**Summary**

Closures are a very powerful mechanism in the JavaScript programming language. All members of an object in the JavaScript are public by default. However, the closure mechanism enables objects to have private members and more. In this tutorial, we will learn about closures and the benefits of using them in your JavaScript code.

**Definition**

So, what is a closure? Douglas Crockford, author of the book *JavaScript: The Good Parts*, used some great examples of the closure mechanism [here](https://www.safaribooksonline.com/library/view/javascript-the-good/9780596517748/ch04s10.html). His definition of a closure is similar to the following:

*The closure is an inner function which always has access to the variables and parameters of its outer function, even when the outer function has returned.*

**Higher-order functions**

Let's consider the following code:

1function counter(initValue) {

2

3 var currentValue = initValue;

4

5 var increment = function(step) {

6

7 currentValue += step;

8 console.log('currentValue = ' + currentValue);

9 };

10

11 return increment;

12}

13

14var incrementCounter = counter(0);

15

16incrementCounter(1);

17incrementCounter(2);

18incrementCounter(3);

JavaScript

In the JavaScript language, functions are objects, so they can be passed as arguments to other functions, and they can be returned from other functions and assigned to variables.

In our example, we have created the counter function, which returns the increment function and assigns it to the incrementCounter variable, so that variable contains a reference to the increment function (the objects in the JavaScript are being copied by reference, not by value).

That reference enables us to call the increment function outside of the counter function's scope, so in our case, we called it from the global scope. Besides that, the line var incrementCounter = counter(0); initializes the currentValue variable.

Every call after this, incrementCounter serves as a reference for the inner function (referred to by increment). As such, the line incrementCounter(1) actually invokes the increment function with parameter 1. Since that function is a closure, it still has access to the variables and parameters of its outer function, although we called it outside of its outer function. Thus, the currentValue will be increased by 1 and its value will become 1. In the subequent function calls, currentValue will be increased by 2 and 3, respectively, so the output of the code will be:

1currentValue = 1

2currentValue = 3

3currentValue = 6

In JavaScript, local variables of a function will be destroyed after the function returns, unless there is at least one reference on them. In our example, the currentValue is referenced in the increment function, which is referenced in the global scope. Therefore, the currentValue and increment will not be destroyed until the entire script has executed. That's why we can invoke the increment function from the global scope even after the counter function has returned; increment is being called by reference because it is stored in the incrementCounter function.

**Closure via objects**

The counter function can also return more than one function, wrapped into an object. If we had decrement function and wanted to return it, as well, that could be accomplished with the following code:

1function counter(initValue) {

2

3 var currentValue = initValue;

4

5 var increment = function(step) {

6

7 currentValue += step;

8 console.log('currentValue = ' + currentValue);

9 };

10

11 var decrement = function(step) {

12

13 currentValue -= step;

14 console.log('currentValue = ' + currentValue);

15 }

16

17 return {increment: increment,

18 decrement: decrement};

19}

JavaScript

The returned item has increment and decrement properties whose values are increment and decrement functions, respectively. In that way, we can expose those inner functions to the global scope, so we can call them from it:

1var myCounter = counter(0); // myCounter now refers to the object returned by counter

2

3myCounter.increment(1); // call the appropriate "property" function using dot notation

4myCounter.increment(2);

5myCounter.decrement(1);

6myCounter.increment(3);

7myCounter.decrement(2);

JavaScript

If more than one function is being returned, they must be wrapped into an object, like in the example above. Even when a single inner function is being returned by an outer function, the output can be wrapped into an object, but that is typically overkill.

We could also have functions inside of the counter function that we don't want to expose to the global (outer) scope. For instance, we could have a "private" function for logging the currentValue:

1function counter(initValue) {

2

3 var currentValue = initValue;

4

5 var logCurrentValue = function() {

6 console.log('currentValue = ' + currentValue);

7 }

8

9 var increment = function(step) {

10

11 currentValue += step;

12 logCurrentValue();

13 };

14

15 var decrement = function(step) {

16

17 currentValue -= step;

18 logCurrentValue();

19 }

20

21 return {increment: increment,

22 decrement: decrement};

23}

JavaScript

In this case, the logCurrentValue function cannot be accessed from the global scope, since it wasn't returned by the counter function. It can be used just within the increment and decrement functions, but this function remains private in that it cannot be called externally.

Also, the currentValue and initValue variables are private for the counter function object. In a nutshell, this is how we can emulate private members in JavaScript -- by restricting access using scope.

**Multiple object references**

Let's see how this mechanism works if we create multiple function objects. For instance, let's use counter to create two objects called myCounter1 and myCounter2 and call their property functions:

1var myCounter1 = counter(0);

2var myCounter2 = counter(3);

3

4myCounter1.increment(2);

5myCounter2.increment(2);

6myCounter1.decrement(1);

7myCounter2.decrement(1);

JavaScript

In the first two lines of the code, we have actually created 2 different objects, myCounter1 and myCounter2. These objects have the same properties, increment and decrement, which are references to the increment and decrement functions. The myCounter1 object is created by calling the counter function with parameter 0, and the myCounter2 by calling it with parameter 3. This means that increment and decrement functions will have different values of the currentValue variable in the outer scope, so their calls on those two objects will produce different results. In this case, we have 2 different increment and decrement functions because we created 2 different objects each with a pair of functions that has its own scope.

This is a rather confusing edge case, but it makes the code easier to understand. The two variable references point to separate objects because myCounter1 and myCounter2 were instantiated using separate calls to counter. Thus, their calls to increment and decrement are separate as well.

The output of the code above will be:

1currentValue = 2;

2currentValue = 5;

3currentValue = 1;

4currentValue = 4;

js

**Use Case: Timeouts**

Closures are mostly used in callbacks, such as timeouts, event handlers, and so on, as well as in modules.

Let's consider the following code:

1for (var i = 0; i < 5; i++) {

2

3 console.log(i);

4}

JavaScript

The output of this code are numbers from 0 to 4, sequentially.

Let's change it up a little bit:

1for (var i = 0; i < 5; i++) {

2

3 setTimeout(function() {

4

5 console.log(i);

6

7 }, 3000);

8}

JavaScript

The output of the code above will be completely different; the number 5 will be logged 5 times.

Let's analyze this code. The function, which is passed as an argument to the setTimeout function, will be executed 3s after the first loop iteration. Therefore, the value of the variable i, which is in its outer scope (this function is also a closure), will become 5 before its execution because all 5 iterations will certainly be done within those 3 seconds. That function will be executed 5 times, 1 time for every iteration, and it will log 5 five times.

In other words, by the time that the Timeout finishes, the value of i will be 5, and this value is the one that will get logged.

**Using closures to fix the issue**

In order to get the desired output, we will need to add one more closure, so our code should look like this:

1for (var i = 0; i < 5; i++) {

2

3 setTimeout((function outer(i) {

4

5 return function inner() {

6 console.log(i); // 0 1 2 3 4

7 }

8

9 })(i), 3000);

10}

JavaScript

So, now our code contains 2 closures, outer and inner, which both have access to the variable i, but there is a slight difference between them. The outer scope of the outer function is created once and the outer scope of the inner function is created 5 times, for every iteration separately. The outer scope of the outer function contains the variable i, created in the for loop, while the outer scope of the inner function contains this variable, an argument of the outer function. So, those are two different variables with the same name.

In the first iteration the value of the variable i is 0. We have invoked the outer function immediately with parameter 0 and it returned the inner function. Thus, the inner function was actually provided as an argument to the setTimeout function instead of the outer function. Therefore, the inner function will be called after 3s and it will have access to its own outer scope. The value of the variable i in its scope is 0, so the inner function will log 0 at the moment of its execution.

The same will happen in the second iteration, but this time the variable i will be 1, and so on. Finally, we will get desired output in the console.

**An easier way**

However, there is a simpler solution for this problem. The ECMAScript 2015 (or ES6) introduced some very useful new features and some of them are *let* and *const* keywords, which create block scoped variables. If we just replace the keyword var with let in this example, we will get the completely different output:

1for (let i = 0; i < 5; i++) {

2

3 setTimeout(function() {

4

5 console.log(i); // 0 1 2 3 4

6

7 }, 3000);

8}

JavaScript

The keyword let created a block scope for the variable i, for every loop iteration particularly. Therefore, we didn't have to put the line console.log(i) into a new closure in order to create a new scope for i, because the let keyword did it for us.

**Use Case: Event Handlers**

Let's consider the following example: there is a button on an HTML page and we want to show users information about how many times the button was clicked. We could write the following code to implement this functionality:

1<!DOCTYPE html>

2<html>

3<head>

4 <title></title>

5 <meta charset="utf-8" />

6</head>

7<body>

8 <button id="test-button">Test</button>

9 <script>

10 var counter = 0;

11 var element = document.getElementById('test-button');

12

13 //event handler

14 element.onclick = function () {

15

16 counter++;

17 alert('Number of clicks:' + counter);

18 };

19 </script>

20</body>

21</html>

HTML

This code works fine, but we had to define a global variable for counting user clicks on the button. That's generally not recommended since we are polluting the global scope. We use the counter variable only in the event handler, so why not declare it inside the handler? However, if we just move its declaration to the handler's scope, it will not work as expected; every time the handler is called (i.e. every time the user clicks on the button), the counter will be reset to zero.

The solution is to use a closure, like in the following code:

1<!DOCTYPE html>

2<html>

3<head>

4 <title></title>

5 <meta charset="utf-8" />

6</head>

7<body>

8 <button id="test-button">Test</button>

9 <script>

10 var element = document.getElementById('test-button');

11

12 //event handler

13 element.onclick = (function outer() {

14

15 var counter = 0;

16

17 return function inner() {

18

19 counter++;

20 alert('Number of clicks: ' + counter);

21 };

22 })();

23 </script>

24</body>

25</html>

HTML

In this code, we have defined the outer function which returns the inner function. The outer function is immediately invoked and after its execution, the inner function is being assigned to the onclick handler. Now, whenever the user clicks on the button, the inner function will be called instead of the outer function. Since it is a closure, it has access to the outer function's scope, so it can easily change the value of the counter variable. Therefore, the line var counter = 0 will be executed just once, before returning the outer function. After that, the outer function will be called every time the user clicks on the button and it will increment the counter by 1, so the counter will always contain the accurate number of button clicks.

**The Module Pattern**

One of the most popular patterns in JavaScript, the Module pattern, contains closures in its implementation. For instance, we could define the counter module in the following way:

1var counter = (function() {

2

3 var currentValue = 0;

4

5 var logCurrentValue = function() {

6 console.log('currentValue = ' + currentValue);

7 }

8

9 var increment = function (step) {

10

11 currentValue += step;

12 logCurrentValue();

13 };

14

15 var decrement = function (step) {

16

17 currentValue -= step;

18 logCurrentValue();

19 }

20

21 return {

22 increment: increment,

23 decrement: decrement

24 };

25})();

JavaScript

The module pattern is a special case of the singleton pattern, so there can be just one instance of the counter module. This is accomplished by the immediate invocation of function (IIFE) which returns an object that contains references to the public functions and assigns it to a global variable, in our case counter. We can call functions of the counter module like we called them on the myCounter object (substituting counter for myCounter). So the following code would work:

1counter.increment(1);

2counter.increment(2);

3counter.decrement(1);

JavaScript

This will produce the following output:

1currentValue = 1

2currentValue = 3

3currentValue = 2

In this case, the currentValue has been initialized to zero during the module initialization.

**Conclusion**

This tutorial covered the most important things to remember when using closures. Hopefully, it gave you a better understanding of how this mechanism works and the kinds of problems that closures can solve.

134

Pluralsight uses cookies. [Learn more about your privacy](https://www.pluralsight.com/privacy)

* [Features](https://help.pluralsight.com/help/using-pluralsight)
* ·
* [Author](https://www.pluralsight.com/authors)
* ·
* [Mobile & offline apps](https://www.pluralsight.com/downloads)
* ·
* [Blog](https://www.pluralsight.com/blog)
* ·
* [Help center](https://help.pluralsight.com/)
* ·
* [How-to videos](https://learn.pluralsight.com/resource/tutorial/how-to)
* ·
* [Terms of use](https://www.pluralsight.com/terms)
* ·
* [Privacy policy](https://www.pluralsight.com/privacy)