

Callback Functions in JS ft. Event Listeners

Friday, 18 August 2023 10:26 PM

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

→ Callback f^n 's allows us to do async things in JS, even though JS is a synchronous single-threaded language.

eg:

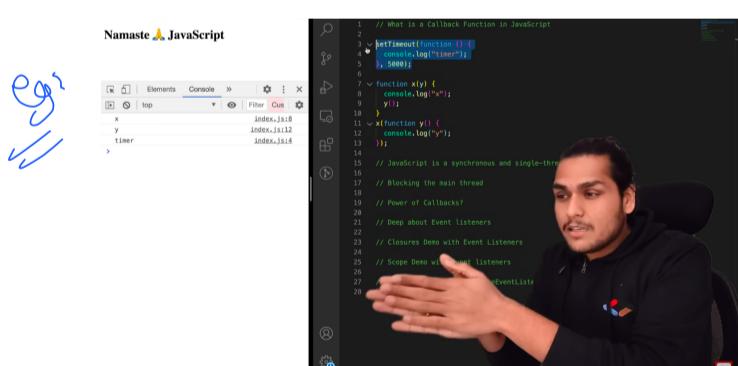
```
2
3     function x(y){}
4         y
5     x(function y(){})
6
7
8 }
```

→ We call these callback f^n s, because we are passing them as an argument to another f^n , so the f^n 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 f^n .

eg: setTimeout also takes in a callback f^n as its first parameter.

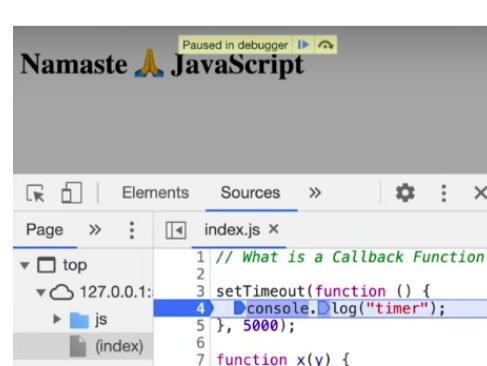
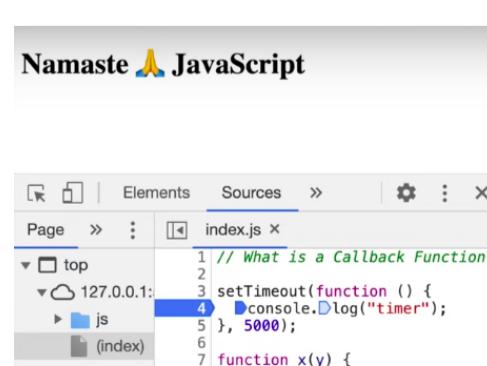
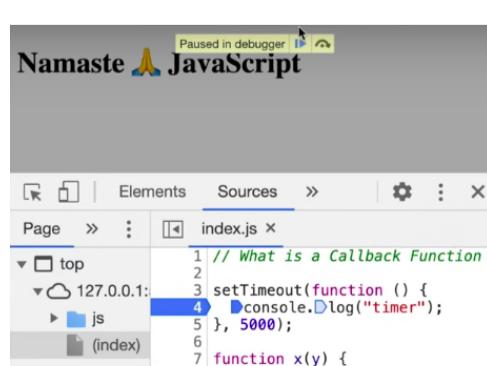
→ This is an example of asynchronous behaviour also as setTimeout, when JS encounters setTimeout it doesn't wait for its timer to expire, instead it stores the callback f^n somewhere & attaches a timer to it and goes on executing the next lines of code.

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



→ This is self-explanatory, JS waits for none and after storing the callback f^n along with its timer somewhere else, it executes later lines of code & then after 5 sec finally prints "timer".

→ Browser demo of the above code:



The first screenshot shows a breakpoint at line 12 of index.js. The call stack shows two frames: 'y' at index.js:12 and 'x' at index.js:9. The second screenshot shows the call stack has been cleared, indicating both functions have run. The third screenshot shows a setTimeout function at index.js:4, with a note 'JavaScript is a synchronous' above it.

→ So, what happened is that call stack contains CSE & ECs 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.

Namaste JavaScript

The screenshot shows a setTimeout function in the call stack at index.js:4. The console tab shows variables x, y, and timer defined at index.js:8, 12, and 4 respectively.

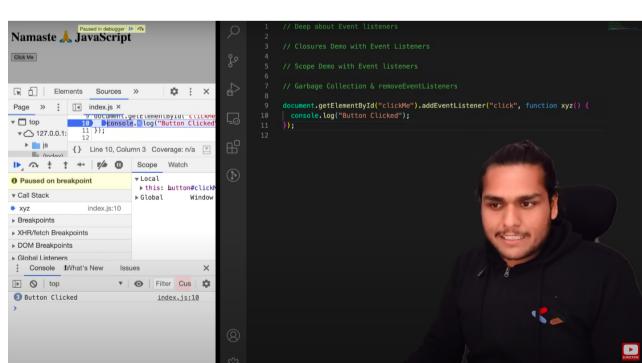
Note: JS contains only 1 call stack which is the main thread. Everything happens in this call stack only.

So if any operation blocks this call stack 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.

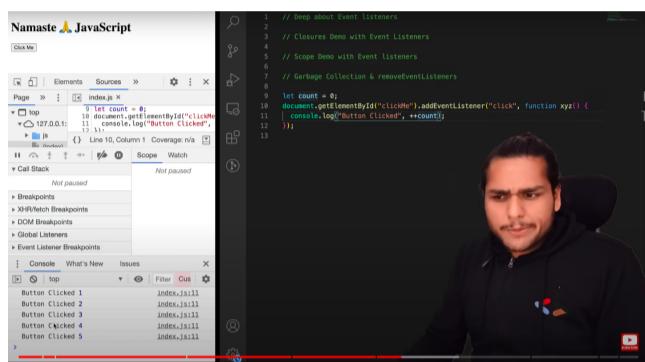
→ Event Listeners :



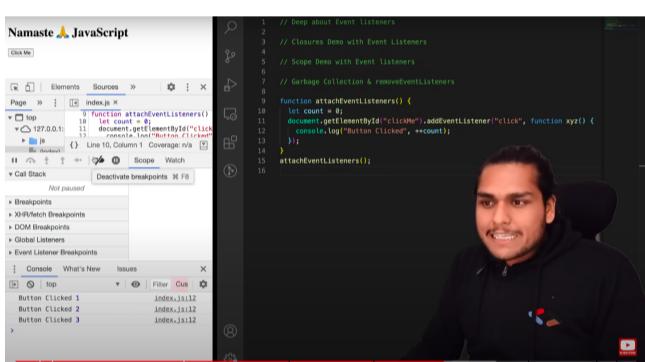
The above code means that, an **event listener** takes in a fn which

is a callback fn. We need a callback fn here because we need to do async stuff and we need the fn 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's empty initially. Therefore, the callback fn is being called on the event of button click.

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



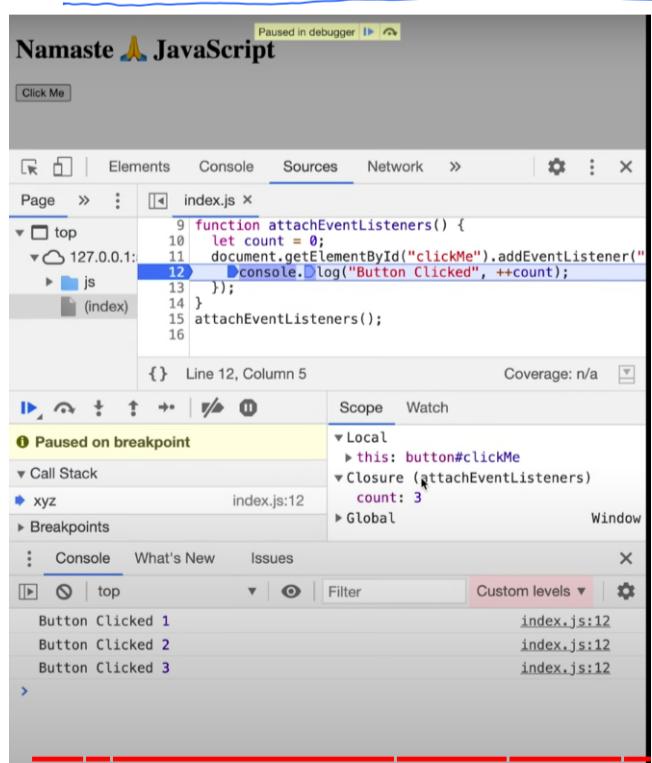
→ This is correct, but we know that we mustn't use global variables. So, we will use the concept of closures to make count secure.



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

→ Now, the callback fn forms closure with count and so count is only accessible through the callback fn.

→ 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 fn forms a closure with its parent fn.

Note:

Namaste 🙏 JavaScript



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

```
Styles Computed Layout Event Listeners DOM Breakpoints Properties >
C  Ancestors All  Framework listeners
click
button#clickMe Remove index.js:11
useCapture: false
passive: false
once: false
handler: f xyz()
[[Scopes]]: Scopes[2]
> 1: Global {window: Window, self: Window, document: document, name: 'xyz'}
> 0: Closure (attachEventListeners) {count: 0}
[[FunctionLocation]]: <unknown>
__proto__: f ()
prototype: {constructor: f}
name: "xyz"
Console What's New Issues
top Filter Custom levels
```

in the world we are connected 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, its callback fn forms a closure with the parent fn. So event listeners are memory heavy. Even though the call stack remains empty due to the async nature of callback fn's, 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.