Olivier De Meulder Follow
Software engineer NY Times, dad, husband, TaeKwonDo black belt, cook, news junkie. Here are some of my mind ramblings.
Sep 7, 2017 · 17 min read

I never understood JavaScript closures

Until someone explained it to me like this ...

As 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.

Before we start

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:

1. JavaScript creates a new execution context, a local execution context
2. That local execution context will have its own set of variables, these variables will be local to that execution context.
3. The new execution context is thrown onto the *execution stack*. Think of the execution stack as a mechanism to keep track of where the program is in its execution

When does the function end? When it encounters a `return` statement or it encounters a closing bracket `}`. When a function ends, the following happens:

1. The local execution contexts pops off the execution stack
2. The functions sends the return value back to the calling context. The calling context is the execution context that called this function, it could be the global execution context or another local execution context. It is up to the calling execution context to deal with the return value at that point. The returned value could be an object, an array, a function, a boolean, anything really. If the function has no `return` statement, `undefined` is returned.
3. The local execution context is destroyed. This is important. Destroyed. All the variables that were declared within the local execution context are erased. They are no longer available. That's why they're called local variables.

A very basic example

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.

```
1: let a = 3
2: function addTwo(x) {
3:   let ret = x + 2
4:   return ret
5: }
6: let b = addTwo(a)
7: console.log(b)
```

In order to understand how the JavaScript engine really works, let's break this down in great detail.

1. On line 1 we declare a new variable `a` in the global execution context and assign it the number `3`.

2. Next it gets tricky. Lines 2 through 5 are really together. What happens here? We declare a new variable named `addTwo` in the global execution context. And what do we assign to it? A function definition. Whatever is between the two brackets `{ }` is assigned to `addTwo`. The code inside the function is not evaluated, not executed, just stored into a variable for future use.

3. So now we're at line 6. It looks simple, but there is much to unpack here. First we declare a new variable in the global execution context and label it `b`. As soon as a variable is declared it has the value of `undefined`.

4. Next, still on line 6, we see an assignment operator. We are getting ready to assign a new value to the variable `b`. Next we see a function being called. When you see a variable followed by round brackets `(…)`, that's the signal that a function is being called. Flash forward, every function returns something (either a value, an object or `undefined`). Whatever is returned from the function will be assigned to variable `b`.

5. But first we need to call the function labeled `addTwo`. JavaScript will go and look in its *global* execution context memory for a variable named `addTwo`. Oh, it found one, it was defined in step 2 (or lines 2–5). And lo and behold variable `addTwo` contains a function definition. Note that the variable `a` is passed as an argument to the function. JavaScript searches for a variable `a` in its *global* execution context memory, finds it, finds that its value is `3` and passes the number `3` as an argument to the function. Ready to execute the function.

6. Now the execution context will switch. A new local execution context is created, let's name it the 'addTwo execution context'. The execution context is pushed onto the call stack. What is the first thing we do in the local execution context?

7. You may be tempted to say, “A new variable `ret` is declared in the *local* execution context”. That is not the answer. The correct answer is, we need to look at the parameters of the function first. A new variable `x` is declared in the local execution context. And since the value `3` was passed as an argument, the variable x is assigned the number `3`.

8. The next step is: A new variable `ret` is declared in the *local* execution context. Its value is set to undefined. (line 3)

9. Still line 3, an addition needs to be performed. First we need the value of `x`. JavaScript will look for a variable `x`. It will look in the local execution context first. And it found one, the value is `3`. And the second operand is the number `2`. The result of the addition (`5`) is assigned to the variable `ret`.

10. Line 4. We return the content of the variable `ret`. Another lookup in the *local* execution context. `ret` contains the value `5`. The function returns the number `5`. And the function ends.

11. Lines 4–5. The function ends. The local execution context is destroyed. The variables `x` and `ret` are wiped out. They no longer exist. The context is popped of the call stack and the return value is returned to the calling context. In this case the calling context is the global execution context, because the function `addTwo` was called from the global execution context.

12. Now we pick up where we left off in step 4. The returned value (number `5`) gets assigned to the variable `b`. We are still at line 6 of the little program.

13. I am not going into detail, but in line 7, the content of variable `b` gets printed in the console. In our example the number `5`.

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.

Lexical scope.

We need to understand some aspects of lexical scope. Take a look at the following example.

```
1: let val1 = 2
2: function multiplyThis(n) {
3:   let ret = n * val1
4:   return ret
5: }
6: let multiplied = multiplyThis(6)
7: console.log('example of scope:', multiplied)
```

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.

1. Declare a new variable `val1` in the global execution context and assign it the number `2`.
2. Lines 2–5. Declare a new variable `multiplyThis` and assign it a function definition.
3. Line 6. Declare a new variable `multiplied` in the global execution context.
4. Retrieve the variable `multiplyThis` from the global execution context memory and execute it as a function. Pass the number `6` as argument.
5. New function call = new execution context. Create a new local execution context.

6. In the local execution context, declare a variable `n` and assign it the number 6.

7. Line 3. In the local execution context, declare a variable `ret`.

8. Line 3 (continued). Perform an multiplication with two operands; the content of the variables `n` and `val1`. Look up the variable `n` in the local execution context. We declared it in step 6. Its content is the number `6`. Look up the variable `val1` in the local execution context. The local execution context does not have a variable labeled `val1`. Let's check the calling context. The calling context is the global execution context. Let's look for `val1` in the global execution context. Oh yes, it's there. It was defined in step 1. The value is the number `2`.

9. Line 3 (continued). Multiply the two operands and assign it to the `ret` variable. 6 * 2 = 12. `ret` is now `12`.

10. Return the `ret` variable. The local execution context is destroyed, along with its variables `ret` and `n`. The variable `val1` is not destroyed, as it was part of the global execution context.

11. Back to line 6. In the calling context, the number `12` is assigned to the `multiplied` variable.

12. Finally on line 7, we show the value of the `multiplied` variable in the console.

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.

A function that returns a function

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.

```
 1: let val = 7
 2: function createAdder() {
 3:   function addNumbers(a, b) {
 4:     let ret = a + b
 5:     return ret
 6:   }
 7:   return addNumbers
 8: }
 9: let adder = createAdder()
10: let sum = adder(val, 8)
11: console.log('example of function returning a function:
', sum)
```

Let's go back to the step-by-step breakdown.

1. Line 1. We declare a variable `val` in the global execution context and assign the number `7` to that variable.
2. Lines 2–8. We declare a variable named `createAdder` in the global execution context and we assign a function definition to it. Lines 3 to 7 describe said function definition. As before, at this point, we are not jumping into that function. We just store the function definition into that variable (`createAdder`).
3. Line 9. We declare a new variable, named `adder` , in the global execution context. Temporarily, `undefined` is assigned to `adder` .
4. Still line 9. We see the brackets `()` ; we need to execute or call a function. Let's query the global execution context's memory and look for a variable named `createAdder` . It was created in step 2. Ok, let's call it.
5. Calling a function. Now we're at line 2. A new local execution context is created. We can create local variables in the new execution context. The engine adds the new context to the call

stack. The function has no arguments, let's jump right into the body of it.

6. Still lines 3–6. We have a new function declaration. We create a variable `addNumbers` in the local execution context. This important. `addNumbers` exists only in the local execution context. We store a function definition in the local variable named `addNumbers`.

7. Now we're at line 7. We return the content of the variable `addNumbers`. The engine looks for a variable named `addNumbers` and finds it. It's a function definition. Fine, a function can return anything, including a function definition. So we return the definition of `addNumbers`. Anything between the brackets on lines 4 and 5 makes up the function definition. We also remove the local execution context from the call stack.

8. Upon `return`, the local execution context is destroyed. The `addNumbers` variable is no more. The function definition still exists though, it is returned from the function and it is assigned to the variable `adder`; that is the variable we created in step 3.

9. Now we're at line 10. We define a new variable `sum` in the global execution context. Temporary assignment is `undefined`.

10. We need to execute a function next. Which function? The function that is defined in the variable named `adder`. We look it up in the global execution context, and sure enough we find it. It's a function that takes two parameters.

11. Let's retrieve the two parameters, so we can call the function and pass the correct arguments. The first one is the variable `val`, which we defined in step 1, it represents the number `7`, and the second one is the number `8`.

12. Now we have to execute that function. The function definition is outlined lines 3–5. A new local execution context is created. Within the local context two new variables are created: `a` and

`b`. They are respectively assigned the values `7` and `8`, as those were the arguments we passed to the function in the previous step.

13. Line 4. A new variable is declared, named `ret`. It is declared in the local execution context.
14. Line 4. An addition is performed, where we add the content of variable `a` and the content of variable `b`. The result of the addition (`15`) is assigned to the `ret` variable.
15. The `ret` variable is returned from that function. The local execution context is destroyed, it is removed from the call stack, the variables `a`, `b` and `ret` no longer exist.
16. The returned value is assigned to the `sum` variable we defined in step 9.
17. We print out the value of `sum` to the console.

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 `}`.

Finally, a closure

Take a look a the next code and try to figure out what will happen.

```
1: function createCounter() {
2:   let counter = 0
3:   const myFunction = function() {
4:     counter = counter + 1
5:     return counter
6:   }
7:   return myFunction
8: }
```

```
 9: const increment = createCounter()
10: const c1 = increment()
11: const c2 = increment()
12: const c3 = increment()
13: console.log('example increment', c1, c2, c3)
```

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.

1. Lines 1–8. We create a new variable `createCounter` in the global execution context and it get's assigned function definition.
2. Line 9. We declare a new variable named `increment` in the global execution context..
3. Line 9 again. We need call the `createCounter` function and assign its returned value to the `increment` variable.
4. Lines 1–8 . Calling the function. Creating new local execution context.
5. Line 2. Within the local execution context, declare a new variable named `counter` . Number `0` is assigned to `counter` .
6. Line 3–6. Declaring new variable named `myFunction` . The variable is declared in the local execution context. The content of the variable is yet another function definition. As defined in lines 4 and 5.
7. Line 7. Returning the content of the `myFunction` variable. Local execution context is deleted. `myFunction` and `counter` no longer exist. Control is returned to the calling context.
8. Line 9. In the calling context, the global execution context, the value returned by `createCounter` is assigned to `increment` . The variable increment now contains a function definition. The function definition that was returned by `createCounter` . It is no

longer labeled `myFunction`, but it is the same definition. Within the global context, it is labeled `increment`.

9. Line 10. Declare a new variable (`c1`).
10. Line 10 (continued). Look up the variable `increment`, it's a function, call it. It contains the function definition returned from earlier, as defined in lines 4–5.
11. Create a new execution context. There are no parameters. Start execution the function.
12. Line 4. `counter = counter + 1`. Look up the value `counter` in the local execution context. We just created that context and never declare any local variables. Let's look in the global execution context. No variable labeled `counter` here. Javascript will evaluate this as `counter = undefined + 1`, declare a new local variable labeled `counter` and assign it the number `1`, as `undefined` is sort of `0`.
13. Line 5. We return the content of `counter`, or the number `1`. We destroy the local execution context, and the `counter` variable.
14. Back to line 10. The returned value (`1`) gets assigned to `c1`.
15. Line 11. We repeat steps 10–14, `c2` gets assigned `1` also.
16. Line 12. We repeat steps 10–14, `c3` gets assigned `1` also.
17. Line 13. We log the content of variables `c1`, `c2` and `c3`.

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.

```
1: function createCounter() {
 2:   let counter = 0
 3:   const myFunction = function() {
 4:     counter = counter + 1
 5:     return counter
 6:   }
 7:   return myFunction
 8: }
 9: const increment = createCounter()
10: const c1 = increment()
11: const c2 = increment()
12: const c3 = increment()
13: console.log('example increment', c1, c2, c3)
```

1. Lines 1–8. We create a new variable `createCounter` in the global execution context and it get's assigned function definition. Same as above.

2. Line 9. We declare a new variable named `increment` in the global execution context. Same as above.

3. Line 9 again. We need call the `createCounter` function and assign its returned value to the `increment` variable. Same as above.

4. Lines 1–8 . Calling the function. Creating new local execution context. Same as above.

5. Line 2. Within the local execution context, declare a new variable named `counter` . Number `0` is assigned to `counter` . Same as above.

6. Line 3–6. Declaring new variable named `myFunction` . The variable is declared in the local execution context. The content of the variable is yet another function definition. As defined in lines 4 and 5. Now we also create a *closure* and include it as part of the function definition. The closure contains the variables that are in scope, in this case the variable `counter` (with the value of `0`).

7. Line 7. Returning the content of the `myFunction` variable. Local execution context is deleted. `myFunction` and `counter` no longer exist. Control is returned to the calling context. So we are returning the function definition *and its closure*, the backpack with the variables that were in scope when it was created.

8. Line 9. In the calling context, the global execution context, the value returned by `createCounter` is assigned to `increment` . The variable increment now contains a function definition (and closure). The function definition that was returned by `createCounter` . It is no longer labeled `myFunction` , but it is the same definition. Within the global context, it is called `increment` .

9. Line 10. Declare a new variable (`c1`).

10. Line 10 (continued). Look up the variable `increment`, it's a function, call it. It contains the function definition returned from earlier, as defined in lines 4–5. (and it also has a backpack with variables)

11. Create a new execution context. There are no parameters. Start execution the function.

12. Line 4. `counter = counter + 1`. We need to look for the variable `counter`. Before we look in the *local* or *global* execution context, let's look in our backpack. Let's check the closure. Lo and behold, the closure contains a variable named `counter`, its value is `0`. After the expression on line 4, its value is set to `1`. And it is stored in the backpack again. The closure now contains the variable `counter` with a value of `1`.

13. Line 5. We return the content of `counter`, or the number `1`. We destroy the local execution context.

14. Back to line 10. The returned value (`1`) gets assigned to `c1`.

15. Line 11. We repeat steps 10–14. This time, when we look at our closure, we see that the `counter` variable has a value of 1. It was set in step 12 or line 4 of the program. Its value gets incremented and stored as `2` in the closure of the increment function. And `c2` gets assigned `2`.

16. Line 12. We repeat steps 10–14, `c3` gets assigned `3`.

17. Line 13. We log the content of variables `c1`, `c2` and `c3`.

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.

Not so trivial closures

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.

```
let c = 4
const addX = x => n => n + x
const addThree = addX(3)
let d = addThree(c)
console.log('example partial application', d)
```

In case the arrow function throws you off, here is the equivalent.

```
let c = 4
function addX(x) {
  return function(n) {
     return n + x
  }
}
const addThree = addX(3)
let d = addThree(c)
console.log('example partial application', d)
```

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`.

Conclusion

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.*