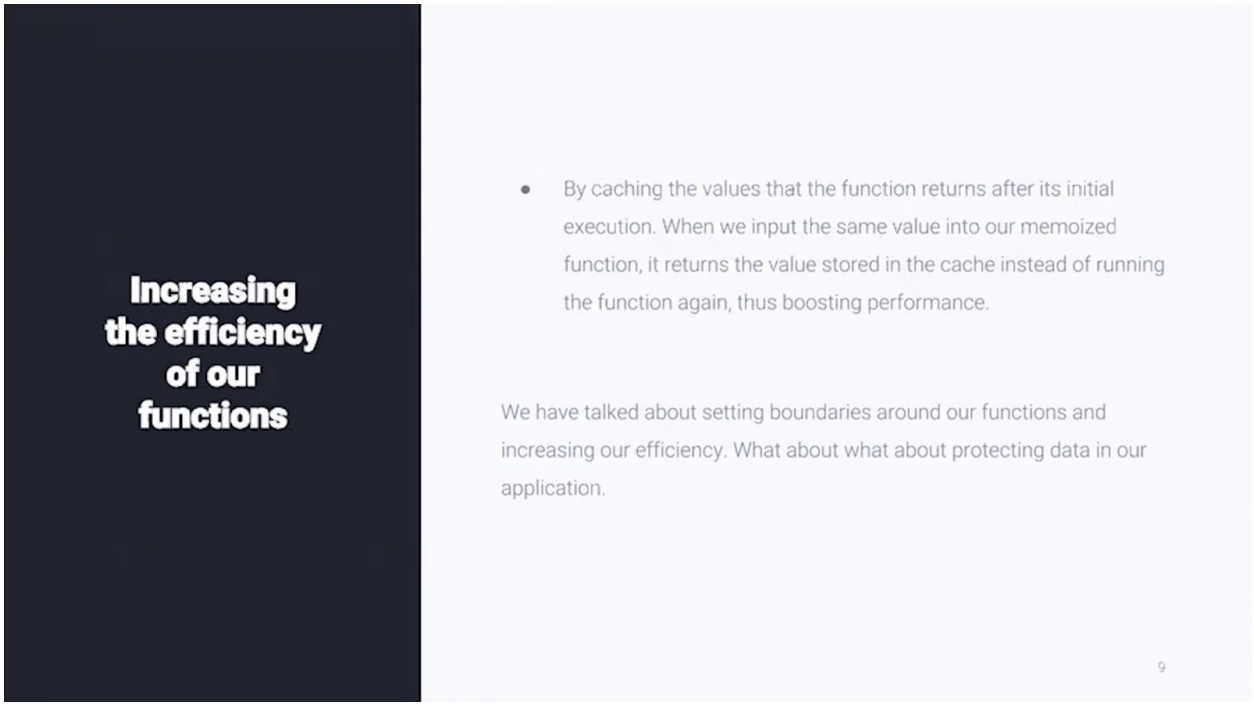
Lecture: 7 Using Closure in the Module Pattern



Phillip: Alright. Let's move on to our final piece. So we've talked about, again, we've talked about how to set boundaries around how many times we can call a function. We've set up logic that we can optimize each subsequent call to our function and we've generalized both of those strategies so that we can pass in a function and have it returned out the memoized version or the onceified version of that function. So we're looking really, really good right now as far as closures goes for our actual implementations into our applications. Ok.

So let's kind of switch gears a little bit here and let's talk about something more broad or a very common maybe bug or issue, that developers might see themselves running into as applications begin to scale. And very specifically when you have multiple engineers, 10, 20, 30 engineers, working on a code base at the same time.

And this isn't where I want to talk about global memory. So right now if we had to think about, like let's say that we had a variable of just name. Let's say we had a variable name and we wanted to save a value into that variable. Let's say we saved Ariel. We saved Ariel into the variable name. And we want it to persist that data for the entire life cycle of our application. We wanted to make sure that data was always available to us, that name data was always available to us for any point in time in our application.

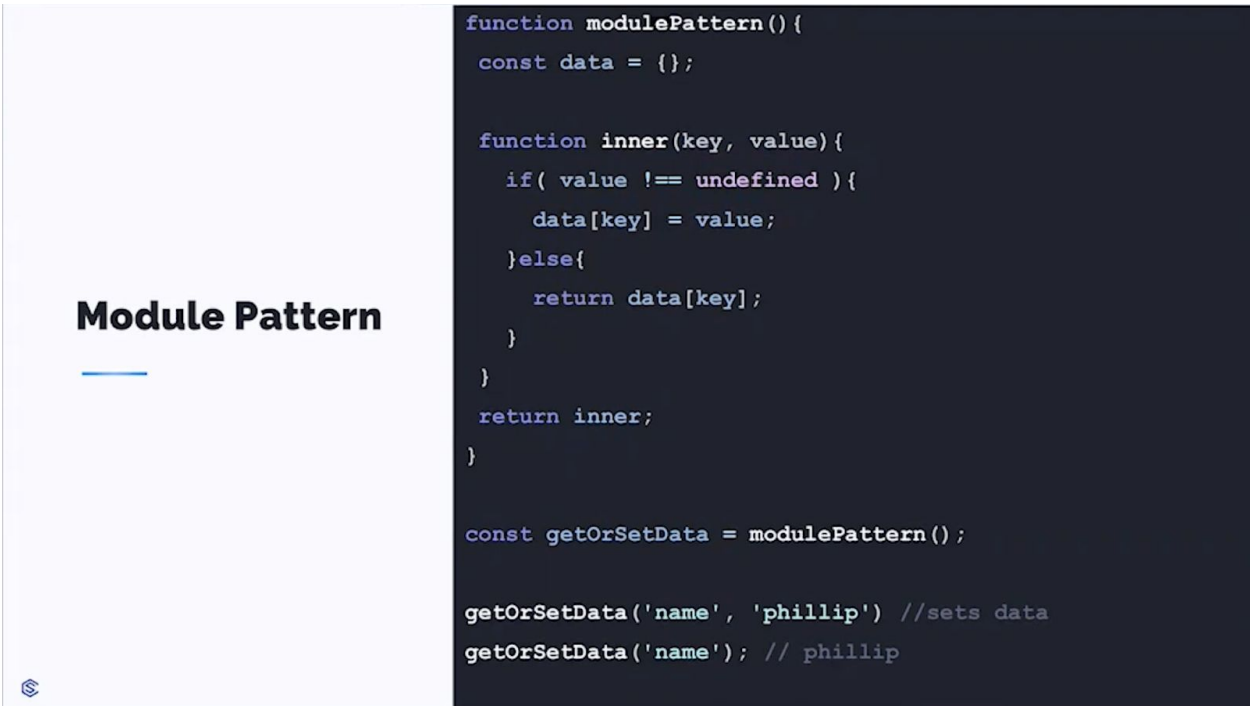
Well, the only logical way or what most people immediately think about doing is to make sure that you have that data stored somewhere for our entire application, is that you would store it in global memory because that's the only other time that you can be

for sure that, that execution context wouldn't be deleted or deallocated from memory. It would have to be in the global namespace, in the global execution context.

But the problem with that comes when you're working with 20, 30 other engineers on the same code base with a very common variable name such as name or results or value or something very common like that. Everybody wants to use that variable name. So you run into these issues sometimes where you have two people, two developers at the same code base, they're essentially fighting over the same label for that space in memory.

So you write Ariel, you write a variable name and give it the, initialize it to the value of Ariel, and then Donald, you come in a day later and you write and go, "Hey, I want to use the variable name," not knowing that it's already been used somewhere else in the code base. So you define a variable name and you initialize it to the value of Donald.

Well now Ariel comes in to check back in on her code and everything's busted because now the value of name that she originally thought it was no longer that value. This is an issue in large scaled production level code for sure. So JavaScript or ECMAScript engineers came out with a design pattern called the module pattern. And it's a way of using the same pieces that we've already learned about right now using the closed over variable environment, the backpack, returning a function from a function. They used this same concept to create sort of protected data, just like protected pockets of data. That can still be stored in global memory in a sense, but it's protected. It's better protected than just having a globally exposed variable. You have to know how to get at that information, how to get at that name variable and that value, a very specific way so that it can't just be very easily overwritten by accident by a different developer. So, this is referred to as the module pattern and the module pattern is very, very complex. There's tons and tons of things that go into it, tons and tons of functionality. We're going to be focusing in on, kind of the most basic holistic implementation of it, which is how to set data and retrieve data from within these protected pockets that we're going to be creating, aka our closed over variable environments.



So again, this is not implementing any new pieces of closure or our mental model of JavaScript. This is just setting up logic that uses those different pieces in a more unique and different way than what we've looked at so far. So this is our module pattern. This is our basic implementation of it. And you'll see a lot of similarities between this and some things like memoize function. So let's dive in a little bit deeper. Let's walk through this code line by line and let's see how this whole process works out. And we'll whiteboard it so we'd have a good visual representation of it and see where those real benefits come in.

So I'm going to take all of this code that we have right here. I'm going to copy it, and I'm going to drop it down here so we can whiteboards through it. There we go. Let's make it a little bit smaller so we have some room to work here. Is that ok as far as size goes? Good? Alright, cool. Alright. Donald, why don't you walk me through it man? What's the first thing that we're doing inside this bit of code?

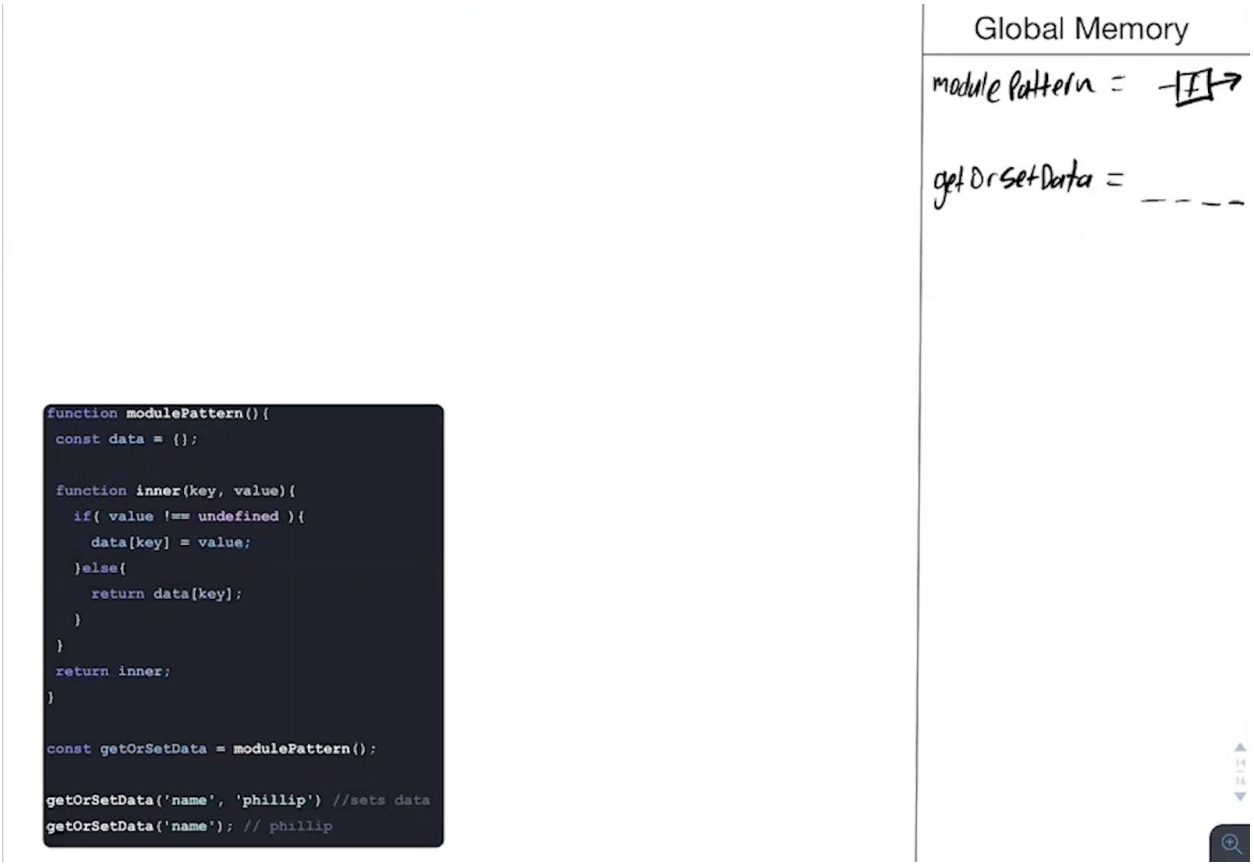
Donald: Sure. We're declaring a function with the label modulePattern in global memory.

Phillip: Perfect. modulePattern. It is this function definition right there. Good. Keep on going with this Donald, what's the next piece that we're going to run into?

Donald: Sure. We're going to declare a constant with the label getOrSetData and-

Phillip: getOrSetData, yup.

Donald: Yup. And we're assigning it the return value of the invocation of modulePattern.



Phillip: Absolutely. So at this point right now we don't know exactly what it is, but we know we're going to have to go run some work or do something in order to get that value. So it's uninitialized for the time being. So let's go ahead and run that work. So the code that we're running right now is getOrSetData is equal- Ariel, do you say data or data?

Ariel: I say data.

Phillip: You say data? I say data as well. Donald you're weird man, you say data.

Donald: Data, data.

Phillip: That says a lot about you. I don't know what it says, but-

Donald: Tomato, tomahto.

Phillip: If you say tomahto, you're a weird guy.

Donald: No, no. I say tomato.

Phillip: You're a weird person if you say tomahto. Do you say tomahto, Donald?

Donald: No, I don't.

Phillip: Ok good. Because if you say data and you also tomahto... Ok, we're running. getOrSetData and we're setting it equal to the evaluator result of running our modulePattern function. There it is right there. Alright. So we're running a new function, which means we're going to create a new execution context. So here we go. We have our execution context, right? Let's see. There we go. Alright, so we have our execution context. What's the first thing that we're going to do inside this call to module pattern? Ariel, walk me through it.

Ariel: First, you're going to declare a constant called data and it's going to be assigned to an empty object.

Phillip: Perfect. There we have the object. Oh, that was not as good of curly braces as I wrote before. You can see the drawing abilities slightly declining as the day goes on. So we have our data variable. What's the next thing that we're going to do, Ariel?

Ariel: You're going to store the function definition of inner in local memory.

Phillip: Perfect. And let's give that a color itself. Let's give it a blue color. There we go. And then Ariel also let me know what is the JavaScript automatically do at this point as well, since we're creating a function?

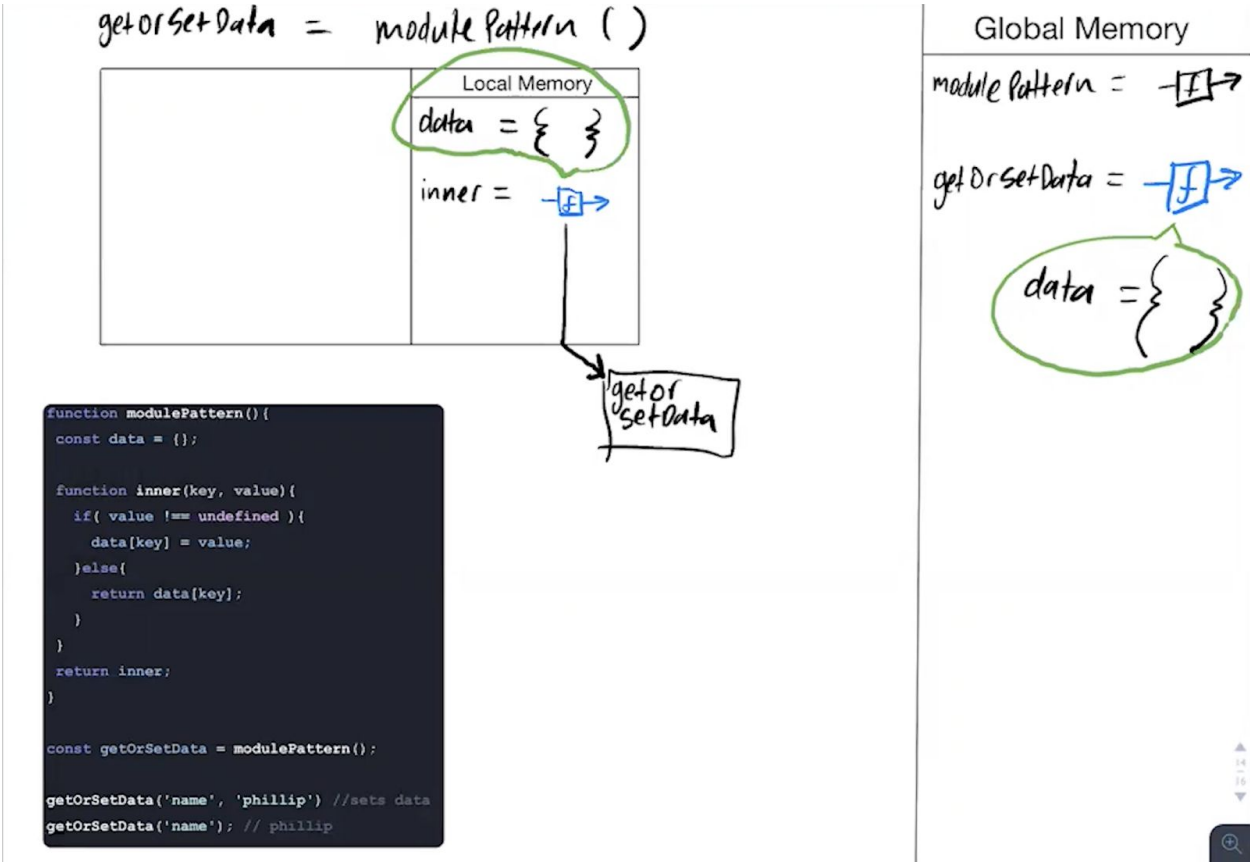
Ariel: It opens its own execution context. Oh, I'm sorry. It creates the closed over variable environment.

Phillip: Perfect. I think Donald's been the only one that I- He's the only one I've actually asked this every single time we've gotten to it. So I caught you a little bit off guard there, but yeah, we create our closed over variable environment, which right now is simply encompassing our data objects. There we go. We have it. What's the next thing that's going to happen on this call Ariel?

Ariel: Then you're just going to return that function definition.

Phillip: Yeah. We're going to return that function definition and store it under what new name?

Ariel: getOrSetData.



Phillip: Yes, getOrSetData. Perfect. There it is, so now, inside of our global variable environment, getOrSetData is no longer uninitialized. It is now this blue function, but it was originally our function definition associated with inner. But in addition to that, we also have our closed over variable environment or our backpack. And inside that we have our data variable- No that wasn't very good, was it? Ariel is like, "No, that was a terrible curly bracket." Alright, so now we have our data variable that's initialized to an empty object. Perfect. What's the next thing now- Well now, I mean, I guess now we want to experiment with it, right? We have our getOrSetData that has a backpack associated with it in global memory. So now let's start kind of experimenting with, ok, so how do we now add data to this backpack and retrieve data from this backpack?

And we'll kind of see how now the data that is stored inside this backpack is kind of this protected data from accidental overrides to it in a way. So let's continue on with our code here and let's experiment with a couple of different variations of a call to getOrSetData and see how it works out. So Donald, what's the next bit of code that we're actually going to be running here?

Donald: Cool. We're going to be running what was formerly known as inner.

Phillip: Perfect. And we're going to pass it a couple of arguments, right?

Donald: Correct. We're going to pass it-

Phillip: What was formerly known as inner, which is now getOrSetData.

Donald: Yup.

Phillip: And then we're going to pass in some arguments. So let's move this down a little bit so we have some room to work here. Alright, so we're running the code, just so everybody can see. So we're running this code right here, getOrSetData, passing in the name argument and also passing in the Phillip argument. Alright, so we're going- Oh, I'm sorry. Here, let's write out our code. We're running the code, getOrSetData and we are passing it the arguments name and Philip. There we go.

So this means that we're creating a new execution context since we're running a new function. Let's grab our execution context. There it is. Alright, so just like you said Donald, the functionality that we're actually running here at this point is our original functionality that was associated with our variable or our label inner, but now we're running it under its new name, global execution context. Excuse me. So what's the first thing that we're going to do in this bit of code now, Donald?

Donald: Yup. We're going to set the parameters in local memory. So we going to have key and that's going to be name and the value label would be Philip.

Phillip: Perfect. There you go. Key is name, value is Phillip. And now if we kind of look at our actual code for inner, we're going to be running some conditionals as well. And let's walk through our logic. Ok. So, on this first conditional that we have here, JavaScript hits it and it says, "Ok, we have an if statement. So we're going to run and evaluate this condition to see if it's true or false."

So the condition that we're running or that we're going to evaluate is value not equal to undefined, ok? So break it down here. First thing JavaScript does, it says, "Value. What the hell is value?" Where's the first place it looks Donald?

Donald: Local memory.

Phillip: Looks in local memory. Does it find that a variable called value?

Donald: Yes, it does.

Phillip: Yes it does. And what is the value of value?

Donald: Phillip.

Phillip: Phillip. Ok. So it has value and the value of value is Phillip. So with this condition that we're evaluating right now, it says value not equal to undefined. Would that evaluate to true or false?

Donald: True.

Phillip: It would evaluate to true. Exactly because the only reason that it would evaluate to false is if the value was undefined. Ok. So the next bit of code that we're going to run, we know is going to be associated with the if code block. So the code that we're running says data key. So let's break that down even smaller. So data, first place it looks local memory. Does it find data there, Donald?

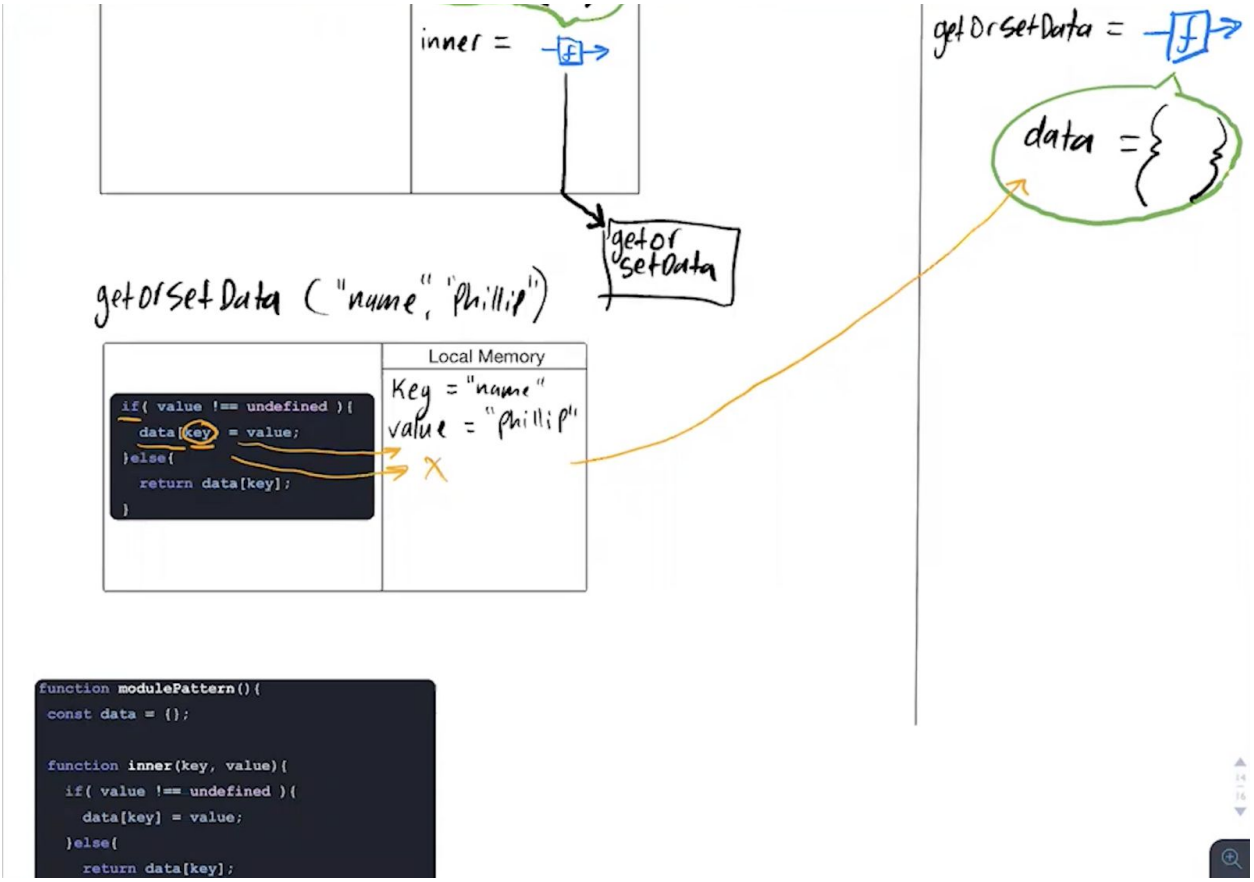
Donald: No, it does not.

Phillip: Where does it look next for data?

Donald: In the closed over variable environment.

Phillip: Yeah. Looks right there in the closed over variable environment. And then does it find data or data if you will?

Donald: Yes, it finds data.



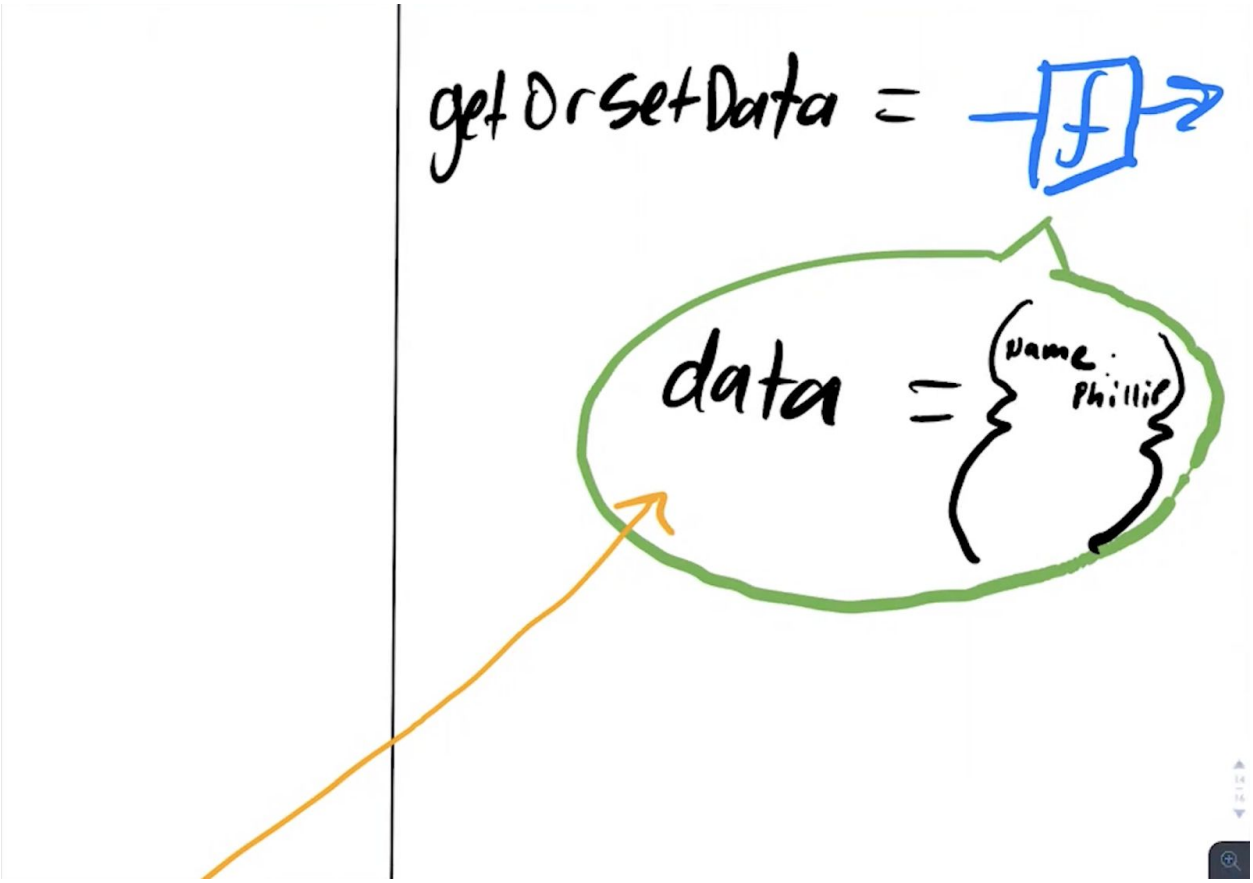
Phillip: Yeah, it does find data. Data, whatever you want to call it. So finds data. Alright, finds it there, great. And it says, "Is there a key on this data objects, that is the value of

whatever key is at this moment?" So right now, if we were to evaluate what key is, we would look at local memory. What is the value of key Donald?

Donald: It would be name.

Phillip: It would be name. So is there a key of name on the data objects?

Donald: No, there is not.



Phillip: No, there is not. But JavaScript knows that if it doesn't find that it's ok, it's actually going to create one. So it creates name. So zoom in here so we can see that. It creates. I tried to rewrite that bigger and it came out the exact same size. But it is a key that says name and we are going to store the value or we're going to initialize its value as being whatever the value is of value. So what is the value right now of value, Donald?

Donald: It would be Phillip.

Phillip: It would be Phillip. So inside of our data objects, we would have name and its value. I'm going to drop it down a little bit here just so we can fill it or fit it, Phillip. There we go. Ok. Zoom back out there and that's it. That's the only thing that happened when we ran that function because now we're done with our code. Ok. I mean not super exciting I suppose. We just gave it some value, gave it some data and it's stored it in our

backpack. We're going to find out this is actually very, very powerful. Let's run the next bit of code and see what happens when we run the same function, but in a slightly different manner.

See what it gets us, and then we'll kind of sum all this up as to why this is so important and why this is so powerful. So the next piece of code that we are not going to hit, because now we're done running our getOrSetData, function invocation, passing in name Phillip. So now, we're going to run the function. Now we're done with that. Now we're running getOrSetData. But this time we're only passing in one argument of name. That's it. So let's run that bit of code and see all this plays out. Let's drop this down a little bit here. Get rid of that.

The code that we're going to run now is getOrSetData and we're going to pass it the argument of just the string name. Alright, so remind me Ariel, when we call a new function, what is it that we create?

Ariel: A new execution context.

Phillip: A new execution context, there is. Alright, and walk me through Ariel, what's the first thing that we're going to do inside this call to getOrSetData name? I'll scroll down so you can see it.

Ariel: First, you're going to pair the parameters and arguments.

Phillip: Ok. So first thing that we do is we- Ok, so what's our first parameter that we have?

Ariel: It's key.

Phillip: Key. And that is set to the value of?

Ariel: Name.

Phillip: Name. Exactly. Ok. And now we have a value parameter. So we have to create our value parameter. So we have value. But now, what is the argument that's being passed in for that parameter, Ariel?

Ariel: There isn't one.

Phillip: There isn't one. But JavaScript can't just have this label that points to nothing in memory. There has to be something associated with that label in memory. You can't just have an empty space in memory. So JavaScript defaults to giving it the value of undefined. So value is at this point undefined. We didn't pass it anything. Alright, cool. So the next bit of code that we're going to hit now is our conditionals again. So let's grab those, copy that, let's paste that in here.

Alright, there we go. We have it right there. So next bit of code that JavaScript is going to hit is it's going to hit our if conditional and says, "Alright, we're going to run some sort

of conditional statement." And the first piece that it has to evaluate is value. So right now, where is it going to look for the variable or the label value, Ariel?

Ariel: In local memory.

Phillip: Local memory. Does it find it?

Ariel: Yes.

Phillip: Yes it does. Cool. So then it says, "Ok, I found the variable value." Now it's going to check the value of that variable. It's going to say if it's not equal to undefined, this will evaluate to true. But right now, what is the value of value, Ariel?

Ariel: Undefined.

Phillip: Undefined. So this conditional right here is going to evaluate to true or false?

Ariel: To true.

Phillip: Yeah, it's going to evaluate you to- No, I'm sorry. See I got a little mixed up there. It says, remember, as long as the value is not equal to undefined. So if it is equal to undefined, it's going to evaluate to false. It's kind of a little bit of a- It's because it's like a double negative that you have to kind of think about there. But our value is undefined so this condition will evaluate to false. So we won't actually run the code associated with the if statement here, we won't run that.

We're actually going to end up running the else block, which is return data at key. Alright, so return JavaScript knows what that is. Next thing it has to look for is, what is data? Or data. I feel like JavaScript would say data. I feel like me and JavaScript would be very similar in that sense. JavaScript would say data. So we're going to say data. JavaScript first looks inside local memory and, Ariel, does it find data in local memory?

Ariel: No.

Phillip: No it doesn't. So where's the next place it looks, Ariel?

Ariel: In its backpack.

Phillip: In its backpack. So it looks up here in the backpack where it does find the data objects and then it says, "Ok, is there a key on that object that is whatever the value of key is?" Well right now, what is the value of key, Ariel?

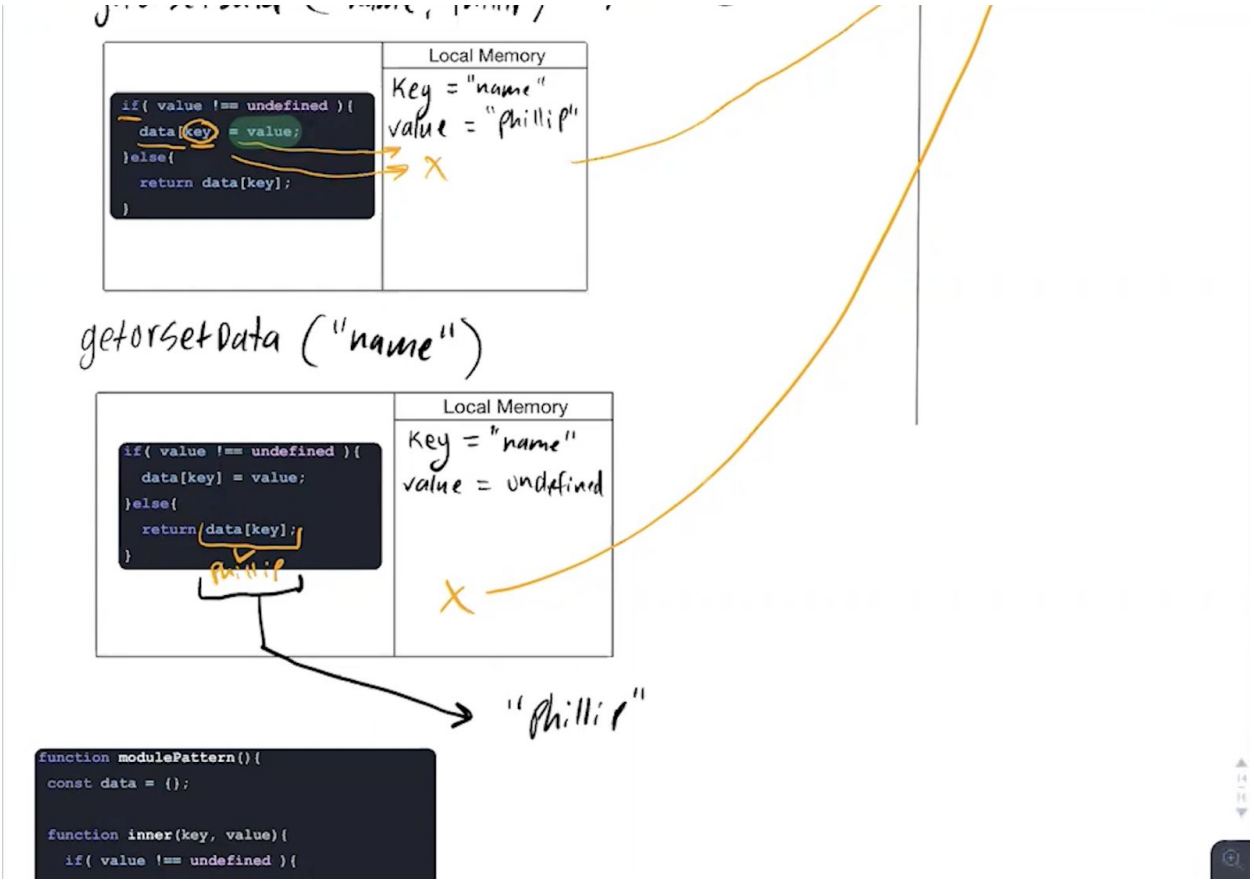
Ariel: It's name.

Phillip: It's name. So does our data object in our backpack have a key name?

Ariel: Yes.

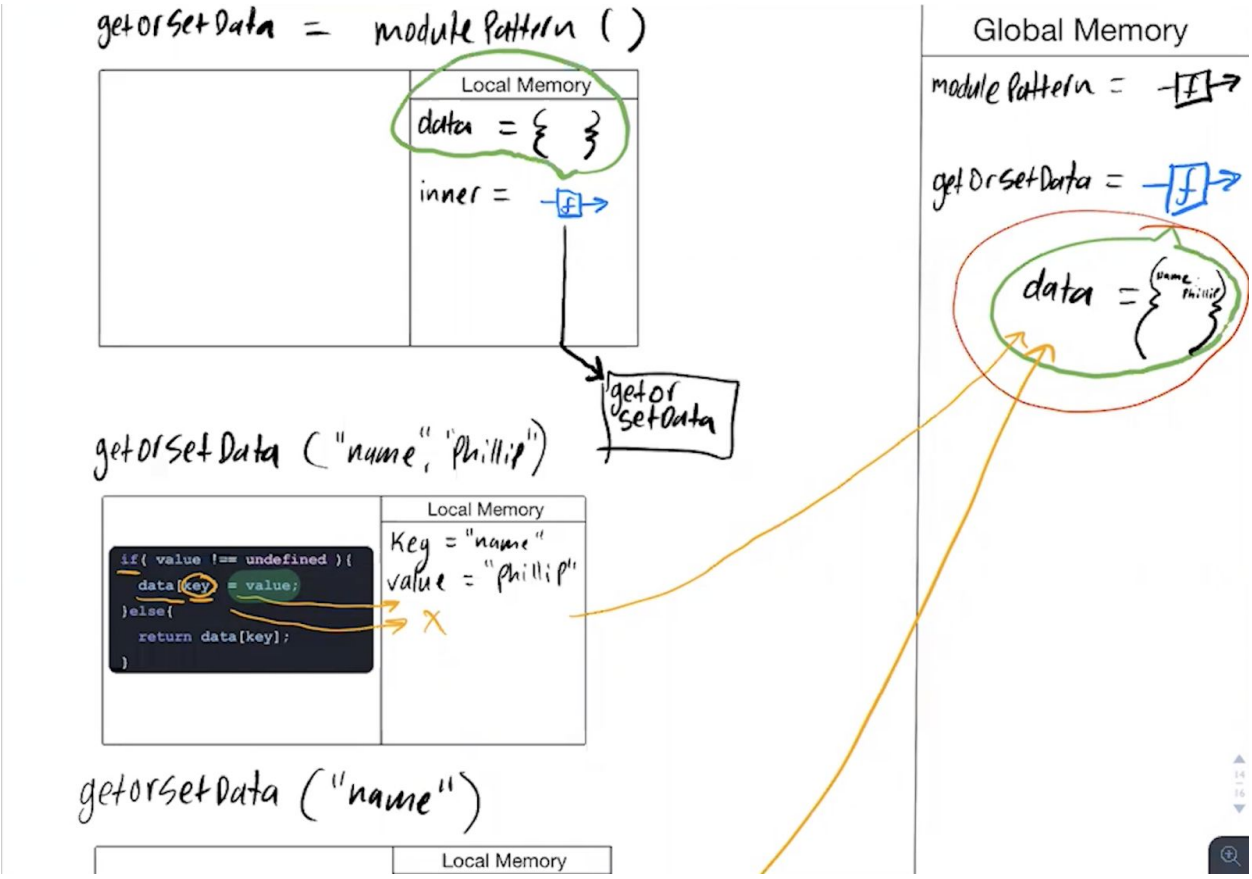
Phillip: Yes it does. And what is the value of that name?

Ariel: Phillip.



Phillip: Phillip. So, this entire piece of code right here, data at key evaluates to Phillip. We take that Phillip value and we return it out into its global execution context where we then have that data. And we can do whatever you want with it. We can use it in some sort of other function or we can console.log it or whatever our heart's desire is for the application that we're currently building out. Now, this seems like kind of an uneventful function that we just created.

We essentially just created a function that when we call it the first time and we set some data and we called it a second time, we retrieve that data. But think about this for a second. In our current mental model of how our application is set up with its data, there is no way of accessing any of this data, it's stored inside of our backpack with the exception of running getOrSetData in a very specific way.



This is what allows us to store data that's accessible from the global execution context but in a protected sentence. It is literally impossible for another developer right now to come in and overwrite that name variable by accident. Somebody would literally need to know that the function they have to call in order to reset that value is to call getOrSetData, pass in specifically name as our first argument, and Phillip as our second argument.

Otherwise, it would just either add a new property to the data objects or they just wouldn't be able to access it and they wouldn't be able to access it. Access, access. See I'm talking about Donald's pronunciation of data and data and I can't say access; this is terrible. The only way that we will be able to retrieve that data specifically is to call that same function- So the two kind of terms that we use for this function is we use either a getter or a setter function.

Now in this implementation we're creating one function that does both just depending on how we call it. But in other implementations you may have two completely separate functions. One function that's called get data and one function that's called set data. But they're very, very specific on how you access those pieces of data. It makes it very, very difficult to overwrite that variable or overwrite that name property on our data objects by mistake. Which is what makes it kind of bulletproof when it comes to working on teams with large groups of engineers. So this is our way- These are some of the more

advanced concepts or advanced features or implementations of using all the same things.

Think about it. Every single one. We just went in four implementations of more advanced features of closures, all of which use the exact same patterns. Create a function inside of a function, return that function out, save it under a new label, and then access the adjacent data through that function from where it was defined. All the same stuff. Every single implementation. We just set up the logic inside that function in slightly different ways to give different rule sets and different parameters, different boundaries on what happens with those functions from function call to function call.

This is very, very powerful stuff in JavaScript. Closures is by far one of my favorite concepts in all of JavaScript. One, because of just the innate ability to persist data from function call to function call. But two, because of just the numerous, numerous other pieces of functionality that we get from JavaScript, that all rests on the concept of closure. Like onceifying functions, like memoizing functions, like the module pattern. All of these things are based off of one specific concept in JavaScript, which is closures.

So let's have thumbs, final thumbs on all of these concepts so far before we wrap for the day. [Thumbs up] I'm clear; I'm ready to go off and start implementing these things. I'm ready to go off and start building my tic tac toe game with my onceified functions. [Thumbs sideways] I have a couple of questions, which is totally understandable at this point. [Thumbs down] I have no idea what's going on. I'm going to watch the more in depth lecture again on closures and I'm going to come to JavaScript the Hard Parts again. Let's have thumbs on these concepts. Ariel's clear! Donald's clear! Wonderful, wonderful, wonderful. Guys, it's been an absolute pleasure. Thank you very much and I'll catch you next time.