JavaScript Closures

Summary: in this tutorial, you will learn about JavaScript closures and how to use closures in your code more effectively.

Introduction to JavaScript closures

In JavaScript, a closure is a function (http://www.javascripttutorial.net/javascript-function/) that references variables in the outer scope from its inner scope. The closure preserves the outer scope inside its inner scope.

To understand the closures, you need to know how the lexical scoping works first.

Lexical scoping

Lexical scoping defines the scope of a variable (https://www.javascripttutorial.net/javascript-variable-scope/) by the position of that variable declared in the source code. For example:

```
let name = 'John';

function greeting() {
    let message = 'Hi';
    console.log(message + ' '+ name);
}
```

In this example:

- The variable `name` is a global variable. It is accessible from anywhere including within the `greeting()` function.
- The variable `message` is a local variable that is accessible only within the `greeting()` function.

If you try to access the `message` variable outside the `greeting()` function, you will get an error.

So the JavaScript engine uses the scope to manage the variable accessibility.

According to lexical scoping, the scopes can be nested and the inner function can access the variables declared in its outer scope. For example:

```
function greeting() {
    let message = 'Hi';

    function sayHi() {
        console.log(message);
    }

    sayHi();
}

greeting();
```

The `greeting()` function creates a local variable named `message` and a function named `sayHi()`.

The `sayHi()` is the inner function that is available only within the body of the `greeting()` function.

The `sayHi()` function can access the variables of the outer function such as the `message` variable of the `greeting()` function.

Inside the `greeting()` function, we call the `sayHi()` function to display the message `Hi`.

JavaScript closures

Let's modify the `greeting()` function:

```
function greeting() {
    let message = 'Hi';

    function sayHi() {
        console.log(message);
    }
```

```
    return sayHi;
}
let hi = greeting();
hi(); // still can access the message variable
```

Now, instead of executing the `sayHi()` function inside the `greeting()` function, the `greeting()` function returns the `sayHi()` function object.

Note that functions are the first-class citizens in JavaScript (https://www.javascripttutorial.net/javascript-functions-are-first-class-citizens/) , therefore, you can return a function from another function.

Outside of the `greeting()` function, we assigned the `hi` variable the value returned by the `greeting()` function, which is a reference of the `sayHi()` function.

Then we executed the `sayHi()` function using the reference of that function: `hi()` . If you run the code, you will get the same effect as the one above.

However, the interesting point here is that, normally, a local variable only exists during the execution of the function.

It means that when the `greeting()` function has completed executing, the `message` variable is no longer accessible.

In this case, we execute the `hi()` function that references the `sayHi()` function, the `message` variable still exists.

The magic of this is closure. In other words, the `sayHi()` function is a closure.

A closure is a function that preserves the outer scope in its inner scope.

More JavaScript Closure example

The following example illustrates a more practical example of closure.

```
function greeting(message) {
   return function(name){
        return message + ' ' + name;
   }
}
let sayHi = greeting('Hi');
```

```
let sayHello = greeting('Hello');

console.log(sayHi('John')); // Hi John
console.log(sayHello('John')); // Hello John
```

The `greeting()` function takes one argument named `message` and returns a function that accepts a single argument called `name`.

The return function returns a greeting message that is the combination of the `message` and `name` variables.

The `greeting()` function behaves like a function factory. It creates `sayHi()` and `sayHello()` functions with the respective messages `Hi` and `Hello`.

The `sayHi()` and `sayHello()` are closures. They share the same function body but store different scopes.

In the `sayHi()` closure, the `message` is `Hi`, while in the `sayHello()` closure the `message` is `Hello`.

JavaScript closures in a loop

Consider the following example:

```
for (var index = 1; index <= 3; index++) {
    setTimeout(function () {
        console.log('after ' + index + ' second(s):' + index);
    }, index * 1000);
}
```

Output

```
after 4 second(s):4
after 4 second(s):4
after 4 second(s):4
```

The code shows the same message.

What we wanted to do in the loop is to copy the value of `i` in each iteration at the time of iteration to display a message after 1, 2, and 3 seconds.

The reason you see the same message after 4 seconds is that the callback passed to the `setTimeout()` a closure. It remembers the value of `i` from the last iteration of the loop, which is 4.

In addition, all three closures created by the for-loop (https://www.javascripttutorial.net/javascript-for-loop/) share the same global scope access the same value of `i`.

To fix this issue, you need to create a new closure scope in each iteration of the loop.

There are two popular solutions: IIFE & `let` keyword.

1) Using the IIFE solution

In this solution, you use an immediately invoked function expression (https://www.javascripttutorial.net/javascript-immediately-invoked-function-expression-iife/) (a.k.a IIFE) because an IIFE creates a new scope by declaring a function and immediately execute it.

```
for (var index = 1; index <= 3; index++) {
    (function (index) {
        setTimeout(function () {
            console.log('after ' + index + ' second(s):' + index);
        }, index * 1000);
    })(index);
}
```

Output

```
after 1 second(s):1
after 2 second(s):2
after 3 second(s):3
```

2) Using let keyword in ES6

In ES6, you can use the `let` (http://www.javascripttutorial.net/javascript-variables/#let) keyword to declare a variable that is block-scoped.

If you use the `let` keyword in the for-loop (http://www.javascripttutorial.net/javascript-for-loop/) , it will create a new lexical scope in each iteration. In other words, you will have a new `index` variable in each iteration.

In addition, the new lexical scope is chained up to the previous scope so that the previous value of the `index` is copied from the previous scope to the new one.

```
for (let index = 1; index <= 3; index++) {
    setTimeout(function () {
        console.log('after ' + index + ' second(s):' + index);
    }, index * 1000);
}
```

Output

```
after 1 second(s):1
after 2 second(s):2
after 3 second(s):3
```

Summary

- Lexical scoping describes how the JavaScript engine uses the location of the variable in the code to determine where that variable is available.
- A closure is a combination of a function and its ability to remember variables in the outer scope.