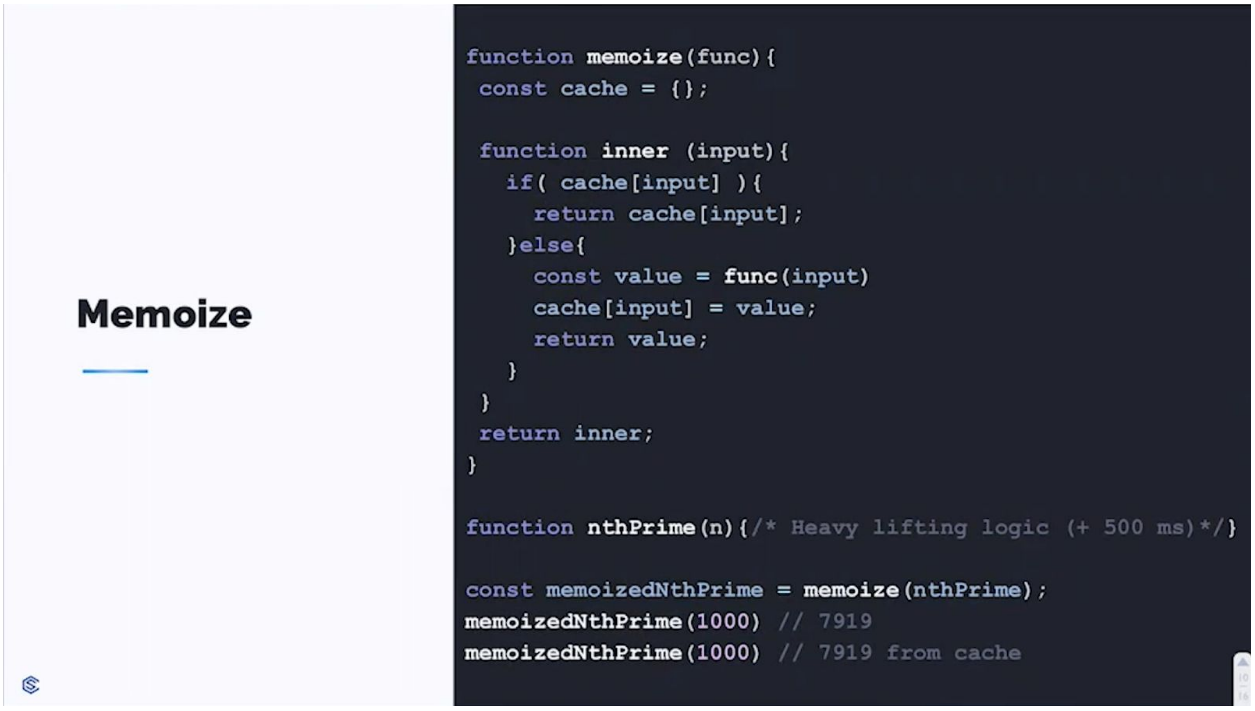
Lecture: 5 Using Closure to Create a Memoize Function



Memoize

```
function memoize(func){
 const cache = {};

 function inner (input){
   if( cache[input] ){
     return cache[input];
   }else{
     const value = func(input)
     cache[input] = value;
     return value;
   }
 }
 return inner;
}

function nthPrime(n){/* Heavy lifting logic (+ 500 ms)*/}

const memoizedNthPrime = memoize(nthPrime);
memoizedNthPrime(1000) // 7919
memoizedNthPrime(1000) // 7919 from cache
```

Phillip: Let's look into something a little bit more interesting. Something that may be considered to be little bit more powerful when using closures. And that is this idea of using a memoize function or memoizing as it's usually referred to. Memoizing a function. Similar to how we had onceifying a function, we have memoizing a function.

So, let's say - totally different application idea now - let's just say, Ariel, you're building out an application that has a bunch of numbers on the screen. And, you built out this application that's designed so that when a user clicks on a number, it displays to them in the dom, on the page, in the browser, it displays to them that primeth number. So, if they click on "100", it will display to them the 100th prime number or "1,000" it will display to them the 1,000th prime number. So, you're going to need to create a function that's going to run every time that they click on a specific button and when that function fires, it goes through its process and it does all of its computational steps to figure out what the 1,000th prime number is or the 100th prime number is, whatever the case may be.

Well, that function is not a simple thing. That function is going to have quite a few steps in it to figure out what the 1,000th prime number is. Let's just say for sake of ease and sake of argument that this function - we'll call it nthPrime, so it gives you the nthPrime number. nthPrime, when you run it with an input of 1,000, let's say that it takes 500 milliseconds, so a half second. And you may think, eh, it's no big deal it's a half a second. This is an eternity in JavaScript. 500 milliseconds for a function to run? That's a long time in JavaScript because you also have to remember that there's tons of other stuff that's also happening in the browser at the same time. But 500 milliseconds. Very, very long time for a JavaScript function.

So how can we make this more efficient? Well, using closure, we can use this process of memoizing that nthPrime function, so that we create a persistent cache, we persist memory. For every single time the function is called, it saves not only what the input number is, so 100, 1,000, 1,000,000, whatever the case may be, not just the input number, the argument, but also what is the value that the function returned out after it ran all of its computational steps? So again, taking back to 1,000 as our base example here. Input 1,000 and the first time you run it, it takes 500 milliseconds. There's nothing we can really do about that. We just kind of have to bite the bullet on that one, we have to run the functionality and have a little bit of a less efficient function for the first run.

But then we save the fact that the function was called with an argument of 1,000 and what the return value was, which is 7,919. I know that off the top of my head because I'm very, very smart. So 7,919. That's the return value. We save all of that in the cache. So that next time someone logs into your application, Ariel, and clicks the button that says 1,000, that function doesn't take 500 milliseconds to run, rather it looks in its cache, it goes, "Hey, has this function been called before with this input?" And if it has, let's just return that value that we returned before. There's no point in doing all the computational steps again, if we can just return that value that we remember that we gave out before. Now this takes a function that was running at 500 milliseconds every time we ran it, to 1 millisecond. One step. Looks in the cache, it grabs the value, it sends it out. Done. Very, very quick. Super, super powerful functionality to have in our application.

Ariel, does that intrigue you more? Does that sound something like more what you would do in an application?

Ariel: Yes.

Phillip: Yeah, absolutely. Let's go ahead and explore how this might look. So we have the code actually for us right here on the page. We have our memoize function, and we're not even going to go through the process of- we're going to start off on the professionalized memoize function, where we pass in the function and it returns out a memoized version of it. We're not going to lead into it like we did with once.

So let's go ahead and just- let's just jump into this right now. So I have all that code, it's pasted down here. It's a little bit smaller. Again, can you guys see it? Is it ok? I can zoom in when necessary if you guys can't see it. So, we have our function right here, we have our code. Let's start diving into this and see what's happening in this every single step of the way.

Carl, why don't you jump us off here, man. What's the first thing that's happening in this call, or in this code snippet?

Carl: You're declaring a function, memoize, and it will take in a parameter, func.

Phillip: Alright, so we have our function definition of memoize, and it's stored where, Carl?

Carl: In global memory.

Phillip: In global memory, fantastic. What's the next line of code that we're going to hit, Carl? This is another one of those ones you really have to look the spacing and all that because it's a little bit smashed together, but tell me what's the next piece that we're going to hit?

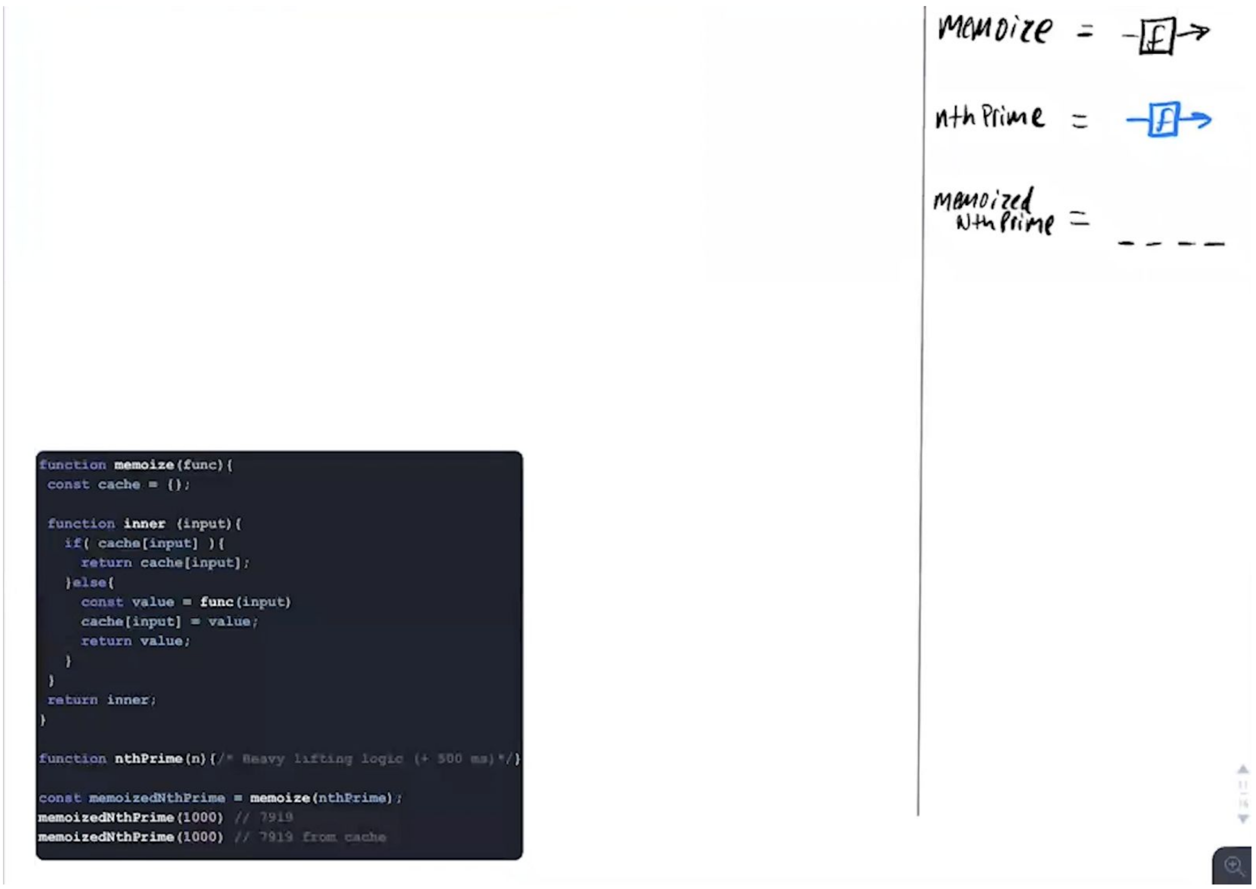
Carl: The next line is you're going to declare another function called nthPrime, with the parameter of n.

Phillip: Good. And we're going to make this function a different color because we're going to be passing it around. We're just going to make our nthPrime function blue. There it is right there. And now you may be looking ahead a little bit right now and looking at the function definition of nthPrime, and you'll notice that there isn't any actual code in there. Because I didn't want to whiteboard-out all of the functionality of nthPrime, so we're just going to say it's kind of a black-box function right now. Some stuff is happening in there, but we're not going actually whiteboard all the way through it.

But we know the purpose of nthPrime is that we give it an argument, we give it 1,000 and it returns to us the 1,000th prime number. Which again, is 7,919. So I just want to flex that knowledge on you guys again. I really hope that's true; I say that all the time and I'm kind of- I'm only 90% sure that it's true, so I'm going to have to look into that. I might have to adjust this slide. We may need to adjust this recording if that's not true.

But 7,919 it returns out the 1,000th prime number. Great. Ok, what's the next piece that we're going to hit, Carl? Keep on going with this.

Carl: Next, you're going to declare a variable memoizedNthPrime. And it will be assigned the evaluation of memoized with the argument nthPrime.



```
function memoize(func){
 const cache = {};

 function inner (input){
   if( cache[input] ){
     return cache[input];
   }else{
     const value = func(input)
     cache[input] = value;
     return value;
   }
 }
 return inner;
}

function nthPrime(n){/* Heavy lifting logic (+ 500 ms)*/}

const memoizedNthPrime = memoize(nthPrime);
memoizedNthPrime(1000) // 7919
memoizedNthPrime(1000) // 7919 from cache
```

Phillip: Perfect, yeah, that's great technical communication, Carl. Right on, spot on. So, we don't know exactly what it is right now, but we know it's going to be the return value of running memoized, passing in the function definition of nthPrime. Great. So the code that we're running is memoizedNthPrime is equal to the evaluated result of running memoize, passing in the function definition of nthPrime, which is our blue function definition. Again, we know that this is- these two things are the same function definition. This function definition here is this function definition here, but we don't label it because again, we're just passing in the function definition. It's not the label nthPrime. We're going to name it something completely different inside our execution to memoize.

So, we're calling a new function, which, Donald, that means we're creating what?

Donald: A new execution context.

Phillip: Yeah, a new execution context. I guess it's starting to become a little bit more natural now. So we're creating a new execution context, there it is, and Donald, what is the first thing that we're going to do inside this call to memoize?

Donald: We're going to store the parameter, which is the function definition of nthPrime in local memory.

Phillip: Good, so what's parameter called? What's the label-

Donald: Func.

Phillip: Yeah, so func is equal to our function definition that we passed in, which was the function definition that was originally associated with nthPrime. Great. Alright. What's the next piece that we're going to hit, Donald?

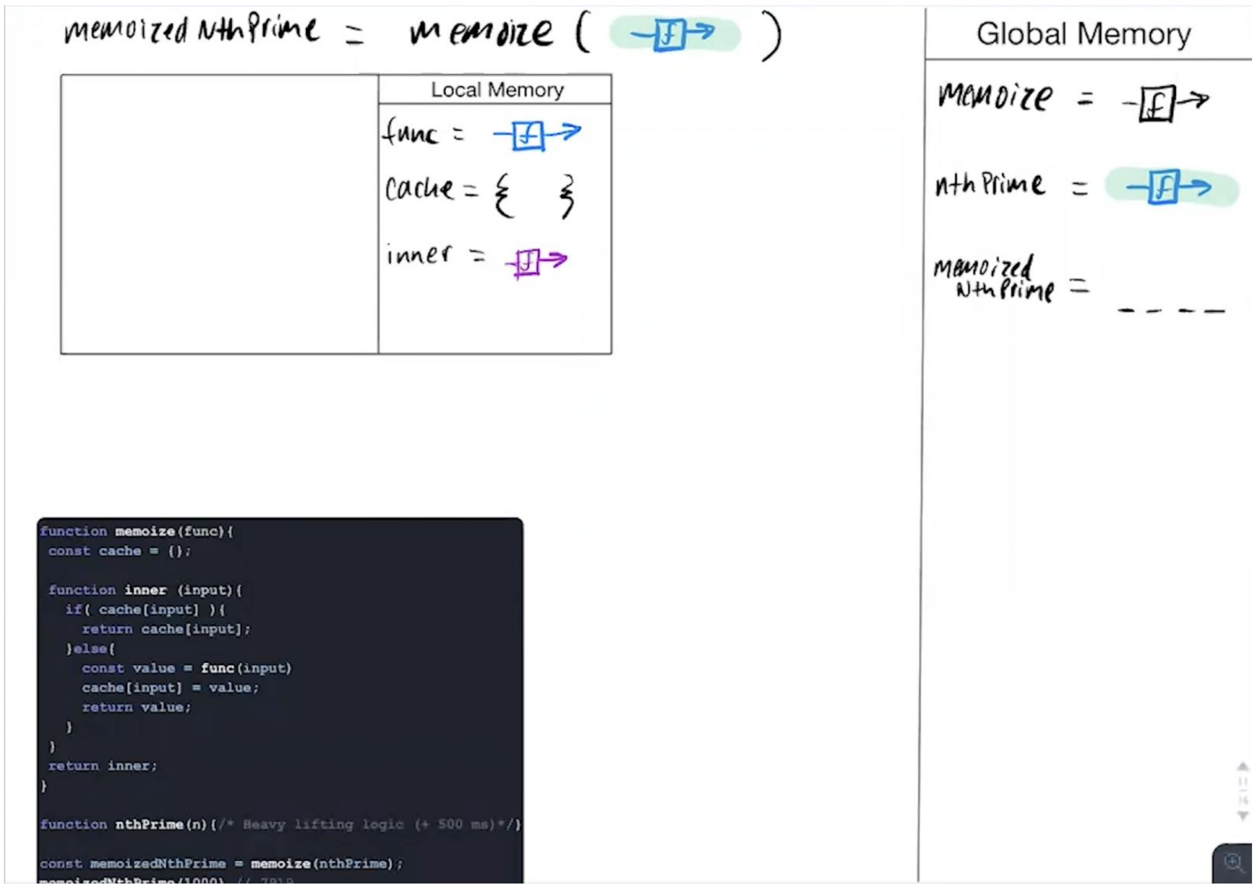
Donald: Sure, since we're executing it, we're going to declare a constant variable with the label cache and we're assigning it an empty object literal.

Phillip: Perfect. Exactly, so there it is, cache, and it is an empty object literal. Great. Next thing that we're going to do, Donald?

Donald: Yep. Sure. We're going to return the function definition of inner into the-

Phillip: Well, you skipped the part where we actually created the function definition.

Donald: Oh yeah, sorry. We create the function definition inner in local memory.



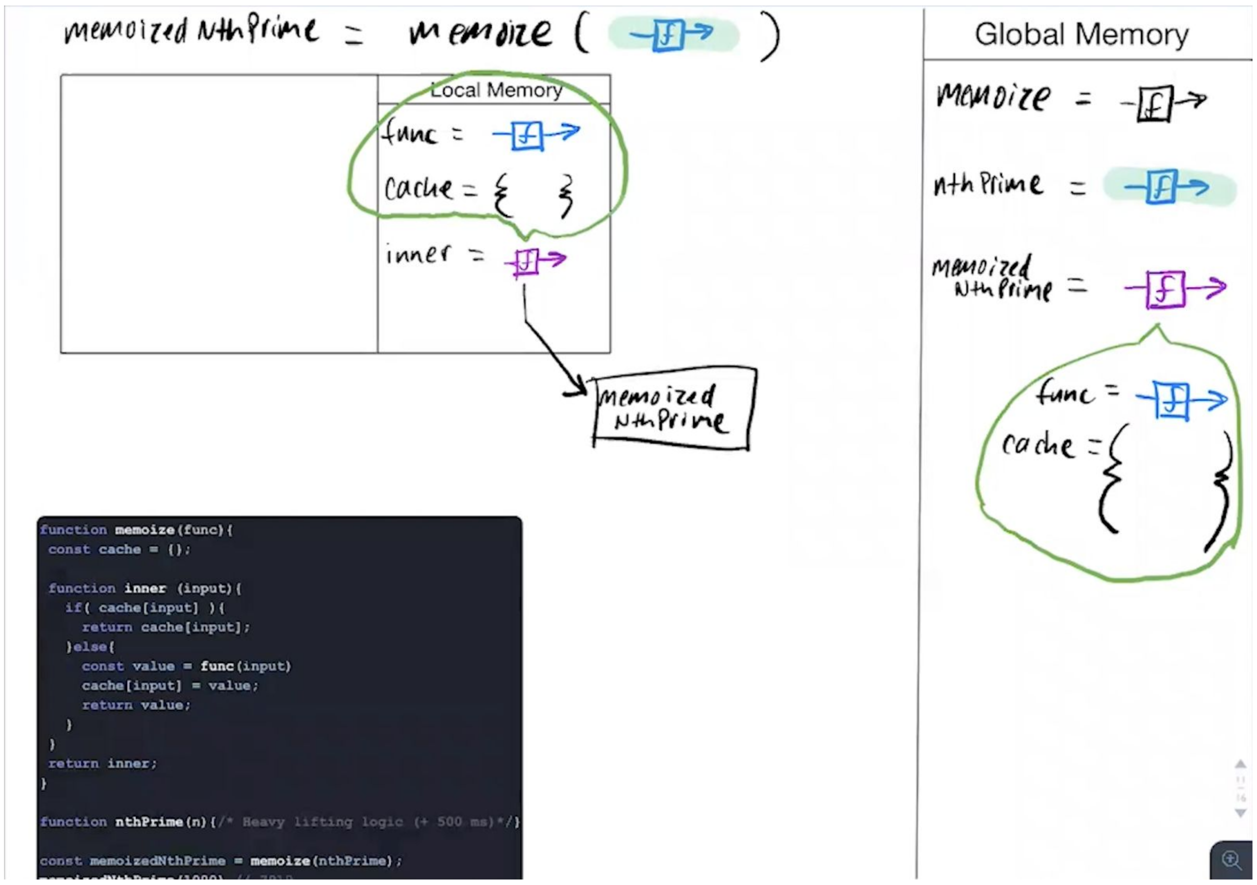
```
function memoize(func){
 const cache = {};

 function inner (input){
   if( cache[input] ){
     return cache[input];
   }else{
     const value = func(input)
     cache[input] = value;
     return value;
   }
 }
 return inner;
}

function nthPrime(n){/* Heavy lifting logic (+ 500 ms)*/}

const memoizedNthPrime = memoize(nthPrime);
```

Phillip:	And of course, I want to hit this step. One, because it's being thorough, of course, but also because there's another big piece that's going to happen right here of course that we know, when we create a function definition, what is the other thing that JavaScript automatically does for us under the hood, Donald?
Donald:	Yep. It automatically creates for us the closed over variable environment.
Phillip:	Perfect. Our closed over variable environment which grabs all of our live, persistent data. It's going to persist the data and all that live data that's around our- or that's adjacent to our inner function definition itself. Great. And now what's the next thing we do, Donald? You had it right the first time you said it.
Donald:	Yep. And then we're going to return the function definition as well as the closed over variable environment.
Phillip:	Perfect. And we return the function definition of inner, which is then saved in what new label in the global execution context?
Donald:	In the global memoizedNthPrime.



```
function memoize(func){
 const cache = {};

 function inner (input){
   if( cache[input] ){
     return cache[input];
   }else{
     const value = func(input)
     cache[input] = value;
     return value;
   }
 }
 return inner;
}

function nthPrime(n){/* Heavy lifting logic (+ 500 ms)*/}

const memoizedNthPrime = memoize(nthPrime);
```

Phillip: Yeah, memoizedNthPrime. Alright, there we go. So memoizedNthPrime is no longer uninitialized. It is now our function definition, which is our purple one. There it is. And attached to that, just like Donald said before, attached to that is also our closed over variable environment, or our backpack. It stores all of our stuff, all the stuff that we made a binding to inside of our call to memoize. So we have func, which is our blue function, which was originally our nthPrime function. So we have func, which is our blue function here. Good. Then we have our cache. Let's see if I can squeeze this in big enough here. I'm thoroughly impressed with my ability to draw curly brackets. That was pretty good. I know you guys are equally as impressed with that.

Ariel: I am pretty impressed.

Carl: Spot on.

Phillip: I appreciate that, yep, there you go. Alright. So, you can just add that to my list of resume items right there. Can draw the hell out of some curly brackets.

Okay, so now we've gone through kind of the setting up of our code. We've gone through this process now, it's still very similar process that we had going on with once, but we're setting up some logic that will be a little bit different from here. So now, big question: Carl, this one's for you. Big question, Carl. We now want to run the functionality that was originally associated with the label inner. But we want to call it in our global execution context. What is the code that we're actually going to run to run that functionality?

Carl: memoizedNthPrime.

Phillip: Exactly. memoizedNthPrime. Cool. And that's exactly what we actually have down here in our code. We're going to run memoizedNthPrime and because memoizedNthPrime is set up to take in an argument, we're going to pass it our argument 1,000. So we're going to do that. So let me pull down this code a little bit further now. And let's write out the code that we're actually running here, which is memoizedNthPrime, passing in an argument of 1,000. memoizedNthPrime, passing in our argument of 1,000.

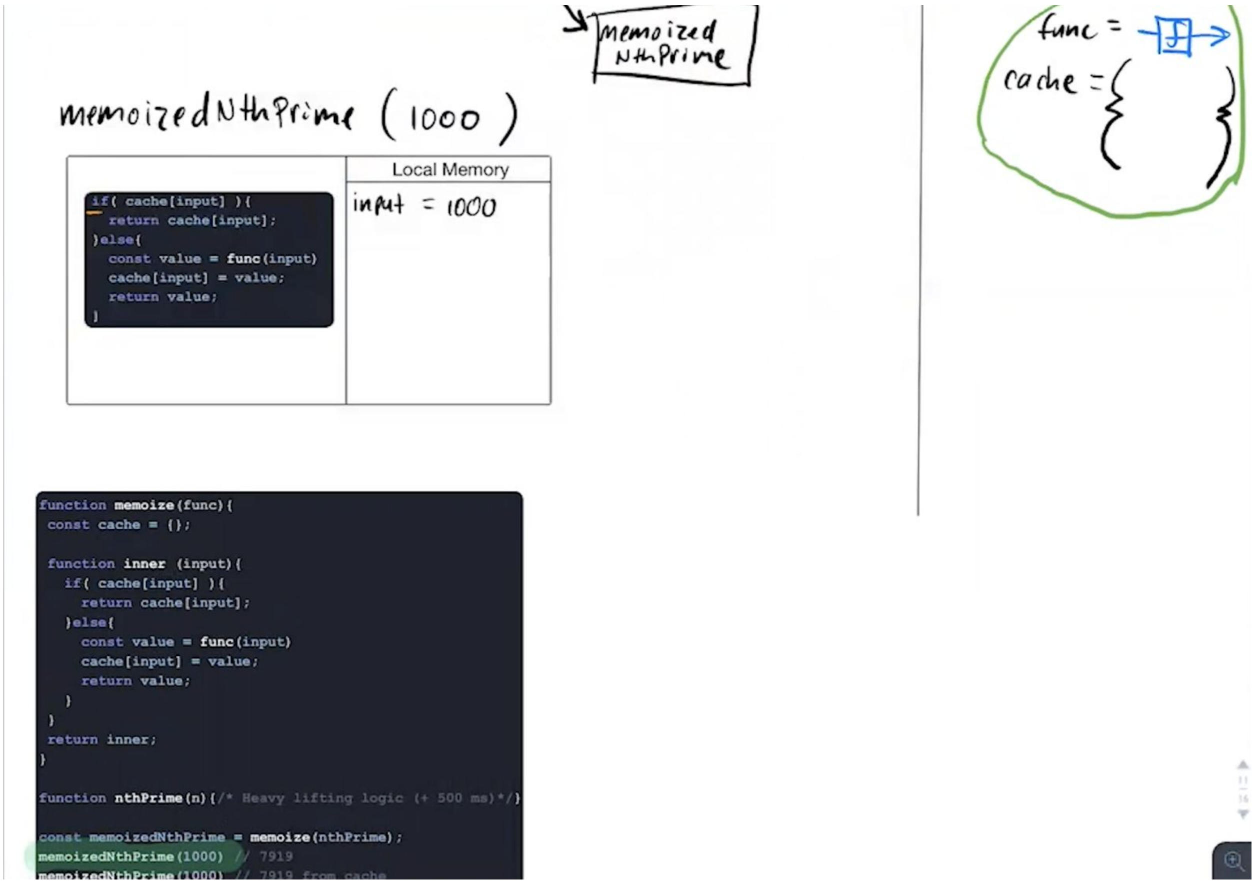
Alright, and Carl, when we call a new function, what are we going to create?

Carl: An execution context.

Phillip: An execution context, exactly. There we go. There's our memoizedNthPrime execution context and like you said, Carl, the code that we're actually running inside of memoizedNthPrime is the code that was originally associated with inner.

Ariel: Um, first you're going to pair the argument with the parameters, so input is now going to be equal to 1,000.

Phillip: Exactly. Input is equal to the number 1,000. Perfect. And then, Carl, as you were saying, we're now going to run the functionality or will be running the functionality of our inner function. So we're going to grab that code from here, copy, paste that. There we go. Shrink that down. Alright, and then Ariel, we hit this bit of code, JavaScript sees it, we have an if statement, so it knows it's going to be running some sort of conditional. The condition is cache, bracket, input.



```
if( cache[input] ){
  return cache[input];
}else{
  const value = func(input)
  cache[input] = value;
  return value;
}
```

```
function memoize(func){
 const cache = {};

 function inner (input){
   if( cache[input] ){
     return cache[input];
   }else{
     const value = func(input)
     cache[input] = value;
     return value;
   }
 }
 return inner;
}

function nthPrime(n){/* Heavy lifting logic (+ 500 ms)*/}

const memoizedNthPrime = memoize(nthPrime);
memoizedNthPrime(1000) // 7919
memoizedNthPrime(1000) // 7919 from cache
```

So let's go through this thing a little slow right here. The first thing technically, that JavaScript is going to see, is it's going to see cache. So where is it going to first look, because it's going to say, "Cache, I don't know what the hell that is." So where is it going to first look for cache, Ariel?

Ariel: In local memory.

Phillip: In local memory. Does it find cache in local memory?

Ariel: No.

Phillip: No, it doesn't. Ok, so where is the next place that it's going to look?

Ariel: It's going to look in its backpack.

Phillip: It's going to look in its backpack. So it's going to look out here, and is it going to find cache there?

Ariel: Yes.

Phillip: Awesome. So it's found cache, which is this empty object literal. So the next thing it's going to say is, "Ok, is there a property on this cache object that is equal to the evaluated result of input?" So first thing it's got to figure out is what the hell is input? So where is it going to look for input, Ariel?

Ariel: First in local memory.

Phillip: First in local memory. Does it find input there?

Ariel: Yes.

Phillip: Yes it does. What is input?

Ariel: 1,000

Phillip: 1,000. So now it's saying, is there something on the cache object, is there a key on the cache object equal to 1,000. Is there, Ariel?

Ariel: No, there isn't.

Phillip: There's not. So, what does- does this condition evaluate to true or false?

Ariel: It evaluates to false.

Phillip: It evaluates to false, so we're not going to run the code associated with the if block. Instead, we're going to run the code associated with the else block. So we're going to run this code down here. Ok. So we're going to keep walking through this step by step. Actually, Ariel, take me through, what is the bit of code running in the else block? What's the first piece that's happening?

Ariel: First you're going to declare a constant called value.

Phillip: Where are we declaring that?

Ariel: Inside local memory.

Phillip: Inside local memory, good. So we have our constant value. Do we know the value is of value at this moment?

Ariel: No.

Phillip: No, it's uninitialized right now, right? We have to go do some work. Specifically, we have to run the func function. So again, this starts getting a little bit convoluted here because I'm like, "Damn, what was func? It's so hard for me to remember what something else was." The best thing to do here is just keep following the mental model that we've been building out through this entire lecture. Which is, follow what it is that JavaScript would do. So when JavaScript hits that line of code, it's going to go, "func, huh, what is func?" Where's it first going to look, Ariel?

Ariel: In local memory.

Phillip: In local memory. Does it find it?

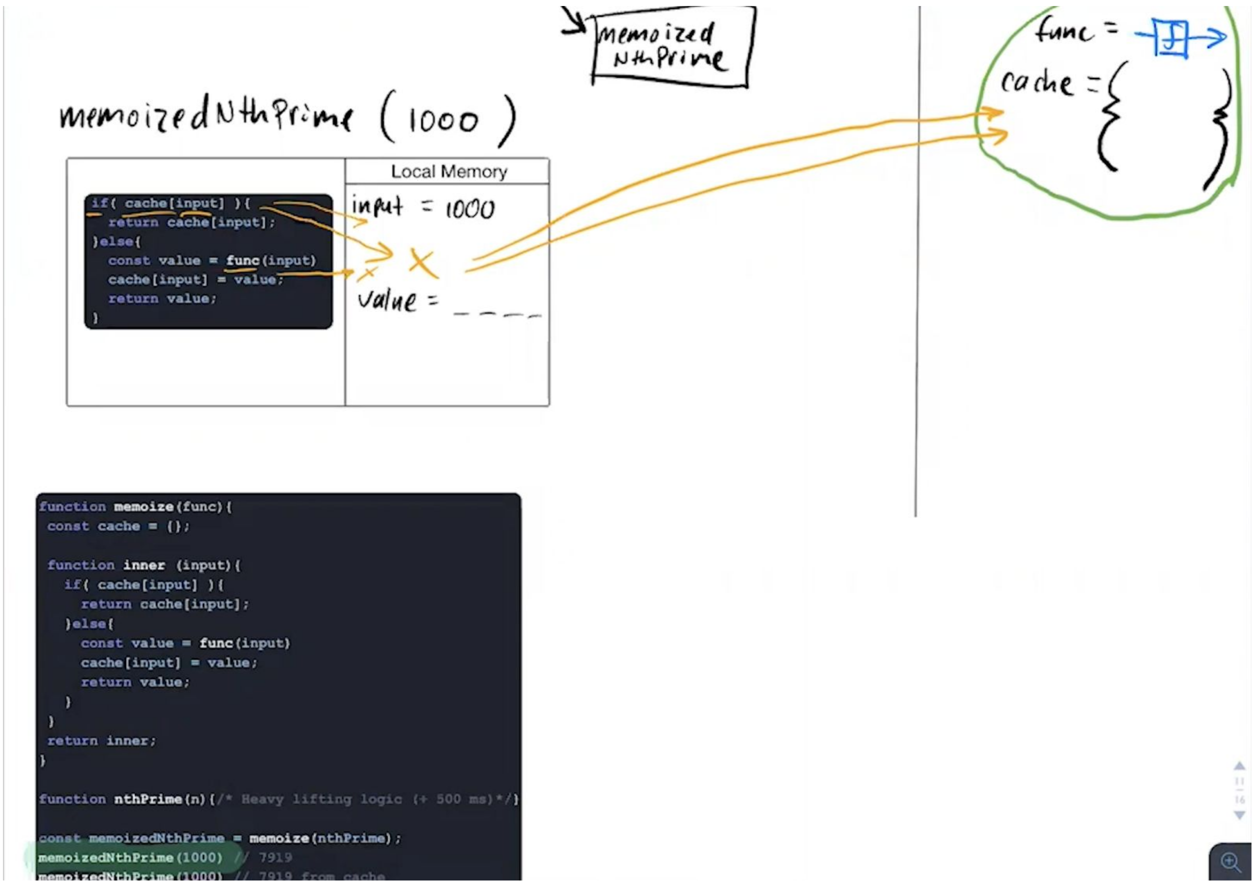
Ariel: Nope.

Phillip: Nope, so where does it look next?

Ariel: In its backpack.

Phillip: Backpack. Does it find func in its backpack?

Ariel: It does.



```
if( cache[input] ){
  return cache[input];
}else{
  const value = func(input)
  cache[input] = value;
  return value;
}
```

```
function memoize(func){
 const cache = {};

 function inner (input){
   if( cache[input] ){
     return cache[input];
   }else{
     const value = func(input)
     cache[input] = value;
     return value;
   }
 }
 return inner;
}

function nthPrime(n){/* Heavy lifting logic (+ 500 ms)*/}

const memoizedNthPrime = memoize(nthPrime);
memoizedNthPrime(1000) // 7919
memoizedNthPrime(1000) // 7919 from cache
```

Phillip: This is a weird sentence I've never said before. Does it find func in its backpack? So it does find func in the backpack. In the closed over variable environment. So what is func? Func is a function, which is good because we put parens on the end of it and tried to give it an input, so we wanted to be able to run some sort of functionality. And here's another big question. And we're putting all of this on Ariel right now because she's so far in deep in this function, I don't want to switch it up to anyone else. Ariel, what is, when we run func, what is- what was that functionality originally referred to as?

Ariel: The nthPrime function.

Phillip: The nthPrime function, exactly. We know that because we're just really good at following JavaScript code and also because we color coded it in blue so it's a little bit easier to follow. So nthPrime, which we have right here, is the same as this function. Same function definition as the one that we have inside of our backpack. It just has a different label. So when we call func, we're still really running the functionality that's associated with nthPrime.

Now if you remember at the very beginning I said we weren't going to actually go into the specifics of what nthPrime actually did because it was just be really difficult to whiteboard and it doesn't give us anything extra when it comes to closures, but we do know that we've set up the nthPrime function, aka also the func function, so that we give it an input, we give it an argument, it return to us that number-th prime. So if we give it 1,000, it returns to us the 1,000th prime number. So, what is the value, or what is the argument that we're passing into func, Ariel?

Ariel: The 1,000.

Phillip: We're passing it input, which evaluates to 1,000. Alright. So we pass in 1,000, and, Ariel, do you know off the top of your head, what is the 1,000th prime number?

Ariel: It's 7,919.

Phillip: Amazing. Amazing that you know that off the top of your head, Ariel. It is exactly that, it is 7,919. So 7,919 is returned from that function call. Great. So, value, again, we're kind of pseudo coding our way through this because we just don't have enough room to whiteboard everything. But we essentially know- let me erase a couple of these pieces here- so we essentially know that this bit of code right here is going to evaluate to- oh wait, it's not going to evaluate to 1,000. It's going to evaluate to 7,919. And that is what's going to be saved inside of our value variable. Ok, so let's backtrack that. There we go.

So our value variable is no longer uninitialized. It is now 7,919. Ok, I love it. That was a lot of talking on just that one piece of code. Ariel, what is the next piece of code that's going to run inside our memoizedNthPrime function call?

Ariel: Now, we're going to create a key on the cache object. It's going to be input, and that's going to hold the value of value.

Phillip: Yeah, absolutely, that was perfect. Absolutely. So if we were to break that down just for the sake of being very, very thorough on how that process worked out, is again- you know, you're kind of like, "Oh, I don't know what's actually happening here in JavaScript. I don't know what the values are for all these things." Think about how JavaScript would incrementally, very slowly go through this entire piece of code. So first it's going to say, "Cache. What the hell is cache?" First it's going to look in local memory. Not going to find it there. So then it's going to look into the backpack, and it's going to look for cache there. Does it find it? Yes, it does.

And then it says, "Ok, is there a value or a key on that cache object that's associated with the value of input." Input is 1,000. It says, "Nope. There is no 1,000th- there is no key on that cache object of 1,000." So what does JavaScript do? It creates it. And it sets its value equal to whatever value is. Ariel, right now, what is value?

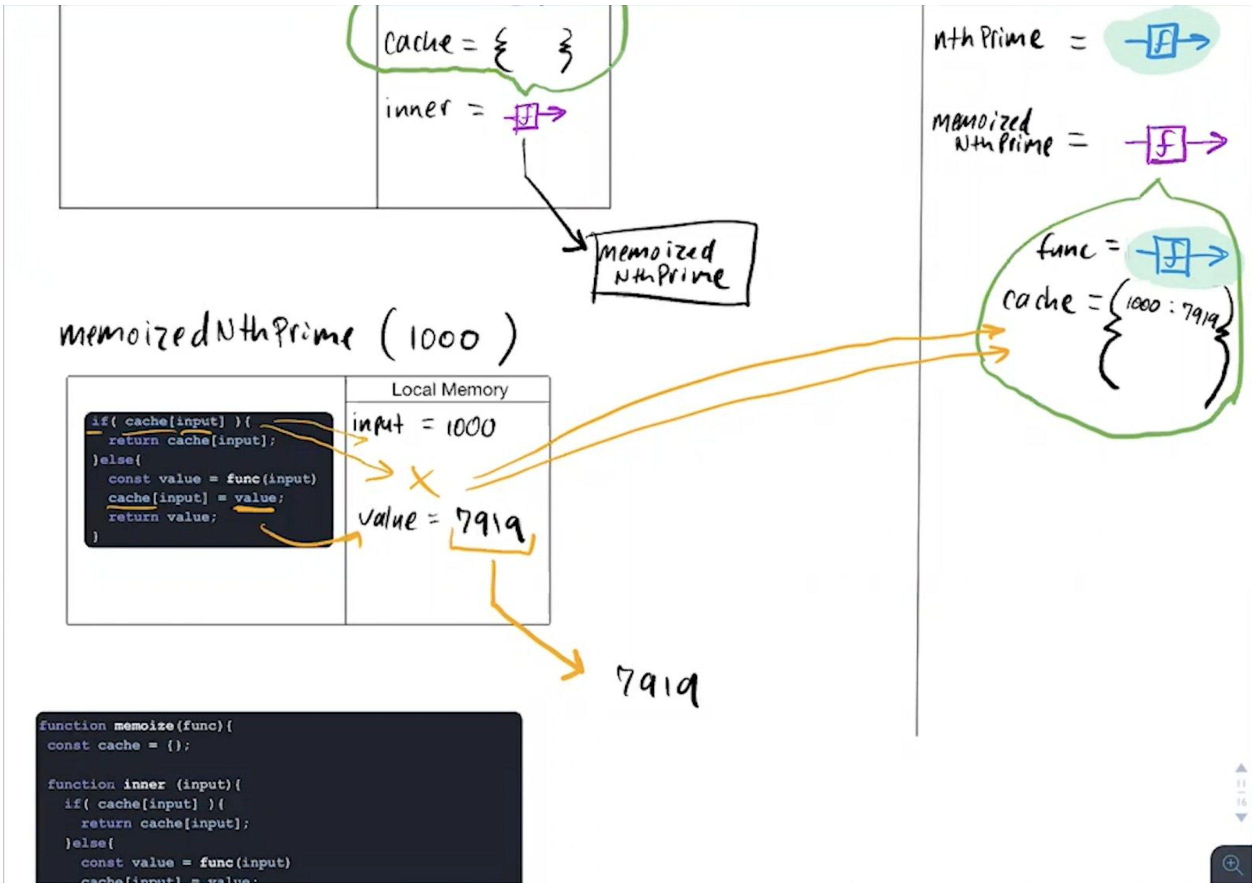
Ariel: It's 7,919.

Phillip: Exactly, because JavaScript knows that because it's going to look right here. It's going to say, "What is value?" 7,919. So now, we have a key on our cache object in our backpack of 1,000 with a value of 7,919. Ok. And then the final thing that comes from this bit of code, Ariel, is what?

Ariel: You're going to return value.

Phillip: We're going to return value. So we're going to return value.

Ariel: Or the value of value.



```
if( cache[input] ){
  return cache[input];
}else{
  const value = func(input)
  cache[input] = value;
  return value;
}
```

```
function memoize(func){
 const cache = {};

 function inner (input){
   if( cache[input] ){
     return cache[input];
   }else{
     const value = func(input)
     cache[input] = value;
```

Phillip: The value of value, which is 7,919, let's erase some of these x's so we can see what we're doing here. 7,919, and then we would do something with that value. So in our global execution context now, you would have 7,919. And again, remember what our ultimate goal here was, the application that Ariel, you were building out because you don't like the idea of building out tic-tac-toe, you were building out a function that when you push a button, it returns to you a number when you push the button it has a number associated with it. It returns to you that number-th prime. So you click 1,000, now we have this value of 7,919 that came from our function.

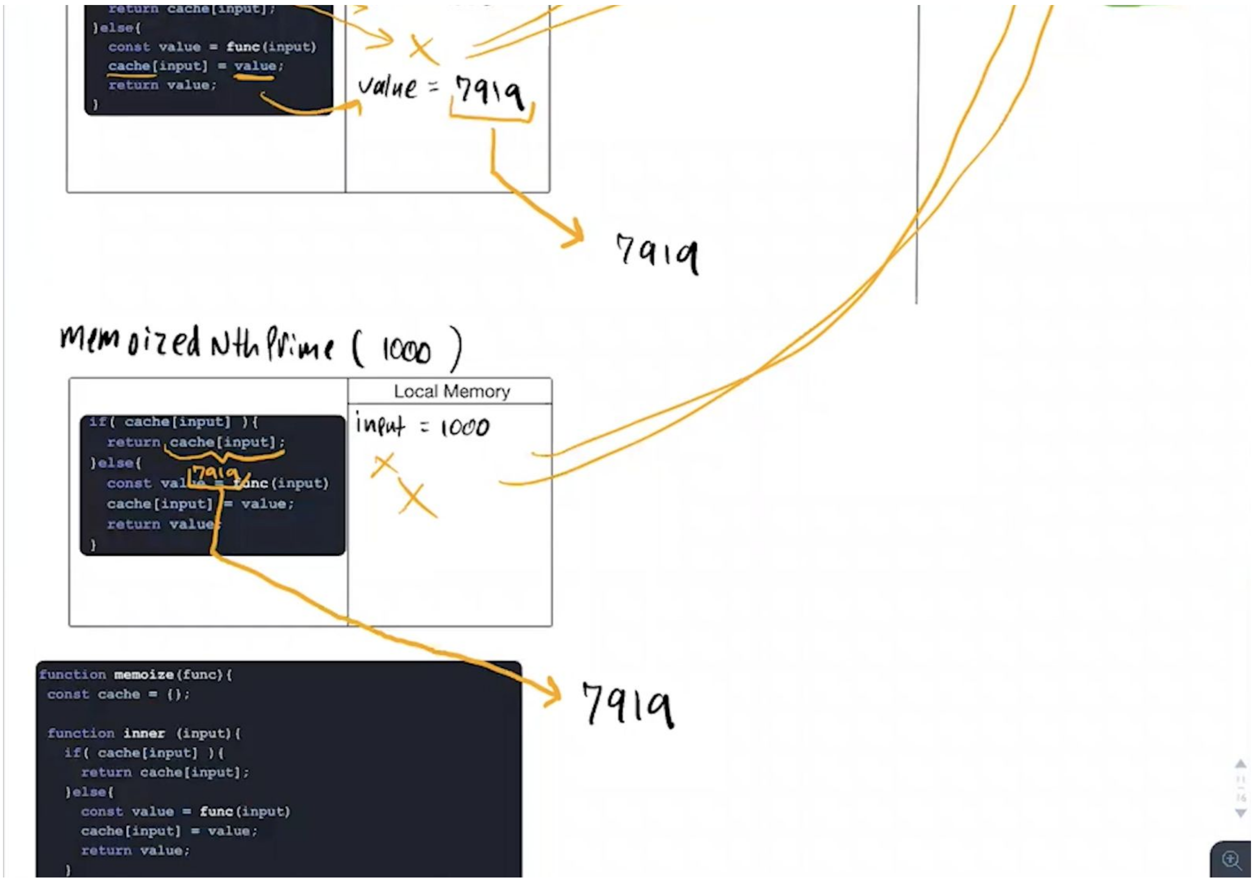
Then we'd take it and we'd display it on the dom or whatever the case may be.

Ok. Sounds good. That whole process we'll say took 500 milliseconds. A half second. There's a lot of work going on there. There's a lot of stuff, so we'll say it took 500 milliseconds. So let's run it again. Let's run it again and see now, now that we have things kind of set up and we have our cache object collecting data, let's see now how this would play out if we ran it a second time.

So let me move- and then we'll have questions on this piece as well. Because I know this is a lot to take in right now. So we'll move this thing down and let's run the code again of memoizedNthPrime. And we're going to pass to it again, 1,000. So let's say that our user in our application clicked that button a second time and wants to see again what the 1,000th prime number is. So we're calling a new function.

Ah, everybody, let's give it a shot. We're calling a new function, which means we are going to create a new-

Ariel: Execution context.



```
  return cache[input];
}else{
  const value = func(input)
  cache[input] = value;
  return value;
}
```

```
if( cache[input] ){
  return cache[input];
}else{
  const value = func(input)
  cache[input] = value;
  return value;
}
```

```
function memoize(func){
 const cache = {};

 function inner (input){
   if( cache[input] ){
     return cache[input];
   }else{
     const value = func(input)
     cache[input] = value;
     return value;
   }
```

Phillip: That wasn't bad. That wasn't bad. That was a good one. We create our execution context, there it is. Alright. And now, Donald, walk me through what's going to happen in our- actually, Donald, you did the first time I think. Carl, what's the first thing that we're going to do inside this memoizedNthPrime call? Let me scroll down here. So we're calling nthPrime. Sorry, we're calling memoizedNthPrime, which we know was originally the function inner, you told us that piece, so we have our inner function that's now running. So we know that that's this whole piece right here. Good. So walk me through, Carl. What's the first thing that we're going to do inside of our call to memoizedNthPrime that was originally the functionality of inner.

Carl: First thing that we're going to do is assign the parameters. We're going to assign it the actual argument.

Phillip: Perfect. What's our parameter?

Carl: 1,000.

Phillip: What's our parameter?

Carl: The parameter is input.

Phillip: The parameter is input. The argument is?

Carl: The argument is 1,000.

Phillip: Perfect. There we go. Alright. And then what's the next thing? Well, I mean the next thing is that we're going to run all of our conditionals and stuff, so let me cut that piece out so that we can get a good, clear view of it. So the next piece is that we're going to run all of this code right here, copy that, paste that there, shoot that down, alright.

Alright, so our next piece, Donald, I don't think you've taken us through this next piece yet. Donald, our next piece, we have our if conditional here. JavaScript says, "Ok, we're going to evaluate a condition." Condition that we're going to evaluate is cache input, or cache bracket input. So the first thing it's going to do is it's going to say, "Cache, what is that?" Where's the first place it's going to look for cache, Donald?

Donald: It's going to look in local memory?

Phillip: Good. Does it find it there?

Donald: No, it does not.

Phillip: No, it doesn't. Where's the next place it looks out to, Donald?

Donald: The closed over variable environment.

Phillip: Perfect. The closed over variable environment. Does it find cache there?

Donald: Yes, it does.

Phillip: Perfect. So then it moves on to its next piece, which is- first it's going to say, "Input, well, what the hell is input?" What is input at this point, Donald? Or where does JavaScript look for it first?

Donald: It looks for it first in local memory.

Phillip: Yep. Does it find it?

Donald: Yes, it does.

Phillip: Yes, it does. What's the value of it?

Donald: 1,000.

Phillip: 1,000. So now when JavaScript says, "Ok, I know what cache is, I know what input is, now is there a key of 1,000 on our cache object?" And, Donald, is there?

Donald: Yes, there is.

Phillip: Yes, there is. So this condition, let me get rid of some of these lines here. This condition that we have going on right here, does it evaluate to true or false?

Donald: It would evaluate to true.

Phillip: It evaluates to true, which means we are going to run the code that's associated with our if block, which says return cache at input. So again, JavaScript goes through the same process again. It says, "Ok, return, I know what that is. That's the keyword that I know in JavaScript that says return a value into the next execution context outwards."

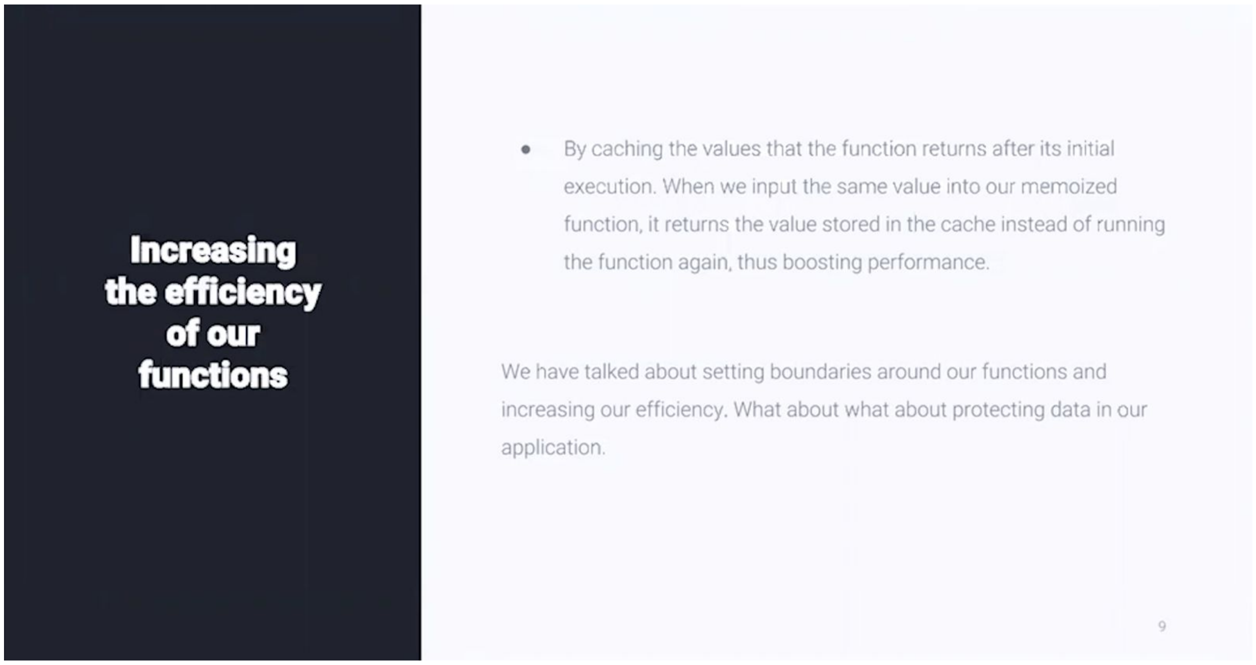
Ok, so what am I going to return? Cache, what is cache? So again, looks in local memory, doesn't find it, looks into its closed over variable environment, where it does find it. And then it says, "What is the key input?" Input is 1,000, so it says, "What is cache at the 1,000th key? What is the value associated with that?" Donald, what is the value associated with cache at the key of 1,000.

Donald: The value is 7,919.

Phillip: 7,919, absolutely. So it takes that value, which is- let me erase this. This whole piece here evaluates to 7,919, which it then returns. Takes that, returns it out. We're going to return it way over here so I can see it. It returns out the value 7,919. Now again, why was that so special? Well, the first time that we ran that, we had a lot of talking to do, a lot of figuring out of what these value were, where were they being stored, what this function was, so on and so forth. On this implementation, because all that data was already saved inside of our closed over variable environment, one-step lookup. We didn't have to go through the process of running nthPrime. We just went, grabbed it from the cache, and returned it out the bottom of the function.

So in reality, we can probably assume that this took like 1 or 2 milliseconds, as opposed to 500 milliseconds as it did on the first implementation, or the first running of memoizedNthPrime. This will drastically, drastically improve the efficiencies of our application. Think about how- as soon as all those buttons have been pushed one time, now they're just like that (snaps), every single time they're pushed. Just quick, popping out those values, not having to go through the whole process of figuring out the nthPrime functionality. Amazing, amazing implementations into our application. Well done, Ariel. It's a good thing that you put nthPrime into that application that you're building.

Lecture: 6 Q&A



Increasing the efficiency of our functions

- By caching the values that the function returns after its initial execution. When we input the same value into our memoized function, it returns the value stored in the cache instead of running the function again, thus boosting performance.

We have talked about setting boundaries around our functions and increasing our efficiency. What about what about protecting data in our application.

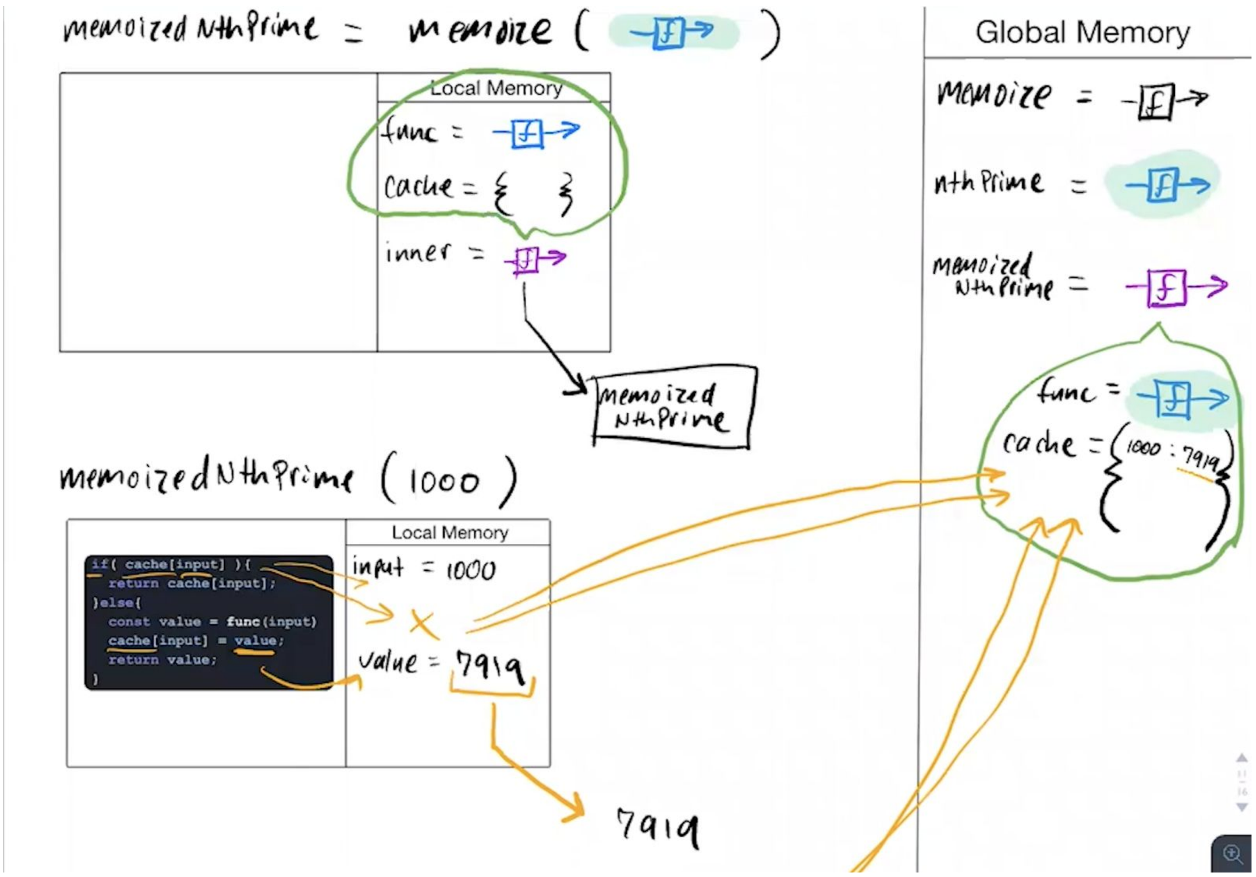
9

Phillip: Honestly guys, this is super pro-level stuff when it comes to using closure. We've talked about setting boundaries around how many times we could run a function in an application's life cycle. We've talked about how to persist data from function call to function call to make our application more efficient. This is complex stuff.

Let's have thumbs on these concepts so far. [Thumbs up] I'm clear; I'm ready to move onto something with more of a broader scope maybe, I don't know, an entire JavaScript design pattern that uses closure. [Thumbs sideways] I have some questions. [Thumbs down] No idea what's going on.

Ariel's got a question. Good. What's your question, Ariel?

Ariel: Ok. I'm following you. I'm following the execution of running closures. At first I was like, "I guess I just need to let it absorb a little bit more." At first, I was like, "Well I could do that using I guess reduce or something," but the take-home is that the environment is persisting every time that the function is running versus when the function is finished, you can't really touch that information anymore.

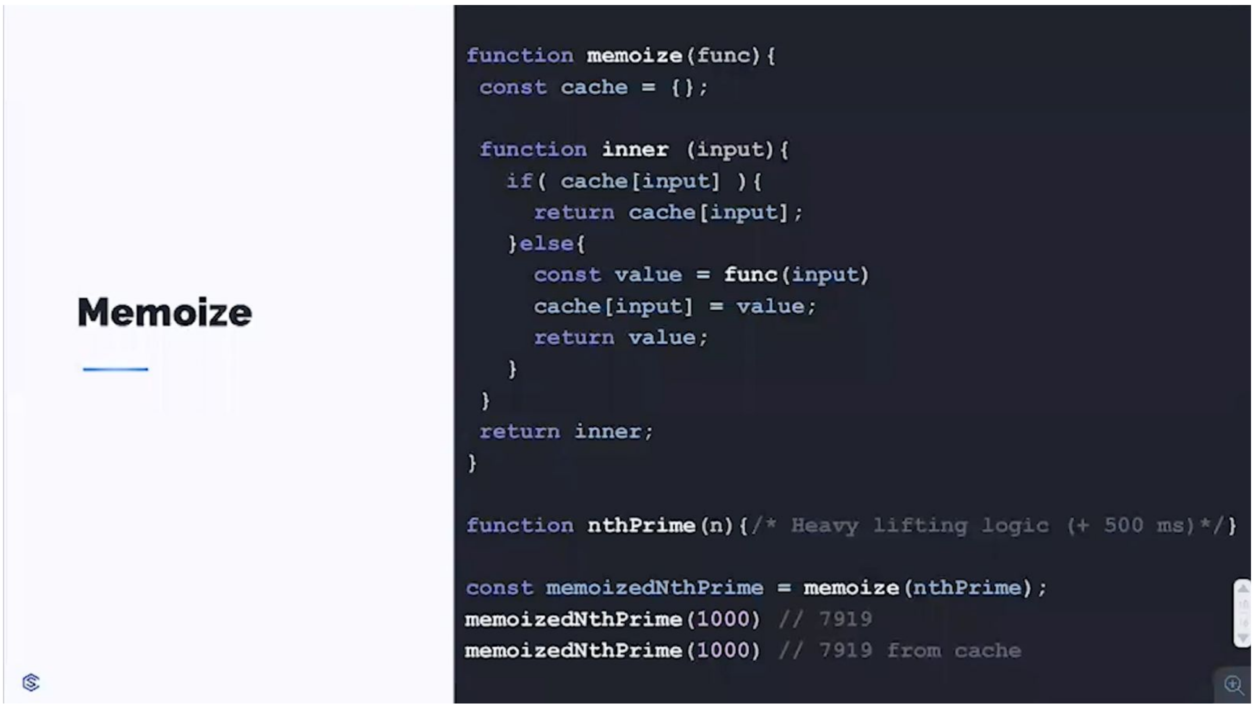


```
if( cache[input] ){
  return cache[input];
}else{
  const value = func(input)
  cache[input] = value;
  return value;
}
```

Phillip: Yeah, absolutely. The implementation of this, of figuring out what the thousandth prime number is, yeah absolutely. We could do that all day long. It may take us a minute how to do the logic for it, but we can do that all day long. The main take-away, the benefit that you're getting out of using closure to do this, is that we're making it way more efficient to do this on any subsequent calls. That's only possible by being able to persist memory from function call to function call, and specifically what we're storing in memory from function call to function call. In this example, we're storing arguments and return values from previous calls. Yeah, absolutely.

Carl, what's your question?

Carl: It just seems like in order to initiate closure, you have to call, to run through the function at least once in order to get the evaluation of the data in order for it to be stored in global memory, or the backpack.



Memoize

```
function memoize(func){
 const cache = {};

 function inner (input){
   if( cache[input] ){
     return cache[input];
   }else{
     const value = func(input)
     cache[input] = value;
     return value;
   }
 }
 return inner;
}

function nthPrime(n){/* Heavy lifting logic (+ 500 ms)*/}

const memoizedNthPrime = memoize(nthPrime);
memoizedNthPrime(1000) // 7919
memoizedNthPrime(1000) // 7919 from cache
```

Phillip: Yeah, without that first call to memoizedNthPrime, it doesn't have anything stored in that cache, so there's stuff we can do to get around that piece of it. We just, at some point, we have to go through the logic of figuring out what the thousandth prime number is, or what the ten thousandth prime number, or the millionth prime number is. But, the good piece is- or the improvement that we get it, is the efficiencies that we get, is the optimizations that we get, is when we run it a second time.

Carl: Right.

Phillip: And we don't go through that whole process that we would if we didn't have closure. If we didn't have closure, think about it, we could sit there and click 1,000 every single time, like 10 times in a row, and every single time that we click it, JavaScript doesn't remember anything inherently. It doesn't remember previous calls inherently, we as developers have to set up that cache, we have to set up that closure to remember those things. Otherwise, every single time you click it, it takes half a second. It takes half a second. It takes half a second.

Now, we click once, half second. We click it again, one millisecond. One millisecond. One millisecond. Very, very quick. Very, very quick.

Carl: I think you said it before, when you have multiple calls, in that step, you'll start to see the real benefit, because it keeps it on DRY.

Phillip: Yeah, after you have multiple calls to memoizedNthPrime, that's when you'll seek the optimizations. Absolutely. Cool. Any other questions that we have before we move onto our final piece?