Closures in JavaScript

A closure is the combination of a function bundled together (enclosed) with references to its surrounding state (the lexical environment). In other words, a closure gives you access to an outer function's scope from an inner function. In JavaScript, closures are created every time a function is created, at inner function creation time.

Example’s

1. Case

function x(){

var a = 10;

function y(){

console.log(a);

}

return y;

}

const z = x();  
console.log(z);// y(){ anonymous function}   
  
Explanation: In this case z have the reference of the whole function of x  
inside the x it will return function of y along with it lexical scope

case 2:

function x(){

var a = 10;

function y(){

console.log(a);

}

return y;

}

const z = x();  
z() // 10  
  
Explanation: whenever function goes inside it has the lexical environment and function y is referring(reference) to variable ‘a’ which has 10 function y is remember the memory location of the variable a.  
  
Ex: 3  
function x(){   
let num = 10;  
function y(){   
console.log(num)   
}   
num = 20;   
return y;   
}   
let result = x();  
console.log(result())  
// output:   
20  
undefined  
Ex: However, **y()** itself doesn't return anything, so when you call **console.log(result())**, it prints the value returned by **result()**, which is **undefined**. This is because **y()** doesn't explicitly return any value, so its return value is **undefined** by default in JavaScript. If you want **result()** to log **20**, you should call **y()** directly inside **x()** before returning it

Case:4

function z(){  
var b = 200;

function y(){  
var a = 7;

function x(){

console.log(a);

}x()

}y()

}z()

Explanation: In this case x is referencing to variable a which is 7 along with parent function y which is referencing to variable b which is 200 it maintain the lexical scope.

Ex: 3

var sum = function(a){  
console.log(“Live Viewers: ”, +a);

var c = 4;

return function(b){

return a + b + c;

}   
}

var store = sum(200);

console.log(store(4))  
Explanation: 1) First with sum function it passed 200 which value got a as a 200  
 2) then c value has 4 which total become is 204  
 3) and last as you can see that in store function has passed 4 so it will go inside with previous value 204 it will add with it which will become 208   
 4) Means 204 was retained the value this is beauty of the closures

Ex: 4

var sum = function(a, b, c){

return{

sumTwo:function(a,b){

return a + b;

},

sumThree:function(a, b, c){

return a + b + c;

}

}

}

var store = sum(1,2,3);

console.log(store.sumTwo(1,2)) // 3

console.log(store.sumThree(1,2,3)) //6

Ex: 5

function init() {

var name = "Mozilla"; // name is a local variable created by init

function displayName() {

// displayName() is the inner function, that forms the closure

console.log(name); // access variable declared in the parent function

}

displayName();

}

init();

Ex: init() creates a local variable called name and a function called displayName(). The displayName() function is an inner function that is defined inside init() and is available only within the body of the init() function. Note that the displayName() function has no local variables of its own. However, since inner functions have access to the variables of outer functions, displayName() can access the variable name declared in the parent function, init().

Ex: 5

function makeFunc() {

const name = "Mozilla";

function displayName() {

console.log(name);

}

return displayName;

}

const myFunc = makeFunc();

myFunc();

Explanation:

1)The reason is that functions in JavaScript form closures. A closure is the combination of a function and the lexical environment within which that function was declared.

2)This environment consists of any local variables that were in-scope at the time the closure was created.

3) In this case, myFunc is a reference to the instance of the function displayName that is created when makeFunc is run.

4)The instance of displayName maintains a reference to its lexical environment, within which the variable name exists.

5)For this reason, when myFunc is invoked, the variable name remains available for use, and "Mozilla" is passed to console.log.

**Closure scope chain**

Every closure has three scopes:

1) Local scope (Own scope)

2) Enclosing scope (can be block, function, or module scope)

3) Global scope

Ex:1

// global scope

const e = 10;

function sum(a) {

return function (b) {

return function (c) {

// outer functions scope

return function (d) {

// local scope

return a + b + c + d + e;

};

};

};

}

console.log(sum(1)(2)(3)(4)); // 20

Ex: 2

// global scope

const e = 10;

function sum(a) {

return function sum2(b) {

return function sum3(c) {

// outer functions scope

return function sum4(d) {

// local scope

return a + b + c + d + e;

};

};

};

}

const sum2 = sum(1);

const sum3 = sum2(2);

const sum4 = sum3(3);

const result = sum4(4);

console.log(result); // 20

Emulating private methods with closures

Note: This Problem is followed by module design pattern  
Ex: const counter = (function () {

let privateCounter = 0;

function changeBy(val) {

privateCounter += val;

}

return {

increment() {

changeBy(1);

},

decrement() {

changeBy(-1);

},

value() {

return privateCounter;

},

};

})();

console.log(counter.value()); // 0.

counter.increment();

counter.increment();

console.log(counter.value()); // 2.

counter.decrement();

console.log(counter.value()); // 1.

Another Example of the closure  
Ex: function foo() {

var name = 'Roadside Coder'; // name is a local variable created by foo

function displayName() { <----- A Closure // displayName() is the inner function

alert(name); // variable used which is declared in the parent function

}

displayName();

}

foo();

**Why do we use Closures?**

Closures makes it possible for a function to have "private" variables. JavaScript closure is used to control what is and isn't in scope in a particular function, along with which variables are shared between sibling functions in the same containing scope.

## Mostly asked Interview Questions for Closures

**1. What are the advantages of Closures?**

There are several advantages of using closures in JavaScript. Some of them are:

* Closure enables the use of nested functions that are used to get the values created in the execution context of that of the parent function.
* They can prove as a perfect solution while solving a problem of hierarchy in any program.
* They have huge importance as far as functional programming is concerned. It solves the for loop problem of functional programming.
* It is also used to simulate private scope for variables.

**2. What is the difference between closure and scope?**

When you declare a variable in a function, you can only access it in the function. These variables are said to be scoped to the function.If you define any inner function within another function, this inner function is called a **closure**. It retains access to the variables created in the outer function.

Whereas a **scope** in JavaScript defines what variables you have access to. There are two kinds of scope – global scope and local scope.

Ex: What will be the output ?

let count = 0;

(function immediate() {

if (count === 0) {

let count = 1;

console.log(count); // Output 1

}

console.log(count); // Output 0

})();

Explanation:

\*\*Solution : \*\*

1 and 0 is logged to the console.

The first statement let count = 0 declares a variable count.

immediate() is a closure that captures the count variable from the outer scope. Inside of the immediate() function scope count is 0.

However, inside the conditional, another let count = 1 declares a local variable count, which overwrites count from outer the scope. The first console.log(count)logs 1.

The second console.log(count) logs 0, since here count variable is accessed from the outer scope.

**4. Can you create a function named createBase to show the below functionality?**

var addSix = createBase(6);

addSix(10);// returns 16

addSix(21);// returns 27

**Solution :**

You can create a closure to keep the value passed to the function createBase even after the inner function is returned. The inner function that is being returned is created within an outer function, making it a closure, and it has access to the variables within the outer function, in this case the variable baseNumber.

function createBase(baseNumber) {

return function(N) {

// we are referencing baseNumber here even though it was declared

// outside of this function. Closures allow us to do this in JavaScript

return baseNumber + N;

}

}

var addSix = createBase(6);

addSix(10); //16

addSix(6); // 21

**5.How do you optimise time using closures?**

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); // this takes 37ms

console.timeEnd("6");

console.time("12");

find(12); // this takes 135ms

console.timeEnd("12");

**Solution :**

function find() {

let a = [];

for (let i = 0; i < 1000000; i++) {

a[i]= i\*i

}

return function(index){

console.log(a[index]);

}

}

const closure = find()

console.time("6");

closure(6); // this takes 0.25 ms

console.timeEnd("6");

console.time("12");

closure(12) // this takes 0.025ms

console.timeEnd("12");

1. Solution: **Pre-computation of Array**: In the original code, the **a** array is computed every time the **find** function is called. This means that for every call to **find**, the loop runs from 0 to 999999, computing **i \* i** and storing it in the **a** array. This is inefficient because the same values are computed repeatedly for each call to **find**.
2. **Using Closure**: By using a closure, we can compute the **a** array once and then return a function (**find**) that has access to this pre-computed array. This function retains access to the variables (**a**) from the scope in which it was created.
3. **Optimized Access**: With the closure approach, when **find** is called, it directly accesses the pre-computed **a** array. There's no need to recompute the array every time **find** is called, which significantly reduces the computational overhead, especially for large arrays.

**6. What will be the output for the following?**

for (var i = 0; i < 3; i++) {

setTimeout(function log() {

console.log(i); // What is logged?

}, 1000);

}

\*\*Solution : \*\* 333

**Explanation:**

1. **var Declaration Scope:**
   * The variable i is declared using var inside the for loop. In JavaScript, var is function-scoped or globally scoped if not within a function.
   * This means that there is only one instance of i that is shared across all iterations of the loop.
2. **Closure and setTimeout:**
   * The setTimeout function is asynchronous and executes its callback function (log) after the specified delay (1000 milliseconds in this case).
   * The log function forms a closure, capturing the reference to i rather than its value at the time setTimeout is called.
3. **Loop Execution:**
   * The for loop runs three times, incrementing i from 0 to 2.
   * After the loop finishes, i is incremented to 3 and the loop exits.
4. **Asynchronous Callback Execution:**
   * After approximately 1000 milliseconds, the setTimeout callbacks are executed.
   * By this time, the loop has completed, and i has the final value of 3.
   * Since the log function captures the reference to i, when the callbacks execute, they all log the current value of i, which is 3.

Thus, the value of i at the time the log function is executed is 3 for all three setTimeout callbacks.

To achieve the expected behavior of logging 0, 1, and 2, you can use let instead of var. The let keyword is block-scoped, meaning a new instance of i is created for each iteration of the loop.

javascript

Copy code

for (let i = 0; i < 3; i++) {

setTimeout(function log() {

console.log(i); // Logs 0, 1, and 2

}, 1000);

}

In this version:

* Each iteration has its own instance of i, so when the log function executes, it captures
* the correct value of i for each iteration.

**7. How would you use a closure to create a private counter?**

You can create a function within an outer function (a closure) that allows you to update a private variable but the variable wouldn't be accessible from outside the function without the use of a helper function.

function counter() {

var \_counter = 0;

// return an object with several functions that allow you

// to modify the private \_counter variable

return {

add: function(increment) { \_counter += increment; },

retrieve: function() { return 'The counter is currently at: ' + \_counter;

}

}

}

// error if we try to access the private variable like below// \_counter;// usage of our counter function

var c = counter();

c.add(5);

c.add(9);

// now we can access the private variable in the following way

c.retrieve();// => The counter is currently at: 14

**8. What is module pattern?**

var Module = (function() {

function privateMethod() {

// do something

}

return {

publicMethod: function() {

// can call privateMethod();

}

};

})();

The return statement of the Module contains our public functions. The private functions are just those that are not returned. Not returning functions makes them inaccessible outside of the Module namespace. But our public functions can access our private functions which make them handy for helper functions, AJAX calls, and other things.

Module.publicMethod(); // works

Module.privateMethod(); // Uncaught ReferenceError: privateMethod is not defined

One convention is to begin private functions with an underscore, and returning an anonymous object containing our public functions. This makes them easy to manage in a long object. This is how it looks:

var Module = (function () {

function \_privateMethod() {

// do something

}

function publicMethod() {

// do something

}

return {

publicMethod: publicMethod,

}

})();

**9. Rewrite the function in such a way the output gets printed once even though the function is called multiple times.**

let view

function likeTheVideo(){

view="Roadside Coder"

console.log( "Subscribe to", view)

}

likeTheVideo(); // Subscribe to Roadside Coder

likeTheVideo(); // Subscribe to Roadside Coder

likeTheVideo(); // Subscribe to Roadside Coder

likeTheVideo(); // Subscribe to Roadside Coder

**Solution :**

let view;

function likeTheVideo(){

let called = 0;

return function(){

if(called>0){

console.log("Already Subscribed");

}

else{

view="Roadside Coder"

called++;

console.log( "Subscribe to", view)

}

}

}

let isSubscribe = likeTheVideo()

isSubscribe() // Subscribe to Roadside Coder

isSubscribe() // Already Subscribed

\*\*10. Write the polyfill for "\_.once" method in lodash. \*\*

function once(func) {

let called = false;

let result;

return function (...args) {

if (!called) {

called = true;

result = func.apply(this, args);

}

return result;

};

}

// Example usage:

const myFunc = once(function() {

console.log("This function should only be called once.");

});

myFunc(); // Outputs: "This function should only be called once."

myFunc(); // No output, as the function has already been called once.

\*\*11. Write the polyfill for "\_.memoize" method in lodash. \*\*

// Polyfill for \_.memoize method

function memoize(func) {

const cache = {};

return function (...args) {

const key = JSON.stringify(args);

if (!(key in cache)) {

cache[key] = func.apply(this, args);

}

return cache[key];

};

}

// Example usage:

const fibonacci = memoize(function(n) {

if (n <= 1) {

return n;

}

return fibonacci(n - 1) + fibonacci(n - 2);

});

console.log(fibonacci(10)); // Outputs: 55

console.log(fibonacci(20)); // Outputs: 6765