<https://stackoverflow.com/questions/2728278/what-is-a-practical-use-for-a-closure-in-javascript>

**Application of JS Closures**

1. **To delay the use of a variable by using setTimeout**
2. **To create private variables**
3. **Simulating object oriented programming**
4. **Factory function**
5. **Throttling**
6. **Debouncing**

**Brief Description of closures**

Closures are one of the most powerful features of JavaScript. JavaScript allows for the nesting of functions and grants the inner function full access to all the variables and functions defined inside the outer function (and all other variables and functions that the outer function has access to). However, the outer function does not have access to the variables and functions defined inside the inner function. This provides a sort of security for the variables of the inner function. Also, since the inner function has access to the scope of the outer function, the variables and functions defined in the outer function will live longer than the outer function itself, if the inner function manages to survive beyond the life of the outer function. **A closure is created when the inner function is somehow made available to any scope outside the outer function.**

1. **To delay the use of a variable by using setTimeout**

**WHAT ?**

setTimeout inside of a for-loop does not print consecutive values.

for (var i = 1; i <= 2; i++) {

setTimeout(function() { alert(i) }, 1000);

}

But 3 is alerted both times, instead of 1 then 2.

**HOW ?**

Scope of setTimeout is for-loop here.

Iteration1:

i =1, setTimeout with function() { alert(i) } added to web API -> loaded to Execution Queue after 1000ms

Iteration2:

i =1, setTimeout with function() { alert(i) } added to web API -> loaded to Execution Queue after 1000ms

Iteration3:

i =1, setTimeout with function() { alert(i) } added to web API -> loaded to Execution Queue after 1000ms

Now, loop is completed with i = 3.

First setTimeout will be added to Call stack, function() { alert(3) }. Here i = 3 and not 1 because for loop is the scope of setTimeout and within it value of i is 3.

Similarly, remaining two setTimeout are executed and that is the reason 3 is printed every time.

**WHY ?**

*setTimeout & setInterval*

When JavaScript engine execute .js file line by line, if it finds a line as statement or function call it load it on stack and execute but when it encounters a setTimeout or setInterval call, then function handler associated with setTimeout or setInterval is taken out by Time API (one of web API of browser) and hold it for that time. Even if it was setTimeout(.., 0) with zero time interval, then also it will be taken out by Time API.

Once this time is over, Time API put that function at end of execution queue.

Now Execution of that function depends on other functions calls which are ahead of it in the queue.

**SOLUTION #1**

The problem here is because of scope. If we manage to save the value of “ i “ somehow for each setTimeout then we might be able to solve this issue.

Using closures a scope can be created like shown below,

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

function increment(i){

setTimeout(function() { console.log(i) }, 100);

}

increment(i);

}

Now, with each iteration a new scope is created on which each setTimeout callback functions can be closed. So, when setTimeout comes back value of “ i “ will be same as it was saved at the time it was first encountered in the script.

We can also use IIFE.

An **immediately invoked function expression** (or IIFE, pronounced "iffy") is a JavaScript programming language idiom which produces a lexical scope using JavaScript's function scoping.

for (var i = 1; i <= 2; i++) {

(function(){

var j = i;

setTimeout(function() { console.log(j) }, 100);

})();

}

The cleaner one would be,

for (var i = 1; i <= 2; i++) {

(function(i){

setTimeout(function() { console.log(i) }, 100);

})(i);

}

**SOLUTION #2**

Using bind(): The bind() method creates a new function that, when called, has its “ this “ keyword set to the provided value.

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

function increment(i){

setTimeout(function() { console.log(this) }.bind(i), 100);

}

increment(i);

}

Here, we bind setTimeout callback function with value of i.

That is why “ this “ inside of callback function stores the context that is value of “ i “.

OUTPUT:

Number {1}

Number {2}

Number {3}

Number {4}

Number {5}

1. **To create private variables**

Technically every function you make in Javascript on a browser is a closure because the window object is bound to it.

Suppose, you want to **count the number of times user clicked a button** on a webpage.  
For this, you are triggering a function on **onclick** event of button to update the count of the variable

<button onclick="updateClickCount()">click me</button>

**Now there could be many approaches like:**

1) You could use a **global variable**, and a function to increase the **counter**:

var counter = 0;

function updateClickCount() {

++counter;

// do something with counter

}

But, the pitfall is that **any script on the page can change the counter, without calling updateClickCount()**.

2) Now, You might be thinking of declaring the variable inside the function:

function updateClickCount() {

var counter = 0;

++counter;

// do something with counter

}

But, Hey! Every time updateClickCount() function is called, the **counter is set to 1 again.**

3) Thinking about **Nested functions**?

Nested functions have access to the scope "above" them.  
In this example, the inner function updateClickCount() has access to the counter variable in the parent function countWrapper()

function countWrapper() {

var counter = 0;

function updateClickCount() {

++counter;

// do something with counter

}

updateClickCount();

return counter;

}

This could have solved the counter dilemma, if you could reach the updateClickCount() function from the outside and you also need to find a way to execute counter = 0 only once not everytime.

4) **Closure to the rescue! (Self-invoking function)**:

var updateClickCount=(function(){

var counter=0;

return function(){

++counter;

// do something with counter

}

})();

The self-invoking function only runs once. It sets the counter to zero (0), and returns a function expression.

This way updateClickCount becomes a function. The "wonderful" part is that it can access the counter in the parent scope.

This is called a **JavaScript closure**. It makes it possible for a function to have "*private*" variables.

The counter is protected by the scope of the anonymous function, and can only be changed using the add function!

**More lively example on Closure:**

<script>

var updateClickCount=(function(){

var counter=0;

return function(){

++counter;

document.getElementById("spnCount").innerHTML=counter;

}

})();

</script>

<html>

<button onclick="updateClickCount()">click me</button>

<div> you've clicked

<span id="spnCount"> 0 </span> times!

</div>

</html>

1. **Simulating object oriented programming:**

{function counter() {

var a = 0;

return {

inc: function() { a = a+5; console.log(a); },

dec: function() { a = a-1; console.log(a);},

reset: function() { a = 0; console.log(a);}

}

}

let func = counter();

func.inc(); //5

func.dec(); //4

func.reset(); //0

}

Here all three functions are using the single copy of var a. Just like in JAVA where we have a class and only methods inside of that class are able to access the class variables and make changes to it.

1. **Factory function**

function makeAdder(x) {

return function(y) {

return x + y;

};

}

var addFive = makeAdder(5);

console.log(addFive(2)); //7

console.log(addFive(10)); //15

1. **Throttling:**

Throttling puts a limit on as a maximum number of times a function can be called over time. As in "execute this function at most once every 100 milliseconds."

Code :

const throttle = (func, limit) => {

let isThrottling

**return function()** {

const args = arguments

const context = this

if (!isThrottling) {

func.apply(context, args)

isThrottling = true

setTimeout(() => isThrottling = false, limit)

}

}

}

1. **Debouncing:**

Debouncing put a limit on a function not be called again until a certain amount of time has passed without it being called. As in "execute this function only if 100 milliseconds have passed without it being called."

Code:

const debounce = (func, delay) => {

let debouncing

**return function()** {

const context = this

const args = arguments

clearTimeout(debouncing)

debouncing = setTimeout(() => func.apply(context, args), delay)

}

}

As you can see closures helped in implementing two beautiful features which every web app should have to provide smooth UI experience functionality.

**Anonymous functions Defined functions**

======================================================================================================

Cannot be used as a method Can be used as a method of an object

Exists only in the scope in which it is defined Exists within the object it is defined in

Can only be called in the scope in which it is defined Can be called at any point in the code

Can be reassigned a new value or deleted Cannot be deleted or changed