27 MAY 2020 / #JAVASCRIPT

JavaScript Closure Tutorial – With JS Closure Example Code

Anchal Nigam

Read more posts by this author.

heard this term before. When I started my journey with JavaScript, I encountered closures often. And I think they're one of the most important and interesting concepts in JavaScript.

You don't think they're interesting? This often happens when you don't understand a concept – you don't find it interesting. (I don't know if this happens to you or not, but this is the case with me).

So in this article, I will try to make closures interesting for you.

Before going into the world of closures, let's first understand **lexical scoping**. If you already know about it, skip the next part. Otherwise jump into it to better understand closures.

Lexical Scoping

You may be thinking – I know local and global scope, but what the heck is lexical scope? I reacted the same way when I heard this term. Not to worry! Let's take a closer look.

It's simple like other two scopes:

```
function greetCustomer() {
    var customerName = "anchal";
    function greetingMsg() {
      console.log("Hi! " + customerName); // Hi! anchal
    }
   greetingMsg();
}
```

the outer function's variable. This is lexical scoping, where the scope and value of a variable is determined by where it is defined/created (that is, its position in the code). Got it?

I know that last bit might have confused you. So let me take you deeper. Did you know that lexical scoping is also known as **static scoping**? Yes, that's its other name.

There is also **dynamic scoping**, which some programming languages support. Why have I mentioned dynamic scoping? Because it can help you better understand lexical scoping.

Let’s look at some examples:

```
function greetingMsg() {
  console.log(customerName);// ReferenceError: customerName is not define
}

function greetCustomer() {
   var customerName = "anchal";
   greetingMsg();
}

greetCustomer();
```

Do you agree with the output? Yes, it will give a reference error. This is because both functions don’t have access to each other’s scope, as they are defined separately.

Let’s look at another example:

```
  console.log(number1 + number2);
}

function addNumbersGenerate() {
  var number2 = 10;
  addNumbers(number2);
}

addNumbersGenerate();
```

The above output will be 20 for a dynamically scoped language. Languages that support lexical scoping will give `referenceError: number2 is not defined`. Why?

Because in dynamic scoping, searching takes place in the local function first, then it goes into the function that *called* that local function. Then it searches in the function that called *that* function, and so on, up the call stack.

Its name is self explanatory – "dynamic" means change. The scope and value of variable can be different as it depends on from where the function is called. The meaning of a variable can change at runtime.

Got the gist of dynamic scoping? If yes, then just remember that lexical scoping is its opposite.

In lexical scoping, searching takes place in the local function first, then it goes into the function inside which *that* function is defined. Then it searches in the function inside which *that* function is defined and so on.

So, **lexical** or **static scoping** means the scope and value of a variable is determined from where it is defined. It doesn't change.

Look again at the above example and try to figure out the output on your own. Just one twist – declare `number2` at the top:

```
var number2 = 2;
function addNumbers(number1) {
  console.log(number1 + number2);
}

function addNumbersGenerate() {
  var number2 = 10;
  addNumbers(number2);
}

addNumbersGenerate();
```

Do you know what the output will be?

Correct – it's 12 for lexically scoped languages. This is because first, it looks into an `addNumbers` function (innermost scope) then it searches inwards, where this function is defined. As it gets the `number2` variable, meaning the output is 12.

You may be wondering why I have spent so much time on lexical scoping here. This is a closure article, not one about lexical scoping. But if you don't know about lexical scoping then you will not understand closures.

Why? You will get your answer when we look at the definition of a closure. So let's get into the track and get back to closures.

What is a Closure?

> Closure is created when an inner function has access to its outer function variables and arguments. The inner function has access to –
> 1. Its own variables.
> 2. Outer function's variables and arguments.
> 3. Global variables.

Wait! Is this the definition of a closure or lexical scoping? Both definitions look the same. How they are different?

Well, that's why I defined lexical scoping above. Because closures are related to lexical/static scoping.

Let’s again look at its other definition that will tell you how closures are different.

> Closure is when a function is able to access its lexical scope, even when that function is executing outside its lexical scope.

Or,

> Inner functions can access its parent scope, even after the parent function is already executed.

Confused? Don't worry if you haven't yet gotten the point. I have examples to help you better understand. Let’s modify the first example of lexical scoping:

```
function greetCustomer() {
  const customerName = "anchal";
  function greetingMsg() {
```

```
  return greetingMsg;
}

const callGreetCustomer = greetCustomer();
callGreetCustomer(); // output - Hi! anchal
```

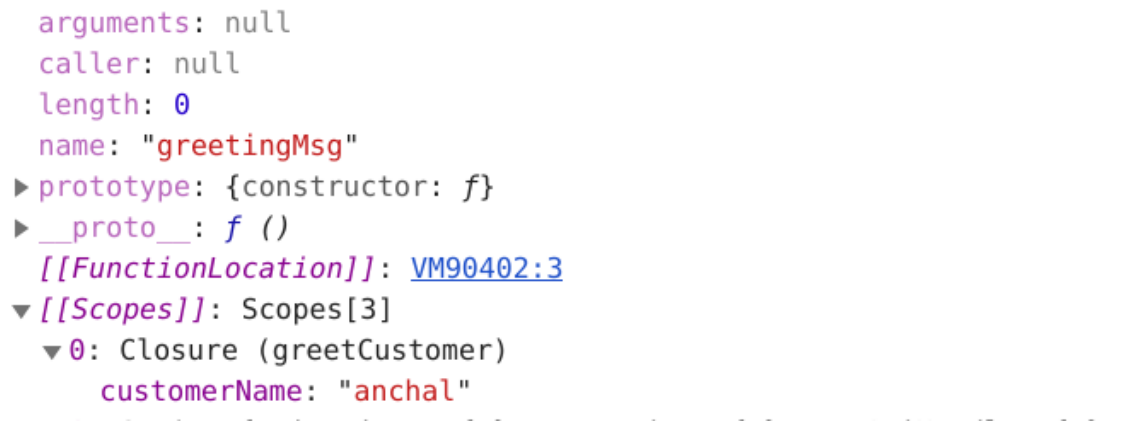
The difference in this code is that we are returning the inner function and executing it later. In some programming languages, the local variable exists during the function's execution. But once the function is executed, those local variables don't exist and they will not be accessible.

Here, however, the scene is different. After the parent function is executed, the inner function (returned function) can still access the parent function's variables. Yes, you guessed right. Closures are the reason.

The inner function preserves its lexical scope when the parent function is executing and hence, later that inner function can access those variables.

To get a better feel for it, let's use the `dir()` method of the console to look into the list of the properties of `callGreetCustomer` :

```
console.dir(callGreetCustomer);
```



From the above image, you can see how the inner function preserves its parent scope (`customerName`) when `greetCustomer()` is executed. And later on, it used `customerName` when `callGreetCustomer()` was executed.

I hope this example helped you better understand the above definition of a closure. And maybe now you find closures a bit more fun.

So what next? Let's make this topic more interesting by looking at different examples.

Examples of closures in action

```
function counter() {
  let count = 0;
  return function() {
    return count++;
  };
}

const countValue = counter();
```

```
countValue(); // 1
countValue(); // 2
```

Every time you call `countValue`, the count variable value is incremented by 1. Wait – did you think that the value of count is 0?

Well, that would be wrong as a closure doesn't work with a value. It stores the **reference** of the variable. That's why, when we update the value, it reflects in the second or third call and so on as the closure stores the reference.

Feeling a bit clearer now? Let's look at another example:

```
function counter() {
  let count = 0;
  return function () {
    return count++;
  };
}

const countValue1 = counter();
const countValue2 = counter();
countValue1();  // 0
countValue1();  // 1
countValue2();   // 0
countValue2();   // 1
```

I hope you guessed the right answer. If not, here is the reason. As `countValue1` and `countValue2`, both preserve their own lexical scope. They have independent lexical environments. You can use `dir()` to check the `[[scopes]]` value in both the cases.

Let's look at a third example:

This one's a bit different. In it, we have to write a function to achieve the output:

```
const addNumberCall = addNumber(7);
addNumberCall(8) // 15
addNumberCall(6) // 13
```

Simple. Use your newly-gained closure knowledge:

```
function addNumber(number1) {
  return function (number2) {
    return number1 + number2;
  };
}
```

Now let's look at some tricky examples:

```
function countTheNumber() {
  var arrToStore = [];
  for (var x = 0; x < 9; x++) {
    arrToStore[x] = function () {
      return x;
    };
  }
  return arrToStore;
}

const callInnerFunctions = countTheNumber();
callInnerFunctions[0]() // 9
callInnerFunctions[1]() // 9
```

Every array element that stores a function will give you an output of 9. Did you guess right? I hope so, but still let me tell you the reason. This is because of the closure's behavior.

The closure stores the **reference**, not the value. The first time the loop runs, the value of x is 0. Then the second time x is 1, and so on. Because the closure stores the reference, every time the loop runs it's changing the value of x. And at last, the value of x will be 9. So `callInnerFunctions[0]()` gives an output of 9.

But what if you want an output of 0 to 8? Simple! Use a closure.

Think about it before looking at the solution below:

```
function callTheNumber() {
  function getAllNumbers(number) {
    return function() {
      return number;
    };
  }
  var arrToStore = [];
  for (var x = 0; x < 9; x++) {
    arrToStore[x] = getAllNumbers(x);
  }
  return arrToStore;
}

const callInnerFunctions = callTheNumber();
console.log(callInnerFunctions[0]()); // 0
console.log(callInnerFunctions[1]()); // 1
```

Here, we have created separate scope for each iteration. You can use `console.dir(arrToStore)` to check the value of x in `[[scopes]]` for different array elements.

That's it! I hope you can now say that you find closures interesting.

To read my other articles, check out my profile here.

Donate

Full Stack Developer

JavaScript Substring

What Does a VPN Do?

Docker Remove Image

Tar GZ

What is a CSV File?

Correlation VS Causation

Permutation VS Combination

Computer Programming

JWT

How to Find a Square Root

Python List Append

What is Chromium?

Smoke Testing

Clear History

Incognito Mode

Linux Add User

MD5 Hash

What is Cached Data?

Completing the Square

Error 403 Forbidden

CSS Inline Style

Our Nonprofit

About Alumni Network Open Source Shop Support Sponsors Academic Honesty

Code of Conduct Privacy Policy Terms of Service Copyright Policy