Cristi Salcescu Follow
Enthusiastic about sharing ideas @Senior Systems Engineer at Systematic
Mar 27 · 5 min read

Why you should give the Closure function another chance

In JavaScript, functions can be nested inside other functions.

> *Closure is an inner function that has access to the parent function's variables, even after the parent function has executed.*

As we can see, it becomes interesting when the inner function survives the invocation of the parent function. This will happen in the following situations:

- The inner function is used as a callback for an asynchronous task like a timer, an event, or an AJAX call
- The parent function returns the inner function or returns an object storing the inner function

Closure and Timers

In the following example, we may expect that the local variable `x` will be destroyed immediately after `autorun()` is executed, but it will be alive for 10 minutes. That's because the variable `x` is referenced by the inner function `log()`. The `log()` function is a closure.

```
(function autorun(){
    var x = 1;
    setTimeout(function log(){
       console.log(x);
    }, 600000);
})();
```

When `setInterval()` is used, the variable referenced by the closure function will be destroyed only after `clearInterval()` is called.

Closure and Events

We are creating closures every time variables from the outer functions are used in event handlers. The event handler `increment()` is a closure in the next example.

```
(function initEvents(){
    var state = 0;

    $("#add").on("click", function increment(){
        state += 1;
        console.log(state);
    });
})();
```

Closure and Asynchronous Tasks

When using variables from the outer function in an asynchronous callback, the callback becomes a closure. Variables will be alive till the asynchronous task finishes.

Timers, events, and AJAX calls are maybe the most common asynchronous tasks. There are also other asynchronous APIs like :

HTML5 Geolocation API, WebSockets API, and `requestAnimationFrame()`.

In the next example, the AJAX callback `updateList()` is a closure.

```
(function init(){
  var list;
  $.ajax({ url:
"https://jsonplaceholder.typicode.com/users"})
      .done(function updateList(data){
         list = data;
         console.log(list);
      });
})();
```

Closure and Encapsulation

Another way to see closure is as **a function with private state**. Closures encapsulate state.

For example, let's create a `count()` function with a private state. Every time it is called, it remembers its previous state and returns the next consecutive number. The `state` variable is private, there is no access to it from the outside.

```
function createCount(){
   var state = 0;
   return function count(){
      state += 1;
      return state;
   }
}


var count = createCount();
console.log(count()); //1
console.log(count()); //2
```

We can create many closures sharing the same private state. In the next example, `increment()` and `decrement()` are two closures sharing the same private `state` variable. This way we can create objects with private state.

```
function Counter(){
    var state = 0;
    function increment(){
       state += 1;
       return state;
    }
    function decrement(){
       state -= 1;
       return state;
    }

    return {
       increment,
       decrement
    }
}
var counter = Counter();
counter.increment();//1
counter.decrement();//0
```

> *If you want to understand better the advantages of Factory Functions vs Classes take a look at Class vs Factory function: exploring the way forward*

Asynchronous Task Closures and Loops

In the next example, I'll create five closures for five asynchronous tasks, all sharing the same `i` variable. Because `i` changes during the loop, all logs will display the same value—the last one.

```
(function run(){
    var i=0;
    for(i=0; i<5; i++){
        setTimeout(function logValue(){
            console.log(i);          //5
        }, 100);
    }
})();
```

One way to fix this problem is to use an IIFE (Immediately Invoked Function Expression). In the next example, there are still five closures but over five different `i` variables.

```
(function run(){
    var i=0;
```

```
    for(i=0; i<5; i++){
      (function autorunInANewContext(i){
          setTimeout(function logValue(){
             console.log(i); //0 1 2 3 4
          }, 100);
      })(i);
    }
})();
```

Another option is to use the new `let` statement available as part of ECMAScript 6. It will create a variable local to the block scope for each iteration.

```
(function run(){
    for(let i=0; i<5; i++){
        setTimeout(function logValue(){
           console.log(i); //0 1 2 3 4
        }, 100);
    }
})();
```

I consider `let` to be the best option for this issue from a readability perspective, as well.

Closure and Garbage Collection

In JavaScript, the local variables of a function will be destroyed after the function returns, unless there is at least one reference to them. The private state of a closure becomes eligible for garbage collection after the closure itself has been garbage collected. To make this possible, the closure should no longer have a reference to it.

In the next example, I first create a closure `add()`

```
function createAddClosure(){
    var arr = [];
    return function add(obj){
       arr.push(obj);
    }
}
var add = createAddClosure();
```

Then I define two functions. `addALotOfObjects()` adds a lot of objects to the closure's private state. `clearAllObjects()` sets the closure reference to `null`. Both functions are then used as event handlers.

```
function addALotOfObjects(){
    for(let i=0; i<50000;i++) {
       add({fname : i, lname : i});
    }
}

function clearAllObjects(){
    if(add){
       add = null;
    }
}

$("#add").click(addALotOfObjects);
$("#clear").click(clearAllObjects);
```

Clicking "Add" will add 50,000 items to the closure's private state.

C	# New	# Deleted	# Delta	Alloc. Size	Freed Size	Size Delta
▶	50 000	0	+50 000	2 000 000	0	+2 000 000

Memory snapshot after adding 50000 items

I clicked "Add" three times and then I pressed "Clear" to set the closure reference to `null`. After doing that, the private state was garbage collected.

C	# New	# Deleted	# Delta ▲	Alloc. Size	Freed Size	Size Delta
▶	0	150 000	−150 000	0	6 000 000	−6 000 000

Memory snapshot after setting the closure reference to null

Conclusion

Closure is the best tool in our toolbox for creating encapsulation.

It makes work with callbacks for asynchronous tasks simple. We use the variables we want, and they will be alive by the time the callback is called.

Closure impacts the lifetime of its private state. The private state will be garbage-collected only after the closure function itself is collected.

> *It's maybe the best feature ever put into a programming language.*
>
> *Douglas Crockford on Closure*

More on Functional Programming in JavaScript :

How point-free composition will make you a better functional programmer

Here are a few function decorators you can write from scratch

Class vs Factory function: exploring the way forward

Make your code easier to read with Functional Programming