**If you don not have understanding of Closures, I highly recommend to go through below articles first**.

- [Part 11: Closures in JS 🤷🏻‍♀️](https://medium.com/@swati.developer17/part-11-closures-in-js-%EF%B8%8F-e0874a86b2de)

- [Part 12: setTimeout + Closures Interview Questions 🤓](https://medium.com/@swati.developer17/part-12-settimeout-closures-interview-questions-cfa0d402831f)

**Question 1: What are closures In JS?**  
A function alongs with it’s reference error together forms a Closure.

**Question 2. How you’ll explain more about it?**

* Each and every function in JS have access to it’s outer lexical environment that means it has access to variables and functions which are in environment of it’s parent.
* Even If this function is executed in some other scope instead of original scope, it will still remember it’s outer lexical environment where it was originally present in the code.

**Question 3. What about an example to explain all this?**

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

So, in above example, We have nested function *inner()*inside function*outer() . inner()*function have access to*variable a*which is present in it’s outer environment.

Also, when you return *inner()*function and invoke*outer() f*unction, it will still console value of*variable a .*So, it remembered it’s outer lexical env even when called in other scope.

So, *inner()*function*=****Closure***💡

**Question 4. What is** *outer()()***? Why there are two parenthesis?***Two parenthesis are used to call inner function.*

Example :  
- *outer()*will return below output:



- *outer()()*will return below output:



Also, This can be done in other way as well

function outer(){  
 var a =10;  
 function inner(){  
 console.log(a);  
 }  
 return inner;  
}  
  
var close = outer(); // this will give you outer function  
close(); // this will give you inner function and equivalent

**Question 5. What if***var***declaration is moved to Line 5, just before return statement. What will happen? Will it still be a Closure?**

function outer(){   
 function inner(){  
 console.log(a);  
 }  
 var a =10;  
 return inner;  
}  
  
var close = outer(); // this will give you outer function  
close(); // this will give you inner function and equivalent

Answer is **YES, it will still be. Closure.**It doesn’t matter what the sequence is, means where the statement is. For closure, all matter is function has access to it’s lexical environment. Also, output will be same.

**Question 6. What if***var****is replaced by****let***in Line 5. What will happen?**

function outer(){   
 function inner(){  
 console.log(a);  
 }  
 let a =10; // let instead of var  
 return inner;  
}  
  
var close = outer(); // this will give you outer function  
close(); // this will give you inner function and equivalent

Answer is **It won’t affect anything.** It ‘ll still be a closure and output will be same.

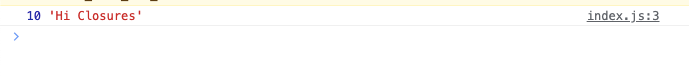
**Question 7. What if** *outer()***have a parameter? What happens then?**

function outer(b){   
 function inner(){  
 console.log(a, b); // try printing b here  
 }  
 let a =10; // let instead of var  
 return inner;  
}  
  
// Passing argument here   
var close = outer("Hi Closures"); // this will give you outer function  
close(); // this will give you inner function and equivalent

Answer is **It won’t affect anything.** It ‘ll still be a closure and we can try printing value of *b*as well in console.

Reason: *b*is also the part of outer environment

Output will be:



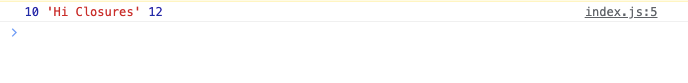
**Question 8. What if***outer()***function is nested into another function? Will***inner()***function have access to***outer()***function’s environment as well?**

function outest(){  
 var c = 12;   
 function outer(b){   
 function inner(){  
 console.log(a, b, c); // try printing b and c here  
 }  
 let a =10; // let instead of var  
 return inner;  
 }  
 return outer;  
}  
  
  
// Passing argument here   
var close = outest()("Hi Closures"); // this will give you outer function  
close(); // this will give you inner function and equivalent

Answer is **YES,***inner()***function have access to***outer()***function’s environment as well.**

Try printing *variable c*inside*inner()*function*.*

Output will be :



**Question 9. What if we have global variable with conflicted name?**

function outest(){  
 var c = 12;   
 function outer(b){   
 function inner(){  
 console.log(a, b, c); // try printing b and c here  
 }  
 let a =10; // let instead of var  
 return inner;  
 }  
 return outer;  
}  
  
let a =100;  
// Passing argument here   
var close = outest()("Hi Closures"); // this will give you outer function  
close(); // this will give you inner function and equivalent

*inner()*function has a closure with *variable a*on **Line 7** i.e*let a = 10*.So the*inner()*function has *reference for variable a*is at Line 7.  
*let a = 100;*at **Line 13** is in different scope. So, it will print output with value 10.

But **if we comment Line 7,***then it will go to outer environments looking for a and prints 100.*Check below:

function outest(){  
 var c = 12;   
 function outer(b){   
 function inner(){  
 console.log(a, b, c); // try printing b and c here  
 }  
 //let a =10; // let instead of var  
 return inner;  
 }  
 return outer;  
}  
  
let a =100;  
// Passing argument here   
var close = outest()("Hi Closures"); // this will give you outer function  
close(); // this will give you inner function and equivalent

Also, **if we comment Line 13,***then it will print “ReferenceError: a is not defined”.*Check below:

function outest(){  
 var c = 12;   
 function outer(b){   
 function inner(){  
 console.log(a, b, c); // try printing b and c here  
 }  
 //let a =10; // let instead of var  
 return inner;  
 }  
 return outer;  
}  
  
// let a =100;  
// Passing argument here   
var close = outest()("Hi Closures"); // this will give you outer function  
close(); // this will give you inner function and equivalent

**Question 10. What are the advantages of Closure?**

1. Module Design Patterns
2. Currying
3. Function Like once
4. memoize
5. maintaining state in async world
6. setTimeouts
7. Iterators
8. Data Hiding and Encapsulation

**Question 11. What are disadvantages of Closure?**

1. There could be over consumption of memory because every time a closure is formed, it consumes a lot of memory.
2. Those closed over variables are not Garbage Collected, so it means it is accumulating a lot of memory
3. If not handled properly, it can lead to memory leaks.

**Question 11. What is Garbage Collector?**

Garbage Collector is a program in browser or in JS engine, which free up unutilised memory.  
In languages like c, c++, it is upto developers how we allocate and deallocate memory but in high level programming language like JS, most work is done by JS engine.  
So, whenever there are some unused variables, Garbage collector takes those out of memory

**Question 11. How Closures and Garbage Collector related to each other?**

function a(){  
 var x = 10;  
 return function b(){  
 console.log(x);  
 }  
}  
var y = a();

In above example, *function b()*is closure. So, ideally after executing*function a()*, all it’s memory including*variable x*should be garbage collected. It wasn’t needed anymore. But because *function b()*is closure and*variable x*is being referred and assigning value *variable y*, so it won’t free up as later on , we might call*y() .*

This way more closures means keeping a lot of memory accumulating.

Although, some modern browsers like chrome and JS engines like V8, they have smart garbage collection mechanism, in which it somehow identify unreachable variables and collects these garbage variables.

**Question 13. What is Smartly garbaged variable, examples?**

function a(){  
 var x = 10, z = 20;  
 return function b(){  
 console.log(x);  
 }  
}  
var y = a();

In above example, *variable z*is not being used . So when*function b()*is returned ,*variable z*will be smartly garbaged but*variable x*is not. So, *variable z*won’t be in memory.