Javascript Closures

Daniel Rangel

August 10, 2024

Abstract

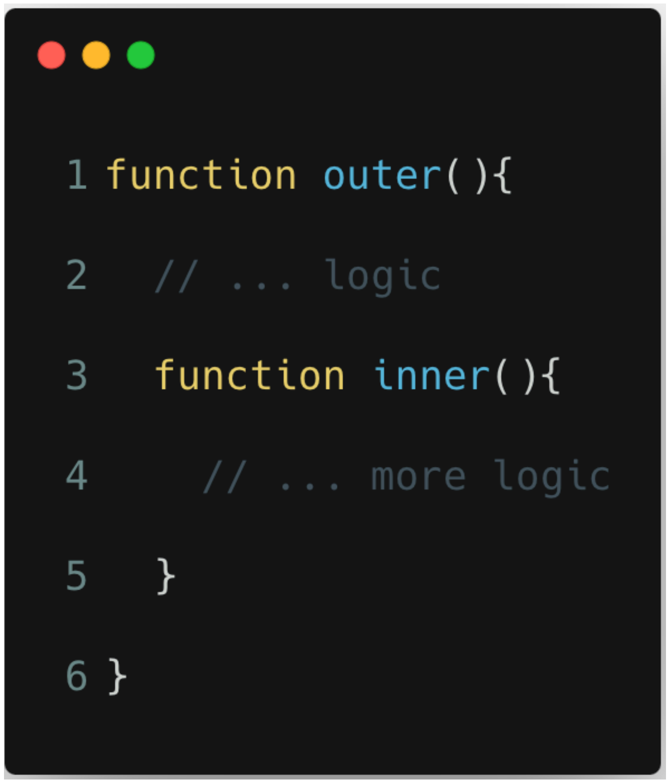
This article is the result of cumulative knowledge through both practical experience and academic research concerning the subject of closures in Javascript. The purpose of the research was to personally develop a better perception of this characteristic of the language. I am not a narcissist who presumes millions of people will be engaging with his article in the future, thus, this article is not aimed to teach anyone other than myself. I do not mean to say that I hope you find this article a trivial vacuum of time. The truth is that, just as any writer, I desire that my content proves practical and didactical to the readers. I only must declare that this is not the primary intent of it.

Closures

Closures are one of the most intricate elements of Javascript. They can be challenging to understand and, sometimes, even more challenging to explain. Yet, in simple terms, a closure may be defined as a "function, inside a function."[1]

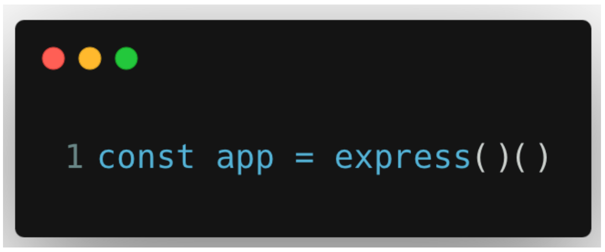
Figure 1: Example of a closure.

[1] MDN Web Docs, "Closures," *Mozilla Developer Network,* accessed August 9, 2024, https://developer.mozilla.org/en-US/docs/Web/JavaScript/Closures.



```
function outer(){
  // ... logic
  function inner(){
    // ... more logic
  }
}
```

Being a Javascript erudite is not a prerequisite to start using closures. All Javascript developers, amateur and expert, have used, or at least read, a closure at some point in their career,[2] as they are almost everywhere.[3] During my initial stages of Javascript learning, I remember encountering the following scenario:



```
const app = express()()
```

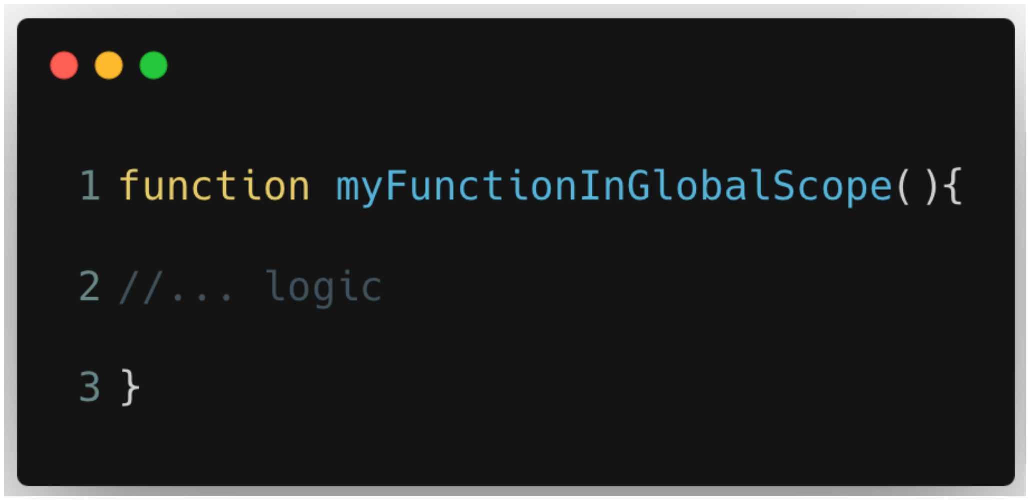
[2] Ved Antani, *Mastering JavaScript : Explore and Master Modern JavaScript Techniques in Order to Build Large-Scale Web Applications,* Community Experience Distilled, (Birmingham, UK: Packt Publishing, 2016), 68.

[3] Antani, *Mastering JavaScript*, 68.

My initial response to the functional expression above was to assume that, for some obscure reason, Javascript permitted certain functions to be invoked twice. I did not recall learning any programming techniques of the same nature in college.[4] Therefore, I was led to adopt the notion that this was a concept particular to Javascript. But invoking the same function twice is not possible in any programming language.[5] In retaking the express example provided above, it is only needed to locate the app's main function in the express library to discover that this is not the representation of a function being called twice. This is only one example of a closure. Perhaps one of NodeJs's most well-known closures, given the popularity of the express library.[6]

Where There Is a Function There Is a Closure

Functions can be thought of as closure creators but closures are not limited to the bundling of two or more functions.



```javascript
function myFunctionInGlobalScope(){
//... logic
}
```

[4] This was attributed to the fact that none of the programming languages I utilized in college allowed closures as they are known in the traditional sense.

[5] I mean, at least not in the ones I am acquainted with.

[6] At the time of writing the express library is among the most popular Javascript packages.

The image above is nothing that a Javascript developer, regardless of their experience tier, has not seen before. Yet, even a simple instance as this one, is a closure.[7] In Javascript we would define this closure, a closure within the global scope. This means that it does not matter where we find ourselves in the code, we have access to it. This concept is true about all closures, not only those instantiated at the global scope. Moreover, the greatness of Javascript allows us to keep the state of a closure in memory through its context. In programming jargon this is called lexical scoping or lexical context.[8] The definition of the lexical context of a function can be said to be all the variables accessible at the time of declaration.[9]

When Are Closures Helpful?

In essence, it could be said that closures are functions declared inside another function.[10] Yet, the benefit of nesting one function inside another must be explained. If a function returns another function, would it not be easier to return the value directly, rather than a function around it? Yes it would. But this would work only when the function's value needs to be accessed directly. Nonetheless, this is not always the case. In addition, returning a function, rather than its computed value directly, means that we can enhance a function by wrapping it with another function. This is often called a function factory in programming.

[7] Antani, *Mastering JavaScript*, 68.

[8] Ved Antani, Simon Timms, and Dan Mantyla, J*avaScript: Functional Programming for JavaScript Developers, Learning Path* (Birmingham: Packt Publishing, 2016), 52.

[9] Antani, *Mastering JavaScript*, 68.

[10] Function factories are helpful when we need to create functions that share similar computations or when we need to instantiate the same function more than once while maintaining each its own state. For more information about closures see Mozilla.

In conclusion, closures are helpful when we need to:

- Maintain state between function calls.
- Create private variables that are not accessible from outside the function.
- Pass around functions that remember the environment where they were created.