

Lecture: 1 Introduction to Closure

JavaScript the Hard Parts

Closure

But first - In JSHP we start with a set of fundamental mental models

- 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
- Build iterators and maintain state in an asynchronous world

8

Will: In this session, we are going to cover the most powerful part of JavaScript: closure. The most esoteric part, the part that even the most experienced developers do not have a grasp on under the hood, and therefore can't use closure to all of its effect. We're going to discover that if we understand closure, we can build out helper functions, professional-level functions that help us write code day to day, like once, like memoize. We can then also build out the module pattern, a way of organizing our code that ensures it's much more maintainable. No one's overwriting my global variables, and yet I can persist, hold on to data for the whole life span of my application. All going to use closure. It's going to allow us to understand iterators, asynchronous JavaScript, but more than that, for me understanding closure and the pieces that go into it — execution contexts, lexical or static scope, the returning of a function from a function, and we're going to also sort of touch on the heap — understanding closure is understanding JavaScript. And honesty, borderline, is understanding programming to some extent.

I think closure is a little window into everything in JavaScript, and a lot of things in mature programming. We are going to go to all of it in the Hard Parts style, where I'm going to be calling on each of you. And for our online audience who are going to see this later on, they are going to hear each of you verbalize a code as we diagram it.


But it's all going to begin, all that specification to come, is all going to begin with pieces are going to feel quite trite, quite basic, too easy. And for our online long term audience, are going to think, hold on, I know how JavaScript Basic's codes runs. But know, that if we get this stuff down, laying ahead, is a deep dive into things like this — hear this phrase, persistent lexical or static scope reference data, closed over variable

environments. These are the advanced concepts to come, that to me only become salient if you've understood the foundational pieces that we're going to see here. So for online audience later, even if you're feeling the early stuff is, "I got it", you'll benefit far more from the later stuff if you get a clear foundation in the basics. Which are honestly the basics of JavaScript.

Lecture: 2 Principles of JavaScript


JavaScript principles

We will diagram each line of code to understand how it works under-the-hood




Functions

Code we save ('define') and can use (call/invoke/execute/run) later on by using the function's name/label with parentheses




Thread of execution

JavaScript goes through the code (globally or in a function) line by line and does whatever the line of code says to do



Memory

A store of data ('Variable environment') where anything defined in the function is stored



```
let num = 3;

function multiplyBy2 (inputNumber){
  const result = inputNumber*2;
  return result;
}

const output = multiplyBy2(num);
const newOutput = multiplyBy2(10);
```

Will: Let's see those basics of JavaScript. The underlying principles of how JavaScript runs, here they are, we're going to start off by walking through our code as the JavaScript interpreter, which is the engine that goes through the code line by line and says "what did the developer tell me to do, oh told me to do that". As we go through that code line by line, two things happen. One, we tend to save data, that means save stuff like here num is 3 to memory. And two, we tend to run code on that data. And we often save code and we don't run it immediately, we use it later on by referring back to it, those are known as functions.

Here we go, let's walk through this first portion of code here together. Let's walk through this first portion of code together, and see exactly how JavaScript runs, even this most basic of code. Alright. Here it is. There's our most basic code declaring num 3, declaring a function multiplyBy2, calling it twice. Let's do it starting with Kate. Kate, take it away. What are we doing here in the very first line of our code?

Kate: In the global memory, we're going to declare a new variable with the label num and we're going to assign it the value 3.

Will: Yeah. Memory is the fancy word in JavaScript for where we store our data as we go. Line one, there it was, num is now 3. For the rest of our application, if you see num, always 3. Excellent. Thank you to Kate. Let's now turn to Natalie, line two, Natalie, what do we do?

Natalie: We are in global memory. We are declaring a function multiplyBy2 and assigning it the value of the entire function definition.

Will: Yeah! In JavaScript, functions are not dissimilar to numbers. Num is the label, 3 is the value. Value - label. Label - value. It's really weird to say that function is a value, it means it's something that can be stored, and honestly, by the way, we'll see in a moment, can be given multiple labels. Interesting. Okay - excellent, there it is, there it is, there it is. Okay Elaina. Num is 3, multiplyBy2 is a function. Elaina, what might you think is the third line of JavaScript here? What Elaina might you think is the third line of JavaScript?

Elaina: In the next line of code, we are creating a constant output in the global memory.

Will: You are spot on. But what might, intuitively, folk think after you declared? What might people think is the third line here?

Elaina: Probably const results.

Will: Why is it not Elaina?

Elaina: Because we did not invoke the function multiplyBy2 yet.

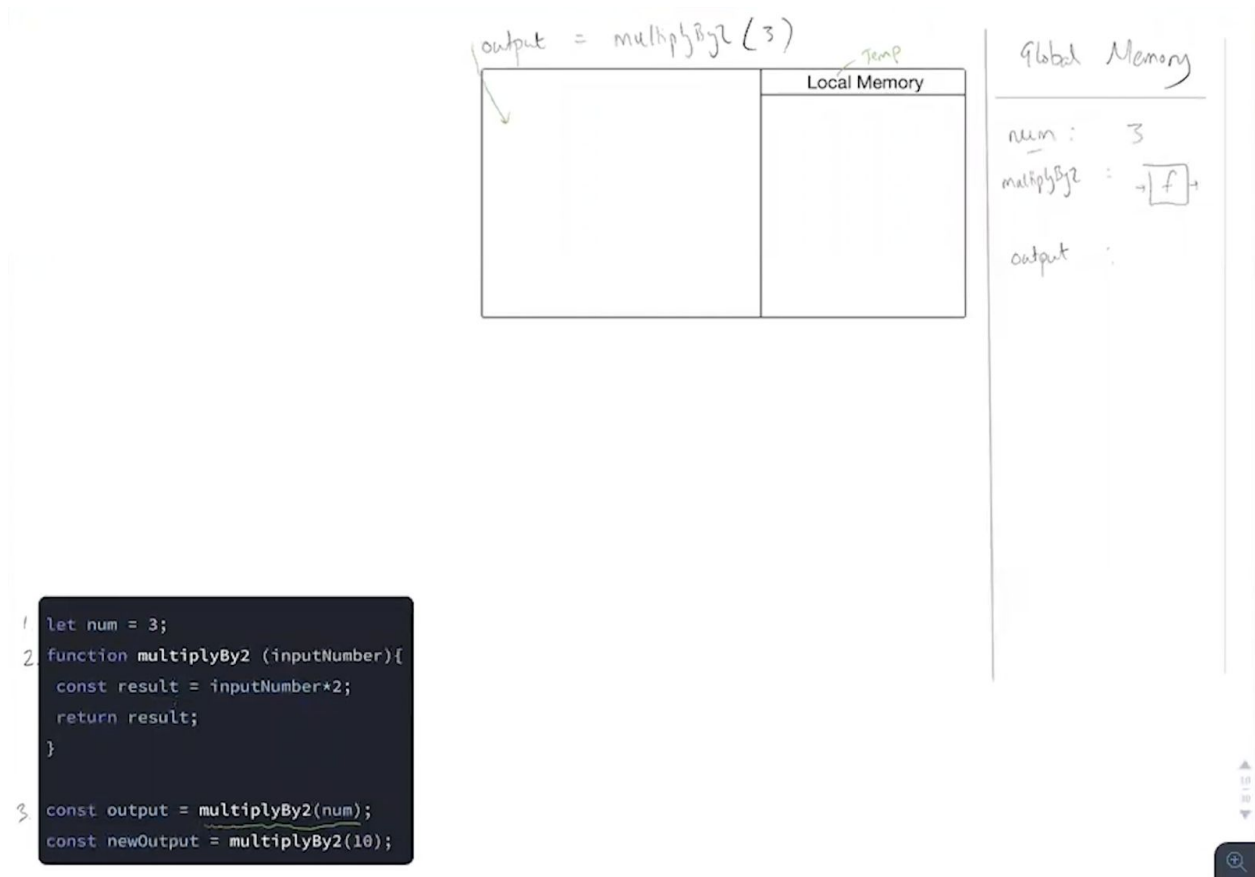
Will: Exactly. When JavaScript sees function multiplyBy2, it grabs all the following code between the curly braces and saves it all in one code, and never goes inside, never introspects, it never goes "oh, what's it telling me to do here". It ignores it, throws it all to be saved in memory, there it is, and moves the hell on. To what line Elaina, as you said correct line three is?

Elaina: We declared const output in the global memory and for this very second, it's initialized, but then we will set it equal to the result of invoking the function multiplyBy2 with the argument now.

Will: Absolutely. Invoking, or calling, the function multiplyBy2 with the input of num which immediately, JavaScript is an evaluated language, that means it turns anything that's a mystery into an actual value. Num is a mystery, so it looks it up, oh it's number 3, and puts 3 as the input. And now folk, I want to introduce you to a concept known as an execution context. It has become your favorite concept in all of JavaScript and execution context, Kate's already unmuted because she is ready to say it out loud.

And execution context, people, is the fancy, the posh, word for, actually you've already seen it, the two things you need to run code. And all you do in coding is run code. Two things, a memory, that means a space to store stuff that gets announced, variables, constants, functions. And JavaScript's ability, processing power of the computer to go

through the code line by line and do it. We're already in one, we're already in what's called the global execution context. As soon as we start running code, we need a memory, there it is. And we need to go through the code line by line. That's known as the thread of execution. It threads its way through and executes, does, executes is a fancy word for does, does the code line by line. Threads through, executes through. Line one, line two, line three, and saves stuff to memory.



When you run a function, you create a mini version of that. It's called a local execution context. Local meaning just for what we are doing that function. I'm going to show it with this little box here. Here it is. It has two parts. It has, I'm going to do it in green, the thread of execution. We don't continue down the page to new outputs until we've finished inside the running of multiplyBy2. That's where we are right now. And anything we save, anything declared, inside of multiplyBy2, is stored in this local, or I might call it sometimes temporary. Because it's deleted when we come back out of that function. And Natalie, I'm going to come out of a function, what key word tells me to leave a function?

Natalie: Return.

Will: Return, spot on. Excellent. So into it we go, and the first thing inside of this local memory of running multiplyBy2, are our parameter argument combinations. Parameter, folk, is the placeholder which we then fill in with a number, which in this case was 3, so

3 fills in the parameter input number. Argument 3 goes in the parameter input number. Jim, tell me about our parameter argument combinations inside multiplyBy2 here.

Jim: So, we've got a new execution context establishing input number giving it the value, equal to num, which happens to be 3.

Will: Excellent, so input number our parameter is given the value of the argument which is 3. Happened to have the label num temporarily but we turned it into 3 before we've even started running the function. It got passed in as number 3. Okay, beautiful, then we hit declaring result as a constant is going to be input number by 2 which is 6. And then the final line says this, return result. Joe, do you want to talk me through what JavaScript does in terms of its look up with return results. Tell me what actually returns when it sees, does it return the label result, what happens?

Joe: First, it checks the local memory and it finds 6. And because 6 is not...

Will: A value?

Joe: It's a value, exactly.

Will: It's not unevaluated; it's like a finished work.

Joe: What I meant is, it doesn't have to go into the heap. For now, sorry, but it returns 6, it returns the value 6.

Will: Beautiful, there it is. Out into whatever is on the left hand side. You run the function on the right hand side. Its output's stored on the left hand side. Which here, Joe, is what?

Joe: Output.

Will: Output, excellent, there it is. Output. Beautiful. And our thread, therefore, of execution is now finished inside of multiplyBy2 and comes out, where it goes back to what we call the global execution context, where the next line, let's just make it clear what those lines of code were. So that was line four, this was line five, and now we hit line six, which Joe says to do what?

Joe: Sorry, my thing is blocking it right now. Hold on one second.

Will: Oh don't worry!

Joe: Okay, so new output. So in global memory it reserves a space for new output and then assigns, for a second, it's undeclared and then it assigns it the value of the evaluated results of multiplyBy2.

Will: Right, so in other words, we've got to go and run multiplyBy2 again. And there it is, run multiplyBy2 again, this time with the input of 10, creates, everyone together, for the first time today. A brand new...

Everyone: Execution context!

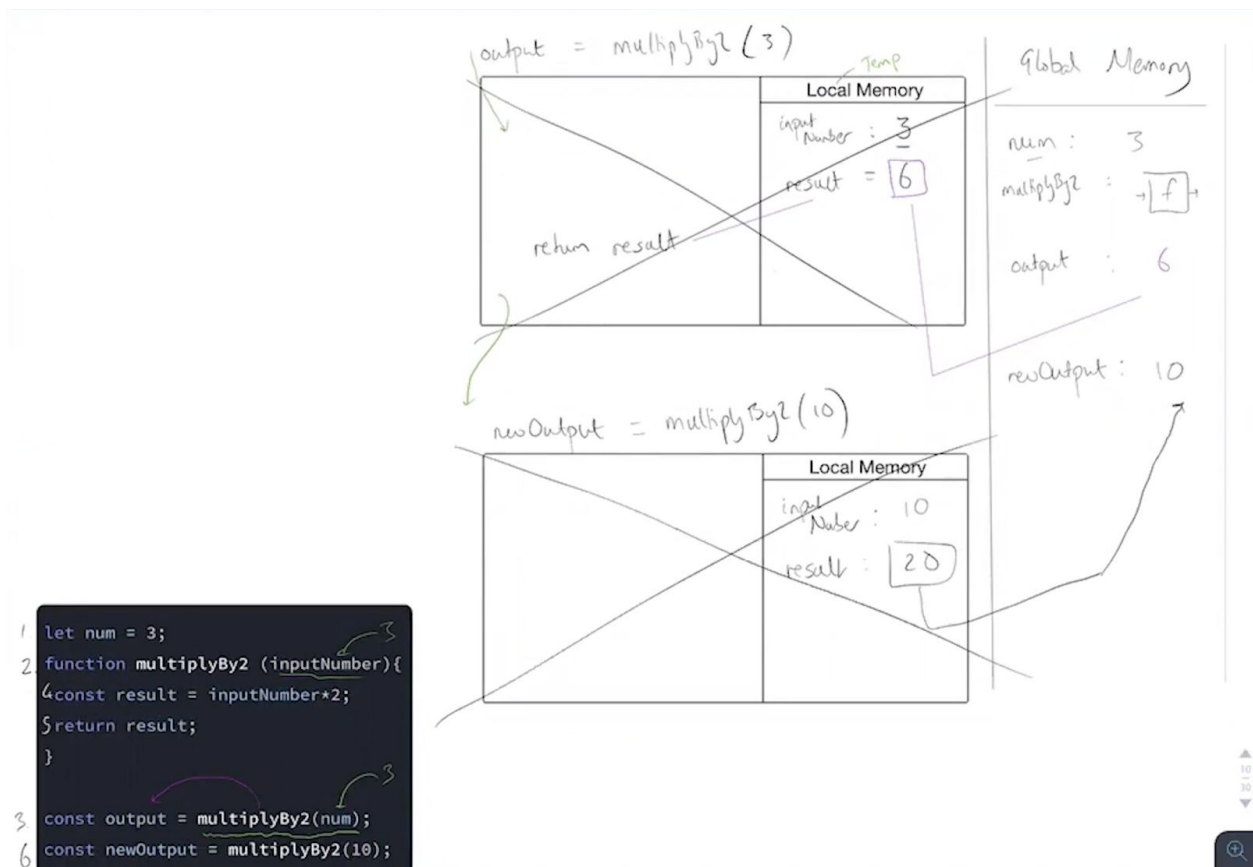
Will: Beautiful, indeed. There it is. Into it we go, and input number, this time, Kate, does it remember the previous input number of 3?

Kate: No.

Will: No, it doesn't give a damn about the previous local memory, that's forgotten. That was forgotten when we exited it earlier. All we held onto was a 6 in global, and global memory always sticks around. Instead, input number is 10, result is then 20, return the value of result into what global constants, Kate?

Kate: Into new output.

Will: Beautiful, there it is. All right folks, so our local memories between these two runnings of multiplyBy2, have no connection. A brand new local memory every single time. We are going to discover today that if instead, a function had also a permanent, or persistent, memory, in addition to it's brand new start from scratch every time local memory, this could change everything we do in programming. All these fancy things we talked about up here, a function that can remember it being run before, and therefore not allow it to run again. A function like memoize, that will save previous results so we don't have to recalculate them. All those things will become possible, the module pattern, if we could give our functions permanent memories, rather than the temporary memories that get deleted each time they finish running. All of that to come.



Lecture: 3 Functions Can Have Multiple Labels

Functions can have multiple names

Functions are just a set of instructions/block of code that has been stored/saved to memory. We can give that set of instructions multiple labels



Functions Two parts to defining a function:

1. The function's name (label)
2. The function's code (definition)

Behind the scenes:

- Functions store a link/reference/address pointing to a location in the 'heap'



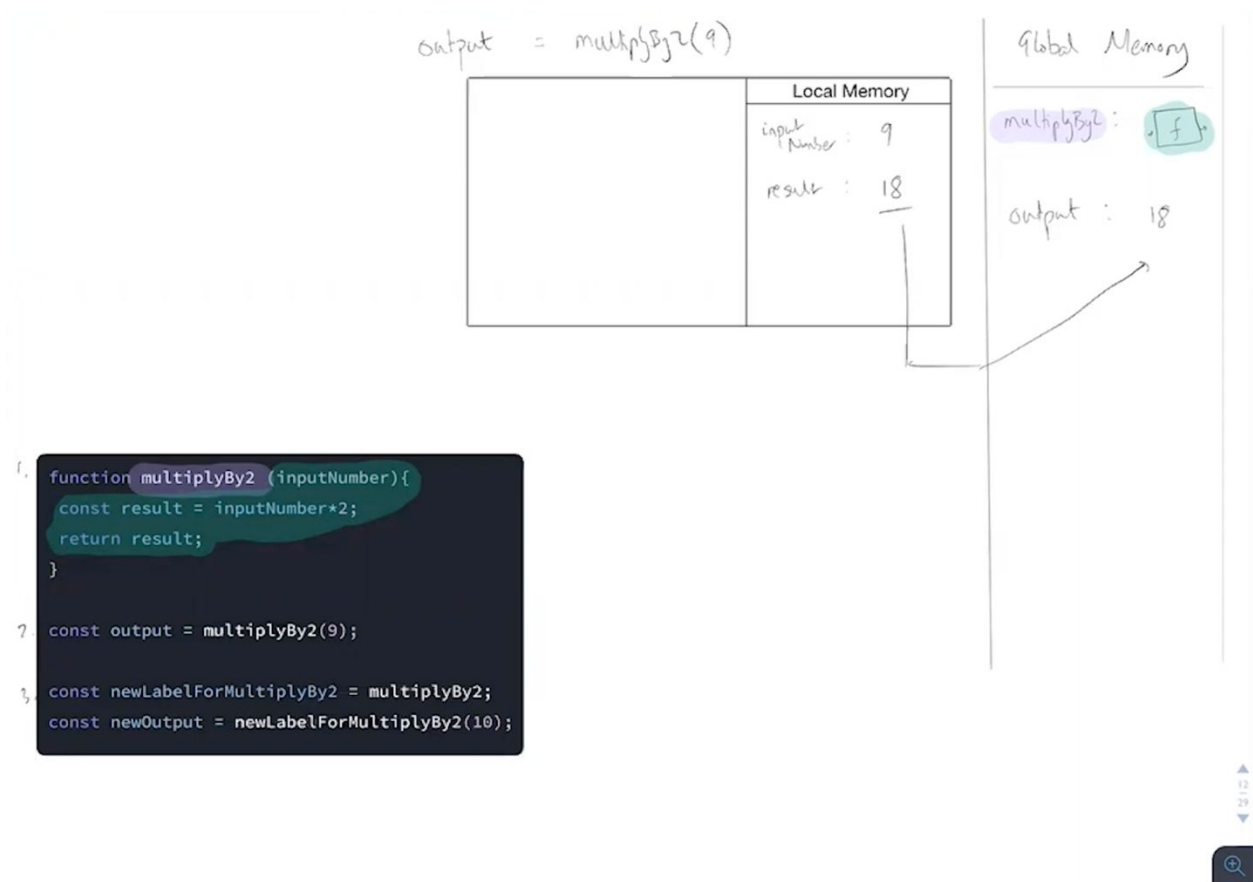
```
function multiplyBy2 (inputNumber){  
  const result = inputNumber*2;  
  return result;  
}  
  
const output = multiplyBy2(9);  
  
const newLabelForMultiplyBy2 = multiplyBy2;  
const newOutput = newLabelForMultiplyBy2(10);
```

Will:

What are we doing when actually save a function? Let's have a look. We saw it a moment ago. But, I want us to really think about exactly what we are doing when we save a function in terms of our computer's memory. We're going to go pretty sophisticated here. So see this block of code on the right hand side? You've got a function being saved, multiplyBy2. We're going to use it with the input of 9, return the result into output. And then, huh, something in other languages you'd look at and go "No, this is not a thing". You're going to declare a label, new label for multiplyBy2, equals multiplyBy2 and then try and run that as a function. This may seem very unfamiliar. If you work in other programming languages this is not a conventional thing to do. But if we get this down, then the next bit, which is the bit that I say the New York Times engineers tend not to have got, the next bit will be even easier for us to get. We'll still go through the next bit slowly, if you get the next bit down, closure is effortless.

Closure is a simple concept if you understand how functions are actually stored in JavaScript. Okay, here we go. Functions can have multiple names. Hm. Functions are just a set of instructions, as code, or a block of code, that has not yet been used, it's not run through it, even said grabbed it and saved it to be used later on. Which means, perhaps you could give those set of instructions multiple labels. Huh. Let's see this play out, folks.

Let's start running our code, add a global memory, there it is. Pre-made global memory. Oh, no... Oh, well. A pre-made global memory. Add our global memory and then let's grab the code that we are going to walk through line by line. Right. And it's going to be a very revealing code, I think. There it is. Alright. So, line one, Joe. Joe what are we doing here in line one of our code?



Joe: We're storing the function multiplyBy2 in global memory.

Will: Yeah, absolutely. So that has two parts to it, doesn't it? So we declared the function multiplyBy2. There it is. What does that actually mean? It's got two parts. One, we have a label. Just like if I declared num is 3, I have a label num. And then, my label has a value, just like the value of num being 3. Turns out in JavaScript, a function is just like the number 3, a thing you can store. Its code as a value. A thing you can store associated with a label. Let's do that. I'm going to highlight which bit is which. So, there's my label, okay. There's my label. And now, I'm going to store the function's code, including the parameter name, there it is.

Okay, I'm going to go into some more detail in a second, but just for now this should be okay. Let's have the color green. So this, from the input, the body of the function, all of that is this portion here. Okay. Got it. Good, let's keep going. Next line, Natalie, what's the next line tell us to do? We've saved that multiplyBy2 function, that was line one. What's line two?

Natalie: So, line two is the line that says const output equals multiplyBy2 with the argument of 9 past in, so we are going to declare a constant output in global memory and assign it the value of the definition of multiplyBy2, or-

Will: Oh, oh.

Natalie: Actually, right now it's uninitialized.

Will: Uh huh, so Natalie fell there into the most natural trap, which is that we sort of see an equal sign, hold on, output multiplyBy2. But what symbols, Natalie, tell us that the right hand side there is absolutely un-saveable right now. We can not save that as the value of output because it is a command and I know it's a command. Go do something, why, Natalie?

Natalie: Because it has the parentheses.

Will: Because it has the parentheses on there. Exactly. Alright, so what is it a command to go and do, Natalie? It's a command to go and?

Natalie: To invoke the function multiplyBy2.

Will: To create a?

Natalie: A local execution context.

Will: Execution context to run the code of the function. Perfect. Into it we go. First thing in the local memory of it, and it is repetitive, but we're going to see something very interesting in a second. Into it we go. First thing in the local memory, Natalie, is what?

Natalie: The variable input number, and we are going to assign that the value of 9, which was the argument passed in.

Will: So, Natalie, forgive me. So I'm going to say I protocol it variable but instead parameter.

Natalie: Okay.

Will: And the argument exactly 9. Perfect, and we know this result is going to be 18, return the value of result out into output. There it is, 18. Beautiful. Excellent. Now, things get interesting. What the hell are we doing in this next line down here? (Whistle!) Okay. Who wants to give it a shot? Kate, you want to give it a shot?

Kate: Sure.

Will: Okay Kate, here we go. So, what are we doing in this third line, so to speak, down here?

Kate: So we're going to create a new const in global memory and the label of it it will be new label for multiplyBy2.

Will: Beautiful. Actually, I'm going to put it, it doesn't really matter what order our stuff is in local memory, I'm going to put it here. I might give you a clue, Kate, as to what we're going to have happen here. So a new label for multiplyBy2, there it is.

Kate: Okay, and-

Will: Okay, what are we assigning to it?

Kate: We're going to store in that const, the function definition of multiplyBy2.

Will: Is it a copy of it? Let's have a look. Let's have a look, Kate.

Kate: Not yet. I don't think it is.

Will: And you're absolutely right. But let's see why. So when I, so when JavaScript sees this reference here to multiplyBy2, what does it do? Kate, what does JavaScript do whenever it sees a work that's not a keyword and it's puzzled by it? What's it do? Where's it go look?


Kate: In the local or global memory.


output = multiplyBy2(9)

Local Memory	
input Number :	9
result :	18

newOutput

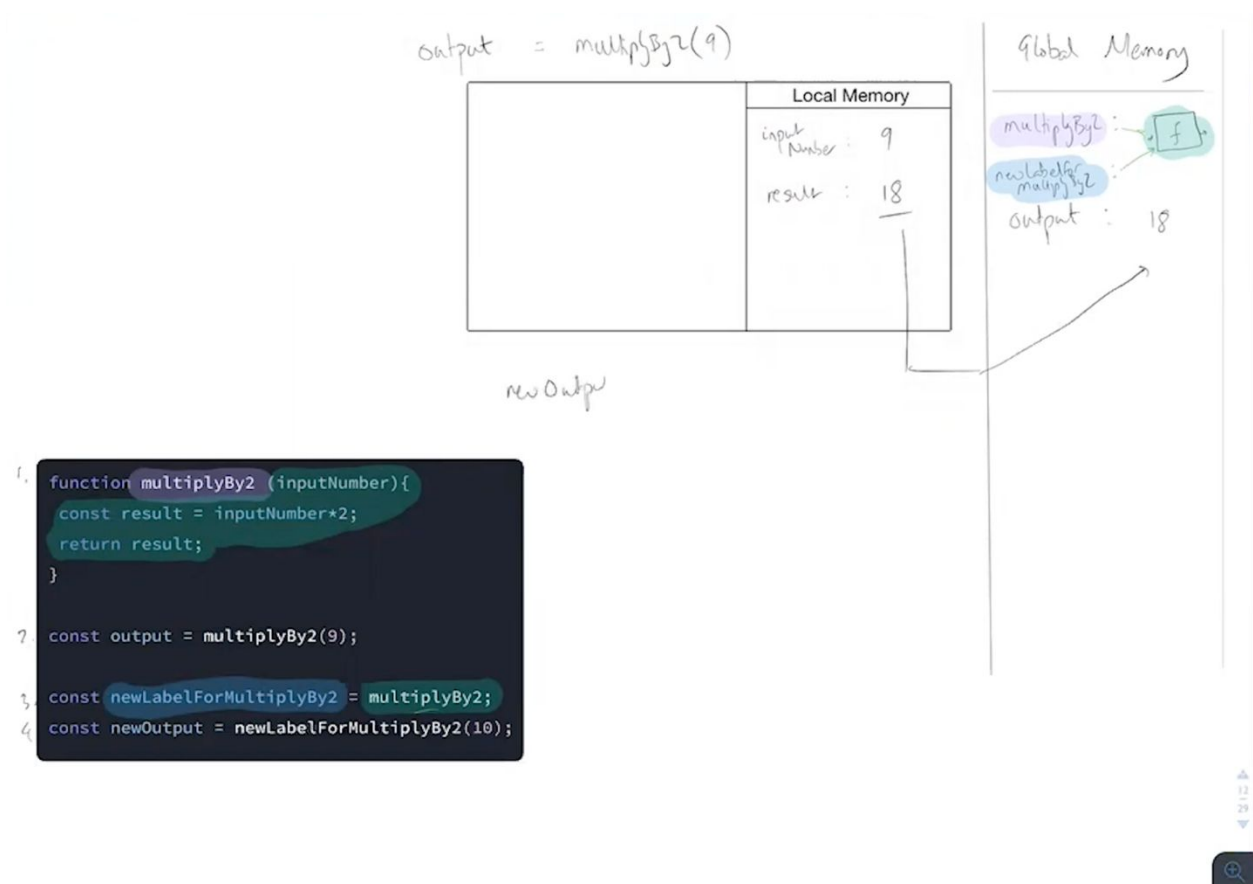
Global Memory

multiplyBy2 : 

newLabelForMultiplyBy2 : 

output : 18

```
1. function multiplyBy2 (inputNumber){  
  const result = inputNumber*2;  
  return result;  
}  
2. const output = multiplyBy2(9);  
3. const newLabelForMultiplyBy2 = multiplyBy2;  
4. const newOutput = newLabelForMultiplyBy2(10);
```



Will: That's it. So let's go do that. And it never takes a label and stores the label, it uses the label to find whatever the value is. The thing on the right hand side here. Which is the function definition of multiplyBy2. And so it's going to have new label for multiplyBy2 be assigned as a new label for that underlying function definition previously given the label

multiplyBy2. We now have two labels for the same function. I'm going to make that clear by using a different color here. Let's use blue for new label for multiplyBy2 and then we can see, people, right? So what is multiplyBy2, oh folks, I'm going to use exactly the same color of what really multiplyBy2 is. It's not the label. No, no, no, no. It's not the label. It's the underlying function that is being assigned to new label for multiplyBy2. I'll show you why in a second. In terms of something known as the heap, which we'll see in a second. Alright, so, let's just do it though for now.

So we've now got two labels for the functionality that was originally given the label multiplyBy2. Now we've got two labels for it. multiplyBy2 and new label for multiplyBy2. Alright, let's hit the final line. Joe, talk me through the final line here. New output.

Joe: So, new output is stored in global memory and then for like a half a second or a millisecond we're looking to evaluate what will be stored there as the evaluated result of new label for multiplyBy2 with the argument of 10.

Will: Beautiful. Very eloquent as well, Joe. My terrible long names for functions. Exactly, there it is. By the way, folks, these long names for functions, they call these semantic function names, or semantic labels for functions. Semantic meaning they have meaning in their own right. They have meaningful names. Excellent, there it is. We open execution context for it, we go in, oh but look, we're calling new label for multiplyBy2, we go looking in memory for what it is. Ah, it's the functionality formerly known as multiplyBy2. Because new label multiplyBy2 was just another label for the underlying definition that was previously known as multiplyBy2. So in we go, and Joe, what's the first thing we do inside the execution of new label for multiplyBy2? So to be clear, people, we're now on this line, executing new label for multiplyBy2, what's the first thing we do inside, Joe?

Joe: Declare in local memory an input number and then assign a new value 10.

Will: Beautiful. And then set response to 20 and return the value of result out into new output. There it is. Beautiful. Alright, we used a new label for the underlying function that was assigned previously the label multiplyBy2. Basically, see a function declaration as a two-part thing. There's the setting up the label and then there's the storing all the code, meaning, I can set up another label for the same underlying code and it looks like I'm setting up as a label for multiplyBy2, I'm not. I'm setting it up as a label for whatever functionality is stored currently under the label multiplyBy2. Okay, this is going to be turning out to be incredibly important in understanding closure. This is going to turn out to be remarkably important for understanding the next thing that's coming, which is returning a function from a function. Once you understand that, it will help us can understand closure.

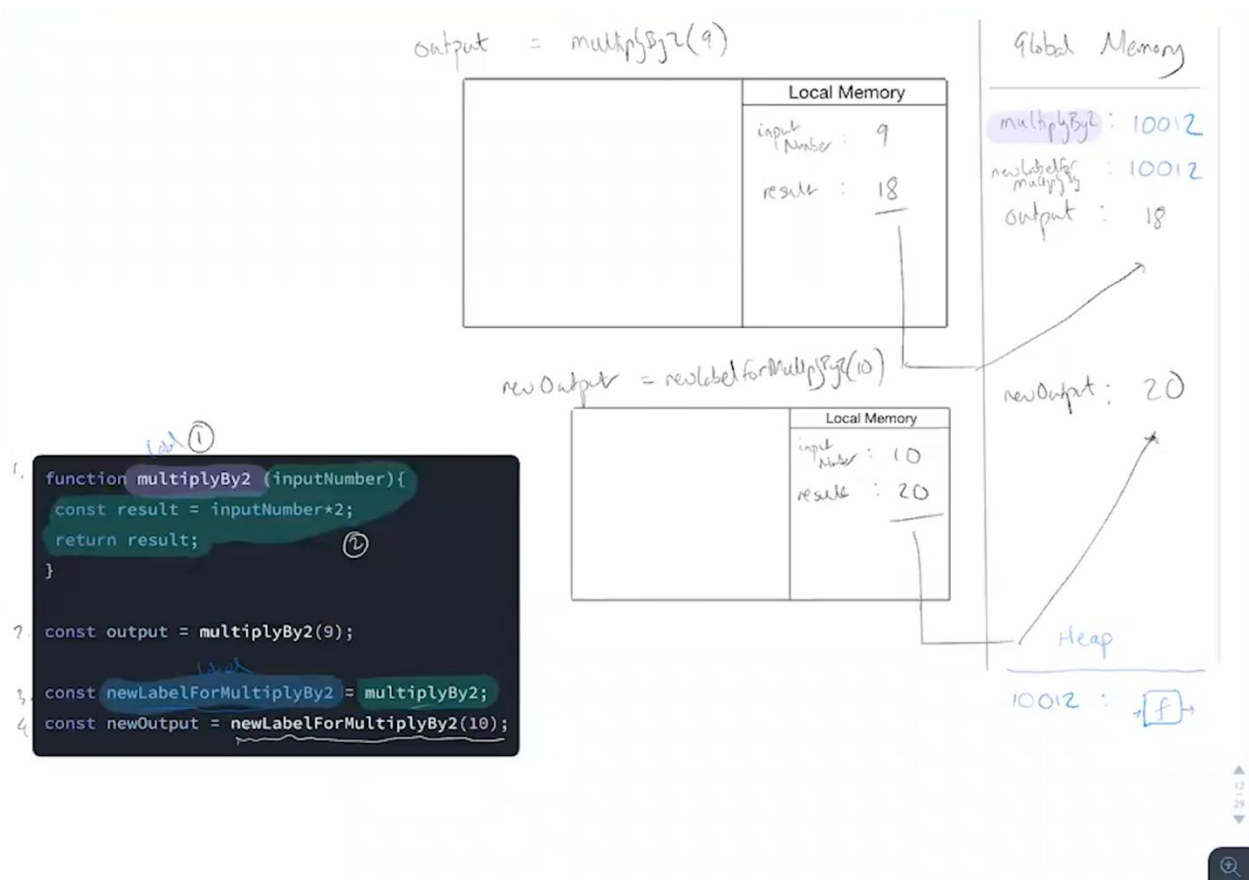
Alright, here we're going to have a bit of an interlude that I don't know if we're going to include as automatically in the video. But, we're going to have as an interlude. Which is, talking about what is actually happening under the hood here. A little bit, what they call hand-waving, meaning not perfectly precise but enough to describe close to what's happening. So, check this out. How do you have two labels for the same function? Over here, how do you have that? That's very strange.

What actually is happening, people, is there's a messy, consider the global memory that we look at normally as being a fast access store of data. It's pretty restricted on what you can save to it. You can't save, like, giant objects, or giant arrays, or giant functions. So where are they, or really any objects, arrays or functions, so where are they stored? Where are they stored? Well, folks it actually turns out they're stored, functions, objects and arrays, are stored in a special separate, I'm going to use my blue pen for this, a special separate position in memory, check it out, called the heap. It's kind of a flexible, less optimized for fast look-up store of more complex data in your application. Functions, objects, and arrays. So actually, when I declare in line one the function multiplyBy2, here's what actually happens. Check it out, people. Here's what actually happens.

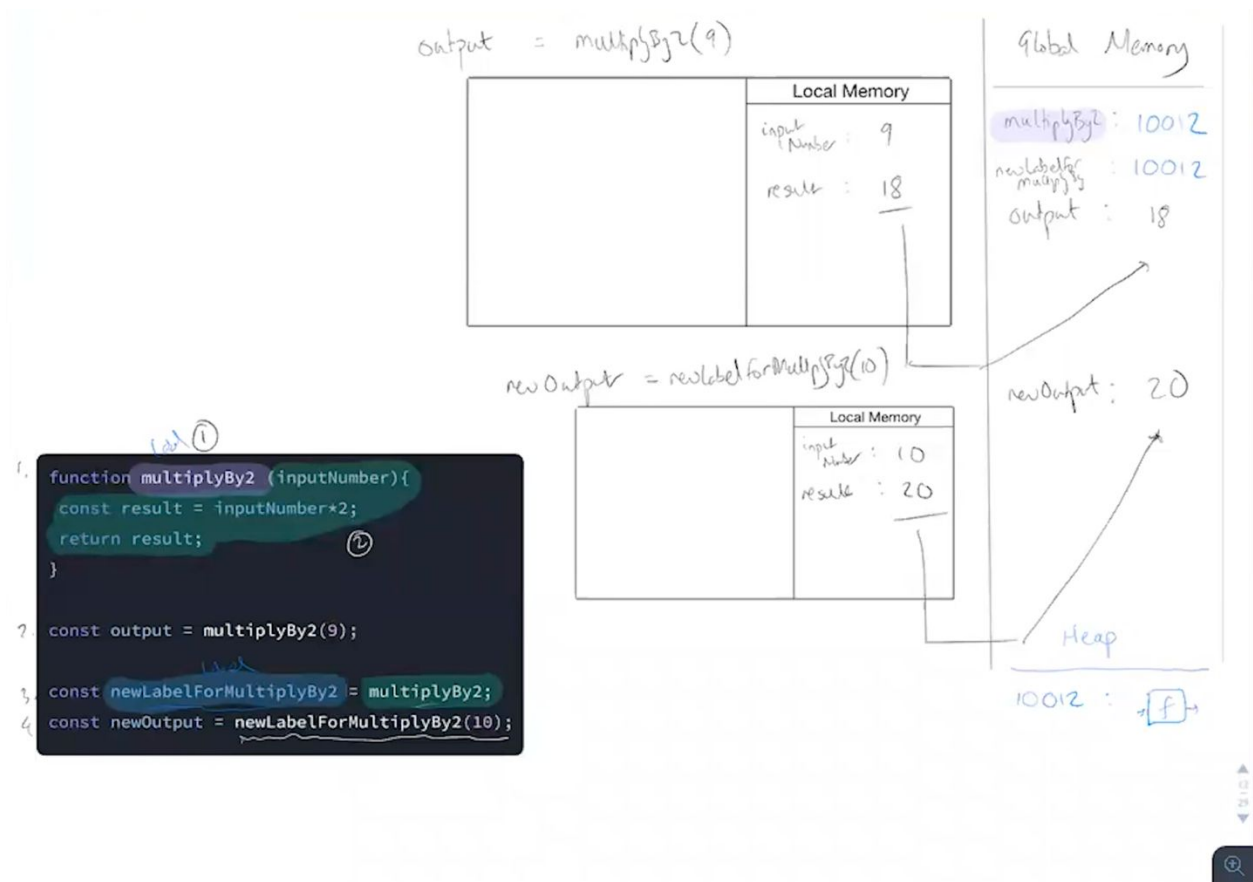
We declare the function multiplyBy2 and I assign to it an address, like a fast, like a link to a position in this place known as the heap, which is where I actually store my underlying function. Given that, Kate, what do you think when I do my new label for multiplyBy2 is multiplyBy2? What do you think I really store in new label for multiplyBy2, Kate?

Kate: I'm going to guess that it's also 10012?

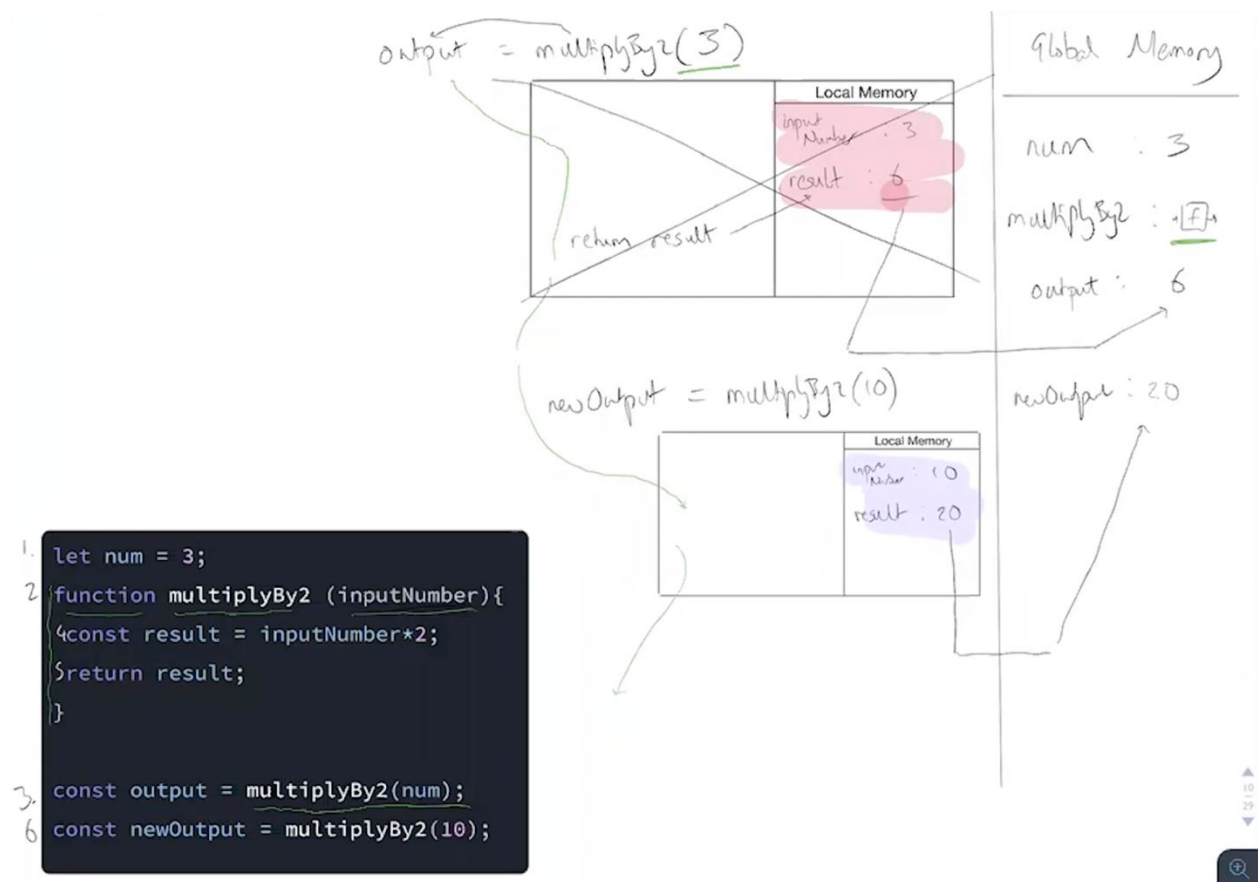
Will: She's spot on. Exactly. This is really what's happening when we're declaring assigning functions. Which, by the way, folks, also means we might be able to pass around into functions, outer functions, whole functions, because really there we're just going to pass in the address to another function, pass out an address to another position in memory. Because the underlying function is stored separately, completely, all together, in the heap. When you hear the term pass by reference or pass by value this is referring to if I'm passing an input into a function or returning an output from a function and the thing I'm passing in is either an array, an object or a function. Those aren't stored directly in the global memory at all. They're stored in this heap place. And what I'm actually passing into a function when it's an array, an object or a function I'm passing in, what I'm actually passing in is the address to that function.



Lecture: 4 Q&A



Will: Alright, at this point folk, let's have thumbs on where our understanding's at. Throw out those edge case questions people. Jump in. I want to see those edge case questions. Know that there's going to be people who benefit massively from- Elena has one, Natalie has one, Jim's clear, Joe has one. Why don't we start with Natalie? Jump in Natalie.



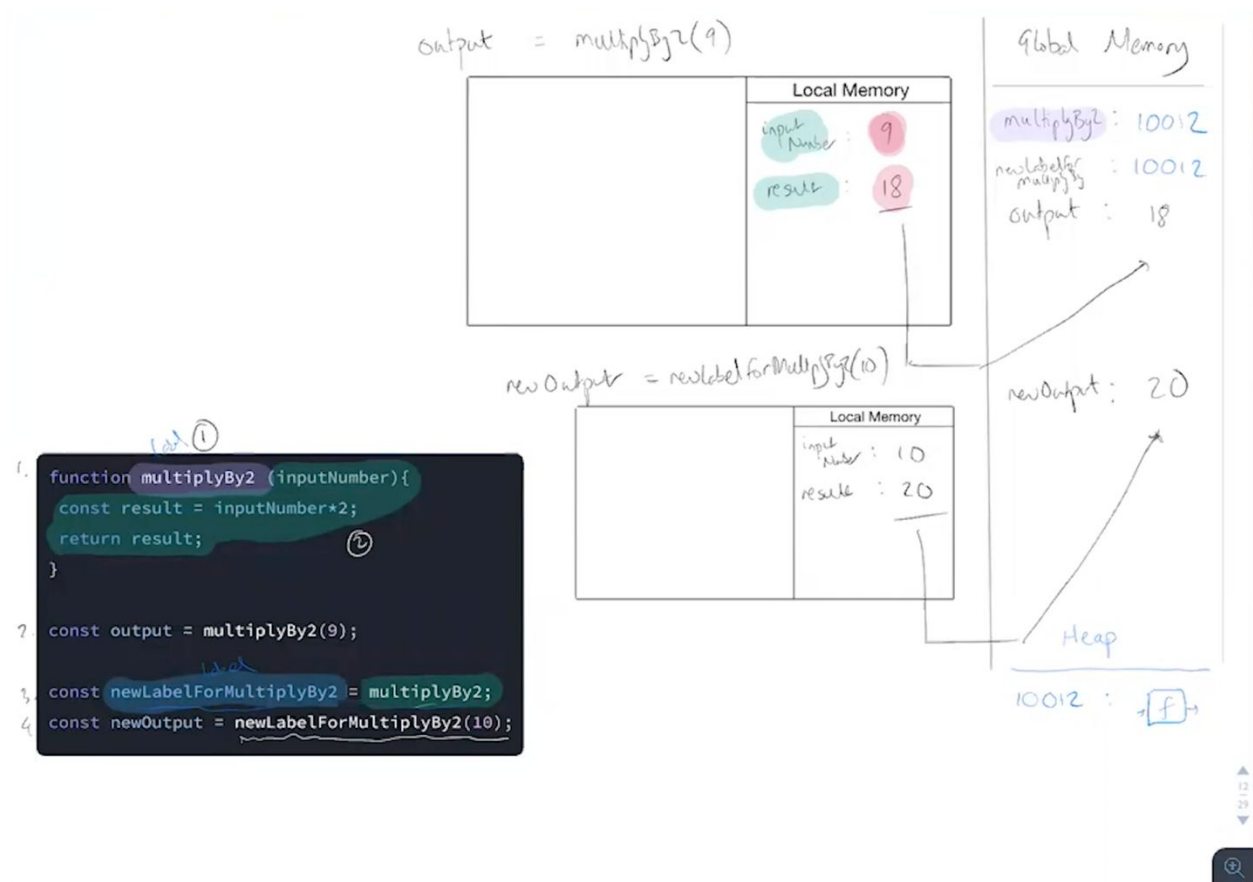
Natalie: Okay. My question goes back to the first function that we diagrammed out, so maybe if you could scroll out. Yes. On line- Where was it? On line three, where we declared the constant output, you were saying that because this is a constant variable and it can't be reassigned, we would say that it's initially uninitialized and not undefined because if we gave it the label undefined then we couldn't change it later once we-

Will: The value undefined. You wouldn't be able to change its value. Yeah, Absolutely. Yeah.

Natalie: Yeah. If instead, we didn't make it a constant variable, we used let, in that case does it start undefined and then the value changes?

Will: That is a great question. Natalie, why the first question you ask is the toughest to answer? I'll tell you with var, it does start undefined and then it reassigned. With let, we will re-ask this question a little bit later on by which time I will have the answer, Natalie. Great question Natalie. Don't ask questions I don't know the answer to. This is a disaster, Natalie. We'll leave this in for the outtakes bit.

Alright. Elena, go for your question. It better be basic. Better be just, "how am I using this pen?" or something.



Elena: So I have two questions. First, I have two objects, right? Two different objects but they are absolutely equal. Values are equal. But when you compare them-

Will: Right. But then they're saved in separate positions, you're saying.

Elena: Yes. Yes.

Will: In the heap...they're saved in separate positions. Correct. Yes, okay.

Elena: Yes. And when we compare them, the result will be false, correct?

Will: It depends on the way we're comparing. So if we're comparing them for their similarity then we get yes, they're equal. If we're instead comparing them for their, are they the same position, same thing, in memory, well then we're going to get false. And you need to use prequels or doubles, which will distinguish which type of comparison you're making.

Elena: Mm-hmm (affirmative). So if I used double equal sign, that will be return true, correct?

Will: That would be true if you use three equals because their pattern is the same. Their- what do they call it?- their shape. Their shape is the same. The object's shape is identical. But the objects are separate in memory. Great question Elena.

Elena: Okay. And the second one. Here, line four, when we call the function you label for multiplyBy2. 10 is actually the argument correct?

Will: 10 is our argument. Let me just show you this. Result to me is the label and this on the right hand side here, 18 is the value. I've never found a name besides label. I don't think it's fair to say it's- because the constant to me is the overall combination of the two. Where as it's a constant it's comprised a label and a value. When it comes to those being the input of a function, the label is known as, Elena the label is known as the- When it's the input of a function, the label is known as the- The value when it's the input of a function, the value is the argument and the label is the

Elena: Parameter?

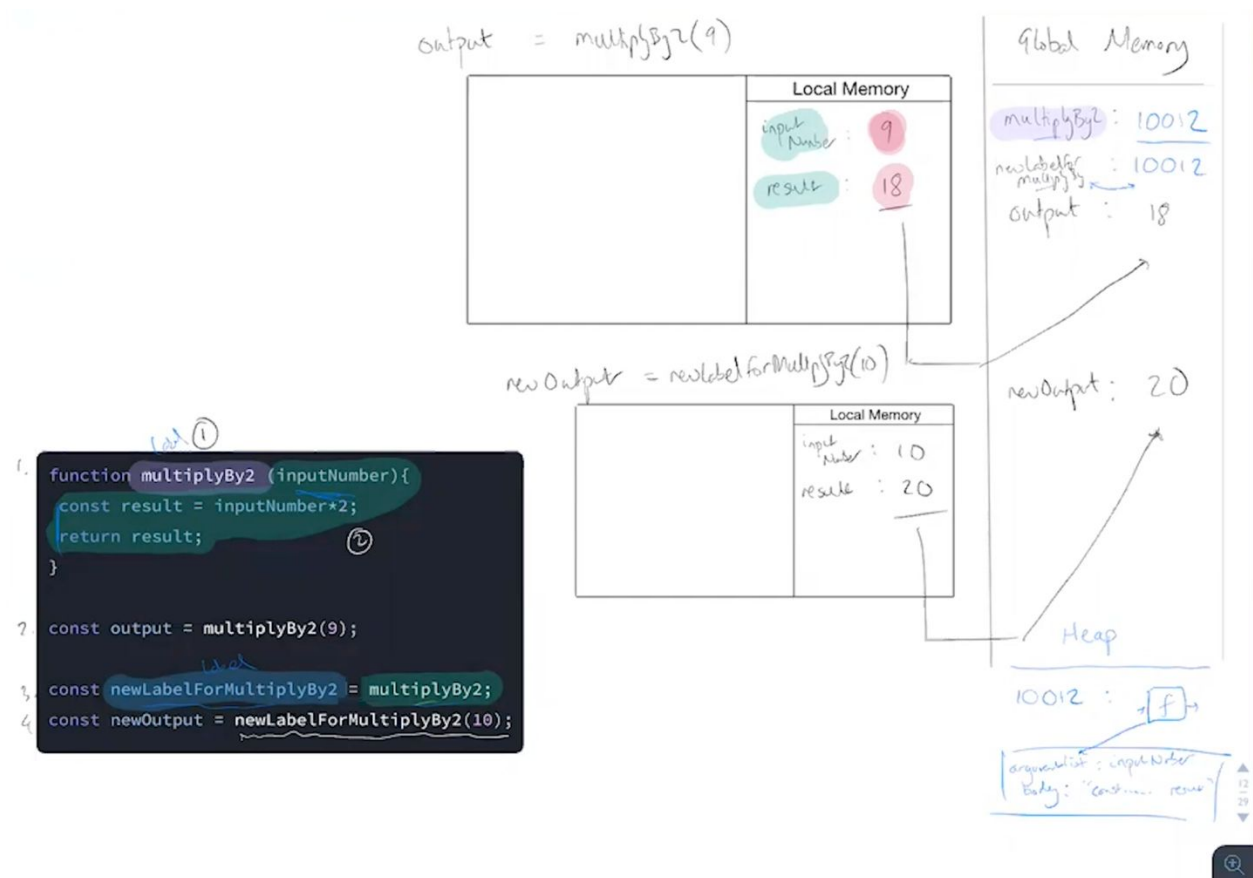
Will: Parameter, exactly. When it comes to inputs, the argument is the value.

Elena: Mm-hmm (affirmative).

Will: And the label is known as the parameter. So inputNumber is our parameter and 9 is our argument.

Elena: Okay.

Will: Excellent. Great question Elena. Joe you have one, jump in.



Joe: I don't know if this actually matters, but I am just curious how the function definition is stored in memory because I kind of visually in my own head imagine that as a string that looks like-

Will: What a great- you know what I didn't even put this in because I hoped somebody would ask it. Let's have a look at how it's stored. I just want you to know it's going to look like an object. While separately functions are objects in JavaScript, this is not what we're seeing here. This is an under the hood pairing or kind of setup. Have a look at this. Here's how the function would actually be stored. Remember this function in position 10012 is, well, had the label `multiplyBy2`, and now also another label, `new label for multiplyBy2`. How is the function stored under the hood? Have a look at this. So you get, they call it I think argument list. Which I think is a weird name for it really, given that there's already parameter list. But argument list, which would be here what, Joe?

Joe: Input number.

Will: Spot on. There it is. Input number. And then body, which here, Joe, would be

Joe: Const result, yeah, the next few lines.

Will: Think of it as a string, all the way through to result. Okay, and it looks something like an object. That's why I tend not to bring this up. This is not a JavaScript object. This is one layer deeper into JavaScript's belly, and this is just how it's stored under the hood. Okay, think of it as sort of a pairings of this part of the function, here it is, this part of the function, here it is, and so forth. Does that make sense Joe? Great. That's how it's stored actually under the hood. And that's why, if you look, that's why that could very reasonably have multiple labels. Because you've taken the two pieces, the body and the parameter list, and that's all the function is. And it's got a label, which when you run, it kicks all that stuff off.

Alright, excellent Joe, great question. Kate, Jim, any questions? Kateek, any questions my friend?

Kate: Maybe, actually.

Will: Go ahead, Kate.

Kate: So when it, let me think if I can phrase this properly, with const, you can't actually reassign the value of the label, or that's assigned to the label, but you can modify the thing that label points to. Correct? And that's because a const points to a certain space in the heap? And you can't change that pointer, but you can change-

Will: Well because we don't get to manually change these addresses, do we?

Kate: Right, no.

Will: We're not allowed to. So this, when you declare with const, you have made a connection here that is controlled by the const restriction. You can't switch it out. But, you can certainly, if you were assigned an array using const or assigned an object using const, add elements, remove them. Add properties, remove them, you just can't change the pointers. I don't like the term pointers; they're not explicit. You can't change the link, which is that address. You can't change the address out completely. You're going, "Ah I change my mind, you know what actually, it points to a totally different array, or a totally different object." Makes sense, Kate?

Kate: Yes, thank you.

Will: And that's really why const is restrictive, in a good way. It prevents others overwriting our code, but actually still allows us to, it's the binding, they say, the link between the label and the address to the thing stored in that label. That is immutable. Can't be changed. But the thing that's stored, the object, the array, whatever, that's far from immutable. That's entirely mutable, meaning it can be changed. Okay. Great Question, Kate.

Alright, excellent folk. Let's keep moving. All of this foundation was to ensure that when we get to the next piece, returning a function from a function, it should follow more naturally. And then once we get that down, closure follows effortlessly.