whatthefuck.is · *a closure*

Closures are confusing because they are an “invisible” concept.

When you use an object, a variable, or a function, you do this intentionally. You think: “I’m gonna need a variable here,” and add it to your code.

Closures are different. By the time most people approach closures, they have already used them unknowingly many times — and it is likely that this is true for yourself, too. So learning closures is less about understanding a *new* concept and more about recognizing something you have *already been doing* for a while.

tl;dr

You have a closure when **a function accesses variables defined outside of it**.

For example, this code snippet contains a closure:

```
let users = ['Alice', 'Dan', 'Jessica'];
let query = 'A';
let user = users.filter(user => user.startsWith(query));
```

Notice how `user => user.startsWith(query)` is itself a function. It uses the `query` variable. But the `query` variable is defined *outside* of that function. That’s a closure.

You can stop reading here, if you want. The rest of this article approaches closures in a different way. Instead of explaining what a closure is, it will walk you through the process of *discovering* closures — like the first programmers did in the 1960s.

Step 1: Functions Can Access Outside Variables

To understand closures, we need to be somewhat familiar with variables and functions. In this example, we declare the `food` variable *inside* the `eat` function:

```
function eat() {
  let food = 'cheese';
  console.log(food + ' is good');
}

eat(); // Logs 'cheese is good'
```

But what if we wanted to later change the `food` variable *outside* of the `eat` function? To do this, we can move the `food` variable itself out of our function into the top level:

```
let food = 'cheese'; // We moved it outside

function eat() {
  console.log(food + ' is good');
}
```

This lets us change the `food` "from the outside" any time that we want to:

```
eat(); // Logs 'cheese is good'
food = 'pizza';
eat(); // Logs 'pizza is good'
food = 'sushi';
```

```
eat(); // Logs 'sushi is good'
```

In other words, the `food` variable is no longer *local* to our `eat` function, but our `eat` function nevertheless has no trouble accessing it. **Functions can access variables outside of them.** Stop for a second and make sure that you have no problem with this idea. Once it has settled comfortably in your brain, move to the second step.

Step 2: Wrapping Code in a Function Call

Let's say we have some code:

```
/* A snippet of code */
```

It doesn't matter what that code does. But let's say that **we want to run it twice**.

One way to do it would be to copy and paste it:

```
/* A snippet of code */
/* A snippet of code */
```

Another way to do it would be to use a loop:

```
for (let i = 0; i < 2; i++) {
  /* A snippet of code */
}
```

The third way, which we're particularly interested in today, is to wrap it in a function:

```
function doTheThing() {
  /* A snippet of code */
}

doTheThing();
doTheThing();
```

Using a function gives us the ultimate flexibility because we can run this function any number of times, at any time — and from anywhere in our program.

In fact, **we can even call our new function only *once***, if we wanted to:

```
function doTheThing() {
  /* A snippet of code */
}

doTheThing();
```

Notice how the code above is equivalent to the original code snippet:

```
/* A snippet of code */
```

In other words, **if we take some piece of code, "wrap" that code in a function, and then call that function exactly once, we haven't changed what that code is doing**. There are some exceptions to this rule which we will ignore, but generally speaking this should make sense. Sit on this idea until your brain feels comfortable with it.

Step 3: Discovering Closures

We have traced our way through two different ideas:

- **Functions can access variables defined outside of them.**
- **Wrapping code in a function and calling it once doesn't change the result.**

Now let's see what happens if we combine them.

We'll take our code example from the first step:

```
let food = 'cheese';

function eat() {
  console.log(food + ' is good');
}

eat();
```

Then we'll wrap *this whole example* into a function, which we're going to call once:

```
function liveADay() {
  let food = 'cheese';

  function eat() {
    console.log(food + ' is good');
  }

  eat();
}

liveADay();
```

Read both snippets one more time and make sure that they are equivalent.

This code works! But look closer. Notice the `eat` function is *inside* the `liveADay` function. Is that even allowed? Can we really put a function inside another function?

There are languages in which a code structured this way is *not* valid. For example, this code is not valid in the C language (which doesn't have closures). This means that in C, our second conclusion isn't true — we can't just take some arbitrary piece of code and wrap it in a function. But JavaScript doesn't suffer from that limitation.

Take another good look at this code and notice where `food` is declared and used:

```
function liveADay() {
  let food = 'cheese'; // Declare `food`

  function eat() {
    console.log(food + ' is good'); // Read `food`
  }

  eat();
}

liveADay();
```

Let's go through this code together — step by step. First, we declare the `liveADay` function at the top level. We immediately call it. It has a `food` local variable. It also contains an `eat` function. Then it calls that `eat` function. Because `eat` is inside of `liveADay`, it "sees" all of its variables. This is why it can read the `food` variable.

This is called a closure.

We say that there is a closure when a function (such as `eat`) reads or writes a variable (such as `food`) that is declared outside of it (such as in `liveADay`).

Take some time to re-read this, and make sure you can trace this in the code.

Here is an example we've introduced in the tl;dr section:

```
let users = ['Alice', 'Dan', 'Jessica'];
let query = 'A';
let user = users.filter(user => user.startsWith(query));
```

It may be easier to notice the closure if we rewrite it with a function expression:

```
let users = ['Alice', 'Dan', 'Jessica'];
// 1. The query variable is declared outside
let query = 'A';
let user = users.filter(function(user) {
  // 2. We are in a nested function
  // 3. And we read the query variable (which is declared outside!)
  return user.startsWith(query);
});
```

Whenever a function accesses a variable that is declared outside of it, we say it is a closure. The term itself is used a bit loosely. Some people will refer to the *nested function itself* as "the closure" in this example. Others might refer to the *technique* of accessing the outside variables as the closure. Practically, it doesn't matter.

A Ghost of a Function Call

Closures might seem deceptively simple now. This doesn't mean they're without their own pitfalls. The fact that a function may read and write variables outside has rather deep consequences if you really think about it. For example, this means that these variables will "survive" for as long as the nested function may be called:

```
function liveADay() {
  let food = 'cheese';

  function eat() {
    console.log(food + ' is good');
  }

  // Call eat after five seconds
  setTimeout(eat, 5000);
}

liveADay();
```

Here, `food` is a local variable inside the `liveADay()` function call. It's tempting to think it "disappears" after we exit `liveADay`, and it won't come back to haunt us.

However, inside of `liveADay` we tell the browser to call `eat` in five seconds. And `eat` reads the `food` variable. **So the JavaScript engine needs to keep the `food` variable from that particular `liveADay()` call available until `eat` has been called.**

In that sense, we can think of closures as of "ghosts" or "memories" of the past function calls. Even though our `liveADay()` function call has long finished, its variables must continue to exist for as long as the nested `eat` function may still be called. Luckily, JavaScript does that for us, so we don't need to think about it.

Why "Closures"?

Finally, you might be wondering why closures are called that way. The reason is mostly historical. A person familiar with the computer science jargon might say that an expression like `user => user.startsWith(query)` has an "open binding". In other words, it is clear from it what the `user` is (a parameter), but it is not clear what `query` is in isolation. When we say "actually, `query` refers to the variable declared outside", we are "closing" that open binding. In other words, we get a *closure*.

Not all languages implement closures. For example, in some languages like C, it is not allowed to nest functions at all. As a result, a function may only access its own local variables or global variables, but there is never a situation in which it can access a parent function's local variables. Naturally, that limitation is painful.

There are also languages like Rust which implement closures, but have a separate syntax for closures and regular functions. So if you want to read a variable from outside a function, you would have to opt into that in Rust. This is because under the hood, closures may require the engine to keep the outer variables (called "the environment") around even after the function call. This overhead is acceptable in JavaScript, but it can be a performance concern for the very low-level languages.

And with that, I hope you can get a closure on the concept of closures!

> *If you prefer a more visual approach to the JavaScript fundamentals, check out* ***Just JavaScript****. It is my illustrated course in collaboration with* ***Maggie Appleton****.*

composition →