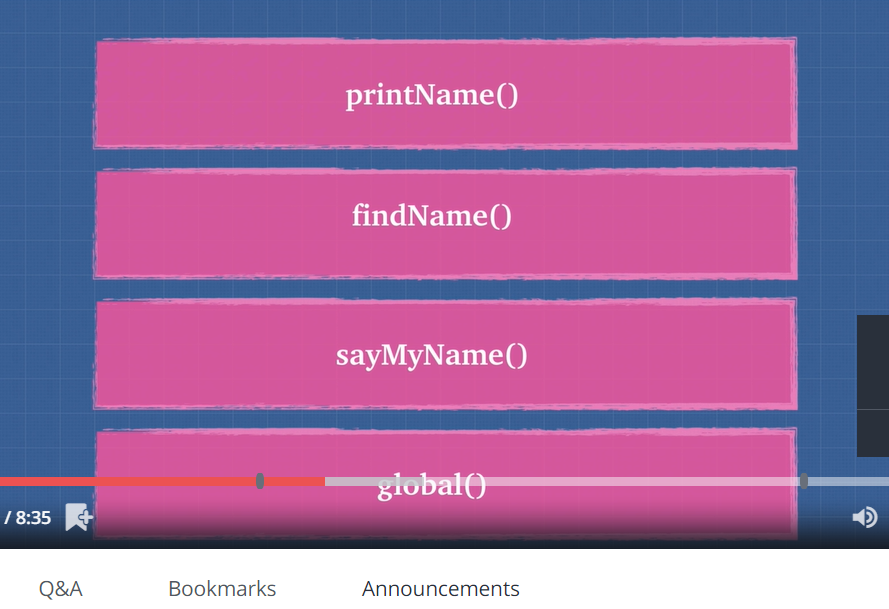
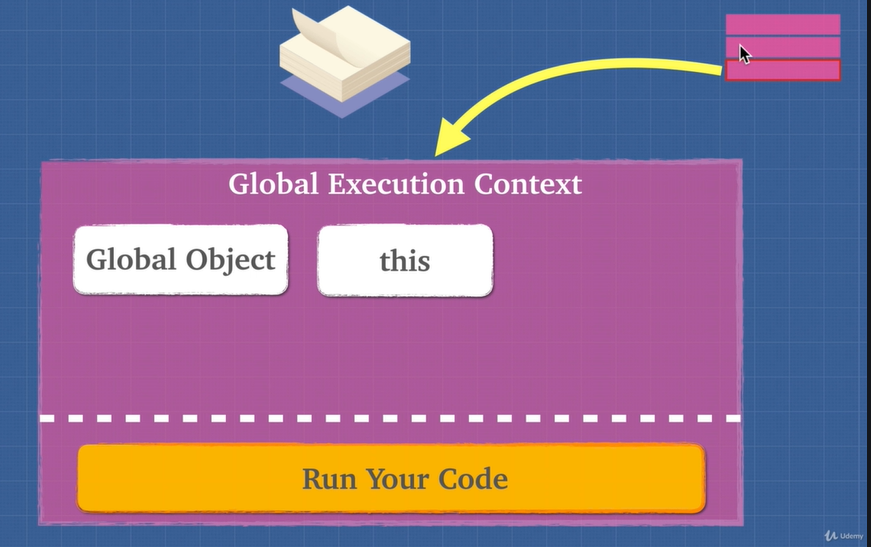
26)Execution Context

When we call a function js creates a separate execution context for that function and add it to call stack.



When we execute a file we create a global execution context, then with different function calls , we also create execution contexts for them. So global execution context is first thing on call stack. Above figure shows that. So when ever code is run inside js, it is run in some execution context.

When we run our file, js creates a global execution context and provides us these 2 objects- global object(in browser it is window) and this object. In starting these 2 objects are equal to each other. even if our file is empty we will get global execution context and with in that we will have these 2 objects. As we execute our functions, we will have different execution contexts.



All global variables and functions are added as properties of this global object, which is window in case of browser. in case of node js, this window object is global. Once we have done this which is called creation phase.

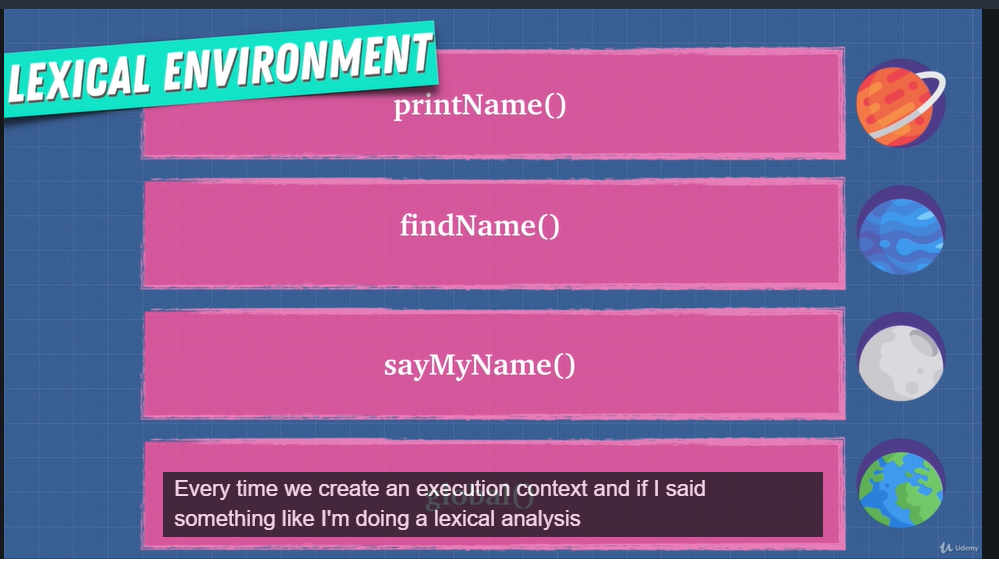
Then we have execution phase, where we run our code.

27)Lexical Envirenment

You might have heard different variations of this word- lexical scope, lexical analysis.

Lexical environment is where you write something. this is how I think about it.

You can think of lexical environment as little universe that gets created each time you create execution context.



If we say compiler is doing lexical analysis then it is checking where words where written and their location, that is what universe they are part of.

Lexical means at compile time where code is written and based on when compiler or interpreter sees our code it will know different things about that code. Lets say function is lexically inside another function. now compiler knowing where it was written can decide and make decision as to where to put things and what actions to take and what function has access to in that word.

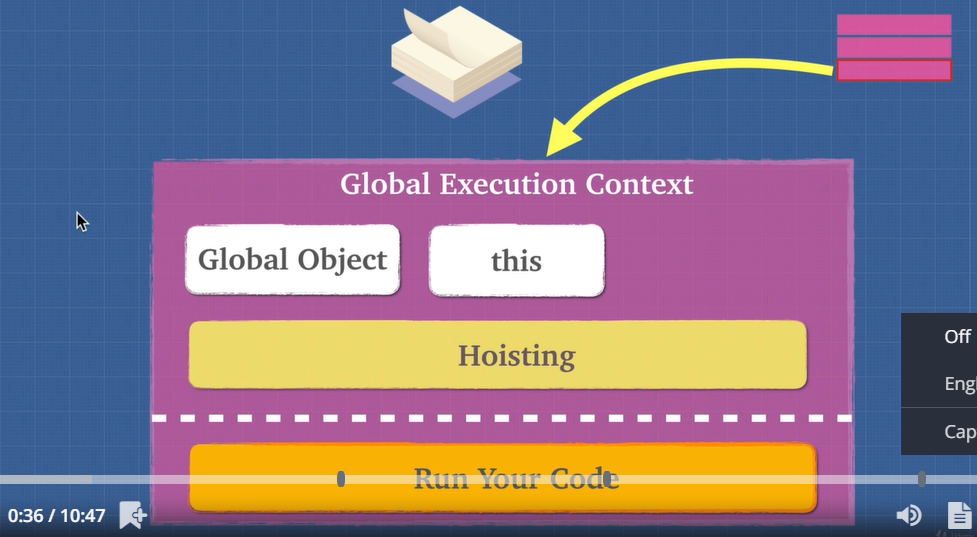
Each time a function is called, we create different lexical environment, different universe. Inside of that universe , we can do different things, we can have different information. These words can communicate with each other in different ways.

So execution context tells you which lexical environment is currently running.

In js where function is written determines which variables it has access to , it not decided by where we call function. later is called dynamic scope. So javascript has lexical scope.

28)Hoisting

In this diagram we have creation phase and we have execution phase. This white line separate them. In creation phase we have one more step, hoisting.

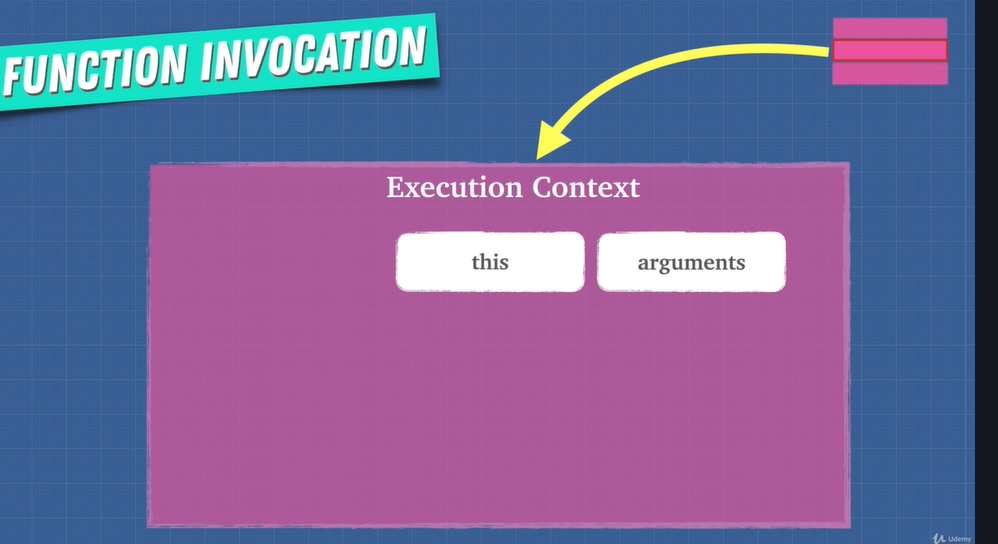


Hoisting is moving variable and function declaration to the top of their respective environment during compilation phase.

So this is what happens- global execution context is created, global object and this object is provided. Then we allocate space in memory for variable and fucntions.

32)function invocation

When we execute a function then new execution context is created. Now in this execution context we get 2 things- this object and arguments object.



Now this argument object is not available in global execution context. But some lecture back we said that we should not use arguments keyword, it is bad for performance. Lets see why that is.

33)Arguments Keywords

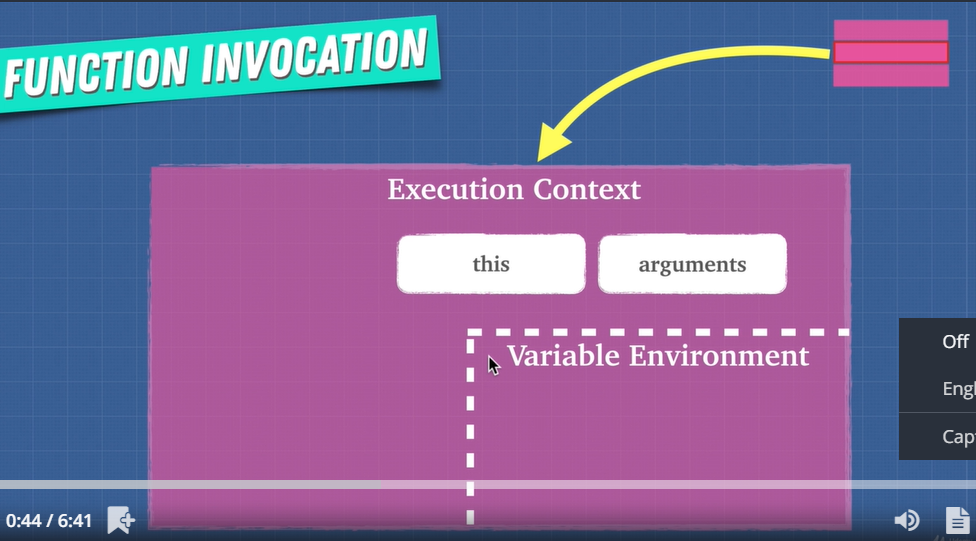
We said that we can do some things with arguments which will make harder of compiler to optimize our code. So in modern js we should avoid using argumnets keywords. Here are alternatives-

1)convert arguments object to array by using Array.from

2)use rest parameters in functions.(see your js notes for rest parameter)

34)Variable environment

so we know that there different execution context is created for each function. what about the variables that we create in function. now in execution context we space for these variables. You can see variable environment I figure. This is space where variables can live in different stack words. Technically they live in javascript engine memory but they need to know how they relate to one another.



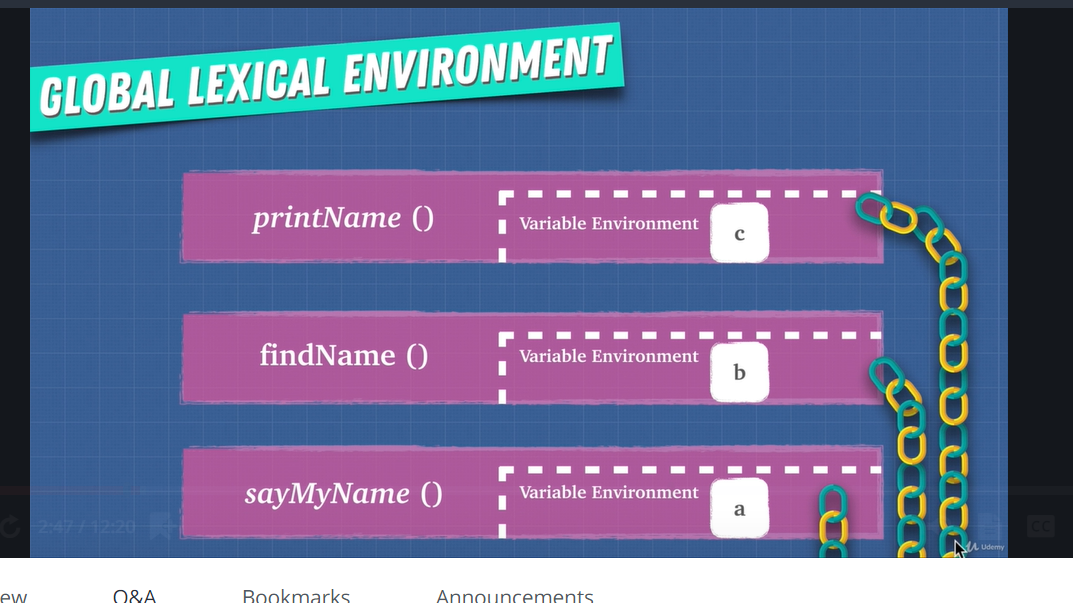
You can consider each execution context as it’s own planet.

Then we saw that we can have variable of same name in different execution contexts(this is created for different functions). Now some of variables may be actually stored in call stack or execution context or we in execution context we have refrence to somewhere in heap where these variables are actually stored. So thing to keep in mind-

Each execution context has its own variable environment.

35)scope chain

There is one more thing about execution context that we have not discussed. This missing things is – each word, i.e each context has link to outside word or link to it’s parent. This outer envirement depends upon where function sits lexically. Here we can see that these execution context has link or chain to outside word, this chain is called scope chain-



This chain gives us access to variables of parent environment. so first js finds a variable in current execution context, if it does not find it then it goes to its parent scope. This process goes until we are in global scope. If even here we cannot find variable then we throw refrence error. This is called scope or static scope or lexical scope.

So even before running code, just by seeing the code we can determine which variables will be accessed where. So compiler looks at our code and makes these scope chain even before it runs our code. This is all possible because we have lexical or static scope.

Scope means where I can access my variable.

Now we said that we shuld not use eval and with operator. they make it hard for engine to optimize code. This is because of issues they have with scope. With eval and with you can change how scope and scope chain works internally in javacsript and that makes things difficult for us. Because we have lexical scope, compiler can just look at code and create scope chian and understand what is going to happen. But if we start changing it around with eval and with, it will trick js engine.

36)[[scope]]

Here we actually saw the scope chain in code.

37)exercise: js is weird

Here we saw-

1)accidental global variables

2) why we cannot use f2 to call function-

Var f1 = function f2() {};

41)IIFE

Here iife was explained. It was also shown how it was used in j query and backbone js.

We will know more about in IIfe in module section.

43)Execrcise:Dynamic Scope vs Lexical Scope

This is not lexically scoped. Common problem in js is that this points to global object

When we want it to point to something else. Now a days we have arrow function to solve this problem. This is arrow fucntions is lexically bound. Second solution is to use call, apply and bind. See code snippets in this.js.

44)call(), apply(), bind()

When we call function by-

a(), this is just a shorthand for – a.call().

See code sniipets in callAndBind.js

See copy notes

46)Bind and currying

See coe sniipets in snippets folder and see notes