Callback Functions in JS ft. Event Listeners

Friday, 18 August 2023 10:26 PM

→ Callback functions: We can pass a fⁿ into another fⁿ as an argument in JS. So, that fⁿ that is passed into another fⁿ is known as a callback fⁿ.

→ Callback fⁿs allows us to do async things in JS, even though JS is a synchronous single-threaded language.

eg:

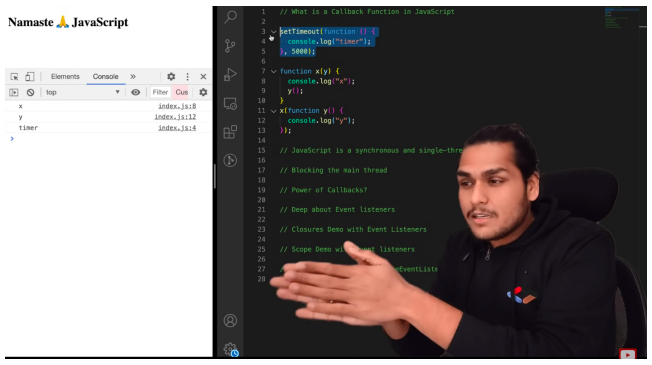
```
function x(y){

}
x(function y(){

})
```

→ We call them callback fⁿs, because we are passing them as an argument to another fⁿ, so the fⁿ which is receiving them as parameters has the control as to when to call them. Like in above 'x' has control over 'y' and 'x' will decide when 'y' is called back, hence 'y' is a callback fⁿ.

eg: setTimeout also takes in a callback fⁿ as its first parameter.
// This is an example of asynchronous behaviour also as setTimeout, when JS encounters setTimeout it doesn't wait for its timer to expire, instead it stores the callback fⁿ somewhere & attaches a timer to it and goes on executing the next lines of code

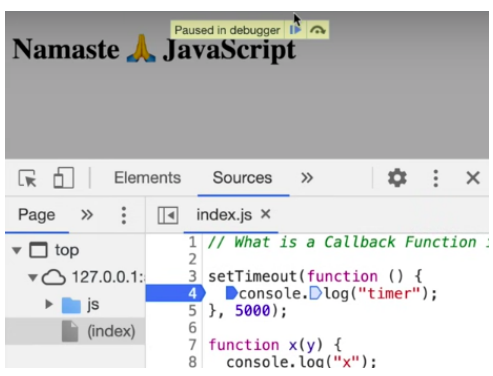
Note: The functionality setTimeout gives us wouldn't have been possible without callback fⁿ.

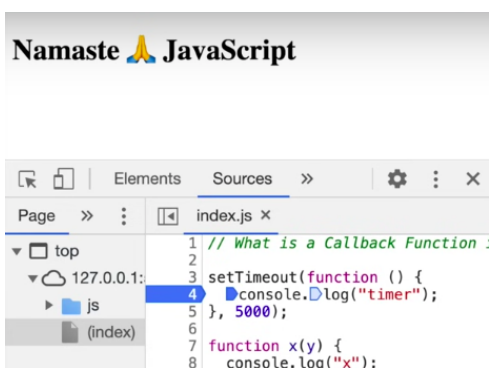
eg:

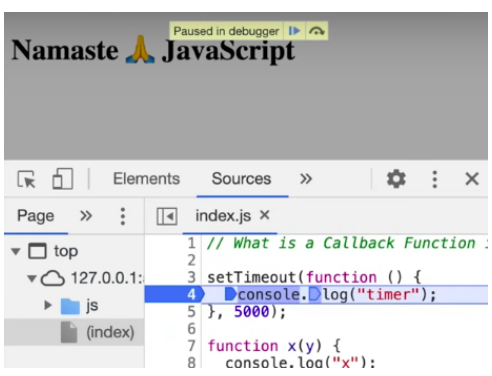


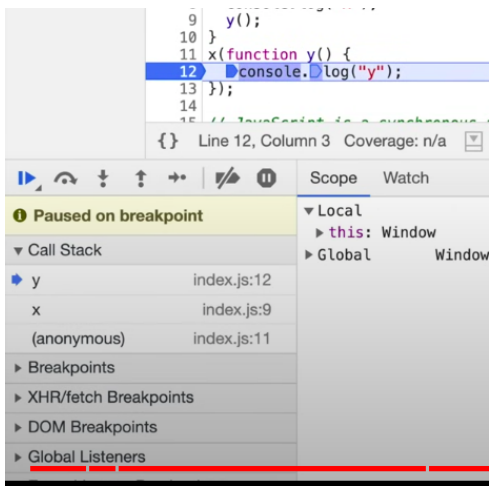
→ This is self explanatory, JS waits for none and after storing the callback fⁿ along with its timer somewhere else, it executes later lines of code & then after 5 secs finally prints 'timer'.

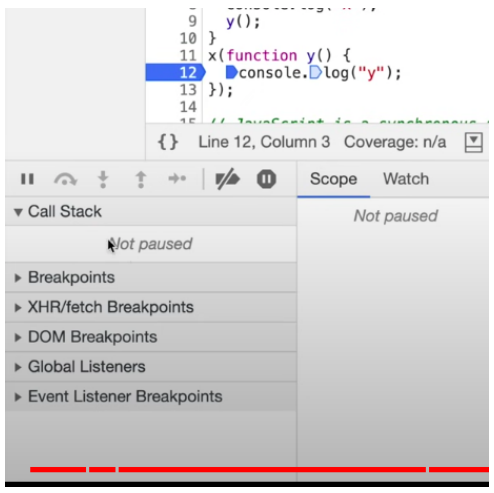
→ Browser demo of the above code:

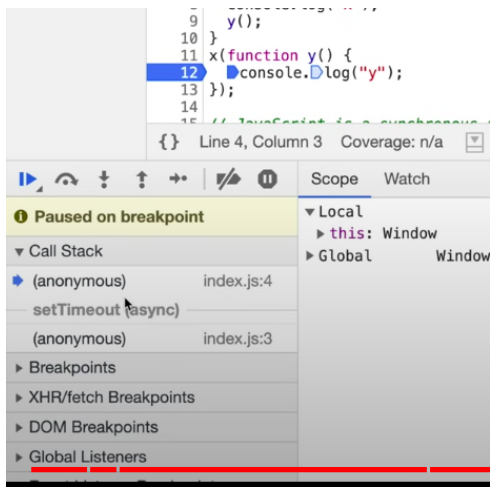








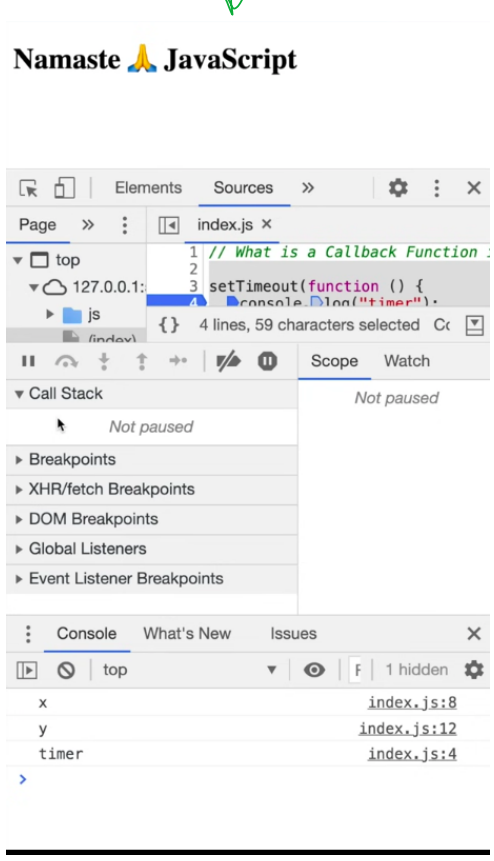




→ So, what happened is that call stack contains GEC & EC's of x & y.

→ Then the call stack becomes empty after x() & y() are executed.

→ Then after 5 secs, call stack contains the memory of the callback fn until finally the whole call stack becomes empty due to end of the program execution.



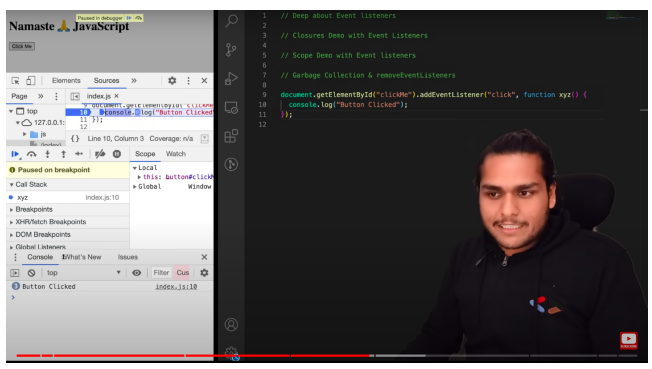
Note: JS contains only 1 callstack which is the main thread. Everything happens in this callstack only.

So if any operation blocks this callstack then that is called as blocking the main thread.

If, for example, fn x() takes 20-30 secs to execute then since JS only has 1 call stack so it won't execute anything else for those 20-30 secs. That's why we should never block the main thread & we should always use asynchronous operations for things which take time.

Eg: setTimeout is async, so it does not block the main thread.

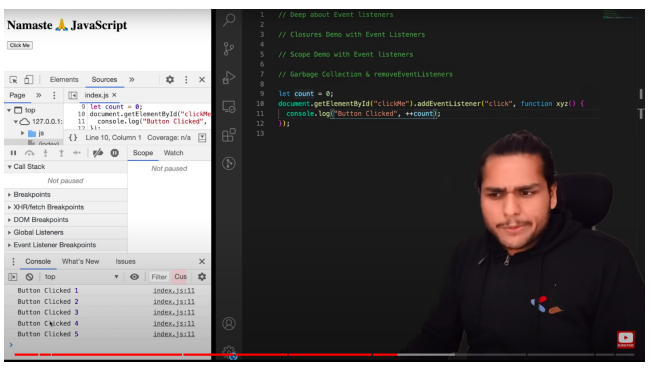
→ Event Listeners :



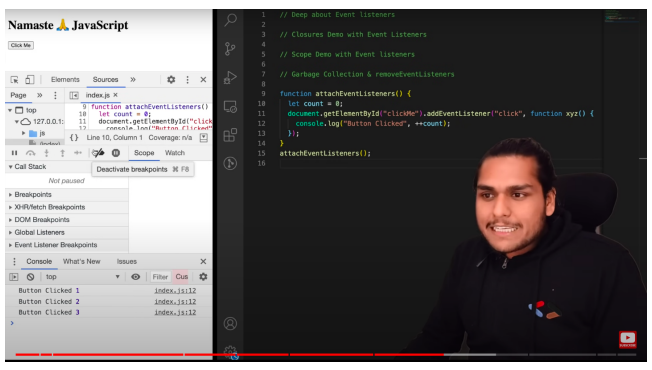
The above code means that, an event listener takes in a fn which

is a callback f^n. We need a callback f^n here because we need to do async stuff and we need the f^n to be called back once there is an event of button (whose id is clickMe) click. We can see in the callstack that it is filled only when the button is clicked otherwise it is empty initially. Therefore, the callback f^n is being called on the event of button click.

→ Counting the no. of times a button is clicked:



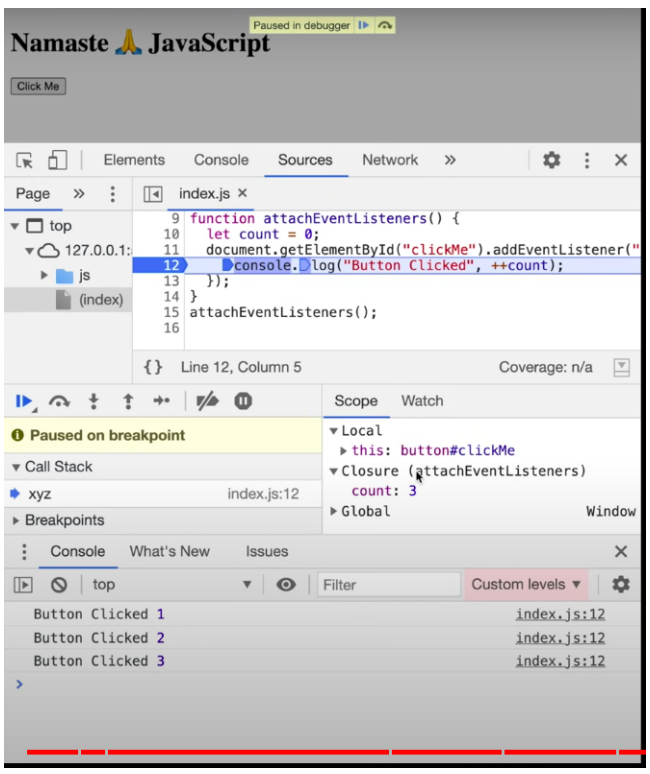
→ This is correct, but we know that we mustn't use global variables. So, we will use the concept of closures to make count secure.



→ Therefore, we wrap the above code inside a f^n, so that count is not accessible to everyone. This promotes data hiding.

→ Now, the callback f^n forms a closure with count and so count is only accessible through the callback f^n.

→ Browser demo of above:

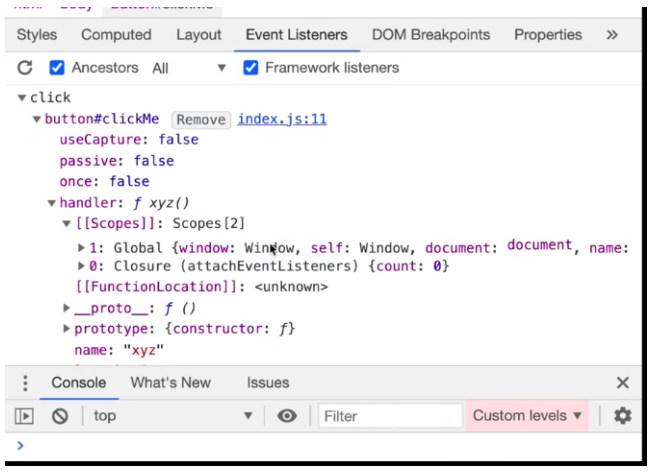


→ When we put a debugger, we can see that 'xyz' is pushed to call stack on the event of button click.

→ We can also see that the callback f^n forms a closure with its parent f^n.

Note:

→ If we click on elements, then on clicking a particular element we can see the event listeners attached to



it.

→ In the handler, we can see the scopes which are global and closure with parent fn.

→ Garbage collection and remove event listeners :

Whenever we attach an event listener, it's callback fn forms a closure with the parent fn. So event listeners are memory heavy. Even though the callstack remains empty due to the async nature of callback fns, but JS engine can't free up the memory due to the formation of closure of callback fn with its parent fn.

So due to so much memory consumption, a good practice is to remove the event listener.

On removing the event listener, all the memory that the closure was occupying would be garbage collected.