· github : divyaranjan10

DATE

# CHAP 12: Crazy JS Interview ft. Closures

Q1. What is closure in Javascript?
⇒ A function bundled with it's lexical envi-
-ronment is called a closure. [For more details,
see chapter 10 notes].

Q2. Can you give an example of a closure in JS?
⇒
```
function outer() {
  var a = 10;
  function inner() {
    console.log(a);
  }
  return inner;
}
outer()();
```

· Output
```
10
```

· So, in this, the inner() has access to the 'outer'
environment, therefore it has access to variable
'a' which is in outer scope. So, when the last
line gets executed i.e, 'outer()()', we'll get the
answer as 10, even if it is outside, it still

DATE

remembers the 'reference of a' and hence 10 is printed.

Q3. Use of double parenthesis ()() in JS?
⇒ So, the inner() function can directly be called using double parenthesis, i.e outer()().

Q4. Are let declarations closed over?
⇒ Yes, it will still work even if we use 'let' instead of 'var'.

```
function outer () {
    function inner () {
        console. log(a);
    }
    let a = 10;
    return inner;
}
outer ()();
```

• Output

```
10
```

DATE

Q5. Are function parameter closed over?

Ans.
```
function outer(b) {

    function inner() {
        console.log(a, b);
    }
    let a = 10;
    return inner;
}
var close = outer("Hello World");
close();
```

OUTPUT
```
10, "Hello World"
```

=> Yes, it will work. This happens because our function forms an closure with outer environment.

Q6. What if the outer() function put inside another function? Will inner() function have access to that function also?

DATE___________

```
=> function outest (){                · Output
     let c = 30;                  10 "Hello WORLD" 30
     function outer (b){
        function inner (){
           console. log (a,b,c);
        }
        let a = 10;
        return inner;
     }
     return outer;
   }
   var close = outest ()("Hello World");
   close ();
```

⟶ inner()

=> Yes, it will have access of the outest () function scope, because closure is a function bundled together with it's lexical environment, and lexi--cal environment means local ~~scope~~ memory along with it's parent's lexical environment.

=> That's why inner () will have access to it's parent's parent i.e outest().

DATE

Q7. Conflicting name global variables in JS

=> 
```
function outest (){                          Output
   var c = 20;                     10 "Hello World" 20
   function outer (b) {
      function inner (){
         console.log(a, b, c);
      }
   (12.1) let a = 10;
      return inner;
   }
   return outer;
}
(12.2) let a = 100;
var close = outest ()("Hello World");
close();
```

-> So, in line (12.1) and (12.2) conflicting name 'a' is there, (12.1 in outer's local scope), and ((12.2) in global scope) but the output of a will be 10 because 'a' has formed a closure which is in outer's local scope not the global one.

=> But if we remove (12.1), then output will be 100

DATE________

because now 'a' will refer to the global variable 'a' and point it's value.

Q8. Advantages of closures.
=> Discussed in chapter 10

Q9. Data hiding & encapsulation in JS.
=> Suppose we have a variable and we want to have data privacy over it that no outside function could access it, basically we can encapsulate that data so that other functions can't see it.
· Therefore, closures can also be used for data hiding and encapsulation. So other code cannot access this value. [Summary point by Jagrut Sharma]

Q10. Example of data privacy using closures.
=>
```
function counter (){
    var count =0;
    return function increaseCount(){
        count ++;
```

DATE____________

```
            console.log (count);
        }
    }
                                        · Output
                                          1
    var counter 1 = counter ();           2
     counter 1();                         1
     counter 1();                         2

(12.3) var counter 2 = counter ();
     counter 2();
     counter 2();
```

⇒ So, we can say that in this example we have maintained data privacy over 'count' variable, so that an outside function can not access this particular variable.

⇒ Counter question on (12.3), what if we want to call this funcⁿ again and store it in a variable and similarly use it as above?
• So, the it will be a fresh 'counter()' in

DATE_____________

itself. So, it will again take 'count = 0' and increment thereafter.

Q11. Function constructor in Javascript.
=> In this example we'll use function constructor for different functions, i.e for increment and decrement.

```
function counter (){
    var count =0;
    this. incrementCounter = function(){
        count ++;
        console. log (count);
    }
    this. decrementCounter = function (){
        count --;
        console. log (count);
    }
}

var counter1 = new counter();
counter1. incrementCounter();
counter1. incrementCounter();
```

Output

```
1
2
1
```

DATE____________

counter1.decrementCounter();

Q12. Disadvantages of closures.
=> Overconsumption of memory : Everytime a closure is formed it consumes a lot of memory and sometimes what happens is the closed over variables are not garbage collected. So, in simpler words, it is accumulating a lot of memory space to create a lot of closures in our program and if not handled properly there can also be memory leaks.

Q13. What is Garbage collector in JS?
=> Garbage collector is like a program in the browser or JS Engine that freezes the unutilized memory. For example, like the unused variables it takes out of the memory, it kind of freezes the memory when it no longer needed.
. Unused variables are automatically deleted in High-level Programming language by Garbage collector.

DATE________

Q14. Relation b/w Garbage Collection, Memory leaks and Closures ?

=>
```
function a() {
    var x = 0;
    return function b() {
        console.log (x);
    }
}
var y = a();
y();
```

=> So now this function b() forms a closure with 'var x', so once this function a() execution has finished, 'x' should have been garbage collected but because this variable 'x' has formed a closure with function 'b' and has returned, that means it still can not free up 'b' until we know that we no longer needed, so if we form more closures it kinda of accumulates more memory and those variables are not garbage collected.

DATE___________

Q5. Example of Smart Garbage collection by V8 JS Engine in Chrome.

=> Some browsers ~~like~~ V8 now have smart garbage collectors that automatically delete variables that are not used outside closure.

Ex:
```
function a() {
    var x = 0;  var z = 1;
    return function b() {
        console.log (x);
    }
}
var y = a();
y();
```

=> So, variables like 'z' which are ~~used~~ not used in 'function b()', although it does closure with 'x' and 'z' but when returned outside if 'z' is not used inside funcⁿ b() scope, it will get automatically deleted by these smart garbage collectors.