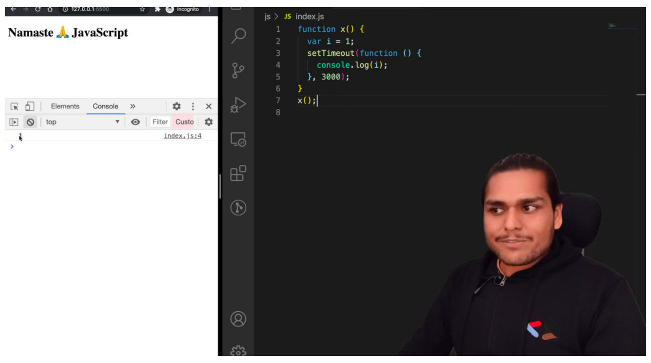
setTimeout + Closures in JS

Thursday, 17 August 2023 11:42 PM

eg:



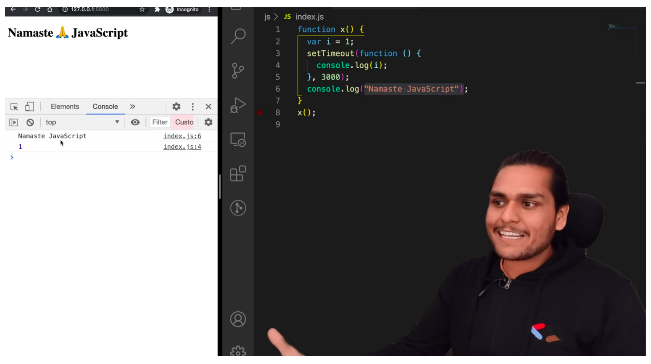
Namaste 🙏 JavaScript

```
function x() {
  var i = 1;
  setTimeout(function () {
    console.log(i);
  }, 3000);
}
x();
```

```
1                                  index.js:4
```

→ This is a simple example of setTimeout, where 'i' is printed after 3 sec.

eg:



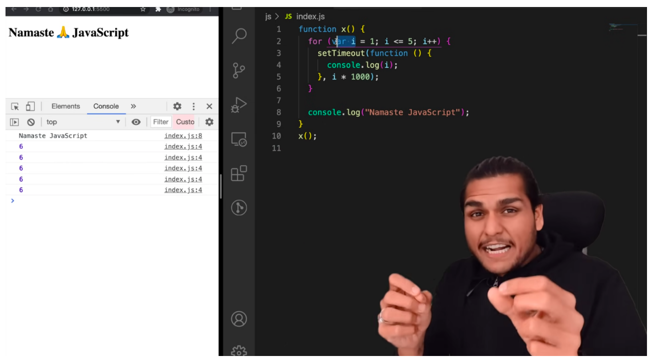
Namaste 🙏 JavaScript

```
function x() {
  var i = 1;
  setTimeout(function () {
    console.log(i);
  }, 3000);
  console.log("Namaste JavaScript");
}
x();
```

```
Namaste JavaScript                 index.js:6
1                                  index.js:4
```

→ In above 'namaste JavaScript' is printed before 'i' because JS waits for no one.
When JS encounters a setTimeout callback fn, it stores the fn somewhere and executes the line below that, and when 3sec are completed it prints 'i'. Now the callback fn has the reference of fn x()'s lexical environment so it behaves like a closure and can have reference of 'i' anytime.

→ Tricky JS Question:



Namaste 🙏 JavaScript

```
function x() {
  for (var i = 1; i <= 5; i++) {
    setTimeout(function () {
      console.log(i);
    }, i * 1000);
  }

  console.log("Namaste JavaScript");
}
x();
```

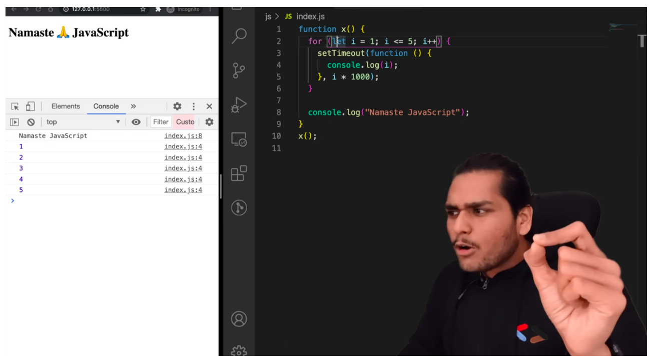
```
Namaste JavaScript                 index.js:8
6                                  index.js:4
6                                  index.js:4
6                                  index.js:4
6                                  index.js:4
6                                  index.js:4
```

→ Now why do we get the above o/p?
→ This is because, firstly JS does not wait for anyone so when it encounters setTimeout it stores the fn somewhere and attaches a timer to it and goes on with its execution. Now, the callback fn inside setTimeout also has a reference of its parents lexical environment. So, it becomes a closure.
We know that 'var' is global scoped, so each callback fn has a reference of 'i' which is in global scope, so ultimately when the timer expires all callback fn's print '6' because they all store a reference to 'i' which has now been changed to '6' in

global scope due to the complete execution of 'for loop'.

→ So how can we solve the above situation?

→ We can solve it by using let instead of var.



Namaste 🙏 JavaScript

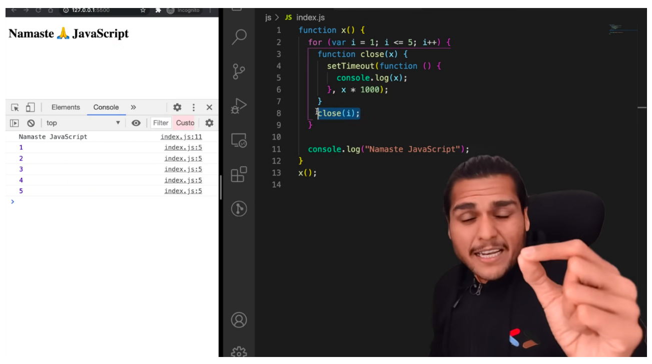
```
Namaste JavaScript        index.js:8
1                         index.js:4
2                         index.js:4
3                         index.js:4
4                         index.js:4
5                         index.js:4
```

```js
function x() {
  for (let i = 1; i <= 5; i++) {
    setTimeout(function () {
      console.log(i);
    }, i * 1000);
  }

  console.log("Namaste JavaScript");
}
x();
```

This solves the problem because let is block scoped, so for each callback f^n, a new copy of 'i' is created and stored as a reference inside the callback f^n.
And as let is block scoped, so each callback f^n has different values for 'i'.

→ If we need to solve this without using let then:



Namaste 🙏 JavaScript

```
Namaste JavaScript        index.js:11
1                         index.js:5
2                         index.js:5
3                         index.js:5
4                         index.js:5
5                         index.js:5
```

```js
function x() {
  for (var i = 1; i <= 5; i++) {
    function close(x) {
      setTimeout(function () {
        console.log(x);
      }, x * 1000);
    }
    close(i);
  }

  console.log("Namaste JavaScript");
}
x();
```

What we've done here is that we've created a separate copy of 'i' for each callback f^n using the concept of closures.
Now, each callback f^n has a copy of 'i' in its parent's lexical environment.