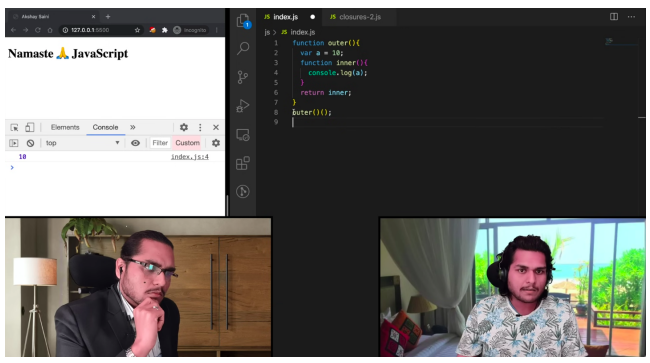
Closure Interview Questions

Friday, 18 August 2023 12:20 PM

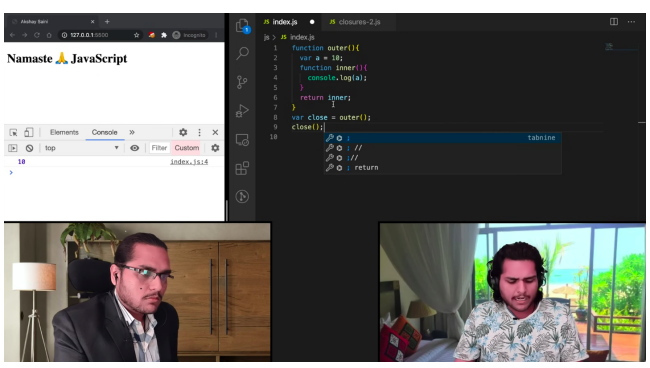
Q: What is a closure?



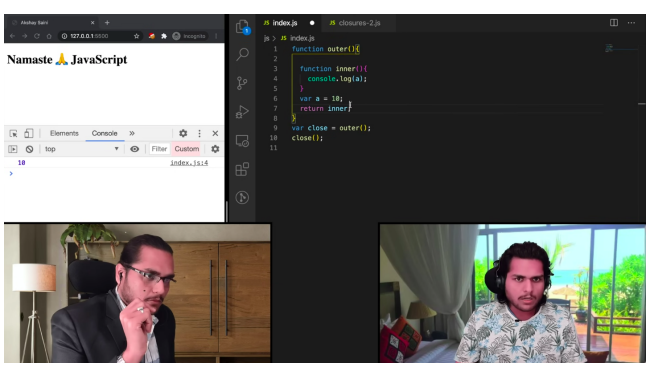
(eg explaining closure).

→ A closure is a fⁿ bundled together with the lexical environment of its parent.
So, wherever we call that fⁿ, it remembers the reference of its parent.

Note: In above, we can see that we used double parenthesis, this is used to invoke the inner fⁿ. There is no difference b/w the above code and the code below:

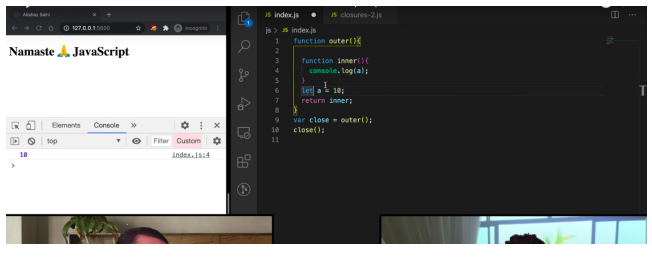


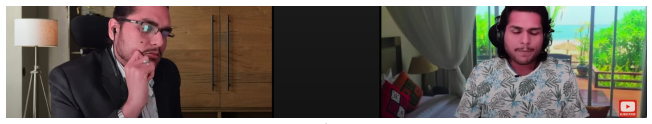
Note: If we move line 2 statement below line 5:



Still there will be no difference, and inner() will still form a closure with outer(), it will work the same way.
Inner fⁿ forms a closure with the outer() fⁿ but it does not do this in any particular order.

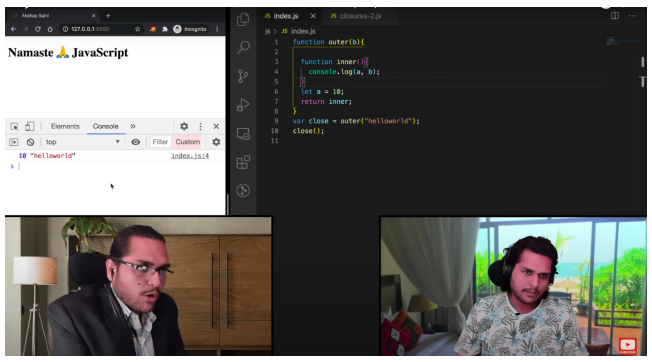
Note: If we use let instead of var, still it behaves the same way. Just the difference is that we cannot access 'a' outside the block directly but still inner will have a reference

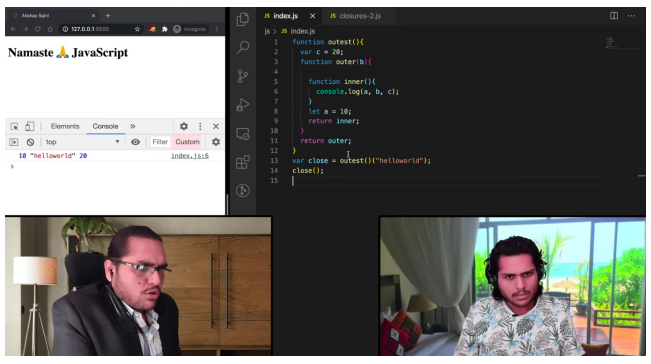




of 'a' and will form a closure and 'a' will not be in global scope but will be in block scope.

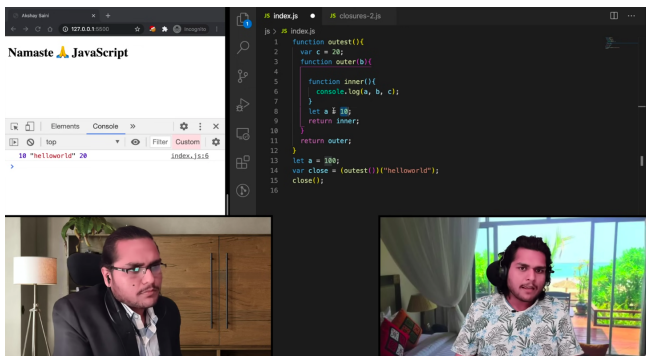
Note: If we pass a parameter inside outer fn, then also the program behave in the same way and value of 'b' is printed as when inner() forms a closure with outer() it also has access to outer()'s parameters.





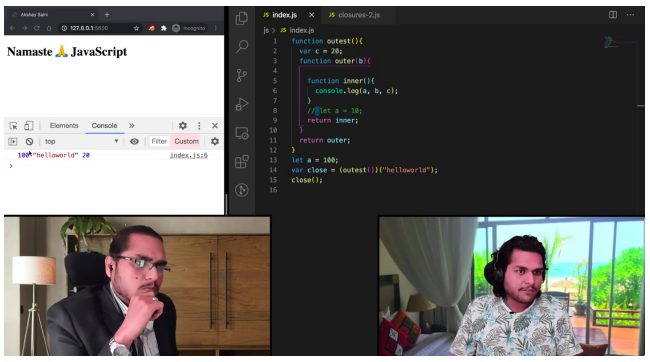
If we wrap the previous code inside another fn, then also inner() fn will form closure with outest() and it will have access to the lexical environment of outest().

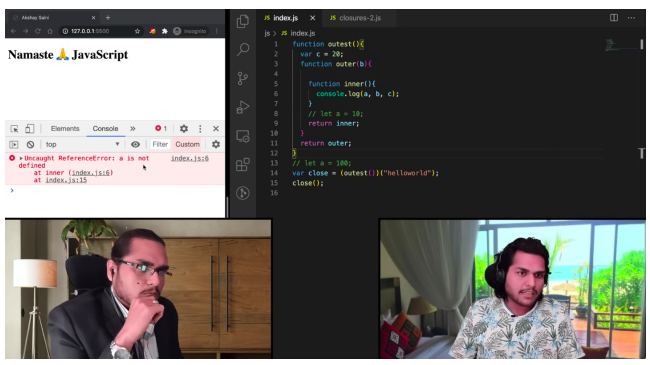
In order to explain like 13, we can say that firstly, outest() is called and it returns outer(). Once we get that, (outer())("hello world") is called with the argument passed into it. This returns inner() and finally inner() is called and everything is printed.



If we declare let a=100; in global scope then what happens? Then also, the output will be same as when inner() forms a closure with outer() it has the lexical environment of outer() and so it has the value of 'a' as '10' and it points to this reference of 'a'.

→ But if there was nothing on line 6, then it would have printed 100 as JS engine first tries to find value of 'a' inside inner()'s local memory, if it can't find it there then it goes to the lexical environment of its parent i.e outer(), after that it goes to lexical environment of outest(), if it can't find value of 'a' there then it goes to the global scope & prints 100. If JS engine can't find value of 'a' in global scope also then it gives us reference error: a is not defined.





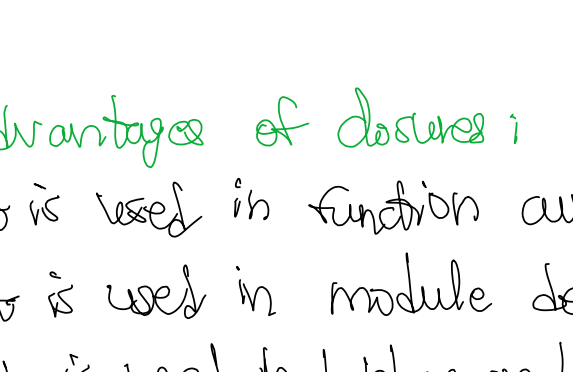
→ Advantages of closures:

- It is used in function currying.
- It is used in module design pattern.
- It is used in higher order f^n's like memoize(), once().
- It helps us in data hiding and encapsulation.

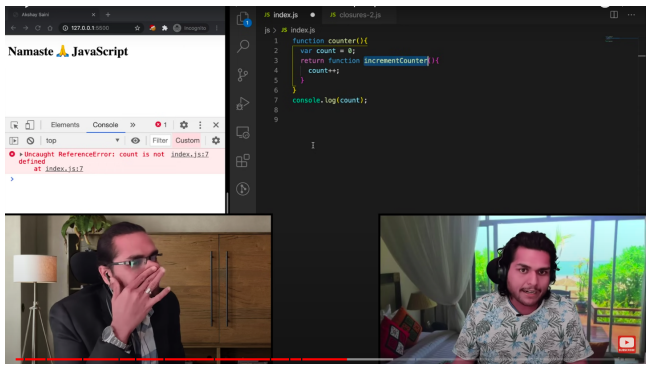
Q. What is data hiding?

ans. If we don't want other f^n's and other pieces of code to have access over some data, then it is called data hiding.

eg:



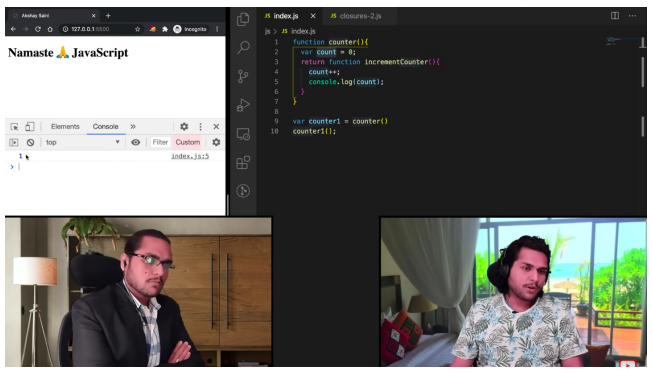
→ Here we can see that there's no data hiding, and anybody can access counter and change its value.



→ In order to implement data hiding, we can wrap above code inside a f^n. So if we try to access the variable now outside the f^n, it gives us reference error.

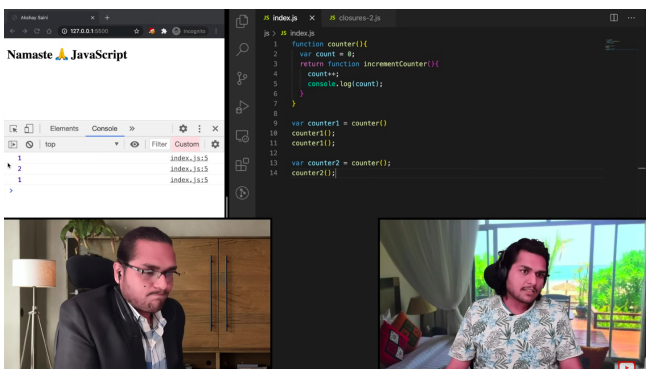
→ So now whenever we need to access the count variable, we will call the f^n and store it inside another variable. Now since

... increment Counter() f^n forms closure with counter() f^n, so we will have access to "count" variable.

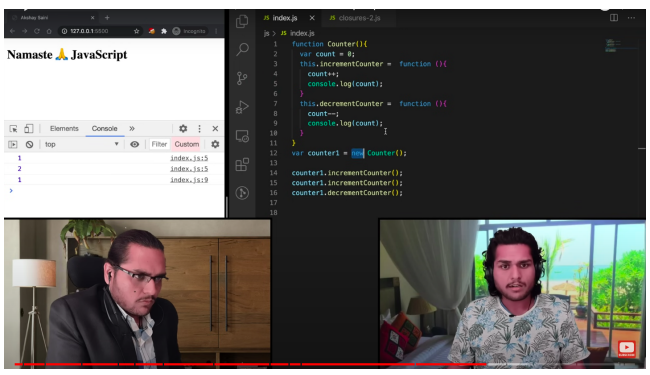


→ So here we have implemented data hiding.

→ this also means that if we call counter() f^n & store it in a new variable, then it won't touch the previous variable's scope instead it will again create a closure with a new copy of count. (Since, var is f^n scoped and a new local EC is made for each invokation)



Q: How can we scale the above code?



→ In order to scale the previous code, we can use constructor functions. A constructor f^n starts with a capital letter.

→ In order to invoke a constructor f^n, we must use new keyword.

→ the f^n's declared on line 3 & line 7 are anonymous f^n's.

→ on line 12, we get access to incrementCounter and decrementCounter variables while still the data is private and these f^n's still form a closure.

→ Disadvantages of closures:

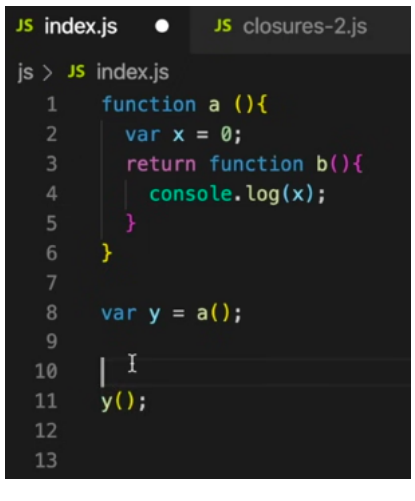
- There could be over-consumption of memory when we use closures because when a closure is created, a lot of memory is consumed as the variables that are closed over are not garbage

collected until the program is over.
If not handled properly it can also lead to memory leaks and can also freeze the browser.

→ Garbage collector: It is like a program in the browser or in the JS engine which frees up the unutilised memory.
JS is a high level language so we don't have to deallocate unused memory ourselves.

→ Relation between garbage collection and Closures:

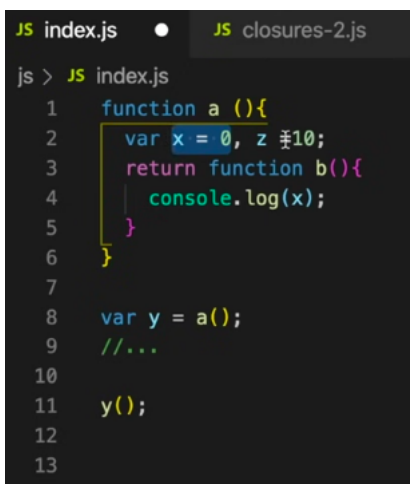
→ In this code, once fn is invoked, JS engine frees up the memory as variables inside that fn are no longer required.



→ But in this case, since a() is stored inside variable 'y', and also fn b() is closed over variable 'x', so variable 'x' can now be accessed anywhere when we invoke y(), & due to this JS engine can't free up that memory.

Note: JS V8 engine and some modern browsers like Chrome do smart garbage collection.

eg:



→ Like here, we can see that 'z' won't be used even when its closed over by b(), so JS engine frees it up from memory.
This is smart garbage collection.