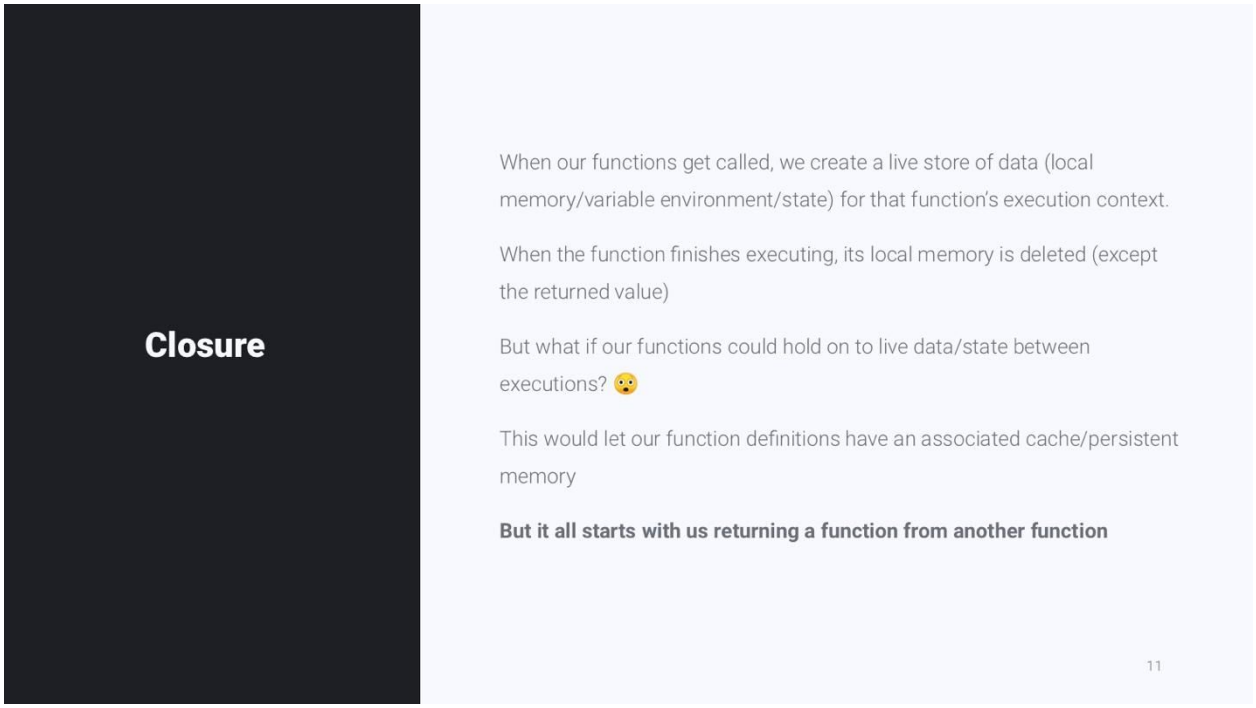
Lecture: 5 Closure Persisting State



Will: First, let's remind ourselves what closure even is. We saw a moment ago when a function is invoked, is called, is run, that means we take the code of it, and we start doing it. By the way, remind me Natalie, how many times do I tend to save a function? Declare it. How many times do I declare a function? Sort of trick question.

Natalie: Only once.

Will: Only once. How many times do I use that saved code?

Natalie: I don't think there's a limit on how many times you could use it.

Will: The point exactly, Natalie. We want to save it, and then use it again and again. Exactly right. Now, each time you use it again, we make a little mini-memory inside our execution context to run that function's code. And with anything we announce, declare, save inside that function, gets stored in there. When we finish running that function, deleted. Gone. Except the returned out value. There it is, except for the returned value. It's what we want. I do not want each time I run my multipleBy2 function to remember the 7,000 previous times I ran it with different numbers. That would be silly. That would be undoing the D.R.Y. principle, the Don't Repeat Yourself. I don't need that previous work. That's not the D.R.Y. principle. It would be inefficient. Why'd I hold onto all that previous work? I only care about the outputs. I got those. Move on. But so often, it would be incredibly powerful if I could actually have my function when it runs remember things from the previous running.

Suppose I wanted to limit my function to only being allowed to run one time. How would I do that? By the first time I run it, saving "Oh, you've already run me!" And next time I try to run it, remembering from the previous time, "You had already run me." Well, suppose I'm trying to have a function that gives me out a new value from a list of values, a new number from a list of numbers, each time I run the function, gives me out the next number. Each time I run the function, gives me out the next number. Known, by the way, as an iterator. How could I remember what the previous number I gave out was so I don't give it out again when I run the function, unless my function remembered the behavior or what happened the previous running of it?

So there is a way that my function could have a permanent, a persistent memory, a persistent store of data kind of attached to it, in addition to its brand new store of data each time it runs. That means it's local, temporary memory. It would change everything about how I run and write code. But it all starts... We would let our function definitions have an associated persistent memory. Let's give it a kind of more intuitive term for that. An associated cache, if you know that term. Or another alternative term would be this, a associated permanent store of stuff, data. It would be amazing if our functions, each time they ran, had attached to them a permanent store of data that could be used. Not a set of commands to save things. No, no, no. Not instructions to save data, but literally a little permanent store of already pre-saved data in which we could update, use, change, but it's remembering stuff from previous runs. It would change everything about how we wrote and ran code. Alright. By the way, it will turn out to be probably the most powerful concept in JavaScript.

Closure

When our functions get called, we create a live store of data (local memory/variable environment/state) for that function's execution context.

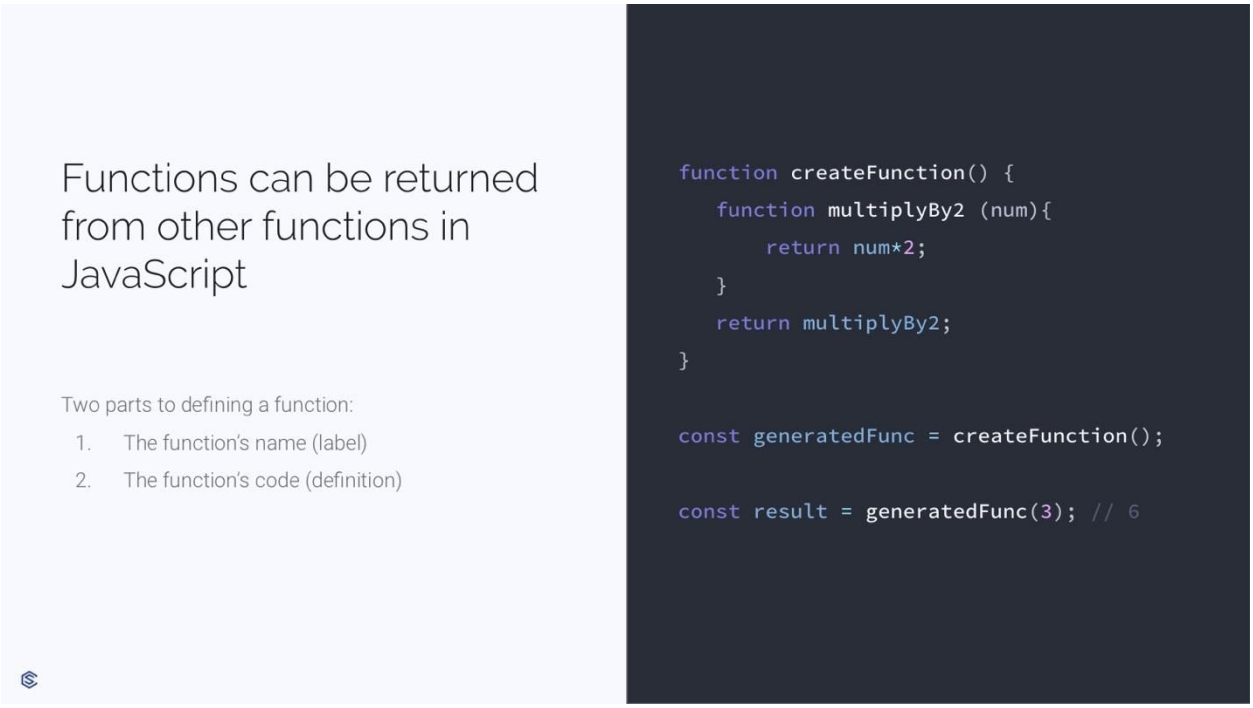
When the function finishes executing, its local memory is deleted (except the returned value)

But what if our functions could hold on to live data/state between executions?

This would let our function definitions have an associated cache/persistent memory

But it all starts with us returning a function from another function

Lecture: 6 Returning a Function from a Function



Will: It all starts though with us returning a function from another function. Let's take a look at it. Here it is. Functions can be returned as output from other functions in JavaScript. If we get this notion down, and hopefully the thing we just saw, helps us do so. Understanding that functions can have multiple labels and that we can move them around. If we get the notion returning a function from a function down, then we have everything we need to understand the most profound concept in JavaScript, closure. Here it is. Let's take a look. Here's our code. Put it over here, and let's start diagramming away. Let's call on to start us off, let's call on, who have we got here? Helena? Jump in, Helena. Tell me what we're doing in our first line of code. And we're not going to talk in terms of the heap. I think we may add that in, Helena. For now, let's just say what we're doing in terms of our regular global memory. Helena, what are we doing in line 1?

Helena: We're declaring createFunction, with the function definition in global memory.

Will: Beautiful. And for our later viewers, if you want to see what that actually does and how that function gets stored, have a look at Joe's question about exactly what is a function stored in JavaScript even look like under the hood? Go back to that question. That's a message to our later viewers. Alright. Excellent, Helena. Thank you so much. That was line 1. Line 2, Kate, take it away.

Kate: Okay. In global memory, we're creating new const with the label generatedFunc, and we're going to set that equal to the invocation of createFunction.

Will: Kate, forgive me. I don't even like saying, set it equal to the invocation of createFunction. Right now, it's nothing. It's not going to be set equal to invocation of anything. It's on initialize and it is just, it's just nothing. There's no value stored it, and there won't be until the final value is stored there, which will be the output, the returned value, the thing that comes out the bottom of running what function, Kate?

Kate: createFunction.

Will: createFunction. And tell me, Kate, this is by the way, a real puzzle for people. Is that createFunction being executed?

Kate: Yes, because there are parenthesis.

Will: But what might you think there needs to be to execute a function, Kate? What do a lot of people instinctively think needs to be there to execute a function?

Kate: You might think you would need a parameter or argument.

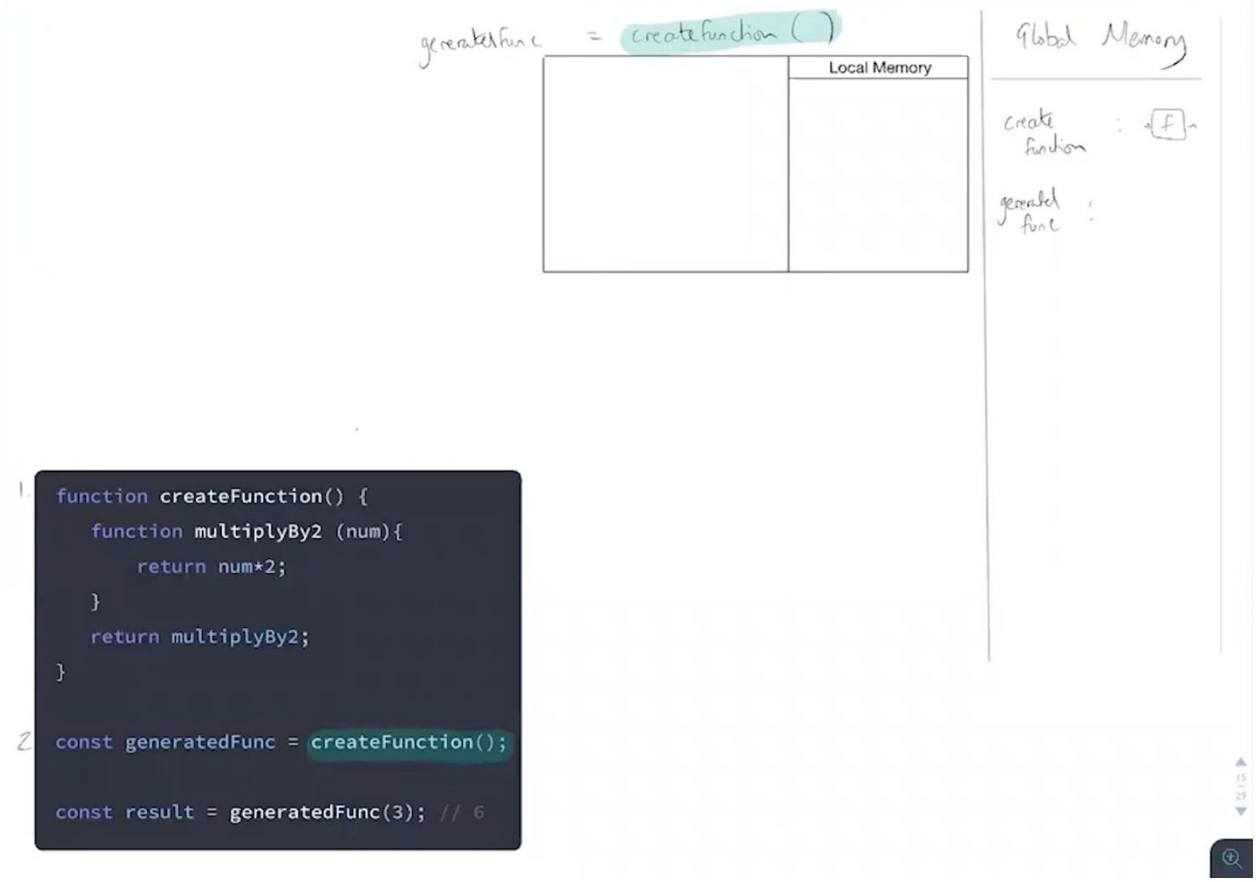
Will: Exactly. An input. But we don't. You can trigger a function running with no argument. That is exactly what's happening here. Empty parenthesis mean go run the function. I mean, there's no parameter to fill in, right? It's not like we're missing something to fill in the hole, there's nothing sitting there, it's empty. Okay, nicely put Kate, excellent. So we are going to say for now generatedFunc is uninitialized, while we head off and start executing createFunction. generatedFunc is declared, and it's going to be the output. Remind me Kate, what keyword inside of createFunction will indicate what's going to be the output of createFunction? What keywords going to indicate that Kate?

Kate: That would be return.

Will: Return, excellent well done. There it is. So createFunction is invoked with parenthesis. We create, let's make it a movement, everyone unmute. Everyone unmute! We're going to create a brand new-

Group: Execution context!

Helena: That was awful.



Will: The elegant and refined, everybody. Alright there it is, execution context. Into it we go, the thread weaves in, this is tricky people. So we're on line 2, we are now entered the running of createFunction. I've highlighted what we're on right now, so we're on the running of createFunction. We go into it, what Joe, is the first thing we do inside of there? And I know this is a tricky line. People always, so Joe, take your time on this one. I know you know, but just for our audience, tell me what we're doing on this line inside, let's call it line 3 for now, inside a createFunction Joe?

Joe: So, in our local memory we're reserving a space for multiplyBy2. And then putting in that function definition.

Will: Yeah, absolutely. Two parts to it, the label for the function and then the definition of the function. The code of it. Which, I almost want to tell you, but remember what's that mean? Well, we're going to look over this in a moment, for now we keep it simple. There it is, the function stored in local memory, do we run it Joe? Do we execute multiplyBy2?

Joe: No.

Will: So, what's our next line say to do Joe? Let's call it line 4 here.

Joe: So return the, I don't think you return a pointer, you return the actual function definition.

Will: Perfect, let's distinguish function definition versus function label. Do we return the function label Joe?

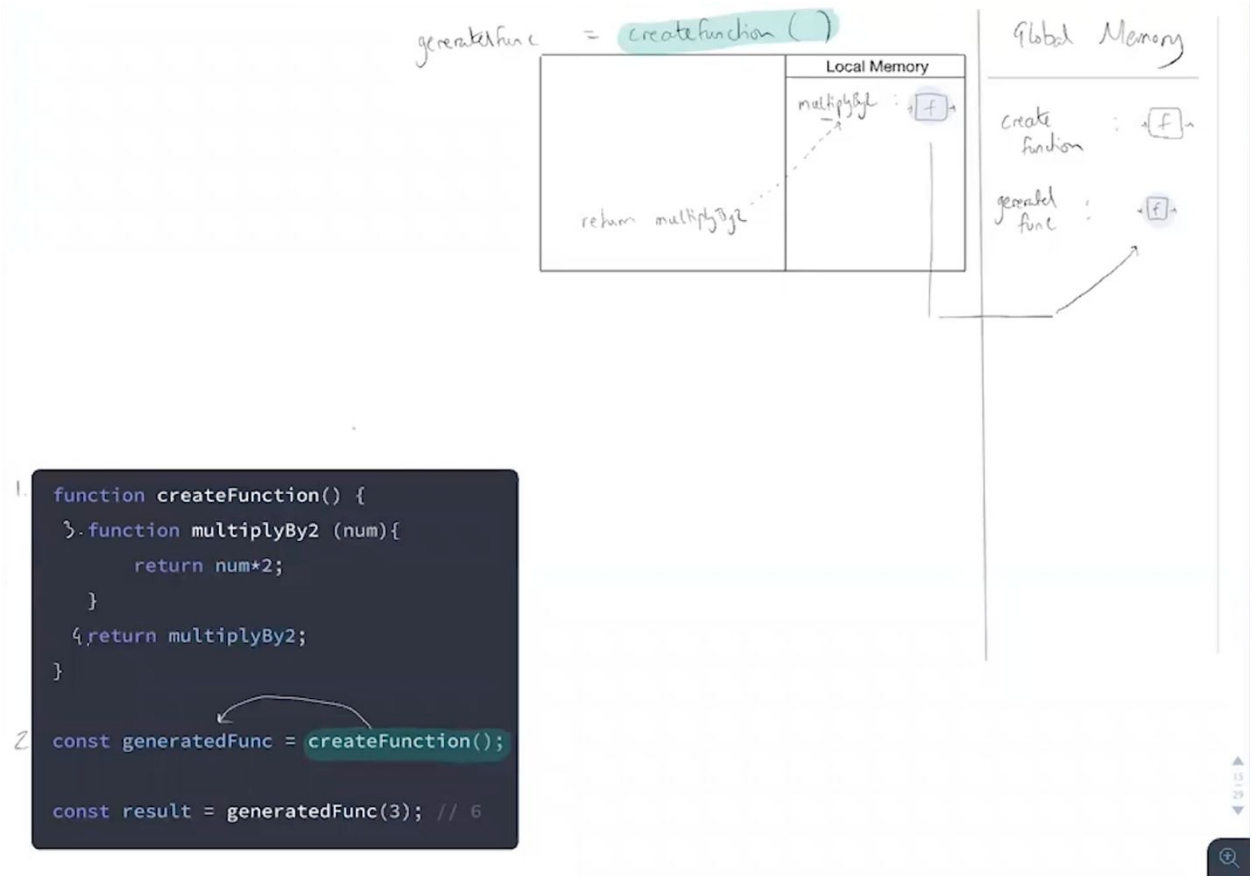
Joe: No.

Will: We definitely don't. What do we use, what's the point of having the function label there at all then?

Joe: So you know, so you can access that memory.

Will: So that you can access it. We use the label to find the function definition, which is why by the way, we could just write inside createFunction, we could just write, return function without a label at all, they call it returning anonymously. We took an intermediate step here by saving it with a label, and then using that label to find the function to then return out. Excellent, so JavaScript sees multiplyBy2 and he goes, "Ah, what is that? Oh it's this functionality." There it is, this functionality, we'll do it in blue, and this functionality is what we're going to now return out without a label. Absolutely no label, into what, so the output of createFunction is going to go into what global label, Kate?

Kate: Into the label generatedFunc.



Will: Beautiful, there it is. And just remind us Kate, what was this function generatedFunc, previously known as when it was born? So we now got a new label for a function that was previously known when it was born as what?

Kate: It was previously known as multiplyBy2.

Will: multiplyBy2, fantastic. So we're going to walk through this again in a moment with the heap, multiple of how this works, I think you probably already spot. So you can already see that we can have two labels, we say earlier, two labels for the same function, no problem. Those two labels, one could be inside of the execution context in a local memory, and then we could have the other label being its new label in the global memory. It's got a new global label, so we return out, what happens to our execution context Kate? What in terms of our call stack for now, just sort of more metaphorically, what happens to our execution context when I exit when I exit the function?

Kate: It goes away, it's deleted, it's gone.

Will: Deleted, exactly. And everything that wasn't returned out gets deleted, including our little label here. All we get is returned out, but just know that this used to be known as, multiplyBy2. Alright, perfect. So now, the real crazy bit, we know head out, we're on line 5. Where Jim, talk me through line 5. Left hand side first.

Jim: We are declaring the const results, and we're assigning it the output value of generatedFunc with the argument 3.

Will: Very nice. So it's initially, uninitialized. While we head off and execute generatedFunc, which everyone together, what was generatedFunc's-

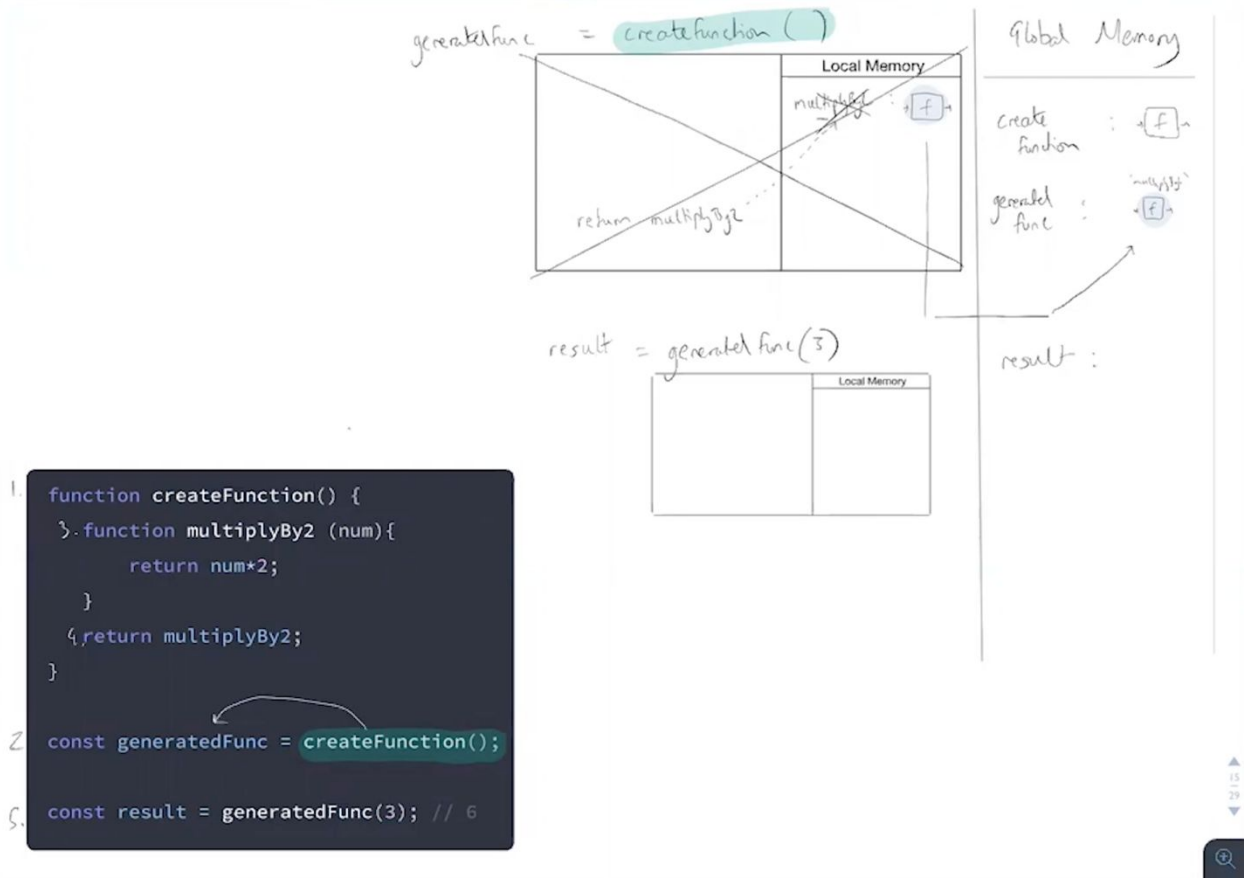
Jim: Did I do that too early?

Will: What was generatedFunc's former name before we renamed it generatedFunc? What was it's original birth name before it got assigned a new name and lost its old name? Everyone together.

Group: multiplyBy2!

Will: Exactly right. So we really, when we execute generatedFunc, we're really executing multiplyBy2. We create, everyone together a new-

Group: Execution context!



Will: Excellent, into it we go. And here's a weird thing people, and I wish this weren't the case, as developers we're running generatedFunc, we know it's really multiplyBy2. So

what do we do? Our eyes wander back up the page to where we saved multiplyBy2, where we you know, gave that as an instruction, if we would ever call createFunction save multiplyBy2. So we, our eyes wander up, and therefore we get a total misperception of what JavaScript does. JavaScript does not go back up the code to find multiplyBy2 earlier. It doesn't have to, because when it ran, generatedFunc, no not generatedFunc, when it ran createFunction, it returned out multiplyBy2 and took the whole code of multiplyBy2 and stored it in the new name, generatedFunc.

And so when it sees run generatedFunc, it goes up here, and finds the code formerly known as multiplyBy2 and runs that. So Natalie, talk me through the running of generatedFunc. You're going to have to look at multiplyBy2s code, but do not think JavaScript's going back up the page to do that, it's looking at global memory where it saved the code of multiplyBy2 under the new global label, generatedFunc. So Natalie, tell me what it's doing in that, what's the first thing that JavaScript does when it starts running generatedFunc, formerly known as multiplyBy2?

Natalie: Okay, the first thing that JavaScript does is declare the parameter num and assign it the value of 3, or declaring the, yeah, parameter num and assign it the value of 3.

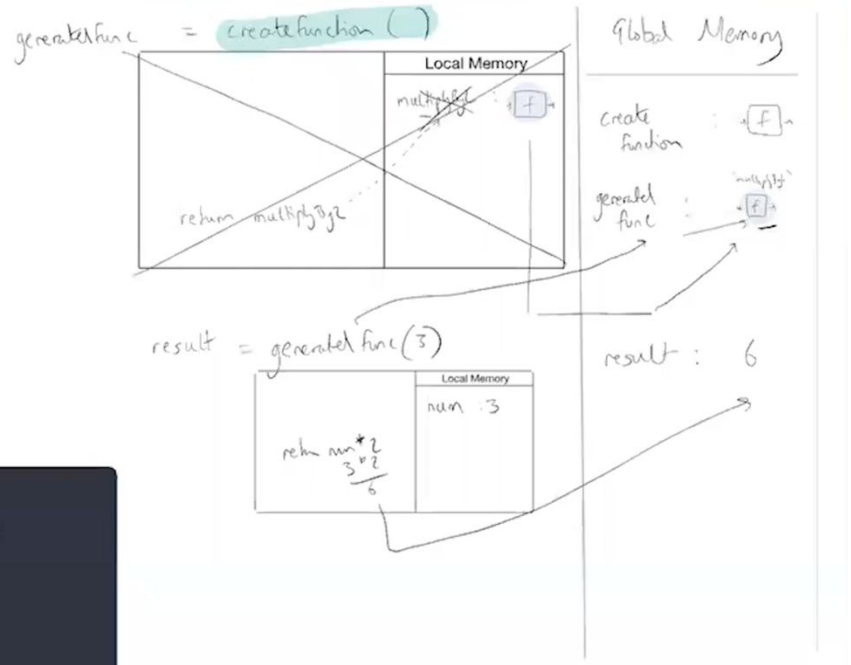
Will: And the value of 3 is also known as the?

Natalie: The argument.

Will: The argument. Excellent from Natalie, and then we've got return, num by 2, which is 3 by 2, which is 6. And we return that out into, Natalie?

Natalie: Into our constant result in global memory.

Will: Constant result in global memory, fantastic. That's it people. Now, you might be asking, "Why did I save multiplyBy2 inside of the running of createFunction, only to then return it out and use it by a new global label generatedFunc. Why not just define the function multiplyBy2 globally?". It's going to turn out, when we birth a function, save it inside of another function's execution context, and then return it out and give it a new global label, by which we then call it, execute it, evoke it, run it. That returned out function, has the most powerful feature of JavaScript bundled onto it as a bonus. It's going to change everything about how we write and run JavaScript.

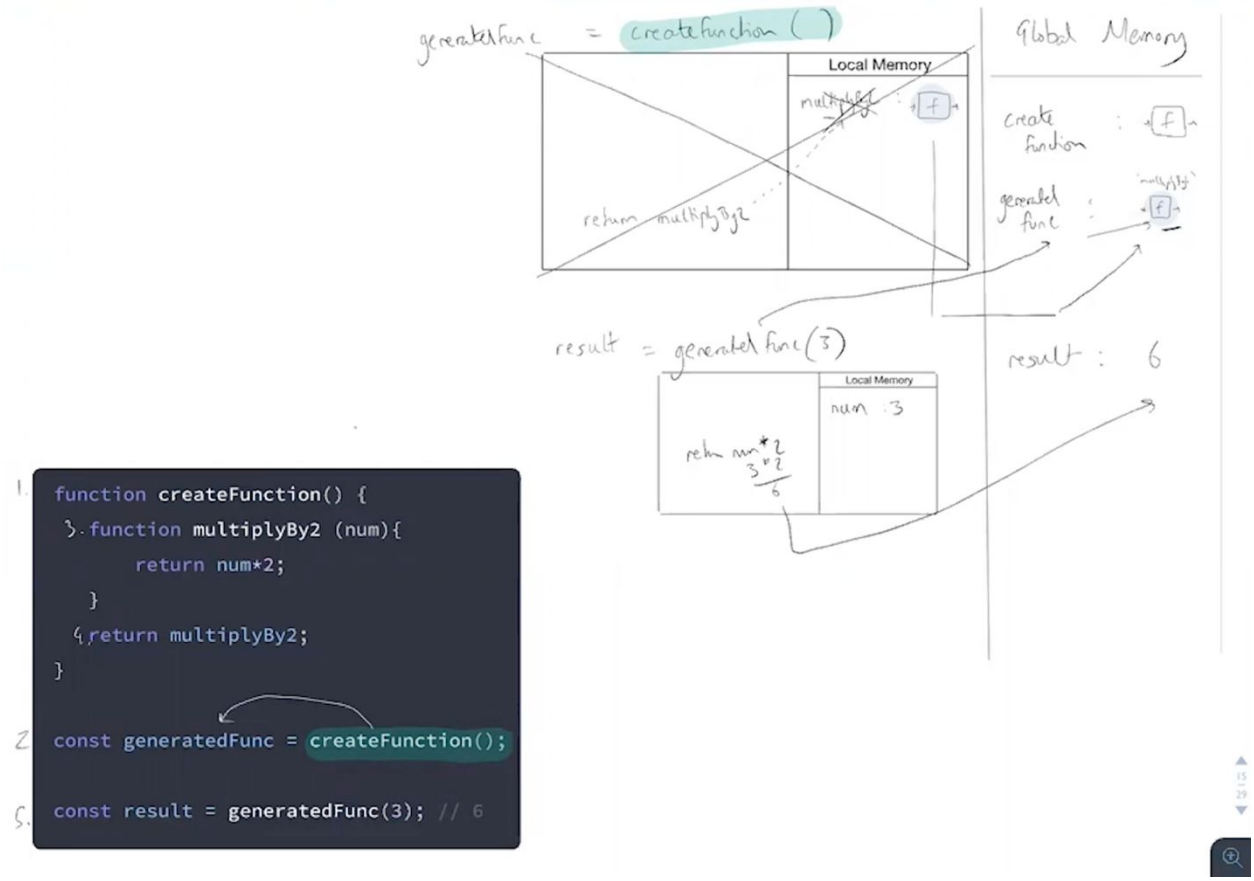



```
1. function createFunction() {
     function multiplyBy2 (num){
       return num*2;
     }
     return multiplyBy2;
   }

2. const generatedFunc = createFunction();

3. const result = generatedFunc(3); // 6
```

Lecture: 7 Q&A



Will: First, let's do questions, and then I think one of those is probably going to illuminate this in terms of the heap. So throw out your questions on what you've just seen here. Super tricky stuff, throw out your questions on what you've just seen here. Someone could just write a question about how it relates to the heap and I'll answer it. Anyone want to be the person ... Natalie, jump in.

Natalie: Yeah. So I'm ... with the concept of the heap here, at what point does JavaScript look to the heap to get these function definitions, and at what point it ... how does that work?

Will: Good question, Natalie. That's what we call a prime question. Good question. Alright. Let's see this in terms of the heap. I'm going to use my blue pen again for the heap. So the heap is down here, and again, as a reminder folk ... the heap is where objects, arrays, and functions, the complex stuff in JavaScript, not like numbers or strings, they're stored in the heap. And then in the global memory, in the portion here, we actually just store a label, not a label sorry, a numbered numeric address, so to speak, to the position in the heap where the actual underlying function, object, or array is stored.

So let's just walk back through in terms of that. So we actually, when we declare createFunction, I guess, we declare it with an address, the zip code of Codesmith's old

New York office. There it is. And then the function is stored there. So createFunction is in the position 10012. Then we declare generatedFunc, we saw the right-hand side said, "Call create function." So Natalie, JavaScript would see, "Call createFunction," it would go and look there, and it would go and grab the function from the heap just as you asked. Excellent. Into createFunction we go. And look at this. We declare multiplyBy2. Joe, what do you think we actually, therefore, do in the local memory?

Joe: Sorry, where?

Will: We're calling createFunction, the first thing we, hit line 3, to call createFunction in line 2, we jump into createFunction, which is line 3 is the first bit. What's the first thing it says to do there, Joe?

Joe: Store or make a space called num in the local memory, and store the function definition-

Will: Not num, multiplyBy2, you're on the line multiplyBy2, so declare the function multiplyBy2, there's the-

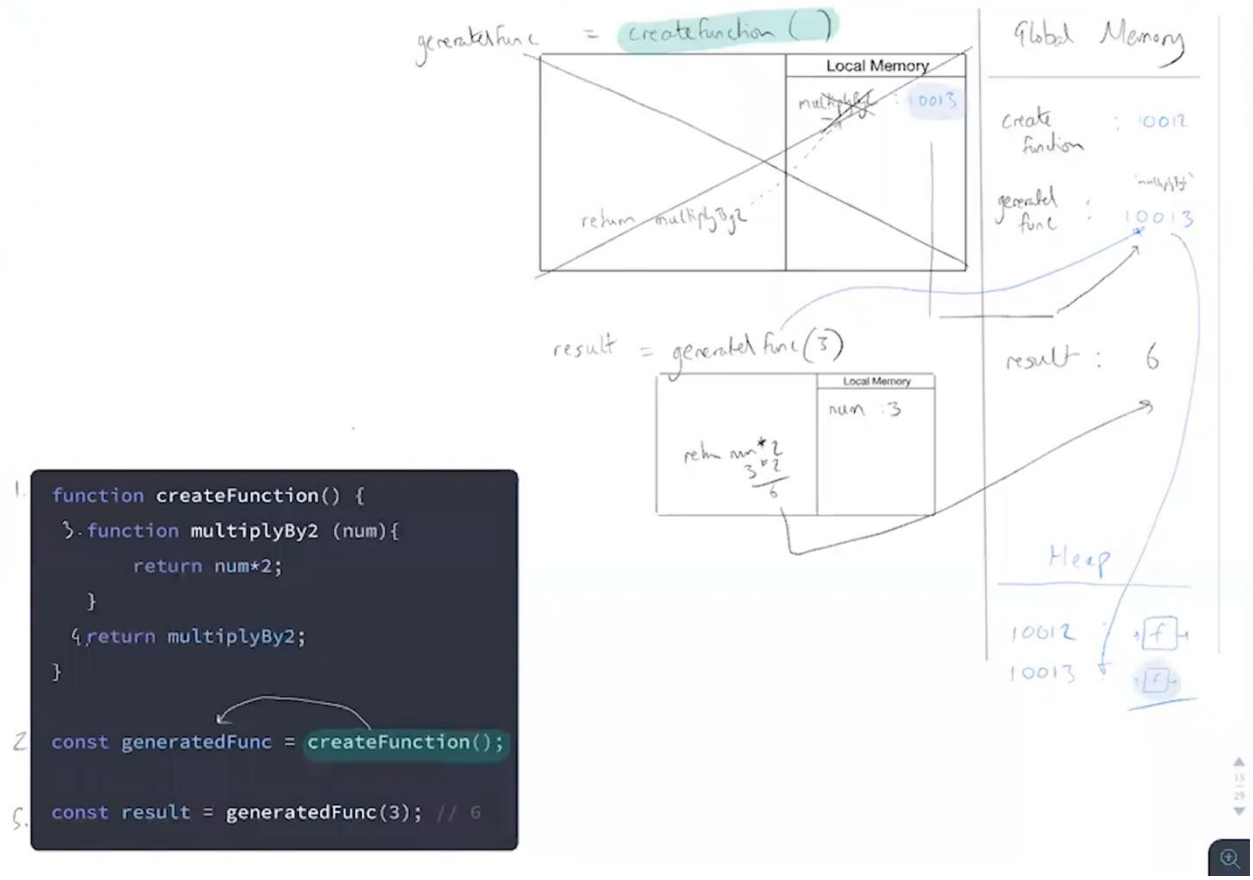
Joe: Oh, sorry. Sorry, sorry, sorry. Yes, yes, yes, yes.

Will: Absolutely.

Joe: My answer was totally wrong. Yeah, so just ... declare function, multiplyBy2, and then store in the local memory or associate with that, the function definition.

Will: Yeah, absolutely. And you answered this perfectly earlier, but let's not talk about in terms of the heap. We're not going to have a whole function stored in local memory because functions are too complex to store in the little mini local memory. So what do we instead store in the local memory, Joe? Where is the function ... what do we actually in reality under the hood, store?

Joe: Ah, yeah, the pointer to the heap or the address to the heap.



Will: An address. Yes, people do call it a pointer. It's a link in the computer to another place in the computer. So we'll say it's a link to roughly, since it's metaphorical, 10013, the zip code of the new Codesmith New York office in Tribeca. There it is. And there's the function saved. Maybe I do this in a different color. Oh, I don't ... yeah, whatever. There it is. I'm going to do it and highlight it in blue as well. There it is. And so, that's multiplyBy2. When I return multiplyBy2, what am I actually returning? I'm just returning a reference, the address, the position of the underlying function, which was temporarily known as multiplyBy2, but that's not a big deal, it was just an address to a position in the heap, and that ... that's not anything ... that is what we're returning out into what global label, Kate? What global label would be returning out that position 10013 to?

Kate: Into the label generatedFunc.

Will: Into the label generatedFunc. And there it is. And so now perhaps we can see that when I run generatedFunc, I go ... I just do this ... I go whee, oh, 10013, whee, there it is. The function that earlier had the name multiplyBy2. And I grab the function, I run it, just got a new label now in global. This is really what's happening under the hood. And folk you might think, "Why do we do this? Who cares?" We do this because when we return a function out of a function or at least the reference to an under the hood heap stored function, out to run another function, into a new global label, and then run it by that global label, oh, it gets the most powerful property in JavaScript. Alright, folk, thumbs on

what we've seen here. You lost me, I'm clear, clarification. Everyone's thumbs out. Jim has a clarification. Natalie's clear. Oh, Jim's clear. Elena has a clarification. Kate has a clarification. Elena, jump in.

Elena: So my question is about this heap. Does the computer know how much memory to save for every function? Like if it's a complex function, for example.

Will: That's exactly the sort of nature and purpose of the heap, is that it is much more flexible with what you can-

Elena: Oh, okay.

Will: With a function, function definitions can't really change. I mean, what you store associated with them can change, we'll see what that means a little bit later on, but an array or an object, the whole notion of the heap is that it should be flexible to add and remove stuff to the underlying stored object or add or remove something underlying some array, and just move the heap's stored data around. Ideally, in as optimal way as possible, but you're moving ... when you reassign, so you add a new property, the heap may shift stuff all around. It's nature is not to be perfectly organized, it's nature is to be more flexible to handle ... to be more versatile, and handle the changing values you want to store in an object, in an array. That's the nature of their complexity, they could grow really big, really small, or really big really fast. And so you want to be able to handle that flexibility or have flexibility to handle that. That makes sense, Elena? Great question. Alright. Any other questions people ... yeah, Natalie, jump in.

Natalie: Yeah, I have a question. So primitive data types, like a string or a number, those wouldn't be stored in the heap, it's only ... or, I guess, it's only variables that are assigned to objects, or arrays, or functions? Is that correct?

Will: Only composite data types. But actually, in fact, in JavaScript exactly how certain ... because a string can be very long, in JavaScript a string can be very, very long. So exactly how long values are stored is going to vary by the engine, and it's going to vary by ... we can be sure that a string, however long it is, will behave as it is stored in the global memory directly. It'll be fixed, you can't change them as a const, and if you pass it in as an input to another function, you actually pass it in directly and you copy it in. Meaning, you can't actually mutate a string when you pass it into a function. You can't change the string inside that function and expect to change the original string outside. So it'll behave as though it's being handled not in the heap. But strings can be very long in JavaScript.

Meaning, there may be some reason to actually not have that in the immediate memory, but it'll always behave as though it is. Okay, very interesting question, Natalie. JavaScript is not as low-level a language as a language like C++, where these sort of things are super strict, where you are limited to how long your string can be if you want it to be stored locally by a specific character or a number. Whereas, here, we are much more flexible with what you can store, but to maintain consistency with other languages, strings, numbers, booleans, true, false, behave very clearly as, as you say, primitives, even if they're much more flexible than traditional primitives would allow

you to be. Great question, Natalie. Alright. Any other final questions? Okay, now we move on. We're almost there. Good because it's been an hour and a half.