Siddharth Singha Roy
Winter 19-20

How JavaScript Works

JavaScript can be hosted in any web browser like Google Chrome, Safari or any web server like Node that accepts JavaScript as input.

The host where JavaScript is hosted has some kind of an engine that takes our code and executes it.

JavaScript engine is a program that executes JavaScript codes. Google has its own engine known as V8 engine that can be used to execute JS codes. Other engines are SpiderMonkey, Tamarin etc.

These engines are responsible for passing our code through a

1. Parser to check if the syntax is correct
2. This will produce an Abstract Syntax Tree
3. Then it will be converted into Machine Code
4. Then the code will run

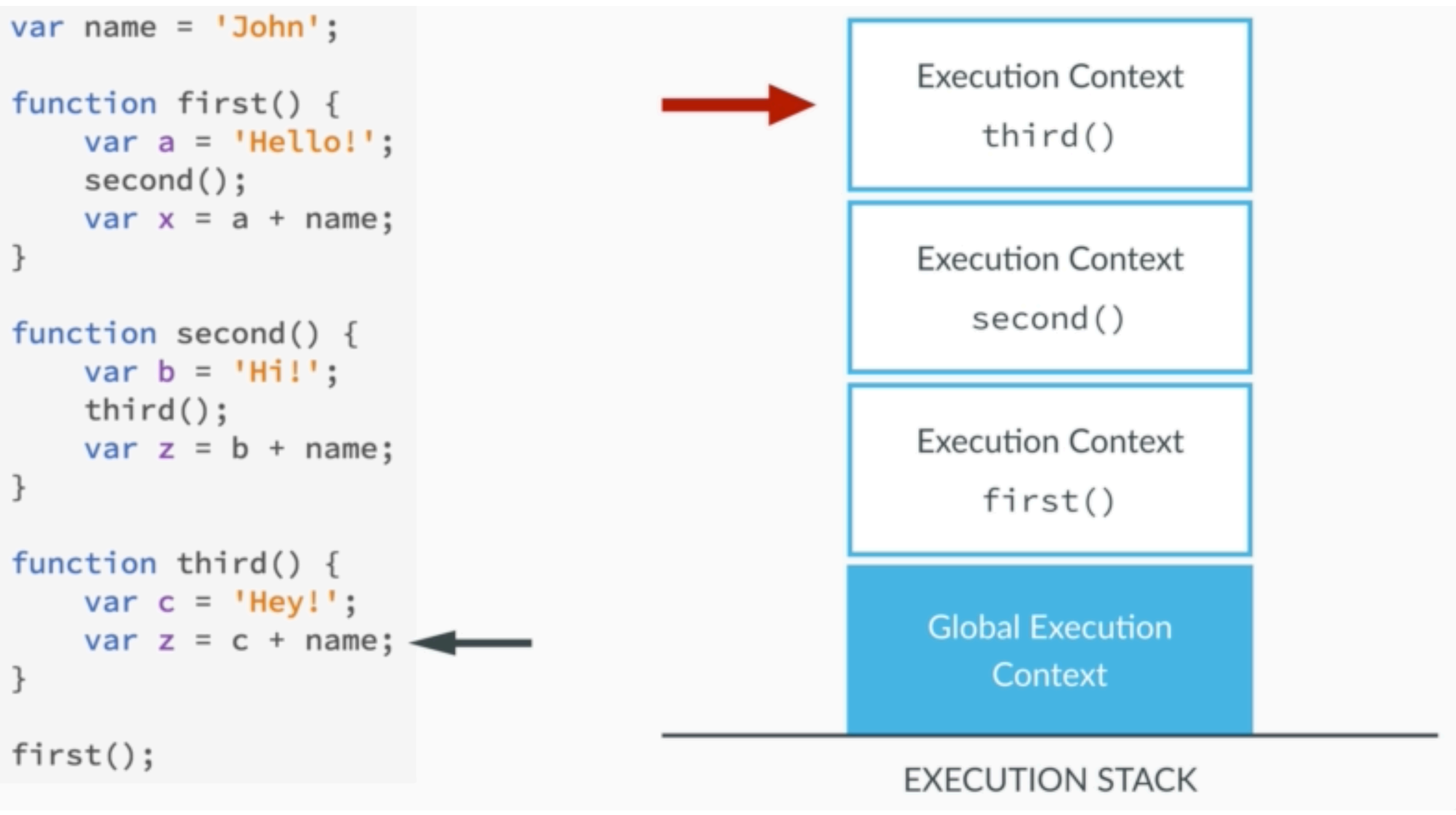
All JavaScript code needs to be run on an environment in which it will be evaluated and executed.

The default execution context is always the global context. In global context all the codes that are not inside functions are executed. The global execution context is associated with an object.

In the case of the browser the global context is the window object. So anything we write in the global context, it gets attached to the window object.

Say that we have declared a variable last_name in the global context. We can also access it using window.last_name. (Only in the browser)

Lastname === window.Lastname. : This is true



```
var name = 'John';

function first() {
    var a = 'Hello!';
    second();
    var x = a + name;
}

function second() {
    var b = 'Hi!';
    third();
    var z = b + name;
}

function third() {
    var c = 'Hey!';
    var z = c + name;
}

first();
```

Execution Context third()
Execution Context second()
Execution Context first()
Global Execution Context
EXECUTION STACK

Each execution context can be treated as an object. For each function we have a new execution context. By default the global execution context in the case of browsers is the window object.

EXECUTION OBJECT

Each execution object has 3 parts

1. Variable Object (VO) : It contains function arguments, inner variable declarations and function declarations. In simple words it is the scope of the data (ie. variables, function declarations) related with that object.
2. Scope Chain : Contains the current variable object and the parent variable objects
3. This variable : Contains the current instance.

When a function is called a new execution context is put on the execution stack. Now this is done in 2 phases

1. Creation Phase : Here the the variable object is created, scope chain is created and this variable is assigned
2. Execution Phase : The function instructions are executed sequentially

CREATION PHASE

The first step is to create the argument object which has all the arguments that were passed into the function

The code is scanned for function declarations and creates a property for each function in the variable object. This means that all the functions will be stored in the variable object even before the code starts executing.

The code is then scanned for each variable declaration and a property is created for each variable object. The value of each property is set to undefined.

The last two points are known as hoisting. This means that the functions and variables are available before even the execution starts.

However variable and functions are hoisted in a different manner. Functions are already defined before the execution phase begins whereas variables are always undefined before the execution phase begins. They are only defined in the execution phase.

In simple words, each execution context has an object which stores all the function and variable declarations even before the code is executed.

Hoisting only works for function declarations not function expressions.

```
calcAge(2018); // This works because of code hoisting
// JS already knows the functions that are there in the code
// even before executing the function

function calcAge(year) {
    console.log(2019-year);
}

//calcAge(2018);



// Code Hoisting
console.log(test);  // undefined

test = 6;
console.log(test);

var test = 5;
```

So for code hoisting, JS knows that we will have a test variable but will have the data undefined. In the case of function hoisting, JS knows the function body entirely.

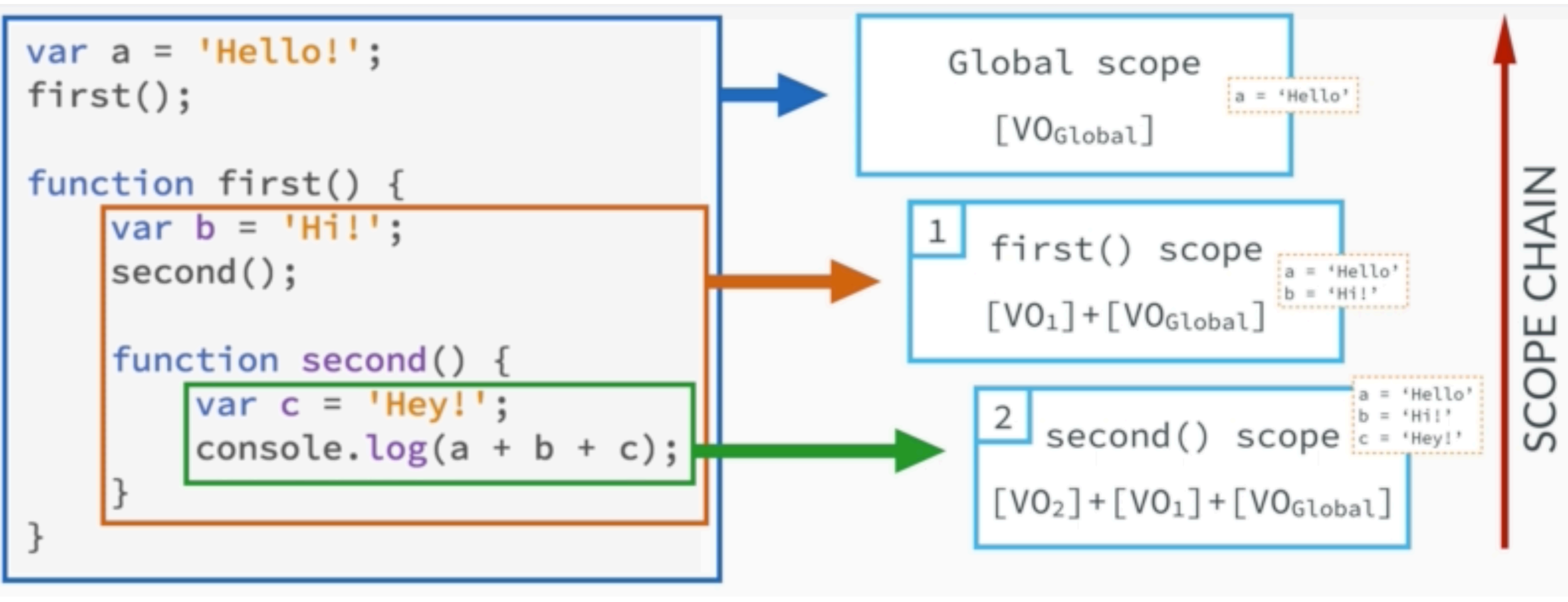
SCOPE

It defines where we can access a variable. Unlike other programming languages a scope can be created only when there is a new function.

JavaScript also supports lexical scoping which means that if we have nested functions the child function gets access to the scope of the parent function. It means that it will get access to the parent variables and functions.

So a child scope will always have access to its parent scope.

But the reverse is never true.



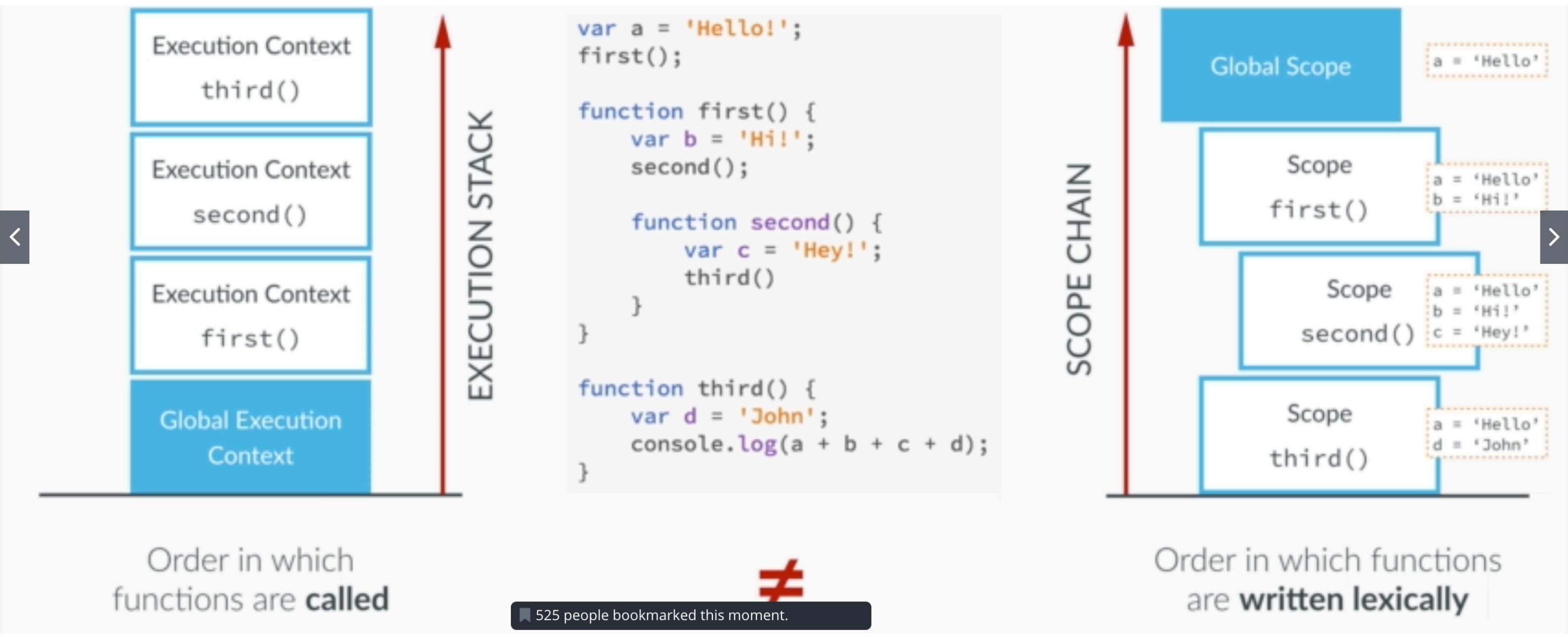
```
var a = 'Hello!';
first();

function first() {
    var b = 'Hi!';
    second();

    function second() {
        var c = 'Hey!';
        console.log(a + b + c);
    }
}
```

second() will first search for 'b' in first() and then in global scope. This is known as scope chain. It moves from bottom to up and never the reverse. The execution stack is the order in which functions are called but scope chain is where the functions are written in our code ie. their lexical position.

In the below example we are able to use first() and third() because of code hoisting.



```
var a = 'Hello!';
first();

function first() {
    var b = 'Hi!';
    second();

    function second() {
        var c = 'Hey!';
        third()
    }
}

function third() {
    var d = 'John';
    console.log(a + b + c + d);
}
```

Execution Context third()
Execution Context second()
Execution Context first()
Global Execution Context
EXECUTION STACK
Order in which functions are **called**
≠
525 people bookmarked this moment.
SCOPE CHAIN
Global Scope
a = 'Hello'
Scope first()
a = 'Hello'
b = 'Hi!'
Scope second()
a = 'Hello'
b = 'Hi!'
c = 'Hey!'
Scope third()
a = 'Hello'
d = 'John'
Order in which functions are **written lexically**

This Variable

This variable is what every execution context gets and is stored in the execution context object.

In a regular function call the this keyword points to the global object.

In a method call, it points to the object that is calling the method.

The this keyword is not actually assigned a value until the function where it is defined is executed. Simply speaking, until the execution is done for a particular statement, this does not get a value.

Since this is associated with the execution context, it is created only when a functions is called ie. execution phase.

In Regular function call, this keyword always points to the global context object. [ALWAYS]

Even if a method calls a function, it will still refer to the global context object.

In Object method call, this keyword points to the object that called the method.

<u>METHOD BORROWING</u>

Suppose John object has a method calcAge(). We want to have the same method for Will object. To do this we can use,

will.calcAge = john.calcAge;

Now if we call will.calcAge(), the this keyword will refer to the will object. This is in accordance with the fact that this keyword is assigned value only after execution. If it was before then this would refer to the initial object ie. John.