**Block**: a block is defined by the curly braces {….}

{

     //this is perfectly a valid js code .

}

{

  //this is perfectly a valid js code .

  //  and when u run this, it does nothing, then y we need block.

  // block is also known as compound stmt.

  // a block is used to combine multiple js stmts into one group.

  //   eg:

  var a = 10;

  console.log(a);

  //as seen above , it is combining multiple js stmts into a group.

  //y we need to  group together all of these stmts-

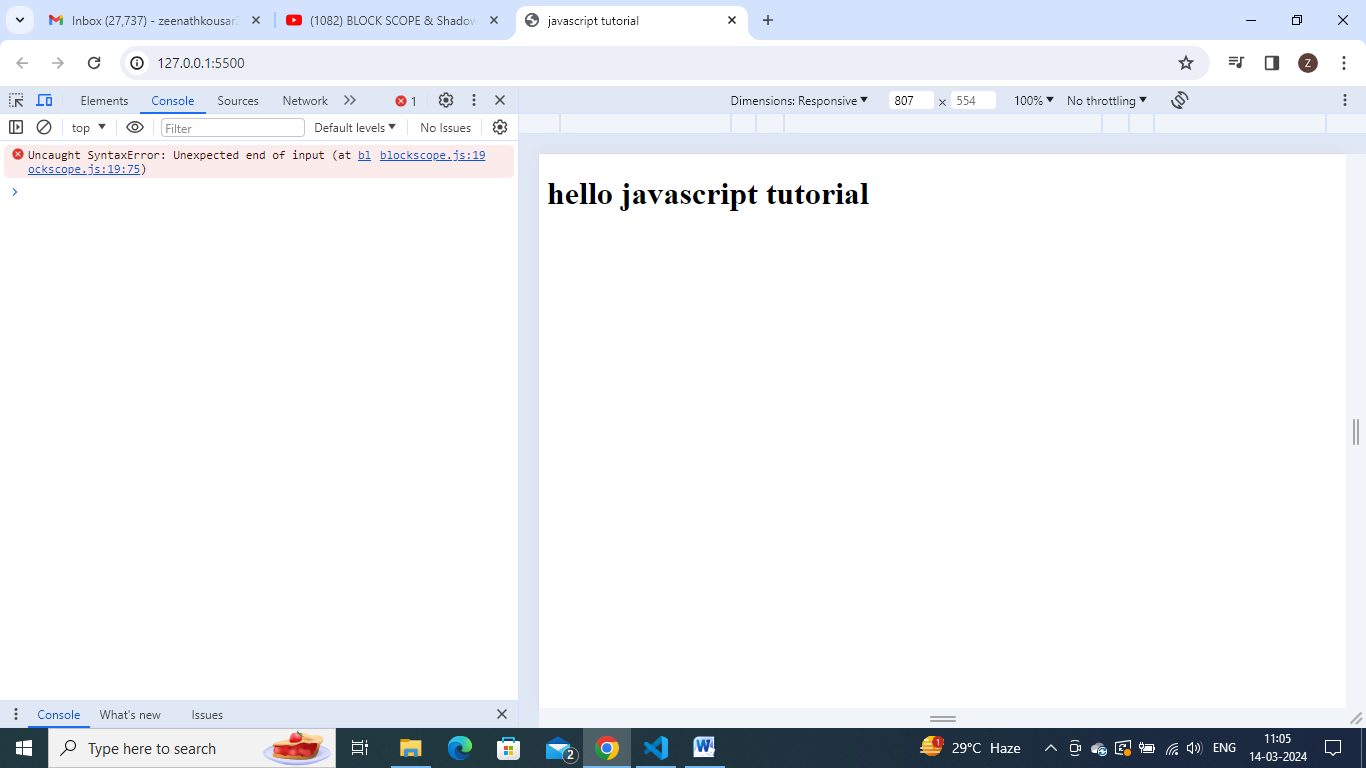
  //we need to group all these stmts together so that we can use  multiple stmts in a place where js expects only one stmt.

  //we group multiple stmts together in a block so that we can use it where js expects one stmt

}

// eg"

if(true)   // if i run this code it gives syntax err as - unexpected end of input



Y it is a syntax err- it is a syntax err bcoz , if expects a stmt(one stmt) over here.

So if only one stmt – we can write like this .

if(true) true;

if (true) true; //it is totally validd - now we get no err

// but suppose we need to write multiple stmts here- we can do it only by grouping them together in a block

if(true){

    var a = 10;

    console.log(a);

}

So this is the use of block.

//so this group of stms can be used in a place where js expects single stmt.

// so 'if' syntax doesnt have {} in itself , we can use it without curly braces also if only one stmt exists.

if (true) console.log("njdkjd"); //is a perfect valid code.

But if u need to write multiple stmts in place of it. We need to use block here.

This is the reason , we combine multiple stmts into a group and use it with for loop,while loop etc.

// Q. What is Block Scope.

// what are variables and functions we can access inside a {} .. is known as block scope.

// and to understand that, lets create 3 types of variables here , and lets see how they behave inside this block.

{

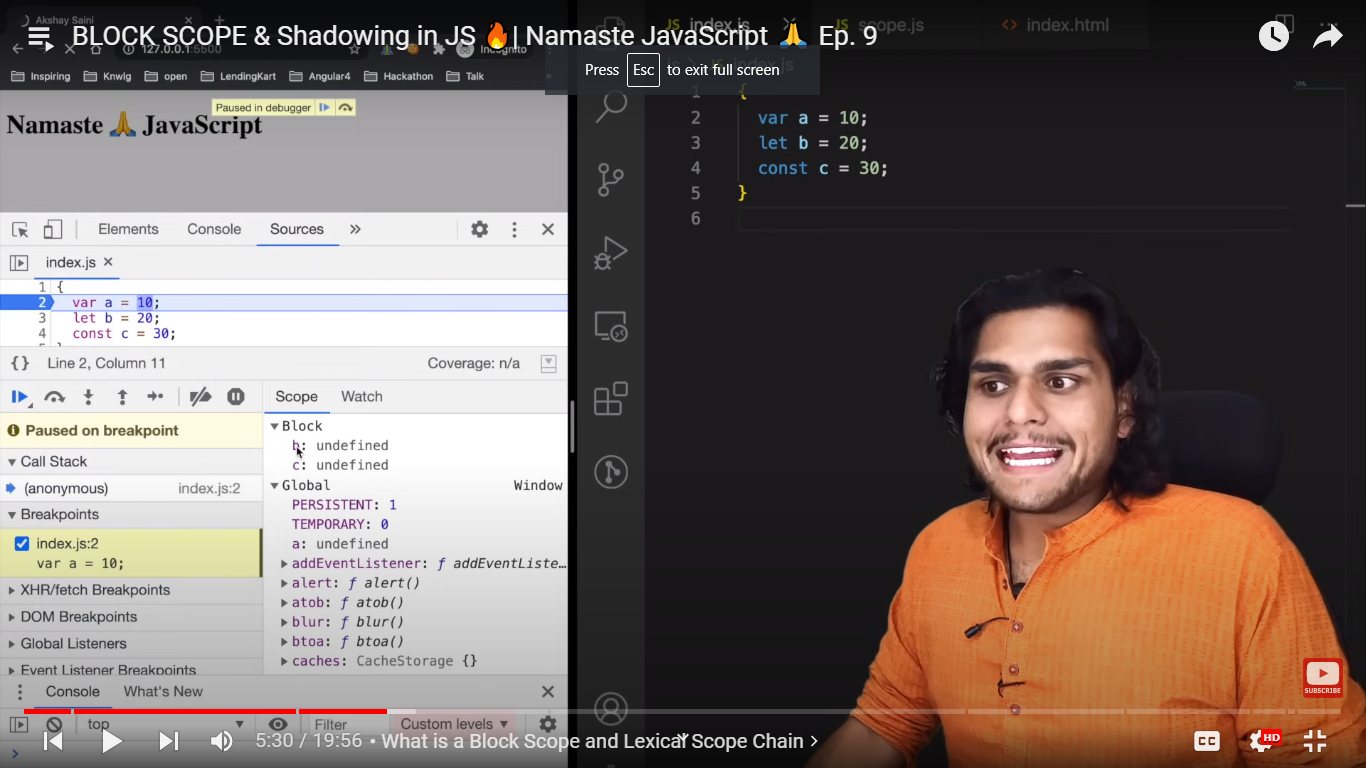
  var a = 10;

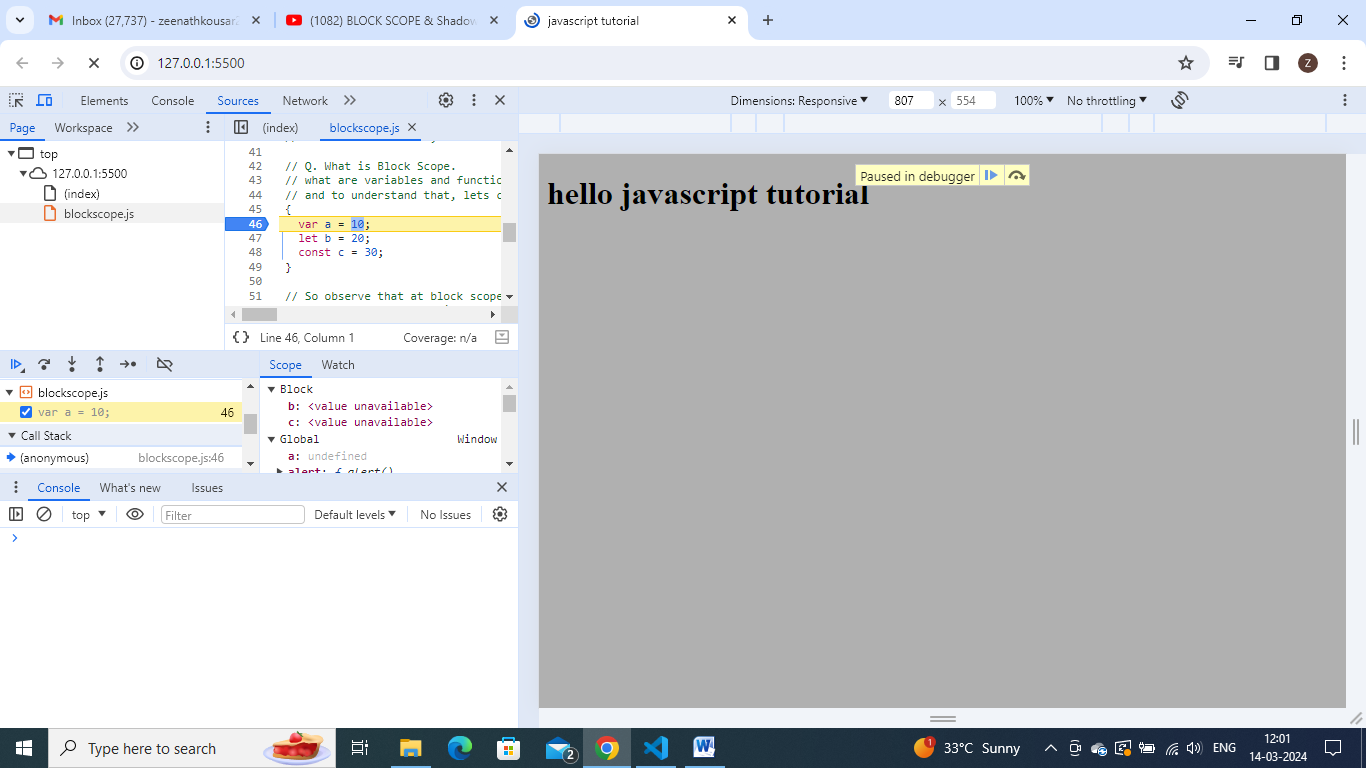
  let b = 20;

  const c = 30;

}

Now execute and keep debugger at first line.





So observe that at block scope , we have b and c as undefined and a is in global scope.

So let an d const variables are in block scope – they are getting hoisted in a separate space and assigned with undefined. But a is getting hoisted in global scope.

So we say that let and const are block scope- that means they get stored in separate memory space which is reserved for this block. Once js is finished executing this block , when it comes to line } ending curly braces , the variables let and const will not be accessible.

We cannot access let and const outside this block.that is so we say that let and const are in block scope .

But u can access var variable ‘a’ even outside , so it is known as global scope.

Lets do console.log and check.

{

  var a = 10;

  let b = 20;

  const c = 30;

  console.log(a);

  console.log(b);

  console.log(c);

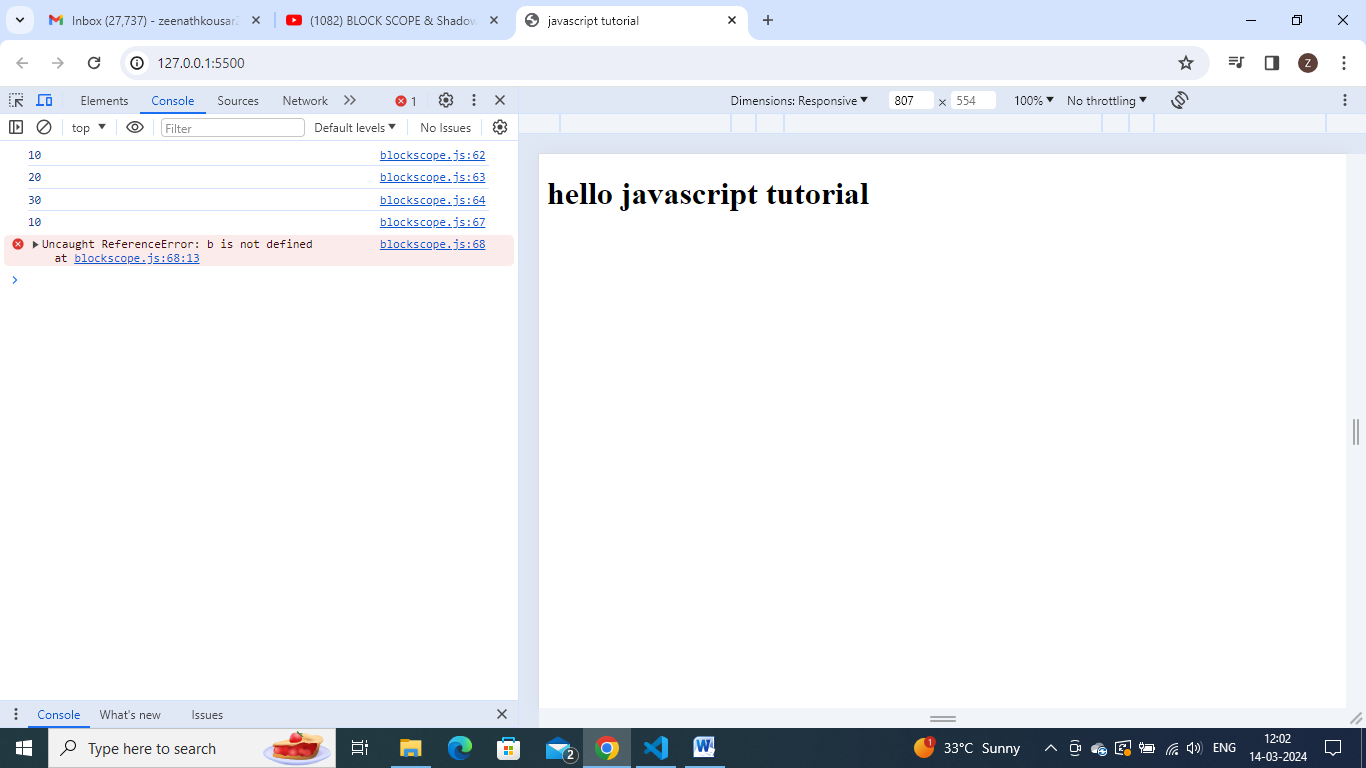
}

console.log(a);

console.log(b);

console.log(c);

now remove debugger and lets check, wt it prints in console.

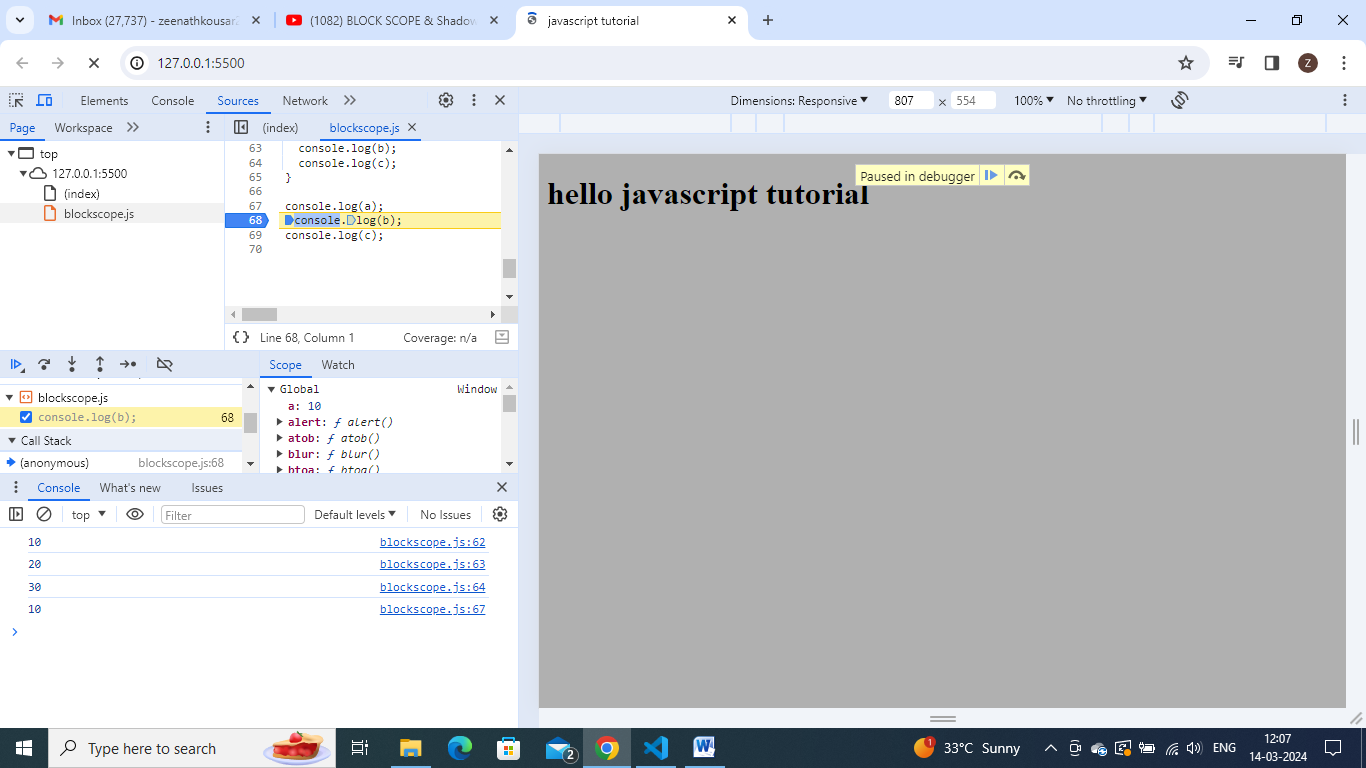


So it is brinting a,b,c values from inside block.

But outside block- it is only accessing and printing value ‘a’ . as it is a var variable.

‘b’ and ‘c’ are not accessible outside , as they are not in global scope.

Keep debugger at outside line console.log(b);



See now check the block scope memory is totally gone , as we came out of block scope.

Now we have moved to global scope- and here we have only ‘a’ with value 10.

So a is accessible here but

**What is shadowing? :**

var a = 100;

{

  var a = 10; // it will shadow the outer variable a=100;

  let b = 20;

  const c = 30;

  console.log(a); //here we get 10 , not 100

  console.log(b);

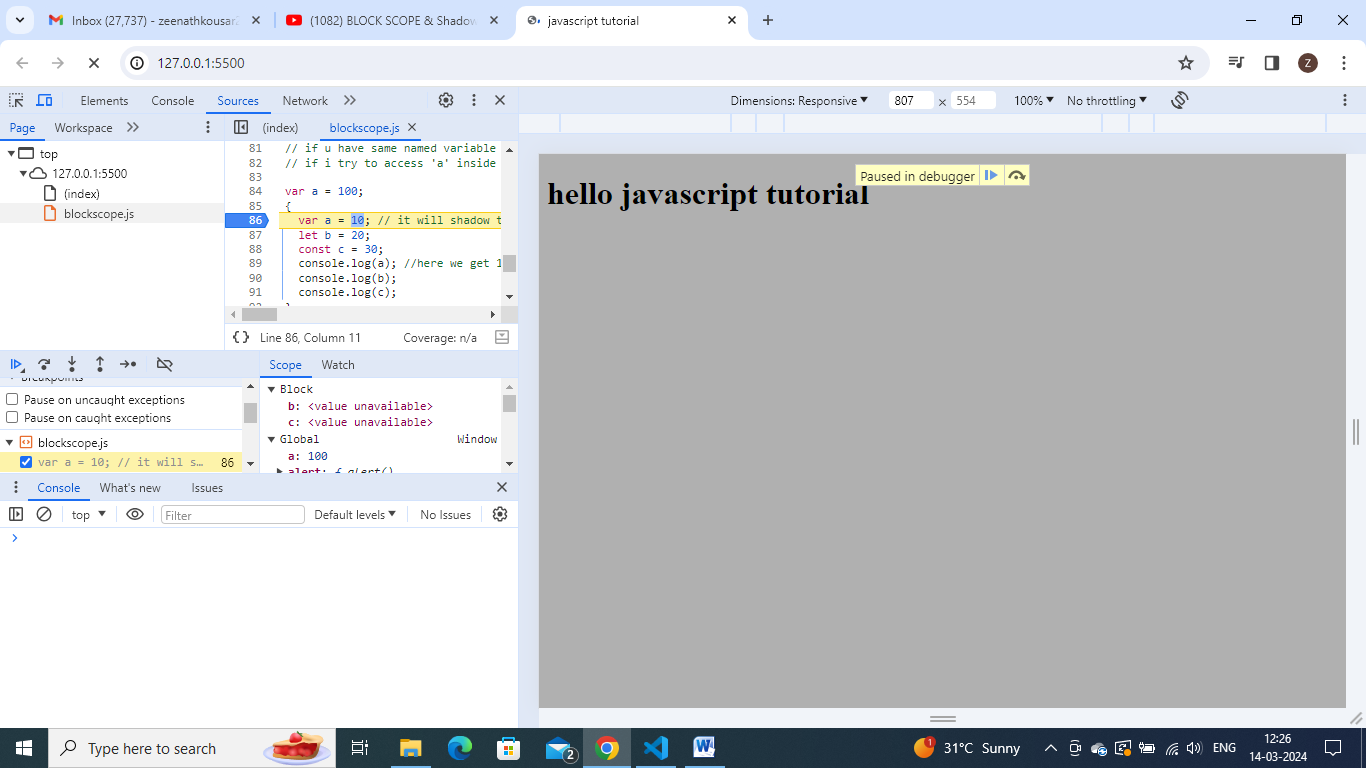
  console.log(c);

}

console.log(a); //we get 10 only here also. so inner variable 'a'=10 shadowed and changed the outer variable 'a' value also from 100 to 10.

// y because they both are pointing to same memory location.

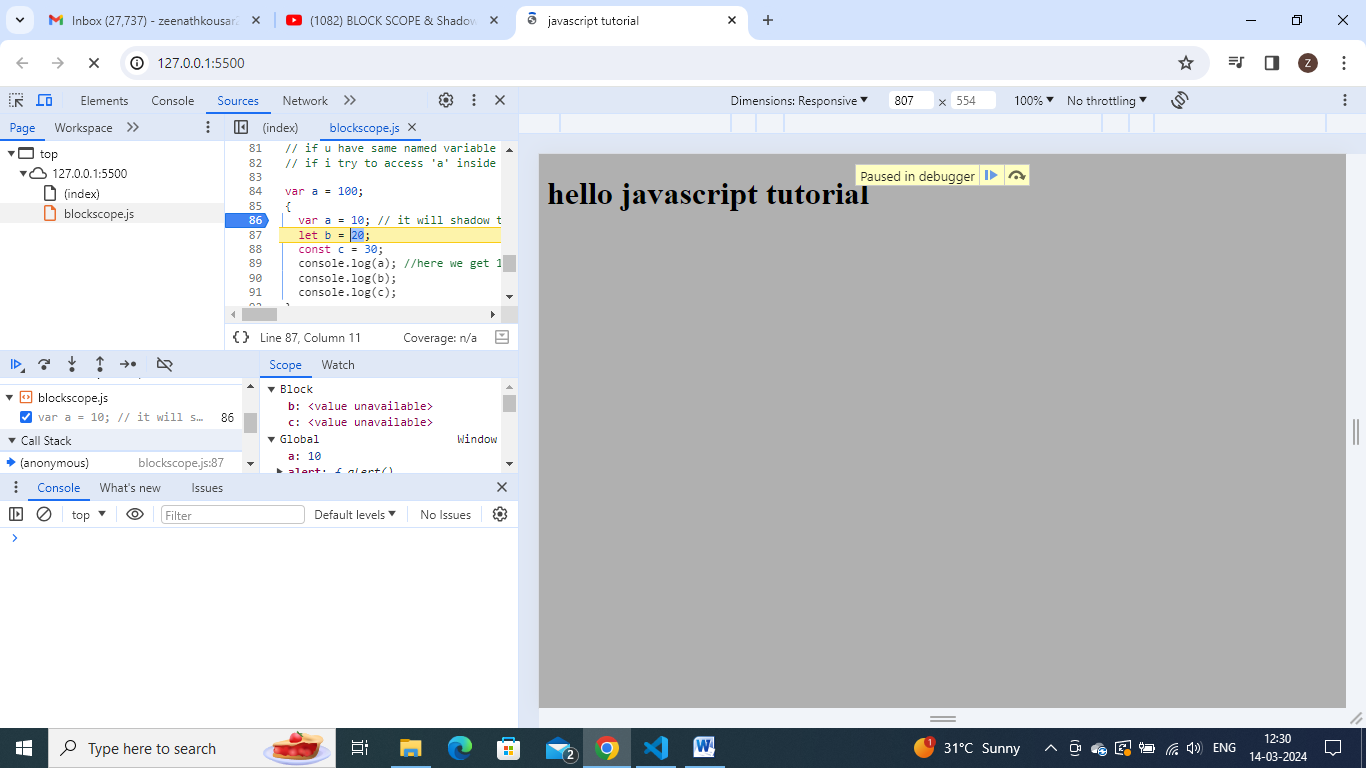
Lets place debugger at var a=10 and check.



See ‘a’ is in global scope.

So after executing first line var a=100, it assigned 100 to a .

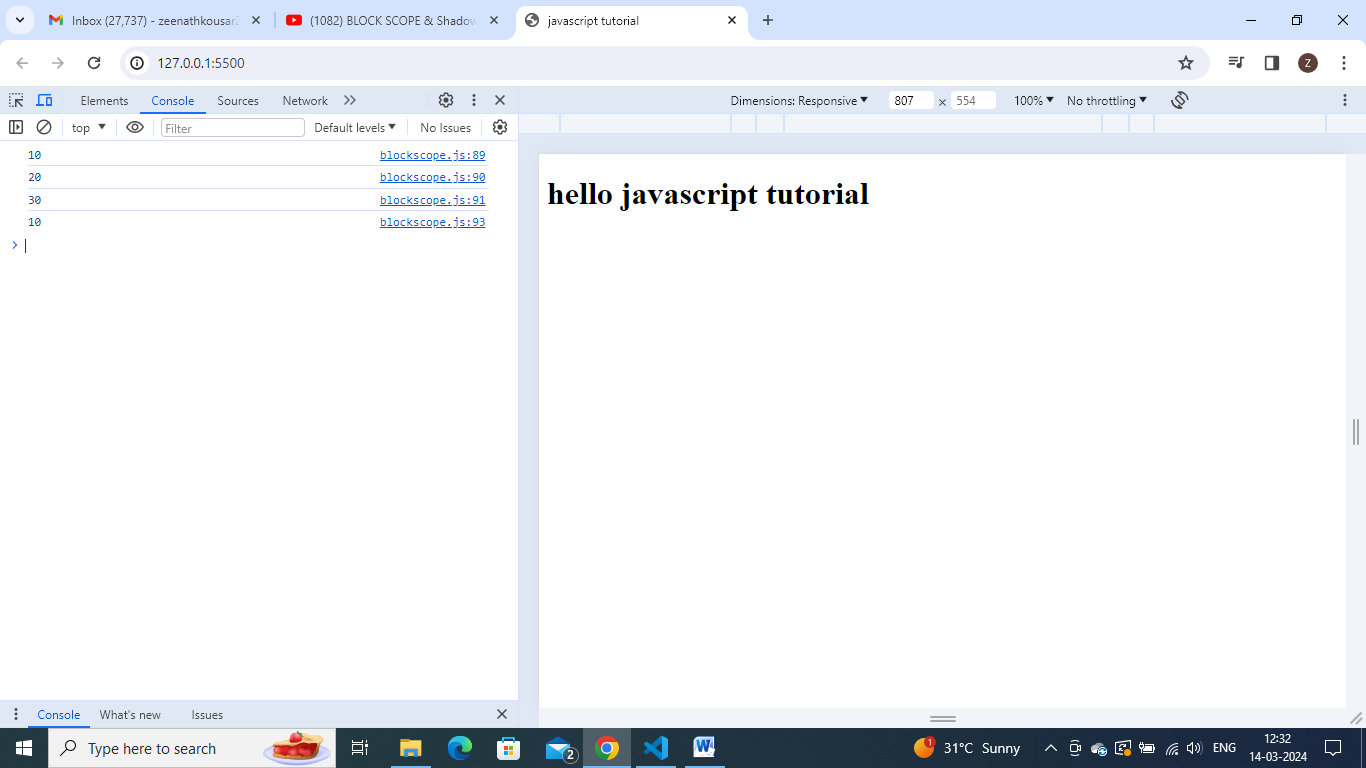
Now move debugger to next line.



See a is 10 now

So both inside line console.log(a) and outer line console.log(a) both are referring to same memory location ‘a’ that is global space.

So if I run the code we get ‘10’ only at both places



But that is not the case in let.

Note:

let a = 100;

{

  var a = 10;

  let b = 20;

  const c = 30;

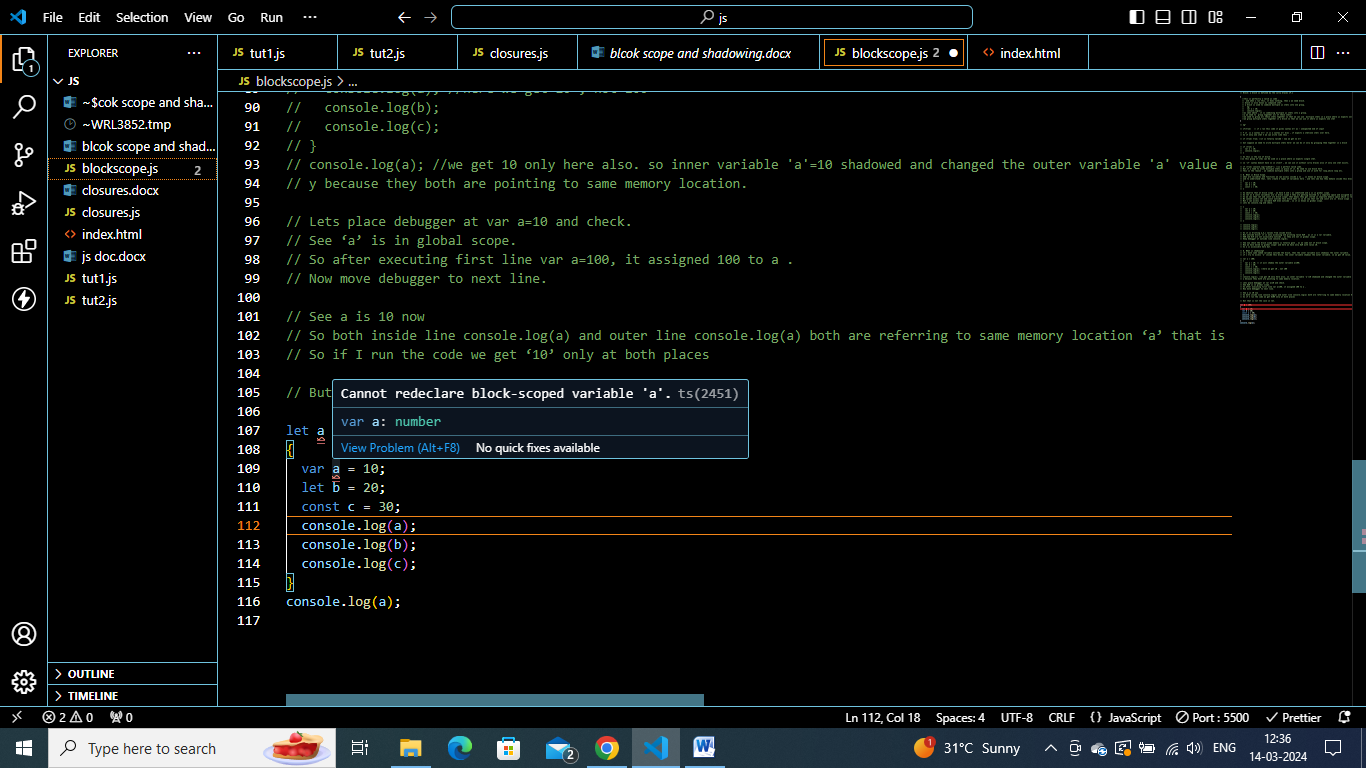
  console.log(a);

  console.log(b);

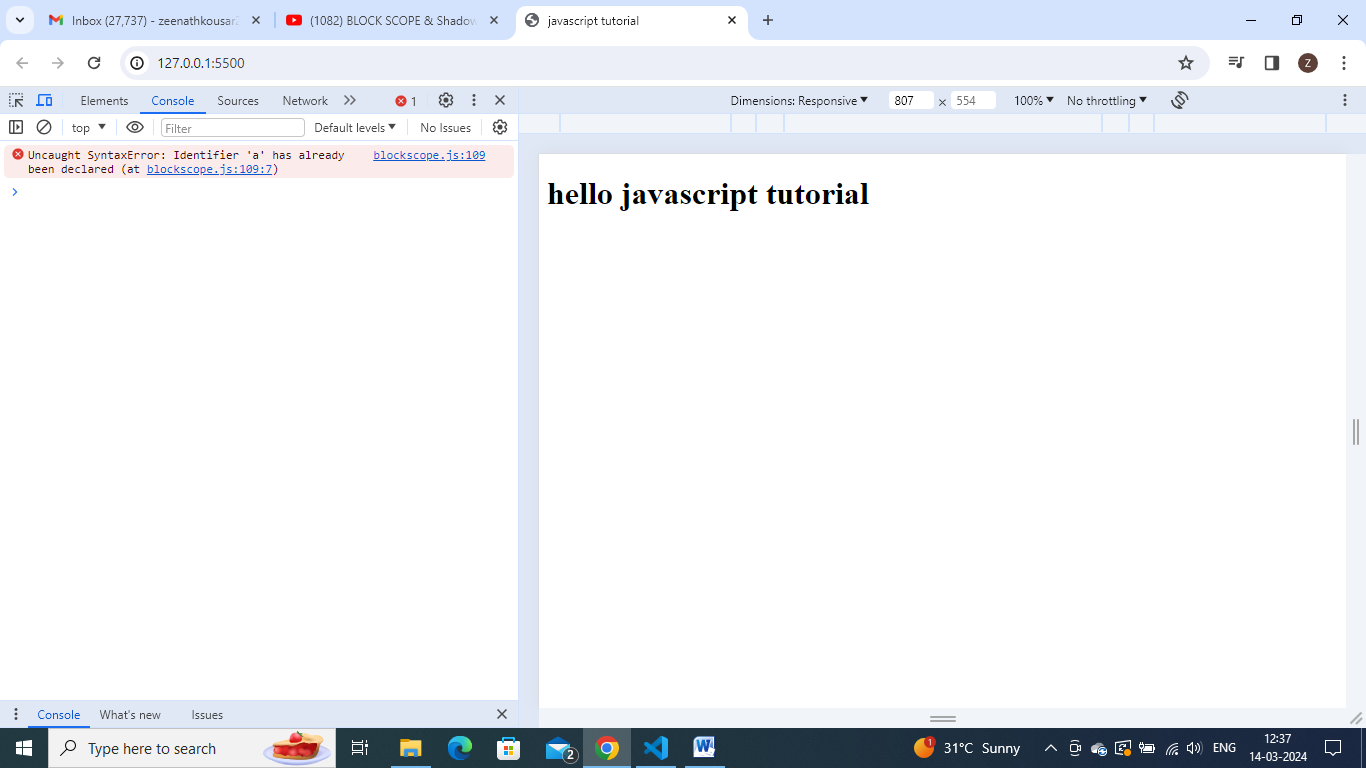
  console.log(c);

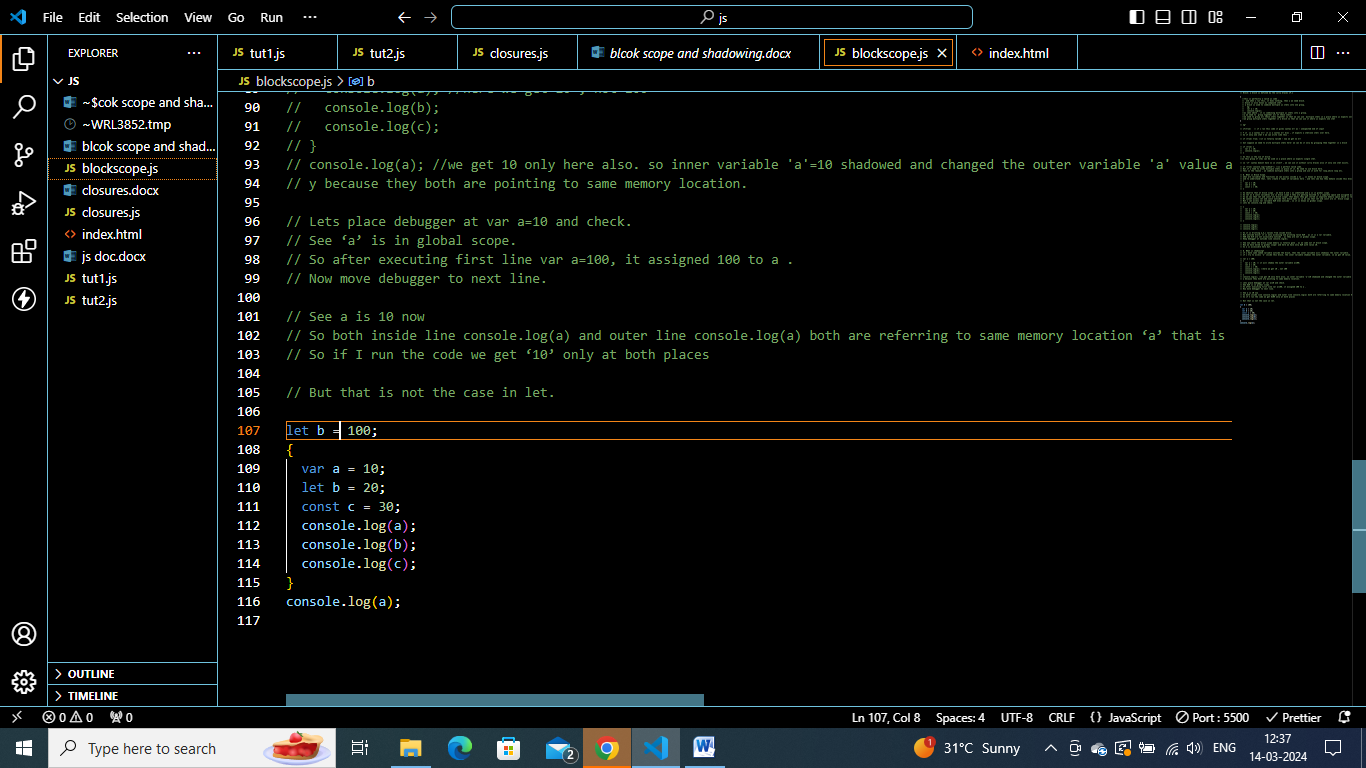
}

console.log(a);



As we redeclare let variable with var inside block – we got redeclaration err.





Redeclaring let variable ‘b’ by using let only inside block is ok.

**Shadowing incase of let:**

let b = 100;

{

  var a = 10;

  let b = 20;

  const c = 30;

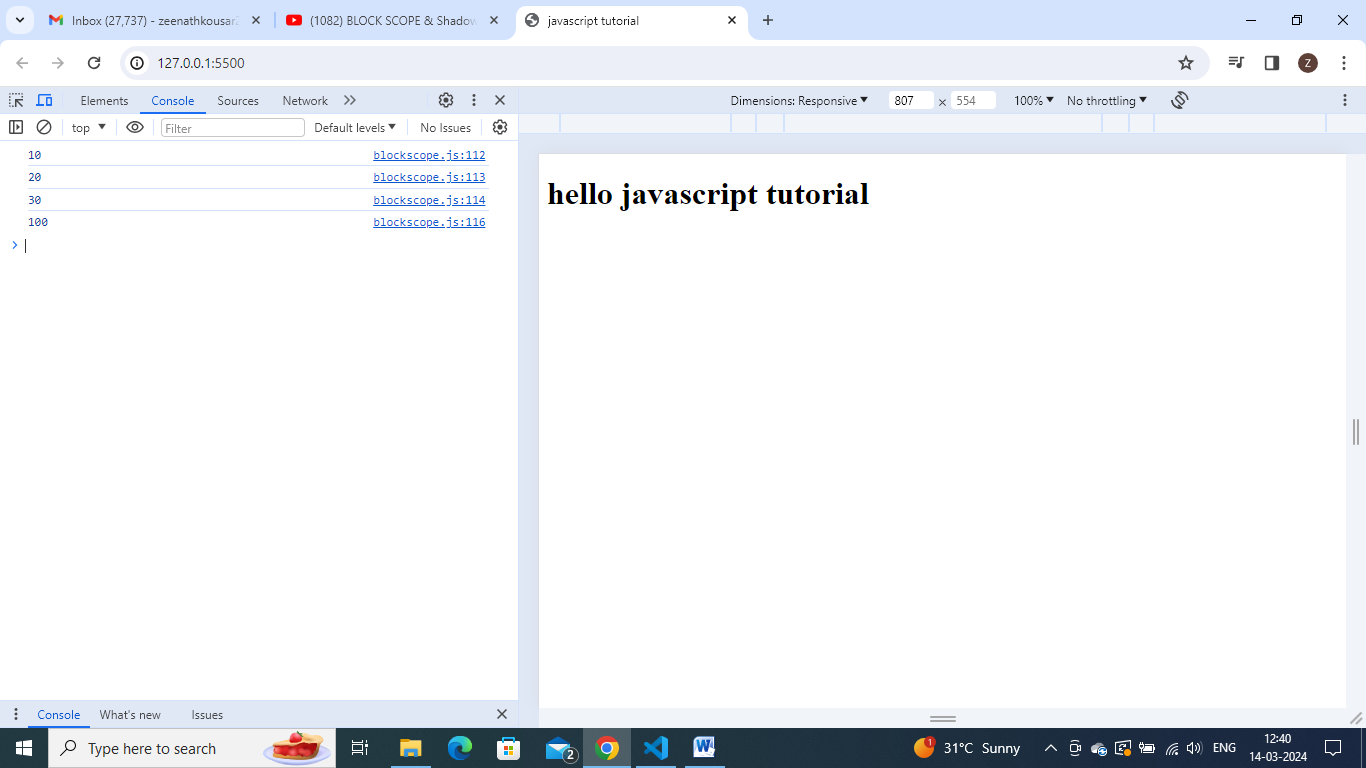
  console.log(a);

  console.log(b);//we get 20

  console.log(c);

}

console.log(b); //we get 100



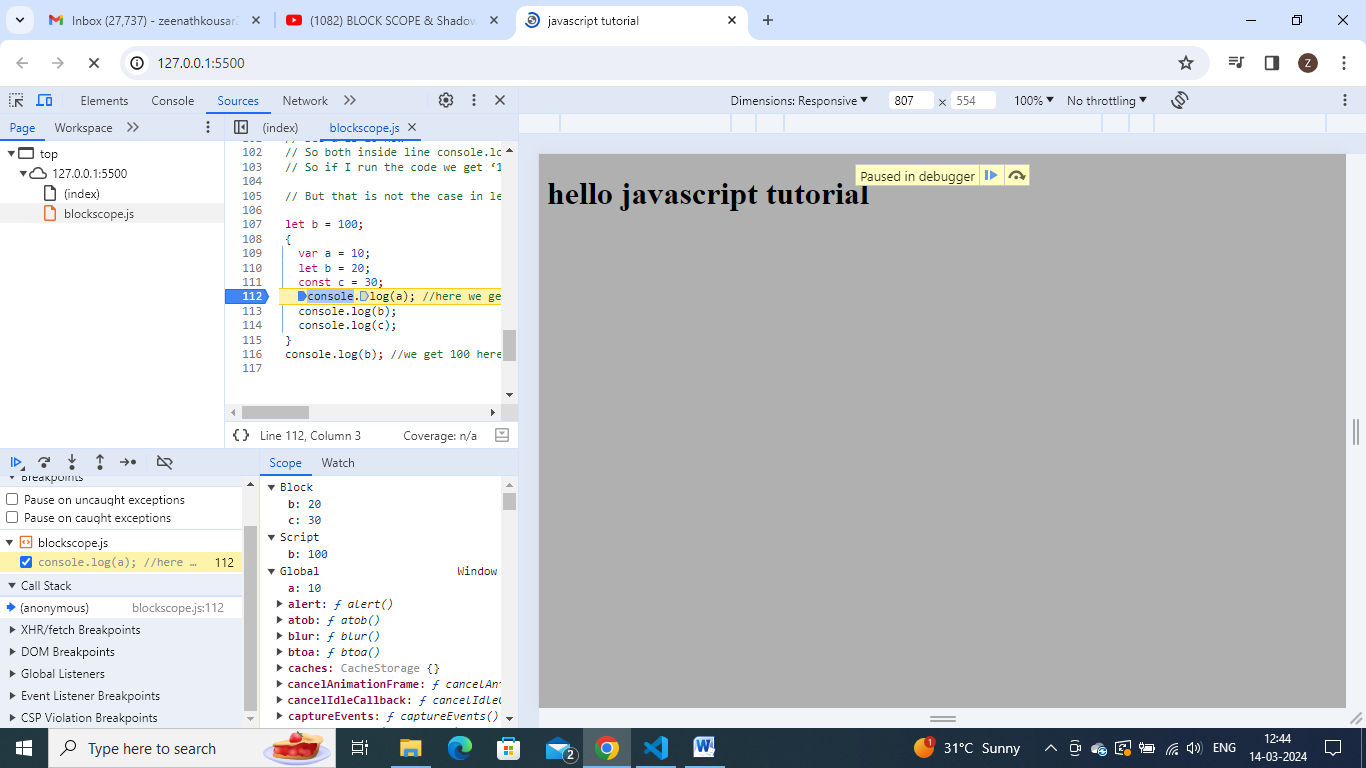
In the same way,inner b is shadowed outer one.

But when I print b outside – I got 100, not 20 .y?

Lets check

Inner b- has block scope and outer b has another scope.

Keep debugger at line console.log(a) inner one and check.



Carefully observe that inner b and c (let and const) are in block scope.

Var variable ‘a’ is in global scope .

Outer variable ‘b’(let ) is in script(some other memory space).

So here, I had 3 types of blocks- block,script,global.

Remember- let and const also gets hoisted but in separate memory space- scipt.

And we also have a scope and a separate obj in memory for this block itself and b and c resides over thre . now if I try to print ‘b’ inside this block, it refer to block scope

And if I print ‘b’ outside the block. It prints 100.

That is y it prints 20 and 100 separately.

And that is known as shadowing.

Similar thing happenes with const also.

const c = 100;

{

  var a = 10;

  let b = 20;

  const c = 30;

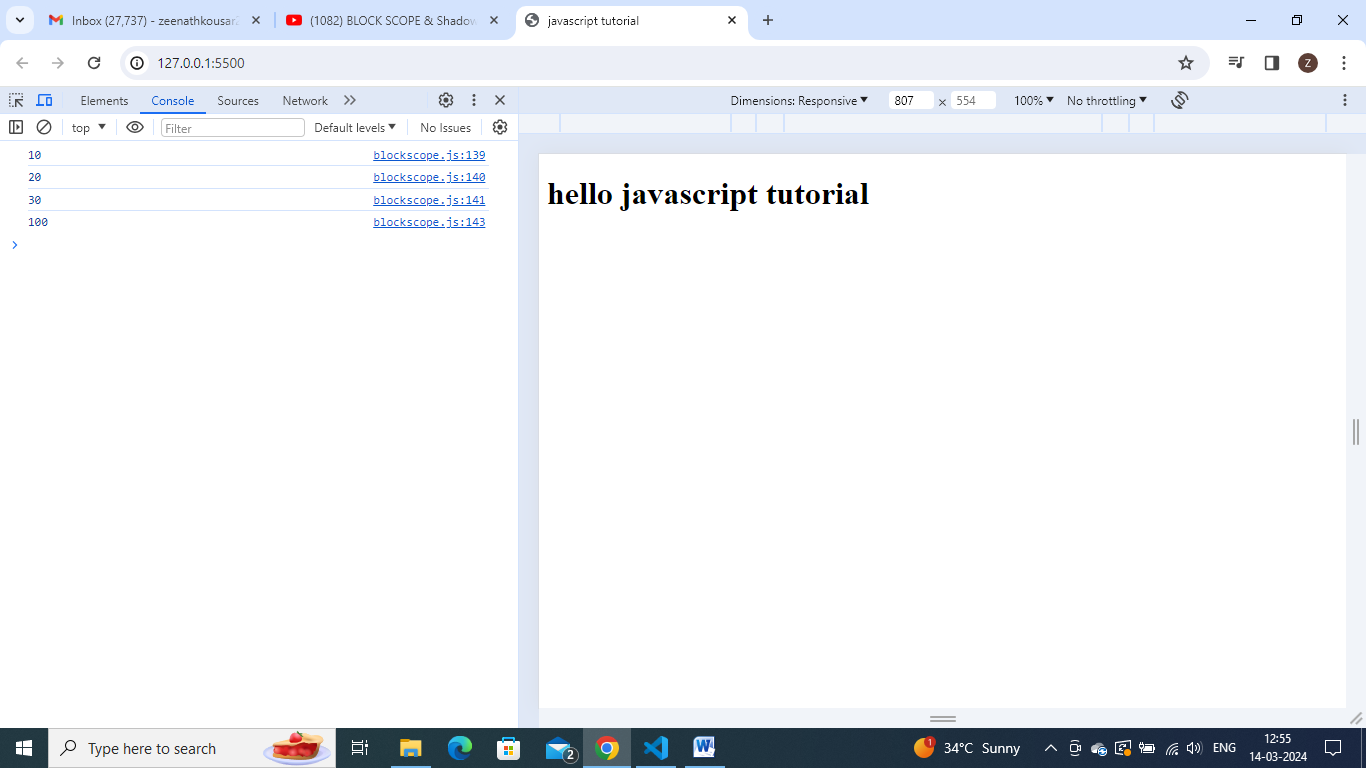
  console.log(a);

  console.log(b);

  console.log(c);//30

}

console.log(c);//100



Shadowing behaves similar way, even when instead of block it is inside a function.

const c = 100;

function hello() {

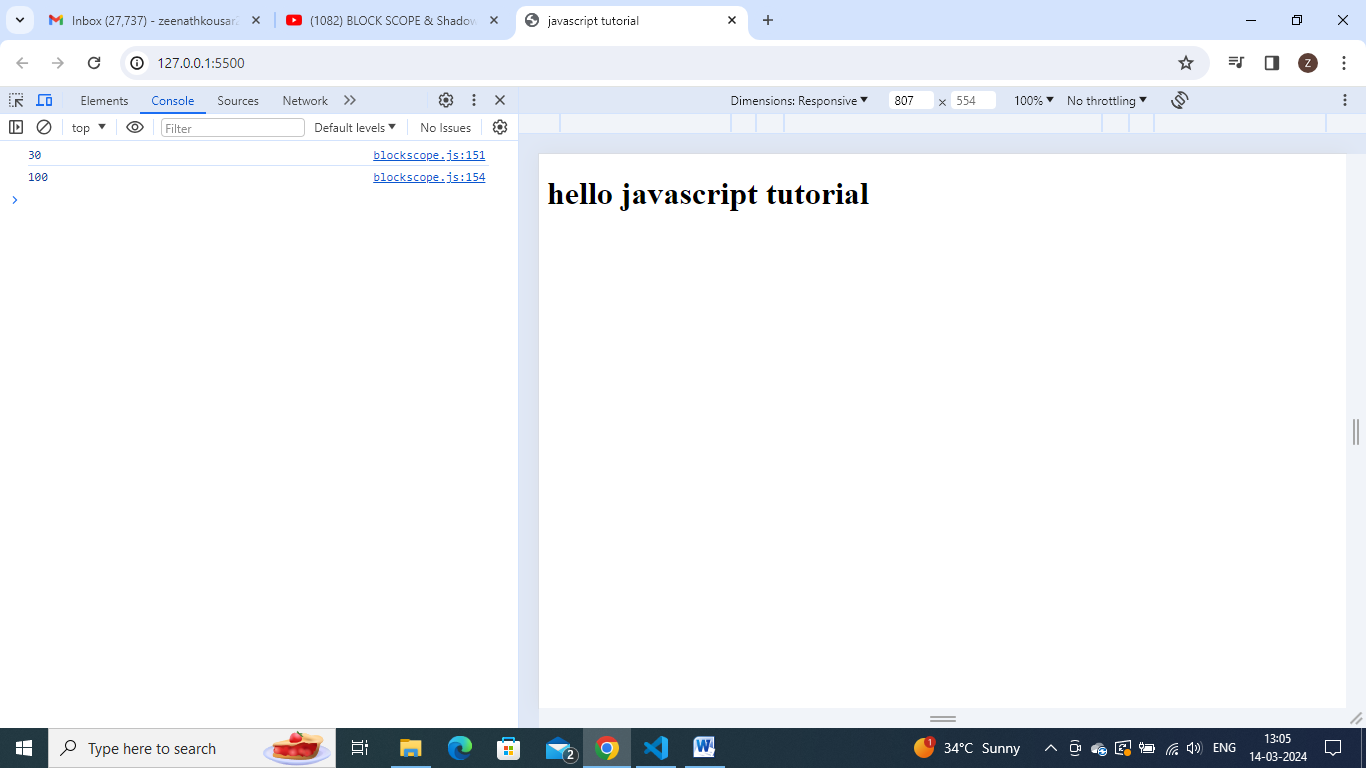
  const c = 30;

  console.log(c); //30

}

hello();

console.log(c); //100



let c = 100;

function hello() {

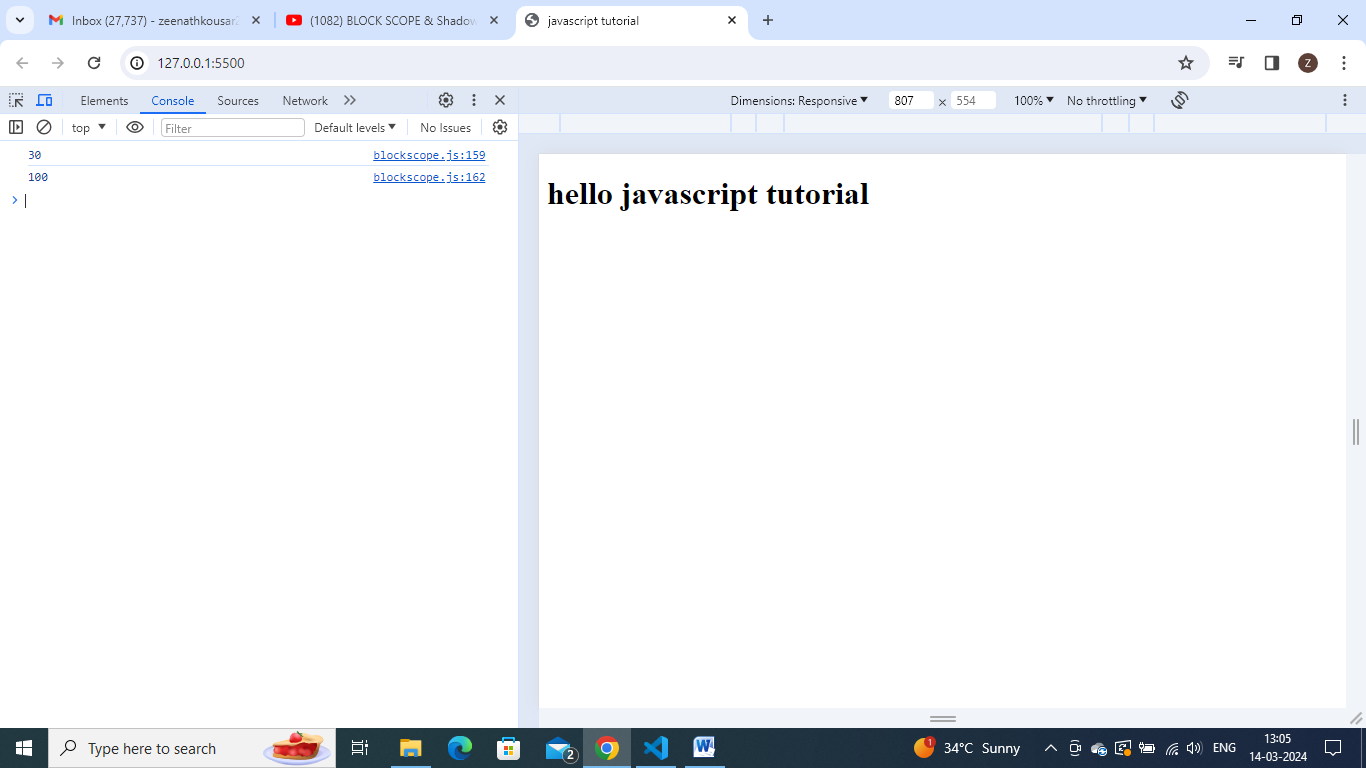
  let c = 30;

  console.log(c); //30

}

hello();

console.log(c); //100



var c = 100;

function hello() {

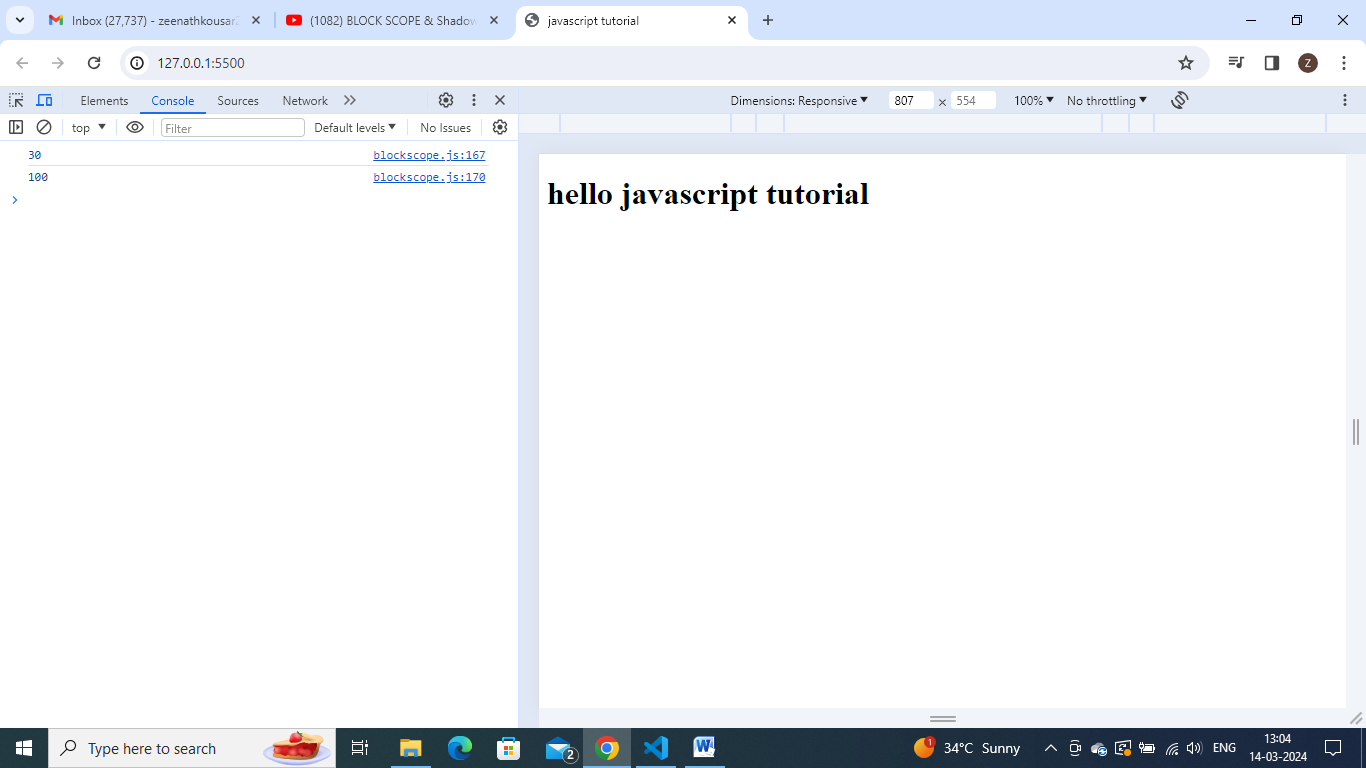
  var c = 30;

  console.log(c); //30

}

hello();

console.log(c); //100



observe:

var c = 100;

function hello() {

  var c = 30;

  console.log(c); //30

}

hello();

console.log(c); //100

var c = 100;

{

  var c = 30;

  console.log(c); //30

}

console.log(c); //30

**//illegal shadowing:**

//suppose if u want to shadow a var variable inside a block - then u can  shadow it easily.

var a=20;

{

    var a=10;//valid

}

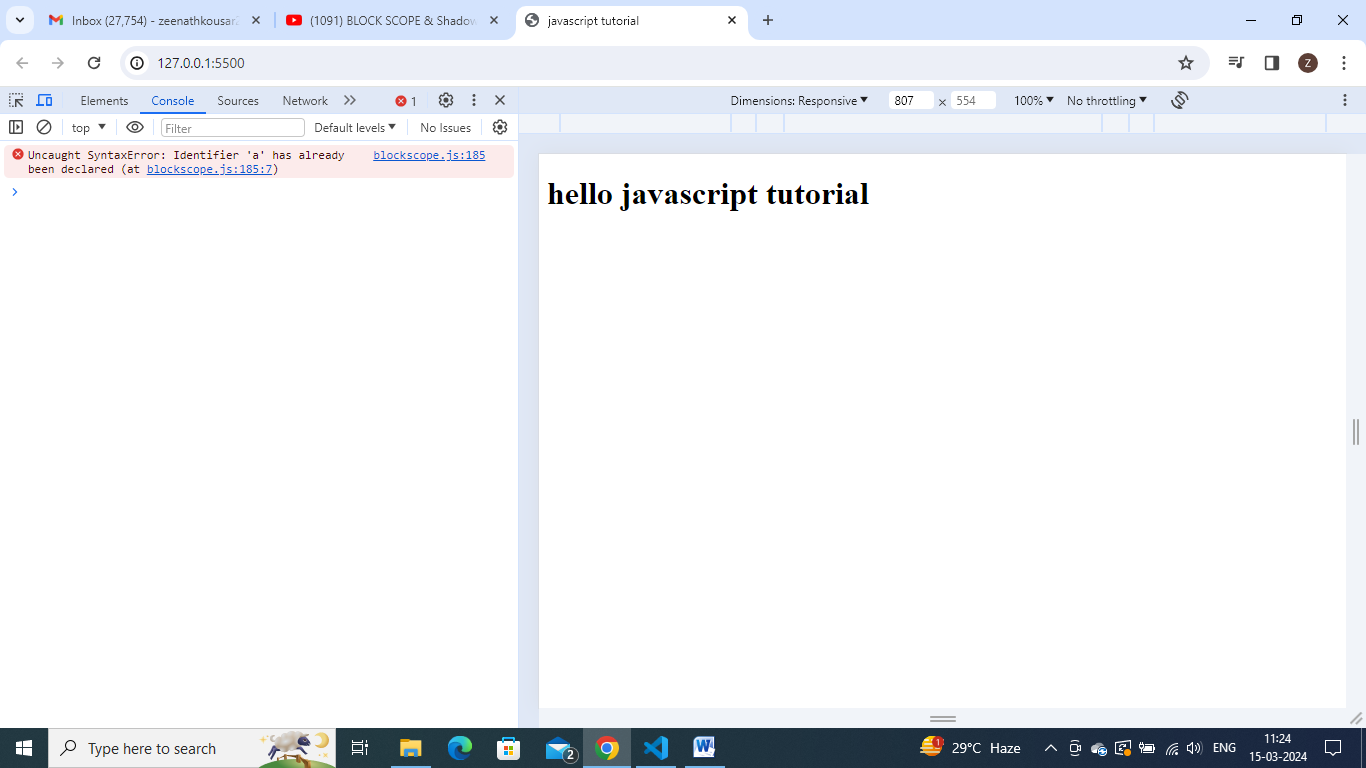
//but suppose trying to shadow let variable inside a block  - we cannot do this- we get err- 'a' is already been declared.

let a = 20;

{

  var a = 10; //illegal shadowing

}



// so shadowing a let variable using var inside a block - u cannot do that.

// but try to shadow it using let- perfectly valid - so u can shadow let using let- valid

let a=20;

{

    let  a=20; //valid – no err

}

//let try to shhadow a var using let - yes valild

var a = 20;

{

  let a = 10;

}

Why like this?

//lets take below illegal shadowing case and discuss.

let a = 20;

{

  var a = 10; //illegal shadowing

}

In this case,

If var a=10 is shadowing something then it should not cross the boundary of scope.in a particular scope, let cannot be redeclared right.so var should not cross its bowndary here.

Wt is boundary of var- var is a function scope. Lets put a function here.

let a=20;

function(x){

    var a=10; //now we wont get any err . it will be fine now, as var is not interfering  with outside ‘a’ now. So this is not a illegal shaowing.

}

// with const also same

const a = 20;

{

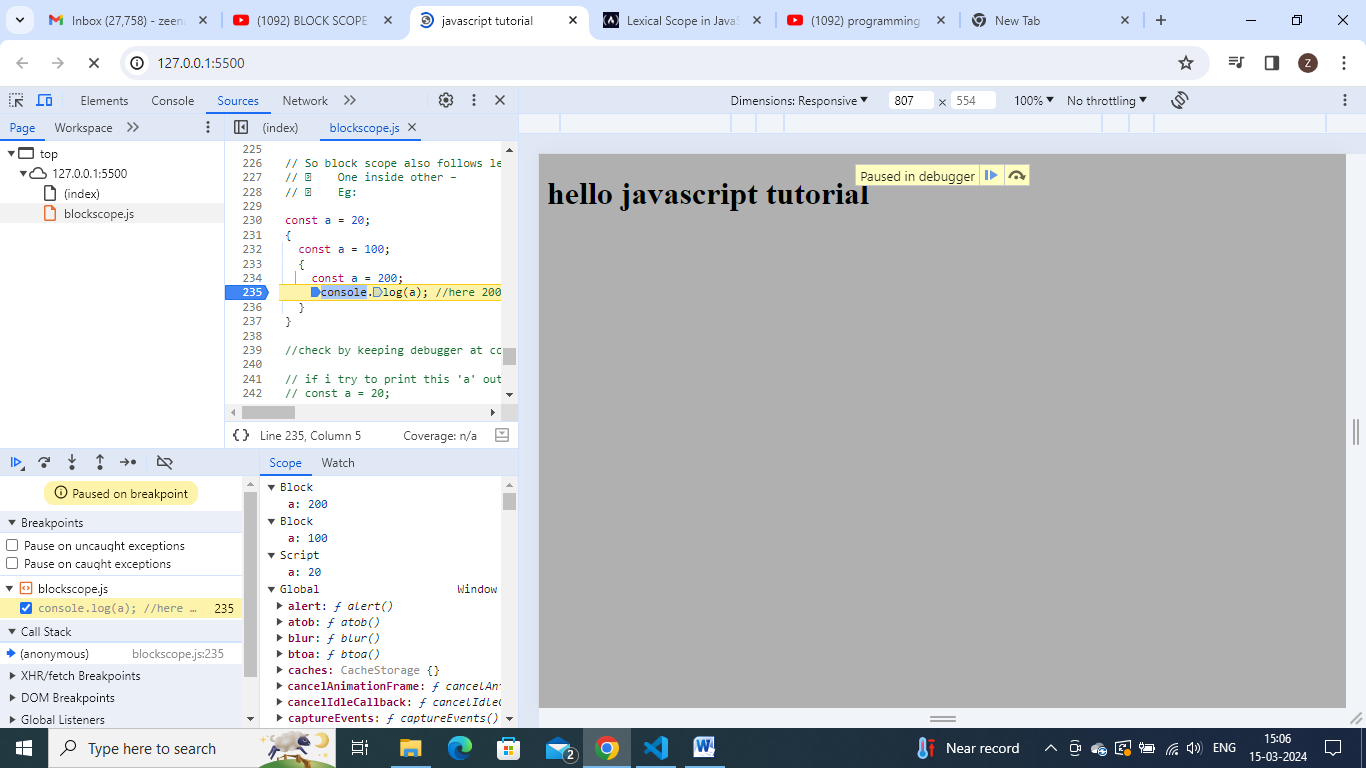
  const a = 10; //no err - fine

}

**Lexical scope:**

So block scope also follows lexical scope.so these scopes are lexically present.

* One inside other –
* Eg:
* const a = 20;
* {
* const a = 100;
* {
* const a = 200;
* console.log(a); //here 200 will get printed
* }
* }
* //check by keeping debugger at console.log(a);



So we can observe that each and every block has its own lexical scope.and again it follows lexical chain pattern i.e.; comment const a=200 , then observe. That if ‘a’ is not here, it will take 100 value

const a = 20;

{

  const a = 100;

  {

    //const a = 200;

    console.log(a); //here 100 will get printed

  }

}

Even if u declare ur function with arrow func also scope will be exactly same.

All scope rules are same for arrow funcs also