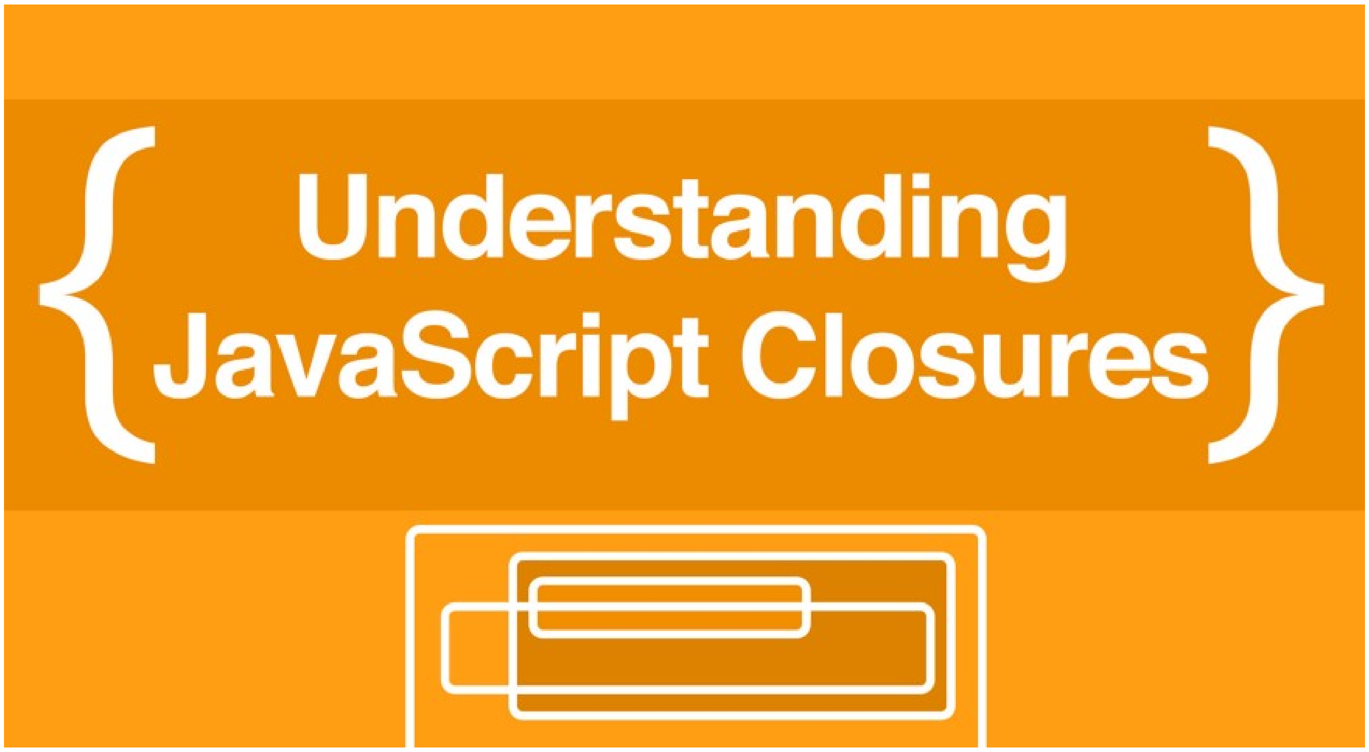
A Beginner's Guide to Understanding JavaScript Closures

Mike Cronin Follow
Oct 13, 2018 · 4 min read ★



Closures are one of those topics that will pop up in most JavaScript interviews. Developers who are just starting out are usually scared of them (I know I was), but there's no reason to be. Like recursion, closures can be over explained in a way that makes them sound complex, but at their core they couldn't be simpler. And I'll prove it by writing one in 6 lines.

A very basic closure

```
function outside() {
  let x = 2
  return function inside(y) {
    return x * y
  }
}
```

Boom. It’s a short example, since a closure is just:

> *A function inside another function that: 1) uses variables from its parent scope and 2) is exposed to the outside world.*

That “exposed” part is a big reason why we even use closures. By only exposing the function, and not the variables it uses, we essentially make them private. But before we talk about what closures are for, let’s talk about making one.

Breaking it down

Of these two functions lets look at `outside` first. In this function’s scope, two things happen: we define a variable `x` to be 2, and we define a function called `inside` and then return the *definition*. Returning it is the “weird” part since we don’t normally return function definitions, we just invoke them. If we call `outside()` , you can see that `inside` ‘s definition is what we get:

```
outside()
//  f inside(y) {
//    return x * y
//  }
```

You may have seen something like this before; a function definition is the return value when we forget to actually invoke a function with `()` :

```
function example() {
  return "My example value"
}

example
// f example() {
//    return "My example value"
// }

// What I MEANT to type was:
example()
// My example value
```

So if the only difference between returning a function's calculated value or its definition is the `()`, what happens if we add another `()`?

```
outside()(4)
// 8
```

We actually invoke the `inside` function now. `outside()` returns a function definition, and then we simply call on that returned value. Here's some pseudo code to illustrate the relationship:

```
// if:
outside() is: function inside(y) {
                return x * y
              }

// then that means:
outside()(5) is: inside(5)
```

That double `()` looks weird though, so normally people do something like:

```
let myClosure = outside()
myClosure(5)
// 10

// remember myClosure is outside(),
// so then this is still the same thing:

myClosure(5) === outside()(5)
// true
```

What about x?

The best part of a closure is that `x`, because we have created a variable that no other part of our program can touch, but we can still use. This is the power of closures: it lets us create values that can't be altered or accessed by any functions other than the closure itself.

The reason this works is because scopes can access values in their parents, but not their children. `x` is defined in the parent scope of `inside` , so `inside` can still access it and multiply it by `y` . However, the only thing that gets exposed to the global scope is the definition of `inside` , *not* the variable `x` . That means in our entire program, the only way we can interact with `x` is by using the closure. Look at this code which **doesn’t** use a closure:

```
let x = 2 // global scope
function inside(y) {
  return x * y  // reaching up to global scope
}
inside(4)
// 8

// but we can change the 'x'
// variable in the global scope and
// alter inside()'s behavior

x = 5
inside(4)
// 20
```

Let’s try to do the same thing, but this time protect our `x` with a closure:

```
function outside() {
  let x = 2
  return function inside(y) {
    return x * y
  }
}

let myClosure = outside()

myClosure(5)
// 10

x = 4
// this is not the same 'x' variable:

myClosure(5)
// 10      // notice how it's still 10
```

`x` is defined in our `outside` function, so anything in the global scope can't reach down into it and change it. When we tried to reassign `x` to 4, that's actually a different `x` variable which now exists separately in our global scope and does not affect how `myClosure` works at all.

As you can see the basics of closures are simple: they just rely on scopes being able to reach up, but not down. Now that you know the basics, try learning about how Immediately Invoked Function Expressions (IIFEs) use closures to create something simple and useful: an untouchable counter.

Happy coding everyone,

Mike