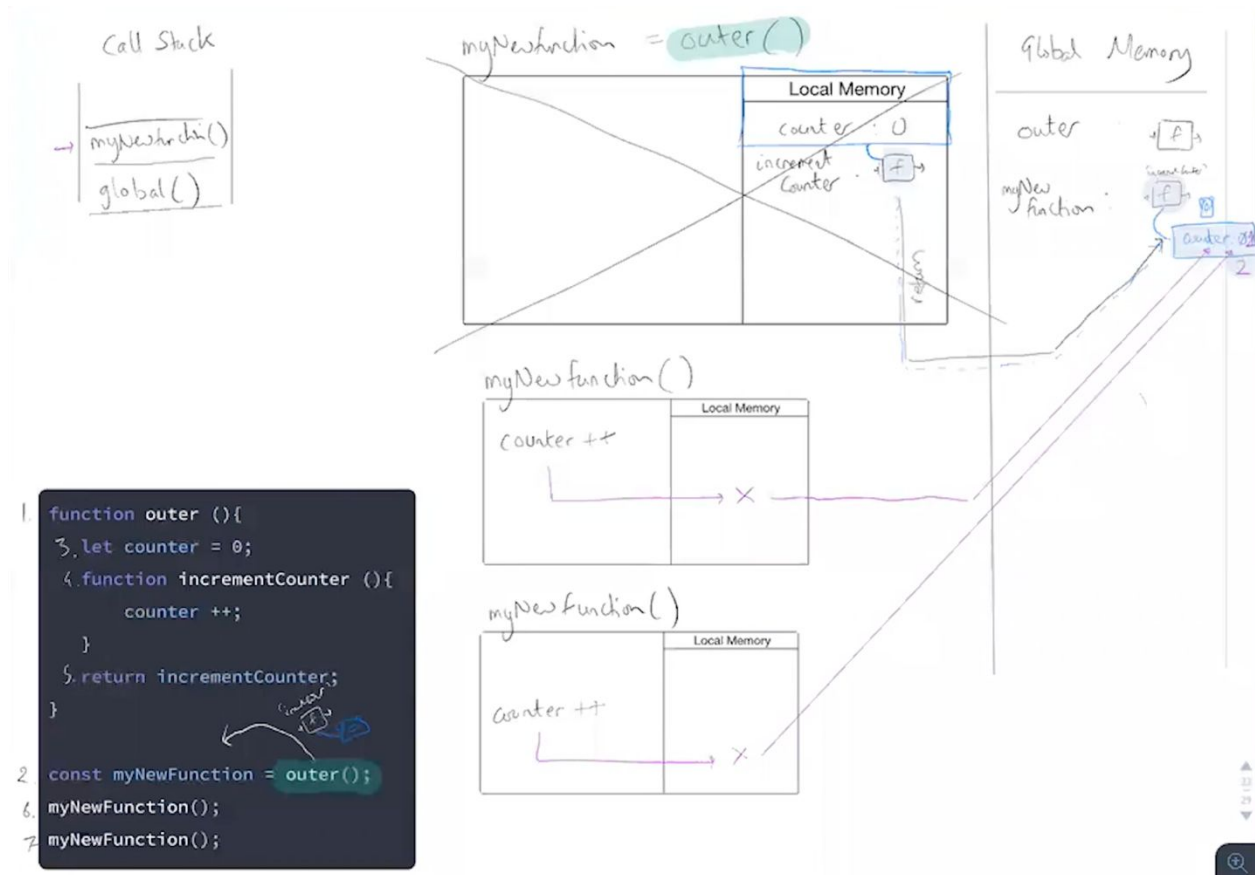


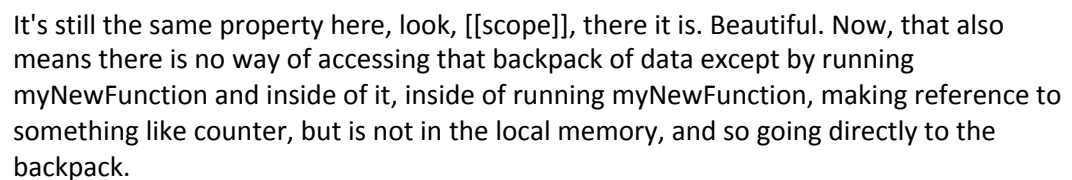
Lecture: 11 Understanding the Scope Property

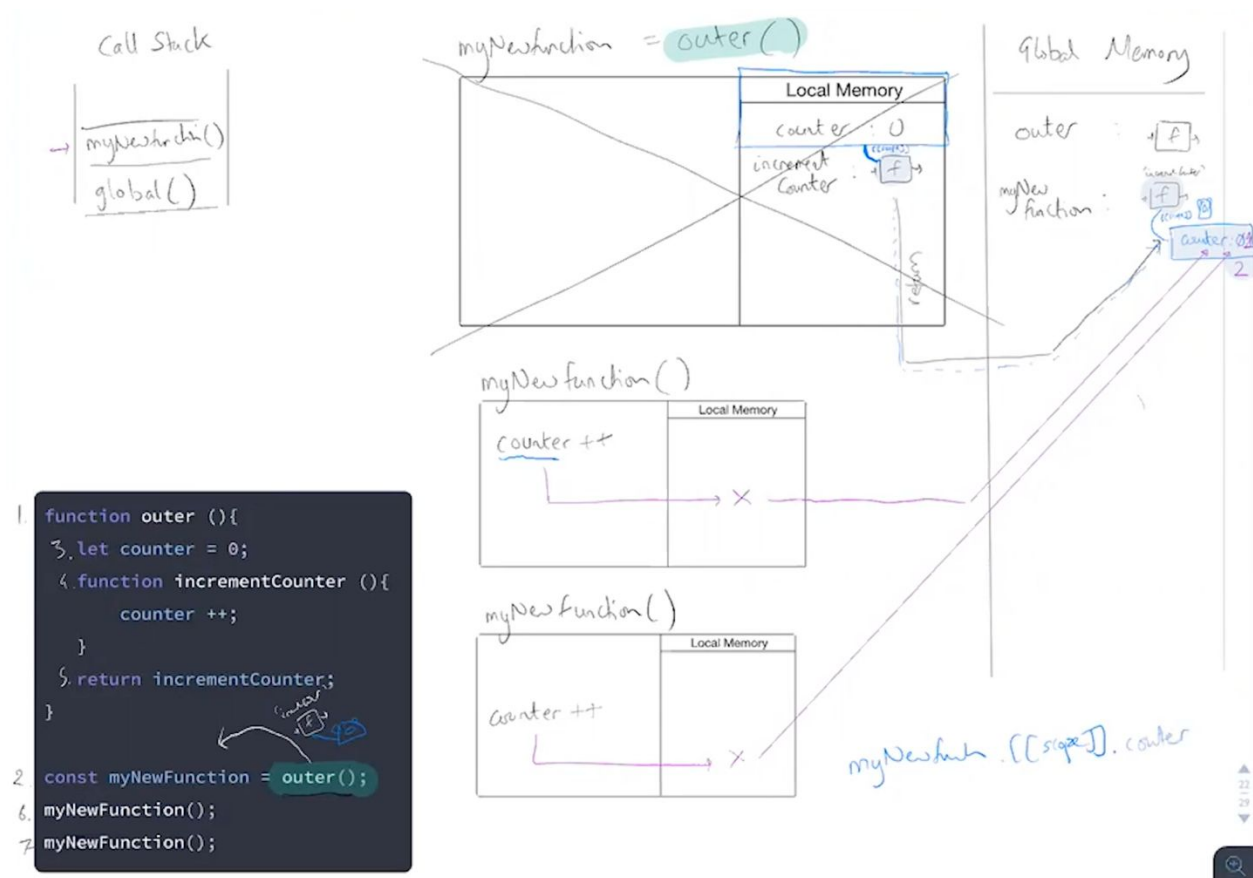


Will: I'm trying to think whether I should have you ask us questions. I'm going to answer a few of my own questions there, and then we'll come to more of your edge case questions. So, the first one I'm going to answer is, where is this backpack actually stored?

Okay, so, here's how it actually works. Let's jump back to when we were running outer. So, we're inside of outer right now, and we declared in line three counter is zero, in line four, we stored incrementCounter as a function. Immediately that we did that, that function stored got a little bond. A little bond, a little link, or a reference to the surrounding the local memory adjacent to the function.

Where the function's being saved, it gets a little link to all its surrounding local memory. All the stuff around it. It gets a little link to that whole collection, and that link is stored on a hidden property of incrementCounter. What's that hidden property called? I'm going to try and write it really small here. Oh, I have a way of doing this, watch this.





It's by the design of JavaScript. Go directly to the function that's been run into its backpack. You cannot do this. In global, you couldn't write, "myNewFunction.[[scope]].counter," that's not a thing. There's no way of getting into that backpack's data, except by running myNewFunction and inside of it having written it back when it was born as incrementCounter with code that references something that's not in the local memory of then incrementCounter, now myNewFunction, and so we go out of the backpack.

We can't access it any other way. That's actually really wonderful. That means this data is protected from being overwritten. We could, by the way, we'll talk about this later, put stuff in that backpack, update the value of the backpack by passing it in as an input, and then storing it in the backpack. We'd call that a setter argument, which would set the value of the backpack or multiple things, you could put as much as you want in there.

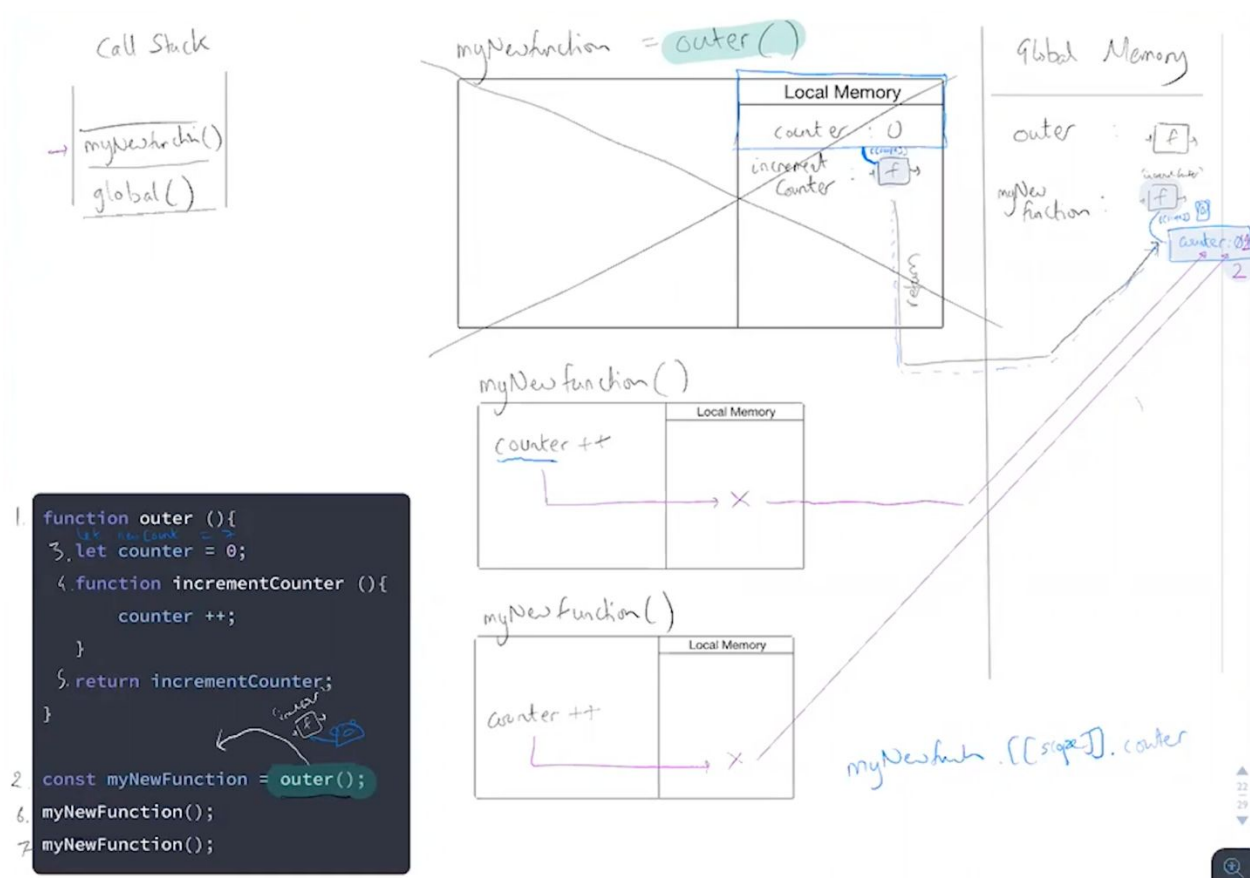
And then you could also get the data out of the backpack back to global by having written myNewFunction such that you will return, for example, counter, and now the output of myNewFunction would be, in this case, two, or in that case actually one at that point. So, we can get the data out of the backpack, but we can't get it directly, to use the function for whom that backpack belongs, run the function, have written it in

such a way as to fill in the backpack's data, or return out from the backpack. Very powerful.

Think of these almost as analogous to private data, you might have heard about in other programming languages. Data that you can't just overwrite without, willy-nilly as we say in England, I know that's an English phrase. You can't overwrite willy-nilly, you have to access and set values and get values, in a very clean, precise, restricted interface.

Interface is just a posh word for accessing data, ways... An interface is your rules for how you're allowed to access data. Your interface without data. The interface here is pretty clean. You protected the data, and the interface is by potentially passing an input to myNewFunction, that would then update the backpack's value and then hopefully you wrote myNewFunction's such that it returns out the backpack's value. So it's a protected data.

Okay, we'll talk more about that in a moment with a module pattern. What else is there to say here? Well, one other piece to say is this, "What if, folk, in my outer function, I had another variable?" Let's say newCount = 7. Well then in my local memory I would have, what? Joe, what would I have in my local memory if I said, "let newCount = 7?" When I was calling outer?



Joe: You'd have that newCount = 7 also in the backpack?

Will: Well, that's the question. Raise your hand if you'd think I'd have ... Well, I've done the answers ... Okay, raise your hand if you think the newCount with seven as the value, would that be in the backpack? Raise your hand if think it'd be in the backpack.

You'd be fair to do so, right? In fact, JavaScript's engines, which are just implementations of the rules of how JavaScript should work as agreed by the ECMAScript committee and then Chrome, their team implements as close as possible and as optimized where it's possible, those rules for how JavaScript should work, they said, "Well, you know what? We know how it should work, but do we want to have newCount is seven in the backpack?" Because Joe, what did I say was the only way I could access data in the backpack, by doing what? By running what function?

Joe: myNewFunction?

Will: Exactly. So, when I return out incrementCounter into the new global label myNewFunction, I already know its contents, so you know, the body of the function, what it can do. What's the only line of code in myNewFunction?

Joe: Count++.

Will: Do I reference newCount?

Joe: No.

Will: Can I access newCount from the backpack in any other way besides if I had rewritten myNewFunction ... Sorry, rewritten incrementCounter in a different way? Exactly. I can't.

So, I can not ever access newCount unless I make reference to it from within incrementCounter, return incrementCounter out and then in myNewFunction, now I have some reference to newCount, some use of it. But I don't use it, and there's no other way of accessing the backpack, so that would be what's called a memory leak.

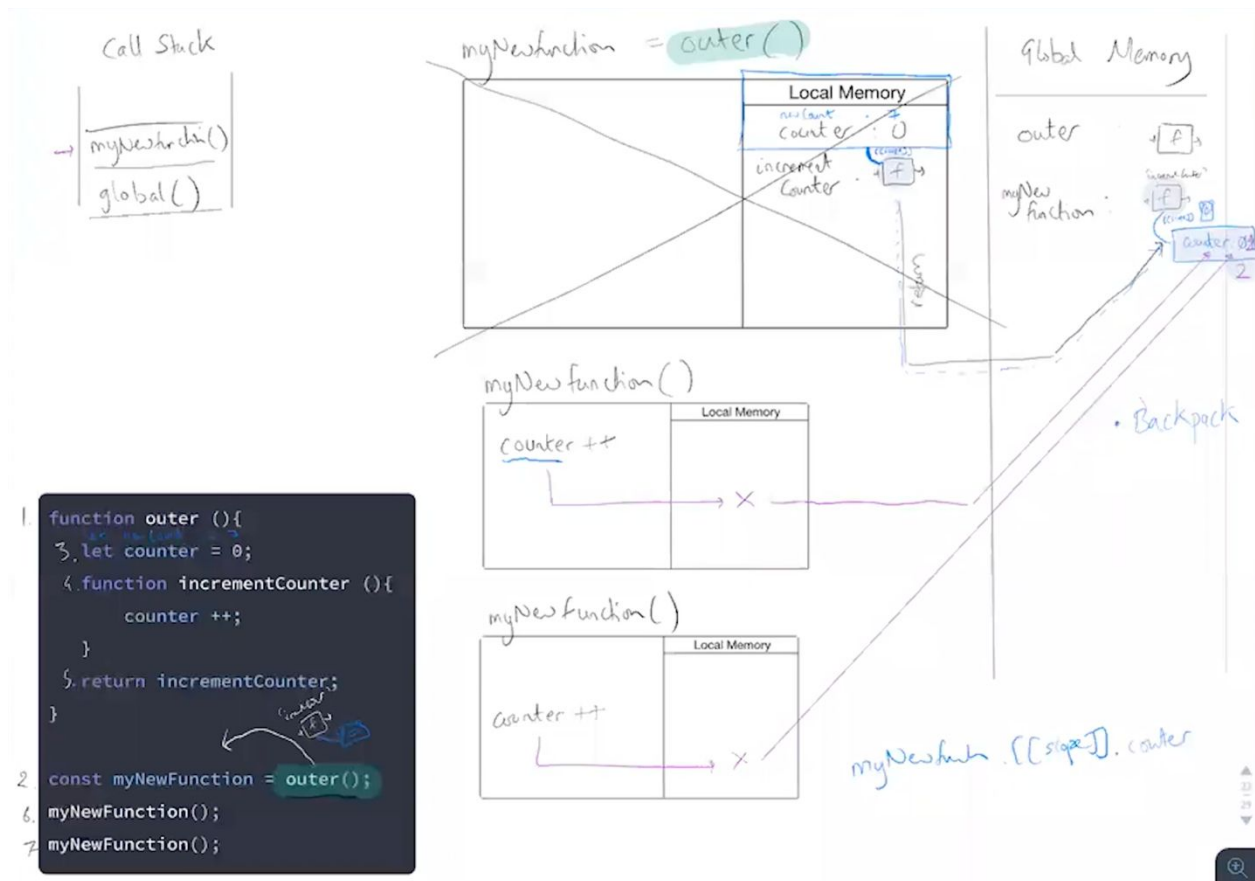
That is memory that is still being, got stuff stored in it, in the computer's memory, but we can never access, so these are redundant, it's known as a memory leak. I don't know if it would leak. I don't know. It's leaking out. It's wasted memory, like water leaking out of a pipe. It's wasted memory. We can't ever access that stuff, and so we don't ever want to even return it out.

And so, actually, Joe, JavaScript's engine, I think Chrome after version 39, something like that, said, "Hold on, let's just check the function incrementCounter that we're returning out. Let's just check it and see what it makes reference to from the local memory. Oh, it only makes reference to counter? Only return counter out the backpack. Leave everything else behind to be deleted automatically."

Joe, does that make sense? Excellent, okay. Alright. One final thing I'm going to add is all the different names for the backpack, and then I want to see if there's any edge case

questions that you folk have because I've kind of covered a lot of them, but there may be still some, I would love to hear some of them.

Lecture: 12 Closure Terminology



Will: Let's then talk about the names of the backpack. The first name, obviously the best name, is backpack. Backpack. There it is. Back. Pack. Number one name.

Number two name. Super sophisticated, but, well, not as sophisticated, but still a very good one. Let me explain it, we pull out the local memory into, along with, the `incrementCounter` function when it gets returned out. Also another name for the local memory. Some people call it - or JavaScript technically calls - the local memory not local memory. Anyone know what they call it?

Natalie: Variable environment.

Will: Variable environment, exactly. Environment just means things around me. I'm in my environment, my things around me, these are the variables, or constants and functions, etc. around me while I'm inside that function. So while I'm inside the running of `outer()`, `outer()`'s variable environment is, `counter` is zero, `incrementCounter` function.

This, here, the local memory, has an alternative name, variable environment. Okay, so when we return the variable environment on the back of the `incrementCounter` function, you can sort of consider that environment of variables, the local memory, kind

of closed over. Like you're closing the lid on it, and pulling it out, saying, "Hey, don't delete it. Just close it and bring it out with me." So there is a name for it, for the backpack, called "closed over variable environment", or the "COVE." Excellent.

But the name I love most for it is this. So, the backpack of data, got that bit. Right, it's data stored in it. It's data and the values are permanent, or persistent. So it's permanent, or persistent, data. That permanent, or persistent, data here, the backpack, is referenced here by what property? Elena, do you remember the name of the hidden property on the incrementCounter, now myNewFunction(), function? What's the hidden property called that references the backpack of data?

Elena: Scope.

Will: Excellent. The little hidden property there, `[[scope]]`. So the data is persistent and it's referenced by the scope property on the myNewFunction() function. Then here's the interesting piece. Scope is just a posh word, folk, for what data, values, numbers, whatever, do I have access to when I run my function? Or run any line of code in my application, but let's say run my function.

There are different sets of rules for what data will be available to me. You could imagine a language that says, "Only data that will be available to you is what's inside that function. You can not go grab data from outside the function, nah ah ah, no touching data outside the function. Only what's inside the function." That would be a certain scope rule, a certain rule for what data's available to me. I don't know what we'd call that, possibly functional scoping, I guess. It's very restrictive to focus only around the interior data of that function pool.

There's another type of function, another type of scope rules for what data's available to me, known as dynamic scoping, which says, "The only data that will be available to you will be the data inside the function itself", like inside myNewFunction(), well counter's not there, and wherever I'm running myNewFunction - if I'm running it in global, whatever data's in global. That's called dynamic scoping, it's basically where I execute my function determines what data will be available to me.

But there's another type of scope rule, known as static, or lexical, scoping. Let's see what lexical actually means. "Relating to the words or vocabulary of a language". Eh, great, really helpful. It means the ordering of the words on the page. So I ordered incrementCounter(), physically placed it, saved it, defined it inside the running of outer(). Therefore, when it was saved, it had access to counter is 0, and all the surrounding local memory of outer().

This is known as lexical, or static, scoping, when I never lose access to that data that was there when I was born. So when I return out incrementCounter, if my rule in JavaScript is I never lose access of data from when I was born, the physical positioning of my function definition determines - the physical positioning of my function where it was saved - determines what data I'll have available to me whenever I run it. Well, if I'm going to run it later, after returning it out of that function I initially stored it in, which

was `outer()`, I'm going to return it out, I better damn bring the data from when it was born.

That is lexical, or static, scoping, that if I save my function, `incrementCounter()`, inside of `outer()`, and return `incrementCounter()` out, I bring the data with me from when I was born. Because my language is lexically, or statically scoped, meaning what data was there when I was born is the data I will have access to when I eventually run my function. Wherever I end up running it, I never lose access to the data that was there when I was initially stored. I'm not changing what data's going to be available to me. I'm not dynamically saying, "Oh, I return the function out!", and nor I can dynamically say, "Oh! Now I'm going to have access to different data!". No. The data that was there when I was born, I'm fixed on, and the only way to do that is to bring the data with me. I bring the function out, I better bring the data with me, attached on its backpack.

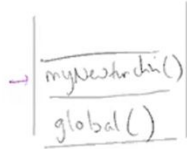
That's why it's called static; it is fixed, it is static, it can't change. What data was there when I was first born, will statically be fixed, the data that's always available to me, wherever I end up running my function. How we do that is by bringing that data with us on the back of the function.

Let's get the damn name. Permanent or persistent, lexical or static; you could call it lexical or static, lexical is the other name for it. Maybe we change it, maybe we call it static, actually. I like that name, static. I think it makes more sense. Permanent static scope reference data. So the scope is static, means the data that was available when you first saved the function is the data that's always going to be available for the rest of the life of that function. Permanent static or permanent lexically scoped reference data. What a godawful name for the backpack!

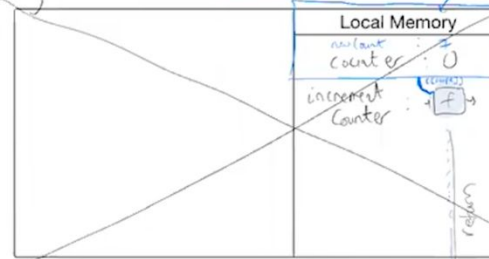
But I know if you walk into an interview and say... well, before I say that, let's talk about what developers call this backpack. What do developers call this backpack? They call it, I think the most unintuitive name. They call the backpack - wait - they call it the closure, and that's all closure is. It's a function having a permanent memory attached to it that it got by being born inside the running of another function. So should I return the inner `incrementCounter()` function out, I brought all that data with me, attached and stuck onto, now, the `myNewFunction()` function. And there it is, that backpack of permanent data is called the closure.

But imagine when you walk into an interview and say - the number one JavaScript question in the Google interview for JavaScript is "Explain closure to me." Because it is the bedrock of JavaScript, the number one question. If you say, "Ah, I prefer not to use the term closure. It's not as precise. I prefer the term 'persistent static or lexical scoped reference data'. I find it just gives more precision." Oh! [Clap] Oh my goodness. They will say, "Mum, we have waited for you. We have waited for the day when you would walk into this room." They may give you their job. I wouldn't blame them.

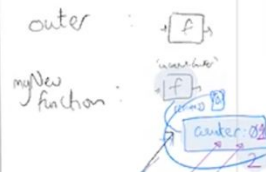
Call Stack



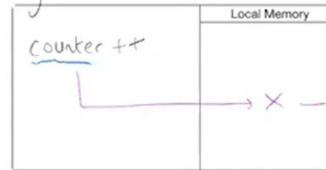
myNewFunction = outer()



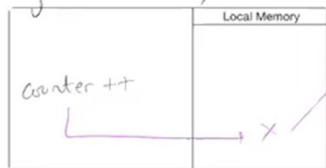
Global Memory



myNewFunction()



myNewFunction()



myNewFunction.[[scope]].counter

- Backpack
- C.O.V.E.
- P.S.S.R.D.
- Closure

```

1. function outer () {
    3. let counter = 0;
    4. function incrementCounter () {
        counter++;
    }
    5. return incrementCounter;
}
2. const myNewFunction = outer();
6. myNewFunction();
7. myNewFunction();
  
```

Call Stack

- myNewFunction()
- global()

Global Memory

- outer : A
- myNewFunction : {
 - closure: {
 - counter: 0
 - incrementCounter: f

Local Memory

- myNewFunction: {
 - counter: 0
 - incrementCounter: f

Backpack

- Backpack
- C.O.V.E.
- P.S.S.R.D.
- Closure

```

1 function outer () {
2   let counter = 0;
3   function incrementCounter () {
4     counter ++;
5   }
6   return incrementCounter;
7 }
8 const myNewFunction = outer();
9 myNewFunction();
10 myNewFunction();
  
```

myNewFunction() Local Memory

- counter ++

myNewFunction() Local Memory

- counter ++

myNewFunction . [[scope]]. counter

Joe: Yeah, so two things. When is the scope, is that created when the functions created or is it created when it's returned?

Will: Great question. It's created immediately the function is saved. As soon as the function's saved, defined, created, declared, whatever you want to call it. It immediately has a property that links to the surrounding local memory. Not a copy of local memory, not a copy, it's just a link to it. When you return the function out, normally when you return something out of execution context, the only thing that gets returned out is the value you return out, everything else gets deleted. Not this time because you still got a link to all that surrounding data so JavaScript says, "Oh it's got to be linked to, you can't delete that and leaves it there attached in the scope property." Great question, Joe.

JavaScript normally deletes all the stuff in local memory of an execution context when you return out from that execution context. The only thing it holds onto is the returned out value. The number, the thing, whatever, and everything else gets deleted. Known as automatic garbage collection. Garbage because it's data you can't ever access again. You can't go back into an execution context, so it's garbage, it's wasted. So it automatically deletes it. This time the garbage collector, as is the JavaScript engine's name for the tool by which it chooses whether to delete stuff or keep it in there, automatically knows not to delete all that data because the returned out function has the little scope property that links to all that surrounding data. And so it says, "Oh, don't delete that stuff." Now it also does a little bonus check. It checks in the function that you return out what things are ever referenced in that function. And it will only return out in the backpack stuff that was referenced in the function. Everything else does get deleted. Pretty sophisticated automatic garbage collector.

Joe, you want to ask your follow up?

Joe: Yeah, sorry. I think the answer is yes, of course. I just want to check to make sure I'm understanding 100%. Could you move line three between lines four and five and have it work still perfectly no problem?

Will: Absolutely, because you're linking to that surrounding local memory. Until you return that function out everything you store in that local memory is going into what could go into the backpack. Now if you don't ever reference it in the return, that incrementCounter function, it will never show up in the backpack. Suppose we were to have... Let's just change this up slightly let's put exactly as you say Joe let's put, "let Joe = true" and then suppose inside of incrementCounter we were to do, "Joe = false." Well then in the backpack, or in sorry the local memory, we would have "Joe is true" and then the link would be to this entire surrounding local memory, Joe. When you return out the incrementCounter function, even though it was defined earlier that Joe is true, the link's to the entire surrounding local memory. So when you return out incrementCounter into myNewFunction you bring the whole surrounding local memory as long as you make reference in the returned out function to the stuff in the local memory, which I did. I said, "Joe is false." And so out returns a function that brings "Joe is false" stored in the backpack. Does that answer your question, Joe?

Excellent. Good. Natalie, your question?

Natalie: Yeah, my question is when-

Will: As this is going to be recorded I feel like I should be in the camera when you ask a question.

Natalie: Ok!

Will: I'm in.

Natalie: You're back?

Will: I'm back.

Natalie: Ok, so when JavaScript does the check to see if which variables are referenced in the rest of the code base and then from that determines-

Will: Just to be clear, just to be precise, the variables that are referenced inside the function that's being returned out.

Natalie: Yes, okay. So, when JavaScript decides which variables within that function that's being returned out to save in the backpack in which that can be picked up by the automatic garbage collector, was that the term?

Will: Yep.

Natalie: I guess, my question is, when exactly does that happen? You said that this backpack can be created when the function is saved. So is it the function definition is saved and then JavaScript goes, "Okay now I need to check to see-"

Will: Oh! Interesting! Honestly, Natalie, what an interesting question. I think it doesn't matter. The questions that matter are the ones where it would have a consequence either way for the code you write. I can't see what consequence that would happen because you're going to return the function out either way. So if the check is done on the return out stage, or the declaration stage of the function, it doesn't really matter. But that's a super interesting question and I assume at the return stage moment, but this is a very specific implementation feature, this is definitely not JavaScript spec, that means the rules of JavaScript. This is the engines implementing JavaScript. Their team would have gone, "What is more performant? What optimizes people's code better? Should we check that immediately on the function saving, declaration moment, or on its returning?" And they'll have decided based on what is more performant and efficient. I don't think it has consequences for our code as we write it but the engine's developers will have thought about that question. Very cool, cool question. Jim, go ahead.

Jim: I'm curious as to how to reconcile all this stuff with what you were talking about earlier with the heap. Do you want to answer now or are you going to cover that in the next bit? Or should I just ask it?

Will: I'm not Jim, go ahead ask it for sure.

Jim: Okay, what you're saying about the heap is that functions are basically passed by reference, which would suggest that if I instantiate myNewFunction, or, excuse me, if I instantiate outer again-

Will: Jim, be really careful when you use the word instantiate. Do you mean when we call outer?

Jim: Let me say it like, line eight I write, "const myNewerFunction = outer"

Will: That is coming on the next slide. You want to hop over to that?

Jim: My other question is can you actually see double bracket scope in the dev tools if you actually look for it?

Will: No. We're going to have to re-answer this question again because I said no and then somebody once emailed me and said you can. I've tried to see. I can't see where you can see it. If you can. If anyone can see it. I don't know... Somebody told me, "Yes you can see it." Elena, looks like you have a comment on this. Have you managed to sort of introspect the function and see the square bracket scope reference on a function definition? I have not. Have you seen it Elena?

Elena: No, I had the same question as Jim.

Will: I have not been able to see it. I will say this. If you folk pause in the Chrome dev tools the code running at this moment. So you're running myNewFunction and you set it to pause on the counter ++ line. It'll show on the right hand side, closure counter 1, or closure counter 2, closure counter 0, whatever. It will actually visualize the backpack's data. And same in VSCode. You'll be able to see the backpack's data if you pause it at the moment where you're inside the function that has the backpack. You'll be able to see and visualize. It will actually say "closure" and then it will have the stuff in the backpack. If you pause inside the running, in this case of myNewFunction, the function that has the backpack. You pause inside of it, it will show you, visualize, on the right hand side, or in VSCode on the left hand side the debugger, the backpack's data.

Alright, excellent questions. Jim, we'll come to yours in a moment. Kate, did you have one? Okay. Elena, go ahead.

Elena: Just two clarifying questions. Joe said before if there is no return keyword, I still have this code, correct?

Will: Say that again?

Elena: If there is no return keywords-

Will: Inside where?

Elena: Inside the function outer, let's say.

Will: Ah, yes. In other words, you're saying all functions get this by default. Even if they're not returning anything.

Elena: Yeah.

Will: Correct, but it's just not very interesting because the data hasn't been deleted around the function anyway so it just looks like it's there.

Elena: Ok, and the other question. Right between line one and line three, we have a new variable, let newCounter, right? And we do not reference to it in our function incrementCounter, but then we decided to change the code, and we say newCounter ++. Does it mean in that case my backpack will automatically change?

Will: Hold on, which code are we changing? incrementCounter's?

Elena: Yes.

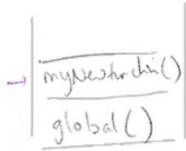
Will: Ah, so you only get to save a function one time. If you think about, you never change its contents. That's a really thoughtful question. I'm really glad you asked that question. Everyone's thinking that question. That is a brilliant question to ask. I would have had to bring it up. You can't change a function's code. You can't add an extra line to it. You can't say, "Push an extra line of code to that function." The only way you can change the function's code, once it's returned out, would be to re-declare myNewFunction, in which case you would overwrite the function and its backpack. There would be no backpack attached, it'd be gone.

And if you were to change it inside when incrementCounter is first declared, inside of outer, we only get to save it once. I guess we could re-declare it immediately below but we haven't returned it out yet so the connections not even made. It doesn't matter so much whether it's immediately returned done or not, the function you declare now, if it now references newCounter well then newCounter will be returned out to the backpack. Once we return out the function, Elena, we can't then go, "Oh changed our mind what that function's going to have in it; make sure it references newCounter." We can't do that. We cannot change a function definition once it's been saved. So when you return out the function, that's the final version that function's going to take.

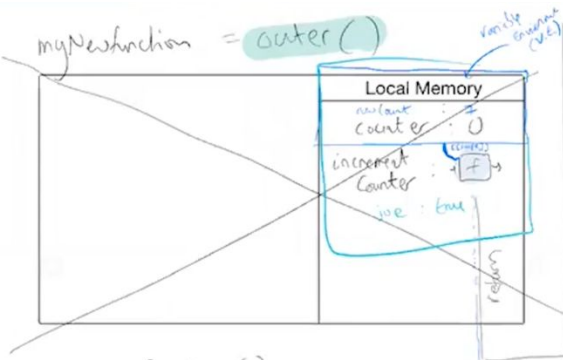
Really interesting, right? You can push to objects, you could remove from objects, you can add elements to arrays, remove elements from arrays... You can't change the contents of a function. One time saving. It's really interesting. Ok does that answer the question, Elena? Perfect.

Alright, final piece, here it is. Amazing patience from all of you. Here's the final piece.

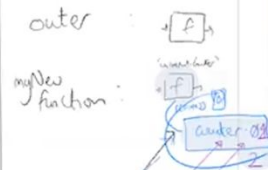
Call Stack



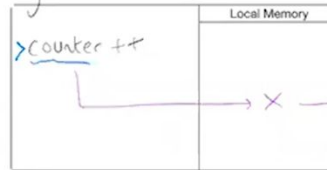
myNewFunction = outer()



Global Memory



myNewFunction()



myNewFunction()



myNewFunction.[[scope]].counter

- Backpack
- C.O.V.E.
- P.S.S.R.D.
- Closure

```

1. function outer () {
    3. let counter = 0;
    4. function incrementCounter () {
        5. counter++;
    }
    6. return incrementCounter;
}

2. const myNewFunction = outer();
6. myNewFunction();
7. myNewFunction();
  
```


Lecture: 14 Closure Summary

The bond

When a function is defined, it gets a bond to the surrounding Local Memory ("Variable Environment") in which it has been defined



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

Will: It's worth probably a quick reiteration. So, incrementCounter. Summary for the recording, summary. We declared outer. We called it, we ran it. There it is. We're invoking outer. Into it we go. Into we go. We saved counter as zero. We saved incrementCounter as a function. As soon as we do so, it gets a bond to the surrounding local memory, there it is. Surrounding local memory. Variable environment in which it was defined, which incrementCounter was defined.

The 'backpack'

1. When *incrementCounter* is defined inside *outer*, it gets a bond to the surrounding Local Memory in *outer*
2. We then return *incrementCounter*'s code (function definition) out of *outer* into global and give it a new name - *myNewFunction*
3. BUT we maintain the bond to *outer*'s live local memory - it gets 'returned out' attached **on the back of** *incrementCounter*'s function definition. So *outer*'s local memory is now stored attached to *myNewFunction* - even though *outer*'s execution context is long gone
4. When we run *myNewFunction* in the global execution context, it will first look in its own local memory for any data it needs (as we'd expect), but then in *myNewFunction*'s 'backpack'



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

I'm saying it all over again here. When *incrementCounter* is defined inside of *outer*, as we run *outer*, do not ever think it's done as we first run this code here. This is line one, this is line two, this is line three, this is line four, this is line five, this is line six, and I guess this is line seven, but we don't go back in. Do not think we go back in. We don't go back in.

The 'backpack'

1. When *incrementCounter* is defined inside *outer*, it gets a bond to the surrounding Local Memory in *outer*
2. We then return *incrementCounter*'s code (function definition) out of *outer* into global and give it a new name - *myNewFunction*
3. BUT we maintain the bond to *outer*'s live local memory - it gets 'returned out' attached **on the back of** *incrementCounter*'s function definition. So *outer*'s local memory is now stored attached to *myNewFunction* - even though *outer*'s execution context is long gone
4. When we run *myNewFunction* in the global execution context, it will first look in its own local memory for any data it needs (as we'd expect), but then in *myNewFunction*'s 'backpack'



```
1. function outer (){  
  {let counter = 0;  
  {function incrementCounter (){  
    counter ++;  
  }  
  {return incrementCounter;  
}  
  
2. const myNewFunction = outer();  
6 myNewFunction();  
myNewFunction();
```

That's because we stored `myNewFunction` and `incrementCounter` in memory as `myNewFunction`. Okay, who knows if we'll actually include this section as I reiterate. When `incrementCounter` is defined inside of `outer` it gets a bond to the surrounding local memory of `outer`, its variable environment. We then return `incrementCounter`'s function definition, lose its label out of `outer` into `myNewFunction`. Give it a new global label. We lost its label. Return out into `myNewFunction`. We then run, what was formally known as `incrementCounter`, now by the label `myNewFunction`, but that return out `incrementCounter` function maintained its bond, its connection, its reference, its scope property link to the surrounding live local memory of `outer`.

The 'backpack'

1. When `incrementCounter` is defined inside `outer`, it gets a bond to the surrounding Local Memory in `outer`
2. We then return `incrementCounter`'s code (function definition) out of `outer` into global and give it a new name - `myNewFunction`
3. BUT we maintain the bond to `outer`'s live local memory - it gets 'returned out' attached **on the back of** `incrementCounter`'s function definition. So `outer`'s local memory is now stored attached to `myNewFunction` - even though `outer`'s execution context is long gone
4. When we run `myNewFunction` in the global execution context, it will first look in its own local memory for any data it needs (as we'd expect), but then in `myNewFunction`'s 'backpack'



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

2. const myNewFunction = outer();
6 myNewFunction();
   myNewFunction();
```

It gets returned out attached from the back of `incrementCounter`. So `outer`'s local memory is now stored, attached, it gets returned out on the back of `incrementCounter` but we lose the label of `incrementCounter`, it gets stored into `myNewFunction`. So `myNewFunction` is function definition, formally known as `incrementCounter`, plus a backpack of live data from where that function was born which contains `counter` as zero.

So, even though `outer`'s execution context is gone, we exited out of it. We brought all of its data with us and stored it globally in `myNewFunction`. When we run `myNewFunction` in the global execution context, it will first open its local memory but any data it needs, but then it will look into `myNewFunction`'s backpack, which is the originally stored state, posh word for live data, from when `incrementCounter` was saved originally inside of `outer`.

Alright, this is why I love diagramming. That was a worded version of the diagrams. Don't you think I love that? But whatever. Alright, well, we talked about the backpack,

we talked about closed over variable environment, persistent lexical or static scope reference data, backpack, closure.

What can we call this 'backpack'?

1. Closed over 'Variable Environment' (C.O.V.E.)
2. Persistent Lexical Scope Referenced Data (P.L.S.R.D.)
3. 'Backpack'
4. 'Closure'

The 'backpack' (or 'closure') of live data is attached incrementCounter (then to myNewFunction) through a hidden property known as `[[scope]]` which persists when the inner function is returned out



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

Lecture: 15 Closure Multiple Closures

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

```
const myNewFunction = outer();  
myNewFunction();  
myNewFunction();  
  
const anotherFunction = outer();  
anotherFunction();  
anotherFunction();
```

Individual backpacks

What if we run 'outer' again and store the returned 'incrementCounter' function definition in 'anotherFunction'

21

Will: Alright. Now we come to Jim's question. Individual backpacks. What if we ran outer, all over again, saw the return incrementCounter function definition in another function?

Folks, this is it, we're going to walk through it all, because this is the recorded version, we're going to walk through it all, one final time. It's going to feel [sigh], but because it's the recorded version, I think it's beneficial for everybody. Normally, we just jump to the second call of outer, we're going to walk through the whole thing all over again, because this is the version for posterity. This is posterity's version. Here we go.

Hopefully that's visible. Not risible. What a nice word, risible. Risible. Good! What are we talking about? Here we go! So we're going to define outer, we're going to run it once, store its result, hope it's a function in myNewFunction. We're then going to run that hopefully result function twice. Hopefully update its backpack values. Run outer again. Store its function in another function. It's return function in another function. It's return increment, and then run another function a couple of times. What's this gonna do? Here it is people. This is it all coming together. This is the final code we see today. Every last piece. Here we go.

Line one. What are we doing, Natalie?

Natalie: In line one, we are declaring a function with the label outer.

Will: Beautiful.

Natalie: Setting it equal to the whole function definition.

Will: Very nice. Next line, Kate.

Kate: Then we are in global memory creating a const called myNewFunction.

Will: Excellent. As soon as we start running our code, our call stack has global on the bottom. Ok excellent. We're still in global and now as you say, Kate, we hit the declaring of the const myNewFunction. What do we save in it?

Kate: Well it is going to start out uninitialized but it's gonna be saved whatever the function invocation of outer returns.



The diagram illustrates the memory state during the execution of the provided code. On the left, a 'Call stack' is shown with a single frame labeled 'global()'. On the right, 'Global Memory' is shown with two entries: 'outer' pointing to a function object (represented by a box with 'f' and arrows) and 'myNewFunction' which is currently empty. Below these diagrams, a code snippet shows the definition of 'outer' which returns 'incrementCounter', and the subsequent declarations and calls to 'myNewFunction' and 'anotherFunction'.

```
function outer () {  
  let counter = 0;  
  function incrementCounter () {  
    counter ++;  
  }  
  return incrementCounter;  
}  
  
const myNewFunction = outer();  
myNewFunction();  
myNewFunction();  
  
const anotherFunction = outer();  
anotherFunction();  
anotherFunction();
```

Will: Just, for folks listening, that is gold technical communication and just do that. Spot on. Got to go and can call outer to get our result. Everyone's falling asleep in the audience right now and so we will have to say it together. We all will say it together to invigorate our audience. Yes, we feel slightly embarrassed by it. Because it's quite awkward; maybe you regret doing this? But everyone's mute's off and we are going execute outer and create a brand new-

All: Execution context!

Will: Yes, it sounds even better when you do it with such dry wit. Excellent, there it is. There's our execution context for the running of outer. We put out on top of the call stack. There it is, nice. Alright, into it we go. First thing we do inside, Elena?

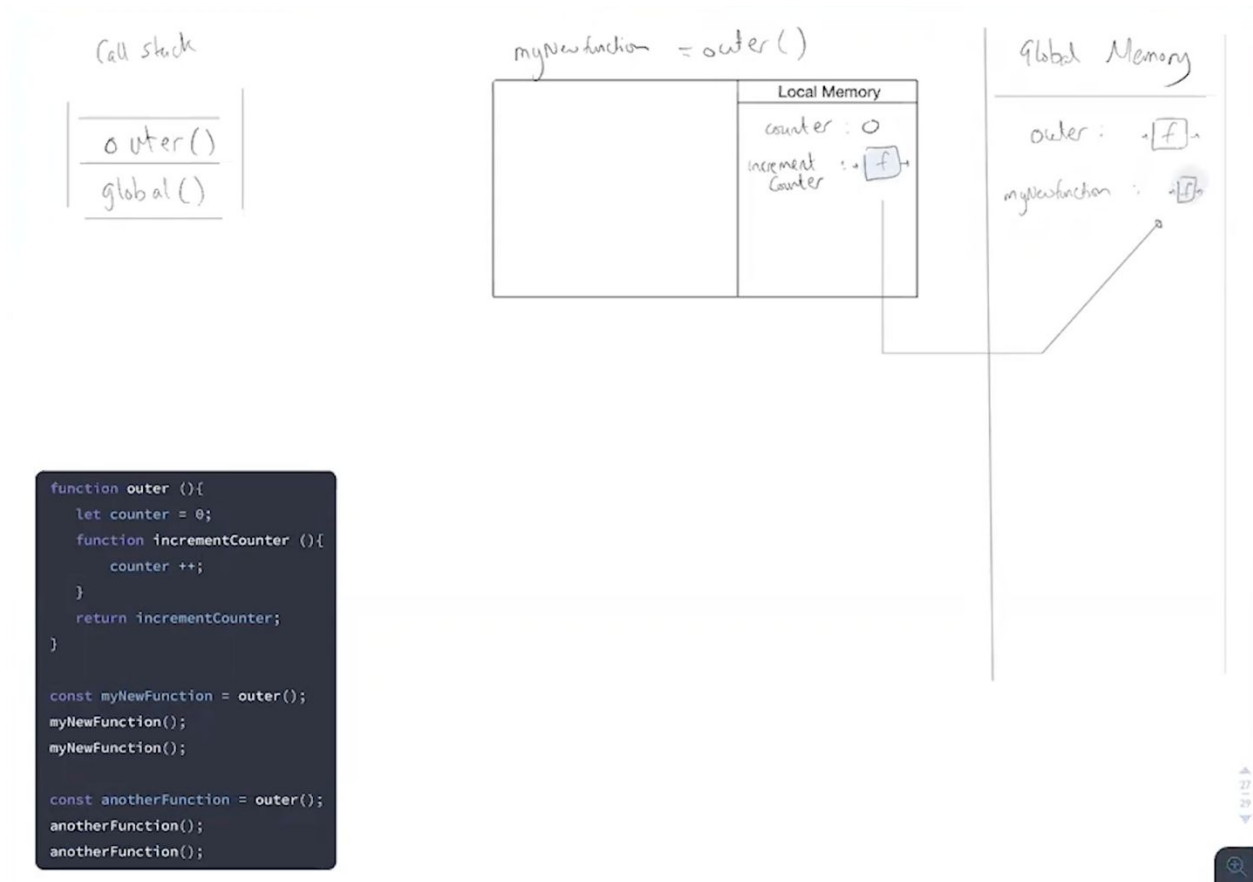
Elena: We are declaring the new variable counter and set it equal to zero.

Will: Thank you Elena, next line?

Elena: We are declaring incrementCounter with the function declaration.

Will: Beautiful, there it is, and are we gonna call that function, Elena, or something else?

Elena: No, we're not. We are going to return the function definition in myNewFunction.



Will: Beautiful. Beautiful. There it is. Into myNewFunction we return our function formerly known as incrementCounter. Now goes by the new global label which is what, Elena? What's the new global label? Just remind us again.

Elena: myNewFunction.

Will: myNewFunction. There it is. Beautiful. Thank you, Elena. I'm just going to note that it was formerly known as, I'm going to do my new thing. I'm going to note that It's formerly known as incrementCounter.. Beautiful. There it is, incrementCounter! Excellent. Now we come and pop off outer from our call stack and we go back to global where we hit, what's our next line? Natalie, we hit our next line which is the call to?

Natalie: To invoke myNewFunction.

Will: Beautiful. We do what to our call stack as we invoke myNewFunction?

Natalie: We push myNewFunction onto the call stack.

Will: Oh! Natalie, we forgot something. This would have been a great opportunity to get this. When we declared incrementCounter, inside of our running of outer, anyone want to tell me what actually happened at that moment?

Natalie: In that moment the backpack was created? Or JavaScript looks-

Will: I think that the backpack, Natalie, is when it gets returned out it brings selectively in the backpack. It's what hidden property on the incrementCounter function is immediately set and links to what?

Natalie: Scope?

Will: Beautiful.

Natalie: Yeah.

Will: Fantastic. We immediately get the scope property, which is a link to the surrounding local memory. There it is. Very nice, and so when we return our incrementCounter we bring with it that backpack, the link to the surrounding local memory, and we bring out here everything because everything is referenced in the returned out function. Out comes the backpack, attached onto the incrementCounter function through its hidden scope property. There it is. So myNewFunction is a function definition, the function formerly known as incrementCounter, plus a backpack of live data, counter is 0. Beautiful. Thank you, Natalie. Now we hit the running of your returned out, now we hit the running of myNewFunction. New execution context, let's put it in, do a little tiny one, tiny, tiny, tiny, tiny one, there it is for myNewFunction. You run it and, Natalie, what's the first line of code inside of it?

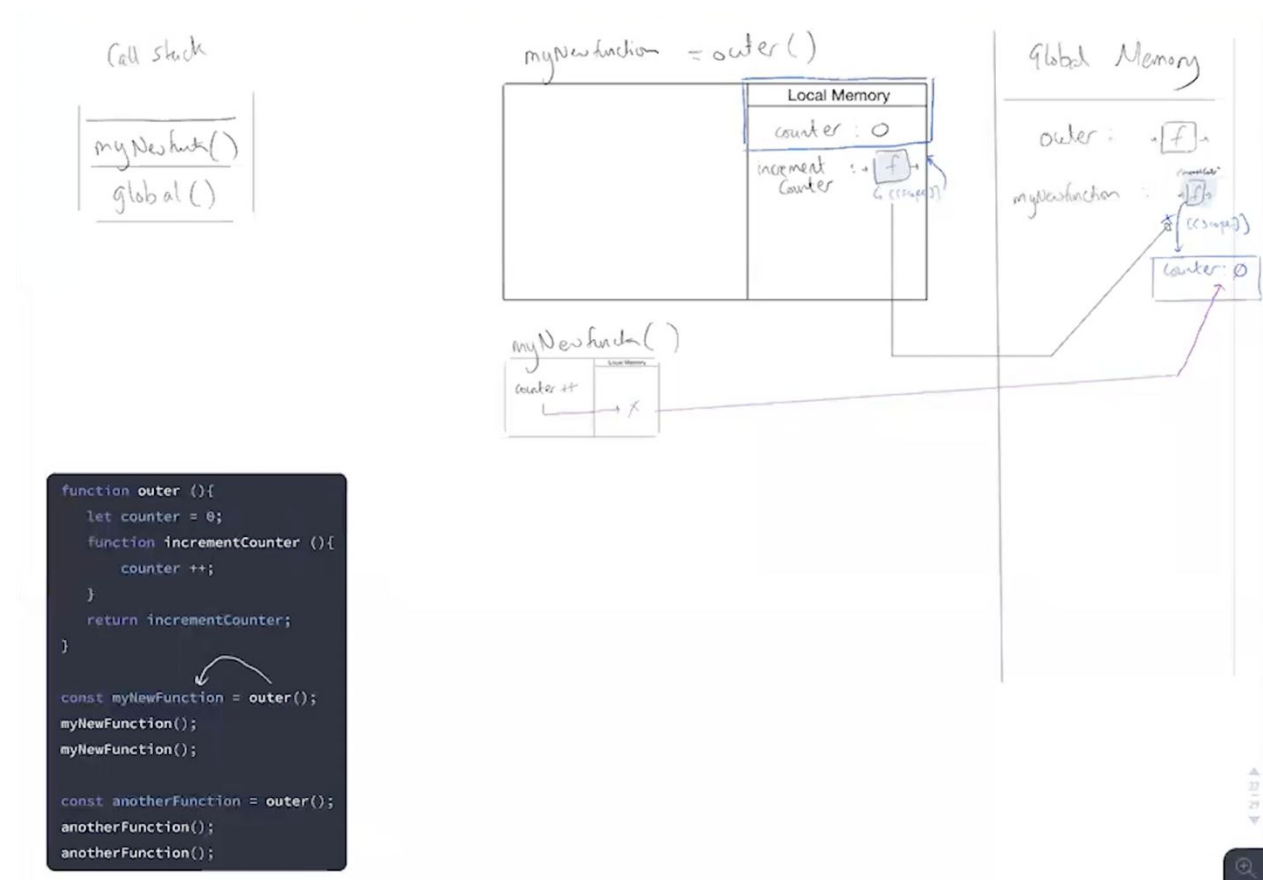
Natalie: Counter ++.

Will: Counter ++. I'm going to use my purple pen for look up. Talk me through the look up, Natalie.

Natalie: So first we look in the local memory of myNewFunction, and we don't find it. We don't find the variable counter so then we look to the backpack of myNewFunction, and we do find counter.

Will: Fantastic. There it is. I don't know if I'm allowed to do it that way. And what do we do to the counter?

Natalie: Increment it by 1.



Will: Fantastic, phenomenal, exactly right. And that's it; that's all what that function does. We exit out of it, we go back to global, because it's next on the call stack, where we hit what now, Kate?

Kate: We have another invocation of myNewFunction.

Will: Very nice, put it on the call stack, give me a second. Beautiful. Kate, we create a new? Create a new what, Kate?

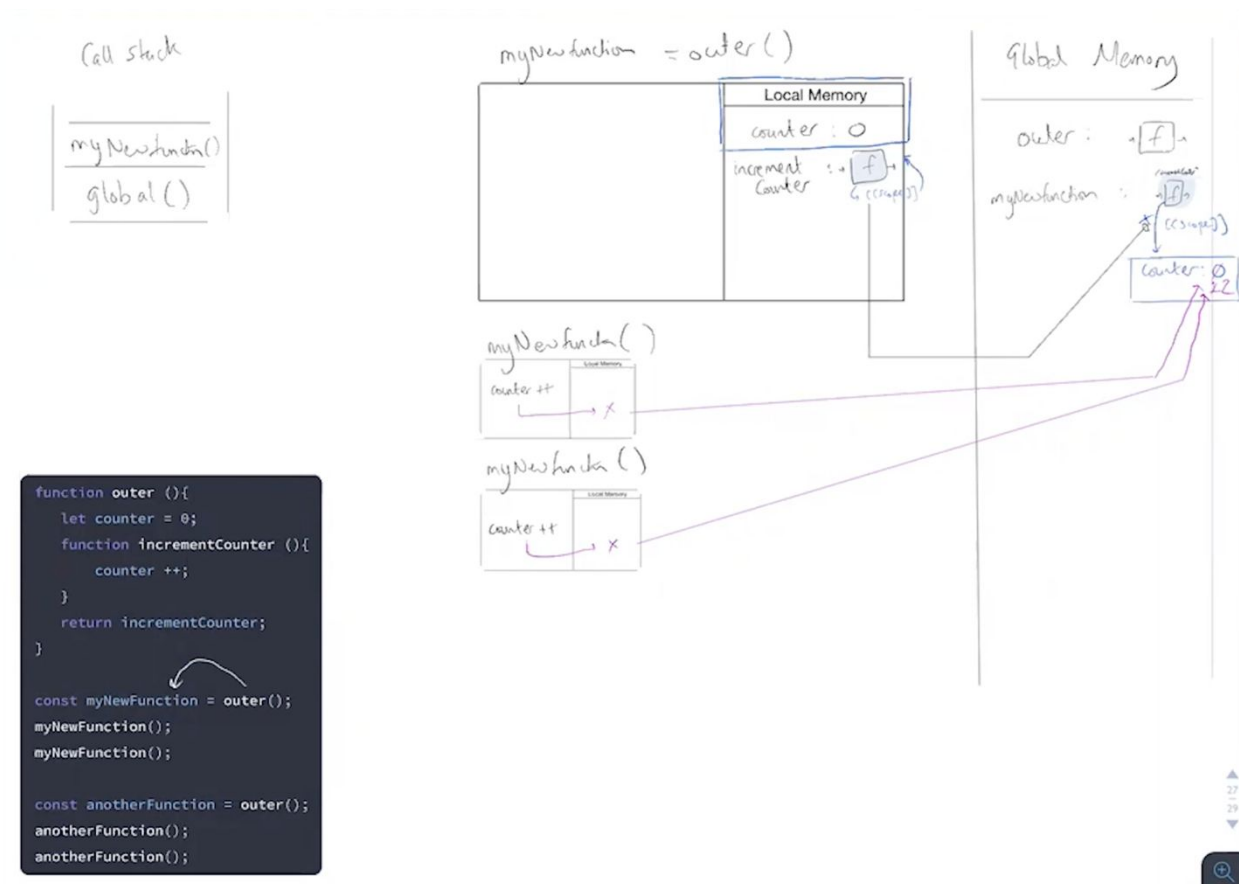
Kate: Oh sorry, my audio got messed up. Create a new execution context.

Will: Into it we go. Where the first line we hit inside of it because it's myNewFunction is just incrementCounter's code, the first line says to do what, Kate?

Kate: It looks for the variable counter, and it looks in the local memory for that, since it doesn't find it; so then it looks to the closed over variable environment.

Will: Very nice, yes, the closed over variable environment where it finds counter is?

Kate: It finds that it's now equal to 1, so then it increments it to 2.



Will: Fantastic, fantastic. Ok, awesome, how fast it becomes so intuitive, but now we move onto another call to `outer`, interesting, so we pop off the code of `myNewFunction` from the call stack. We go back to global, where we hit, what line, Joe, take me through this line.

Joe: We have another function in global memory, which-

Will: Another function declared, which is gonna be, the result of calling what function?

Joe: `Outer`.

Will: I shouldn't say another function, we have a label by the name 'another function', which we don't know what it's even going to have in it yet, until we call what, Joe?

Joe: Outer.

Will: Beautiful, so for now it's uninitialized while we go off and call outer. Let's do it, create an execution context for it. Shrink it down a bit.

Ok into it we go. Jim, we jump into the next call of outer, I know you are particularly interested in this, so let's see what happens. Next call of outer, and what's the first thing it say to do, Jim?

Jim: It declares the variable counter and sets it equal to 0.

Will: Love it! Next line.

Jim: It declares the function incrementCounter and it returns the functionality of incrementCounter back out into anotherFunction.

Will: One second, even though you're spot on, that's exactly right. There it is. Into the label anotherFunction, that's exactly; I like that phrasing as well. There it is. Into the label anotherFunction. Oh, we missed it again! We missed it again!

Jim: Oh excuse me, yes, and it also creates a backpack.

Will: Create a bond to-

Jim: Creates a bond to, sorry, what's it called again? It's called double bracket scope.

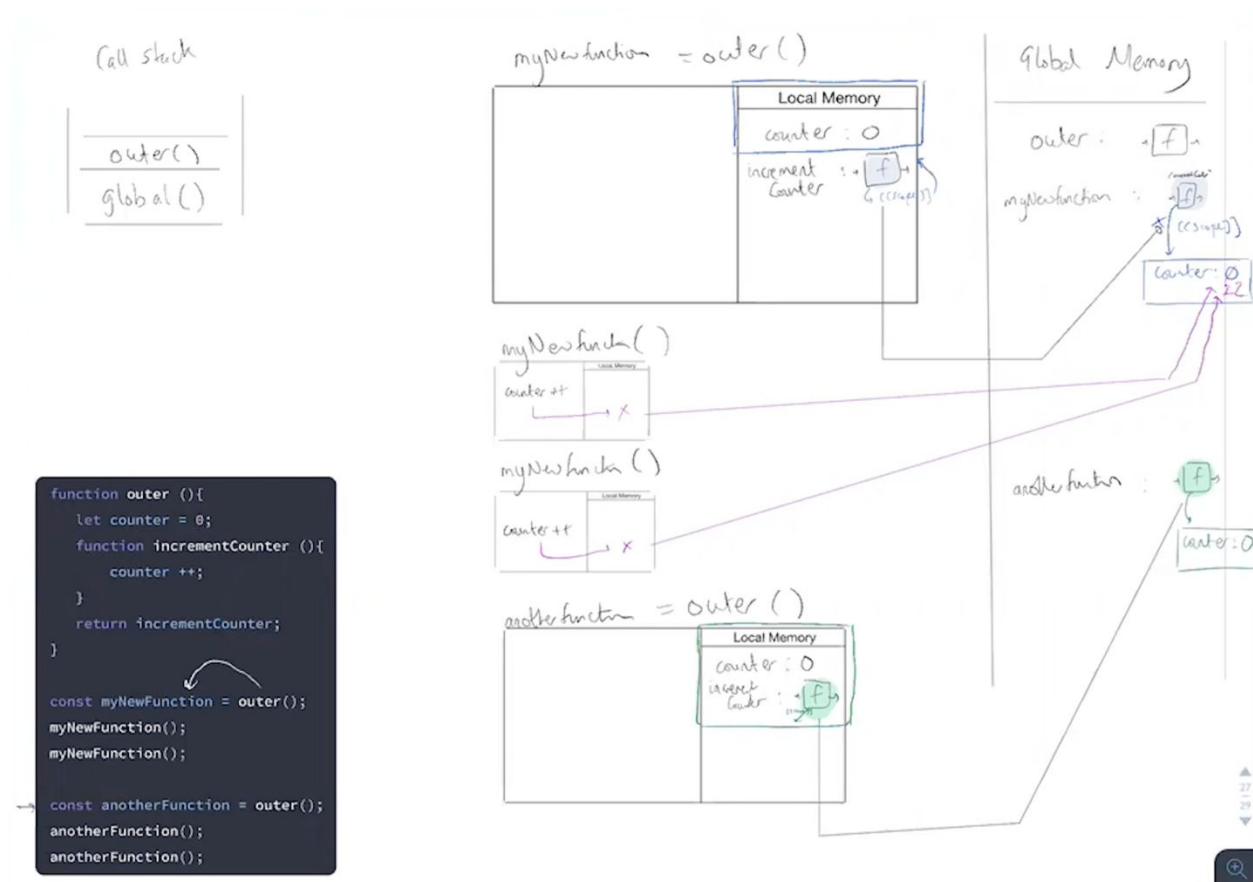
Will: That's right, square brackets, scope, square brackets is fine. Let me just, hold on, let me just zoom in on this. Here we go, so square bracket, square bracket, scope, square bracket, square bracket, which is?

Jim: It contains a new instance of counter assigned to 0. Instance is probably not the right word, but yeah, a new counter that equals 0.

Will: If it's a link to the surrounding local memory, where counter is zero, and yes it is a, separate I guess. Surrounding local memory, there it is.

Alright, and so out returns the incrementCounter function into the new global label, anotherFunction, but it also brings with it, yes-

Jim: A closed over variable environment or a backpack.



Will: A closed over variable environment, ok, beautiful. And now we exit, we pop outer off the call stack, let's keep going, we're almost there there folks. And where do we put this? I guess we kind of squeeze it in here; we're going to call anotherFunction. Ok, you know what, we're going to call it again. And now I'm going to ask each of you, what is going to happen, suppose inside my myNewFunction and anotherFunction, not just incrementCounter, if I would console.log it, after my increment. What would I see, Kate, calling myNewFunction once, call it again, calling anotherFunction once, calling it again, running anotherFunction again, what values for counter, what values would I see in my console if I console.log counter, from those four functions called, myNewFunction, myNewFunction, anotherFunction, anotherFunction and tell me Kate why that would be the case?

Kate: Ok so if your console.log is on the line after the counter ++ line, then those four function calls will result in four lines, which are 1, 2, 1, 2.

Will: Is she right? She's so right! Kate, tell us why?

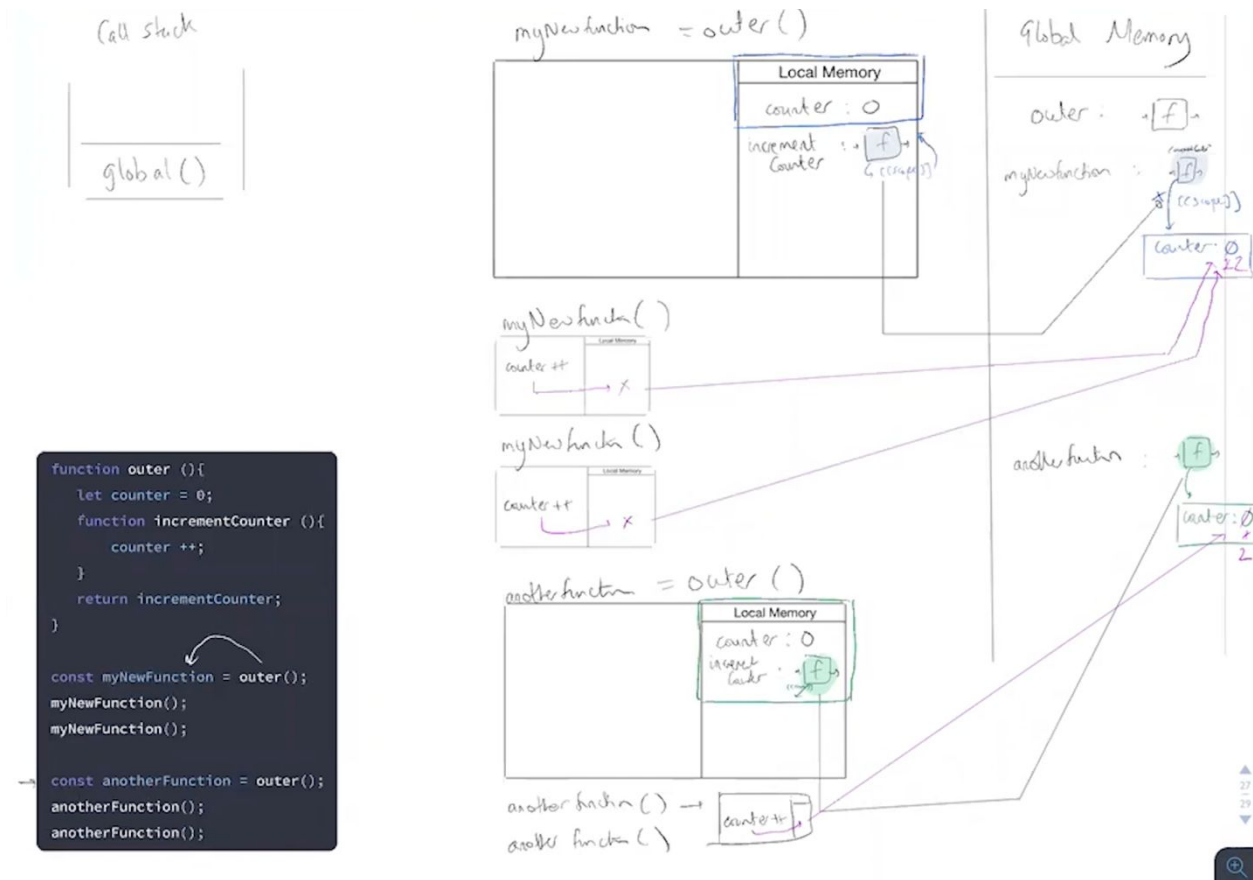
Kate: Well, because the first time when myNewFunction is run, the variable counter, starts at 0, then it gets incremented to 1, and then that gets console logged out, that value of 1, so and so forth.

Will:

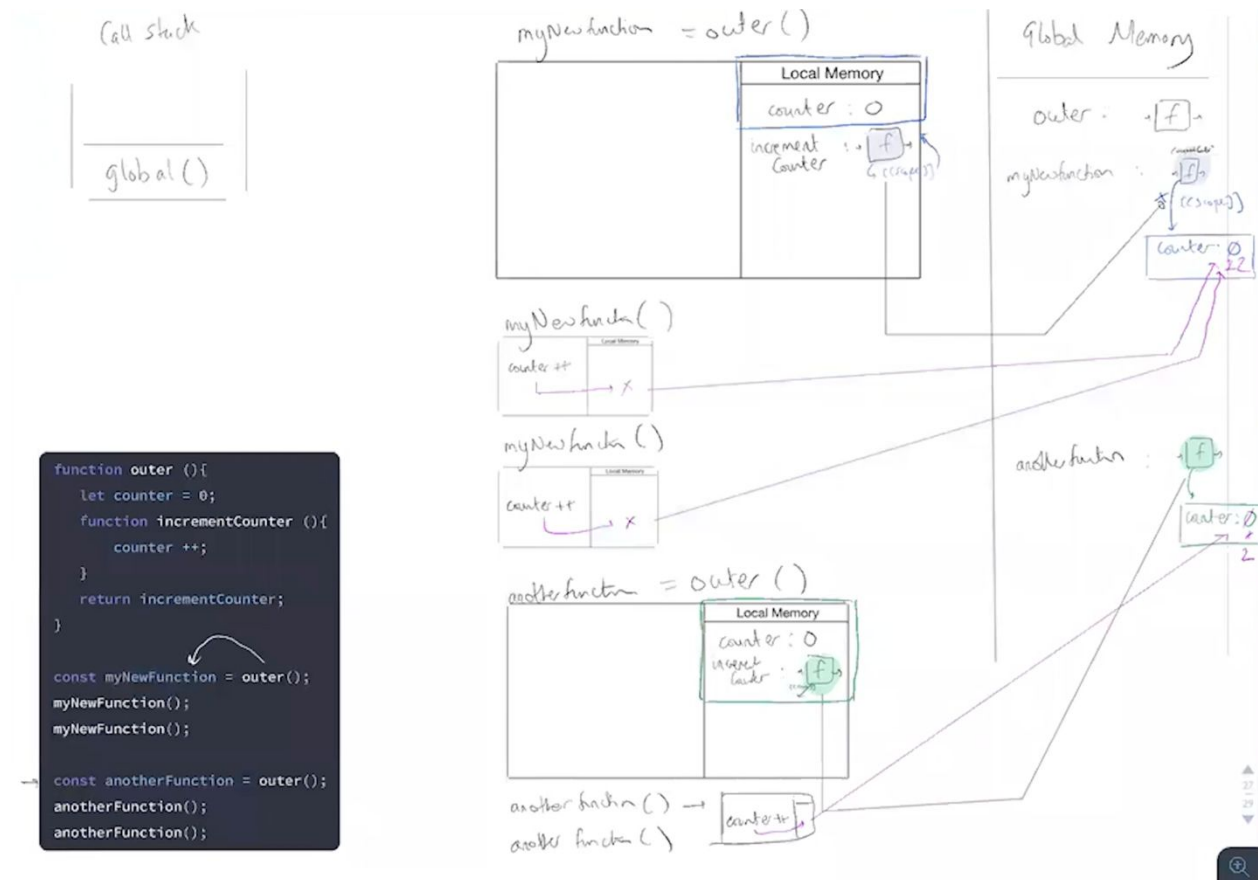
So, it's because myNewFunction, which is the incrementCounter returned out of the first call to outer, and the distinct backpack that came with it from that first outer's execution context, is where, when we run myNewFunction, we go and speak to that backpack of myNewFunction. When we run outer again, we create a brand new execution context with a brand new local memory with counter as 0. A brand new incrementCounter, a brand new position - no - just to be really clear people, these are not the same. They are totally, separately stored in memory. They may look the same, totally separate.

We return out the incrementCounter declared in the second running of outer. There it is, return it out, bring with it its own brand new backpack. There it is. And so when I run anotherFunction, and it says inside of it let's just make sure we see this, run anotherFunction and inside of it, creates an execution context, which says counter ++. We look in its local memory, nothing there, and we head up to its backpack where we find counter is 0 and increment it to 1. We run anotherFunction again, and we increment the backpack value from 1 to 2.

Folks, by returning our function, out from where it was born, returning it out, bringing with it all its live data from where it was born, in a backpack, it is distinct to that function, that return now, individual function, every function we return out of another function, has its own protective live, permanent, persistent store of data, that only it has access to. And this changes so much of how we write and run code in JavaScript.



Lecture: 16 Q&A



Will: I want to do one final thumbs on this, before we spend a bit of time talking about what this principle lets us do as developers. Final thumbs: you lost me, I'm clear, I have clarifications.

Kate's clear, Natalie has a clarification, Jim has one, Elena has one, Joe has one. Joe, why don't you take it away first this time?

Joe: Cool, and this is really small, but I'm just curious. So let's say we didn't have, when we declared 'outer', we didn't have the first line that says 'let counter equal zero'. Then, so it would start looking in local memory, then it would look to the backpack and then does it look to a global memory? Or does it just stop after backpack?

Will: This is a fantastic question. Let's ask that, Joe. So, suppose no counter was... What if this Joe, what if actually counter was declared here such that inside of myNewFunction and anotherFunction we would have in local memory, what Joe?

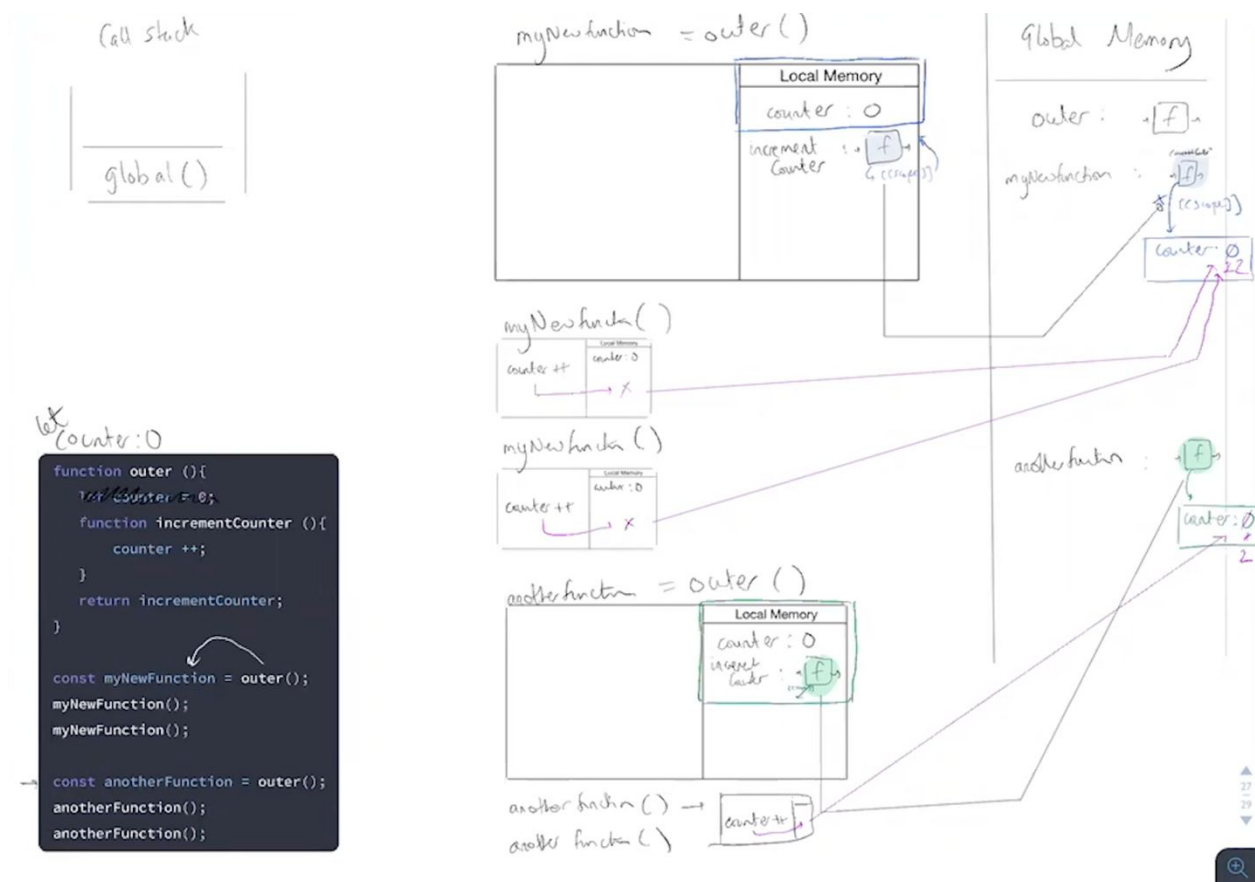
Joe: A counter.

Will: And next time around myNewFunction, counter would be?

Joe: Zero. Yeah. Every time it would be returned it would be one.

Will: Every time we console logged it you would get one, one, one, one. Because our local memory of incrementCounter takes precedence and JavaScript's scoping rules, that is what data is available to me, say once you encounter, once you hit, once you find the data you're looking for you can't keep looking so you find the counter in the local memory, yup, it's one, console log it, it's one every time. Our local memory of incrementCounter, now known as myNewFunction, and its next saving, now known as anotherFunction, the local memory starts from scratch every time. So excellent, Joe.

What if instead of it being in the local memory, what if, as you say, we didn't have counter declared here but counter was declared out in global? What here, Elana, from the four calls to those two different functions would be our console logged counter at the end?



Elana: I think it will be one, two, three, four?

Will: She's spot on. One, two, three, four. We'd always end up finding our way out to the global counter and each time that would be the one we increment. Excellent. Nice question, Joe. Kate was clear. Natalie, you want to ask your clarification here?

Natalie: Yes, so I'm not even sure if it's possible to do this, but let's say that within the function outer, so we declare the function incrementCounter, what if we declared

anotherFunction within outer, would those two functions then have access to the same backpack?

Will: Fantastic question. Yes, but is it very interesting if you're executing them right there and then? Well of course they have access to the data, we know that right? That's standard. If you're executing there and then they'd have access to all the surrounding data. Technically it's because it's accessed through their scope property. Always, even when you run a function in the same place as it was defined you actually go and look on the scope property for what data you have available to you. But it's just like, you know, the data immediately around you so you're not that surprised. We kind of intuitively always thought that, right? So how, Natalie, could I return out two functions at once with a same shared backpack? How can I bundle up functions? Is there some data format that I can bundle up a bunch of functions on, and return out in one go?

Natalie: I mean in... I think you could do that in an object.

Will: Spot on. If you return out an object, you can put as many functions as you want, they all will share that same backpack. Excellent. Elana, what's your question?

Elana: My question is about... Do this... myNewFunction and anotherFunction function references are the same? I mean, they refer to the same memory?

Will: What do you think?

Elana: In heap, I think so, yeah.

Will: Oh, but-

Elana: No?

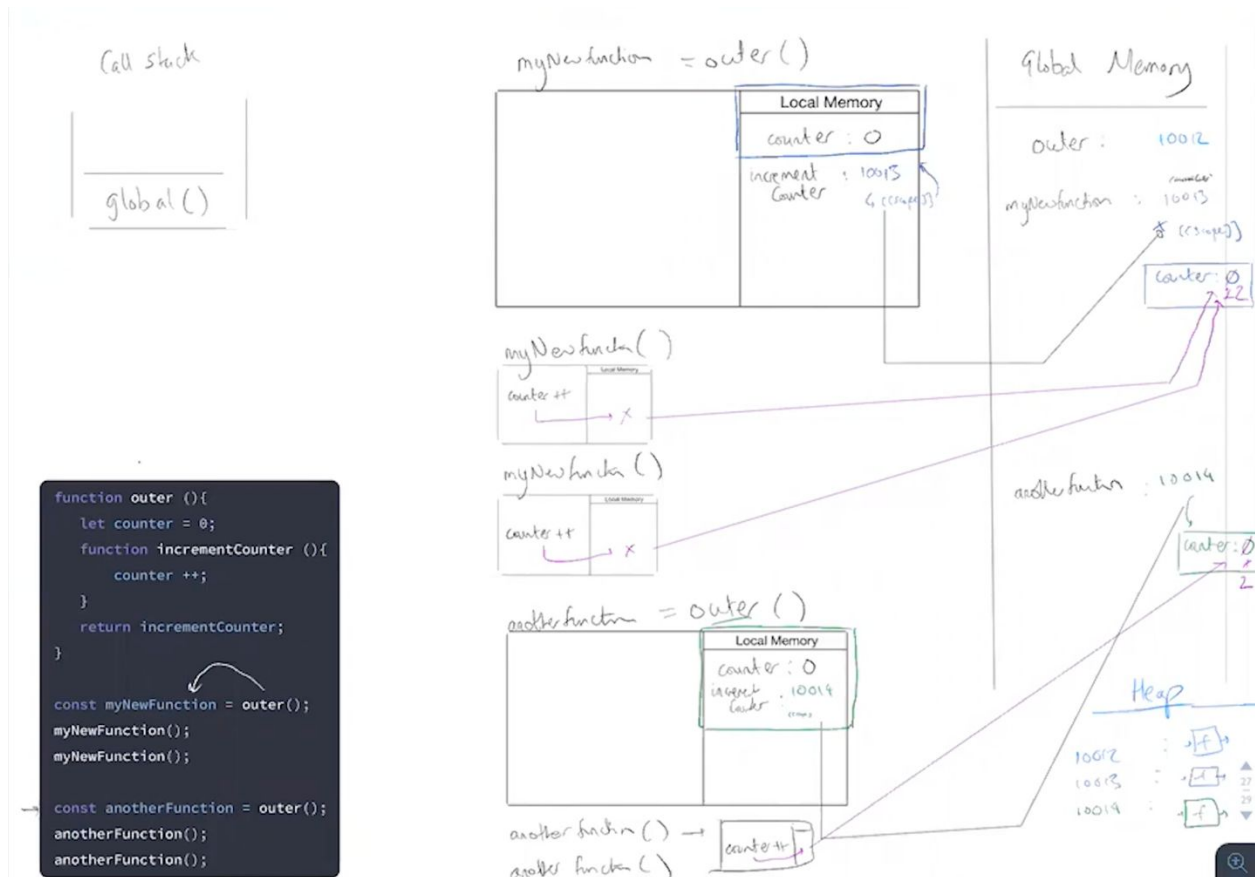
Will: No, because they would not have the same backpack then, right? They'd have the same backpack. They'd have-

Elana: Not the backpack, the same memory in heap. The same address.

Will: Here's a really interesting thing. JavaScript's under the hood optimization engine looks to try and help out with large and complex objects that are shaped in the same way to speed your look-up for them and speed your access to them to try and maybe see if there is any similarities between them in terms of shape. But that's absolutely not the case here. Let's do this now briefly in terms of the heap as a bonus, as a little bonus thing here. So here's the heap, let's walk through this with the heap, very quickly.

So we not longer have the function declared directly, let's get rid of those, and let's instead talk about this in terms of the heap. So we declare outer, gets an address to a function in the heap. There it is. And then we declare myNewFunction, which is a call to outer. Go into outer, we declare counter is zero and we declare incrementCounter is a function, but now it's a reference to what, Elana?

Elana: Like, 10013, I'd say.



Will: Something in the heap, there it is. So we actually return out into global memory and store in `myNewFunction` is just that address. Nice. Then we run `myNewFunction` we're really just running this underlying function here. Ok. Good. Now we declare `anotherFunction`, oh hold on, no. Now we declare `anotherFunction` and that also, maybe I'll do it a different color here, that also is just a damn address. Oh `anotherFunction`, which is a call to `outer`, there we go into it, into `outer` where we declare `counter` is zero. So declare `anotherFunction`, it's a call to `outer`, where is `outer`? There it is. Right, it's the one we stored earlier in the heap by its reference, yep fine. We run `outer`, into we go. `Counter` is zero, we declare a new function `incrementCounter` as a function and it gets... All we actually store is a reference to position in the heap which then gets returned out into `anotherFunction`. Does that make sense, Elana?

Great. Excellent people. Any final questions? Jim, go ahead.

Jim: Yeah, Elana asked basically the question I had in my mind but my question about this is, why when we create `anotherFunction` does it not make a reference to the same place in the heap that `myNewFunction` did? That's one. And two, where does `counter` get stored in the heap, or does it not get stored in the heap?

Will: Counter doesn't get stored in the heap.

Jim: Excuse me?

Will: Counter does not get stored in the heap. Counter gets stored in that more fast access memory position called the stack, actually when we put something on the call stack what we're really describing is a blocked off piece of memory we're going to use for any things, like counter, in the local memory of that function call to be stored in. When I called outer, added to the call stack, outer is... I was actually also referencing, oh, free up some space in really fast access memory to store stuff as I run outer in the local memory and to store links to positions in the heap. So I just now hold onto that when I exit out of outer, rather than deleting it like I usually do.

Jim: So what you're saying is that, the address 10012 doesn't have any reference to counter or... Counter is part of the functionality of outer though, so how does it do that?

Will: Well hold on Jim, let's go back to your previous question. Your previous question was, say your previous question again, do you mind rephrasing it?

Jim: I forgot what it was. My question is... I guess my very first question is why... Because the heap-

Will: Oh yeah your first question-

Jim: As far as functionality at a specific address, so why is creating a new 10014 for functionality that already has an address 10013?

Will: Why is it doing that? Because you are telling it to. You are declaring a function. When you declare a function you are saying store this code in memory. Yes, is it redundant? Yeah, but at the same time it actually isn't because it has its own backpack which is unique to it. You are telling it, Jim, as you run outer another time you are declaring incrementCounter inside you are saying, look declare and save this code. It's doing what you're asking it to.

When you declare a function you're saying go find a unique position in memory to store this code. And if I told you to do it before, fine. And that's why, Jim, in global you won't see me declare the same function twice. But I am declaring the same function here twice on two separate outer calls inside of which I declare the function incrementCounter. I'm just, I'm doing it twice. And so I do definitely tell to call two functions. Yeah. Absolutely. It may look like analogous but it's only looking analogous. I'm very much telling it to store a function once and then call again. Now if you did that in global man, you wouldn't notice, you'd be like, declare outer twice, of course it's got, yeah I'm declaring it twice so it has it stored twice, but it's a bit more funny here because you're doing inside... you're only writing it once inside of outer but when you run outer you're running it twice and so you're storing it twice. You're storing incrementCounter twice. Does that clear that up for you, Jim? Go ahead, man.

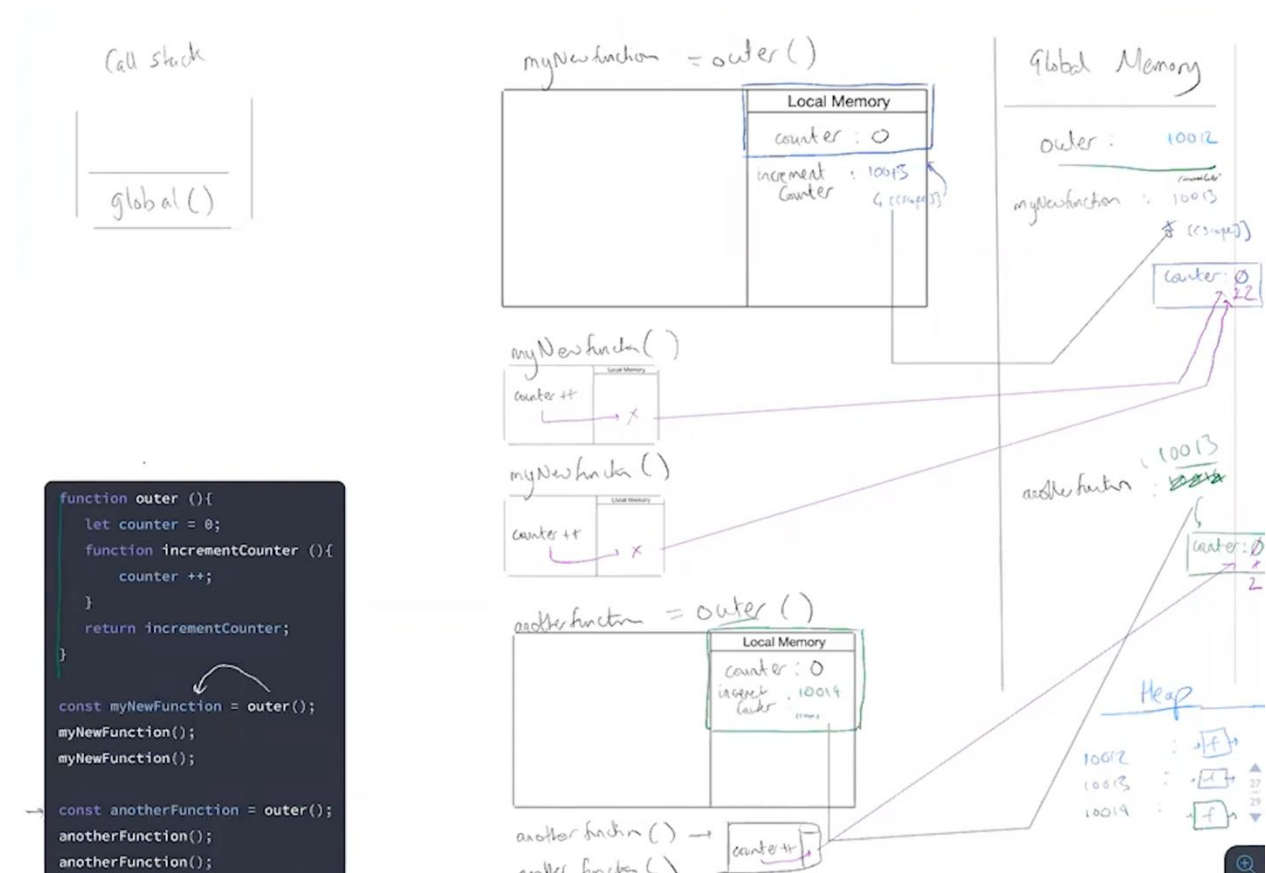
Jim: I'm just going to need to have to think about how this works in practice. I'm still not sure where... What gets stored in the heap and what doesn't? It's-

Will: Functions, objects, and arrays, wherever they're declared, even if inside another function, all you get inside that function is a little link to the underlying stored position in the heap where the function is actually stored. So when I run outer, don't get confused between running and saving a function. So when I save outer initially-

Jim: Okay-

Will: When I save outer initially, there it is saved, I then run it. I go and grab the function to run and into it I go and it saved anotherFunction, which is this one here. Later on when I run outer again, which I then go to outer, grab it, run it, I'm running it again here. Totally separate running of the same function. Inside of it I save incrementCounter all over again and there it is saved. Yeah, Jim, don't worry, the heap piece here can an asset to understanding but also can be a bit of a, you know, okay wow I kind of got it before so don't desperately worry about the heap bit you don't need to be feeling and living it the whole time but if you follow it closely and precisely it does line the dots up for some people.

Jim: Yeah, yeah. I think I just got confused by the whole pass by reference thing. Okay. Great.



Will: Alright, folk. Elana, go ahead, our final question.

Elana: Sorry, yeah. I just realized in my head. So what if declare `const anotherFunction` and set it equal to `myNewFunction`, instead of calling `outer`? Does it mean my `anotherFunction` will have the same backpack?

Will: Yes, yeah, yeah. Because all you're doing there is saying `anotherFunction` is `myNewFunction`, you're saying `anotherFunction` is 10013.

Elana: Okay.

Will: Yeah, absolutely. Alright. Beautiful. Amazing people. Genuinely super, super impressive and thoughtful questions, a level of sophistication there, very much appreciated.

Lecture: 17 Conclusion - Using Closure in Professional Engineering Environment

Closure gives our functions persistent memories

If we understand closure under-the-hood, we get access to an entirely new toolkit for writing quality code

Helper functions

Everyday professional helper functions like 'once' and 'memoize'

Iterators and generators


Which use lexical scoping and closure under-the-hood to achieve the most contemporary patterns for handling data in JavaScript

Module pattern

Preserve state for the life of an application without polluting the global namespace

Asynchronous JavaScript

Callbacks and Promises rely on closure to persist state in an asynchronous environment

22

Will: Alright, here we go. So, let's now talk about what this gives us. Oh, new section! So, let's now talk about what this gives us. What amazing superpowers in JavaScript does this let

us now play out. Well, four big things. One, what I call helper functions. All the time in professional engineering, we want to be able run a function that we then limit to only being allowed to run once. That is to say, we want to once-ify a function, make it only be allowed to run once. Like, imagine I want to say my multiplyBy2 function, you can only run me once. First time you run with me with input seven, return out fourteen. Next time, return out, sorry you've already run me, you can't run me again. In a game like tic-tac-toe, this is super useful.

In the tic-tac-toe game, you run the function when I click on the board, click the cell, O or X. Once I run it and set it to O, I don't want to be able to then run it again and change it to X. I want it to be set from then on. So I limit that function to only run once per set. How can I do that? Well I can have my function store in it's backpack, the first time I run... Well, let's say, multiplyBy2 again but a seven. In the backpack, I can increment the counter to one. Next time I run multiplyBy2 with the input of thirty, check the backpack, oh, counter's already one. And I return out, sorry you can't run me again. Or some people may imagine I return out the first output. So if I ran it with seven before, return out fourteen. I store that in the backpack as well. And when I run it again with thirty it just keeps returning out fourteen from then on. That is a function using its permanent memory. Its backpack. To change how effective it is.

Another one that's super popular is called memoization. Another helper function. Helper functions by the way, excuse me, are just code that you use day to day that help you write code quicker and more easily. Memoization, that means writing a function such that if it's a really slow and complicated function that takes a long time to do certain tasks, if you ever have the function had the same input as it's previously had, do not make it do that really slow task again. Have it remember the previous output from the last time we ran it with that input. So for example, if I did, nPrime, this would be a function that would, if I passed, I don't know, 10,520, it would return out for me, the 10,520th prime number. That's a slow task. That could take, like, a bunch of time. If I ran nPrime again in my app, as it runs live, with the same input, I've already done the work of figuring out the answer. I don't want to do it again, but I'd have to because my functions normally do not remember their previous runnings. So I wouldn't be able to memoize. I wouldn't be able to instantly figure out what the 10,520th prime number was because I forgot; the function does not remember its previous runnings.

Unless... in the backpack of nPrime where we stored it in memory, let's just do our little bit of diagramming. There's our nPrime function that we're running. In its backpack, what if we, in its COVE or its closed over variable environment or its persistent lexical scope reference data or its closure would be, I don't know, let's call it 'store'. Which would be an object with our key being the input, 10,520, and our value being the 10,520th prime number. So the first time we run it, it would be empty, the object would be empty in the backpack, but then we'd do all the slow calculations, estimates, looking at all the smaller prime numbers, getting all the way up to 10,520th prime number and we would store 10,520 as the key and then the 10,520th number as the value and next time we run nPrime with 10,520 as the input, before doing any of the hard work, what do we do? We go and look into the backpack, look in the store, and we would find 10,520 as the input and grab the 10,520th prime number in one step and return it straight out. Super important computer science and certainly programming concept,

memoization. Making our functions remember previous work they've done so we don't have to do it again. And of course, it rests on functions having a backpack. A closure.

Alright. Iterators and generators. So iterators are functions that you have an underlying store of data, an array or an object. Let's say the array is [4, 5, 6, 7, 8, 9, 10]. You have a function that when it runs, the first time it runs, gives you out 4. Next time it runs, it gives you out 5. Next time it runs, it gives you out 6. No hunting off to go grab the data from the array, instead you just run the function, it gives it out to you one by one. Because the data is stored in the function's backpack. And then the little tracking, counter, a variable for how long through the data you are, is also stored there as well. And each time you return out a number from the underlying data in the backpack, also update the counter so that you know to grab out the next element and don't give out the same element twice. You'd move onto the next element in the underlying backpack.

That is an iterator. A generator, really interesting. This is saving a function's execution context midway through it and then coming back to it. Or what is an execution context? Besides a line number that you are in through a function and the memory at that point. So what they actually do is you can exit the function, but all you're really doing is you're keeping the function definition there and then attached to it, you have a line number you're in and then a backpack of data for what was in that function execution at that moment. We're not going to go into generators. There's a lot more to generators to make that fully make sense. But a generator is just a flow of data where you are generating it from some code, but you can pause the flow that's coming out of that code at any one moment and hold onto that function's execution context in that moment and store its data in the backpack of that function. So that when you turn that function back on, you go to its backpack and it's the local memory ready to continue the function running. That's all that the execution context is, is a backpack of data and what line you were in the function.

And ultimately, module pattern. The module pattern and many other design patterns exist to give your code organizational structure. To make sure you're not going to have your team members overwrite your data. So if you've got a global variable-like result in your team with a hundred developers and 500,000 lines of code and you've got a global variable result in your app. Everyone in your team wants to use the global variable result. You can't really use it. But the only way to keep your data alive for the whole of the application, is to store it, where? Kate, if I want to have my application hold onto data for the whole of its lifespan, I can't put it inside a function because when the function exits, it gets deleted. So where is the default place I'd put data I want to hold onto the whole lifespan of the application running, Kate?

Kate: In the backpack, correct?

Will: Right, but let's hold on backpacks for a second. If I want to just normally hold onto data the whole life of my application, where would I store it?

Kate: Oh, you're talking about global memory?

Will: Global memory, yes. It's the only place I know would not go away is global memory. If that's gone, I've left my whole application. But if I put result is '7' in global memory, the other 30 engineers on my team, my colleague, she wants to put result is '20'. She's overwritten mine. That's why we have const by the way, so the overwrites can't happen. Or my other colleague, they want to put result is '50', for their bit of the code. So I've got this conundrum. I want to hold onto data the whole of my application, but I can't put in global memory. What do I do? Exactly as Kate says, I put it in the backpack. I put it in function's backpack. I write functions whose input updates the backpack and whose output is the backpack's contents. This is effectively the module pattern in JavaScript. It's to protect while you're in global - no - it's that when you run your function, you have access to data that is not available and touchable globally, but does persist. Does last the entire life of the application. You can't lose the data. You need the data to stick around, otherwise, you need pseudo global variables. While inside that function, they are permanent. Like globals, because they stick around, but when you exit that function, you can't access them. Only by running the function you get access to them. Really, really cool.

And finally, asynchronous JavaScript. So in asynchronous JavaScript, you're regularly, and is worth watching asynchronous Hard Parts, the one on right now, so thank you all of you who are missing that one. You regularly pass functions to asynchronous requests, they call them, or asynchronous tasks, where the function you pass will be triggered to run automatically when the slow tasks, like speaking to the internet, to get a tweet back for example, so it's an asynchronous task, when that completes. When that completes, you want to have JavaScript automatically trigger a function that will display the tweet that comes back. That's a problem because that function may be referencing stuff when it ends up being eventually run in global way later after it was defined in some sub-function earlier in your code. It may end up being run in global or will end up being run in global and theoretically all the data around it is gone. No, any data it references will be attached to it in a backpack. Super, super advanced pieces there. If you see an asynchronous code a lot, you know about this. If not, don't worry. But if you are having a function be auto-run for you later on by JavaScript, you better hope all the data was there when you wrote that function is still there, but of course it is. It's attached from its scope property; it's attached on that function's backpack. Never fear, the data's still there.

Alright folk, that's it. Closure, the most powerful concept in JavaScript. Amazing, amazing questions from all of you. Amazing talking through the code by all of you.

Closure gives our functions persistent memories

If we understand closure under-the-hood, we get access to an entirely new toolkit for writing quality code

Helper functions

Everyday professional helper functions like 'once' and 'memoize'

$\text{memoize}(10, 520) \rightarrow$
 $\text{memoize}(10, 520)$

Iterators and generators

Which use lexical scoping and closure under-the-hood to achieve the most contemporary patterns for handling data in JavaScript



Module pattern

Preserve state for the life of an application without polluting the global namespace

Asynchronous JavaScript

Callbacks and Promises rely on closure to persist state in an asynchronous environment