

Lecture: 1 Introduction to Closure Advanced Problem Sets



Phillip: Alright, guys. Welcome to JavaScript the Hard Parts, the Advanced Problem Set, kind of the next piece onto the lecture that you just went in-depth on as to how closures actually work in JavaScript. For this portion of the unit, we're going to be diving even further into some more advanced topics and some more advanced techniques on how we can actually use closures to benefit us in the applications that we're building out.

JavaScript the Hard Parts

Closure

Advanced Problems



Closure is the most esoteric of JavaScript concepts



Enables powerful pro-level functions like 'once' and 'memoize'



Many JavaScript design patterns including the module pattern use closure



2

Before I start jumping into all of the whiteboarding though, I do want to do a quick recap on some of the more powerful things and powerful features that we talked about in the previous lecture that go hand-in-hand with closures.

Now, we talked about a couple of things. Specifically, we talked about being able to use closure to do and create these really pro-level functions, things like the once-ify function or the once function, making the ability or gaining the ability to set up rule sets and logic around how many times we can actually run a function in our application.

We kind of touched a little bit onto why that might even be necessary. Why in the hell would we ever need to do that? I want to talk more about that in this section. We also talked about using the memoize function or memoizing a function, the ability to persist memory from function call to function call. We know that's an innate ability of what closure is, but using that memory and that persisted cache in a very particular way, using it in a way that allows us to attach this persistent cache memory to these really heavy lifting functions. Something like maybe, I don't know, like nth prime, a function that takes in an argument, does a bunch of computational steps and then returns out the nth prime number, so like the 1000th prime number, something along those lines.

It's a very heavy lifting function. This function may take 500 milliseconds, but using closures, we can set up this persistent cache that says, "Hey, every time this function is called, remember what the argument is that's being passed in and also what the result was that we passed back out, that we returned out." That way, if we ever get called again with that same argument, we don't have to do all the heavy lifting and functionality. We simply just return out the same value that we gave out before. So drastically, drastically increasing the efficiencies of our applications.

Then we also talked about some more broader concepts, more powerful concepts, I would even say, such as entire design patterns in JavaScript, design patterns like the module pattern, things that are very, very common right now in modern JavaScript. The module pattern allows us to do things like declare data, variables, whatever the case may be, in a way that it can persist throughout our entire application's life cycle.

Well, the only way that we would know how to do that without closures is to simply define something in global memory, but we know as engineers that's generally a bad practice. We don't want to have things hanging out in global memory because it's very easily overwritten by, let's say, another engineer on our team.

Well the module pattern, via the power of closures, allows us to protect that data in a very specific way. So all of these pieces right now, I want to go a little bit more in-depth on. I want to see actually what it looks like in code and as we actually whiteboard it out and gain that mental knowledge, that mental model, of how closures and how JavaScript is using these pieces.

Lecture: 2 Using Closure to Create a Once Function

Once

```
function once() {  
  let hasBeenRun = false;  
  function inner(){  
    if(hasBeenRun === false){  
      hasBeenRun = true;  
      return 'Congratulations, you won!';  
    }else{  
      return 'You cant run me again';  
    }  
  }  
  return inner;  
}  
  
const oncifiedWinner = once();  
oncifiedWinner();  
oncifiedWinner();
```

Phillip:

So, let's go ahead and get started with those pieces. And the first on our list is this "once" function. Again, this is a function that we utilize, using closure, to create another function that will ultimately allow us to specify we only want this function to be able to be ran one time. This is a pretty common thing, actually, in an application. Let's say the situation was something like, Donald. Let's say that Donald is building out a tick tack toe game. So he's building out a tick tack toe game and you have a function in that tick tack toe game, Donald, that is called winner. It's just called the winner function. And when somebody wins the tick tack toe game, you run that function in your application and that function manipulates the dom and makes the background turn a bunch of colors and music plays and confetti hits the screen and all this crazy stuff because somebody just won the game. Fantastic. Well you only ever want that function to be able to be ran in your application one time because there's only one winner in a game of tick tack toe. So this is a very kind of a silly, really simplistic example, but this is definitely a time where you might want to once supply that winner function, so that it can only ever be able to be called one time.

Then maybe if it's called any more times it doesn't run any code at all. Or maybe it simply returns out a string that says "Sorry, you've already called me once. You can't run me anymore than that." Something along those lines.

So, this is good. This sounds like a good idea, but how do we actually do this? What does this look like in code? And what does it look like under the hood when we start white boarding this stuff out? Well that's what we're going to find out right now with the code that you guys see in front of you.

So, we have our code right here, it's our once function. We're going to go through this code line by line technically communicating our way as best we possibly can through each piece, specifying when our closed over variable environment is created by the `[[scope]]` property, and all those pieces all together.

Let's start seeing what that would look like. I have my basic set up right here on my white board. We have our global memory right here on the right. We have our actual code snippet over here on the bottom left, and we're going to be white boarding our way through this.

Can you guys see the code okay right now? How big it is. Okay.

So, the thing is with these more advanced problems is that it's a little bit harder to fit it all on one white board. So, as we go through, you're going to see me kind of messing with stuff as I shrink it down and move it down closer to the bottom of the page and that type of thing. So, just bear with me as we get to those kind of more difficult pieces.

Let's go head and get started here and, Donald, why don't you take me through? What's the first pieces that we're actually doing inside this code?

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

Phillip: Perfect. Declaring our function once in global memory. There it is. Awesome. Keep us going, Donald. What's the next piece that we're actually going to write?

Donald: Sure. Now we are going to declare const label onceifiedWinner and we're going to assign it the return value executing the once function.

Phillip: Perfect, perfect, perfect. So, onceifiedWinner, and we're assigning it, yeah, exactly what you said, the return value of running our actual once function. Cool, so for a split second right here, we don't know exactly what onceifiedWinner is. It's kind of in this state of uninitialized. So, I usually just indicate that by just giving us some dotted lines over here.

So to figure out what that value is going to be, we need to go run some code. The code that we're actually running right now is onceifiedWinner is equal to the evaluated result of running our once function. Perfect.

So, I know we usually get really high and mighty on this and we get all amped up and we all say the answer to this next question is a little bit more difficult. It's a little bit underwhelming when there's only three of us in the room. So, I'm going to kind of call one of you guys individually as we go through these things. So, Carl, tell me man. I know you know the answer to this. And you don't even have to say it in the nice sing-songy voice. When we call a new function in JavaScript, what do we create?

Carl: An execution context.

Phillip: Yeah, we create a new execution context. You can still see, you've done it so many times, it can be hard to get out of the habit of saying "[singing] execution context." It's a little weird. Yeah, you're absolutely right man. We create a new execution context.

So here we go, we have our execution context right there which now means we're running code inside that execution context. Ariel, why don't you take me through what's the first thing that we're doing inside this call now to the once function?

Ariel: You're going to create a variable called hasBeenRun.

Phillip: Hmm! And where do we store that at?

Ariel: In local memory.

Phillip: Perfect. hasBeenRun. And what do we initialize it to?

Ariel: False.

Phillip: False. Good. So we're initializing our hasBeenRun function, or our hasBeenRun variable to be equal to false. Good. What's the next thing that we're going to do, Ariel?

Ariel: Then you are going to store the definition of the function in local memory.

Phillip: Perfect. Alright, and I'm going to change the color of the inner function like definition itself. Not because it's any kind of special thing, but just because I want us to be able to very distinctly follow where that function definition is traveling throughout our white board and which labels we then have attached to it at different points in time. So, we are going to create it as being this blue function right there.

Alright, now here comes a review question from the lecture that we just went through. One of the things that happens automatically in JavaScript when we create a function. Now this, by the way, happens all the time. Every single time a function is created in JavaScript, this piece happens under the hood. Now, we don't talk about it all the time because many times it's not that interesting. It doesn't really become interesting until the next part where we return a function out, but Donald, in JavaScript when we create a function specifically like in this situation here where we're creating the function within another function, what is the other piece that gets created here that JavaScript automatically gives us?

Donald: Yeah, it automatically gives us the closed over variable environment.

onceifiedWinner = once()

Local Memory	
	<i>hasBeenRun = false</i> <i>inner = J →</i>

Global Memory

once = F →

onceifiedWinner = _ _ _ _

```
function once() {
  let hasBeenRun = false;
  function inner(){
    if(hasBeenRun === false){
      hasBeenRun = true;
      return 'Congratulations, you won!';
    }else{
      return 'You cant run me again';
    }
  }
  return inner;
}

const onceifiedWinner = once();
onceifiedWinner();
onceifiedWinner();
```

Phillip: Exactly. So we automatically, in the less technical terms, we create this kind of binding to all of the data that is adjacent to our inner function. So all of the data that's currently in our local variable environment, we create this binding. I usually just draw this out in green to keep it consistent across all of the white boards. We create this binding that attaches itself to our function inner.

Good. Now kind of bonus question here, Donald, what is the property that specifically creates this binding in JavaScript? Do you remember?

Donald: Yes, I believe it's the property `[[scope]]`.

Phillip: Absolutely. Let's see how this works here. Let's see if this looks good. We create the `[[scope]]`. Right there. Perfect. And that property is a hidden property. I mean, I guess it's not necessarily hidden. You can see it in the dev tools, in like Chrome dev tools or something like that. But you can't access it or manipulate it directly. But we do have this `[[scope]]` property creates a binding to all of the live data that's currently in our local memory. Great.

So what's the next piece that we're actually going to hit, Donald?

Donald: Sure. We're going to actually return the definition of inner, store it, so `onceifiedWinner`.

`onceifiedWinner = once ()`

Local Memory
<div style="border: 1px solid green; border-radius: 50%; padding: 2px; display: inline-block;"> <code>hasBeenRun = false</code> </div>
<div style="border: 1px solid blue; border-radius: 50%; padding: 2px; display: inline-block;"> <code>inner = [function]</code> </div>

Global Memory

`once = [F] →`

`onceifiedWinner = -----`

```

function once() {
  let hasBeenRun = false;
  function inner(){
    if(hasBeenRun === false){
      hasBeenRun = true;
      return 'Congratulations, you won!';
    }else{
      return 'You cant run me again!';
    }
  }
  return inner;
}

const onceifiedWinner = once();
onceifiedWinner();
onceifiedWinner();

```

Phillip: Exactly. So just to kind of recap that really quickly here. Let me erase these lines. So just to recap that really quickly, the first thing that we did is we created our `hasBeenRun` variable. We initialized it to `false`. Then we declared our function "inner" which took all this code here and packaged it up all inside this function definition right there. And then, the next piece that we're actually going to hit is our return statement. So it says "return inner". And when JavaScript hits that line of code it's going to go "Oh return, I know what that is. Sure, yeah. It means I'm going to return some value out into the next execution context outwards". But then when it sees inner it goes "Well, what is inner?" So where does it look? It looks inside the local memory where it's currently running code. And it sees that inner, right here, is that function definition. So, what we do and let me clear some of this off, is we're returning this function definition out and what's the label that's going to catch that function definition, Ariel?

Ariel: `onceifiedWinner`.

Phillip: Yeah, `onceifiedWinner`. Perfect. So now inside of our global execution context we no longer have `onceifiedWinner` as being in this state of uninitialized. It now has a value. And that value is our function definition, our blue function definition specifically. That was once referred to as, it once had the title, a label, we can say, inner. But now on the global execution context it now has a new label `onceifiedWinner`. But with it also comes our binding to all of that live stored data that we had when inner was kind of created

inside the call to `once`. So inside of our global memory, we now still have this closed over variable environment or this closure as some people refer to it as, or my favorite of course we all now is the backpack. We have inside of our backpack we have a value `hasBeenRun` and it's still initialized to the value of `false`. Perfect.

Now comes the big question, Ariel. Let me scroll up so you can't see the answer on the code. Now comes the big question. We now have return that function definition out of our call to `once`. Now if we wanted to run the functionality that was originally referred to as `inner` but we wanted to run that now in the global execution context. What is the code that we would actually run?

Ariel: We are going to have to invoke `onceifiedWinner`.

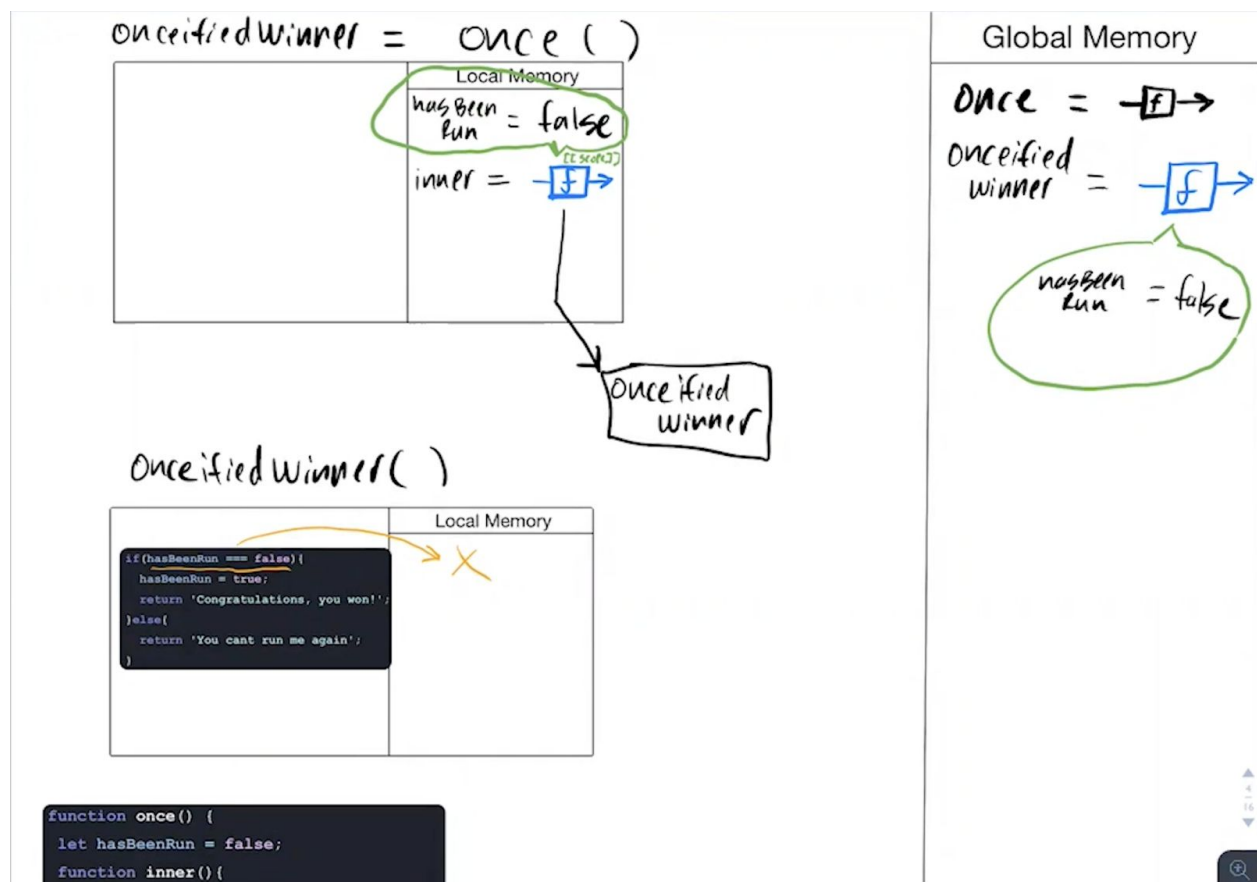
Phillip: Perfect, exactly. We are going to have to invoke `onceifiedWinner`. You see how the blue function definition really really plays out well here, because we know that when we created `inner`, that was our blue piece right here, we returned that out, saved it under a new label. But it's still the same data, the same code that we're going to be running. It just has a new label. Awesome.

Okay, so let's actually go through that process now. Let's run our `onceifiedWinner` function. And again, I know we get really into the weeds here talking about all these little tiny specific pieces of how exactly JavaScript is working with these closures and backpacks and all these different things. But also remember, what was our ultimate goal with this was to create a function winner for Donald's tic-tac-toe game and only allow it to be ran one time ever and that was it. So keep that piece in mind because we're going to see how that plays out right now.

So now we're calling for the very first time our `onceifiedWinner` function. Okay, let's see how that works. So, yup, we're running, just so you guys can follow along here, down here at the bottom we're running now our first call to `onceifiedWinner`. So here's our call or here's our actual code that we are running, `onceifiedWinner`. Perfect. And Carl, we're calling a new function, man. What does that mean that we're going to create?

Carl: A function level execution context.

Phillip: You had to change it up there a little bit. A function level execution context. Just so it didn't sound as monotonous. Yeah. So we created our execution context. Our local function level execution context. Awesome.



Okay, so we're now running `onceifiedWinner`, which we just said was originally the code that was associated with the function definition of `inner`. So if we look at the function definition of `inner` inside of our original code, we can see that it was all of this piece right here. All of this code. So, that is going to be the code that we are now running inside of our `onceifiedWinner`. So, we're just going to go ahead and grab all of that directly from here, copy that, and then paste that. Too big, let's make it a little bit smaller. Can you guys still see that okay? Not too bad? Okay, cool.

So now we're going to be running the code that's inside `onceifiedWinner`. So, first thing that JavaScript does when it starts running this code is it sees that there's an `if` statement, and it goes "Oh, I know what that is. That's a key word in JavaScript. I know what 'if' is. It means that we're going to be checking some sort of condition". And the condition that it's checking is "`hasBeenRun = false`". It's checking to see is the variable `hasBeenRun`, is it equal to the value of `false`?

So when JavaScript first hits this line of code, where is it going to look for `hasBeenRun`, Carl? We're inside of our call to `onceifiedWinner`. Where is it first going to look for the variable `hasBeenRun`?

Carl: It's going to look in the local memory first.

Phillip: Yeah, it's going to look in the local memory of the onceifiedWinner function. So it looks in there, and it says "Hey, is hasBeenRun here"? And does it find it, Carl?

Carl: It does not.

Phillip: It does not. Okay.

Now before we started learning about how closures works, the next logical place that we would think it would look would be global memory. But we know that's not true at this point.

Carl, where does it actually look next for the value of hasBeenRun?

Carl: It looks in the memory of the function in which it was defined.

Phillip: Hmm. I want to be really, really technically clear here. I want to make sure that we don't think that at this point now JavaScript would look upwards back into the function invocation of once. That's not what happens. I mean, at this point right now, if we really wanted to be specific, this entire execution context is gone. It's de-allocated from memory. It's deleted. There's no way that we can actually look back up in there. But, something did come out with it that was part of the memory of that invocation. So, in reality, you're kind of still right in a sense, Carl. But I want to make sure that we're very, very technically specific here. We don't look back into the memory of once. We look into the memory that was kind of returned out of once, which was our backpack. Our closed over variable environment which is that green piece over here. So, in reality, the next place that we look after local memory is we look inside our backpack. We look inside of our closed over variable environment.

Now Carl, do we find the variable has been run inside the backpack?

Carl: Yes.

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

Carl: It is false.

Phillip: It's false. So, this condition that we were checking over here in our code, is it going to evaluate to true?

Carl: Yes.

Phillip: Yeah, it's going to evaluate to true. hasBeenRun is equal to false. So that tells us that we are now going to run all of the code inside this conditional. Perfect. So let's actually run through what that would look like.

So, next piece of code that we hit is what, Ariel? Inside that condition.

Ariel: So it says if it's false, then hasBeenRun you're going to change the variable value to true.

Phillip: Yeah, we're going to set the variable hasBeenRun to the value of true. So where's the first place that we look for that variable, Ariel?

Ariel: First in local memory.

Phillip: Yup. Do we find it?

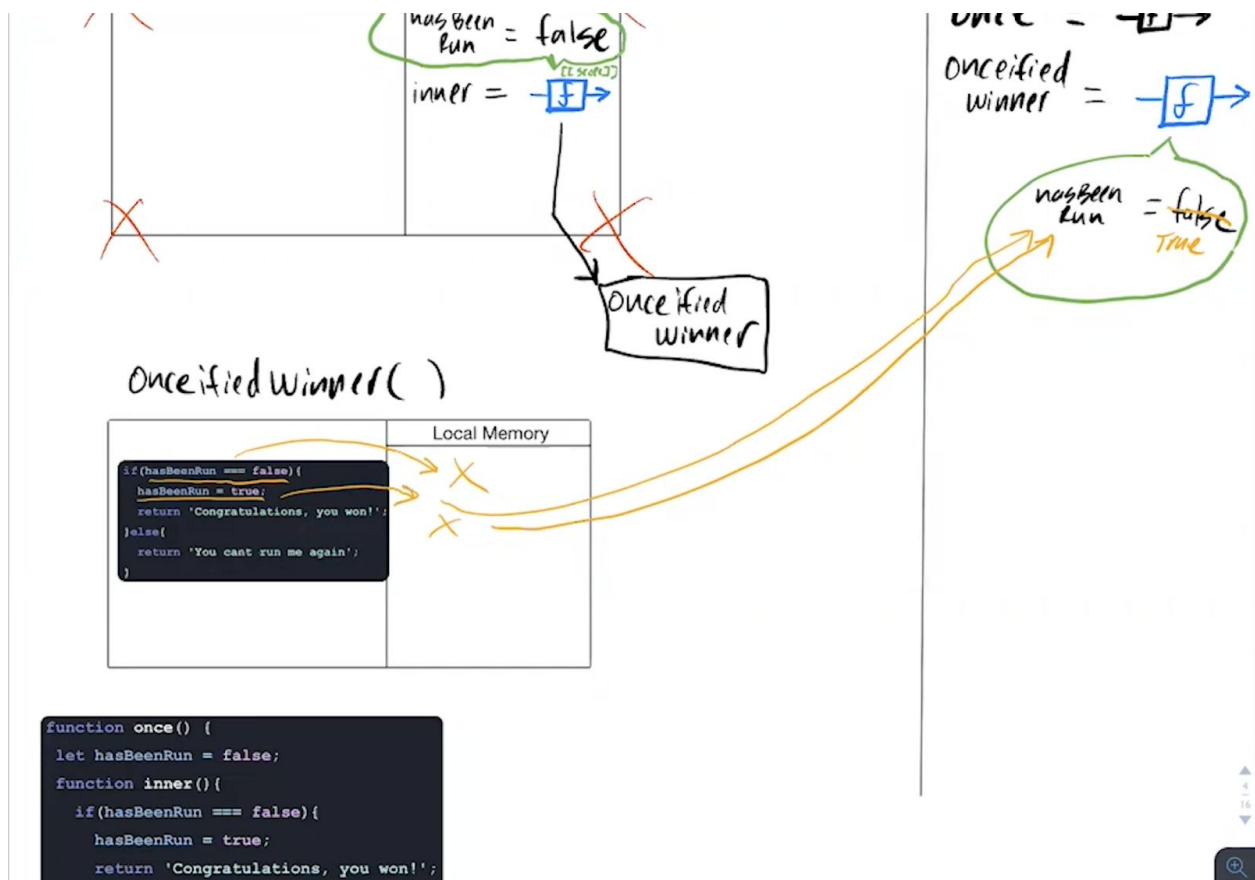
Ariel: Nope.

Phillip: Nope, we don't. Where do we look next?

Ariel: Inside our backpack.

Phillip: Exactly. Inside our backpack, which we then do find it there. And what do we do with the value false?

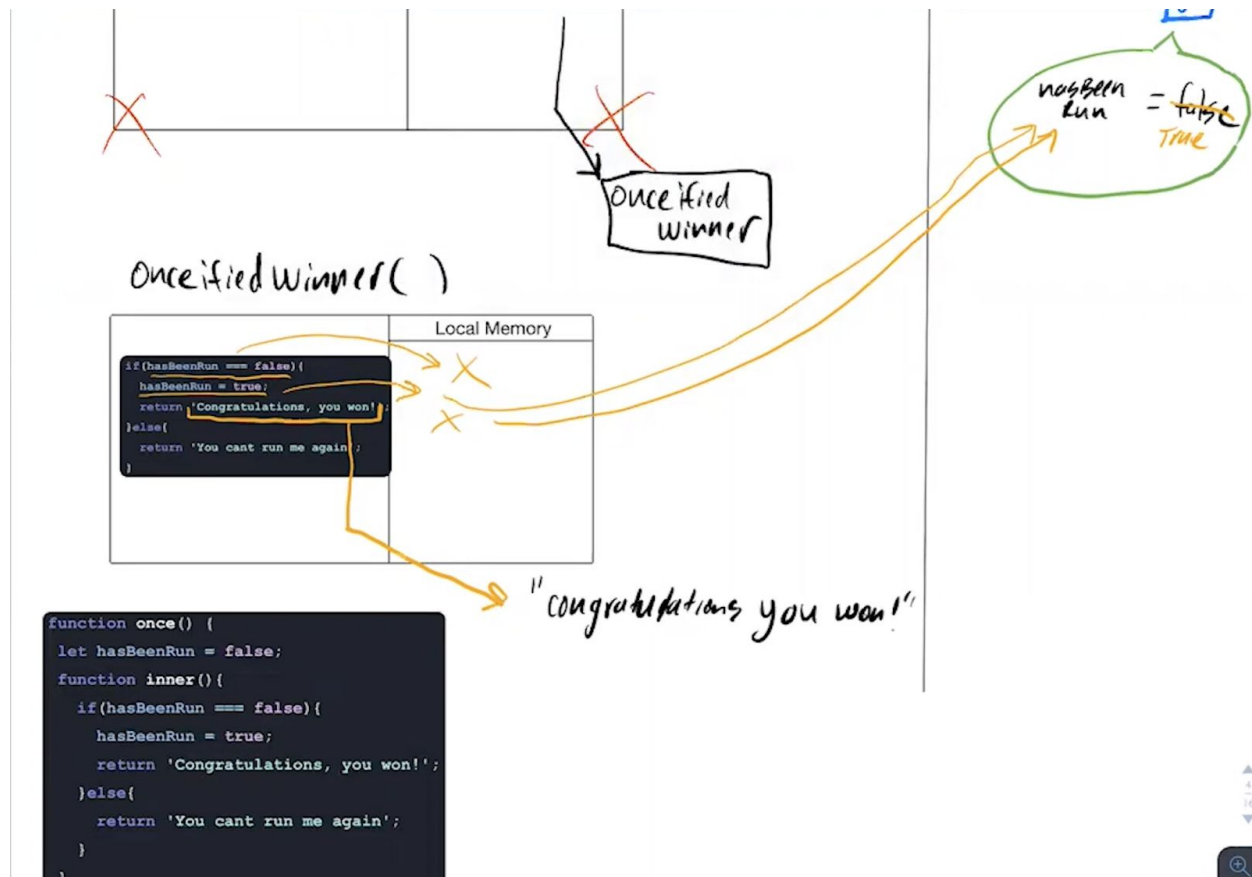
Ariel: We make it true.



Phillip: We change it to true. Perfect. So now, if you guys are kind of thinking ahead of how this is going to play out on our next invocation, we have now set the variable name `hasBeenRun` to true so that later on when we check it, we know that this function has been ran before. So you're kind of teeing this all up for this next call.

But let's continue on. What's the next piece that we actually run into, Ariel?

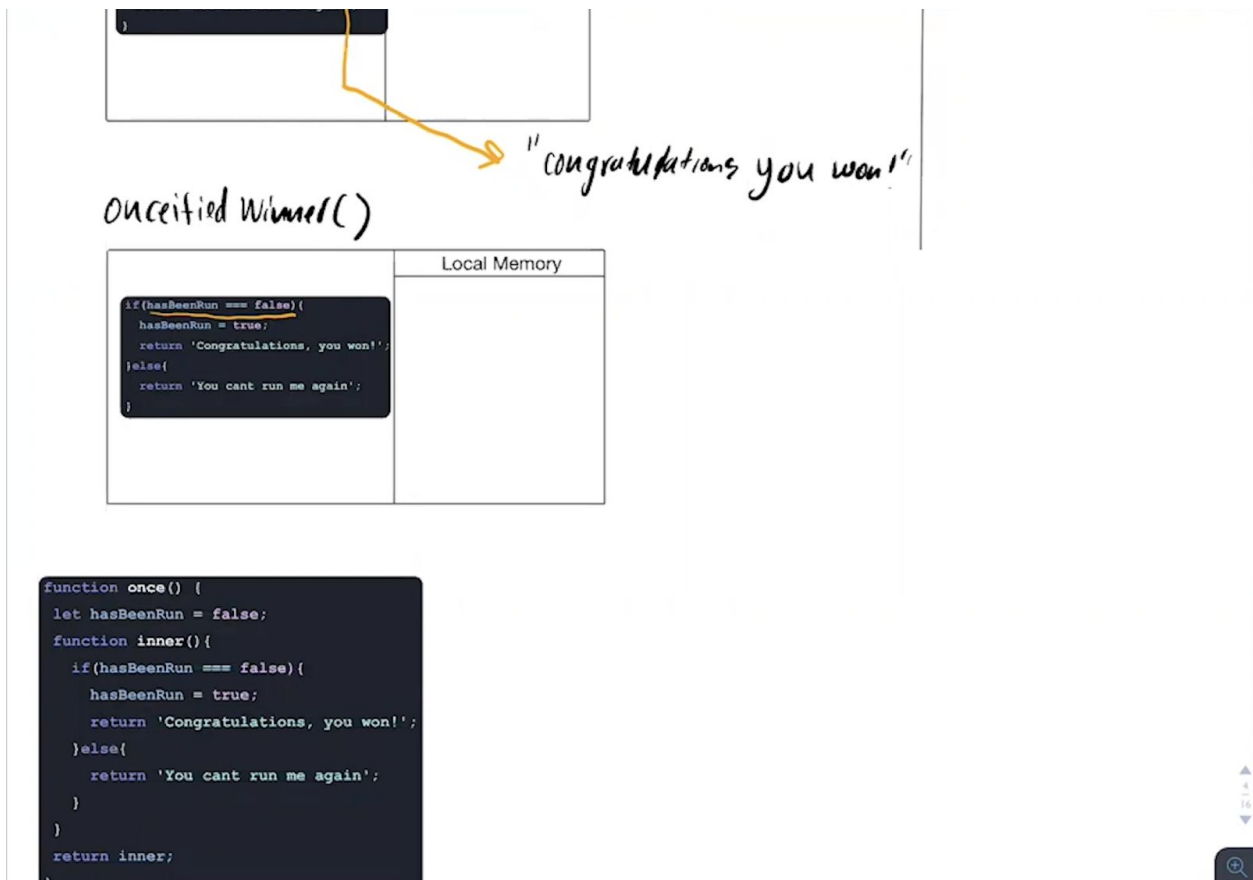
Ariel: Then you're going to just return the string 'Congratulations, you won!'



Phillip: Yeah. So we're going to return this piece right there. So we return that out into our global execution context and we now have in our global execution context the string 'Congratulations, you won!' Good. Now, in reality, we'll talk about this little piece now because now we're kind of thinking "well, what do we do with that string now"? In reality, we wouldn't really just return out a string. We would probably run the function that was associated with all of the stuff that was going to manipulate the dom. All the stuff that was going to make the background color change a bunch of colors and confetti fly out while music plays and all that other stuff. But for simplicity reasons, we're not going to diagram all that piece out. So we're just going to say we returned out the string 'Congratulations, you won!' And now we've ran the functionality of our winner function.

Donald doesn't look impressed. He's like "Okay, big deal. We've ran the function. We've got 'Congratulations, you won!' Awesome". But now the magic comes when we try to run the function a second time. So let's try to run that function a second time and see how this whole piece plays out. So I'm going to take our code that we have right here and I'm going to move it down a bit. Here we go. And you can see that our last piece of code in this piece here. Our last actual line of code is just running the `onceifiedWinner` function a second time. So, let's run that code. `onceifiedWinner`, we're invoking that, which Ariel, means we're going to create what?

Ariel: A new execution context.



Phillip: Exactly. A new execution context. It always seems so weird to do that now with only one person. Alright, we created a new execution context. Awesome. So we're kind of teeing things up to be very, very similar to what they were before. We know inside the `onceifiedWinner` function, we're still running the functionality that was originally referred to as 'inner'. That's what it was originally. So, if we look at our code, we can see that we're, again, just going to run this set of conditionals, this logic right here. We're going to copy that. Let's paste that in, shrink it down so we can see it better. Good, there it is.

Alright, Donald, go ahead and take me through man. What's the first thing that we're going to do- well, actually we kind of teed this up a little bit better here because technically communicating through conditionals can be a little bit difficult sometimes. So we hit the first if statement, which says to JavaScript we're going to be evaluating some sort of condition right now. The condition is checking to see if 'hasBeenRun' is equal to false. Donald, where is the first place that we're going to look for 'hasBeenRun'?

Donald: We're going to look into local memory.

Phillip: Yup, local memory. Do we find it there?

Donald: No you do not.

Phillip: No, we don't. So where's the next place we look?

Donald: It's going to look over in the scope- or the backpack.

Phillip: Yeah, the backpack. The property scope which holds the binding to all of our old data that we had when we had the invocation of once, which we colloquially referred to as the backpack, though we know technically would be the closed over variable environment, or sometimes just referred to as the closure. So we look in there, and do we find it there?

Donald: Yes we do.

Phillip: Yeah we do. And what's the value of it right now?

Donald: It's currently true.

Phillip: It's currently true. So does this condition evaluate to true or false?

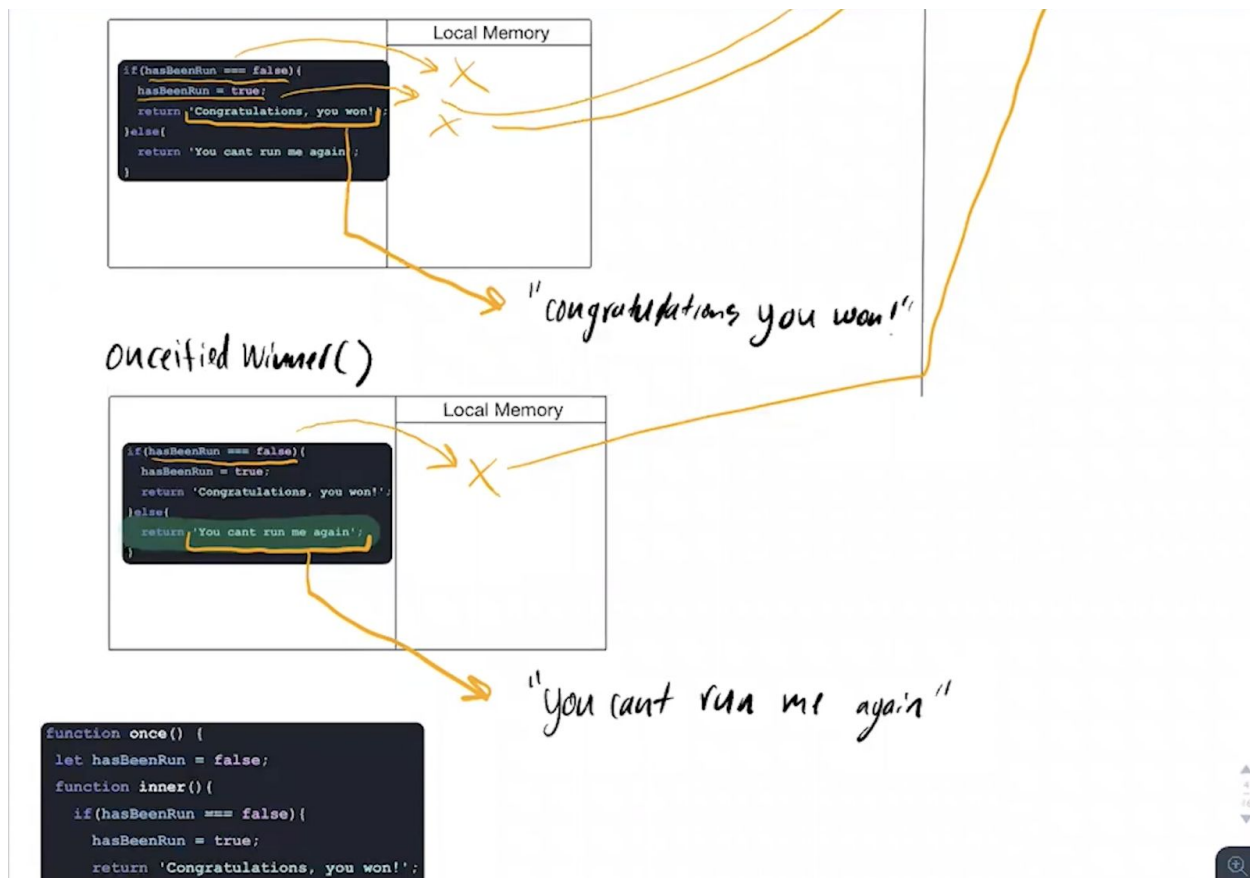
Donald: It would evaluate to false.

Phillip: It would evaluate to false. So now we see the kind of change from the logic that we were running before in our first invocation of this function. So now, we're not going to be running the code associated with the if block. We're now going to be running the code associated with the else block, which says what, Donald?

Donald: It says that you can't run me again.

Phillip: Yeah, it says you can't run me again. And what are we doing with that string?

Donald: We are returning that out.



Phillip: Yeah, we're returning that out. So, we're taking that bit right there and we're returning it out into the global execution context where we have now the string 'You can't run me again'.

Alright. So, this is the process that we would go through to onceify our winner function. The first invocation that we ran to it, it returned out the string 'Congratulations, you won!' Or, in reality, we know that we probably run some sort of function that manipulates the dom at that point. But, you get the gist of it simplistically.

The second invocation, we remembered that we had run this function once before. And we know that we only want ever run the winner function one time in our entire application, so we do something completely different - we return out the string 'You can't run me again'. Or sometimes, who knows? In reality, you probably wouldn't do anything at this point in your code. You would just stop running code then because we don't want to really do anything because we've ran the code more than once. So this is very, very powerful stuff. This can help kind of alleviate a lot of issues in our very complex tic-tac-toe game that Donald was doing.

Problems with the function

- This function works great! When we call our "winner" function we check our Closed Over Variable Environment to see if the function has been called before.
- What if if we also wanted to implement another function in our tic-tac-toe application that only allowed our users to choose their character once

4

So, problems with this function or problems with this approach. Works great. We just saw it play out. We ran it twice and we know that every subsequent call now that we do, that we call, for the onceifiedWinner function, it will always now return out 'Sorry, you've already ran me once. You can't run me again'. But, we are going to run into a bit of an issue with this function. Not that we're going to run into a bit of an issue. We can, we may run into a bit of an issue with this function.

I'll tell you right now, the actual onceify function, the function once that you will find in libraries like `_JS`, which a lot of developers use on a regular basis, it's not built out this way. It's not built out this way because we're going to run into some issues here. Let's say that Donald now has this onceifiedWinner function in his application. Great. Working great. Fantastic.

```

function once() {
  let hasBeenRun = false;
  function inner(){
    if(hasBeenRun === false){
      hasBeenRun = true;
      return 'Congratulations, you won!';
    }else{
      return 'You cant run me again';
    }
  }
  return inner;
}

const oncifiedWinner = once();
oncifiedWinner();
oncifiedWinner();

```

```

function anotherOnce() {
  let hasBeenRun = false;
  function inner(){
    if(hasBeenRun === false){
      hasBeenRun = true;
      return 'You have chosen your "X"s';
    }else{
      return 'You cant run me again';
    }
  }
  return inner;
}

const oncifiedChooseCharacter = once();
oncifiedChooseCharacter();
oncifiedChooseCharacter();

```

But now, Donald is sitting here thinking "Hmm, you know what would also be good? Is if we onceified our function that we run when a player wants to choose which character it is. Like if it's X's or O's." You get into the game, and you have to choose whether or not you want to be X or O. Well you only want to be able to do that function one time too in an entire game. You don't want to be able to change between X's and O's at any given time. So you only want to be able to run that function once as well. So what that code might end up looking like is where you have on the left here we have the function that we just ran, our normal once function. It checks our backpack, it checks our closed over variable environment for the hasBeenRun equals false. If it hasn't been run before, it returns 'Congratulations, you won!'

But now, what would our other function look like if we wanted to onceify our choose character functionality or whatever you want to call it. So, maybe we define it as another once? And then, I mean, it looks almost verbatim, the same thing. The only thing that really changes in these two code snippets is what you're doing when you haven't run the function at all and it's the first time you run it. So, here I just have simplistically 'You've chosen X's'. And then when you run the function a second time, it returns out 'You can't run me again.'

So, realistically, this code is like verbatim. Almost exactly the same. And then let's say that we're building another function that we want to onceify and another function that we want to onceify. We'd have to keep writing, what, once, another once, another other once, so on and so forth. What principal at that point would we be breaking, Carl? If we keep writing that code over and over and over again?

Carl: You would be breaking the DRY Principle.

Phillip: Yeah, you would be breaking the DRY Principle. The "Don't Repeat Yourself" Principle. Chances are, if you are ever repeating yourself over and over and over again in your JavaScript code, there is something that you can be doing better.

Lecture: 3 Rebuilding and Diagramming the Once Function in JavaScript

The “Real” Once

```
function once(func) {
  let hasBeenRun = false;
  function inner() {
    if( hasBeenRun === false ){
      hasBeenRun = true;
      const value = func();
      return value;
    }else{
      return 'You cant run me again';
    }
  }
  return inner;
}

function winner () {
  return 'Congratulations, you won!'
}

const onceifiedWinner = once(winner);
onceifiedWinner();
onceifiedWinner();
```

Phillip: This is where I like to talk about what I usually refer to as the "real" once function. Essentially, how do we make this once function reusable on multiple, multiple, multiple function definitions. Essentially this idea of being able to have this once function and then pass a function into it and then have it return out ultimately a onceified version of that original function. That's the approach that we're going to take on this second problem set. To see really on a professional level, how would this once function actually be built out and we're going to see the benefits as we go through.

Okay. We're going to have some slightly more complicated code going on here but ultimately it's going to be in our benefit because it's going to make this once function reusable. So let's start going through this code line by line. Carl, I'm going to have you kick us off here. What is the first thing- can you see the code Carl? Is it big enough? Okay. Cool. What is the first thing that we're going to be doing in this bit of code.

Carl: First thing you're doing is declaring a function called once.

Phillip: Perfect. Where are we storing that at?


Carl: It will be stored in global memory.

```
function once(func){
  let hasBeenRun = false;
  function inner(){
    if( hasBeenRun === false ){
      hasBeenRun = true;
      const value = func();
      return value;
    }else{
      return 'You cant run me again';
    }
  }
  return inner;
}

function winner (){
  return 'Congratulations, you won!'
}

const onceifiedWinner = once(winner);
```

Global Memory

once = 

- Phillip: Perfect. There we go. We have our once function stored in global memory. I know it's a little bit hard to see a little bit on this because the spacing isn't completely optimal, but can you see, what is the next piece that we're actually going to do now? What's the next bit of code that we're going to run into?
- Carl: The next one is we're going to declare a constant variable and labeled onceifiedWinner and it will be-
- Phillip: You're skipping one piece, Carl. I know it's hard to see it, but can you see the spacing?
- Carl: Oh. Yeah. Yeah. You're declaring another function.
- Phillip: Yeah. Called what?
- Carl: Yeah, winner. You're declaring another function called winner.
- Phillip: Good. We're declaring our other function called winner. We're actually going to color code this one as well. This will be our blue function. Good. Awesome. And then we get to that part that you were just getting ready to hit, Carl. What's the next piece that we have on that?

Carl: Right. And after that I'm going to declare a variable `onceifiedWinner` and it will be assigned the evaluation of `once` with the argument `winner`.


Phillip: Perfect. Exactly. So right now it's left in this state of undefined. We don't what it is. We got to go- first. But absolutely. You can already see where we've changed the process of how this code is running because now instead of just calling `once` by itself, if you remember, if we go back up to our original code on our very first implementation of `once`, the `once` function wasn't taking in any type of arguments. It was just being ran by itself. But now, we're generalizing our `once` function using parameters and arguments and we're sending in now a function definition of `winner`.

So the code that we're actually running here is `onceifiedWinner` is equal to the evaluated result of running `once` and passing in the function definition that is `winner`. So we're going to pass in our blue function. We know that reality of this is the original function definition for `winner`.

Now to be perfectly clear, though, you see that I wrote `winner` right below that function, but in reality we don't pass in that label at all. We just pass in the function definition. We're going to label it something completely different inside the execution of `once`. Actually you know what, we have the blue there. Let's get rid of this `winner` label. We don't even have it at all. So we're passing in the function definition of `winner` here as the same definition that we have right there. Cool.

Now we are running a new function. Donald, this is your turn man. Everybody else has gone so far. We're running a new function which means we're going to create what, Donald?


Donald: A new execution context.


onceHied winner = once ()

Local Memory

Add A Caption...

Global Memory

once = 

Winner = 

onceHied winner = _ _ _ _

```

function once(func){
  let hasBeenRun = false;
  function inner(){
    if( hasBeenRun === false ){
      hasBeenRun = true;
      const value = func();
      return value;
    }else{
      return 'You cant run me again';
    }
  }
  return inner;
}

function winner (){
  return 'Congratulations, you won!'
}

```

- Phillip: Yeah. A new execution context. It's already starting to fade and it sounds natural that one person is doing this. In a minute we're going to get that changed up. We're all going to say it just for the sheer romanticism of it. We're now running our once function and, Donald, take me through man, what's the first thing that we're going to do inside this call to once?
- Donald: Got it. We're going to declare a variable with the label hasBeenRun in local memory and assign it a value for, wait-
- Phillip: Ah. That's not the first piece. There's one other piece.
- Donald: Wait. Hold on. Are we running once, correct?
- Phillip: Yeah. We're running once. There's one thing we need to do.
- Donald: Okay. Sorry. Alright. We're going to create a local variable with the parameter func. F-U-N-C.
- Phillip: Yeah.
- Donald: In local memory.

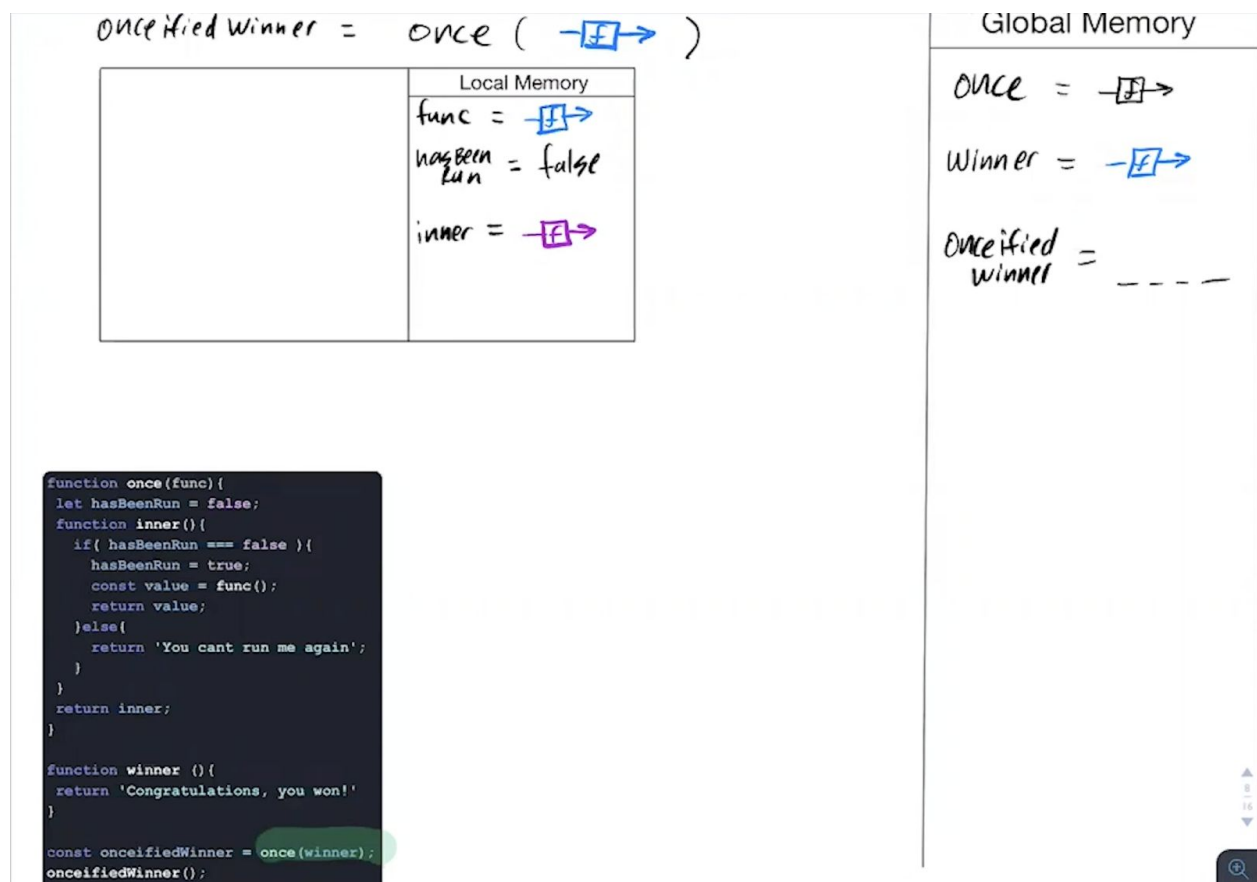
Phillip: It's always a little bit hard on how you communicate that piece. Do we call it a variable? Do we not call it a variable? Technically it has the same functionality of a variable, but technically it's not a variable so we just say we're going to set a parameter `func` to the value of the function definition that we passed in. So yes. You're absolutely right, so we have `func` and we're giving it the value, we're setting the value of the function definition that we passed in. It was originally referred to as `winner`, but we know it as our blue function here. There it is. Good. Now, Donald, what's the next piece that we run into? Or the next piece that we get in our code?

Donald: The next piece we're going to declare a variable in local memory with the label `hasBeenRun`.

Phillip: Good. `hasBeenRun`. And it's initiated to the value of `false`. Good. Alright. And the next piece, Donald. Keep us going.

Donald: Cool. We're now going to declare a function in local memory with the label `inner`.

Phillip: Good. There it is and we're going to color code this one again. We're going to bring in our lovely purple function. There it is. And Donald, don't forget about our piece that JavaScript automatically does whenever we create a function, but it becomes interesting obviously right here at this point, but what is the other piece that JavaScript does for us automatically?



Donald: It automatically creates the closed over variable which will contain the local memory of the outer function.

Phillip: Perfect. Perfect. It creates that closed over variable environment which is going to make a binding to all of this data that's in here which includes our hasBeenRun variable and includes our func function. Perfect. What's the next thing? What's the last piece that we hit inside this call to once?

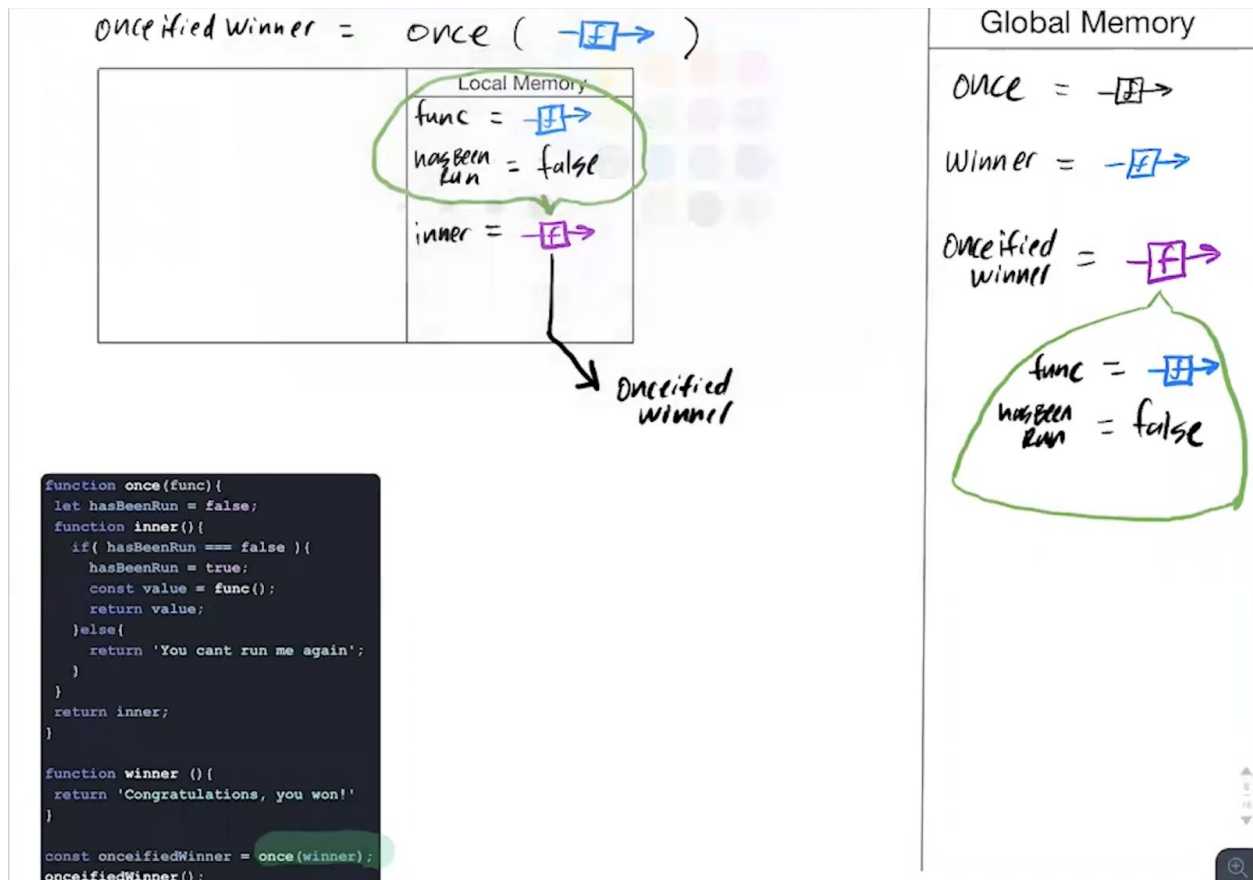
Donald: Yep. We're going to return the definition of inner.

Phillip: Perfect. Not the label, just the definition that's associated with that label inner. So we return that function out. There we go. And what's the new label that catches in the global execution context, Donald? Do you see it?

Donald: onceifiedWinner.

Phillip: Perfect. onceifiedWinner. Now, in our global memory, onceifiedWinner's no longer uninitialized. It is now the function definition, got a lot to write here, our function definition which is our purple function, which was the original value that was associated with the inner label. There it is and it also has our closed over variable environment. Aka our backpack. Aka our closure.

I'm going to make this a little bit bigger here because we got a little bit more to add. Inside that we have our func variable that its value is our blue function. Good. And then we have, let's switch that back, then we have our inner, I'm sorry. Then we have our hasBeenRun. hasBeenRun variable which is set to the value of false. Let's close up our backpack here. It's a very oddly shaped closed over variable environment, but it works nonetheless.



Alright. Now comes the more interesting piece. Some of this stuff now we can infer what's going to happen next. But let's run now our `onceifiedWinner` function because that's the next logical piece and that's the next piece of code that we actually have down here. `onceifiedWinner`. Let's go ahead and run. Let's move this down. I think that's far enough and let's run our code of...oh, switch that back to black. Let's run our code to `onceifiedWinner`. And we're invoking that. Good.

Which creates a new execution context. There we go. There's our execution context. Now remind me just really quickly, Carl, when we're running `onceifiedWinner`, the function definition that's associated with `onceifiedWinner`, what was that function definition originally associated with? What label was it originally associated with? I'll give you a hint. Look at the color coding that we have on this.

Carl: It was once associated with inner.

Phillip: Yeah. It was once associated with inner. So when we look at our code down here, of course we know we're not looking back up into an old execution context, but we do inside of our code just so that we can see what code actually is so that we can technically communicate into it. We know that it is this code that's associated with inner, which is right here. All of this code here is all going to be ran again right now. So let's go ahead and grab all of that. Copy that. Paste it into onceifiedWinner. Let's make it a little bit bigger. There we go.

Now, Carl, we're running the code inside onceifiedWinner so the first piece of code that JavaScript gets to is this if condition. It says "if, alright. I know what that is. We're going to be evaluating some sort of condition and the condition is if hasBeenRun is set equal to false." It doesn't know what hasBeenRun is. It goes, "what the hell is hasBeenRun?" So where's the first place that it looks, Carl?

Carl: It's going to first look in the local memory.

Phillip: Does it find it there?

Carl: It does not.

Phillip: It does not. So where's the next place it looks?

Carl: Next it will look into the backpack that is in global memory.

Phillip: Perfect. It'll look inside the backpack. Which does it find hasBeenRun there?

Carl: It does.

Phillip: It does. Perfect. And what is the value of hasBeenRun?

Carl: It is false.

Phillip: False. So does this condition evaluate to true or false?

Carl: It's true.

Phillip: It evaluates to true. So now we know that we're going to be running all of the code associated with the if block which is right there. Which is slightly different from the code that we ran in our first implementation of once. So now we're going to see really where the magic happens of the two different versions of this. Alright.

The first bit of code that we run is we set the hasBeenRun variable now equal to true because now we're actually going to be running it. We first look for that in local memory. We don't find it. So JavaScript then looks inside of our closure where it does find it and it finds that hasBeenRun variable and then sees that it's false and it changes it to true. Perfect.

Now, the next piece. Carl, technically communicate for me what is actually happening in the next piece of code.

Carl: In the next piece we're declaring a variable value and assigning it to what looks like an anonymous function.

Phillip: Well, yeah. It's-

Carl: An evaluation of-

Phillip: Not an anonymous function because you would see it actually say function. But let's start with the first piece. We're declaring a variable labeled value. Where do we declare that at? Or where do we save that at, Carl?

Carl: That will be saved in the local memory.

Phillip: Good. There it is right there and it's equal to the evaluated results of running func. So maybe now even we're looking at this code and we're saying what is func? I don't remember declaring that at all. Let's think about this how JavaScript would think about it. When it sees something that it doesn't know, when it's not a key word, JavaScript is going to go "func, huh? What is that?" Where's the first place it's going to look, Ariel?

Ariel: It's going to look in local memory first.

Phillip: It's going to look in local memory. Good. Does it find it in local memory, Ariel?

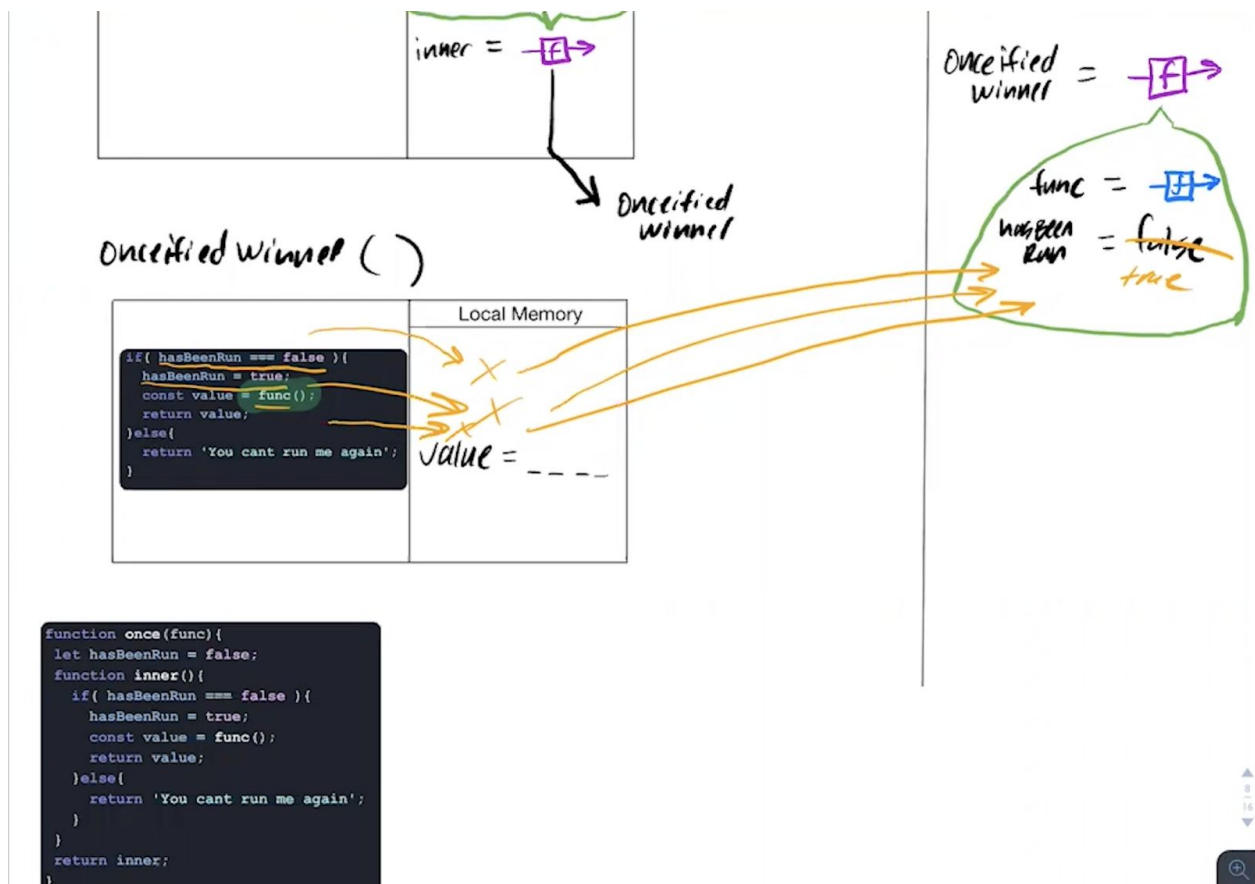
Ariel: No.

Phillip: No it doesn't. So where does it look next?

Ariel: In the closed over environment.

Phillip: Yeah. The closed over variable environment. So it looks here. Does it find it in there?

Ariel: Yes.



Phillip: Yes it does and then we start going, "Oh yeah. I forgot. We declared that function way long ago when we were first inside the execution of once." So we would look at that function and we'd say ok, we're going to run that function right here. Now that function was originally, way back when we first started this application, when we first started this running of this whole bit of code, what was that function itself, Donald?

You had your name as Don on here so I keep wanting to call you Don. Donald.

Donald: That's fine, too. That's fine as well.

Phillip: What was the label that was associated with that function?

Donald: Yeah. I believe the original label was winner.

Phillip: Was winner. Exactly. We can see that right here in our global memory. We then at some point passed that into the once function and it trickled down, but now it's in our closed over variable environment. So we have access to it. We run the functionality of winner.

Now we're not going to actually open up a new execution. We would in reality JavaScript will open up a new execution context and all those pieces, but because we don't have a whole lot of room to work with on this whiteboard and that function is

very, very simple, let's just take a look at it really quickly. The function `winner` simply returns out the string 'Congratulations, you won!' So that value is then being returned out into our value variable in local memory. So after we run that value is going to be set equal to the string 'Congratulations, you won!'

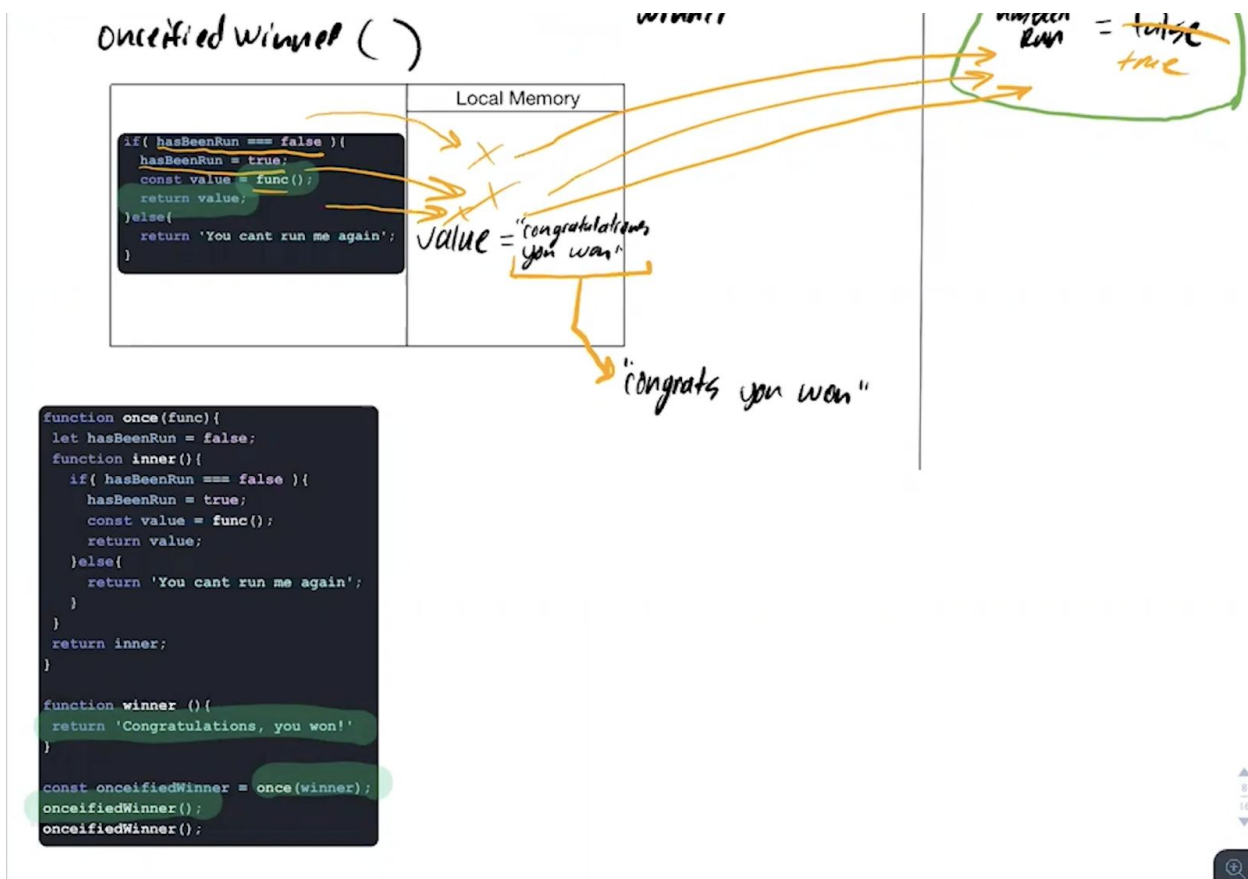
Good. What's our final piece that we have inside this called, `onceifiedWinner`? I see you switched your name.

Donald: Yeah. The final piece we're returning that value-

Phillip: Returning value.

Donald: The returning value to global.

Phillip: We're returning the value of value to the execution context. Yeah. Absolutely. So we take this whole piece right here. Actually, let's change that to orange so it sticks out. We take this whole piece here. This whole value. We return that out into the global execution context where we now have in our global, we're going to say 'Congratulations, you won!' Good. We now have our string in the global execution context. Again, in reality we would run a function here that would then make the dom manipulate and so on and so forth, but simplistically, we now have the string 'Congratulations, you won!'

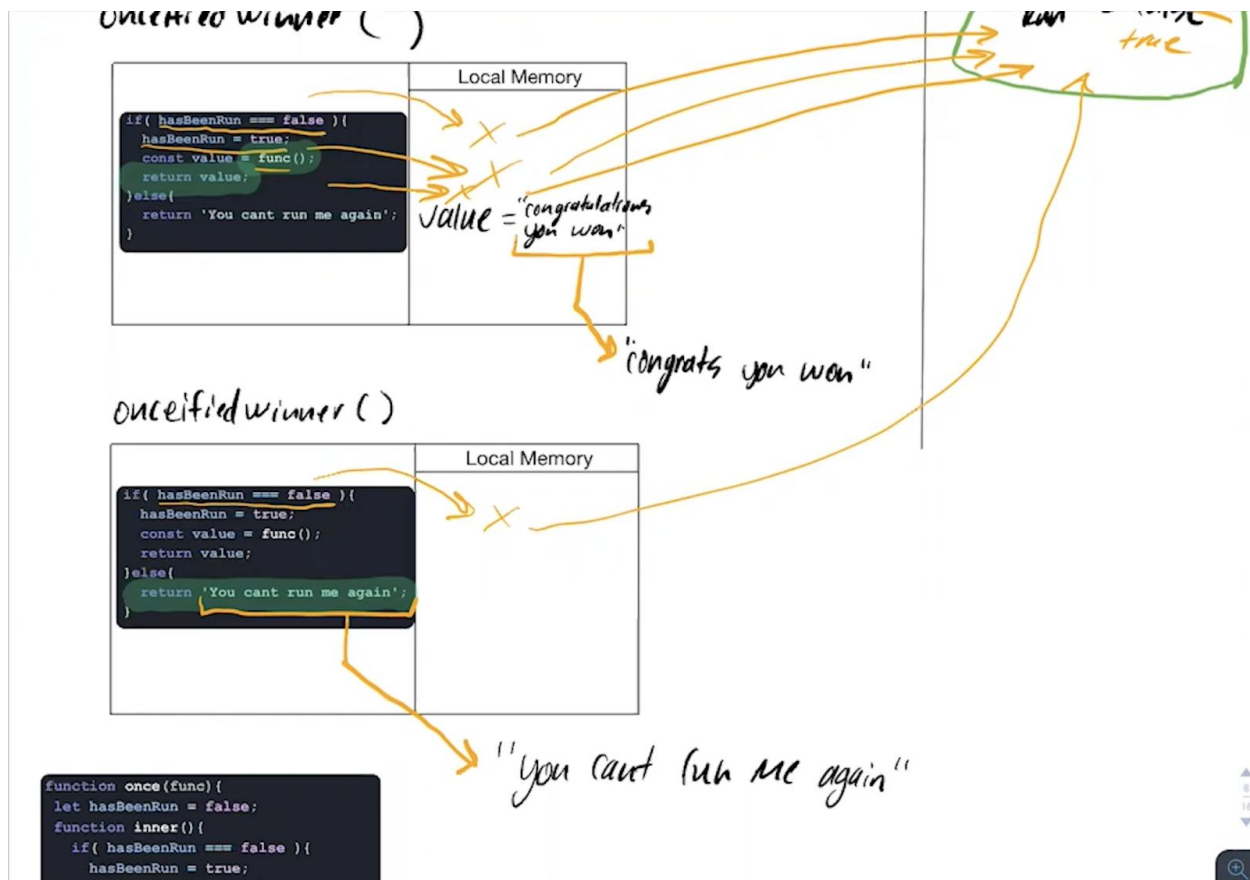


Now, you're thinking ok, well, why is that any different? Well, really it's not. We weren't trying to get different functionality. We were trying to generalize our once function so it can be applied across any number of functions. So real quickly, if we really want to see the magic of this and we want to see how this changes, we need to run the `onceifiedWinner` function one more time. That's what we have down here in the bottom of our code. So let's quickly run through that last piece and then we'll wrap up and talk about why is that we even care about this because it looked like we basically did the same stuff over again.

Let's move this down. Now let's run our function one more time. `onceifiedWinner`. We're running that code. You know what, let's do this now. I feel like this is a good time. We're coming to the last execution. Really, we're coming to the last big execution so everybody unmute yourself, we're calling a new function `onceifiedWinner`, which means we're going to create a new?

All: Execution context.

Phillip: Ah. That's why we don't do it with only three people. Really awkward. Ok. But as awkward as it was, it is absolutely correct. We create a new execution context which we then continue to run the same code that was associated with `onceifiedWinner`. Which originally was our function definition of `inner`. So we're going to take our code again. Let's go ahead and grab all of this code here. Copy. We paste that in right there. Make a little bit smaller. There we go. Alright. And then we run through that code again.



I'll speed this process up just a little bit because we've gone through this now a few times. So the next thing we do is JavaScript hits the line of code that says if hasBeenRun equals false. So we hit the key word if which says we're going to evaluate a conditional. It says is hasBeenRun equal to the value of false? First place we look? We look in local memory. We don't find it there so JavaScript knows to then look upwards into our closed over variable environment, our backpack, where it does find the value, or the variable hasBeenRun and it's initialized to the value of true. So that conditional evaluates to false which means we are going to run all of the code associated with the else block. The else block says return the string. You can't run me again. So that string gets returned out into the global execution context where we now have the string 'You can't run me again.' Alright. That is the functionality of our real once function.

Problems with the function

- Nothing! Pass in a function and it returns a "onceified" version of that function
- Maybe it's not impressive enough?
- Maybe too simple for you?

Well lets make things more interesting then...

7

Now remember, because I'm looking at Carl's face now, he is not enthused with that either. He's like Phillip we just ran the same basic code over again. What was the point of doing that for the last 45 minutes when we had that same functionality before? Well, it's a fantastic question. The real benefit of this comes in the fact that we generalized our once function. We generalized our once function so that we can pass in any number of functions. We can pass in a winner function and then it will turn out a onceified version of that winner function. We can pass in a choose your X or O character function. Or whatever we call it. We pass that in and it returns out a onceified version of that function.

Any function we give it, we pass that in, we ultimately wrap it in this inner function and using closures we pass out a onceified version of that function. This is extremely, extremely powerful in your production scale applications. Not just your tic tac toe application.

Lecture: 4 Q&A

Phillip: Let's have thumbs on this concept. All the pieces we've covered so far. The idea of using closures to onceify our function and the idea of generalizing from our first implementation of once to our second implementation, our real once. Let's have any questions that we have on any of the differences on any of those pieces.

[Thumbs up] I'm good, I'm ready to move on to our next advanced problem set.
[Thumbs sideways] I have some clarifying questions, which is totally ok. [Thumbs down] I have no idea even how to put it into a question. Carl's got one; Carl why don't you go ahead and tell us your question man.

Carl: So just for clarity, if we could go back up to the whole once-

Phillip: The whole once- do you want to see it here or we could just go up there to look at it?

The "Real" Once

```
function once(func) {
  let hasBeenRun = false;
  function inner() {
    if( hasBeenRun === false ){
      hasBeenRun = true;
      const value = func();
      return value;
    }else{
      return 'You cant run me again';
    }
  }
  return inner;
}

function winner () {
  return 'Congratulations, you won!'
}

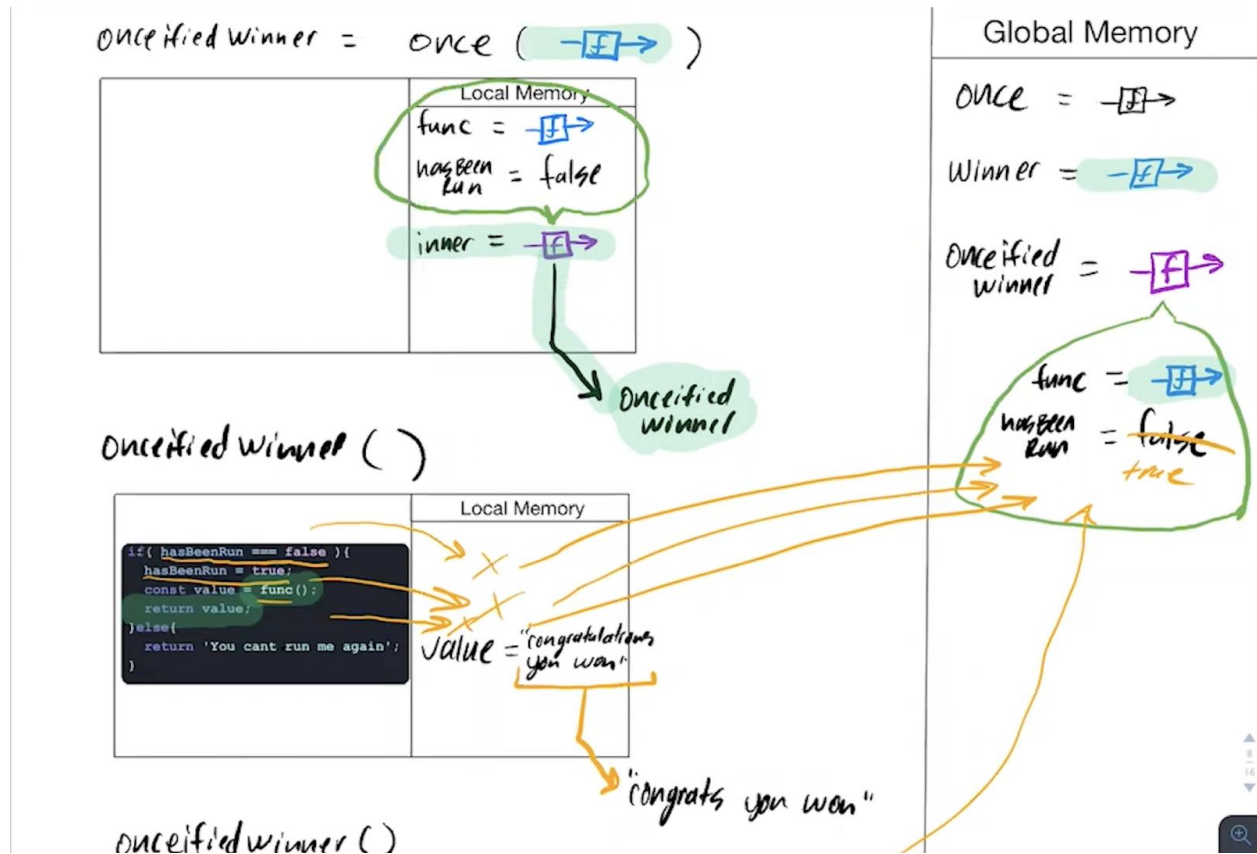
const onceifiedWinner = once(winner);
onceifiedWinner();
onceifiedWinner();
```

Carl: Yes, that's fine. So, I remember we were in the process of going through line by line, when we hit- So we called once, and we started to go through, and hasBeenRun is initialized to false, so we hit the function inner, we immediately jump down, or rather, we only defined it and then we didn't really go into the body of it. We jumped right down to return inner.

Phillip: Correct.

Carl: So just for clarity, when is function inner called?

Phillip: When is the function inner called? Let's look back down to our actual code here.

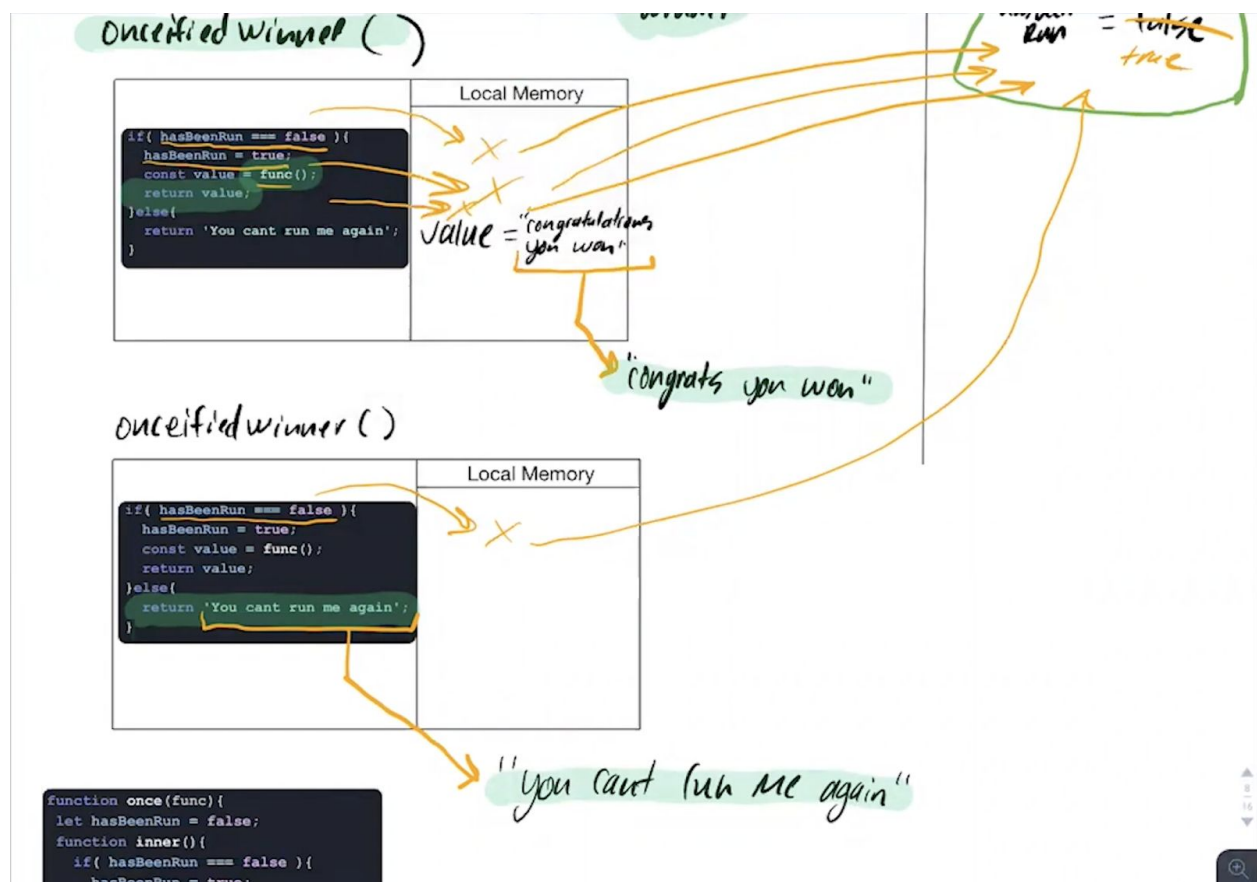


So, when we defined our function inner, and let's see if we can clean any of this up, well we defined our function inner right here. We defined that inside of our call to `once`, so that's similar to this piece that we have right here, ok. So we defined that there `once` and we color coded it just to make it a little bit easier to follow up where that function definition was. We returned that value out and saved it inside the new label: `onceifiedWinner`. So when did we actually call the function definition that was associated originally with the label `inner`? Well we called it when we started running our calls `onceifiedWinner`. Because now you can see that purple function definition right there in global memory, was the same function definition that we had inside of our call to `once`, which was our inner function. So if we look back up at the code, you're asking where did we actually run the functionality of `inner`? Well we were running that every single time we called `onceifiedWinner`. We returned it out and saved it under its new label.

Carl: Because `once` is the return of `inner`, which in itself is a function. So, what if, can we just, to submit this- If we never called `onceifiedWinner`, you'd never see the evaluation of the inner function, or rather, if we never called `onceifiedWinner`, you'd never see the running or the invocation of `inner`.

Phillip: If you never- I'm sorry, say it one more time.

- Carl: So like onceifiedWinner, since it is assigned to the evaluation of once, if we never invoked it, we'd never see the functionality of inner.
- Phillip: Correct, absolutely correct. If we never ran the functionality of onceifiedWinner, we would never actually run the code associated in there. Absolutely.
- Carl: Right.
- Phillip: Good question, man. Donald, you had a question as well.
- Donald: Yeah, I have two clarifying questions. If you were to scroll down a little bit when you're returning the strings to global.



- Phillip: Sure.
- Donald: Since you're not storing that anywhere, what does JavaScript do with that value? Does it just go into the ether and the garbage collector just cleans that out, or what happens to that return value?
- Phillip: Yeah, that's a great question, and that's why I usually like to specifically talk about it in this example because in real life we wouldn't do this, because in reality we are- I like the

way you even described it, we are just returning it out into the ether. I actually use that same term as well which is weird. So yeah, this 'Congratulations, you won!' or like the 'You can't run me again'. That thing just gets returned out into the ether, but JavaScript never grabs it and stores it anywhere.

Donald: Alright, I just have one more. So my last question is, would you say the main difference between the first-

Phillip: Hold on one second. So let's actually run through that one more time. Just that explanation.

Donald: Sure.

Phillip: So great question, Donald. What actually happens to those strings when we return them out? Exactly like what you just said. I love the way that you phrased it. It just kind of gets returned out into the ether, nothing really happens to it. JavaScript doesn't actually catch it and assign it to any spot in memory, so yeah it just kind of goes away, we don't do anything with it. If we wanted to, of course, we could save it into a variable and then utilize it somewhere else. But in this implementation right now while it's still a little bit silly we don't do anything with it right now. You had a follow up question?

Donald: Yeah, I had another question. Would it be safe to say the main difference between the onceified function in the first example we went through and this example is sort of we're making it more sort of like dynamic or more modular?

Phillip: Yes absolutely, that's exactly what we're doing. We are making it more dynamic, we're making it more general, we are modularizing it so that we can utilize it in other pieces of our code anywhere we want. We just pass it a function and it outputs a onceified version of that function. Yeah. Absolutely.

Donald: Right.

Problems with the function

- Nothing! Pass in a function and it returns a "onceified" version of that function
- Maybe it's not impressive enough?
- Maybe too simple for you?

Well lets make things more interesting then...

7

Phillip: Awesome, guys. So now I want to move on to some more in depth pieces. Right now we figured out how to onceify our function, how to set-up boundaries as to how many times we can call a particular function. But I'm looking on Ariel's face and again you guys are a hard crowd to please. Ariel, still even with generalized onceified function, she's still not convinced. She's like "I still don't really think I would use this that much Phillip." Well let's talk about a more in depth version, or not a more in depth version, rather a more in depth practice using closures to do something that's a little bit more efficient, a little bit more powerful I would say in our applications. And this is why I want to get into the memoize function.

Alright so, we just explored the possibilities of using closure via our onceify method. Both our very simplistic implementation of it and then kind of our real implementation of it. And I'm still looking at Ariel's face, and she's not impressed. She's sitting here thinking, "Phillip, I'm never going to use that function in my application, for sure not. I'm not going to be building out a to do list app." I'm sorry, our example was a tic-tac-toe app. "I'm not going to be building out a tic-tac-toe application. So you need to impress me further."