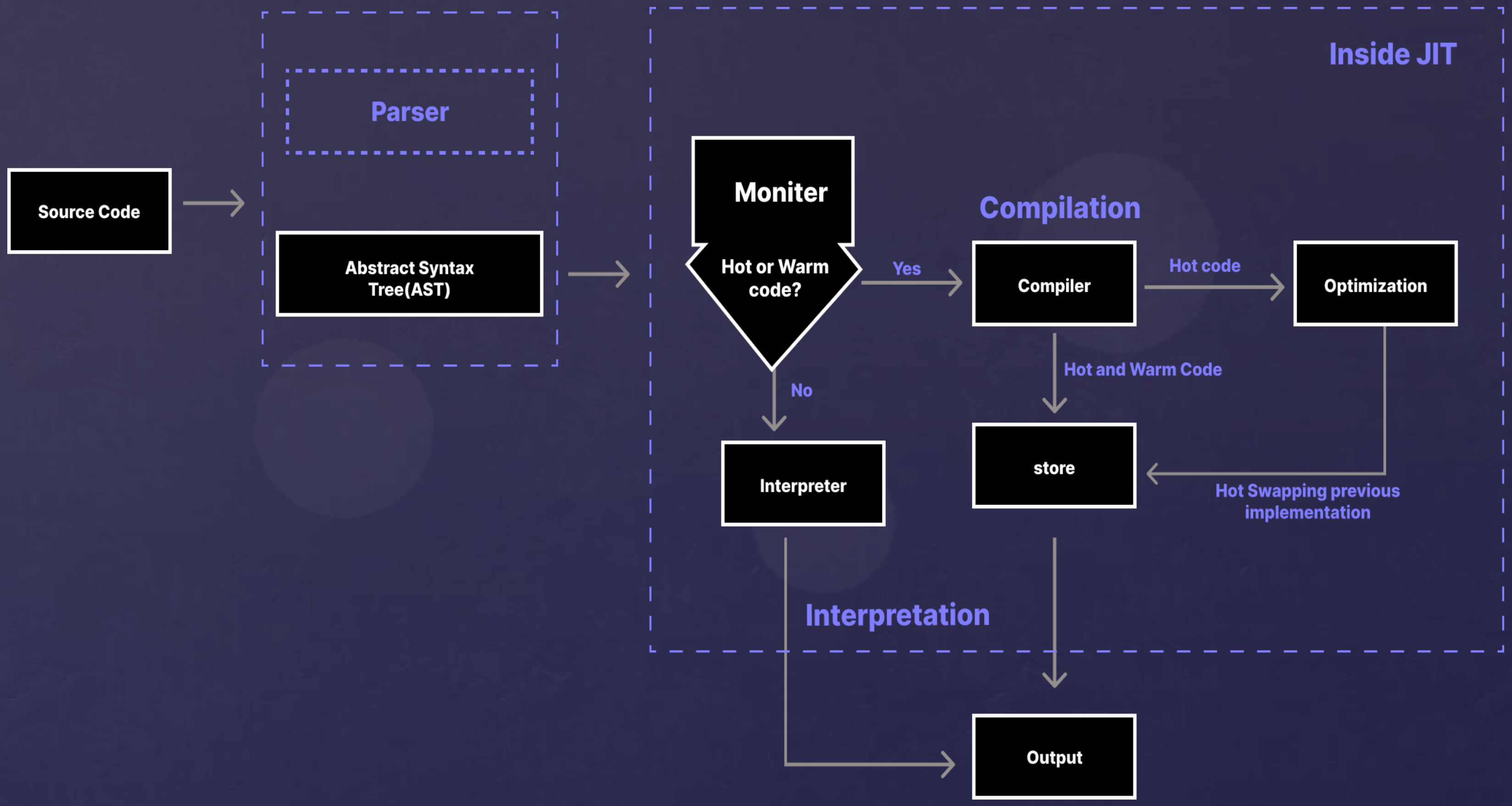
- what’s inside heap and stack ?



Compilation Vs interpretation Vs JIT

Read this :JIT Compilation

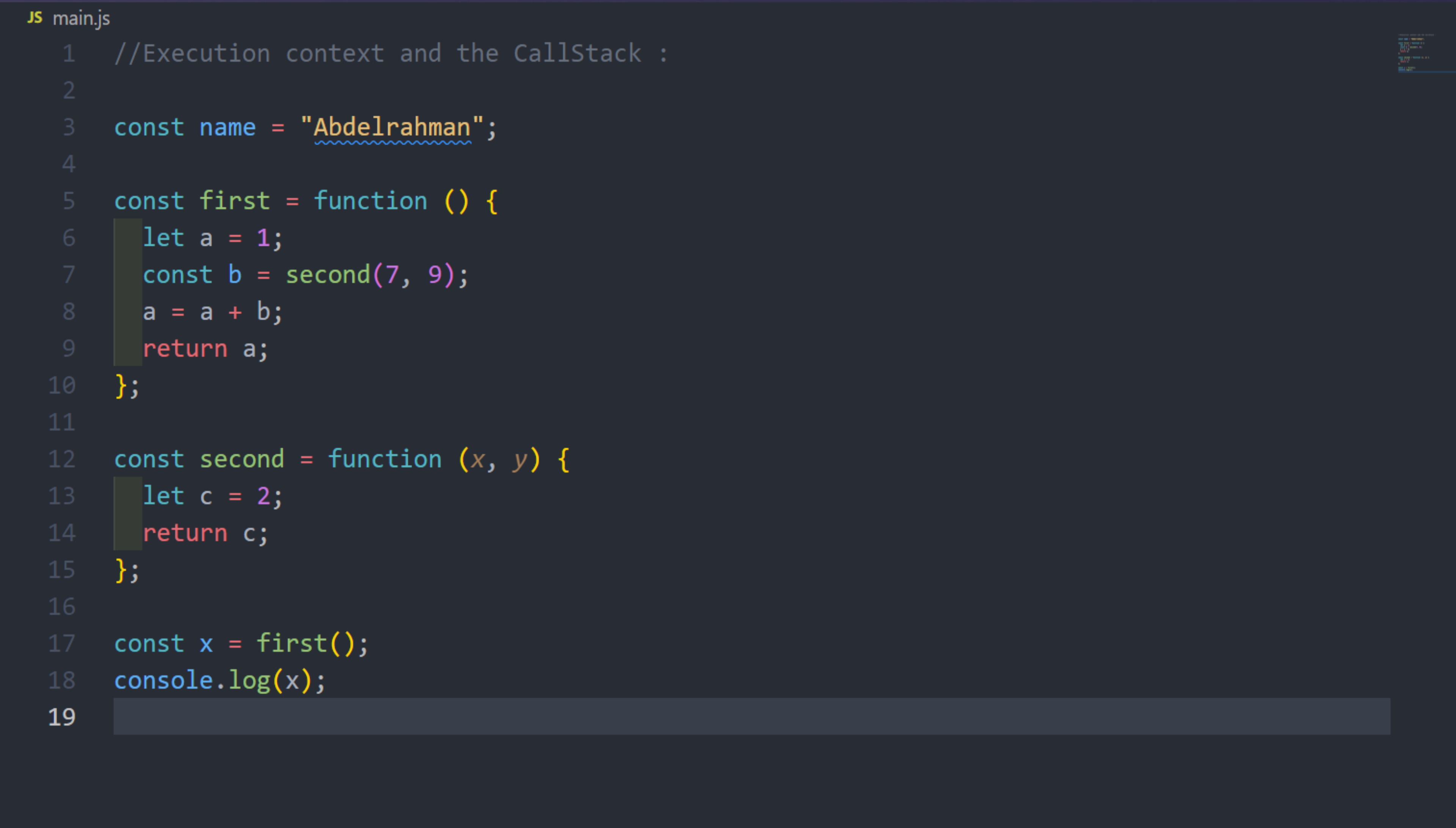
Code → 0's and 1's



Execution context and Callstack:

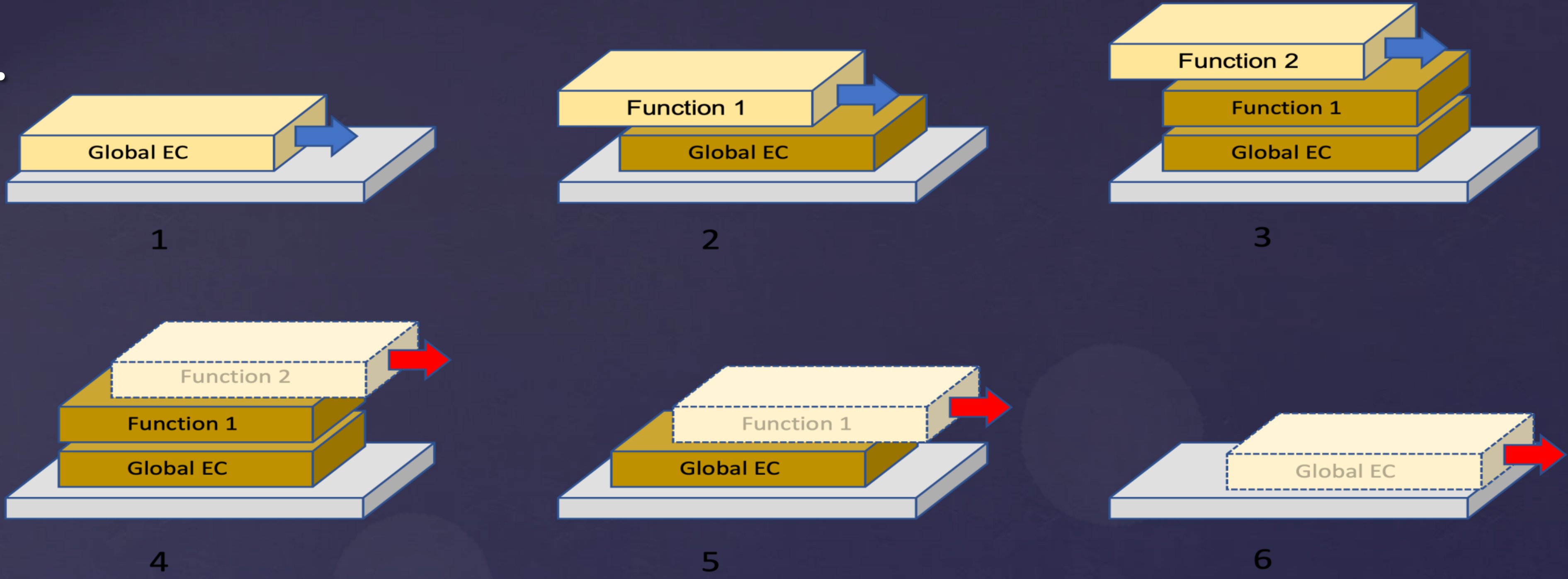
- Everything in js happens in execution context
- EC contains the following :
  - Variable environment
  - Scope chain
  - "This" key word
- Execution context ? Box ? Environment ? Piece of code inside the stack ?
- Global execution context , only one ?!
- One execution context per function ? : For each function call, a new execution context is created
- Variable environment ? Each EC has it's variable environment ! Is it the heap ?

Example of EC and the CallStack



main.js

```javascript
//Execution context and the CallStack :

const name = "Abdelrahman";

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

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

const x = first();
console.log(x);

```



1. We start with an empty stack, and the first thing that happens is we add an execution context (EC) for the global context.

2. Global code calls Function 1, and so we add an EC for that to the stack.

3. Function 1 calls Function 2, and so we add an EC for that to the stack.

4. Now function 2 is done with its job, and so we pop its EC off the stack, returning control to Function 1.

5. Now function 1 is done with its job, and so we pop its EC off the stack, returning control to the global code.

6. Finally the global code is done too, and we pop that off the stack.

Summery for EC ☺

It makes sense now that we say that Java script code runs inside the call stack.

And actually it is more accurate to say that code runs inside of execution contexts that are in the stack.

<u>But the general point is that code runs in the call stack,</u>

Scope Chain

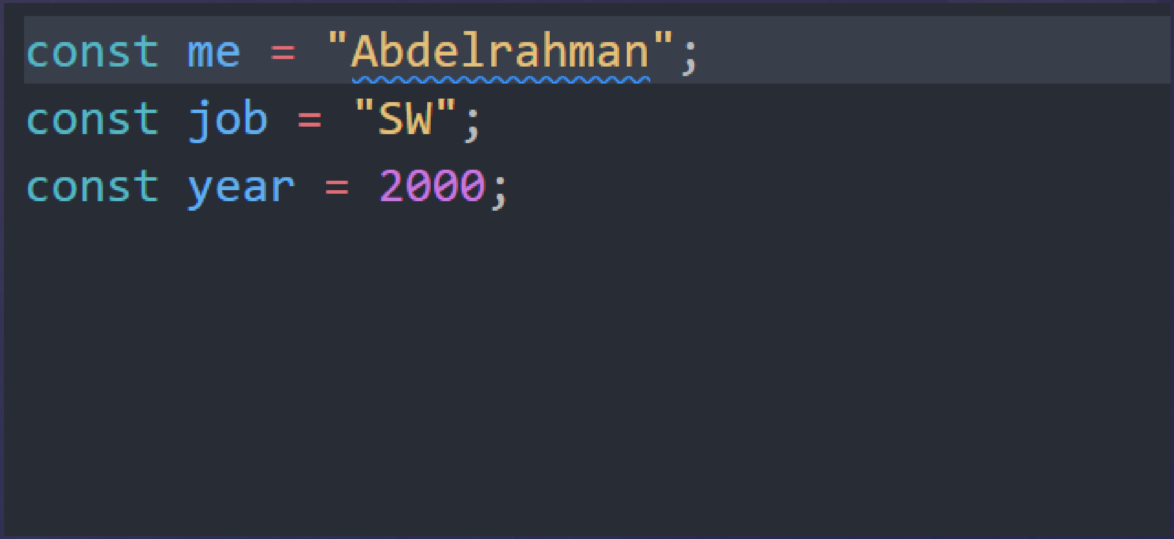
- Scoping: How our program's variables are organized and accessed. "Where do variables live?" or "Where can we access a certain variable, and where not?"
- Scope: Space or environment in which a certain variable is declared (variable environment in case of functions). There is global scope, function scope, and block scope
- Scope of a variable: Region of our code where a certain variable can be accessed

Types of Scopes

Three types :

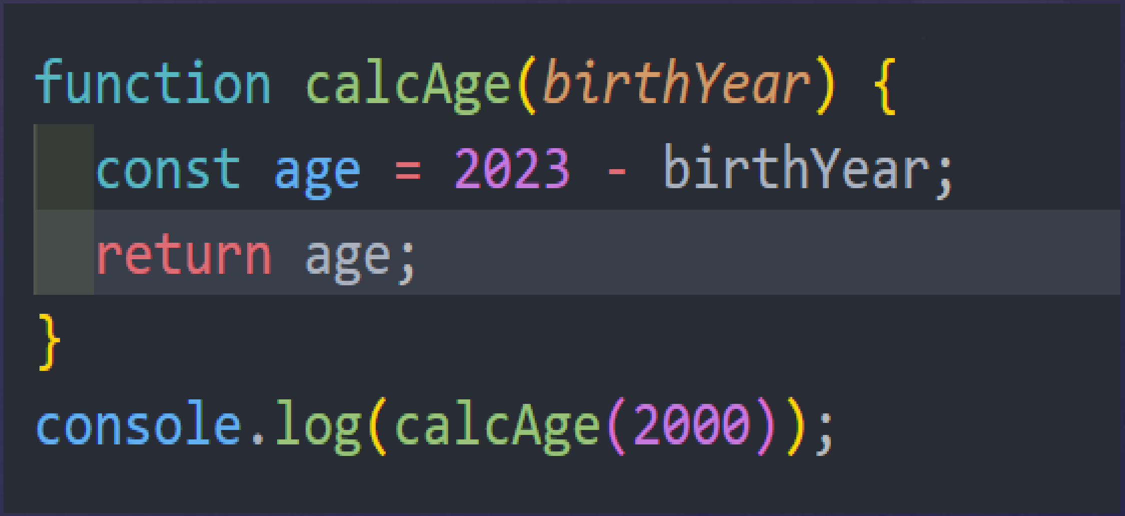
1. Global Scope
2. Function Scope
3. Block Scope

Global Scope



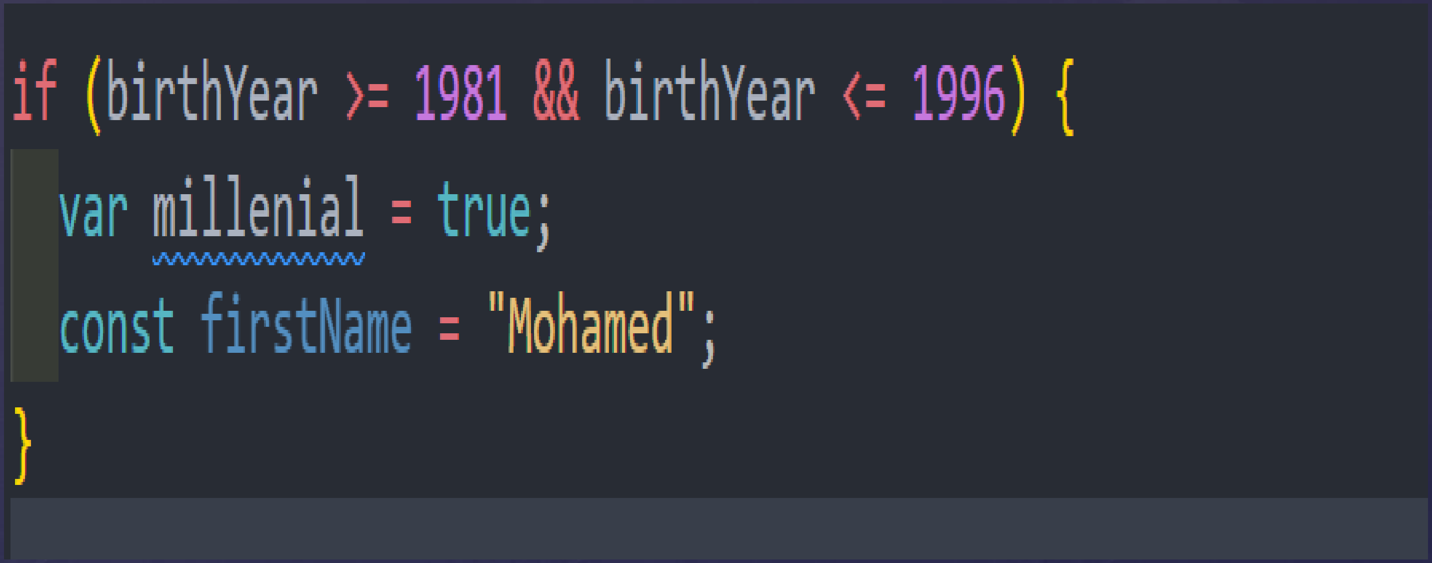
```
const me = "Abdelrahman";
const job = "SW";
const year = 2000;
```

Function Scope



```javascript
function calcAge(birthYear) {
  const age = 2023 - birthYear;
  return age;
}
console.log(calcAge(2000));
```

Block Scope



```
if (birthYear >= 1981 && birthYear <= 1996) {
  var millenial = true;
  const firstName = "Mohamed";
}
```

SUMMARY ☺

- 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: the global scope, scopes defined by functions, and scopes defined by blocks
- Only let and const variables are block-scoped. Variables declared with var end up in the closest function scope
- 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
- The scope chain is a one-way street: a scope will never, ever have access to the variables of an inner scope

Hoisting ?

Resources and Tasks

Watch these videos

- How JavaScript Code is executed?
- Execution Context
- Scope Chain
- Hoisting   You will demonstrate this topic to me ☺

Read those articles

- JavaScript Engine
- Scope Chain
- Hoisting