Okay, so in the last lecture, we have seen when a new execution context is created, but what about how, exactly, that happens? So let's talk about that right now. Let's start by creating a representation

of an execution context so that we can have a bit more detail about how it actually works. First off, as I mentioned in the previous lecture, we can associate an execution context with an object, which is what we have here, and this object has three properties: the variable object, which will contain function arguments in a variable declaration as well as function declarations; the scope chain, which contains the current variable objects as well as the variable objects of all its parents; and "this" variable that we have already seen in action in the introductory lecture about objects. All right, so how is execution context actually created? Let's start with what we already know. When a function is called, a new execution context is put on top of the execution stack, and this happens in two phases:

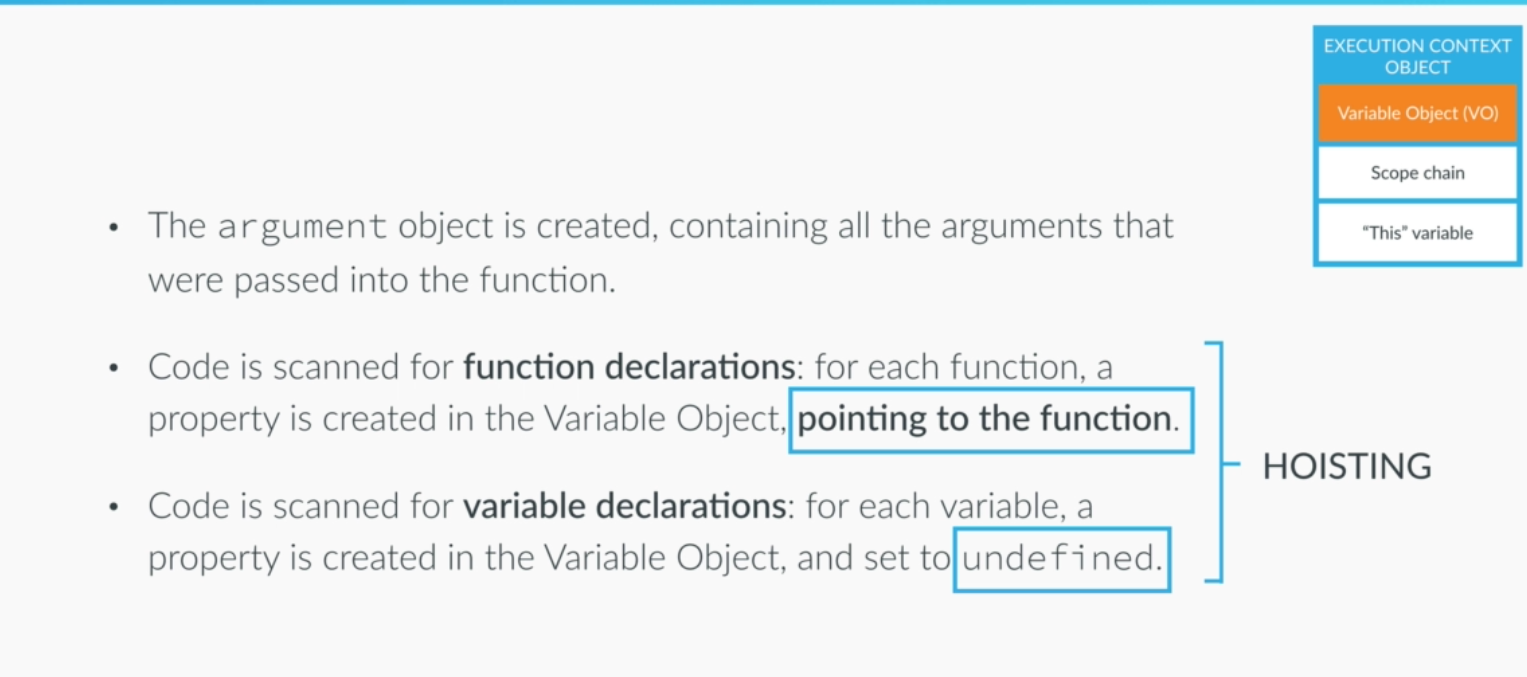
the creation phase and the execution phase. In the creation phase, we first have the creation of the variable object, and we will talk about that in the next slide; then, the creation of the scope chain, and this is a huge topic for itself, so we'll talk about it in the next lecture; and lastly, the "this" variable is determined and set. So basically the properties of the execution context object that we have here on the left side are defined. Then, in the execution phase, the code and the function that generated the current execution context is run line by line, and all the variables are defined. If it's a global context, then it's a global code that is execute it. So let's now look at the creation of the variable object

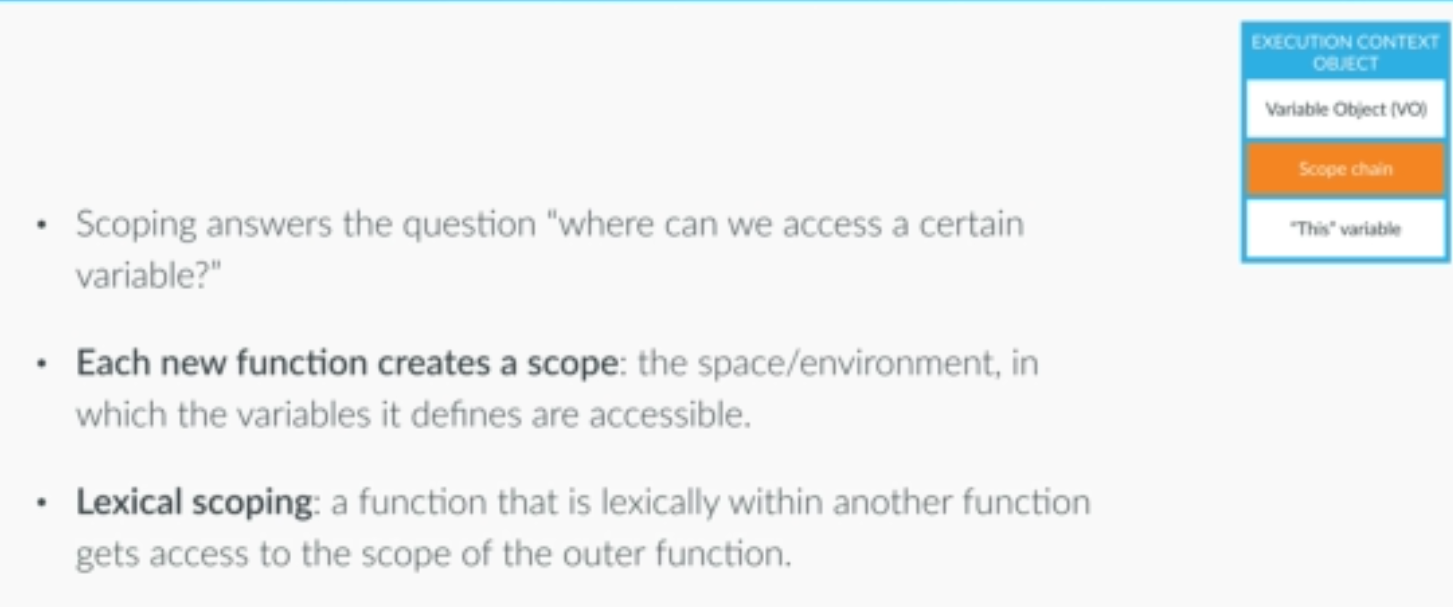
in the creation phase. So this is what happens in the creation of the variable object. The argument object is created, which contains all the arguments that were passed into the function. The code is scanned for function declarations, and for each function a property is created in a variable object pointing to that function. This means that all the functions will be stored inside the variable object,

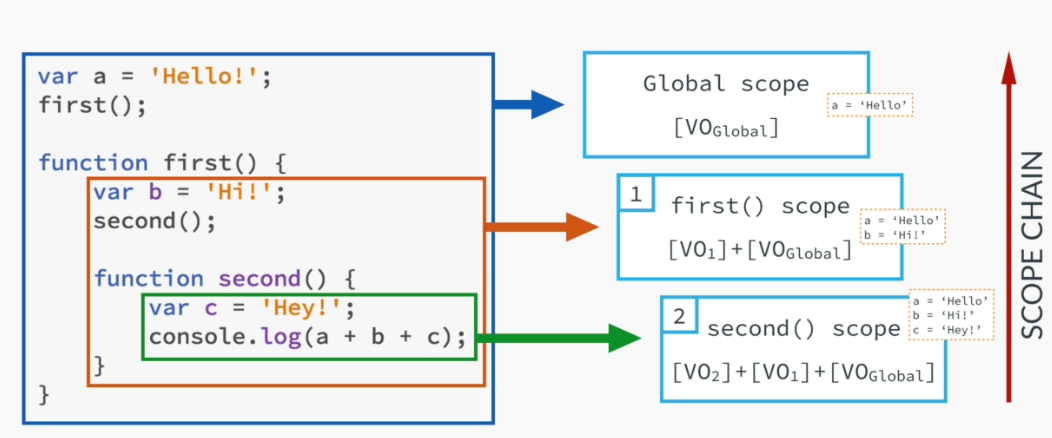
even before the code starts executing, and this is very important for you to know. And lastly, the code is scanned for variable declarations, and for each variable a property is created in a variable object and set to "undefined", and these last two points is what we commonly call "hoisting". Functions and variables are hoisted in JavaScript, which means that 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 up to "undefined" and will only be defined in the execution phase. As you remember from the last slide, the execution phase comes just after the creation phase. Now, hoisting is something that confuses many new JavaScript developers, so it's really important that you understand it right now.

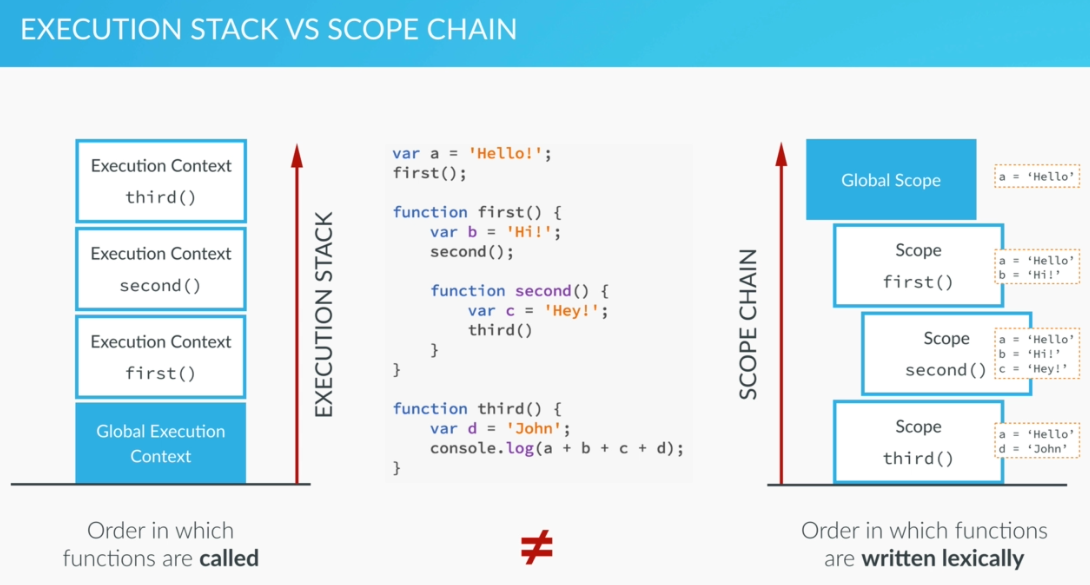
That's why we're going to write some code in the next lecture, so that we can see this in practice.

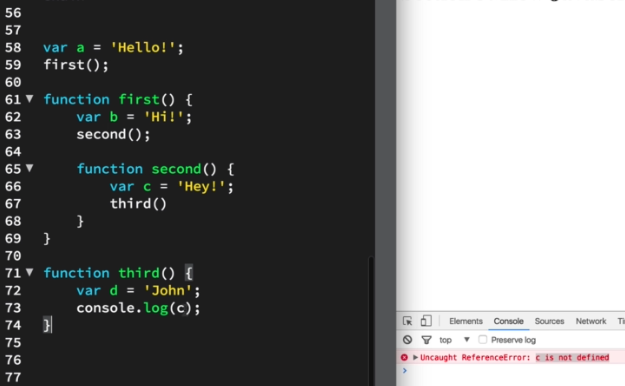
So just to recap all of this once again, each execution context has an object which stores a lot of important data that the function will use while it's running, and this happens even before the code is executed. All right, let's go to the next lecture to see hoisting in practice.











We talked about how execution contexts are created and how that is related to variable and function hoisting. That was the first step of the creation phase. So let's now talk about the second step,

the creation of the scoping chain. So what does scoping even mean? Scoping basically answers the question, "Where can we access a certain variable or function?" In Javascript each function creates a scope, which is a space or an environment in which the variables that it defines are accessible.

In many other programming languages a scope is also created by if blocks, for blocks, or why blocks, but not in Javascript. So here in Javascript the only way that we have to create a new scope is to write a new function and this is something very very important in Javascript. Also, in Javascript we have lexical scoping, and lexical just means where something is written in the code. So it's about the position of somthing within our code. Lexical scoping means that a function that is lexically

within and out of function, which means that it's written inside of another function and gets access to a scope of the outer function, also called the parent function, and with that it also gets access to the variables and functions that the parent function defines. Let's look at an example to get a better idea of this. So here we have some more demo code. In blue we have the global scope, which is a scope

that is accessible from everywhere, from all functions. Like the global execution context, you can think of the global scope as the default scope. So in this scope we have access to the variable a,

and also to the first function. In orange we have the scope of the first function, which defines the variable b and the second function, and this is a local scope, not a global scope. And do you remember what I said about lexical scoping in the last slide? Thanks to lexical scope, this scope has also access to it's parent scope and all the variables functions that it defines. In this case, that's the global scope. Finally, in green, the scope of the second function. In there we have a console.log statement, which attempts to print values of variables a, b, and c. Now, these variables are not defined in that function, only c, but as we already know, a scope has access to the scope of the function in which is sits lexically, so when running this code, the Javascript engine will not find variable b in the current scope, right. So it just goes up and looks in the scope of it's parent function and says, oh, here is b, so let's use that. The same happens for variable a. It does not find it in the current scope and also not in the parent scope, so it goes even more up all the way to the global scope and this is exactly called the scope chain. Only if the Javascript engine does not find a variable anywhere it throws an error and stops execution. Now, it's very important to note that

this does not work backwards. For example, the global scope will never ever have access to the variables b or c unless we return the values from the functions. So locally scoped variables are not visible to their parent scopes. This is why I put that red arrow pointing upwards, because that's the only direction the scope chain works. Now how does this actually work behind the scenes? Remember the execution context object that we had been talking about before? In the creation phase, each execution context object will get exact scope chain, which is basically all the variable objects that an execution context has access to, because, remember the variable object is what stores all the defined variables and functions. So in this example, in the second scope, we have access to the variable object, of the second functions, of the first function, and to global variable object. So what do you think will happen with this code in practice? Let's go back to our text editor and find out. So I have this code already here in my starter code, which is the one that you will have available for download. So here's the exact same code that we had in the slide before. So how exactly do you think this is gonna work? I'll give you a minute to think about it. Alright, so let's see the solution. Great, so the console log actually printed variable a, b, and c, so hey, hi, and hello, right?

Hey, hi, and hello. So why does this work? This works because, of course, the scoping chain

and in this case thanks to the scoping chain, the second function has access to the variables

of the first function and of the global scope. That's because the second function is written inside of the first, which in turn, is written inside of the global scope. That's why we call it lexical scoping.

So another extremely important concept and very much used in practice as well. Alright, so we have seen that the concepts of execution context, scope, and scope chain are very related. Now before moving on, I just want to make sure that you really understand how these are related and where the differences are. So we saw it at the scope chain for each execution context is stored in the execution context object, but I just wanted to make very clear that the execution stack is different from the scope chain. So here we see another simple code example, and we are already know how the execution stack works, right? So the first function gets called, which then calls the second function, which in turn calls the third function and for each of these calls, a new execution context is put on top

of the execution stack. Now, for the scope chain, things are a bit different. We see that the global scope contains the a variable, as well as the first and third functions. Then the scope of the first function contains the second function, and this is how I like to represent it. Now let's remember that the execution stack is the order in which functions are called, but the scope chain is the order in which functions are written in the code, where they are lexically in our code. So the order in which functions are called does not determine the scope of the variables within these functions. What determines the scope of variables is where the functions are written. Since the third function is not in the scope of the second function, of course it cannot access variables b and c defined in the second and first functions. It can only access the global variable a, because the function is written in the global scope. So, it's execution contexts that store the scope chain of each function in the variable object, but they do not have an effect on the scope chain itself. This is important to understand.

So what do you think will happen in this case? Let's go find out. Let's first analyze the code that we have here. So we have, in the global scope, the var a, which is defined as hello, we have the first function here and we have the third function all in the global scope, or in the global execution context. Then, inside of the first function, we have the b variable, we have the second function call,

and the function declaration. We already know that because of hoisting we can call the function before it's written here, right. Okay and then in the third function, we just define variable d and attempt to console log variable c. Alright so I'll give you a minute to think about what's going to happen once I hit the refresh button here in order to run this script. Okay, so I'm going to do it.

I'm going to push the button. Alright and so we have an error, which tell us that c is not defined.

So two things here, the first thing is why can the second function even call the third function?

This buck here came from line 73, right, which means that someone or somewhere, this function was called, which of course, was here. So the third function was called from the second function

and this, once again, is possible because of scoping. The second function has access to the third function because of the scope chain, right? So, the second function has access to the global scape

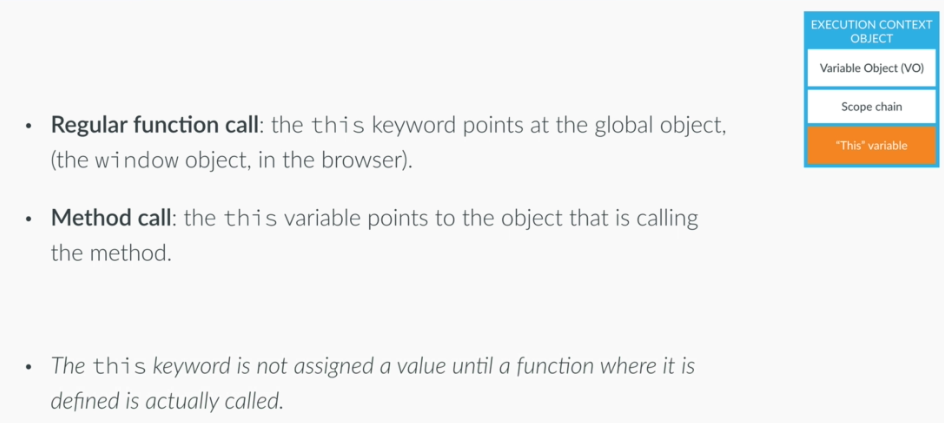
out here, in which the third function also sits lexically. So the second function has the ability to call

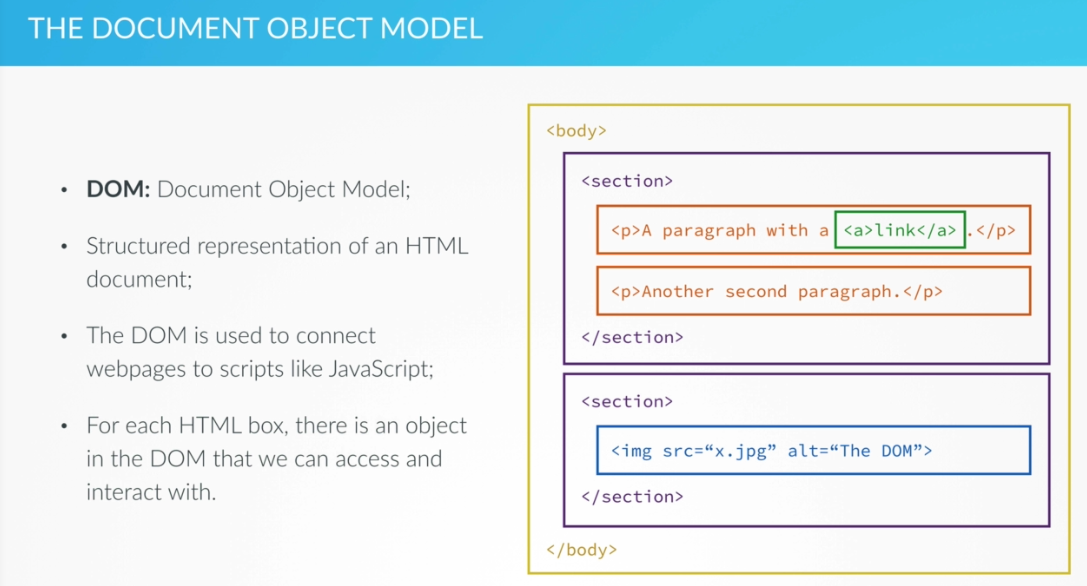
the third function because of the scope chain and there's nothing new here, right? Now the second thing is why do we actually get this error saying that c is not defined, even though it was the second function who called the third function? The second function is the function that defined the c variable and then called the third function, but still the third function cannot access variable c, which it is trying to print here, which as we see resulted in this error. So by now, I'm sure, that it's obvious to you that the third function cannot access variable c because the execution stack is different from the scope chain. To answer the question, who can access the c variable, the order in which the functions

were 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? Try to figure it out by yourself. Did you get it? Okay, so it's variable a and d. Let's just test that. I'm going to console log this one here and then do console.log a plus d. If I run this now, then I have hello John, and hello is a and John is d. So these are the only variables that we have access to

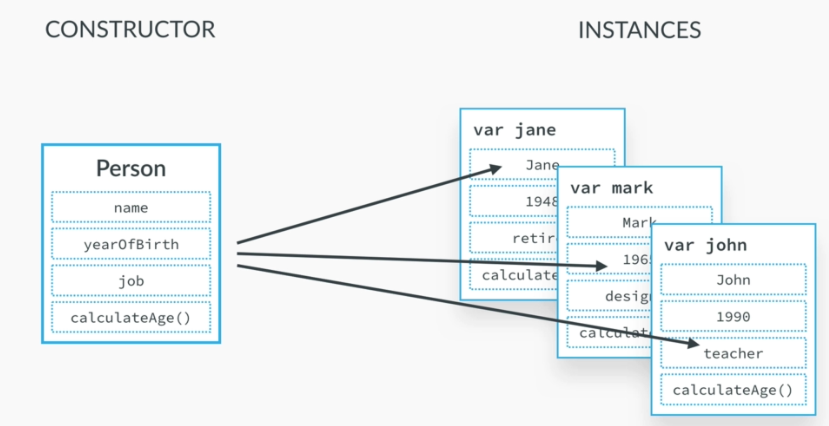
because they are in the scope chain for the third function. Great, this is really cool, right? You are already familiar which a lot of super important and advanced Javascript concepts here and a lot of

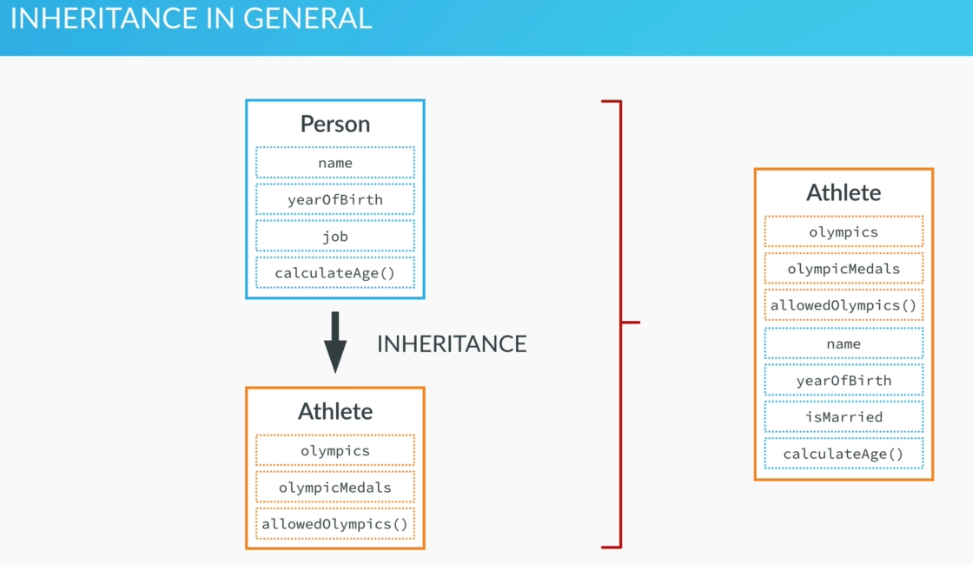
important theory and I can absolutely guarantee you that you already know a lot more than a lot of Javascript beginners. So this is amazing, right? There's only one thing that we have left in this section, and that is the "this" key word. So let's go right to the next lecture where we are going to talk about that. See you there.

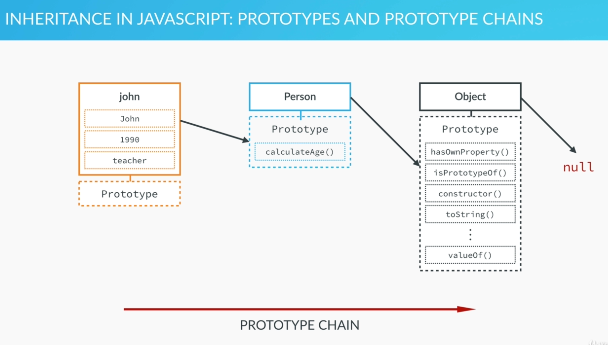












In this lecture we will take closer look at objects in JavaScript. We will talk about object-oriented programming in general, inheritance, and how JavaScript handles objects and inheritance through prototypes. This is probably one of the most important lectures in this course if you really want

to understand JavaScript. So there's a saying in JavaScript which says that, "Everything is an object." And while this is almost correct, it is not 100 percent correct though. Because in JavaScript, we have two big types of values, primitives and objects. And primitives are all the values that have one of the data types that we talked about in the intro lectures. So numbers, strings, booleans, undefined and null values, all of these are primitives, and everything else are objects. And that's why in JavaScript we say that everything is an object. And in fact, even simple things like functions, arrays, or dates are all objects in JavaScript, and that's because in JavaScript objects are not

just for storing a bunch of data, like we have been doing, but for really complex things under the surface. So we have been using objects all along, just without knowing it. So always remember that almost everything is an object in JavaScript, because it's one of the fundamental things that makes JavaScript so unique and different from other popular programming languages. Alright so now that we know how important objects are in JavaScript, let's look at object-oriented programming.

So, in very simple terms object-oriented programming makes heavy use of objects, properties, and methods, and these objects interact with one another to form complex applications. We use objects to store data, structure our code, and keep our code clean. So far, we have only created simple objects holding some data. I remember the John object from the intro lectures and if we had created other persons back then we would simply have written them one-by-one like Mark, Jane et cetera, right? But there is a better way, imagine something like a blueprint from which we can generate as many objects as we want, and we can do that in JavaScript. And right here is an example of that.This is a special Person object that we can basically use as a blueprint to create a lot of Person objects. In other programming languages, this is many times called a class, but in JavaScript we like to call it a constructor or prototype. So based on this constructor, we can create as many instances as we want. So here are our Jane, Mark, and John objects once again,but these were created from the Person constructor. So they are effectively Person instances, and now their name, year of birth,

and job variables are defined, and they all have access to the calculateAge method. So again, the constructor acts like a blueprint and is used to create instances, which of course are also objects.

We will, of course, do all of this with code over the next lectures, this is just to give you a very brief overview. And now let's take this even one step further, with inheritance. In simple terms, inheritance is when one object is based on another object. It's when one object gets access to another object's properties and methods. So, very simple. Back to our persons example, imagine that you also wanted to have an athlete constructor, besides the person constructor, with a couple of different properties and methods. Now, an athlete is also a person, right? There are just some particular properties and methods for an athlete, like for example how many participations in Olympics they have or how many medals they have won. So when we define the blueprint, so the constructor,

for an athlete, then why repeat the same stuff that we have in our Person constructor? An athlete also has a name, year of birth, and job, right? So what we can do is to use inheritance. We make the Athlete object inherit the properties and methods from the Person object, so then the Athlete not only has access to its own properties and methods but also the ones from the Person object. Really cool, right? This allows us to write less code and make more logical programs. So this is a broad overview of how inheritance works in theory, and in all programming languages that have object-oriented futures. So let's now see how JavaScript in particular handles inheritance with our Person and John examples. JavaScript is a prototype-based language, which means that inheritance works by using something called prototypes. In practice, it means that each and every JavaScript

object has a prototype property which makes inheritance possible in JavaScript. So again, inheritance is made possible through the prototype property that every object has. Now all of this may look a bit confusing at first sight, but don't worry, because we will do everything in code later as well. But before doing that, it's super important the we get at least a visual overview of what happens

behind the scenes. Alright, now, how does inheritance actually work? To better understand this, let's go back to our Person example, where the Person object is the constructor and John is one of the instances. Now, if we want John to inherit a method or a property from the Person object we have to add that method or property to the Person's prototype property, just like we see it here.

So in this example, we have the calculateAge method in the Person's prototype property and therefore john inherits the method, and can then call it. And any other object created by the Person constructor would inherit this method as well. So, again, the prototype property of an object

is where we put methods and properties that we want other objects to inherit. What is really important to note here is that the Person prototype is not the prototype of the person itself, but of all instances that I created through the Person blueprint, like, for example, john. So in other words, the Person's prototype property is the prototype of John. And that's not even all, because the Person object itself is an instance of an even bigger constructor, which is the Object object. Yeah, that sounds a bit crazy, but this is actually how it works. Each and every object that we ever create

is an instance of the Object constructor which has a bunch of methods in its prototype property.

And, as you can guess, the Person object inherits these methods and can call them. Plus, the john object also inherits these methods and can also use them. And, with that, we have actually

explained the prototype chain. So, the prototype chain is what makes all this inheritance possible,

and here is how it works. When we try to access a certain method, or property on an object, JavaScript will first try to find that method on that exact object. But if it cannot find it, it will look

in the object's prototype, which is the prototype property of its parent. So it moves up in the prototype chain. If the method is still not there, this continues until there is no more prototype to look at, which is null. Null is the only one that has no prototype, and is therefore the final link in the prototype chain.

And in this case, undefined is returned. This is the reason why, for example, the john object could call the hasOwnProperty method that is stored in the Object Prototype property. And since this is all so crazy important I even took the time to write down a summary for you about the most important things about prototypes,so that you can copy it if you want. So here it goes: Every JavaScript object has a prototype property, which makes inheritance possible in JavaScript. The prototype property of an object is where we put methods and properties that we want other objects to inherit.The Constructor's prototype property is not the prototype of the Constructor itself, but it's the prototype of all instances that are created through it. When a certain method, or property, is called, the search starts in the object itself, and if it cannot be found there, the search moves up on to the object's prototype. This continues until the method is found. And this is the prototype chain. Alright, so this is it about objects, inheritance and prototypes in JavaScript. I promise that we'll now finally go back to our code.

