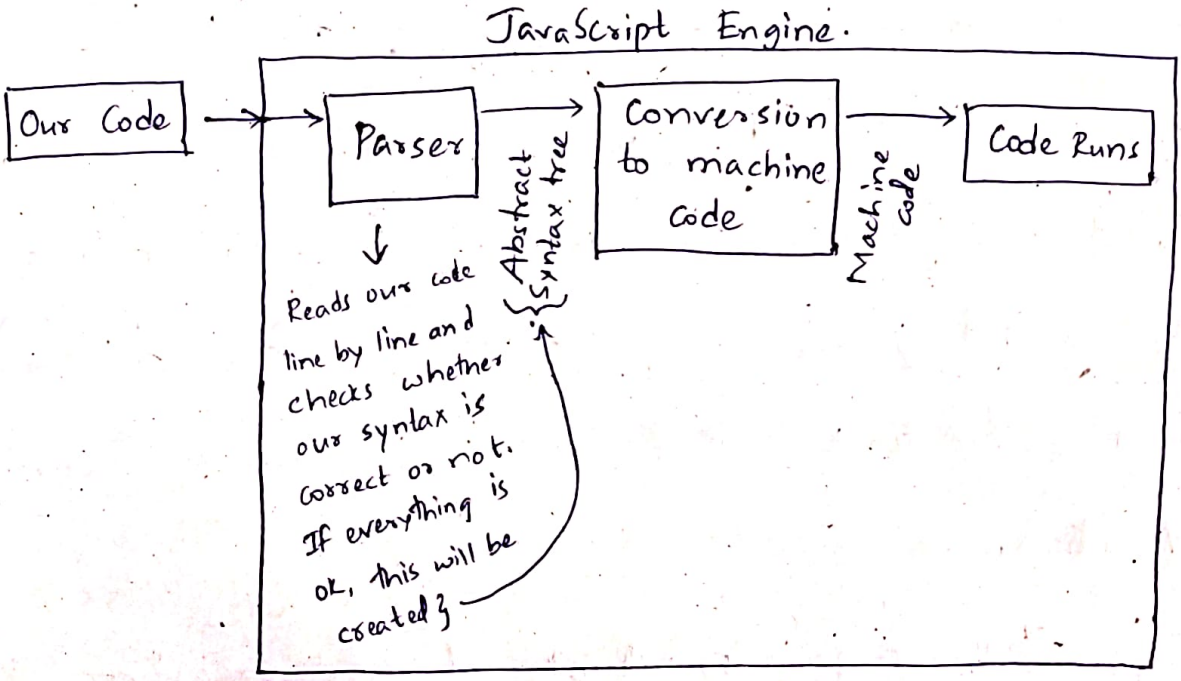
V.IMP

Chapter 3: How JavaScript Works Behind the Scenes:

3. How Our Code is Executed: JavaScript Parsers and Engines:



V.IMP

4. Execution Contexts and the Execution Stack:

This video talks about the order in which the code is run. For that we have to know about Execution Context:

Execution Context: All JavaScript code needs to run in an environment and these environments are called execution contexts. So we 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 a global execution context, all the code that is not inside any function is executed. So remember, the 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. Remember objects from the introductory lectures? 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 a global coritext automatically gets attached to the window object in the browser and it works like this:

```
lastName === window.lastName
```

i.e.: declaring a variable called lastName or window.lastName is the exact same thing. It's like the lastName is a property of the window object and as we talked in the intro lectures, properties are just variables attached to objects.

Consider this code:

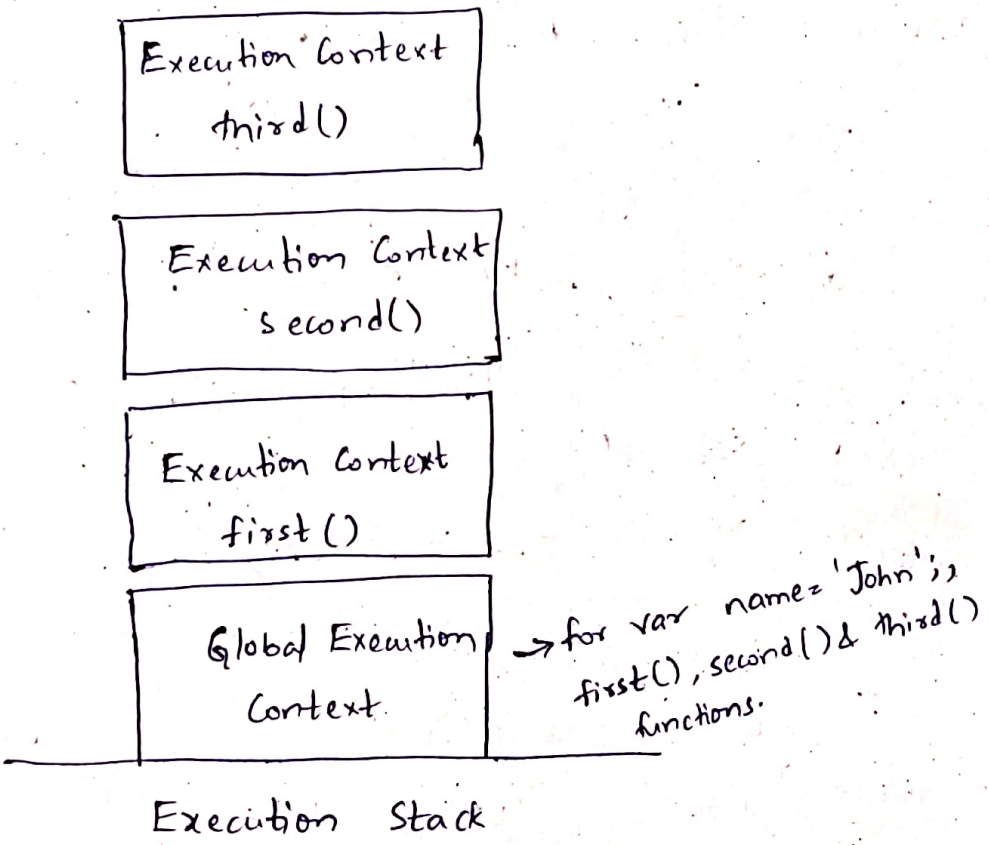
```
var name = 'John';

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();
```

Remember the global execution context is for code that is not inside of any function. But what about the code that is in a function. Each time that we call a function, it gets its own brand new execution context



As we call the very first function, its gets its own execution context and this new context will be put on top of the current context, forming the so called execution stack. Now this becomes the active context in which

the code is executed. Now lets get into our first function. The 'a' variable in the first function will be stored in the execution context for the first() function. and not anymore in the global context. And inside of first() function, we call the second() function. So once again, a new execution context will be created and put on top of the first() execution context. Then this second() execution context becomes the active context. The variable 'b' will be stored in this new execution context. And inside of the second() function, we call the third() function. So once again, a new execution context will be created and put on top of the stack.

Now the function has done its work and now it returns. So what happens to its execution context? It gets removed from the top of the stack. And then,

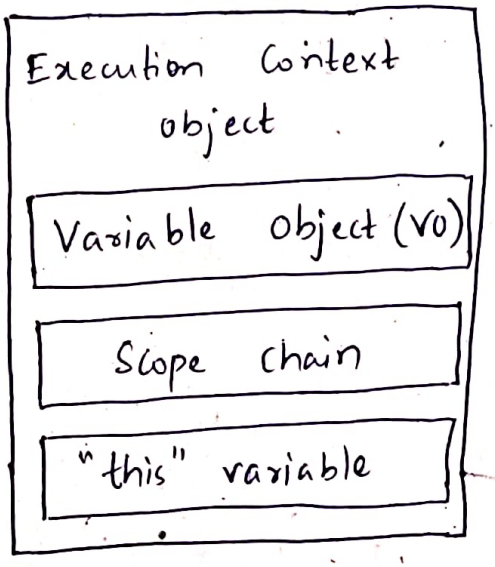
the context of the second function which called the third function is back to being the active context. And we go back to the second() function. So the z variable gets stored in the currently active context and after the function returns, this context also pops out of the stack. Similarly, the first() function returns and then it gets popped out of the stack.

In this way, we executed all three functions, created three execution contexts and popped them off the stack when the function returned.

5. Execution Contexts in Detail: Creation and Execution Phases and Hoisting:

In the last lecture we have seen when a new execution context is created. Let's now talk about how exactly that happens.

In the previous lecture, we have mentioned that we can associate execution context with an object. And consider that object has three properties.



The variable object (VO) contains function arguments, inner variable declarations & function declarations

The scope chain contains current variable objects and variable objects of all its parents. The "this" variable, we have already seen how this works.

When a function is called, a new execution context is put on top of the execution stack. This happens in two phases.

1. Creation phase
2. Execution phase.

1. Creation phase:

→ Creation of the Variable Object (VO).

→ Creation of the scope chain.

→ 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.

Creation of the Variable Object (VO):

→ 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.
This means all the functions will be stored inside a Variable object even before the code starts executing

→ Code is scanned for variable declarations: for each variable, a property is created in the variable object, and set to undefined.

→ Hoisting.

The last two points is called hoisting. Functions and variables are hoisted in JavaScript which means they are available before the execution phase actually starts. They are hoisted in a different way though.

The difference between functions and variables is that functions are already defined before the execution phase starts while variables are set to undefined.

and they will only be defined in the execution phase

Recap: Each execution context has an object which stores a lot of important data that the function will use while it's running and this happens even before the code is executed.

6. Hoisting in Practice:

& script.js:

```
// functions
calculateAge(1965);

function calculateAge(year) {
    console.log(2016 - year);
}
```

It's output will be:

51.

In the creation phase of the creation context, which is in this case the

global execution context, the function declaration calculateAge is stored in the variable object even before the code is executed. This is why when we entered the execution phase the calculateAge() function is already available for us to use it. So for function declarations, we can first use the function and then we can declare it.

script.js:

```
//function expression.
retirement(1956);

var retirement = function(year) {
    console.log(65 - (2016 - year));
}
```

Its output will be:

Uncaught TypeError: retirement is not a function.

The reason for this error is that the retirement function isn't a function declaration but a function expression.

Hoisting with functions only works for function declarations.

Script.js:

```
//variables:

console.log(age);

var age = 23;
```

Output:

undefined.

This is because in the creation phase of the variable object, the code is scanned for variable declarations and the variables are then set to undefined.

If we try this code:

```
console.log(age);
```

Output:

Uncaught ReferenceError: age is not defined

This is bcoz we don't have any definition and JavaScript wouldn't even know this variable.

But in the first variable example, JS knows that there's an age variable but it simply doesn't have a value yet.

script.js:

```
console.log(age)
var age = 23;
function foo(){
    console.log(age);
    var age = 65;
    console.log(age);
}
foo();
console.log(age);
```

This is stored in the global execution context object

This gives output as undefined bcoz

This gets its own execution object context in which we can also store the age variable and it can be the same name bcoz they are stored in two different execution contexts.

Output:
undefined
undefined
65

The most important use case for hoisting is the fact that we can use function declarations before we can actually declare them in our code.

V.IMP

7. Scoping and the Scope Chain:

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. In many other programming languages, scope is also created by if blocks, for blocks and while blocks but not in JavaScript. In JS, the only way to create a new scope is to write a new function.

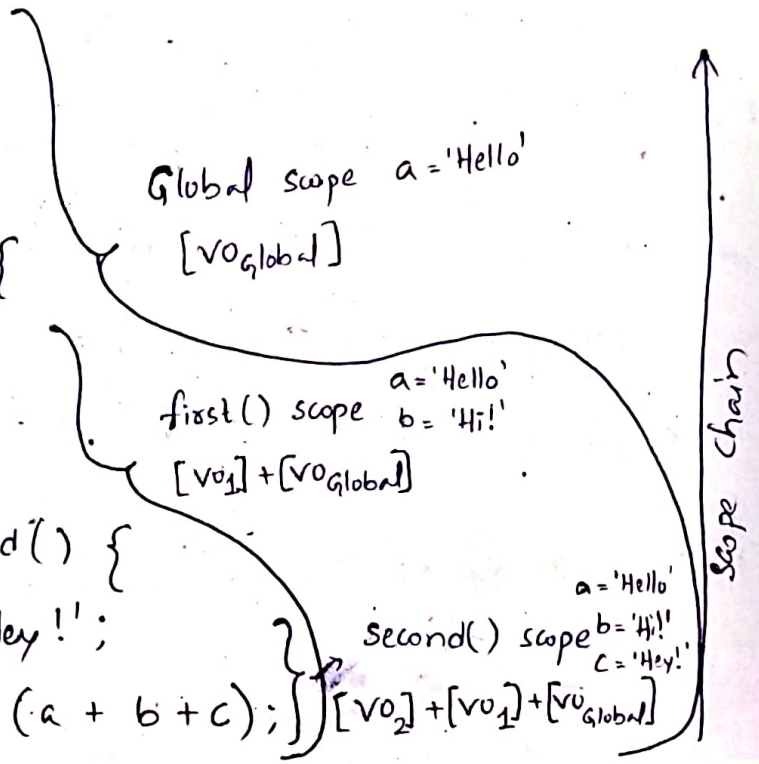
→ Lexical scoping: a function that is lexically within another function gets access to the scope of the outer function.

Example code:

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

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

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



The global scope will not have access to the variables b & c unless we return the values from the function. So locally scoped variables are not visible to their parent scopes

If we run the above code, we'll get this output:

Hello!Hi!Hey!

This output is bcoz of the scoping chain. In this case the second function has access

to the variables of the first function and of the global scope. That's bcoz the second function is written inside of the first. which inturn is written inside of the global scope. That's why we call it Lexical Scoping.

Execution stack vs. Scope chain:

Consider this code:

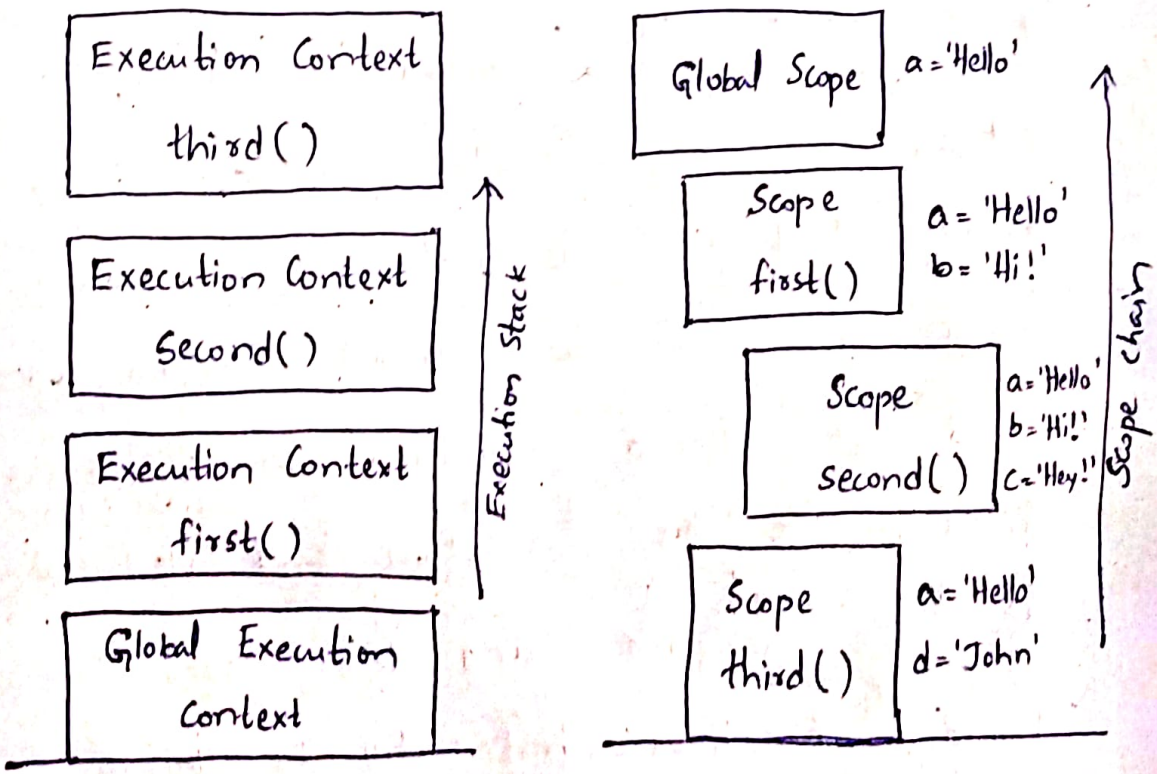
```
var a = 'Hello!';
first();

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

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

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

Order in which the functions are called: ≠ order in which the functions are written Lexically:



If we run the above code, this is the output:

Uncaught ReferenceError: c is not defined

→ Why can the second function even call the third function? Notice that the third function was called from the second function and this is possible bcoz of scoping. The second

function has access to the third function because of the scope chain. So the second function has access to the global scope in which the third function sits lexically. So the second function has the ability to call the third function because of the scope chain

→ Why do we actually get this error saying that 'c' is not defined even though it was the second function who called the third function. So the second function is the function that defined the 'c' variable and then called the third function - but still the third function cannot access variable 'c' which it is trying to print. This resulted in an error. Now its obvious that the third function cannot access variable 'c' bcoz the execution stack is different from the scope chain.

So to answer the question, who can access the 'c' variable, the order in which the functions wer called does not matter. All that matters is that the third function is in a different scope than the second function and so it cannot access variable 'c'. So which variables can the third function actually access? It's variable 'a' and 'd'.

If you change the third function in the code to this:

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

Then the output will be:

Hello! John!

8. The 'this' keyword:

The 'this' variable is a variable that each and every execution context gets.

And it is stored in the execution context object. So where does the 'this' variable or the 'this' keyword points?

→ In a regular function call: the 'this' keyword points at the global object, (the window object, in the browser). This is the default.

→ In a 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.

So, even though it appears that the 'this' variable refers to the object where it is defined, the 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 is

only created as soon as the function is invoked, which also means : called.

V. IMP

9. The 'this' keyword in Practice :

script.js:

```
// The this keyword.

console.log (this);
```

Output :

▶ Window {speechSynthesis: SpeechSynthesis, caches: ...
...}

All this means is that the 'this' keyword in the global execution context is very simply the window object.

Script.js:

```
calculateAge(1985);

function calculateAge(year) {
    console.log (2016- year);
    console.log (this);
}
```

Output: 31

▶ Window { speechSyn: ... }

This is a regular function code and not a method. As we learned in the last lecture, in a regular function call, the 'this' keyword always points to the window object.

Script.js:

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

Output: Object { name: "John", yearOfBirth: 1990 }

So the 'this' variable now is the John object. The 'this' keyword refers to the object that called the method. In this case, it is the John object.

Script.js:

```
var john = {
    name: 'John',
    yearOfBirth: 1990,
    calculateAge: function() {
        console.log(this);
        console.log(2016 - this.yearOfBirth);

        function innerFunction() {
            console.log(this);
        }
        innerFunction();
    }
}
john.calculateAge();
```

Output:

▸ Object { name: "John", yearOfBirth: 1990 }

26

▸ Window { speechSynthesis: SpeechSynthesis, caches...
... }

So once again, the 'this' keywood in here is the jonh object as expected then the outcome is 26. But now comes something that looks strange. That's the fact the 'this' keyword in the inner function is now back to being the window.

It makes sense bcoz it is simply the rule. And the rule is that when a regular function code happens, then the default object is the window object, atleast that's how it happens in the browser. So once again, this is not a method, bcoz the method is called calculateAge, so it's a method

of the john object.

But the function in there, although it's written inside of a method, it's still a regular function. So when we call it, the 'this' keyword will no longer point to the john object, but instead point to the window object.

Remember how we discussed that the 'this' variable is only assigned a value as soon as an object calls a method. Let's see an example.

script.js:

```
var john = {
    name: 'John',
    yearOfBirth: 1990,
    calculateAge: function() {
        console.log(this);
        console.log(2016 - this.yearOfBirth);
    }
}
```

```
john.calculateAge();

var mike = {
    name: 'Mike',
    yearOfBirth: 1984
};

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

This is called Method Borrowing.

we are just calling the calculateAge method of john object from mike object.

So the 'this' keyword becomes something as soon as the method gets called.