How is JavaScript code executed?

We already know that it happens in a call stack in the engine but let's dig a bit deeper now. And let's start by supposing that our code was just finished compiling. Just in the way that we learned in the last lecture. So the code is now ready to be executed. What happens then is that a socalled global execution context is created for the toplevel code. And toplevel code is basically code that is not inside any function. So again in the beginning only the code that is outside of functions will be executed. And this makes sense right? Functions should only be executed when they are called. And actually we saw this happening already in our pig game project.

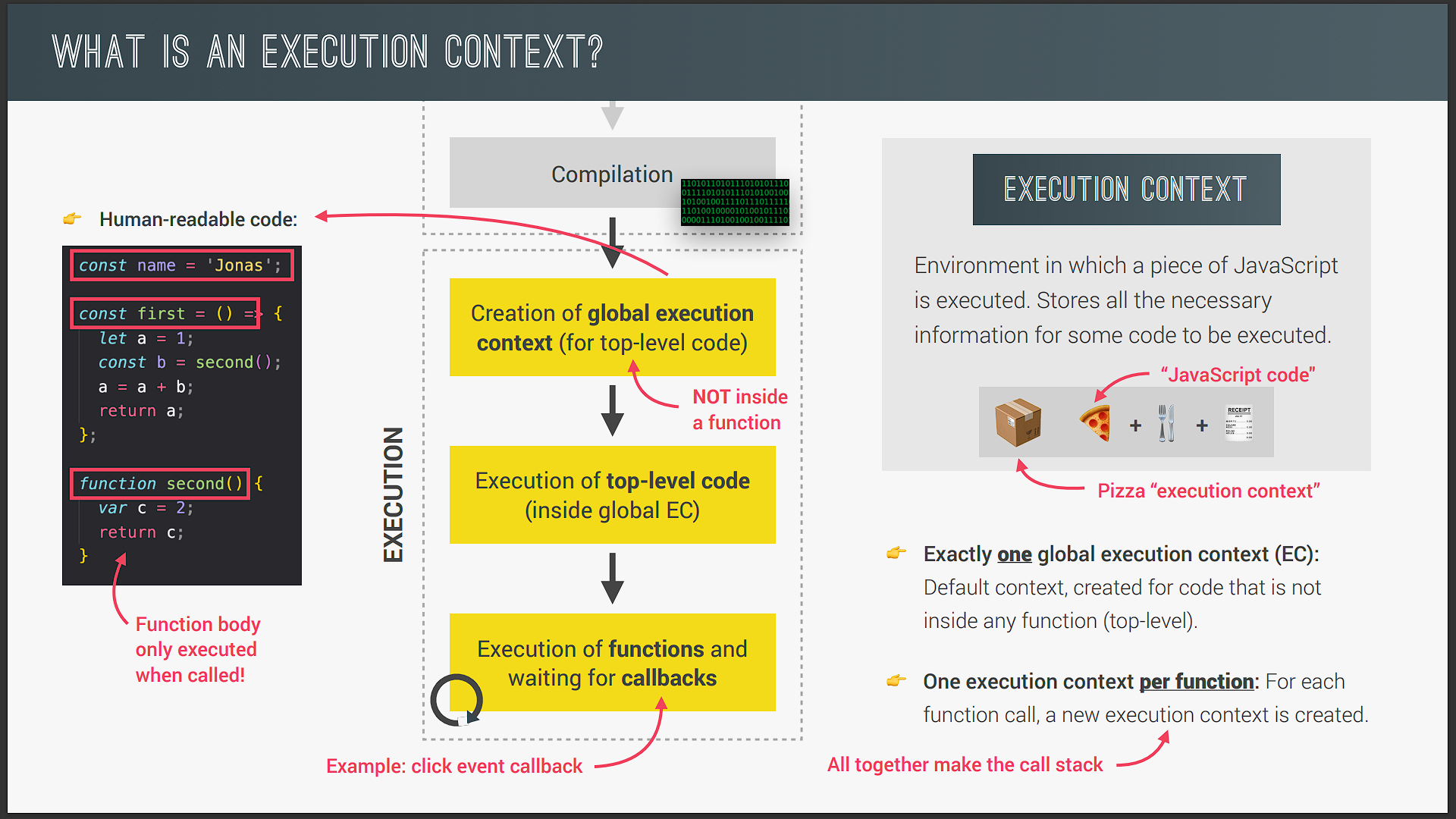
So there we had an init function which initialized our entire project but in order to actually initialize the game the first time that the page loaded w e needed to call that function immediately in our toplevel code. And so that's what I mean here. But anyway we can also see what toplevel code is in this example here. So this name variable declaration is clearly toplevel code right? And therefore it will be executed in the global execution context.

Next we have two functions one expression and one declaration. So these will also be declared so that they can be called later. But the code inside the functions will only be executed when the functions are called. Okay so we know that a global execution context is created for toplevel code. But now what exactly is an execution context? Well an execution context is an abstract concept. But I define it basically as an environment in which a piece of JavaScript is executed. It's like a box that stores all the necessary information for some code to be executed. Such as local variables or arguments passed into a function.

So JavaScript code always runs inside an execution context. And to make this a bit more intuitive let's imagine you order a pizza at a takeaway. So usually that pizza comes in a box right? And it might also come with some other stuff that is necessary for you to eat a pizza such as cutlery or a receipt so that you can actually pay for the pizza before eating it. So in this analogy the pizza is the JavaScript code to be executed and the box is of course the execution context for our pizza. And that's because eating the pizza happens inside the box which is then the environment for eating pizza. The box also contains cutlery and the receipt which are necessary to eat a pizza or in other words to execute the code okay? I hope that made sense and to made the concept of execution context a little bit more clear.

Now in any JavaScript project no matter how large it is there is only ever one global execution context. It's always there as the default context and it's where toplevel code will execute. And speaking of execute now that we have an environment where the toplevel code can be executed it finally is executed. And there is not a lot to say about the execution itself. It's just the computer CPU processing the machine code that it received. Okay and once this first code so the toplevel of code is finished functions finally start to execute as well. And here is how that works.

For each and every function call and you execution context will be created containing all the information that is necessary to run exactly that function. And the same goes for methods of course because they're simply functions attached to objects remember? Anyway all these execution contexts together make up the call stack that I mentioned before. But more on that in a second. Now when all functions are done executing the engine will basically keep waiting for callback functions to arrive so that it can execute these. For example a callback function associated with a click event. And remember that it's the event loop who provides these new callback functions as we learned in the last lecture. All right. So we know now what an execution context is but don't really know what it's made of. So what's inside of it? And so let's find that out next.

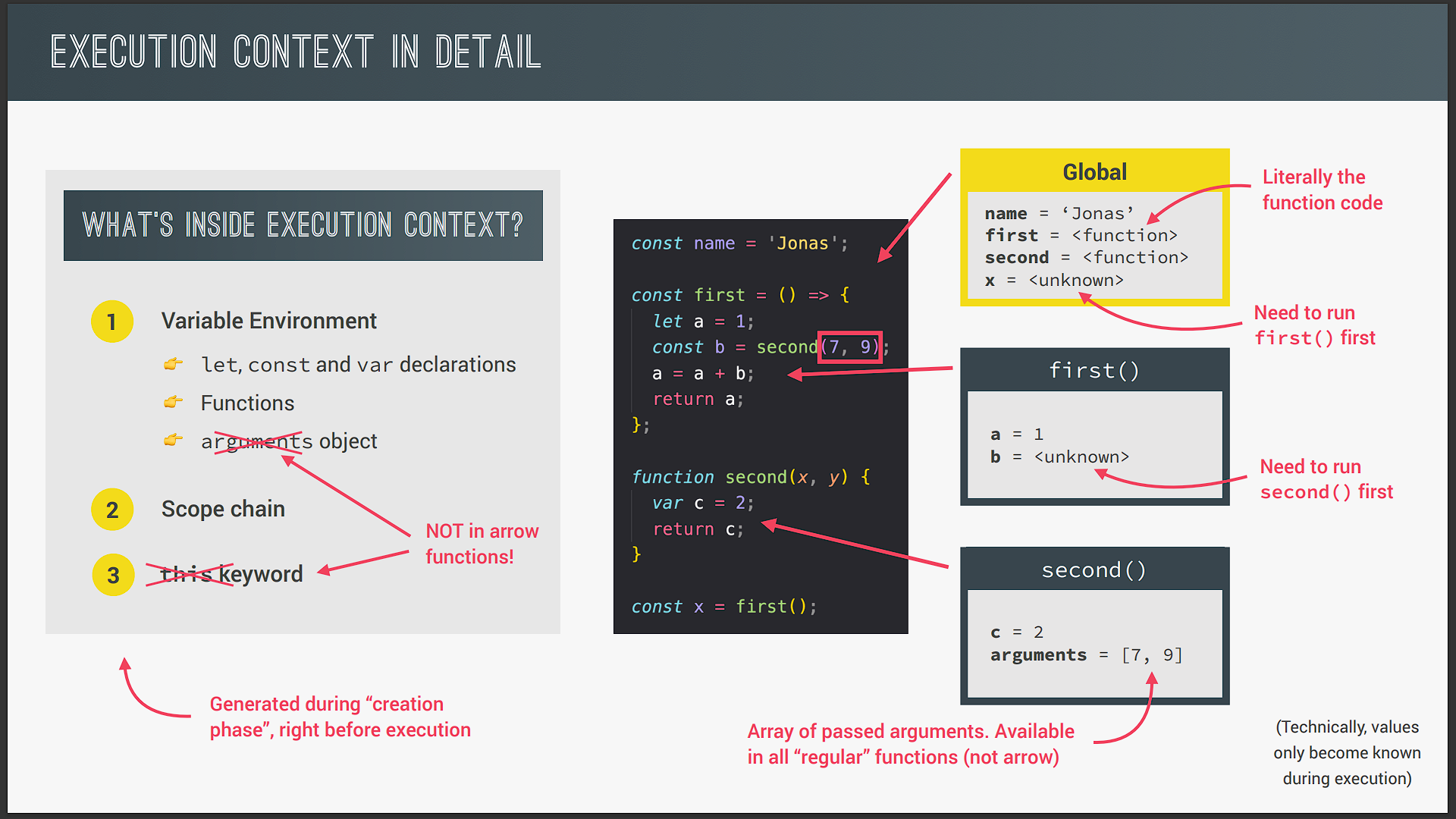


And the first thing that's inside any execution context is a socalled variable environment. In this environment all our variables and function declarations are stored and there is also a special arguments object. This object contains as the name says all the arguments that were passed into the function that the current execution context belongs to. Because remember each function gets its own execution context as soon as the function is called. So basically all the variables that are somehow declared inside a function will end up in its variable environment. However a function can also access variables outside of the function. And we have already seen that in action throughout this course especially in the projects of the previous section. And this works because of something called the scope chain. And we will learn all about scoping and the scope chain later in the section.

But for now what you need to know is that the scope chain basically consists of references to variables that are located outside of the current function. And to keep track of the scope chain it is stored in each execution context. Finally each context also gets a special variable called the this keyword. And once more there is a special lecture just about the this keyword later in the section. Okay. Now the content of the execution context so variable environment scope chain and this keyword is generated in a socalled creation phase. Which happens right before execution.

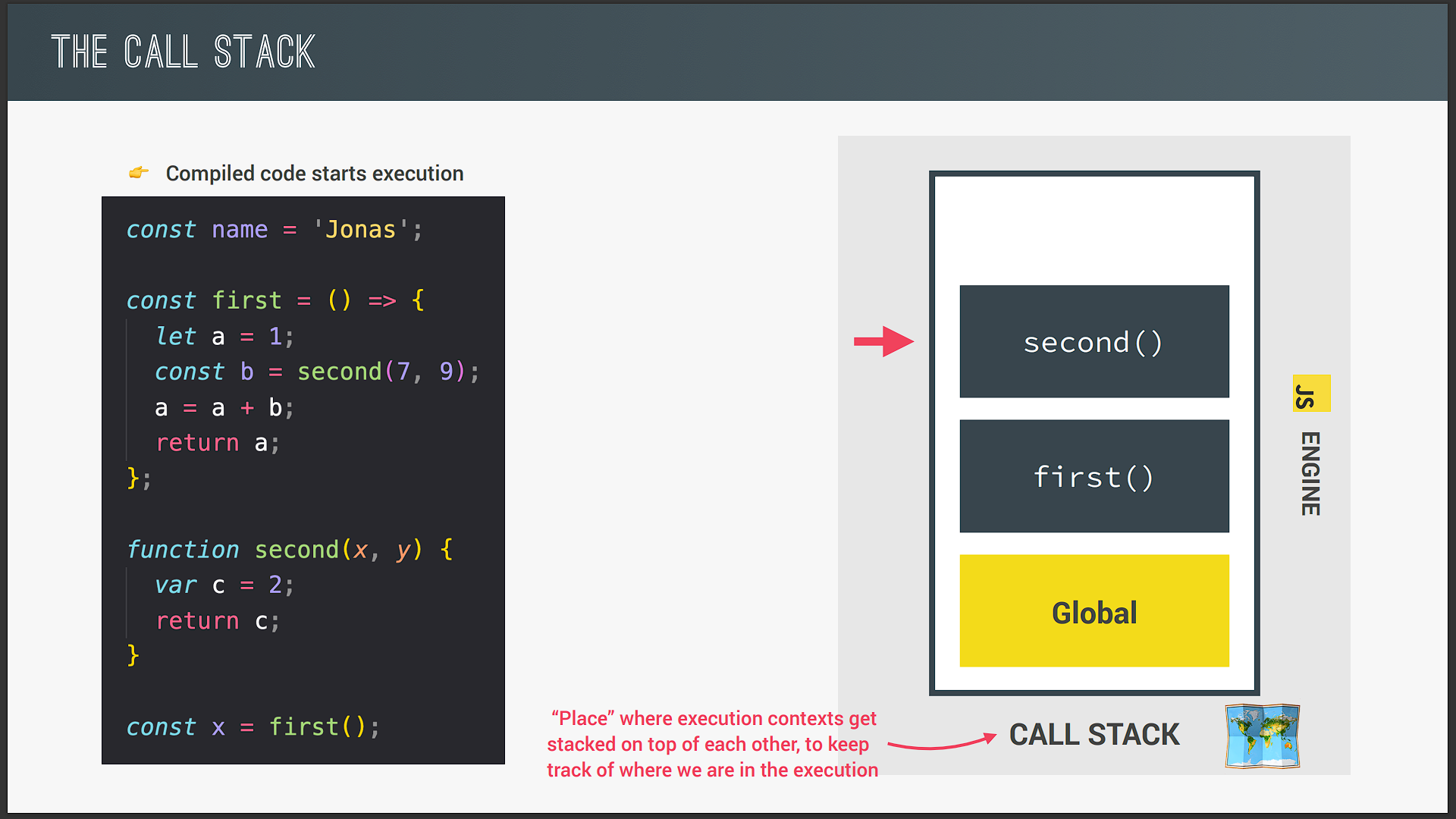
And now just one final but very important detail that we need to keep in mind is that execution contexts belonging to arrow functions do not get their own arguments keyword nor do they get the this keyword okay? So basically arrow functions don't have the arguments object and the this keyword. Instead they can use the arguments object and the this keyword from their closest regular function parent. And this is an extremely important detail to remember about arrow functions and we will come back to it later. So these are the things that are necessary to run each function as well as the code in the toplevel. Now behind the scenes it's actually even more complex but I think we're fine like this aren't we?

And now let's actually try to simulate the creation phase for this code example here. So as you hopefully know by now we will get one global execution context and one for each function. So one for the first function and one for the second function. In the global context we have the name variable declaration the first and second function declarations as well as the X variable declaration. For the functions the variable environment will literally contain all the code of a particular function. Now the value of X is marked as unknown here because this value is the result of the first function that we didn't run yet. But we will simulate this in the next slide.



Now technically none of these values actually become known during the creation phase but only in the execution phase. So this is not % accurate here but it's just to illustrate how these execution contexts work. Okay? So just keep that in mind. Anyway now in the first function we have the a variable set to and the b variable which once again requires a function call in order to become known. Finally the variable environment of the second function contains the C variable set to and since this is a irregular function so not an arrow function it also has the arguments object. And this object is an array which contains all the arguments that were passed into the function when it was called. In this case as you can see that's and . Quite simple right? Well it's simple because this is an extremely small amount of code.

But now imagine there are hundreds of execution contexts for hundreds of functions. How will the engine keep track of the order in which functions we're called? And how will it know where it currently is in the execution? Well that's where the call stack finally comes in.



And remember that the call stack together with the memory heap makes up the JavaScript engine itself. But what actually is the call stack? Well it's basically a place where execution contexts get stacked on top of each other in order to keep track of where we are in the programs execution. So the execution context that is on top of the stack is the one that is currently running. And when it's finished running it will be removed from the stack and execution will go back to the previous execution context.

And using the analogy from before it is as if you bought pizzas with some friends. Each friend has a pizza box and then you put the boxes on top of each other forming a stack in order to keep track which pizza belongs to each friend. Now all this sounds a bit abstract doesn't it? And so to demonstrate how the call stack works let's walk through this code example together so that I can show you exactly what happens. So once the code is compiled toplevel code will start execution. And then as we learned in the beginning of the lecture a global execution context will be created for the toplevel of code right? So this is where all the code outside of any function will be executed.

And what happens with this execution context? That's right it will be put in the call stack. And since this context is now at the top of the stack it is the one where the code is currently being executed. So let's continue now with this execution. So here there is a simple variable declaration. And then the first and the second functions are declared. So nothing fancy but that's just how normal toplevel code gets executed. But then in the last line is where things start to get interesting.

Here we declare the X variable with the value that is gonna be returned from calling the first function. And so let's actually call that function. Now what happens immediately when a function is called? Well it gets its own execution context so that it can run the code that's inside its body. Perfect. And what happens to the context? Well again it is put in the call stack on top of the current context and so it's now the new current execution context. Great. So let's continue.

So we have yet another simple variable declaration here and this variable will of course be defined in the variable environment of the current execution context and not in the global context right? Then right in the next line we have another function call. So let's call that function and move there. And as you guessed a new execution context was created right away for this second function. And once more it is pushed onto the call stack and becomes the new act of context. Now what's important to note here is that the execution of the first function has now been paused okay? So again we are running the second function now and in the meantime no other function is being executed. The first function stopped at this point where the second function was called and will only continue as soon as this second function returns. And it has to work this way because remember JavaScript has only one thread of execution. And so it can only do one thing at a time. Okay? Never forget that.

Now moving to the next line we have a return statement meaning that the function will finish its execution. So what does that mean for the call stack? Well it basically means that the function's execution context will be popped off the stack and disappear from the computer's memory. At least that's what you need to know for now because actually the popped off execution context might keep living in memory. But more about that later in the course. Anyway what happens next is that the previous execution context will now be back to being the active execution context again.

And so let's also go back to where we were before in the code. And I hope that by now you start to see how the call stack really keeps track of the order of execution here. Without the call stack how would the engine know which function was being executed before? It wouldn't know where to go back to right? And that's the beauty of the call stack. It makes this process almost effortless. So I like to use the analogy of the call stack being like a map for the JavaScript engine. Because the call stack ensures that the order of execution never gets lost. Just like a map does at least if you use it correctly. All right. So we returned from the second function or back in the first function where we have this calculation and then finally this first function also returns. And so here the same as before happens.

So the current execution context gets popped off the stack and the previous context is now the current context where code is executed. In this case we're back to the global execution context and the line of code where the first function was first called. So here the return value is finally assigned to X and the execution is finished. Now the program will now actually stay in this state for forever until it is eventually really finished. And that only happens like when we close the browser tab or the browser window. Only when the program is really finished like this is when the global execution context is also popped off the stack. And this is in a nutshell how the call stack works. So hopefully it makes sense now that we say that Java script code runs inside the call stack. And actually it is more accurate to say that code runs inside of execution contexts that are in the stack. But the general point is that code runs in the call stack which is of course true. Great. Next up we will learn some more about the variable environment and how variables are created. So stay tuned for the next video.