I never understood JavaScript closures | by Olivier De Meulder | DailyJS | MediumOlivier De MeulderFollowDailyJS--253ListenShareAs the title states, JavaScript closures have always been a bit of a mystery to me. I have read multiple articles, I have used closures in my work, sometimes I even used a closure without realizing I was using a closure.Recently I went to a talk where someone really explained it in a way it finally clicked for me. I'll try to take this approach to explain closures in this article. Let me give credit to the great folks at CodeSmith and their JavaScript The Hard Parts series.Some concepts are important to grok before you can grok closures. One of them is the execution context.This article has a very good primer on Execution Context. To quote the article:When code is run in JavaScript, the environment in which it is executed is very important, and is evaluated as 1 of the following:Global code — The default environment where your code is executed for the first time.Function code — Whenever the flow of execution enters a function body.(…)(…), let's think of the term execution context as the environment / scope the current code is being evaluated in.In other words, as we start the program, we start in the global execution context. Some variables are declared within the global execution context. We call these global variables. When the program calls a function, what happens? A few steps:When does the function end? When it encounters a return statement or it encounters a closing bracket }. When a function ends, the following happens:Before we get to closures, let's take a look at the following piece of code. It seems very straightforward, anybody reading this article probably knows exactly what it does.In order to understand how the JavaScript engine really works, let's break this down in great detail.That was a very long winded explanation for a very simple program, and we haven't even touched upon closures yet. We will get there I promise. But first we need to take another detour or two.We need to understand some aspects of lexical scope. Take a look at the following example.The idea here is that we have variables in the local execution context and variables in the global execution context. One intricacy of JavaScript is how it looks for variables. If it can't find a variable in its local execution context, it will look for it in its calling context. And if not found there in its calling context. Repeatedly, until it is looking in the global execution context. (And if it does not find it there, it's undefined). Follow along with the example above, it will clarify it. If you understand how scope works, you can skip this.So in this example, we need to remember that a function has access to variables that are defined in its calling context. The formal name of this phenomenon is the lexical scope.In the first example the function addTwo returns a number. Remember from earlier that a function can return anything. Let's look at an example of a function that returns a function, as this is essential to understand closures. Here is the example that we are going to analyze.Let's go back to the step-by-step breakdown.As expected the console will print 15. We really go through a bunch of hoops here. I am trying to illustrate a few points here. First, a function definition can be stored in a variable, the function definition is invisible to the program until it gets called. Second, every time a function gets called, a local execution context is (temporarily) created. That execution context vanishes when the function is done. A function is done when it encounters return or the closing bracket }.Take a look a the next code and try to figure out what will happen.Now that we got the hang of it from the previous two examples, let's zip through the execution of this, as we expect it to run.Try this out for yourself and see what happens. You'll notice that it is not logging 1, 1, and 1 as you may expect from my explanation above. Instead it is logging 1, 2 and 3. So what gives?Somehow, the increment function remembers that counter value. How is that working?Is counter part of the global execution context? Try console.log(counter) and you'll get undefined. So that's not it.Maybe,

when you call increment, somehow it goes back to the the function where it was created (createCounter)? How would that even work? The variable increment contains the function definition, not where it came from. So that’s not it.So there must be another mechanism. The Closure. We finally got to it, the missing piece.Here is how it works. Whenever you declare a new function and assign it to a variable, you store the function definition, as well as a closure. The closure contains all the variables that are in scope at the time of creation of the function. It is analogous to a backpack. A function definition comes with a little backpack. And in its pack it stores all the variables that were in scope at the time that the function definition was created.So our explanation above was all wrong, let’s try it again, but correctly this time.So now we understand how this works. The key to remember is that when a function gets declared, it contains a function definition and a closure. The closure is a collection of all the variables in scope at the time of creation of the function.You may ask, does any function has a closure, even functions created in the global scope? The answer is yes. Functions created in the global scope create a closure. But since these functions were created in the global scope, they have access to all the variables in the global scope. And the closure concept is not really relevant.When a function returns a function, that is when the concept of closures becomes more relevant. The returned function has access to variables that are not in the global scope, but they solely exist in its closure.Sometimes closures show up when you don’t even notice it. You may have seen an example of what we call partial application. Like in the following code.In case the arrow function throws you off, here is the equivalent.We declare a generic adder function addX that takes one parameter (x) and returns another function.The returned function also takes one parameter and adds it to the variable x.The variable x is part of the closure. When the variable addThree gets declared in the local context, it is assigned a function definition and a closure. The closure contains the variable x.So now when addThree is called and executed, it has access to the variable x from its closure and the variable n which was passed as an argument and is able to return the sum.In this example the console will print the number 7.The way I will always remember closures is through the backpack analogy. When a function gets created and passed around or returned from another function, it carries a backpack with it. And in the backpack are all the variables that were in scope when the function was declared.If you enjoyed reading this, don’t forget the applause. Thank you.