# More About Closures in JavaScript

Tuesday, 29 August 2023 9:57 PM

Q.

```
4  let count = 0;
5  (function printCount() {
6    if (count === 0) {
7      let count = 1;
8      console.log(count);
9    }
10   console.log(count);
11 })();
12
```

→ So o/p will be : 1 0

→ Since 'let' is block scoped so line 8 will console '1' as as another let variable will be created for the block which will shadow the global variable.

→ After the block is completed, the inside 'let' variable gets destroyed, so, since f^n forms a closure with global scope, so '0' will be printed.

Q: Write a f^n that would allow you to do this:

```
var addSix = createBase(6);
addSix(10); // returns 16
addSix(21); // returns 27
```

ans.

```
function createBase(num) {
  return function (innerNum) {
    console.log(innerNum + num);
  };
}
```

→ So, createBase() returns a f^n which forms a closure with createBase() & thus, has access to its parameters.

→ Time optimization:

```
function find(index) {
  let a = [];
  for (let i = 0; i < 1000000; i++) {
    a[i] = i * i;
  }

  console.log(a[index]);
}

console.time("6");
find(6);
console.timeEnd("6");
console.time("12");
find(50);
console.timeEnd("12");
```

```
36                          script.js:10
6: 57.974853515625 ms       script.js:15
144                         script.js:10
12: 64.512939453125 ms      script.js:18
>
```

→ This code takes much time to execute.

→ To optimize this we can use closure, so that the loop & let variable remains the same for every time we call closure().

```
4  function find() {
5    let a = [];
6    for (let i = 0; i < 1000000; i++) {
7      a[i] = i * i;
8    }
9
10   return function (index) {
11     console.log(a[index]);
12   };
13 }
14
```

```
36                          script.js:11
6: 0.25 ms                  script.js:18
2500                        script.js:11
50: 0.02587890625 ms        script.js:21
```

→ So, we've optimized this

```
const closure = find();
console.time("6");
closure(6);
console.timeEnd("6");
console.time("50");
closure(50);
console.timeEnd("50");
```

amazingly.

→ Basically we store the closure of the returned fn from find() in closure(), so now we do not recreate that loop whenever closure() is called.

Note: We have a convention to create private variables with '_' (underscore).

→ Module pattern:

```
// Ques 6: What is Module Pattern?

var Module = (function () {
  function privateMethod() {
    // do something
    console.log("private");
  }

  return {
    publicMethod: function () {
      console.log("public");
    },
  };
})();

Module.publicMethod();
Module.privateMethod();
```

```
public                                script.js:12
▶ Uncaught TypeError:                 script.js:18
Module.privateMethod is not a
function
    at script.js:18:8
>
```

→ This is a module pattern which is an IIFE & has a private method & a public method.

→ A public method is accessible outside the module pattern but not a private method.

→ the private methods and variables can only be accessed by the public method.

→ Make a fn that can only be called once;

→ To do this we use the concept of closures.

```
// once function

function once () {
    let called = 0;

    return function () {
        if(called > 0) {
            console.log("Already Called the function");
        } else{
            console.log("Calling the function for the first time");
            called++;
        }
    }
}

let calledTheFunction = once();
calledTheFunction();
calledTheFunction();
calledTheFunction();
calledTheFunction();
```

```
node /tmp/mpqlPJ2elr.js
Calling the function for the first time
Already Called the function
Already Called the function
Already Called the function
```

→ So, here we make a fn once() that returns an anonymous fn which makes a closure with once().

→ Now we store the returned fn or closure in calledTheFunction(), so, the instance of 'called' variable is not created again & again.

→ If we had done `once()()`, then it would not have worked because for each time `once()()` would have run, 'called' variable would have initialised.

→ We have once() fn in lodash library as well.

→ Polyfill of once():

```
function once(func, context) {
  let ran;

  return function () {
    if (func) {
      ran = func.apply(context || this, arguments);
      func = null;
    }

    return ran;
  };
}

const hello = once(() => console.log("hello"));

hello();
hello();
hello();
hello();
```

(once() polyfill)

```
function once(func, context) {
  let ran;

  return function () {
    if (func) {
      ran = func.apply(context || this, arguments);
      func = null;
    }

    return ran;
  };
}

const hello = once((a, b) => console.log("hello", a, b));

hello(1, 2);
hello(1, 2);
hello(1, 2);
hello(1, 2);
```

(when fn takes in some arguments)

→ So, we create a fn 'once()' which take 2 arguments, a 'callback fn' & a 'context' (for telling 'this' keyword where it should point to). once() fn returns a fn which will make the callback fn only run once.

→ Returned fn forms closure with 'once()'. We declare ran outside the returned fn because we do not want ran to be initialized everytime the returned fn is called.

→ Now for the 1st time, 'func' will always be true, since we are passing the callback fn to once(). Thus, if condition will run, and we will use 'apply method to run the fn as we need to pass 'context' & 'arguments as a list.

→ Now, if 'context' is not passed through 'once()' then we will pass 'this' keyword which will point to the local memory of the returned fn.

→ We will also pass the 'arguments'. 'arguments' is an array like object which has the arguments passed to a fn & is automatically available in case of normal fn's but is not available in case of arrow fn's.

→ Finally we set 'func = null;' so that if condition does not run again.

→ We store the closure of the returned fn in hello() & pass the callback fn to 'once()'. Now however many times we call hello() it will only run once.

⇒ Implementing memoize/caching fn:

```
const clumsysquare = (num1, num2) => {
  for (let i = 1; i <= 100000000; i++) {}

  return num1 * num2;
};

console.time("First call");
console.log(clumsysquare(9467, 7649));
console.timeEnd("First call");

console.time("Second call");
console.log(clumsysquare(9467, 7649));
```

```
1 Issue: 1
72413083                                  script.js:10
First call: 40.2109375 ms                 script.js:11
72413083                                  script.js:14
Second call: 42.37109375 ms               script.js:15
```

o/p when the fn is called with same arguments twice.

```
15  console.timeEnd("Second call");
16
```

→ If a fⁿ takes a good amount of time to run then we must save its value somewhere, so that if the same arguments are passed, it uses the memoized value & does not take the same amount of time as before.

```
3  function myMemoize(fn, context) {
4    const res = {};
5    return function (...args) {
6      var argsCache = JSON.stringify(args);
7      if (!res[argsCache]) {
8        res[argsCache] = fn.call(context || this, ...args);
9      }
10     return res[argsCache];
11   };
12 }
```

(memoize/caching fⁿ)

```
res = {
  "5,6": 30,
};
```

(how the res object will look with dynamic key:value pairs)

```
const clumsyProduct = (num1, num2) => {
  for (let i = 1; i <= 100000000; i++) {}

  return num1 * num2;
};

const memoizedClumzyProduct = myMemoize(clumsyProduct);

console.time("First call");
console.log(memoizedClumzyProduct(9467, 7649));
console.timeEnd("First call");

console.time("Second call");
console.log(memoizedClumzyProduct(9467, 7649));
console.timeEnd("Second call");
```

(calling the original fⁿ by passing it to the memoizing fⁿ)

```
1 Issue: 1
72413083                                  script.js:23
First call: 47.67578125 ms                script.js:24
72413083                                  script.js:27
Second call: 0.052001953125 ms            script.js:28
```

(o/p of memoization which is far better than before)

→ Our myMemoize() fⁿ takes in a callback fⁿ & a context.
→ It returns a memoized fⁿ which takes in arguments & forms closure with its parent fⁿ. We declare `res` outside the returned fⁿ so that it is not created again & again when inner fⁿ is returned. otherwise since const is block scoped, a new value of `res` will be created when the inner fⁿ is returned which would be an empty object.
→ Now, we convert the `...args` received to a string using JSON.stringify & store it in `argsCache`.
→ Now `argsCache` will be the dynamic key in `res` object. So if `res[argsCache]` is there in `res` object then we'll return it otherwise we'll call the callback fⁿ using `.call()` method & pass the `context` (if received otherwise pass `this` keyword which points to the inner fⁿ) & pass the arguments using `...args` & store the returned value in `res` object & finally return `res[argsCache]`.