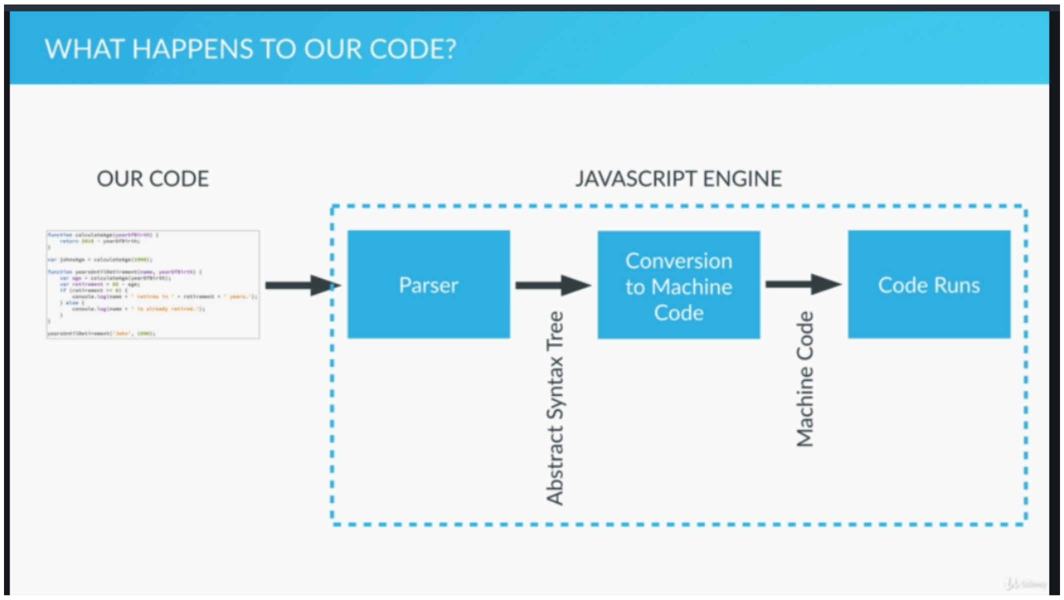
--- 1 ---

HOW OUR CODE IS EXECUTED: JAVASCRIPT PARSERS AND ENGINES

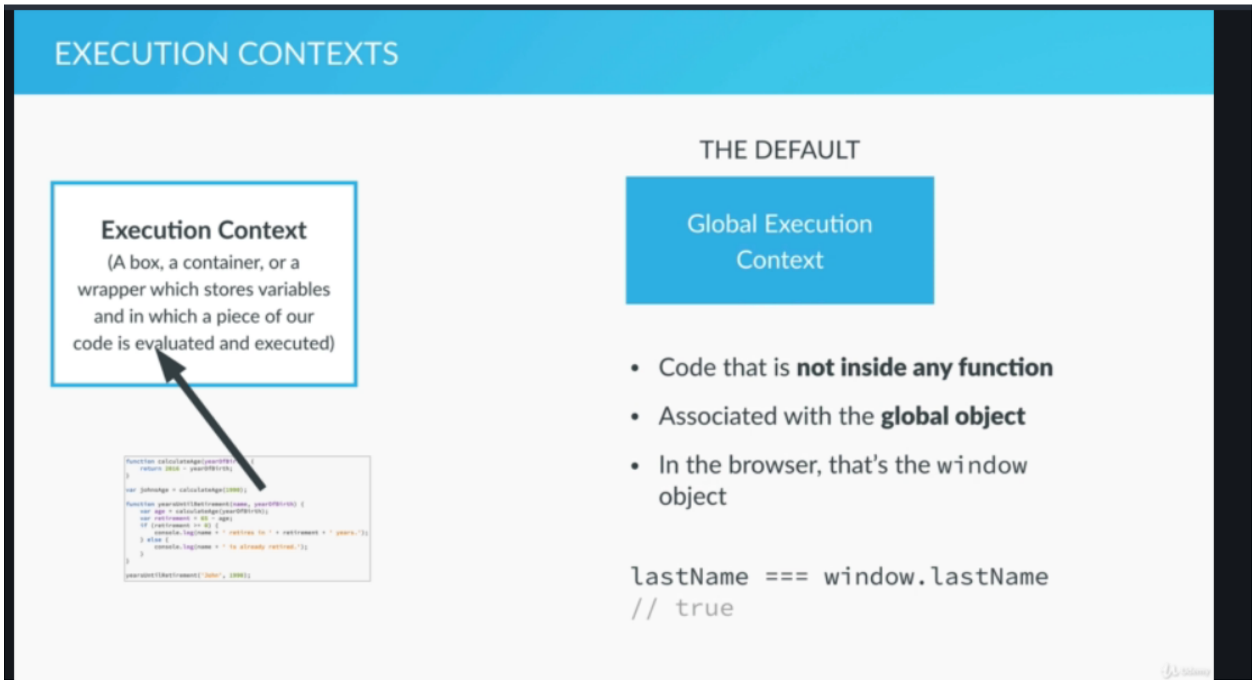
- ➔ JS is hosted inside an environment and that is most typically a browser such as Google Chrome, Firefox, and Safari etc.
- ➔ There can be other hosts, such as Node.JS Web server or even some JS applications that accept JS code -input.
  - ➔ We will focus on browser in this course.
  - ➔ Host where the JS is hosted has some kind of engine that takes our code and executes it. So in very simple terms, JS engine is a program that executes JS code.
  - ➔ There are many different engines such as Google V8 Engine that is using Google Chrome. But there are others like Reno, Spider Monkey, JS Core and many more.
  - ➔ The first thing that happens inside engine is that our code is parsed by a PARSER, which basically reads our code line by line and checks if the syntax of the code that we gave it is correct.
  - ➔ So Parser knows JS rules and how it has to written in order to be correct, to be valid.
  - ➔ And if we make any mistakes, it gives error and stops the execution.
  - ➔ If everything is correct, then the parser produces a data structure known as Abstract Syntax Tree, which is then translated into machine code.
  - ➔ So this JS code is no longer the code, but the set of instructions that can be executed directly by Computers Processor.
  - ➔ And it's only when our code is converted to Machine code that it actually runs and does its works.



```
function calculateAge(yearOfBirth) {
    return 2016 - yearOfBirth;
}

var johnsAge = calculateAge(1990);

function yearsUntilRetirement(name, yearOfBirth) {
    var age = calculateAge(yearOfBirth);
    var retirement = 65 - age;
    if (retirement >= 0) {
        console.log(name + ' retires in ' + retirement + ' years.');
    } else {
        console.log(name + ' is already retired.');
    }
}

yearsUntilRetirement('John', 1990);
```

EXECUTION CONTEXTS AND THE EXECUTION STACK

- ➔ EXECUTION CONTEXTS: All JS code needs to run in an environment, and this environment is called Execution Contexts.
- ➔ You can imagine an execution context like a box, or a container which stores variables and in which a piece of our code is evaluated and executed.
- ➔ The Default Execution Context is always the Global Execution Context.
- ➔ In Global Execution Context, the code which is not inside any function is executed.
- ➔ It's very important, that Global Execution Context is for variables and functions that are not inside of any function.
- ➔ You can also think of an Execution Context as an Object. So the Global Execution Context is associated with the Global Object which in case of the browser is the window object.
- ➔ So everything that we declare in Global Context automatically gets attached to the window object in the browser and it works like this, declaring a variable lastName and window.lastName is the exact same thing.
- ➔ It's like lastName is the property of window object, and as we know properties are just variables attached to object.



EXECUTION CONTEXTS

Execution Context
(A box, a container, or a wrapper which stores variables and in which a piece of our code is evaluated and executed)

```
function calculateAge(yearOfBirth) {
    return 2016 - yearOfBirth;
}

var johnAge = calculateAge(1990);

function yearsUntilRetirement(name, yearOfBirth) {
    var age = calculateAge(yearOfBirth);
    var retirement = 65 - age;
    if (retirement >= 0) {
        console.log(name + ' retires in ' + retirement + ' years.');
    } else {
        console.log(name + ' is already retired.');
    }
}

yearsUntilRetirement('John', 1990);
```

THE DEFAULT

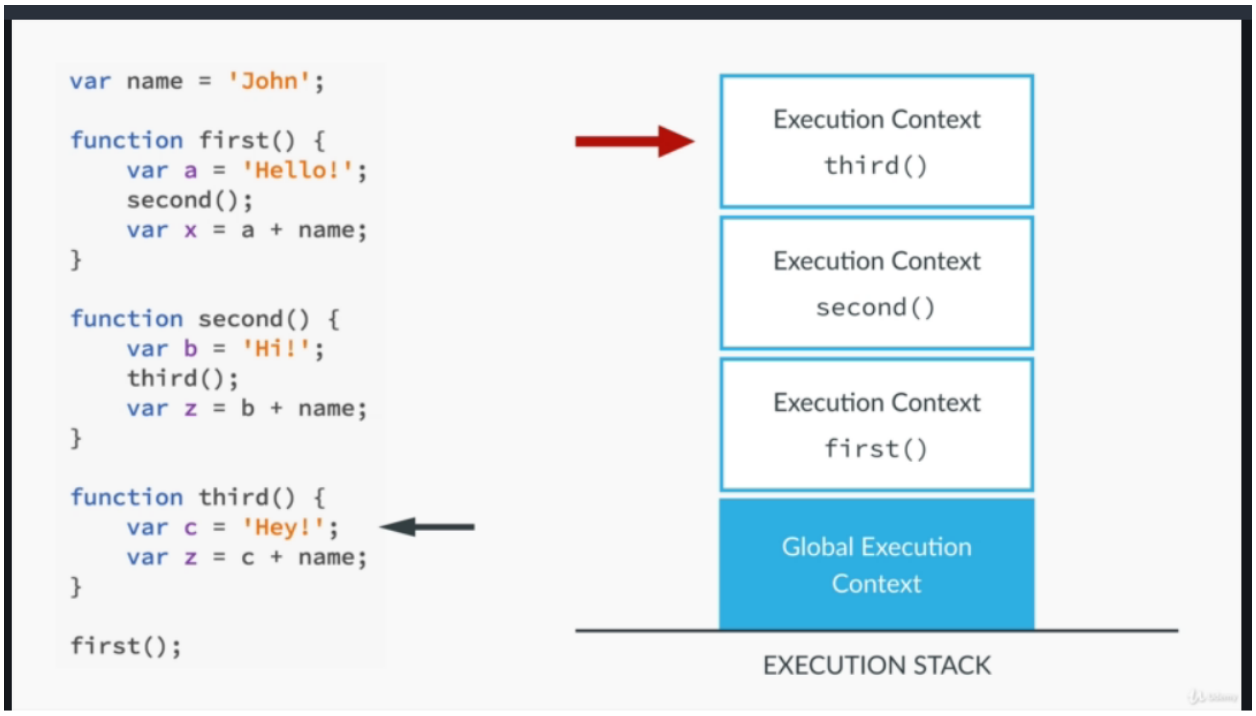
Global Execution Context

- Code that is **not inside any function**
- Associated with the **global object**
- In the browser, that's the `window` object

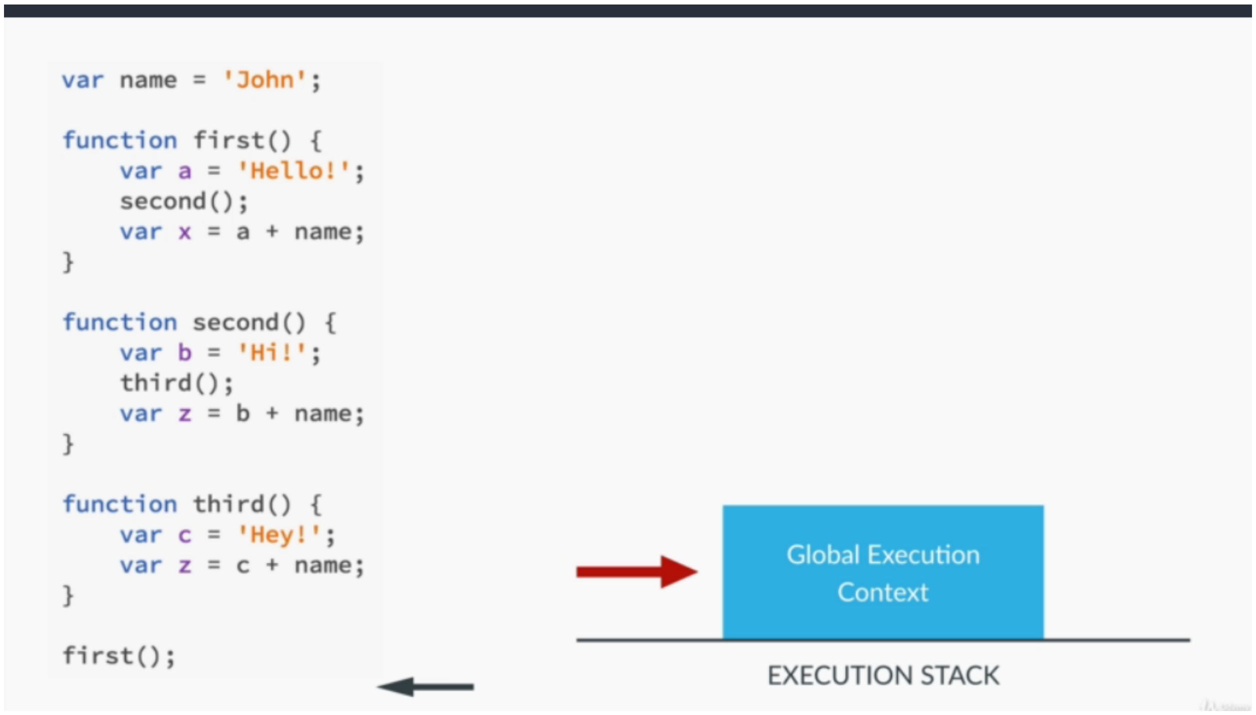
```
lastName === window.lastName
// true
```

- ➔ As we know that Global Execution Context is not for functions, so what happens to the code inside the functions?
- ➔ It's actually simple, whenever a function is called , the code inside function gets its own execution context.

- ➔ Say for e.g. here, Up till the third function (the variable name and the three functions) all are in Global Execution Context on the Execution Stack.
- ➔ Now when the function first() is called, the function gets its new Execution Context, and this new context is put on top of the current context (Global Execution Context) forming the so called Execution Stack.
- ➔ So now the Execution Context for the function first () becomes the active context. The 'a' variable now gets stored in the execution context for this function. Now second () function is getting called.
- ➔ And now the Execution Context for the function second () becomes the active context. The 'b' variable now gets stored in the execution context for this function. Now third () function is getting called.
- ➔ And once more a new execution context is being created and put on top of the stack. And now this third function has only two variables assignment and after it's done, this function returns. And the execution context for the third function gets removed.
- ➔ And then the execution context for the second function goes back to being the active context. The 'z' variable gets stored in the currently active execution context and then the function returns. SO the execution context of second () function also pops off the stack.
- ➔ Back to the first function now, function completes and pops off the execution context from the stack.



```
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();
```



```
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 CONTEXTS IN DETAIL: CREATION AND EXECUTION PHASES AND HOISTING

- ➔ As we know from the previous part, we can associate an execution context with an object and this object has three properties:
  Variable Object (VO): which will contain function arguments, inner variable declarations as well as function declarations
  Scope Chain: which contains the current variable objects as well as variable objects of its parents.
  "This" Variable
- ➔ We already know that when a function is called, a new execution context is put on top of the execution stack and this happens in two phases:
  1) Creation Phase
  2) Execution Phase

→ 1) Creation Phase

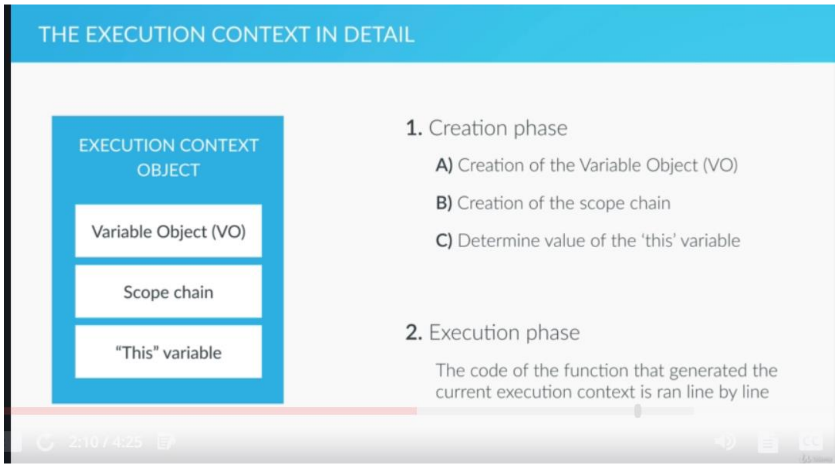
(A) Creation of the Variable Object (VO)

(B) Creation of the Scope Chain

(C) Determine the value of "this" variable.

→ 2) Execution Phase

The code of the function that generated the current execution is run line by line and all the variables are defined. If it’s a global context, then it’s the global code that gets executed.



THE EXECUTION CONTEXT IN DETAIL

EXECUTION CONTEXT OBJECT
- Variable Object (VO)
- Scope chain
- “This” variable

1. Creation phase
   - **A)** Creation of the Variable Object (VO)
   - **B)** Creation of the scope chain
   - **C)** Determine value of the ‘this’ variable
2. Execution phase

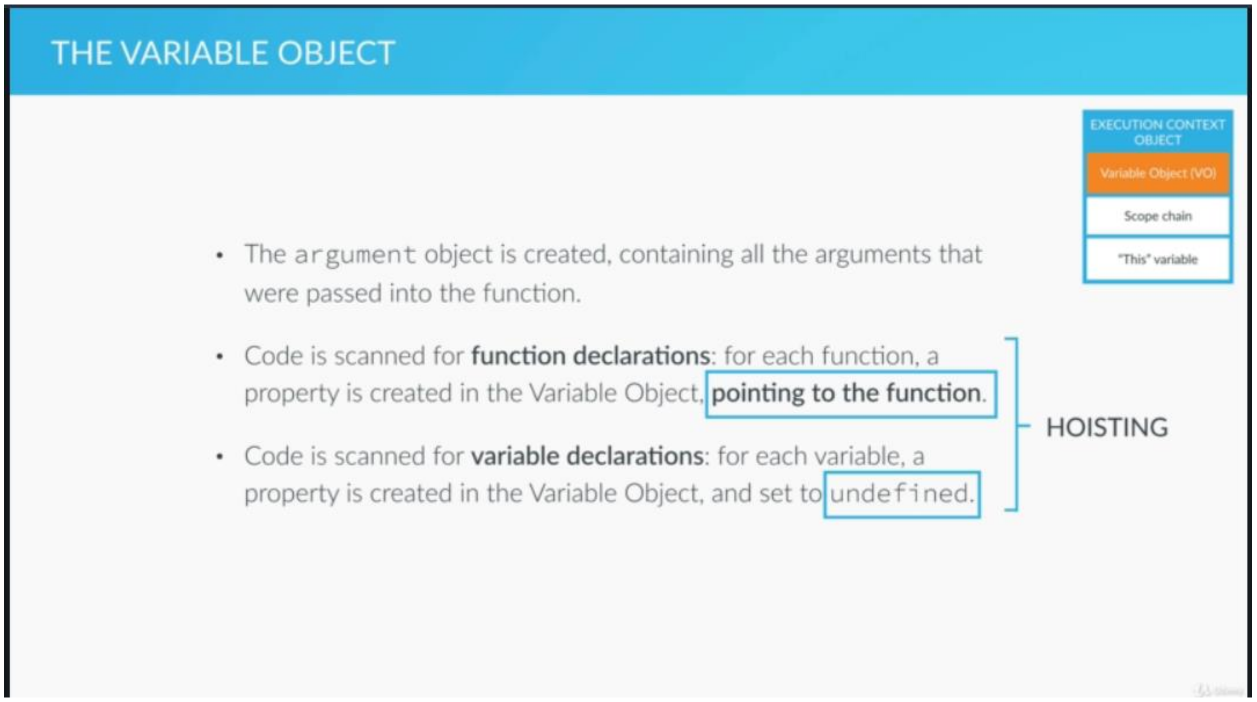
   The code of the function that generated the current execution context is ran line by line

➔ THE VARIABLE OBJECT

- The argument object is created, containing all the arguments that were passed into that function.
- The code is scanned for **function declarations**: for each function, a property is created in the Variable Object, pointing to the function. This means all the functions will be stored inside the variable object, even before the code starts executing (important*)
- Code is scanned for **variable declarations**: for each variable, a property is created in the variable object, and set to undefined.

And these last two points are called hoisting. Function and Variable are hoisted in JS which means they are available before the execution phase actually starts. They are hoisted in different way though. The difference is that the functions are already defined before the execution phase starts whereas the variables are set to undefined and will only be defined in execution phase.

As we remember, the execution phase just comes after the creation phase.



THE VARIABLE OBJECT

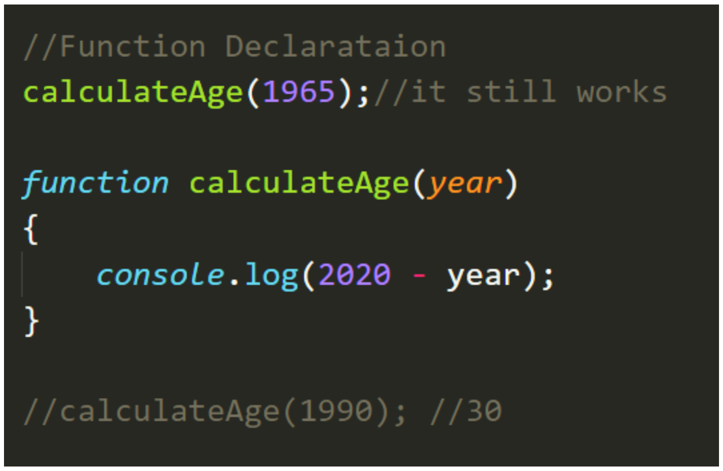
- The `argument` object is created, containing all the arguments that were passed into the function.
- Code is scanned for **function declarations**: for each function, a property is created in the Variable Object, **pointing to the function**.
- Code is scanned for **variable declarations**: for each variable, a property is created in the Variable Object, and set to `undefined`.

--- 4 ---

HOISTING IN PRACTICE

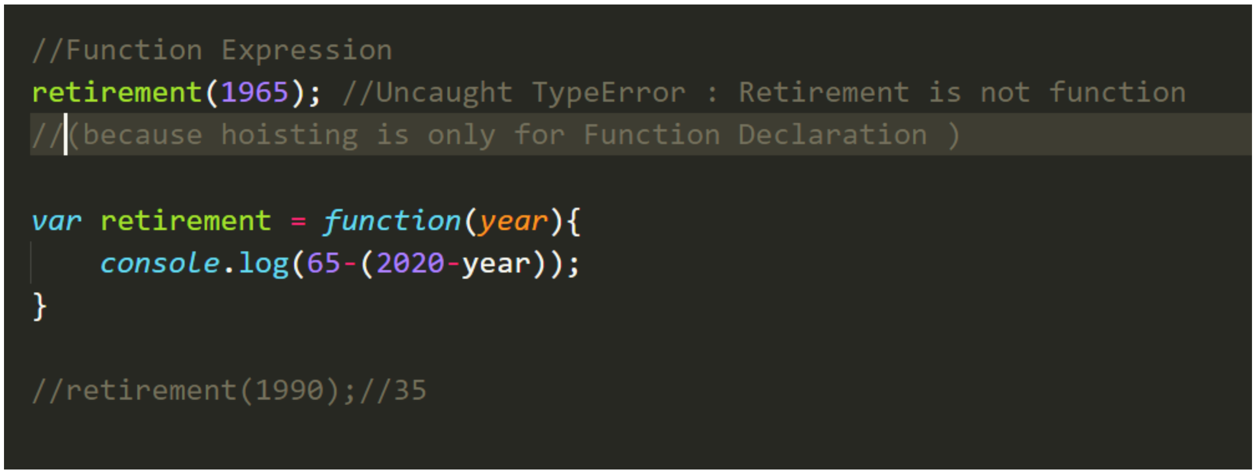
FUNCTIONS

During the creation phase of execution context which here is the global execution context, the function declaration calculateAge() is stored in the Variable Object even before the code is executed. And that is why when we enter the execution phase the function is already available for us to use it, so we don't have to first declare the function and then use it. We can use the function even before we declare it.



```
//Function Declarataion
calculateAge(1965);//it still works

function calculateAge(year)
{
    console.log(2020 - year);
}

//calculateAge(1990); //30
```

But for function Expression, hoisting doesn’t work and we have to declare the function before its being called else it throws error.

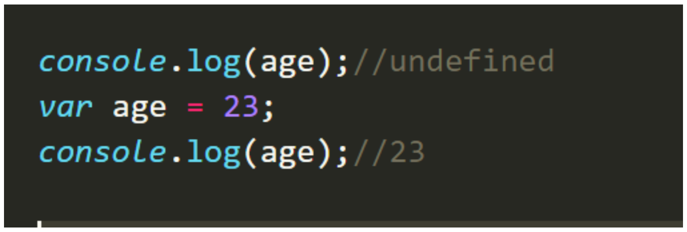


```
//Function Expression
retirement(1965); //Uncaught TypeError : Retirement is not function
//(because hoisting is only for Function Declaration )

var retirement = function(year){
    console.log(65-(2020-year));
}

//retirement(1990);//35
```

VARIABLES

In the creation phase of the Variable Object, the code is scanned for the variable declaration and the variables are then set to undefined.

Variables that don’t have the value yet, will always have the datatype undefined.



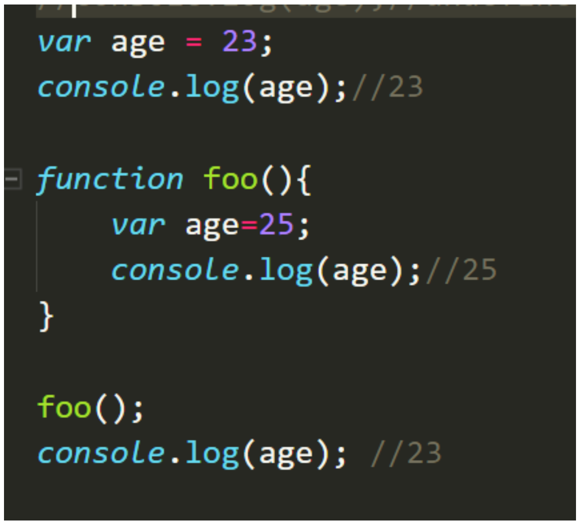
```
console.log(age);//undefined
var age = 23;
console.log(age);//23
```

If we don’t declare the variables and then try to use it we get error Uncaught Reference Error : var varname is not defined.

The age prints 25 and 23 because,

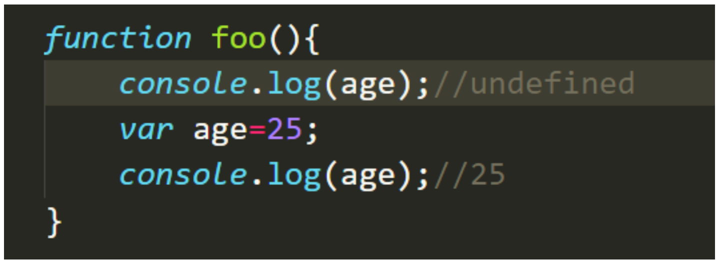
first age var gets stored here in global execution context object here.

foo() function has its own execution context and we can declare a variable with same name, i.e. age which doesn’t really matter because they are declared in the variable object of two different execution context and are hence are treated as two different variables.



```
var age = 23;
console.log(age);//23

function foo(){
    var age=25;
    console.log(age);//25
}

foo();
console.log(age); //23
```

Inside the foo() also hoisting works in the same way,

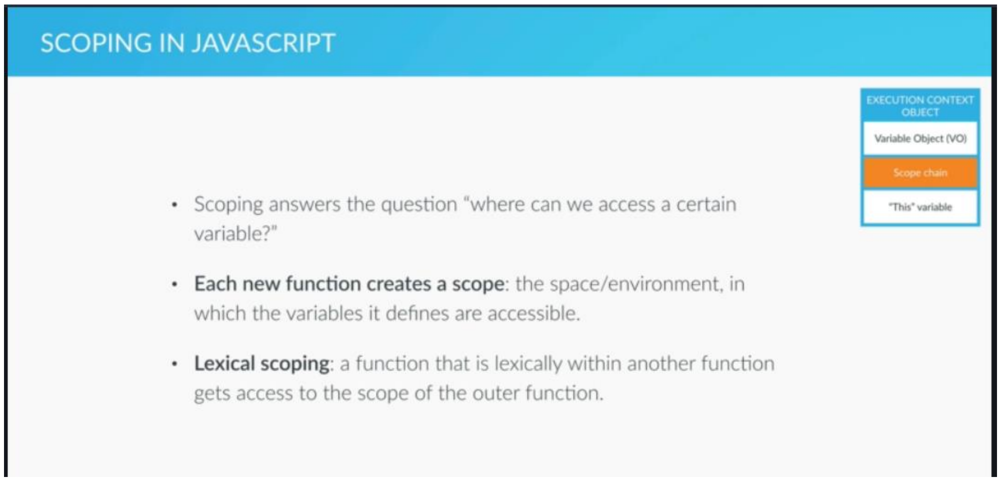


```
function foo(){
    console.log(age);//undefined
    var age=25;
    console.log(age);//25
}
```

SCOPING AND SCOPE CHAIN

What does scoping means? Scoping answers the question "where can we access a certain variable?"

- ➔ Each new function creates a scope: the space/environment, in which the variables it defines are accessible.
- ➔ In many other programming languages scopes are created by if-else blocks , for loops etc., but not in JS. In JS, the only way to create a new scope is to create a function.
- ➔ In JS we have Lexical Scoping: a function that is lexically within another function gets access to the scope of outer function (also called parent function).
- ➔ In this way it gets access to the variables and the function which the parent function defines.



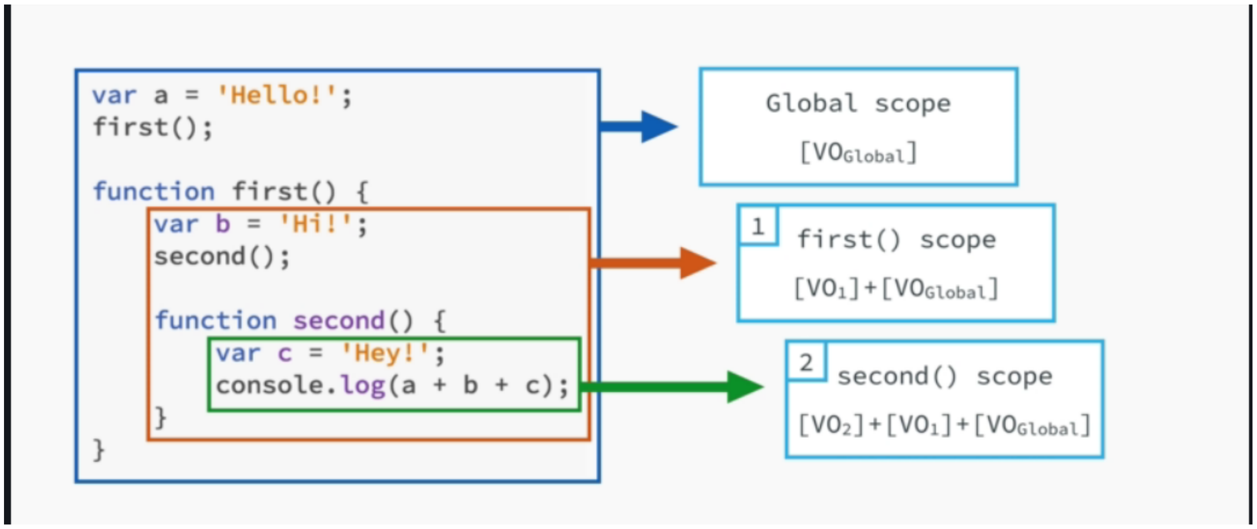
SCOPING IN JAVASCRIPT

- Scoping answers the question "where can we access a certain variable?"
- **Each new function creates a scope**: the space/environment, in which the variables it defines are accessible.
- **Lexical scoping**: a function that is lexically within another function gets access to the scope of the outer function.

EXECUTION CONTEXT OBJECT
Variable Object (VO)
Scope chain
"This" variable

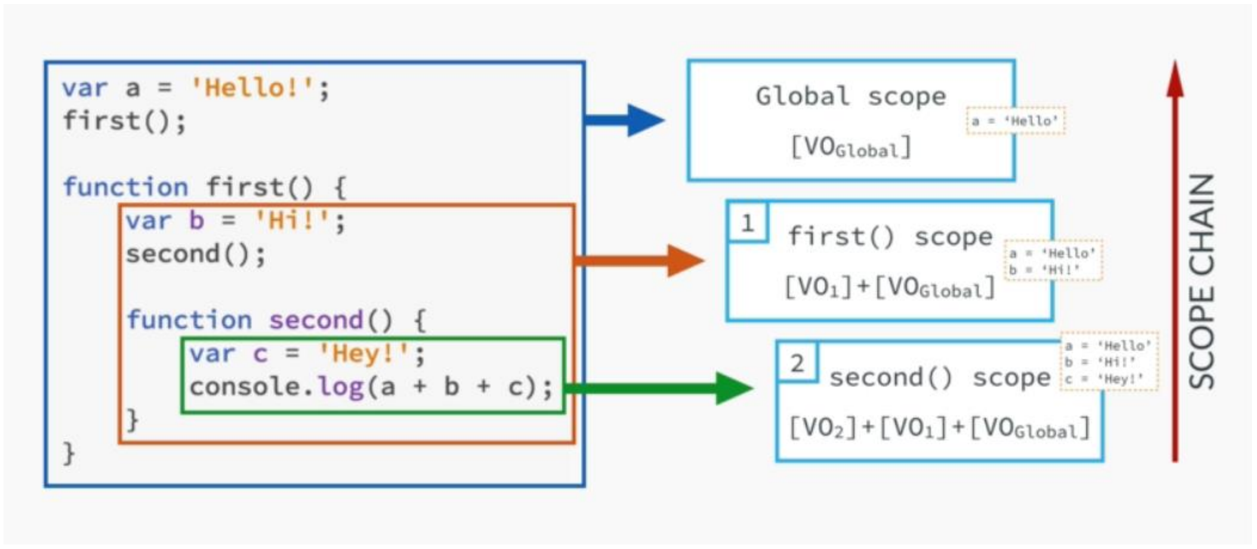
➔ Here in this example, the first() function has access to the Global Scope, thanks to Lexical Scoping and thus it can access var a of Global Scope.

➔ Likewise second() function can access the var b (in first() Scope) due to Lexical Scoping and hence it prints the value of a+b+c



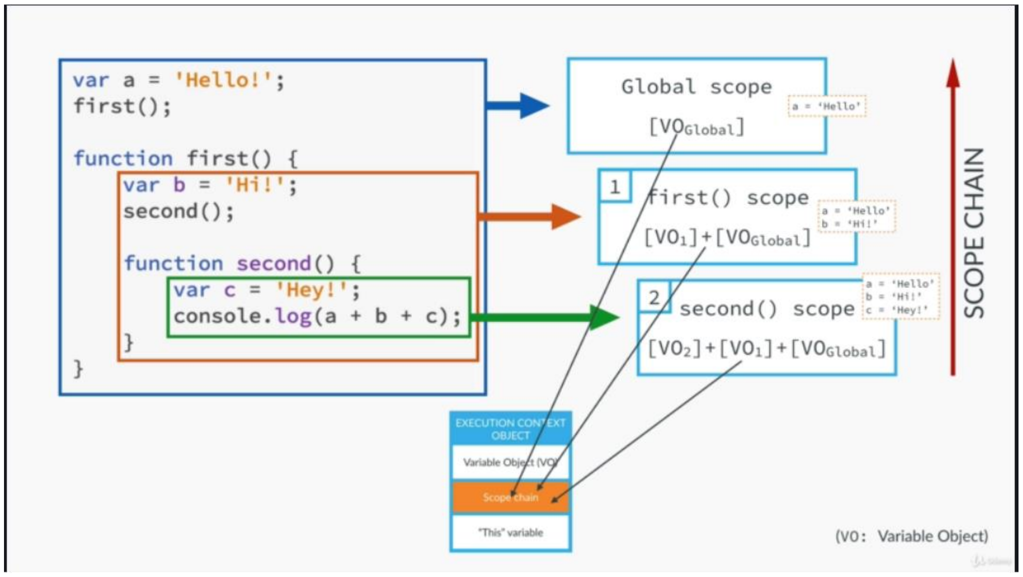
```
var a = 'Hello!';
first();

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

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

➔ For second function, it doesn't find the var a in parent (first()) scope, so it goes more up, all the way to the global scope. And this is exactly called the SCOPE CHAIN.



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

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

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

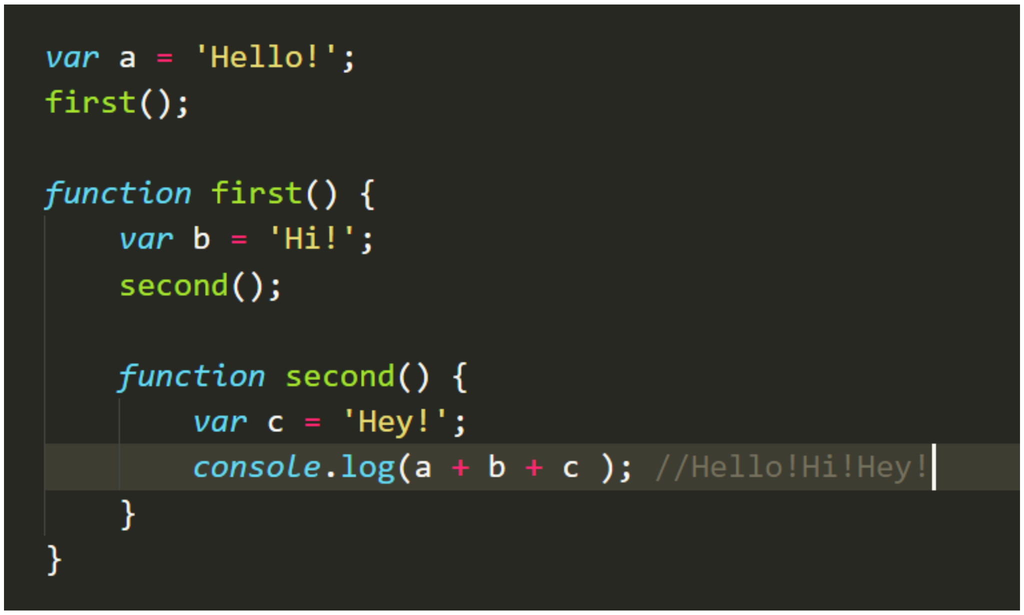
- ➔ Only if the JS engine doesn't find the variable anywhere, it throws the error and stops execution.
- ➔ However it doesn't work in reverse, i.e. locally scoped variables are not visible to their parent scopes, unless we return those variables from the functions. Here the var b and var c wont be accessible from the global scope.
- ➔ How does that work behind the scenes?
- ➔ Remember ECO, in the creation phase, each ECO will get the exact scope chain which is basically all the variable objects that an execution context has access to. (because variable object is what stores all the defined variables and functions.)



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

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

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

ECO : Execution Context Object

- ➔ The Scope Chain of second() will have access to Variable Object of second() fn, first() fn and the Global VO.



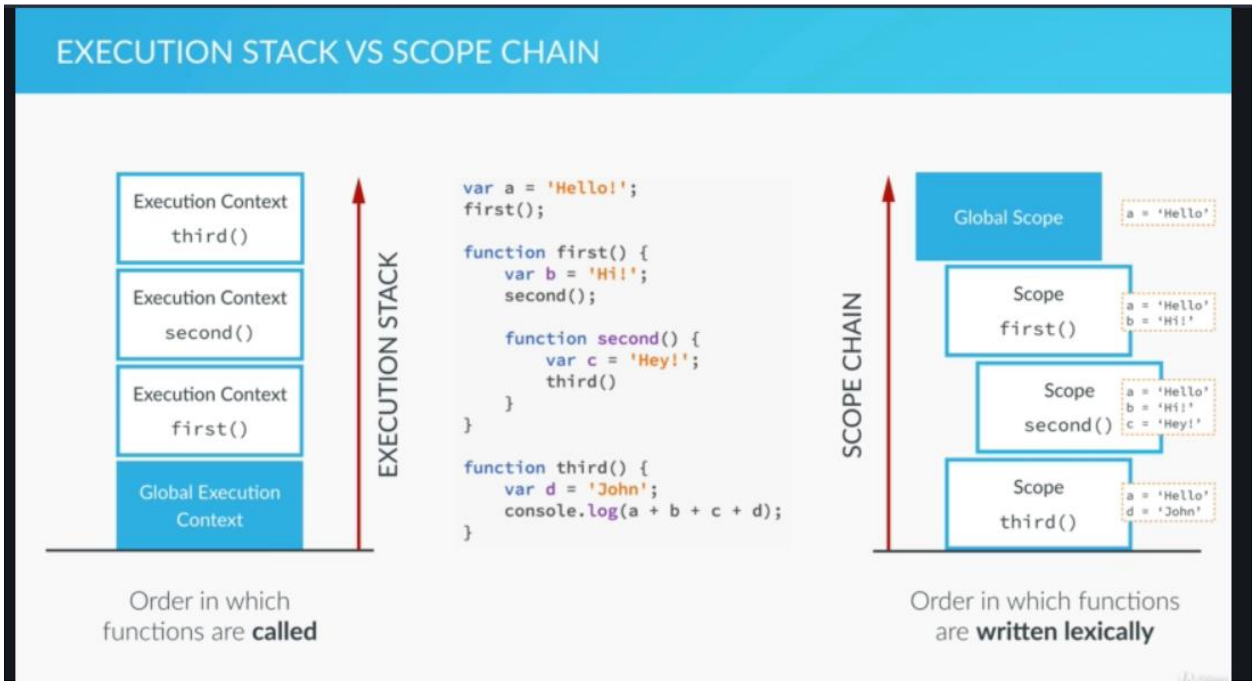
```
var a = 'Hello!';
first();

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

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

Here it’s possible as second() fn has access to variables of first() fn and Global Scope, because second() is written inside first() and first() inside Global Scope, and that’s why we call it Lexical Scoping.

EXECUTION STACK VS SCOPE CHAIN

→ Execution Stack is different from Scope Chain

- ➔ For each new call of function(), a new EC is put on top of Execution Stack.
- ➔ For Scope Chaining, Global Scope contains the var a, first() fn and third() fn. Then the scope of first() fn contains the scope of second() fn

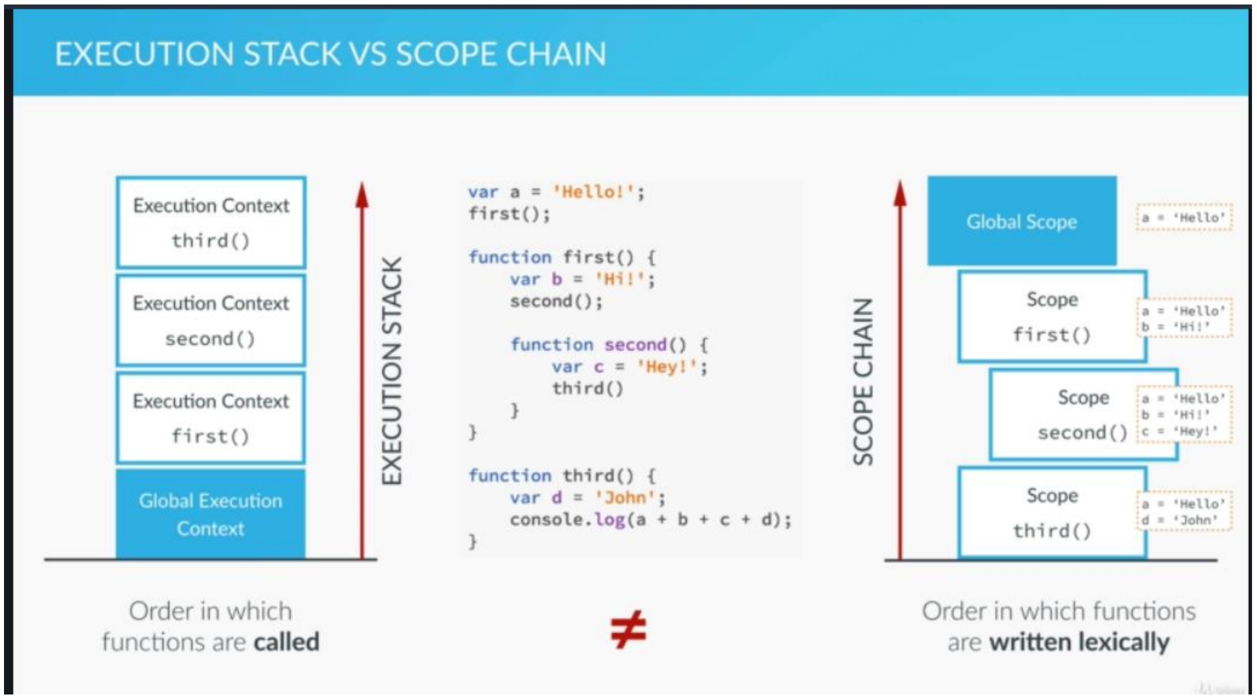


EXECUTION STACK VS SCOPE CHAIN

```
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**
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**

- ➔ Remember that Execution Stack is the order in which functions are called, but the Scope Chain is the order in which functions are written in code.
- ➔ Here the third function cannot access var b and var c, and has access to only the var a in Global Scope, since it is not in scope of second() fn.

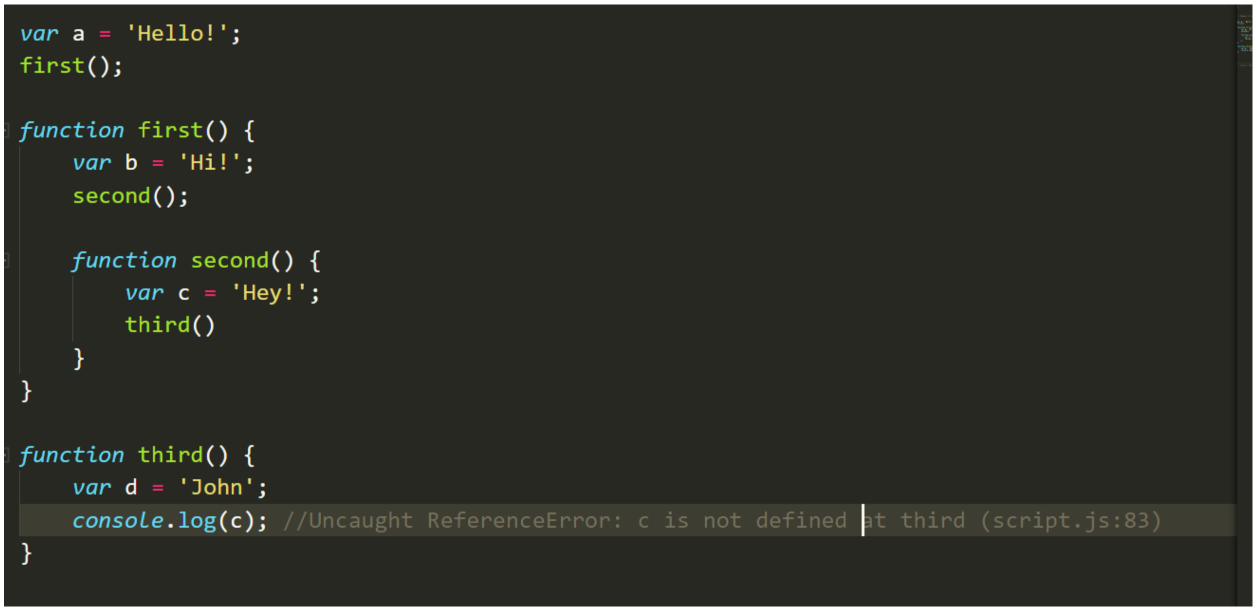


EXECUTION STACK VS SCOPE CHAIN

```
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);
}
```

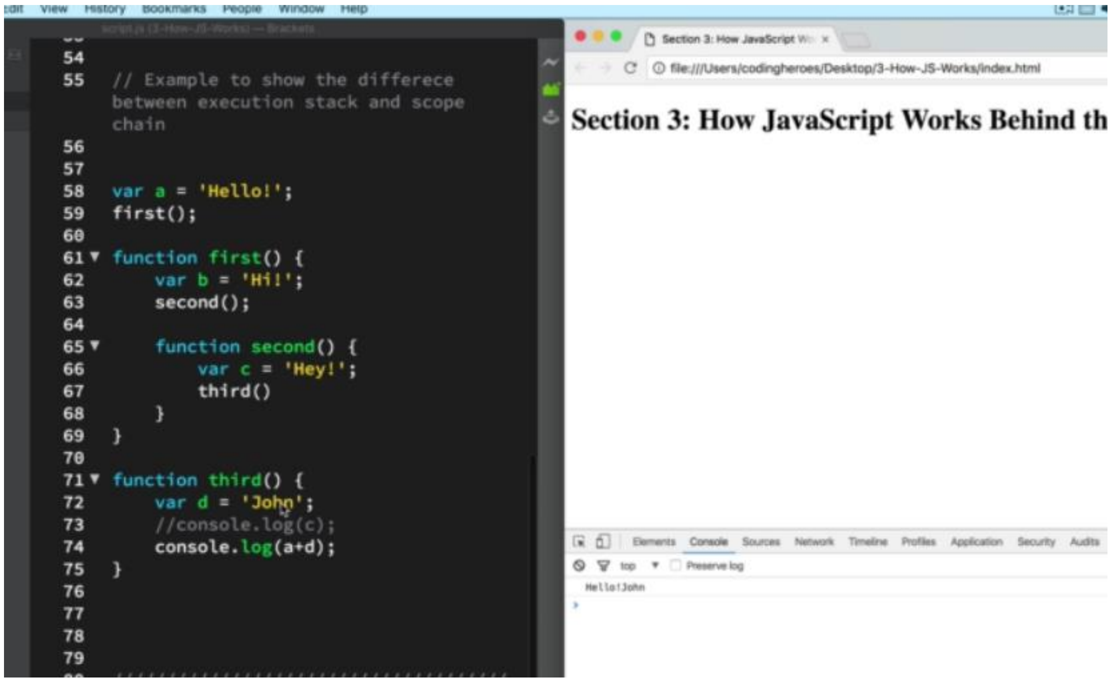
Execution Context third()
Execution Context second()
Execution Context first()
Global Execution Context
EXECUTION STACK
Order in which functions are **called**
≠
SCOPE CHAIN
Global Scope
a = 'Hello'
Scope first()
a = 'Hello' b = 'Hi!'
Scope second()
a = 'Hello' b = 'Hi!' c = 'Hey!'
Scope third()
a =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

- ➔ It's the EC which stores the Scope Chain of each function in the Variable Object.



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

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

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

function third() {
    var d = 'John';
    console.log(c); //Uncaught ReferenceError: c is not defined at third (script.js:83)
}
```

- ➔ Second() can call third() because of Scope Chaining as the second () function can access scope of first() and the global scope.
- ➔ In the example above we get an Error that c is not defined, because c is defined in second() fn which then calls third() and third cannot access the var c because the execution stack is different from the scope chain. The order in which the functions were called doesn't matter, all that matters is that the third function is in a different scope than second function and so it cannot access var c.
- ➔ So which variables can third() access, the answer is var a and var c (as you can see here it prints Hello John).



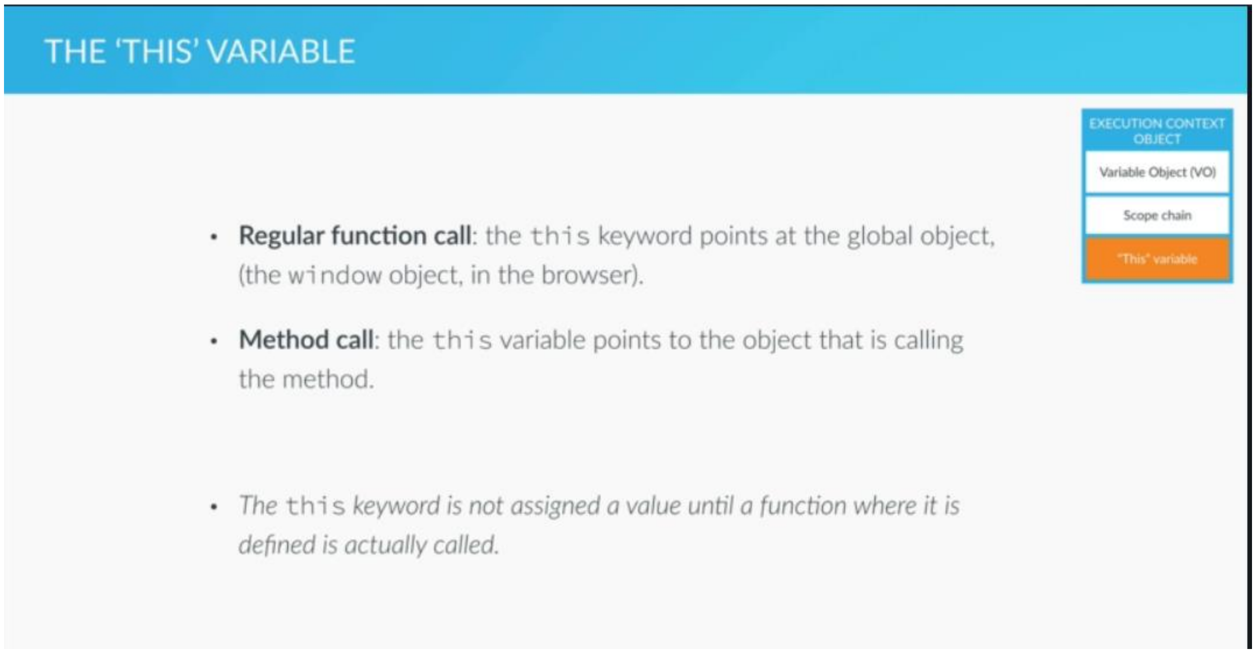
```
// Example to show the differece
between execution stack and scope
chain


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

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

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

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

Section 3: How JavaScript Works Behind th

```
Hello!John
```

THE “THIS” KEYWORD

- ➔ The “this” variable is the variable that each and every execution context gets, and it is stored in the Execution Context Object.
- ➔ In a Regular Function Call : the this keyword points at the global object (the window object, in the browser).
- ➔ In Method Call : the “this” variable points to the object that is calling the method.
- ➔ The “this” keyword is not assigned a value until a function where it is defined is actually called.
- ➔ Again, even though it appears that the “this” variable refers to the object where it is defined, but “this” variable is technically only assigned a value as soon as an object calls a method.
- ➔ The “this” keyword is attached to an Execution Context which only gets created as soon as the function is invoked/called.


EXECUTION CONTEXT OBJECT
Variable Object (VO)
Scope chain
"This" variable

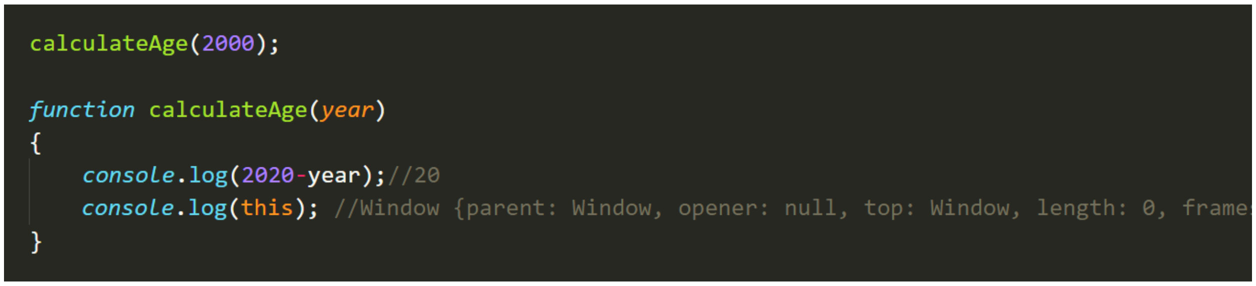
THE 'THIS' VARIABLE

- **Regular function call**: the this keyword points at the global object, (the window object, in the browser).
- **Method call**: the this variable points to the object that is calling the method.
- *The this keyword is not assigned a value until a function where it is defined is actually called.*

- ➔ ‘this’ in global execution context refers to window object, that’s because window object is the default object.

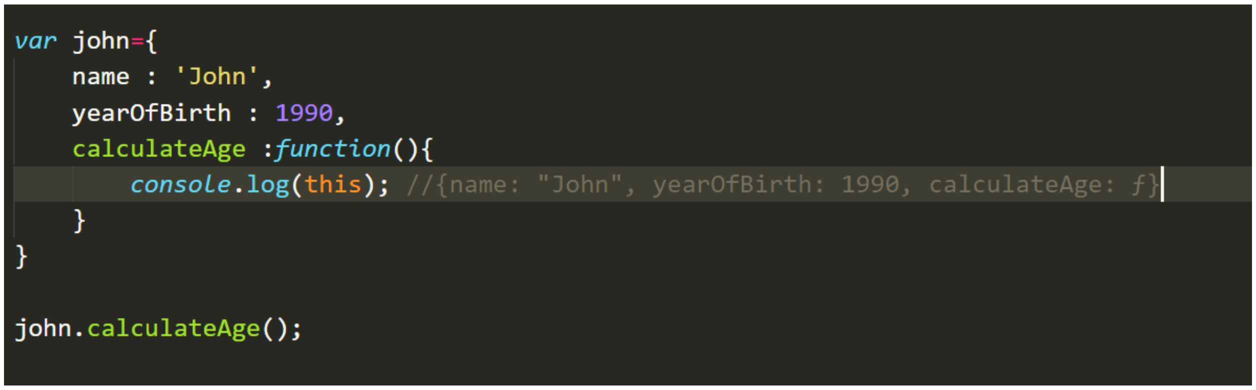
```
console.log(this);
//Window {parent: Window, opener: null, top: Window, length: 0, frames: Window, …}
```

- ➔ Here when we call the method calculateAge, ‘this’ refers to window object, because the object to which this function is attached here is global object

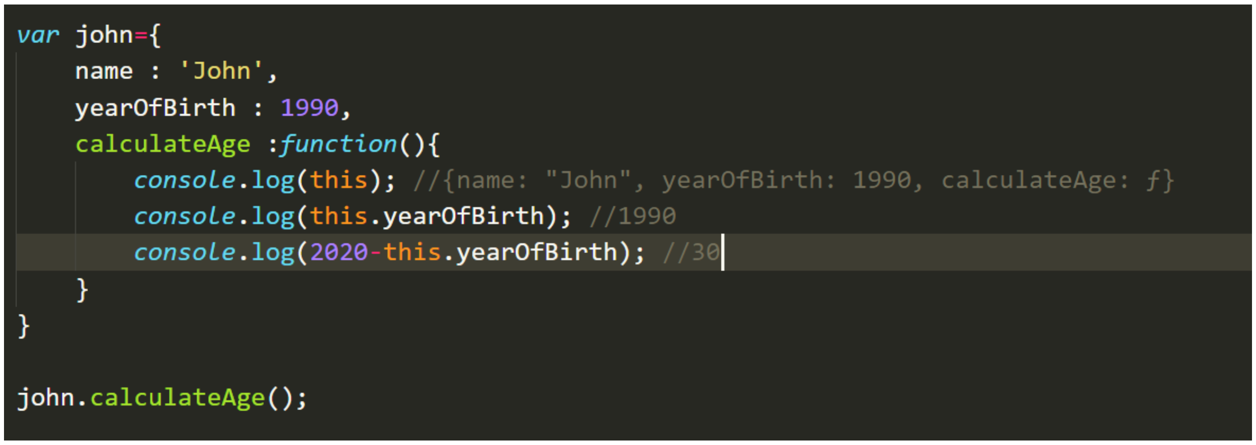


```
calculateAge(2000);

function calculateAge(year)
{
    console.log(2020-year);//20
    console.log(this); //Window {parent: Window, opener: null, top: Window, length: 0, frame
}
```

➔ 'this' variable now is the John Object as 'this' keyword refers to the object that called the method and in this case it was john object.

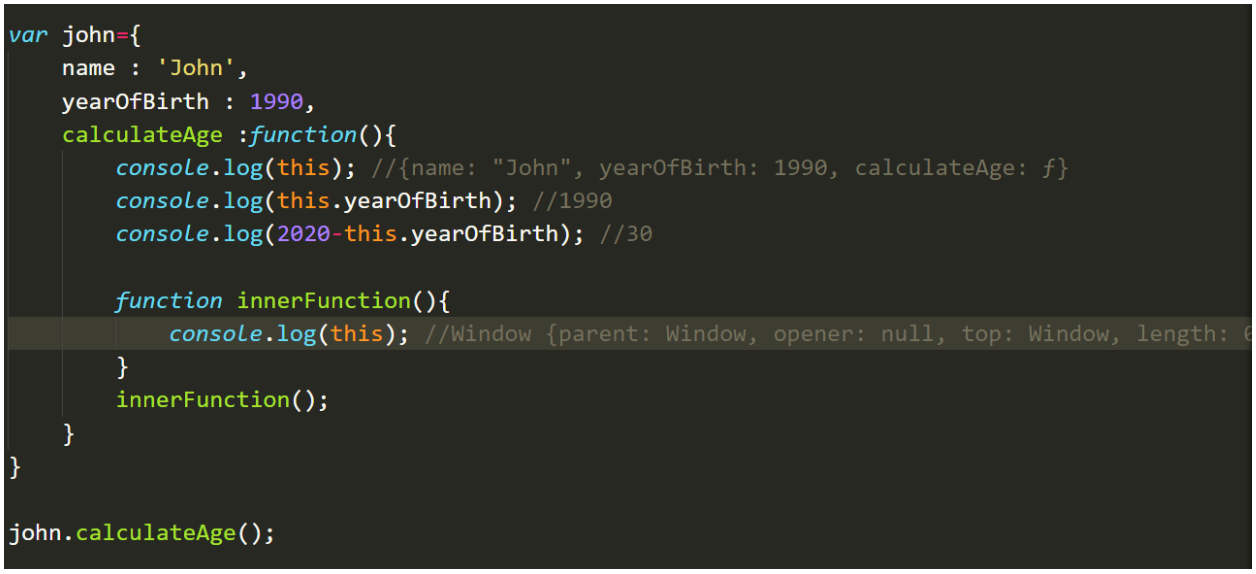


```
var john={
    name : 'John',
    yearOfBirth : 1990,
    calculateAge :function(){
        console.log(this); //{name: "John", yearOfBirth: 1990, calculateAge: ƒ}
    }
}

john.calculateAge();
```



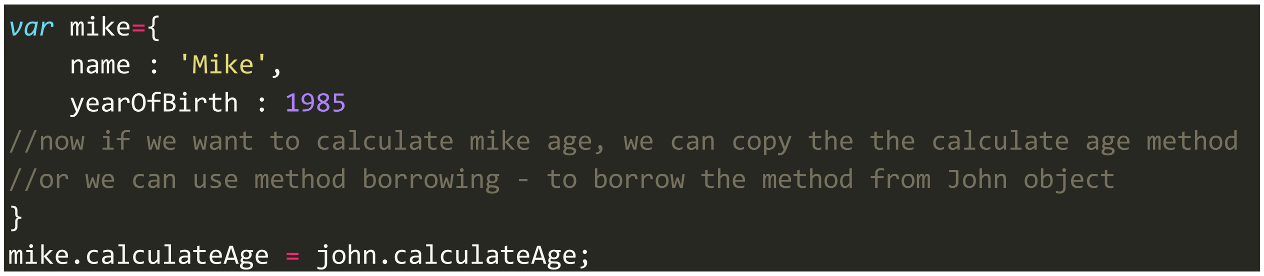
```
var john={
    name : 'John',
    yearOfBirth : 1990,
    calculateAge :function(){
        console.log(this); //{name: "John", yearOfBirth: 1990, calculateAge: ƒ}
        console.log(this.yearOfBirth); //1990
        console.log(2020-this.yearOfBirth); //30
    }
}

john.calculateAge();
```

➔ For the this keyword in innerfunction (),
When a regular function call happens, then the default object is the window object.
Although its written inside john objects calculateAge() fn, it is still a regular function, so when we call 'this', it points to the window object and not john object.



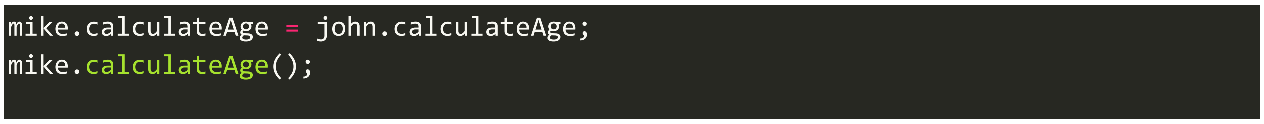
```
var john={
    name : 'John',
    yearOfBirth : 1990,
    calculateAge :function(){
        console.log(this); //{name: "John", yearOfBirth: 1990, calculateAge: ƒ}
        console.log(this.yearOfBirth); //1990
        console.log(2020-this.yearOfBirth); //30

        function innerFunction(){
            console.log(this); //Window {parent: Window, opener: null, top: Window, length:
        }
        innerFunction();
    }
}

john.calculateAge();
```

- ➔ 'this' variable is only assigned a value as soon as the object calls a method
- ➔ Here when we have created Mike Object,

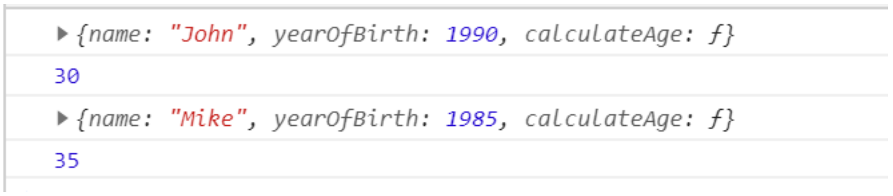


```
var mike={
    name : 'Mike',
    yearOfBirth : 1985
//now if we want to calculate mike age, we can copy the the calculate age method
//or we can use method borrowing - to borrow the method from John object
}
mike.calculateAge = john.calculateAge;
```

- ➔ Here we are not using parenthesis because we are not calling function, we are treating them as variables here. And assigning john's method to Mike Object by a method called Method Borrowing.
- ➔ So, we can say that we want the mike method to be same as John Method
- ➔ When we call the mike method after method borrowing, we get Mike Object on printing 'this', which proves that 'this' variable is only assigned a value as soon as the object calls a method



```
mike.calculateAge = john.calculateAge;
mike.calculateAge();
```



```
▶ {name: "John", yearOfBirth: 1990, calculateAge: f}
30
▶ {name: "Mike", yearOfBirth: 1985, calculateAge: f}
35
```

➔ This variable only becomes something when the method is called.