JavaScript Scope and Closures

Zell Liew Aug 28, 2017 Jan 16, 2019

Scopes and closures are important in JavaScript. But, they were confusing for me when I first started. Here's an explanation of scopes and closures to help you understand what they are.

Let's start with scopes.

(#aa-scope) Scope

A scope in JavaScript defines what variables you have access to. There are two kinds of scope – global scope and local scope.

(#aa-global-scope) Global scope

If a variable is declared outside all functions or curly braces (`{}`), it is said to be defined in the **global scope**.

Hey! This is true only with JavaScript in web browsers. You declare global variables in Node.js differently, but we won't go into Node.js in this article.

JavaScript

```
const globalVariable = 'some value'
```

Once you've declared a global variable, you can use that variable anywhere in your code, even in functions.

JavaScript

```
const hello = 'Hello CSS-Tricks Reader!'

function sayHello () {
  console.log(hello)
}
```

```
console.log(hello) // 'Hello CSS-Tricks Reader!'
sayHello() // 'Hello CSS-Tricks Reader!'
```

Although you can declare variables in the global scope, it is advised not to. This is because there is a chance of naming collisions, where two or more variables are named the same. If you declared your variables with `const` or `let`, you would receive an error whenever a name collision happens. This is undesirable.

JavaScript

```javascript
// Don't do this!
let thing = 'something'
let thing = 'something else' // Error, thing has already been declared
```

If you declare your variables with `var`, your second variable overwrites the first one after it is declared. This also undesirable as you make your code hard to debug.

JavaScript

```javascript
// Don't do this!
var thing = 'something'
var thing = 'something else' // perhaps somewhere totally different in your code
console.log(thing) // 'something else'
```

So, you should always declare local variables, not global variables.

(#aa-local-scope) Local Scope

Variables that are usable only in a specific part of your code are considered to be in a local scope. These variables are also called **local variables**.

In JavaScript, there are two kinds of local scope: function scope and block scope.

Let's talk about function scopes first.

(#aa-function-scope) Function scope

When you declare a variable in a function, you can access this variable only within the function. You can't get this variable once you get out of it.

In the example below, the variable `hello` is in the `sayHello` scope:

JavaScript

```javascript
function sayHello () {
  const hello = 'Hello CSS-Tricks Reader!'
  console.log(hello)
}

sayHello() // 'Hello CSS-Tricks Reader!'
console.log(hello) // Error, hello is not defined
```

(#aa-block-scope) Block scope

When you declare a variable with `const` or `let` within a curly brace (`{}`), you can access this variable only within that curly brace.

In the example below, you can see that `hello` is scoped to the curly brace:

JavaScript

```
{
  const hello = 'Hello CSS-Tricks Reader!'
  console.log(hello) // 'Hello CSS-Tricks Reader!'
}

console.log(hello) // Error, hello is not defined
```

The block scope is a subset of a function scope since functions need to be declared with curly braces (unless you're using arrow functions (https://zellwk.com/blog/es6/#arrow-functions) with an implicit return).

(#aa-function-hoisting-and-scopes) Function hoisting and scopes

Functions, when declared with a function declaration, are always hoisted to the top of the current scope. So, these two are equivalent:

JavaScript

```
// This is the same as the one below
sayHello()
function sayHello () {
  console.log('Hello CSS-Tricks Reader!')
}

// This is the same as the code above
function sayHello () {
  console.log('Hello CSS-Tricks Reader!')
}
sayHello()
```

When declared with a function expression, functions are not hoisted to the top of the current scope.

```javascript
sayHello() // Error, sayHello is not defined
const sayHello = function () {
  console.log(aFunction)
}
```

Because of these two variations, function hoisting can potentially be confusing, and should not be used. Always declare your functions before you use them.

(#aa-functions-do-not-have-access-to-each-others-scopes) Functions do not have access to each other’s scopes

Functions do not have access to each other’s scopes when you define them separately, even though one function may be used in another.

In this example below, `second` does not have access to `firstFunctionVariable`.

```javascript
function first () {
  const firstFunctionVariable = `I'm part of first`
}

function second () {
  first()
  console.log(firstFunctionVariable) // Error, firstFunctionVariable is not defined
}
```

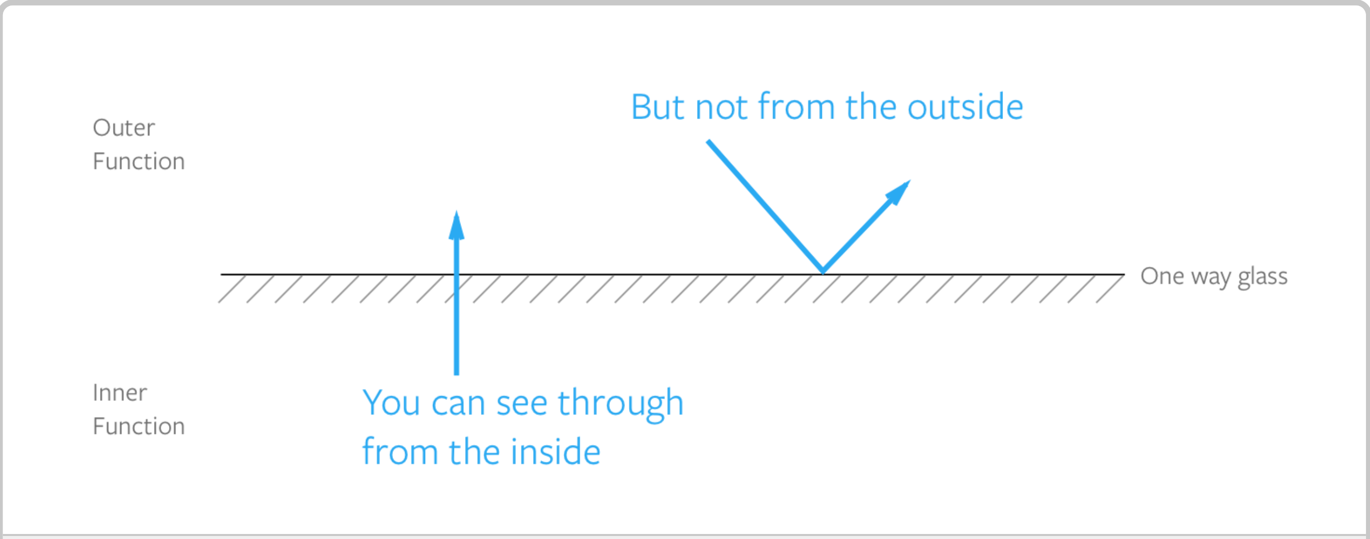
(#aa-nested-scopes) Nested scopes

When a function is defined in another function, the inner function has access to the outer function’s variables. This behavior is called **lexical scoping**.

However, the outer function does not have access to the inner function’s variables.

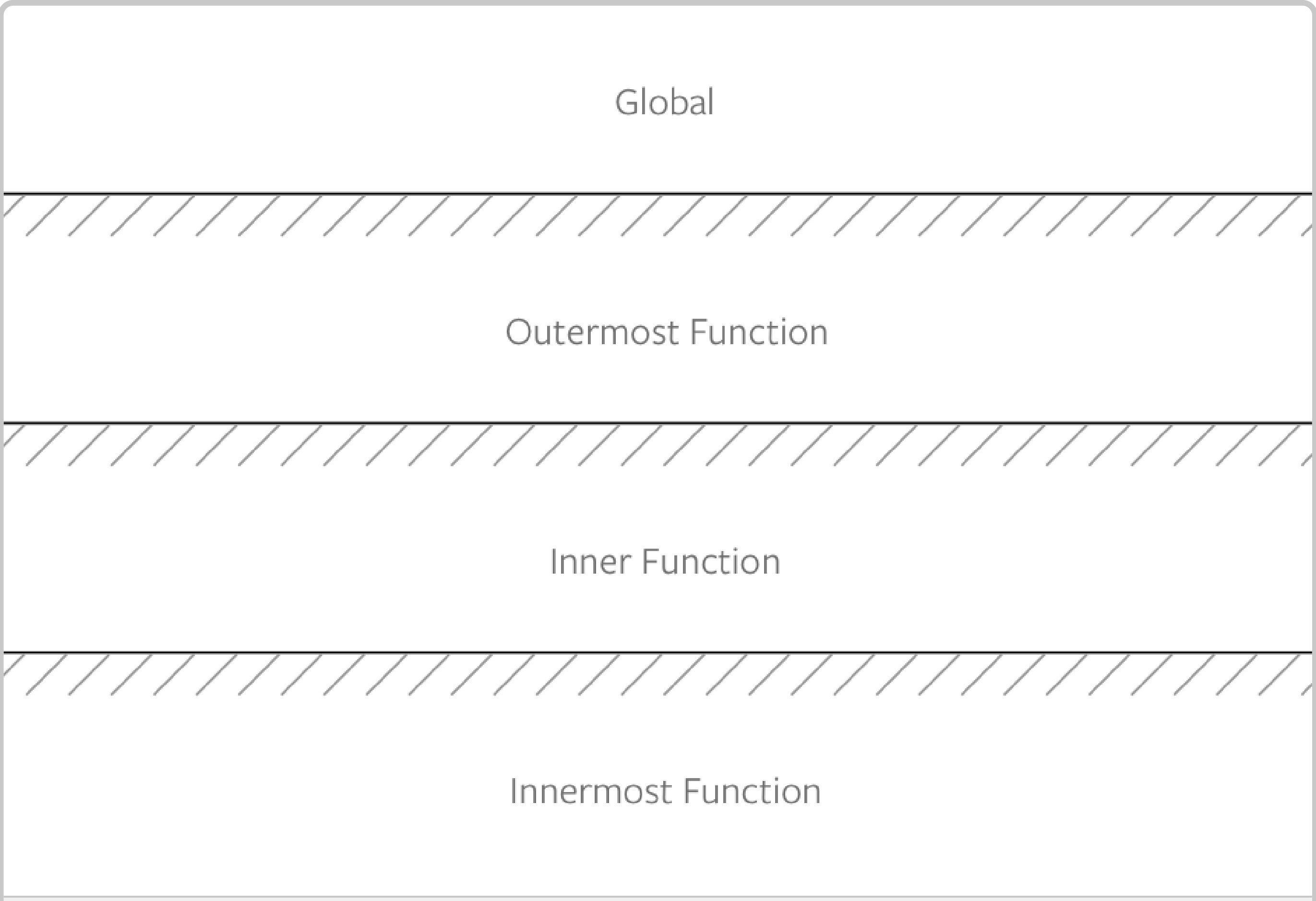
```javascript
function outerFunction () {
  const outer = `I'm the outer function!`

  function innerFunction() {
    const inner = `I'm the inner function!`
    console.log(outer) // I'm the outer function!
  }

  console.log(inner) // Error, inner is not defined
}
```

To visualize how scopes work, you can imagine one-way glass. You can see the outside, but people from the outside cannot see you.



Scopes in functions behave like a one-way-glass. You can see the outside, but people outside can't see you

If you have scopes within scopes, visualize multiple layers of one-way glass.



Multiple layers of functions mean multiple layers of one-way glass

After understanding everything about scopes so far, you're well primed to figure out what closures are.

(#aa-closures) Closures

Whenever you create a function within another function, you have created a closure. The inner function is the closure. This closure is usually returned so you can use the outer function's variables at a later time.

```javascript
function outerFunction () {
  const outer = `I see the outer variable!`

  function innerFunction() {
    console.log(outer)
  }

  return innerFunction
}

outerFunction()() // I see the outer variable!
```

Since the inner function is returned, you can also shorten the code a little by writing a return statement while declaring the function.

```javascript
function outerFunction () {
  const outer = `I see the outer variable!`

  return function innerFunction() {
    console.log(outer)
  }
}

outerFunction()() // I see the outer variable!
```

Since closures have access to the variables in the outer function, they are usually used for two things:

1. To control side effects
2. To create private variables

(#aa-controlling-side-effects-with-closures) Controlling side effects with closures

Side effects happen when you do something in aside from returning a value from a function. Many things can be side effects, like an Ajax request, a timeout or even a `console.log` statement:

```javascript
function (x) {
  console.log('A console.log is a side effect!')
}
```

When you use closures to control side effects, you're usually concerned with ones that can mess up your code flow like Ajax or timeouts.

Let's go through this with an example to make things clearer.

Let's say you want to make a cake for your friend's birthday. This cake would take a second to make, so you wrote a function that logs `made a cake` after one second.

Hey!

I'm using ES6 arrow functions (https://zellwk.com/blog/es6/#arrow-functions) here to make the example shorter, and easier to understand.

```javascript
function makeCake() {
  setTimeout(_ => console.log(`Made a cake`), 1000)
}
```

As you can see, this cake making function has a side effect: a timeout.

Let's further say you want your friend to choose a flavor for the cake. To do so, you can write add a flavor to your `makeCake` function.

```javascript
function makeCake(flavor) {
  setTimeout(_ => console.log(`Made a ${flavor} cake!`), 1000)
}
```

When you run the function, notice the cake gets made immediately after one second.

JavaScript

```
makeCake('banana')
// Made a banana cake!
```

The problem here is that you don't want to make the cake immediately after knowing the flavor. You want to make it later when the time is right.

To solve this problem, you can write a `prepareCake` function that stores your flavor. Then, return the `makeCake` closure within `prepareCake`.

From this point on, you can call the returned function whenever you want to, and the cake will be made within a second.

JavaScript

```
function prepareCake (flavor) {
  return function () {
    setTimeout(_ => console.log(`Made a ${flavor} cake!`), 1000)
  }
}

const makeCakeLater = prepareCake('banana')

// And later in your code...
makeCakeLater()
// Made a banana cake!
```

That's how closures are used to reduce side effects – you create a function that activates the inner closure at your whim.

(#aa-private-variables-with-closures) Private variables with closures

As you know by now, variables created in a function cannot be accessed outside the function. Since they can't be accessed, they are also called private variables.

However, sometimes you need to access such a private variable. You can do so with the help of closures.

JavaScript

```
function secret (secretCode) {
  return {
    saySecretCode () {
      console.log(secretCode)
    }
  }
}
```

```
const theSecret = secret('CSS Tricks is amazing')
theSecret.saySecretCode()
// 'CSS Tricks is amazing'
```

`saySecretCode` in this example above is the only function (a closure) that exposes the `secretCode` outside the original secret function. As such, it is also called a **privileged function**.

(#aa-debugging-scopes-with-devtools) Debugging scopes with DevTools

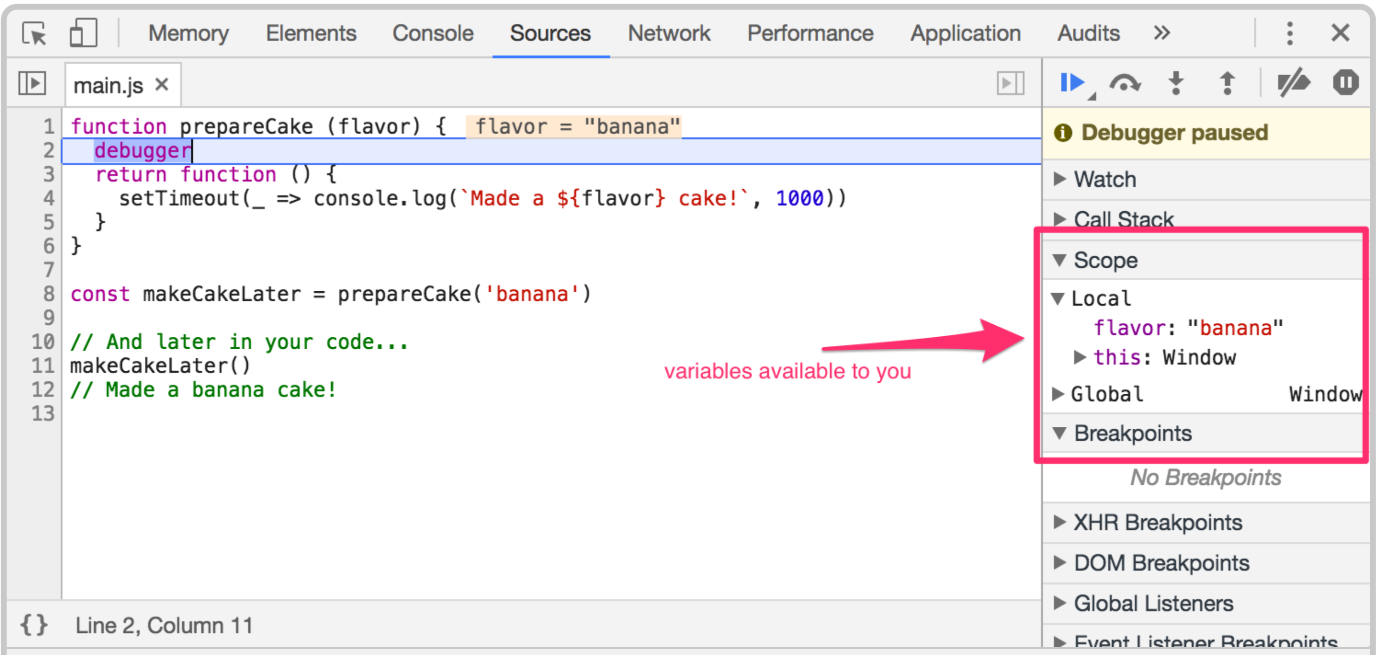
Chrome and Firefox's DevTools make it simple for you to debug variables you can access in the current scope. There are two ways to use this functionality.

The first way is to add the `debugger` keyword in your code. This causes JavaScript execution in browsers to pause so you can debug.

Here's an example with the `prepareCake`:

```javascript
function prepareCake (flavor) {
  // Adding debugger
  debugger
  return function () {
    setTimeout(_ => console.log(`Made a ${flavor} cake!`), 1000)
  }
}

const makeCakeLater = prepareCake('banana')
```

If you open your DevTools and navigate to the Sources tab in Chrome (or Debugger tab in Firefox), you would see the variables available to you.

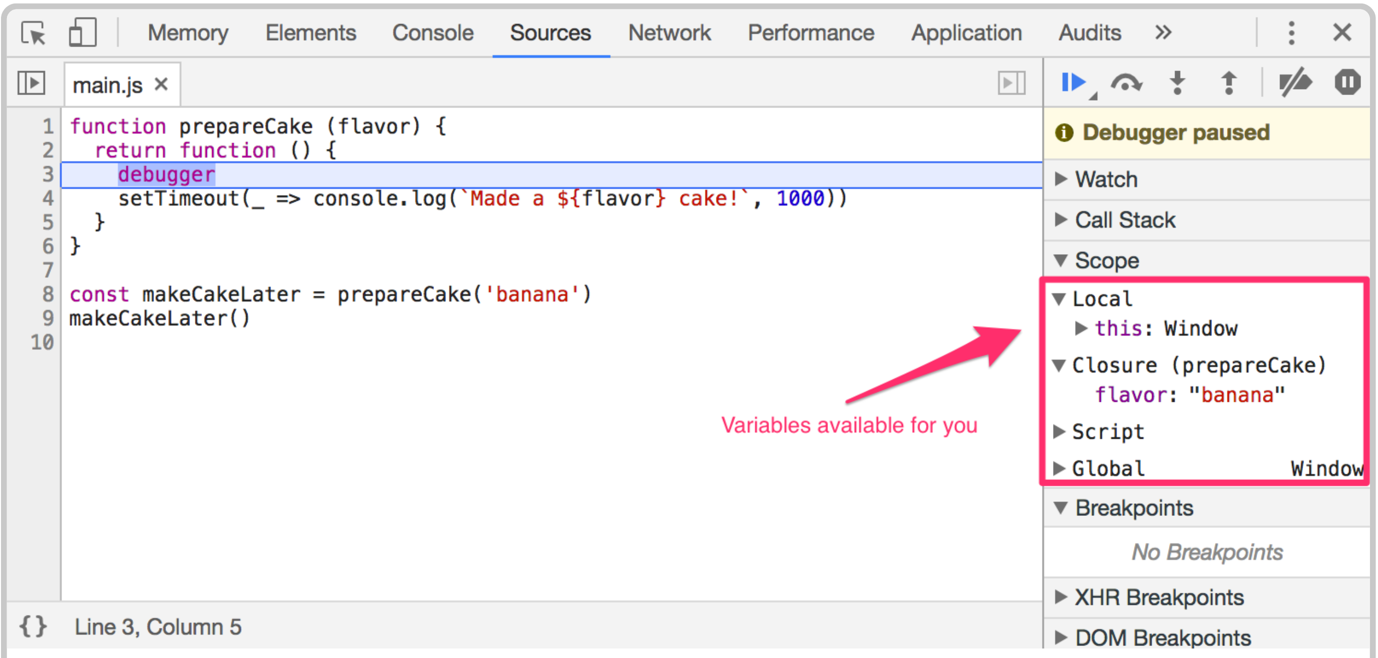


```
function prepareCake (flavor) {  flavor = "banana"
  debugger
  return function () {
    setTimeout(_ => console.log(`Made a ${flavor} cake!`, 1000))
  }
}

const makeCakeLater = prepareCake('banana')

// And later in your code...
makeCakeLater()
// Made a banana cake!
```

Debugging prepareCake's scope

You can also shift the debugger keyword into the closure. Notice how the scope variables changes this time:

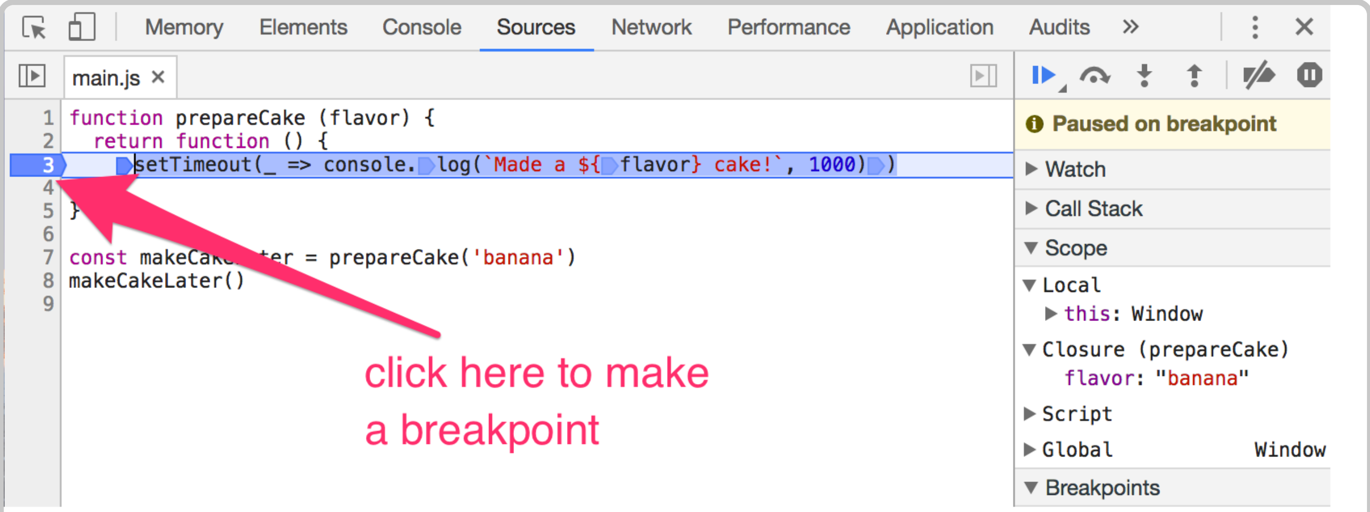
```javascript
function prepareCake (flavor) {
  return function () {
    // Adding debugger
    debugger
    setTimeout(_ => console.log(`Made a ${flavor} cake!`), 1000)
  }
}

const makeCakeLater = prepareCake('banana')
```



```
function prepareCake (flavor) {
  return function () {
    debugger
    setTimeout(_ => console.log(`Made a ${flavor} cake!`, 1000))
  }
}

const makeCakeLater = prepareCake('banana')
makeCakeLater()
```

Debugging the closure scope

The second way to use this debugging functionality is to add a breakpoint to your code directly in the sources (or debugger) tab by clicking on the line number.



```
function prepareCake (flavor) {
  return function () {
    setTimeout(_ => console.log(`Made a ${flavor} cake!`, 1000))
  }
}

const makeCakeLater = prepareCake('banana')
makeCakeLater()
```

Debugging scopes by adding breakpoints

(#aa-wrapping-up) Wrapping up

Scopes and closures aren't incredibly hard to understand. They're pretty simple once you know how to see them through a one-way glass.

When you declare a variable in a function, you can only access it in the function. These variables are said to be scoped to the function.

If you define any inner function within another function, this inner function is called a closure. It retains access to the variables created in the outer function.

Feel free to pop by and ask any questions you have. I'll get back to you as soon as I can.

If you liked this article, you may also like other front-end-related articles I write on my blog (https://zellwk.com/blog/) and my newsletter (https://zellwk.com/newsletter/css-tricks/) . I

also have a brand new (and free!) email course: JavaScript Roadmap (https://jsroadmap.com/)
.