JS-3

Scope , functions and objects

Agenda

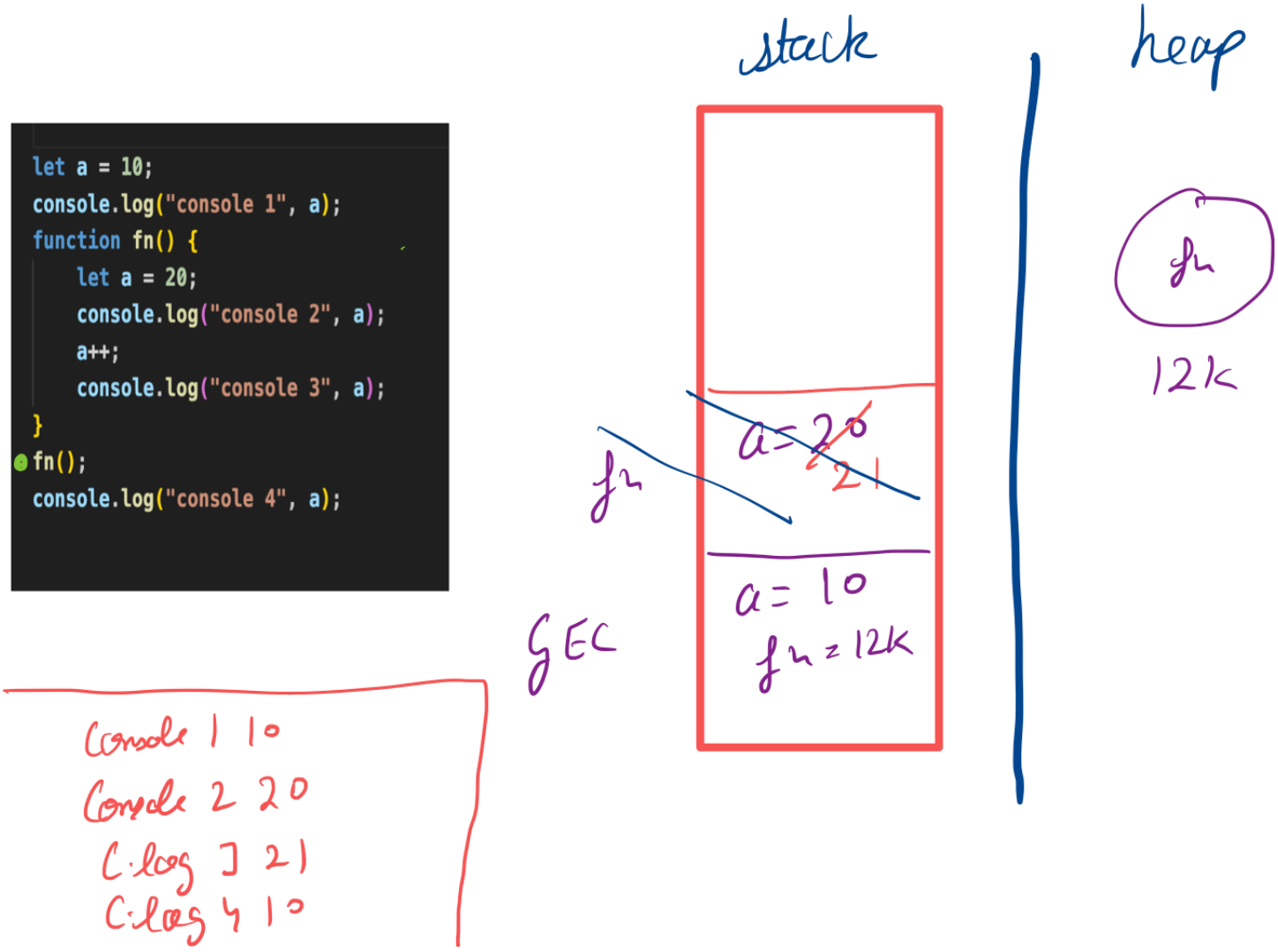
- block scope & Temporal dead Zone
- lexical scope
- functions and first class citizens

Block scope & Temporal dead Zone

Variables have there memory allocated when an execution context is created either when a GEC is created or a functio is called and it's removed from memory when function removed from the call stack

```
let a = 10;
console.log("console 1", a);
function fn() {
    let a = 20;
    console.log("console 2", a);
    a++;
    console.log("console 3", a);
}
fn();
console.log("console 4", a);
```

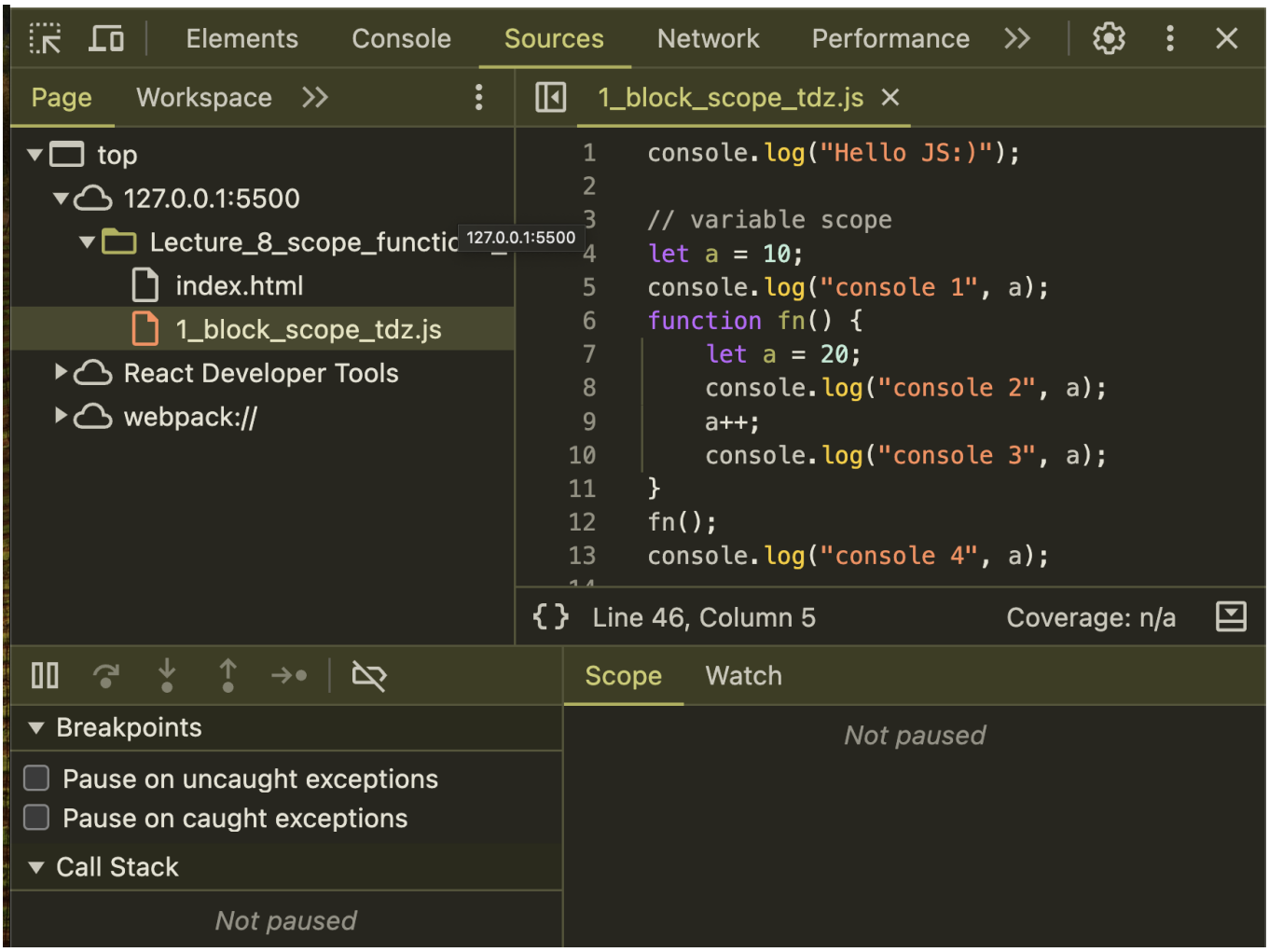
```
let a = 10;
console.log("console 1", a);
function fn() {
    let a = 20;
    console.log("console 2", a);
    a++;
    console.log("console 3", a);
}
fn();
console.log("console 4", a);
```



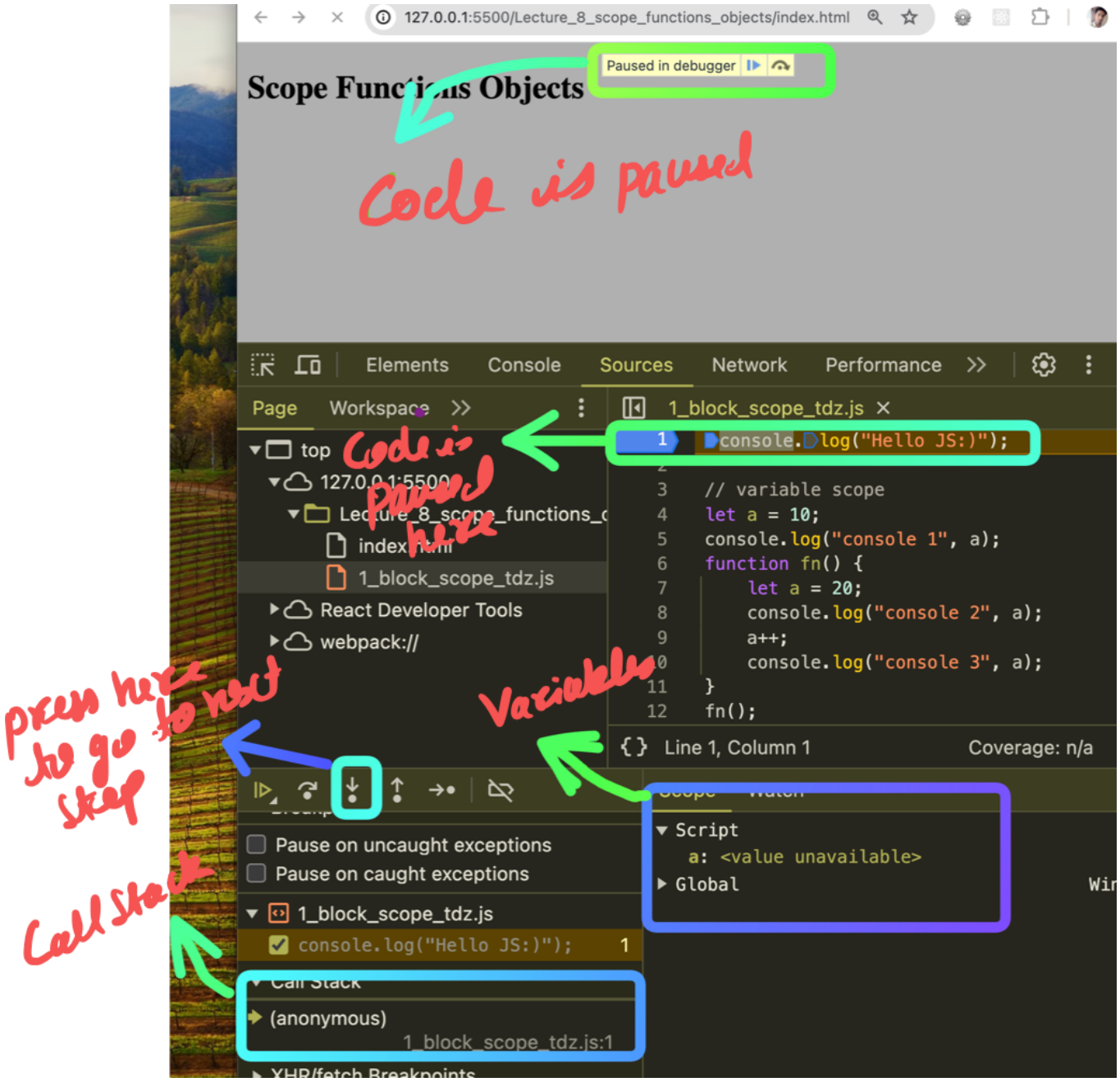
Debugging in JS

You can run your code and see how it executes in browser. here are the steps you need to follow to run the code step by step

- open the html file in browser
- open dev-tools
- go to sources tab
- go to js file you want to debug



- now click on the line from where you want to start the code execution step by step -> it is known as `break point` abd reload the page



let and block scope

`let is block scoped`

Que : what is block -> anything between two curly braces

block is created by function, loop , conditionals

Ouptut of following

```
let a=10
console.log(a) // output : 10
if (true) {
```

```
    let a = 20;
    console.log(a); // output : 20
    }
    conssole.log(a); // output :10
}```

Solution : as there is a =10 in the global area  so first console will give you 10 and again
inside the first if that creates a new block scope the next a will get 20 and if we move to the last line now we are outside the if block so we get the value of outer scope that is 10

### temporal dead zone

every let/const declared varaiable is dead -> temporal dead zone -> where you can't access it's value


### Lexical and outer scope

```js
***
 * EC :
 * a.creation
 *  1.    global code
 *              -> access to it's own variable and function
 *                  -> Hositing
 *                          var -> undefined
 *                          function -> memory
 *
 *  2.    function code
 *                  -> access to it's own variable and function
 *                  -> Hositing
 *                          var -> undefined
 *                          function -> memory
 *              -> window object
 *               -> outer scope
 * b.) execute
 * */
```
```

```
let a = 10;
console.log("value of a in global", a); // output : 10
function outer() {
    console.log("value of a in outer", a); // outpu: 10 (from outer
scope)

    function inner() {
        let a = 20;
        console.log("value of a in inner", a); // output : 20, from
current variables of inner
    }

    inner();
}
outer();
console.log("value of a in global", a); // output : 10 from GEC
```

In JS outer scope is defined by function defintion . In code the place where you have function definition you will check for outer variable from there that is why outer scope is also known as `lexically scope`

```
let varName = 10;
// function definition
function a() {
    console.log("inside ", varName); //10 because of function definition
determines the outer scope
}

function b() {
    let varName = 20;
    console.log("value of varName in b", varName);
    //    function call
    a();
    console.log("value of varName in b again", varName);
}
b();
```

functions and first class citizens

In JS functions are treated as first class citizens or in other words as variables

Let's see important behaviours of a variable

1. It can be assigned a value of reference

```
// variables -> assign ->reference , value
let arr = [10, 20, 30];
let arr2 = arr;
let a = 10;
let b = a;
```

2. It can be passed as to function

```
// variables -> can be passed as a parameter to a function
let arr = [10, 20, 30];
 function fn(params) {
     console.log("Hello Params", params);
 }
 fn(10);
 fn("Hello");
 fn(arr2);
```

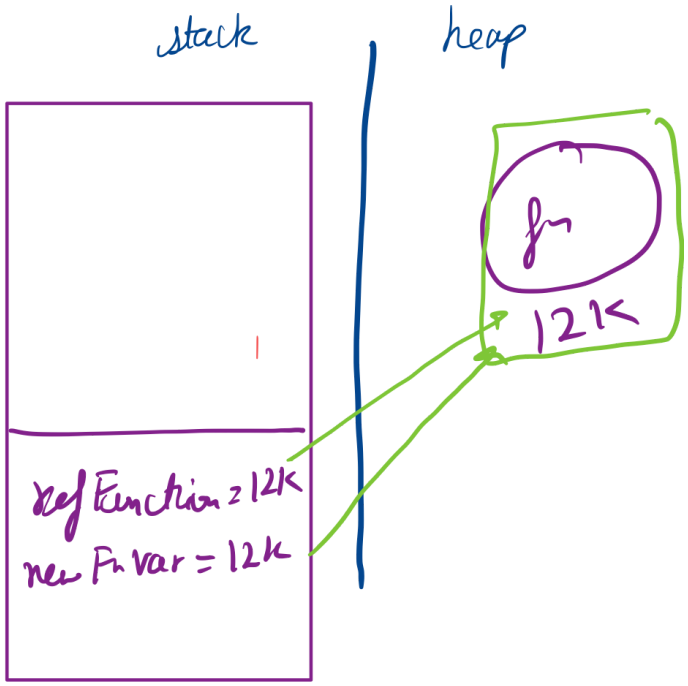
These two behaviours are also available in functions

1. function as a variable. -> these kind of functions are known as function expression

```
const refFunction = function () {
    console.log("Hello i am fuunction");
}

const newFNVar=refFunction;

newFNVar();
console.log("``````````````````");
refFunction();
```



2. You can also pass a function to variable

```
function bigger(paramFN) {
    console.log("Inside bigger")
    paramFN();
}

function smaller() {
    console.log("I am smaller");
}
bigger(smaller)```
```