DZone

What Are Closures in JavaScript?

by Dhananjay Kumar MVB · Aug. 29, 16 · Web Dev Zone

A JavaScript closure is a function which remembers the environment in which it was created. We can think of it as an object with one method and private variables. JavaScript closures are a special kind of object which contains the function and the local scope of the function with all the variables (environment) when the closure was created.

> JavaScript Closure is a special kind of object which contains function and the environment in which function was created. Here environment stands for local scope of the function and it's all variable at the time of closure creation.

To understand closures, first we need to understand SCOPING in JavaScript. We can create a variable or a function in three levels of scoping:

1. Global Scope
2. Function or local scope
3. Lexical scope

I have written in details about scoping here, but let's take a brief walkthrough of scoping before getting into closures.

Scopes in JavaScript

As soon as we create a variable, it is in a Global Scope. So, if we have created a variable which is not inside any function, it is in a global scope.

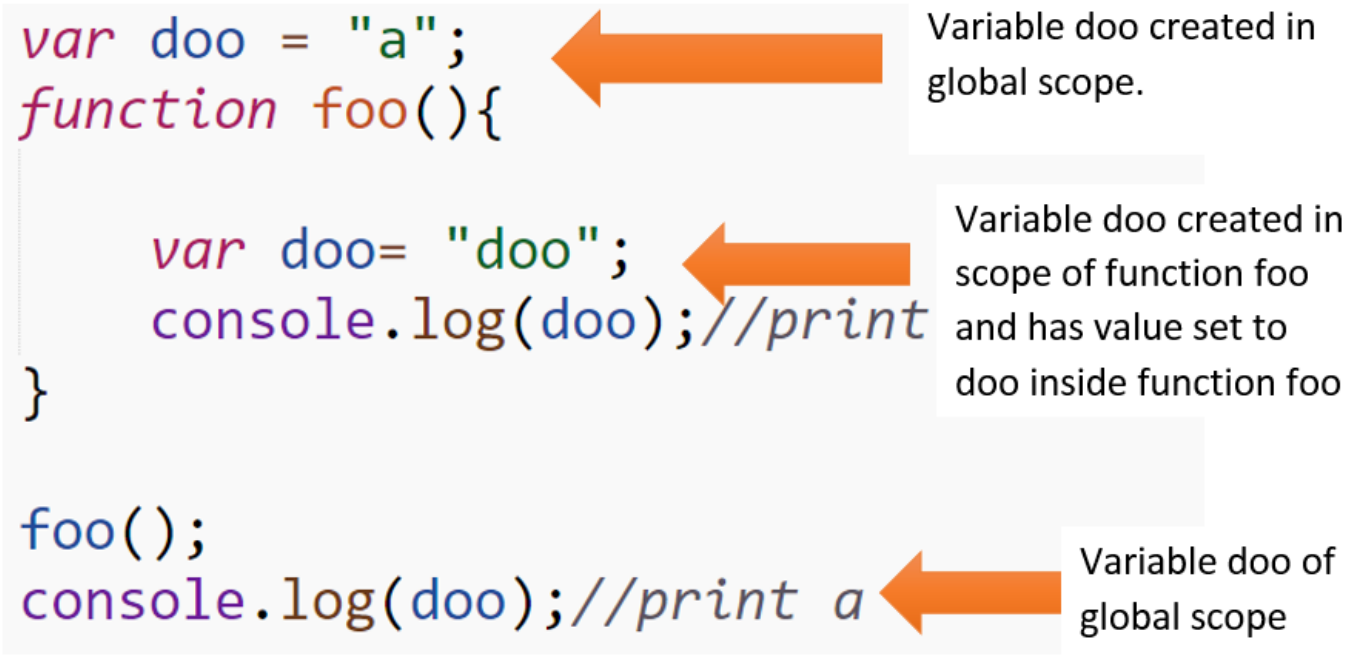
```
1  var foo = "foo";
2  console.log(foo);
```

If something is in a global scope, we can access it anywhere—which makes it our friend and enemy at the same time. Putting everything in a global scope is never a good idea, because it may cause namespace conflicts among other issues. If something is in a global scope, it could be accessed from anywhere and if there are variables with the same name in a function, this can cause conflicts.

Any variable or function which is not inside a global scope is inside a functional or local scope. Consider the listing below:

```
function foo() {
  var doo = "doo";
  console.log(doo);
}
foo();
```

We have created a variable "doo" which is inside the function scope of the function "foo". The lifetime of the variable doo is local to function foo and it cannot be accessed outside the function foo. This is called local scoping in JavaScript. Let's consider the code shown in the diagram below:



Here we have created a variable in the function foo with the same name as a variable from the global scope, so now we have two variables. We should keep in mind that these variables are two different variables with their own respective life times. Outside the function, variable doo with value a is accessible, however inside the function foo, variable doo with value doo exists. For reference, the above code is given below:

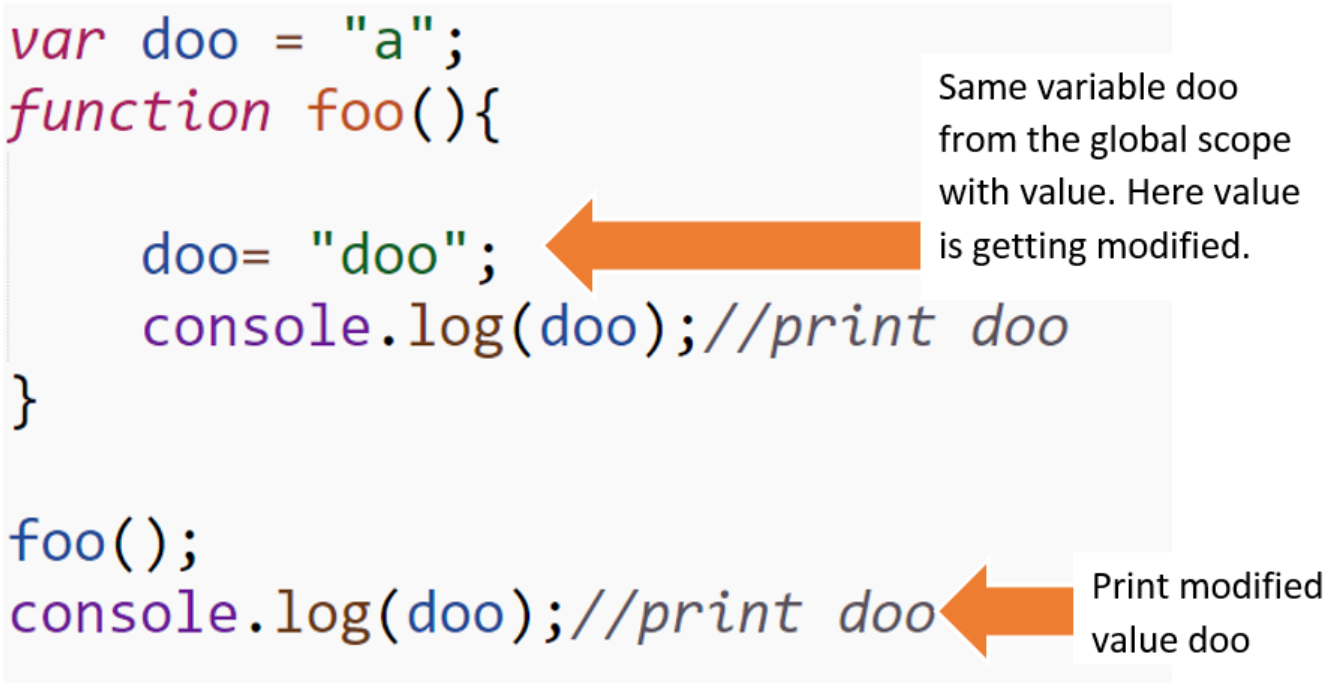
```
var doo = "a";
function foo() {
  var doo = "doo";
  console.log(doo); //print doo
}
foo();
console.log(doo);//print a
```

Let's tweak the above code snippet a bit as shown in listing below:

```
var doo = "a";
function foo() {
  doo = "doo";
```

```
  console.log(doo);//print doo
}
foo();
console.log(doo);//print doo
```

Now we do not have two scopes for the variable doo. Inside the function foo, the variable doo created in the global scope is getting modified. We are not recreating the variable doo inside foo, but instead modifying the existing variable doo from the global scope.



While creating the variable in the local or functional scope, we must use keyword var to create the variable. Otherwise, the variable will be created in the global scope, or if the variable already exists in the global scope, it would be modified.

In JavaScript, we can have a function inside a function. There could be nested functions of any level, meaning we can have any number of functions nested inside each other. Consider the listing below:

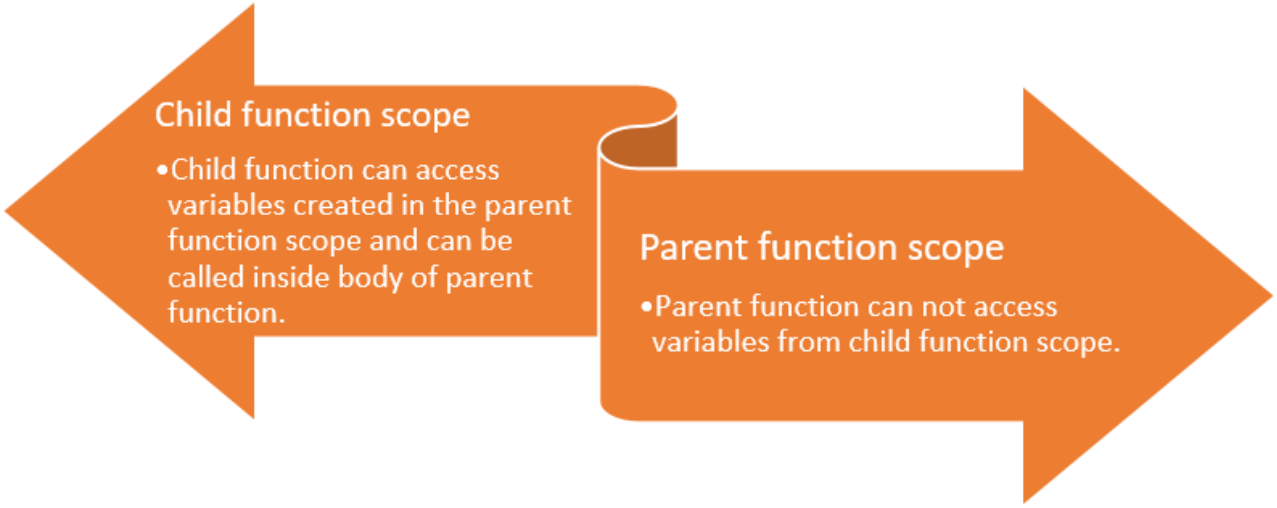
```
function foo() {
  var f = "foo";
  function doo() {
    console.log(f);
  }
  doo();
}
foo();//print foo
```

Essentially, in the snippet above, we have a function doo which is created inside function foo, and it does not have any of its own variables. Function foo creates a local variable f and it is accessible inside function doo. Function doo is the inner function of function foo and it can access the variables

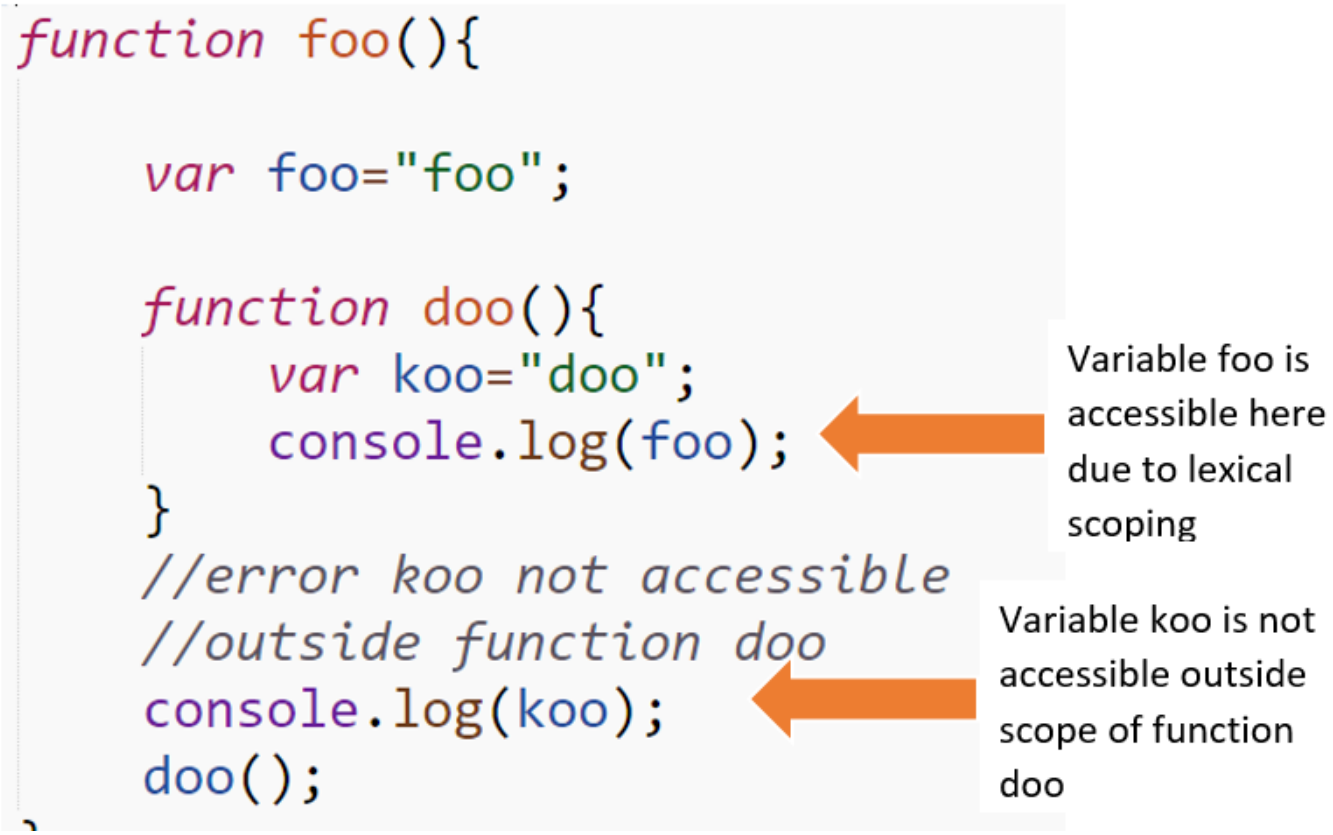
of function foo. Also, function doo can be called inside the body of the function foo. Function doo can access the variables declared in the parent function and this is due to the Lexical Scoping of JavaScript.

There are two levels of scoping here:

1. Parent function foo scope
2. Child function doo scope



Variables created inside the function doo have access to everything created inside the scope of function foo due to the Lexical Scoping of JavaScript. However, function foo cannot access the variables of function doo.



```
}
foo();//print foo
```

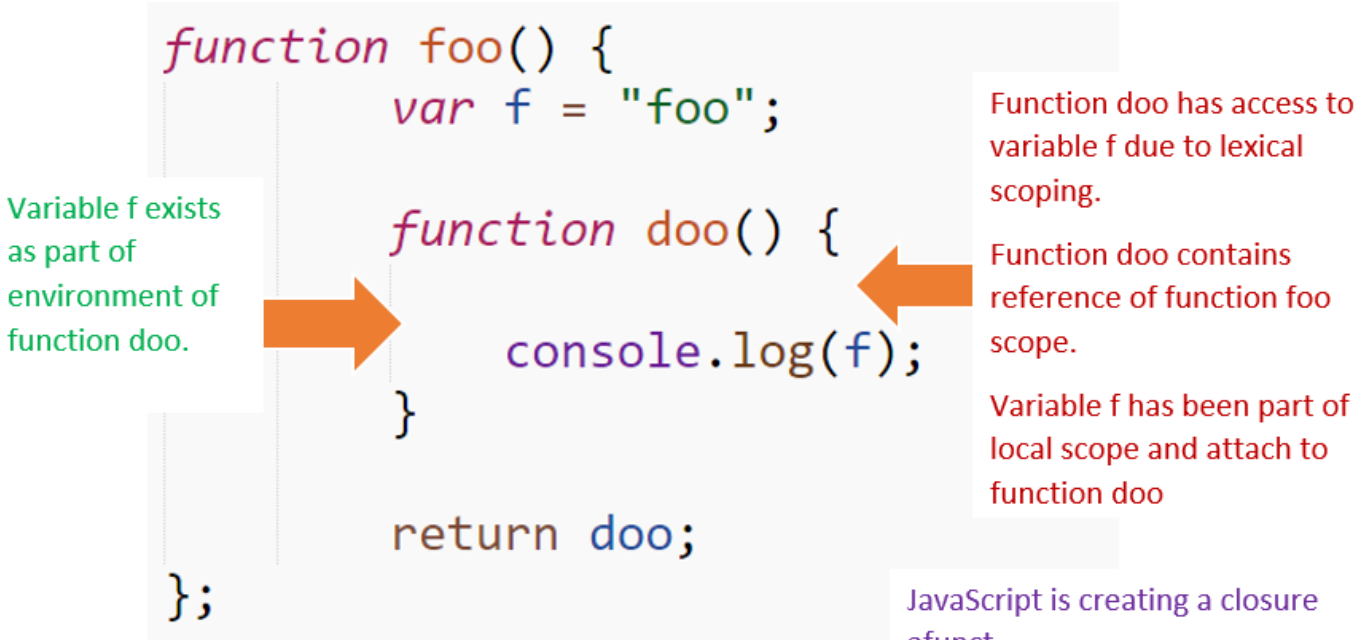
Closures in JavaScript

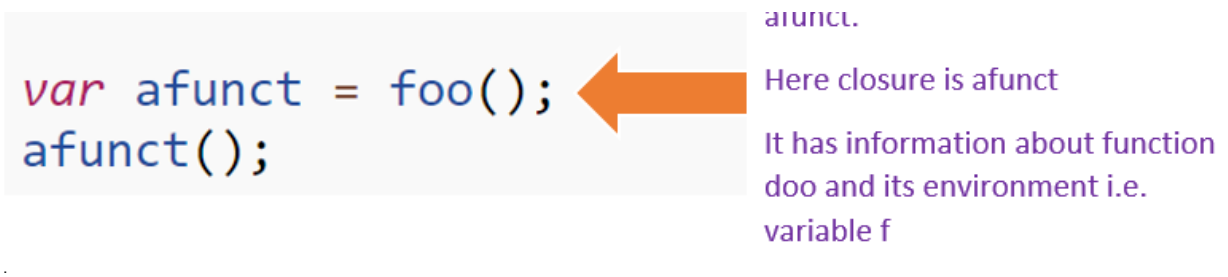
Let us start understanding Closures in JavaScript with an example. Consider the code as shown below. Instead of calling the function doo inside the body of the function foo, we are returning function doo from function foo.

```
function foo() {
   var f = "foo";
   function doo() {
     console.log(f);
   }
   return doo;
 }
var afunct = foo();
afunct();
```

In the above code:

1. function foo is returning another function doo;
2. function doo does not have any of its own variables;
3. due to lexical scoping, function doo is able to access the variable of the parent function foo;
4. function foo is called and assigned to the variable afunct
5. then afunct is called as a function and it prints string “foo”





Surprisingly, the output of the above code snippet is string "foo". Now we might get confused—How is variable f accessed outside the function foo? Normally, the local variables within a function only exist for the duration of that function's execution. So, ideally after execution of foo, variable f should no longer be accessible. But in JavaScript, we can access it because **afunct** has become a JavaScript Closure. The closure afunct has information about function doo and all the local scope variables of function doo at the time of the afunct closure creation.

In case of closure, the inner function keeps the references of the outer function scope. So, in closures:

- The Inner function keeps reference of its outer function scope. In this case, function doo keeps reference of function foo scope.
- Function doo can access variables from the function foo scope reference at any time, even if outer function foo finished executing.
- JavaScript keeps the outer function's (foo in this case) scope reference and its variables in memory until an inner function exists and references it. In this case, the scope and variables of function foo will be kept in memory by JavaScript, until function doo exists.

To understand closure better, let us discuss one more example:

```
function add(num1) {
   function addintern(num2) {
     return num1 + num2;
   } return addintern;
}
var sum9 = add(7)(2);
console.log(sum9);
var sum99 = add(77)(22);
```

```
console.log(sum99);
```

We have two closures here, sum9 and sum99.

When closure sum9 got created, in the local scope of function addintern the value of num1 was 7, and JavaScript remembers that value while creating sum9 closures.

Same in the case of closure sum99... in the local scope of the function addintern the value of num1 was 7, and JavaScript remembers that value while creating closure sum99. As expected, output would be 9 and 99.

We can think of a closure as an object with private variables and one method. Closures allow us to attach data with the function that works on that data. So, a closure can be defined with the following characteristics:

- It is an object
- It contains a function
- It remembers the data associated with the function, including the variables of function's local scope when the closure was created
- To create a closure, the function should return another function reference

Finally, we can define a closure:

"The JavaScript Closure is a special kind of object which contains a function and the environment in which the function was created. Here, environment stands for the local scope of the function and all its variables at the time of closure creation."

Conclusion

In this post, we learned about Closures in JavaScript. I hope you find it useful. Thanks for reading!

Topics: JAVASCRIPT, FUNCTIONAL

Published at DZone with permission of Dhananjay Kumar, DZone MVB. See the original article here.