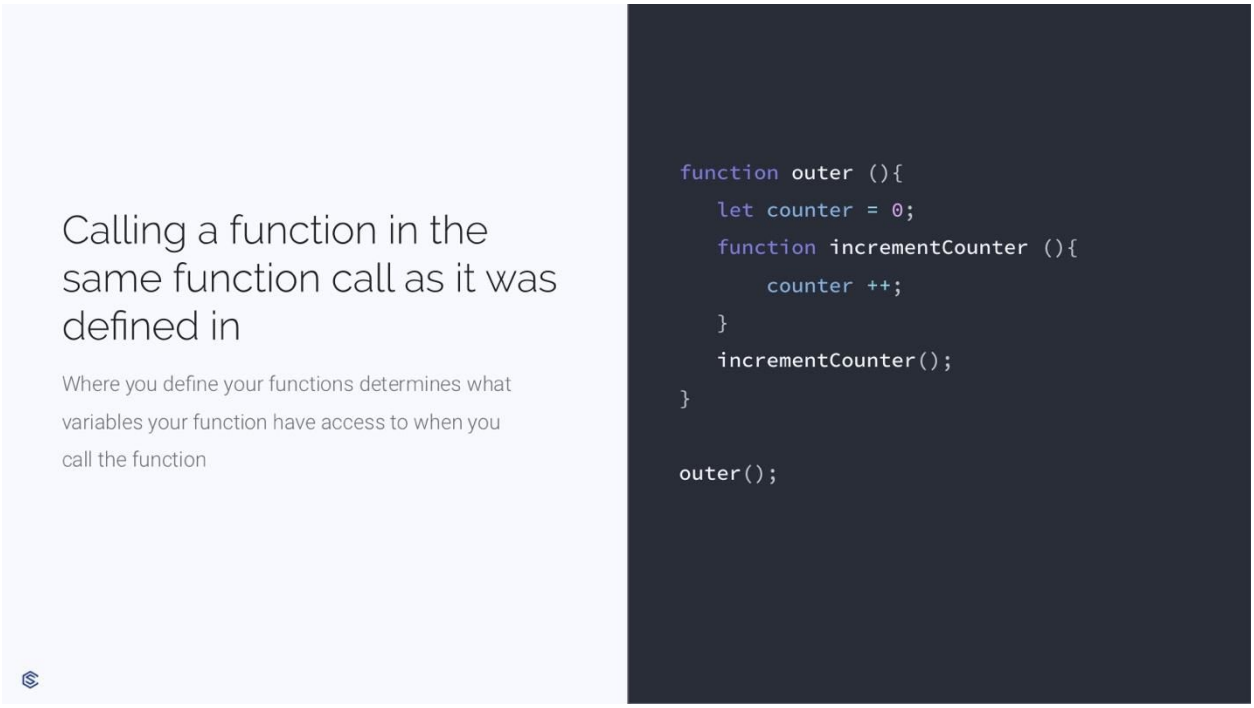
Lecture: 8 Introduction to Scope



Will: Now we move on to the actual meat of this. One final thing we're going to cover before we move onto closure itself, one final thing we're going to cover which is, how do we actually go about executing functions when we execute them directly inside where they were born?

We just saw a moment there defining a function inside another function, not using it, not executing it there, not executing it locally, but returning it out and executing it globally. What if I did execute it locally and what if doing so I was trying to look at or use data inside my locally executed function that wasn't inside that function?

Where would I be allowed to look next? We might take it for granted we're allowed to look next, but it isn't automatic. What I mean by this, have a look here, so, I'm going to call outer. I've declared, saved outer, I'm going to call outer, I'm going to jump in, save counter as zero, save incrementCounter as a function and then the function definition, label the function definition, but then I'm going to call it straight away, and it's going to, by the looks of it, make reference to the simple counter which I've not declared inside of incrementCounter.

So, presumably, I'm going to sort of step out into outer somehow. Now, we will be sure to go, "Oh, of course. Yeah, yeah, I mean, that's coding isn't it?" But why? This is what's called scope. Scope is just a very posh word for, "When I run a line of code inside a function, in global, who knows where, what data will be available to me at that moment that I can refer to? What labels? What names? What namespace, the space for names. What names? What labels can I go look for data in?"

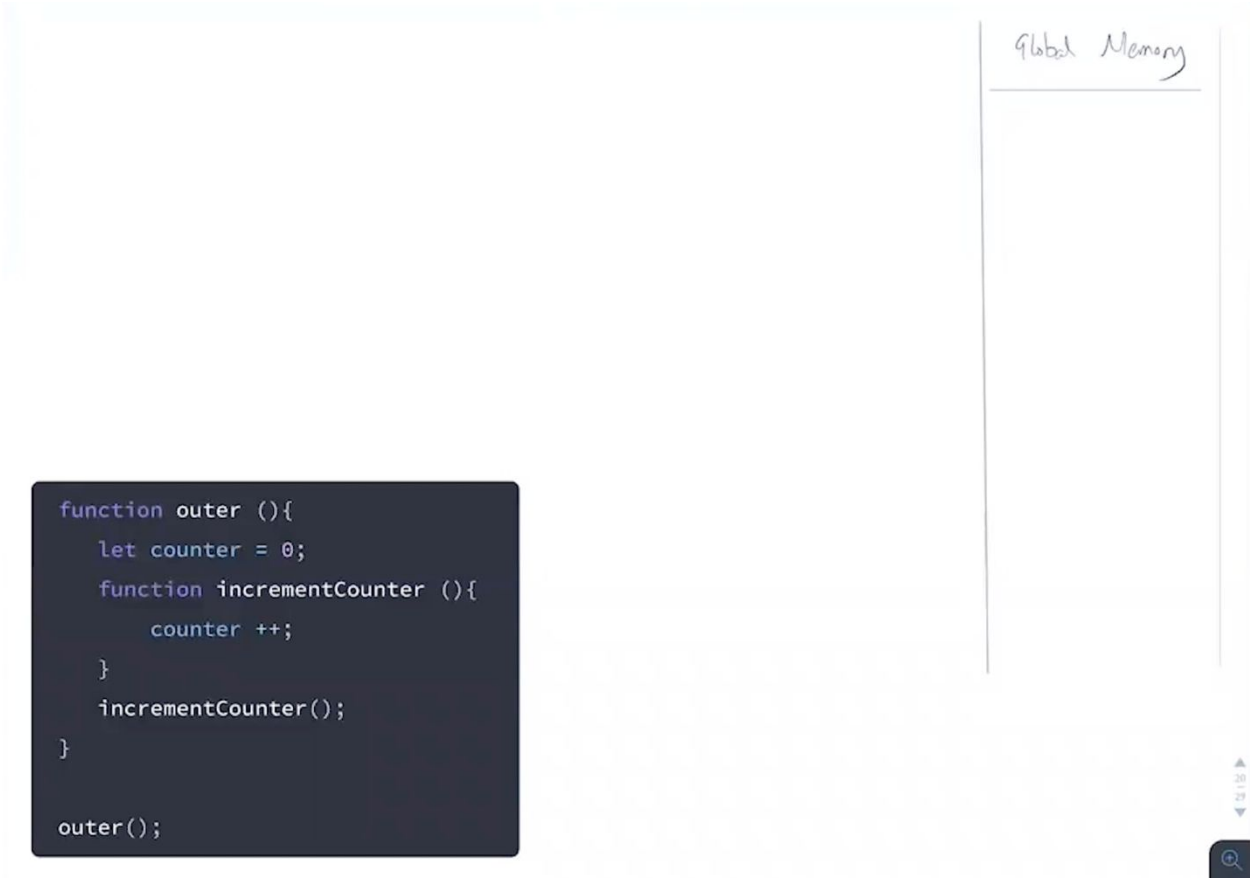
Now we presumably automatically assume that inside that function's execution context memory itself, if we declare something there, yeah, of course we can use that, but if we don't find what we're looking for there, where do we go next and why are we allowed to do that?

Well, that is known as the rules of scoping, and it is a ... every language has specific ... if you do not find data inside the current function, where do you look next? And they're not arbitrary. To us, it's kind of, "Oh, of course, we look at the next function out." But is it? And it's going to turn out that having a precise understanding of where we're allowed to look next actually defines everything we do in JavaScript, and it's actually the bedrock and the heart of what we're doing today.

A very special type of scoping is going to be the answer to how this concept, closure, even exists. All to come, but for now, let's take this code here and just puzzle over when we can't find data in our function's local memory. In this case, it's going to be incrementCounter's local memory as we're running it. Where are we allowed to look next? And honestly, just think about it, why are we even allowed to look anywhere else? That is not automatic.

That's an implementation of a language that we're even allowed to look anywhere else if it's not inside our function itself. And let's figure out actually in JavaScript where are we allowed to look next, because it's not what we think.

Lecture: 9 Call Stack



Will: Here we go. Let's put the code in, let's get up our little global memory. I like to do these things live because, well, I didn't prepare. No, I do things live because, because, why? Oh, because it's more memorable. Natalie, we'll see here what we're actually trying to achieve. We're not trying actually to return or anything in this code. Natalie just raised, should there be return statement? We're not actually trying to return anything, I just want to show you just the question of what data can I look at when I can't find data inside my function itself. That's all we want to see here and really understand that rule.

So let's go, Kate... was that Kate or Natalie that said that? Yeah, Natalie. Kate, jump in. You're up, line 1.

Kate: Okay, we're in global memory, we're going to create a label 'outer' and store in it the function definition.

Will: Which we now know of course puts it in a heap and a label or address. You know, we really don't need to worry about it for now. Excellent. There it is, 'outer', is declared and saved. That was line 1, what was line 2, Kate?

Kate: Line 2 is then outside of the function definition where we actually invoke 'outer'.

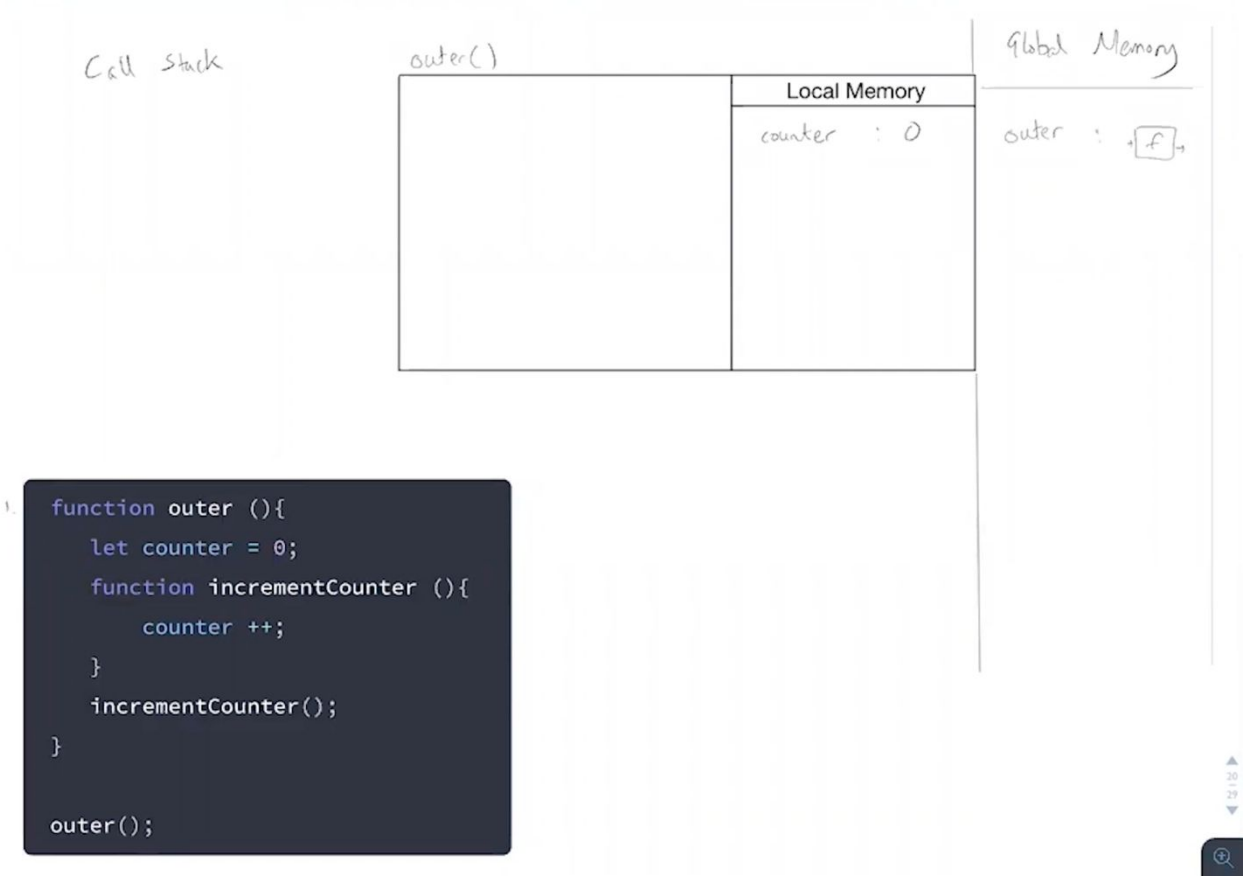
Will: I want to pause there for a second because we should have said this earlier. I may go back and say it and rerecord it, earlier, but that is not obvious. That is the first thing that when I do Hard Parts and have someone with seven years of experience sitting there and say, "Hey, line 1, what are we doing?" "We're declaring the function 'outer'." "Great, line 2, what are we doing?" "We're declaring the counter zero." I've seen people with ten, twelve years of Javascript experience say that, and it's absolutely understandable. I still find myself slipping in that when I'm reading someone else's codebase, codebase being the fancy word for lots of code.

I then read their codebase and I'm like, I'm reading into the function without calling it. Doing so is the most corrupting thing you can do to understanding code, because fundamentally Javascript does not do that. It would never touch all this code here, if that function 'outer' was not invoked at some point with parentheses. Only then does it ever touch the body of a function. Excellent, Kate, you're right. So let's jump into calling 'outer', we create a new, Kate, what?

Kate: Execution context.

Will: There it is, beautiful. Thank you, Kate. And into it we go... Actually let's make this one a bit bigger, let's go a bit bigger. There it is. Just got to move my... Oh, look at that. Fancy. Alright, into it we go, Kate. What's the first thing we put in local memory?

Kate: We're going to create a new variable called counter and assign it the value zero.



Will: Beautiful. Okay, now I want to add something to the mix here called the call stack. It's a final part of major Javascript, the main parts of Javascript, the call stack. The call stack, folk, is Javascript's way of keeping track of what function am I currently running, and where did I start running it. So that when I finish writing this function, I can go back to where I was, and that's best represented... You could store data in all sorts of formats on a computer, you can store it as a list, where you know that position seven of that list has this number. You could store it as an unordered list, where you're not actually bothered the order of the list.

So we have all different ways of storing it. One is called a stack, which is like a stack of plates. You store the data, but you can't grab a random piece from the stack. Just like a stack of plates, you can't grab a random plate from the bottom of the stack, the whole thing would... It's not a thing, you just don't do that. Instead you could add a plate on top and here's the key piece of information, when you take it off, the same plate that was there before is still there.

Well that's just like running a function. You start running a function and when Javascript finishes running that function, it takes that function away and where you triggered running that function is back where you were then. Where you were when you started running that function is where you end up back at. Just like a stack, where you added the plate, you remove the plate, the top plate's still there. Top plate, add a plate on top,

remove it, top plate is still there. Same thing when you run a function, you run a function maybe within global, global execution context they call it, and they start running your function, you create a local execution context to run 'outer'. When you finish running 'outer', you take 'outer's' execution context off the stack, and we might obviously know where to go back to, it goes back to global.

But Javascript doesn't automatically, it has to track that. So you take 'outer' off the top of the stack of functions being called, the call stack. I actually listen to myself saying the word call stack and realized it does just sound to American ears like I'm saying C-O-O-L. I can't do it... Hmm... Oh no, that was horrible. The cool... Car... Okay, this is embarrassing. We're going to cut this piece. The cool stack. I listened to it and I'm like, "Why am I saying it's cool?" I genuinely listened to it for a second and thought, why am I saying it's cool?

I said in a talk, in the talk I went, "So this session's going to be call and response." For a moment I thought, "Did I really just say this session's going to be cool, C-O-O-L?" Even I couldn't believe I would have said something so inane, but I hadn't. I said call, C-A... Anyway. Alright, so this stack of, it's a pretty cool stack of... Kate, wise words indeed. So this stack of calls to functions will keep track of where I am in my code right now, actually, just like a stack of plates it's whatever's top of my call stack is where I'm at right now.

And when I finish running that function that I'm currently in, take it away, they call it pop it off the call stack, and whatever's left is where I go back to. Let's see it in action. Alright, so we start running 'outer'... So here's our call stack. Actually, folk, it always starts with global. Think of all our code as wrapped up in a function called global, and we just started running it. So we're executing the overall code, we create global execution context. Within which we saved 'outer', there it is, but then within which we hit the running of 'outer'. Jim, we start running 'outer', what's going to happen to this damn call stack that Javascript uses to keep track of where we are in that code?

Jim: We're going to add 'outer' to the call stack, right above global.

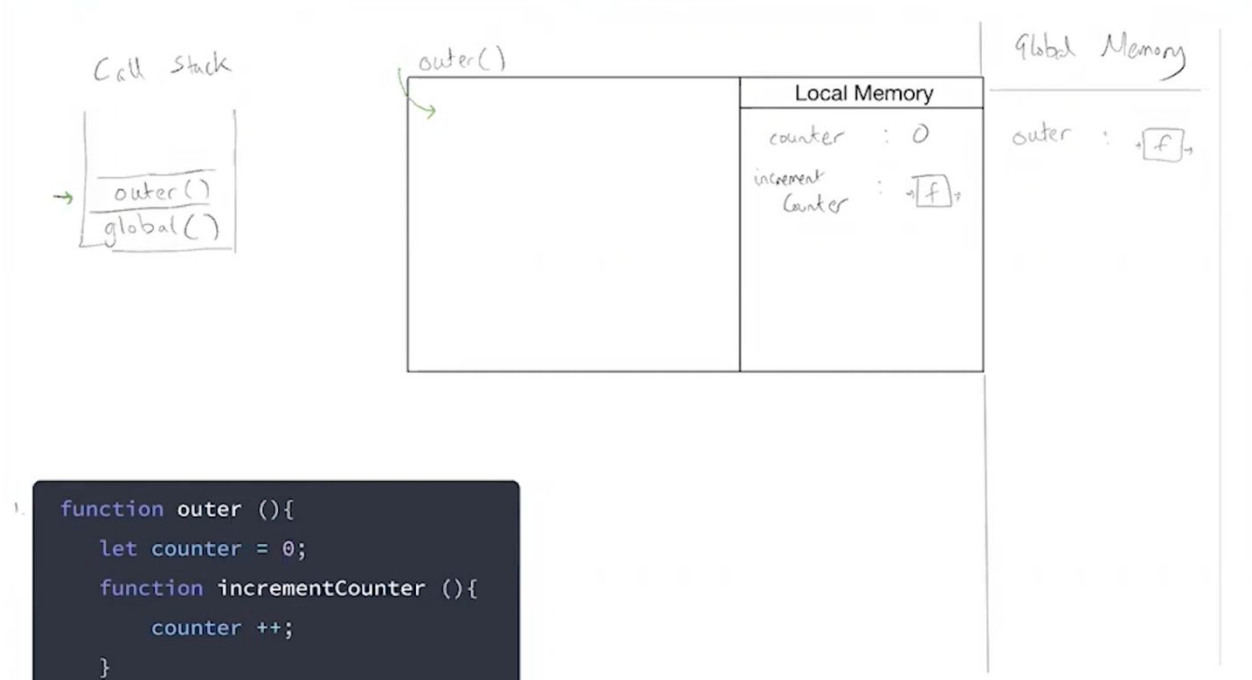
Will: Absolutely. There it is. It's top of the call stack, that means that if I do it in green, top of the call stack, that means we've entered the 'outer' execution context. Into it we go. What, Jim, is the first thing in my local memory? Oh, it's counter is zero. What is the next thing in my local memory, Jim?

Jim: We are declaring the function incrementCounter.

Will: There it is.

Jim: And we're giving a little box with an F and two arrows.

Will: Thanks, Jim.




```
function outer (){
   let counter = 0;
   function incrementCounter (){
       counter ++;
   }
   incrementCounter();
}

outer();
```

Jim: Like that.

Will: Thanks a lot, Jim. Very nice. Now, are we going to return it? No. What are we going to do instead, Jim?

Jim: We're going to invoke it.

Will: We're going to invoke it. We're going to execute it from inside of the execution context of 'outer'. So, Jim, we're going to create a new... What, Jim?

Jim: Execution context. Sorry, I was holding up my sign again. I was going for a laugh. We're going to create a new execution context.

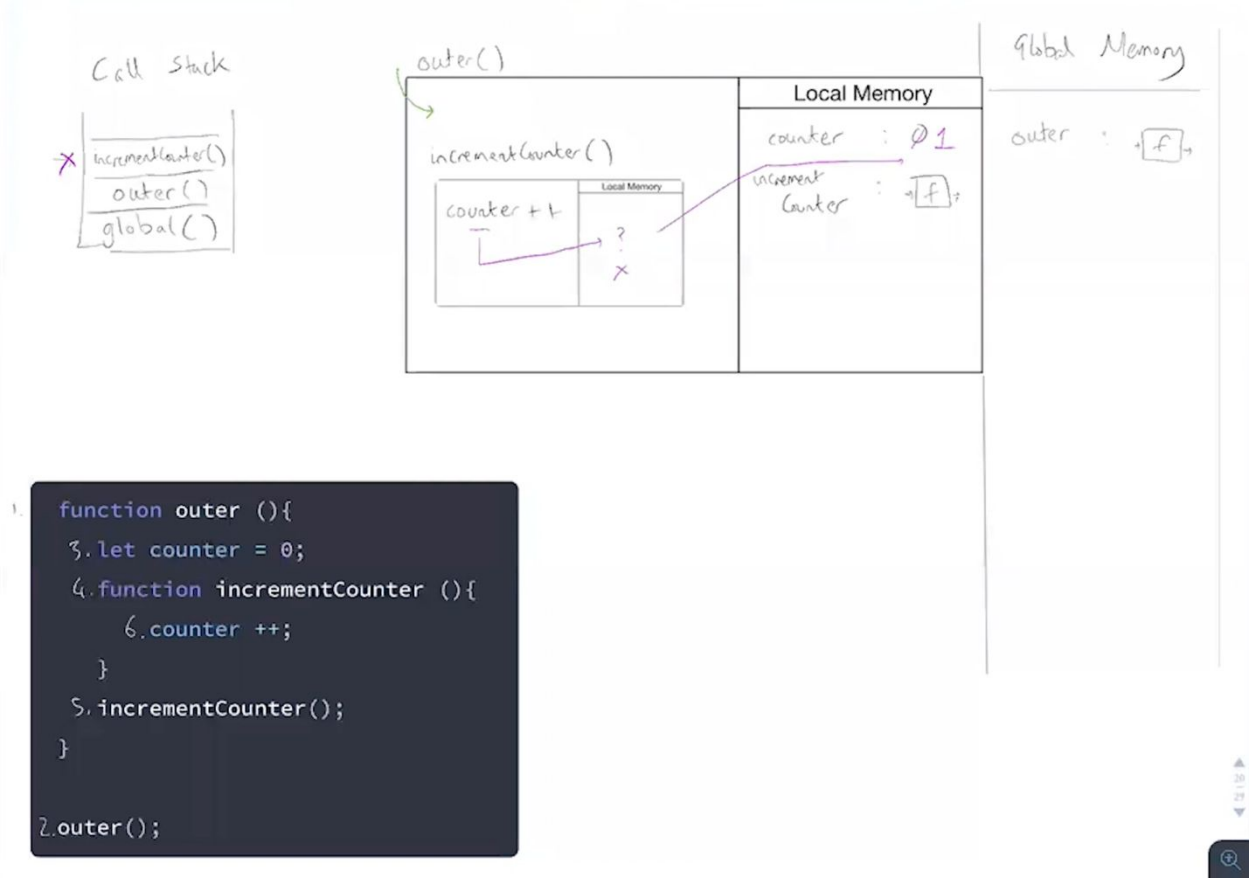
Will: There it is. Beautiful, and in terms of our call stack, Natalie, what happens to our call stack at this point?

Natalie: We are going to push on the function incrementCounter.

Will: Absolutely, and notice already folk, so incrementCounter's now at the top of our call stack. There it is. And I suppose we would have finished running incrementCounter, we'd go back out to 'outer', because we pop off incrementCounter in the call stack and

'outer' would be the thing left in the layer below. Beautiful, excellent, so we go into incrementCounter, and we hit what line, Kate?

Kate: Oh, it will incrementCounter.



Will: Ah-hah (affirmative), it will hit the line counter ++. This is not a simple line, it seems pretty simple, I'm going to use for my purple pen. I tend to use a green pen for this one but I'm going to use my purple pen for Javascript's lookup process. Elaina, counter, where does Javascript always look first for counter?

Elaina: In the local memory of incrementCounter

Will: In the local memory of incrementCounter. Is counter declared inside of incrementCounter, is it in this local memory?

Elaina: No, it's not.

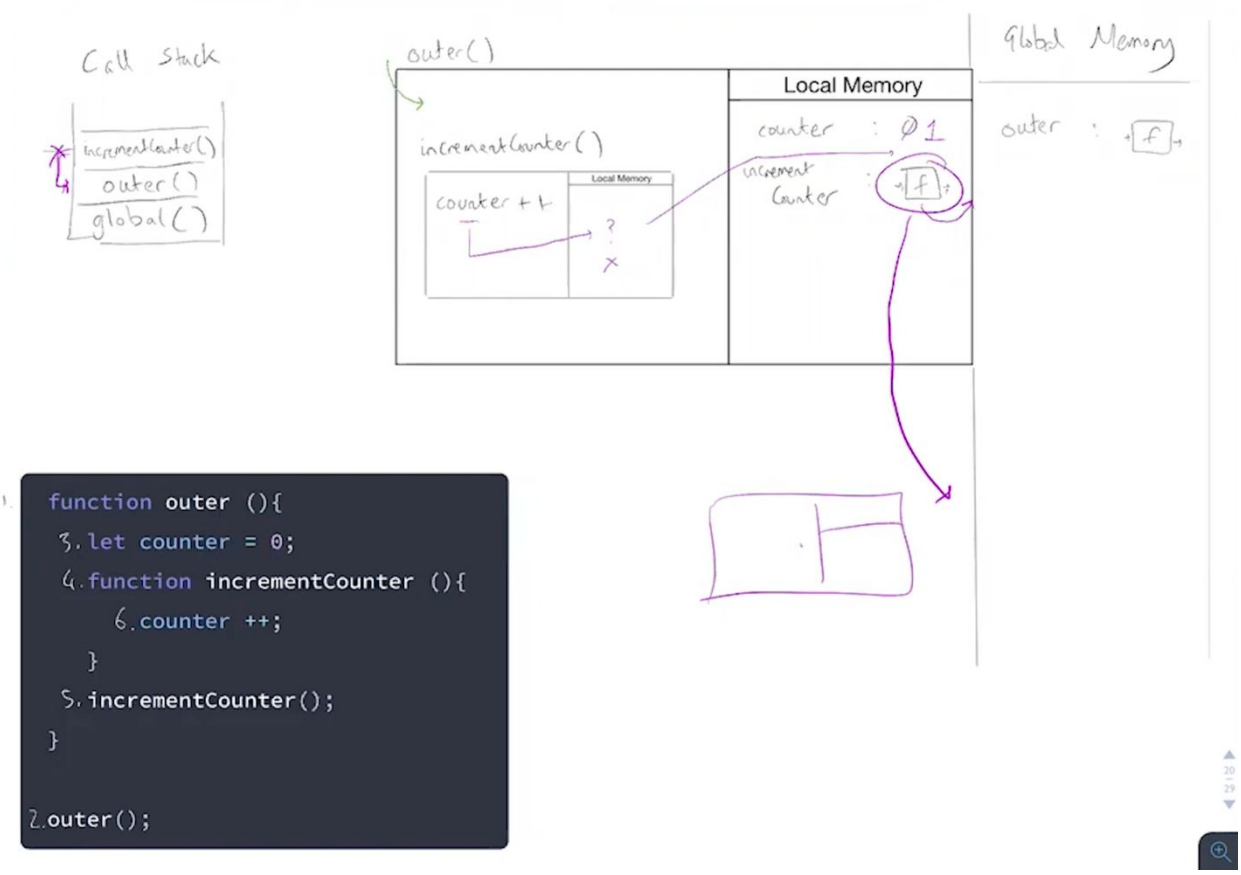
Will: It is not. Okay, so not there. Everything we suggest, where do I look next, Elaina?

Elaina: In the local memory of 'outer'.

Will: Of 'outer', one layer out, where I find...

Elaina: Yes we are...

Will: No problem. We all knew this. I don't know why we took so long on it, we all knew this. So I guess what would happen there is we worked out our call stack, because we called incrementCounter inside of 'outer', without it going to 'outer' to find counter... Wrong. No. Absolutely not, but every developer in the world thinks this is what's happening. We're going to see that this is absolutely not what happened. Well, maybe it is. Sorry, I should have left that more ambiguous. Maybe it is! We don't know. We'll find out, we're not sure right now. Right now it could one of two things, right now it could be that we called increment, we ran incrementCounter inside of 'outer' and therefore because we ran, in other words incrementCounter is above count 'outer', in our call stack, we're running incrementCounter inside of 'outer'. We're allowed to look out to 'outer' next, to global next. Or, or... It could be the fact that we saved, declared, stored incrementCounter inside of the running of 'outer'. That means when we then run incrementCounter, we somehow had a link to this local memory from when we stored incrementCounter. So we're allowed to look and find counter in 'outer', and increment it to one. Which could it be? Who knows. Most mysterious. Right now we don't know, ignore that I just said it's not the fact that we ran incrementCounter inside of 'outer', right? It could be either, it could be either.



The only way we're going to figure out which it is, I'm going to tell you, is if we were take this function... Not the label, the function... Return it out, and then execute it out here. Such that it's being executed in global, and then we can check, is it where I saved the function? Which determines what data's available, or where I'm executing it. By the way, if it's where I saved it, how would the data that was around me from when I saved the function even be available?

Because this execution context here of 'outer', is going to be deleted. Hmm, most intriguing. We're going to jump straight to that right now.

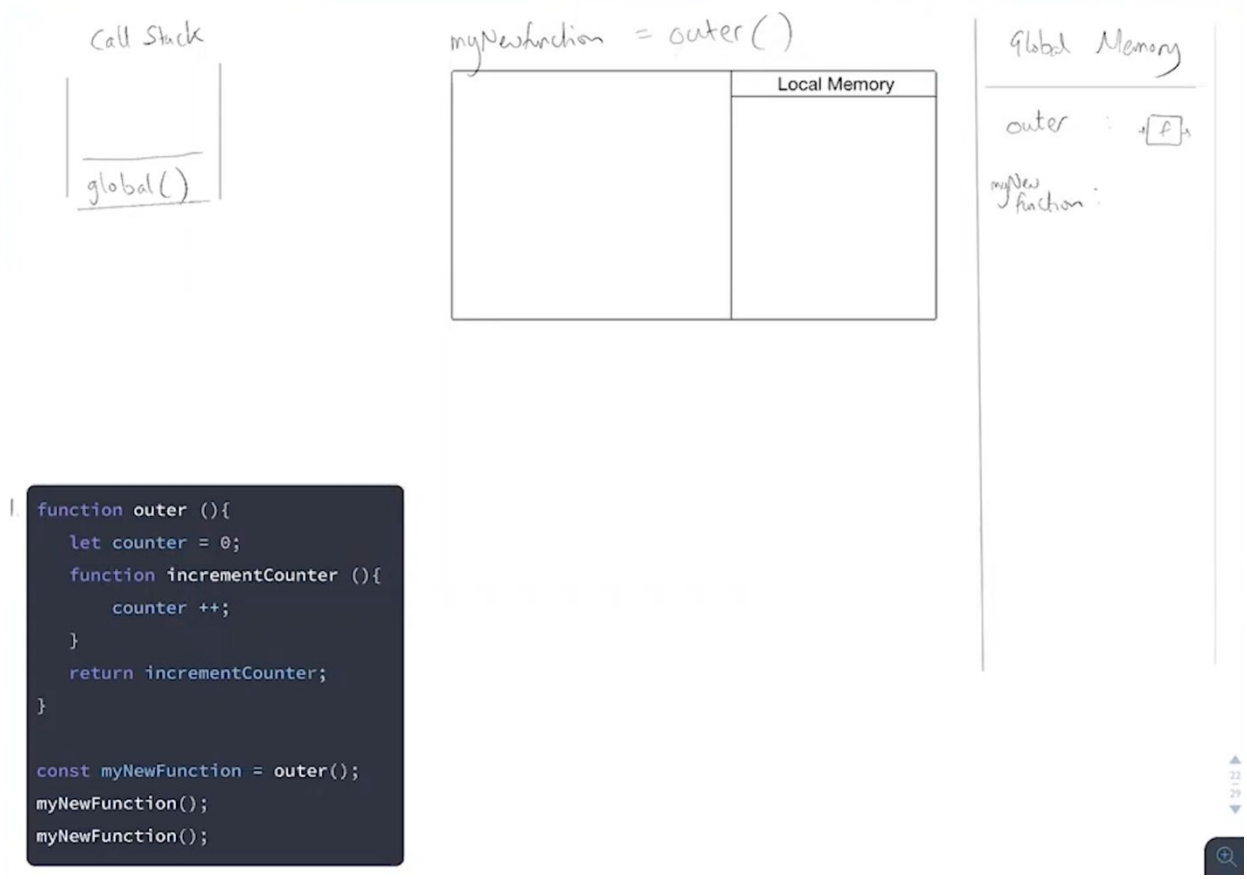
Calling a function outside of the function call in which it was defined, what I mean by that is we defined incrementCounter inside the running of... Well, what? Kate, we defined incrementCounter, the same here. We defined incrementCounter, Kate, inside the execution of what function?

Kate: 'Outer'.

Will: 'Outer', well done. We're going to call incrementCounter, or the functionality saved to it, outside of 'outer'. Look at this, we're going to save 'outer', run it and inside of 'outer', create counter zero, create incrementCounter and then rather than running incrementCounter this time, we are instead going to return it into my new function. Which we're then going to run in global, my new function in global. Already, I am extremely concerned by what's going to happen here. If we get this piece down, this is it. We're there.

Alright, folk. Here we go, let's just into our all important code. We've made massive headway, we're now jumping into the code that really matters. All our foundational pieces are in place, and we are good to go.

Lecture: 10 Closure Functions with Permanent Memories and the Backpack



Will: All our foundational pieces are in place, and we are good to go. Here it is. There's our code. Let's make it a little bit smaller, and here is our global memory. Beautiful. Alright, line 1, Elena, take it away my friend. What are we doing in line 1?

Elena: We are declaring the function outer in global memory.

Will: There it is, well done.

Elena: Function definition.

Will: Yes. Next line Elena.

Elena: The next line we declare a const myNewFunction.

Will: There it is.

Elena: For a millisecond, it's uninitialized.

Will: Uninitialized. While we go, or while JavaScript goes and does what?

Elena: We are evoking the function outer.

Will: Alright, let's just get our call stacks set up this time as well. There it is. There it is. Look at that, my new straight line feature. Sort of straight, not so straight, sort of straight line, maybe, and by the way, we always start off with the global execution context. That's big, this big old one here, where we start running all of our code together. Okay, and then we saved outer, we declare myNewFunction, and now we declare myNewFunction and it's going to be the output of calling outer. So we need to go and run outer's code, which means everyone together, unmute. A moment. This is going to be an invigorating moment. Invigorating moment.

Jim: Execution-

Will: Jim?

All Parties: (Laughter)

Jim: Sorry.

Will: Jim, get a hold of yourself. Three, Two, one a brand new-

All Parties: Execution context!

Will: It seems fun, others online are already saying, well that was funny the first two times, at this point. Hey, online later viewers, this is what keeps you engaged. We're about to hit the most important part, already you were tuning out, you were thinking, why are the pink flowers there? It's a very strange decision. I agree, that was a mistake, we recognize it was a mistake. We should not have done the pink flowers.

But, this brought you back in, this re-engaged you in the all important material. There it is, there's our execution context, into it we go. Into it we go. Line 1 inside of it, Natalie you're up, line 1 inside of. So you've done line 1, sorry it was calling outer. Line 1 was declaring outer. Line 2 was declaring myNewFunction and saying its output is going to be output of outer. So let's go run outer.

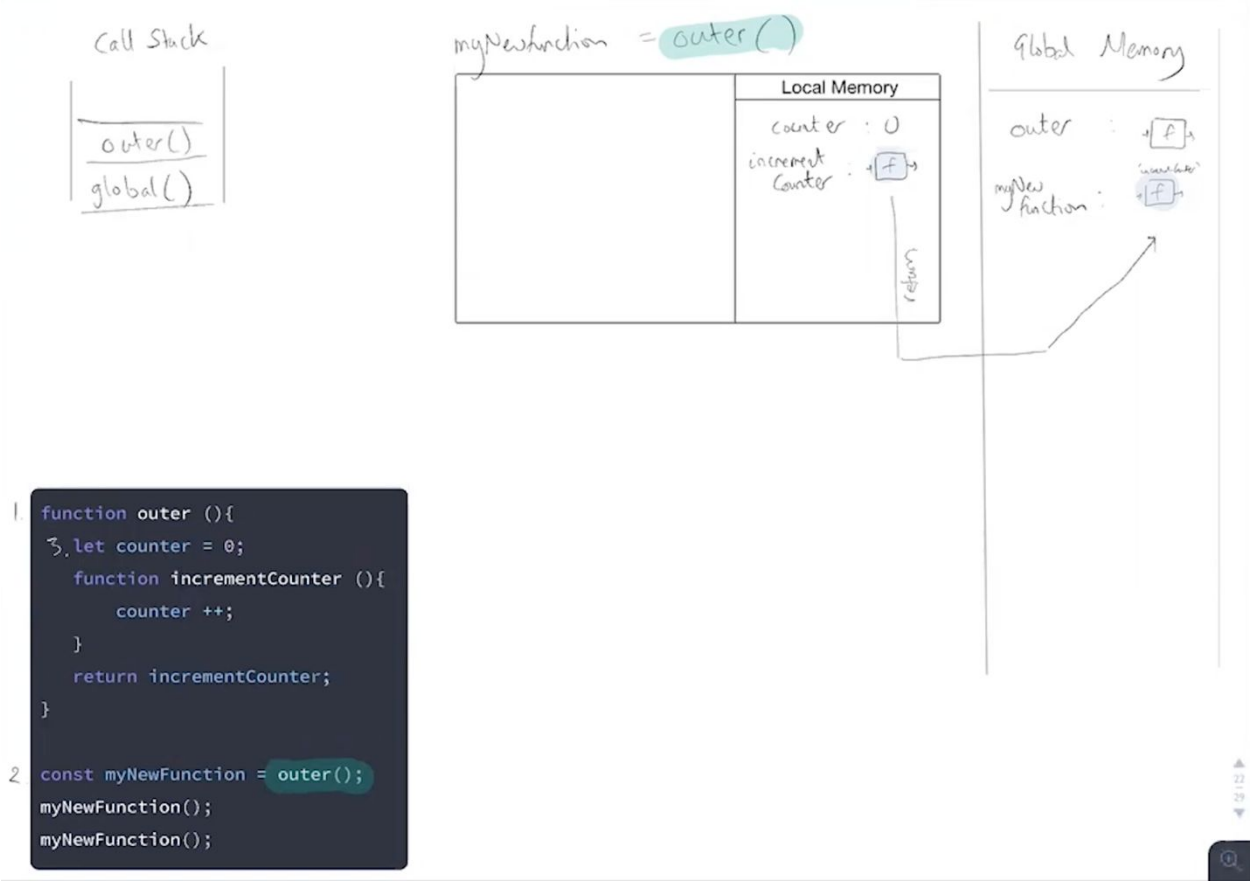
So I'm just going to get us back to where we were so that you can have a good starting point. Natalie, we're now running outer. That's what we're doing. This is the running of outer. Into it we go and we are now hitting line 3 inside of outer, which says to do what Natalie?

Natalie: Declare variable local memory with the label counter and assign it the value zero.

Will: Spot on and focus it's a variable because it's declared let and you can change its value as much as you want. Excellent Natalie thank you. Next line Natalie.

Natalie: The next line we are going to declare a function with the label incrementCounter and save the function definition.

Will: Beautiful. There it is. I'm going to highlight it so it's easy to track with my little blue pen. Okay. Perfect. And now the hard bit. Natalie, am I running incrementCounter here or doing something else?



Natalie: No, you're not running incrementCounter.

Will: What am I doing?

Natalie: You're going to return out the function definition of incrementCounter or the value assigned to the label incrementCounter.

Will: Very nice. Out into what new global label Natalie?

Natalie: myNewFunction. In global memory.

Will: myNewFunction in global memory. There it is, stupendous there from Natalie. We returned out and myNewFunction is a function formerly known Natalie as-

Natalie: incrementCounter.

Will: incrementCounter. Beautiful. And as we exit outer. Oh damn it, I didn't add to the call stack. All the time we were inside of outer it was on top of our call stack. Exited, what keyword exited for us, exits out of for us Natalie?

Natalie: Return.

Will: Return. By the way as a side note. If there were no return statement inside a function as soon as we hit the closing curly brace JavaScript inserts what's called an implicit return and implicitly which means without us saying to do so returns undefined. Just as a side note. Excellent, thank you Natalie. We exit out, we return out, three four five. You return out of outer into myNewFunction, the functionality formerly known as incrementCounter into myNewFunction. Ok. And now we head back out and we're into global. Oh, Kate, call stack. We exited outer, JavaScript panics, does it know where to go back to? We know to go back to Global, but how does JavaScript know to go back to Global? What's it do so it knows where to go next?

Kate: You're saying that because once it hits return then it knows the thread of execution should go back to Global.

Will: But how? What does our call stack do to help it know where to go?

Kate: Meaning that it pops it off the call stack?

Will: It pops out of the call stack. And so Kate what now is top of our call stack?

Kate: The global execution context.

Will: Fantastic. So JavaScript goes back to Global where Kate, what line do we hit in Global?

Kate: We're now on line 6.

Will: Which says to do what Kate?

Kate: To invoke myNewFunction.

Will: Fantastic! And myNewFunction is the functionality formerly known as-

Kate: incrementCounter.

Will: Fantastic. Well done, Kate. There it is. So let's take that execution context. Beautiful. myNewFunction's execution context. Into it we go. Let's put it on the calls, whoops. Let's not forget to put it on the call stack. myNewFunction which we're running. Are we running myNewFunction here inside of outer, Kate?

Kate: No, we are not.

Will: 100% we are not. Definitely definitely definitely not. That execution context, though a little mini app, which is what an execution context is, for running outer, gone dead over. And I guess in theory, all of its local memory deleted. Except the returned out function incrementCounter, which was stored into myNewFunction. Which is now what we are running out in global. Perfect. Thank you, Kate. Let's just make it clear we are in myNewFunction. Kate, what is the first line of code inside of myNewFunction? You're going to have to visually look back up the page, aren't you? To incrementCounter but we know that JavaScript isn't doing that it's looking into the saved myNewFunction, that was the incrementCounter function that was returned out and saved in global. So Kate, talk me through the look up process JavaScript does for myNewFunction.

Kate: Ok. So when it sees myNewFunction it looks in global memory if it knows what that is, and I guess it maybe goes to the heap. In any case, it finds myNewFunction.

Will: Oh it's running it.

Kate: Yes, we are running it now. First line is counter ++ or incrementCounter.

Will: Perfect. Counter ++. Excellent. Counter ++. Beautiful. And Kate. Here we go. Everything we know about JavaScript would suggest JavaScript takes this look up journey, I'm going to use my purple pen again for the look up journey. Where do we look first, Kate? I'm calling myNewFunction, which I am calling inside of global execution context. Where do I look as I run myNewFunction? Looking for counter, where do I look first?

Kate: So we are first going to look in the local execution context or the local memory of myNewFunction.

Will: Yep. Do I find counter?

Kate: We do not.

Will: Oh, why did you not find it there? The first thing I did inside myNewFunction was counter ++, no declaration, no saving so I can't look up something I didn't save.

Kate: Right.

Will: Where, Kate, would every intuition in the history of JavaScript suggest that we look next for that counter?

Kate: Intuitively, you would expect it to then go up the call stacks and look in the global memory.

Will: Down the call stack but-

Kate: Sorry.

Will: Down the call stack, into global. Everything would suggest that and out we go into global memory where Kate, what do we find? Any counter?

Kate: No. It's not there.

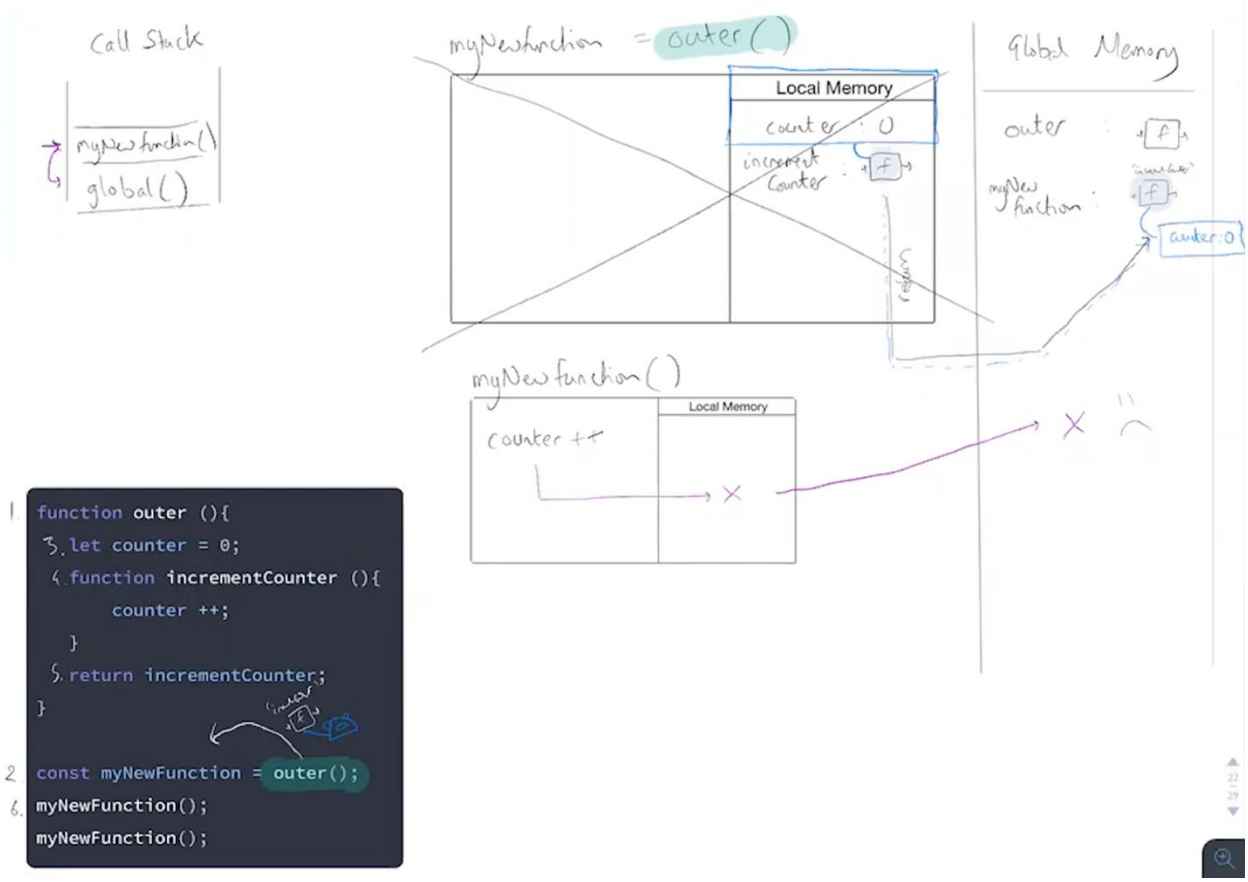
Will: No counter. We have some chat things here. Ah, indeed. There it is. There is nothing there. No counter there. After all this hard work, we have no counter there. How long should I leave this for the recording? How long do I just sit here for a bit? There's no counter there! This is very devastating for all of us. Very sad. We've been running myNewFunction, from incrementCounter but it was born at a time when counter was available but we exited outer by returning our incrementCounter into myNewFunction. And now we don't even have counter, what do we do? Jim, stop typing. What do we even do? What should we do? What shall we do? Well we did say earlier, that what determined what data I had available to me, in my function, was what data was there when my function was initially saved. Not what data's there when my function... Maybe there is something going on with that. Huh. By the way we have to cut this bit into more tight as I got distracted by Jim typey typey. Cut me chastising Jim. As you know I'm not going to cut any of these pieces by the way. These are the bits that make gold. Me chastising wonderful Jim.

Alright. So, by the way we got new Codesmith hats. I think they are actually genuinely quite good. Look at this. I have an enormous head, so Jim stands chastised. This is going to be a disastrous continuity error if we record anything early on again. "He takes the hat on and off a lot. Between literal seconds, these little fast cuts with the hat thing." Alright, so, there is a big struggle. We're calling myNewFunction, it makes reference to something that isn't there. What sort of language has this as a feature? What a strange language. What a strange language design this is, definitely not the languages I dreamed of. Unless. Alright, this drumroll was pretty lame. Unless, something else is going on. Unless, we didn't just return out a little baby function incrementCounter out into myNewFunction.

What if we didn't just return out of function. What if we returned out of function and as it came out and got started incrementCounter, it brought with it a backpack full of data. Full of all the data that was there when it was born. On his back out it comes and brings all the date from when it was born out on its

back. A little backpack. Shall we see? It might be exactly what happened. How am I going to diagram this, let's give it a shot. Here it is, so it turns out folk, when incrementCounter got returned into myNewFunction, it actually grabbed hold of the surrounding data, grabbed hold of it, attached it onto the function, grabbed hold of it on its back and returned it out attached onto the formerly known as incrementCounter function now sorted into myNewFunction.

We returned to our function plus that little backpack of data attached onto, there it is, my little backpack of data attached onto incrementCounter stored now in myNewFunction. Out returned all the surrounding local memory, we'll add a caveat to that a little bit later on, with counter is zero attached on as a little backpack of data. There is my backpack, attached onto the very function definition that we returned out and gave a new global label myNewFunction. We are going to learn that is probably not the right term for it, backpack, but for now we will love this term. And so, Joe, I haven't called on you for a while Joe. I'm going to ask you, Joe, the hard question.



We are running myNewFunction. We see the line counter ++. We are running myNewFunction in global. We left outer long ago. We returned out a function incrementCounter from outer. We left outer long ago. We are now running myNewFunction. And Joe, tell me, I hit the counter ++, I looked in the local memory. Well, you tell me Joe. What happens? Talk me through what the look

up process is after... Let's take it from the very beginning. So I'm calling myNewFunction, I hit counter ++, where do I always look first for my data?

Joe: The local memory in myNewFunction execution context.

Will: Perfectly put. But I don't find it there. I look at my call stack, maybe, no no no, but you might think I do. You think I would look down the call stack and hit global, but no. Where do you think, Joe, that I actually look? Or JavaScript actually looks for that data next?

Joe: The backpack.

Will: Into the function definition, myNewFunction, that I am calling's associated backpack. Exactly. And what do I find there, Joe?

Joe: counter = 0

Will: That's exactly right. I find counter = 0. And what do I do to it, Joe?

Joe: Increment.

Will: Increment it to one. There it is. Beautiful. Let's keep going. We are going to talk about how is this possible? What does this enable us to do, all these things? But we now exit myNewFunction and hit, we exit myNewFunction, we pop it off the call stack, and go back to where Jim? I exit myNewFunction, pop it off the call stack, now back to where?

Jim: Global.

Will: Back to global, excellent. Where, what is my next line of code Jim?

Jim: I don't know because I forgot where we were. Line 7, you're invoking myNewFunction again.

Will: Excellent. Well done, Jim. Good catch up.

Jim: Yeah, thanks.

Will: Invoking myNewFunction again we create a new-

Jim: Execution context.

Will: Execution context. Well done. There we go. A new execution context, there it is. And what's the first line of code inside of it Jim?

Jim: Um, it is going to be Counter ++.

Will: Right! Because myNewFunction really just incrementCounter. We add myNewFunction, sorry I forgot to do this to the call stack. Inside of global, we are running it in global, not the inside of outer. In global. And Jim I'm going to use my purple pen. Actually, Elena, talk me through JavaScript's look up process now. It's running myNewFunction, it's top of the call stack, where does it look for counter first, Elena?

Elena: In local memory of myNewFunction.

Will: Beautiful. Does it find it?

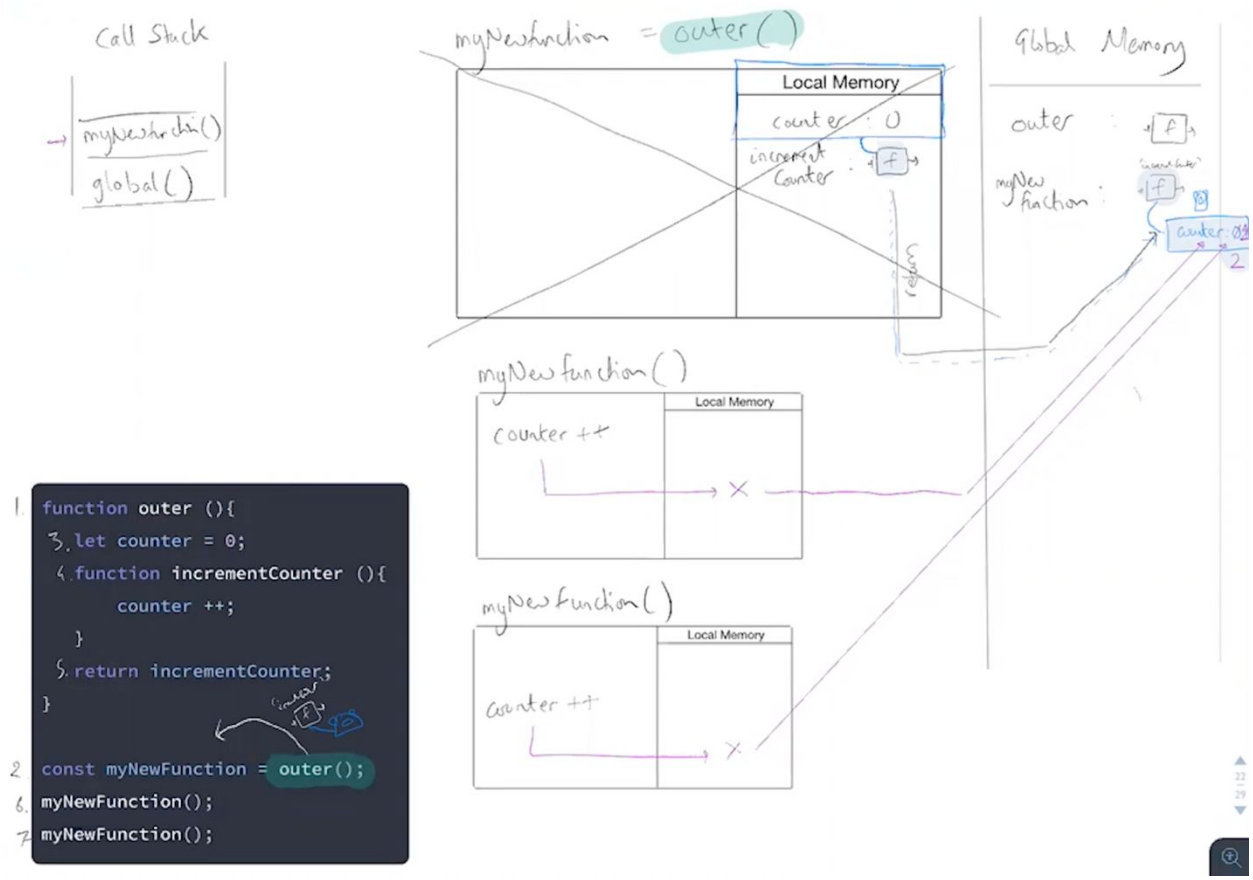
Elena: No, it doesn't. So the next thing it's going to look at is the backpack of myNewFunction.

Will: In the backpack of myNewFunction. Where it finds counter is stored, not counter will be zero. Counter will, no, counter is literally stored as what number at this point Elena?

Elena: 1.

Will: And what are we going to do to it?

Elena: Add one more. (Laugh) So it's going to be 2.



Will: Increment it to the number 2. Phenomenal. Very, very nice from everybody. And so, the first thing to notice people is that this myNewFunction function has two memories. It has its little local memory that is new each time, it's fresh and empty and newly born each time the function is called. And it has this persistent memory, that sticks around permanently attached to the function's definition. Remember there's two stages to a function, two stages to a function. Storing its code, that's this bit up here, and then running it. When you run it, you create a little memory but that memory is temporary. But what if when you ran it, you also had access to a permanent memory? That is attached to the very function definition that we can then go and look into and store stuff into and grab stuff from. Only from inside of the running of myNewFunction. People this is going to change everything we do in JavaScript.

We are going to discover the fancy name for the backpack. And I'll give you one name that's used colloquially, I don't love it, but everyone uses it in developer world. And two names that I firmly believe that if you use in a job interview, and I have heard people tell me, who haven't gone to Codesmith, who have not gone to Codesmith, but who have just been to Hard Parts workshops, I've heard them tell me these two terms have literally got them jobs. These two terms, the most over-the-top, but I think highly precise terms for this backpack, I far prefer to the more colloquial term that developers use for it. They don't use the term

backpack, although it's blowing up. One of who, Olivier De Meulder, the Engineering Manager at the New York Times, who came to Hard Parts a couple of times. He loves the term backpack. He wrote an article about it with 40,000 likes on Medium that references as a backpack 20 times. This term is going to be real. I don't know what this chapter of this workshop is going to be called, this chapter of the video workshop is going to be called, "Will rants about the backpack for a minute." But we're going to learn a term that is very very nice for this. We're also going to learn if our functions now have two types of memory, the memory that when we run them gets created brand new and fresh and green every time, and the permanent memory they can use, what's that lets us do? Because it lets us do a ton of stuff. Once our functions have memories between their runnings, you can just completely change how you write code. Completely going to change how you write code. But also let's talk about some of the edge cases as well about how this backpack works. Like where is it stored? How is the function definition myNewFunction got this associated backpack attached to it? We're going to talk about all those things as well. But for now...