Function is heart of javascript.

Function

Date:

What is an Anonymous function:

A function without Name is a Anonymous function.

What are First Class Functions?

What is difference between.
Function Statement vs. Function Expression vs. Function Declaration?

major difference is hoisting.

a(); o/p- a called
b(); TypeError: b is not a Function

1) Function Statement & function Declaration (Both are same)

```
function a() {
    console.log("a called")
}
```

→ Here a is creating memory.

2) Function Expression

```
var b = function() {
    console.log("b called");
}
b()
```

→ Here we assign putting function as value b.
→ Here b is executed like any other variable treated

3) Anonymous Function

```
function () {

}
```

→ As we know a function without Name is call Anonymous function.
Anonymous does not have there own Identity & it looks like Function Statement (type)

we get error :- Function statements required a function name.

So Anonymous without Name where it is used?

Anonymous function are used in place where function used as values

if we read error we will get to know what type of function it is is Function Statement or Declaration Fun

to read blog article, book (author uses this names of function.

outer Outer Scope or Global scope means.

function a(){ this is outer scope / Global scope used.
}
var b = function

Named Function Expression

var b = function xyz(){ → here if we give function name as xyz = value as we know it is not executed outside scope or Global scope.
console.log("b called");
}

b();
xyz(); → if we call a function xyz() then it will through error
uncaught: Reference Error: xyz is not defined.

if we want to access then var b = function xyz(){ as inside this function
console.log(xyz)
}
o/p xyz(){
console.log(xyz);
}

But if we try to use outside it with through out error.
as Reference Error xyz is not defined

this is called named Function Expression.

if we not give name then it used as anonymous function
i.e. var b = function (){
console.log("b called")
}
as we learned previously,

Difference between Parameters & Arguments

var b = function (param1, param2){
console.log("b called");
}

b(1, 2);

Arguments

Parameter.
this is local variable inside in the function scope.
outside identifiers

In blog, books this word is used.
The values which we pass inside a function are known as argument.

function(param1, param2) → this labels, which gets those value are known as parameter & identifier.

First Class function

Ways How we can use function or play with the function

We can pass function in these types (3)

- By anonymous function Expression.

```
b(function() {
};

var b = function(param1) {
  console.log(param1);
}

o/p  f() {
     }
```

- By named function

```
b(xyz) {     -> Function not write it will work.
}

function xyz() {
}
```

If we pass function into these function we can return function

```
var b = function(param1) {      (2) function
  return function() {           (3) funct^n
  }
}

console.log(b());   (1) function anonymous

o/p  f() {
     }
```

Here to return named fun^n

```
var b = function(param1) {          (2)
  return function xyz() {           (3) return named function
  }
}

console.log(b())     (1)
```

The ability to use functions as values is called first class Function.

The ability of function to be used as values and can be passed this as argument to another function & can be returned from the functions, is this ability is known as First class Function.

~~It~~ is programming concept

First class Functions → Ability to be used like values
First class Citizens ↗

Hoisting in JavaScript.

Sources → mark debugger to let x = 7. this is function - getName() →

Call Stack.

(anonymous) → to find here - windows -

Global -

getName - already function is added here

[[Scopes]]

0: Script

x: 7

1: Global

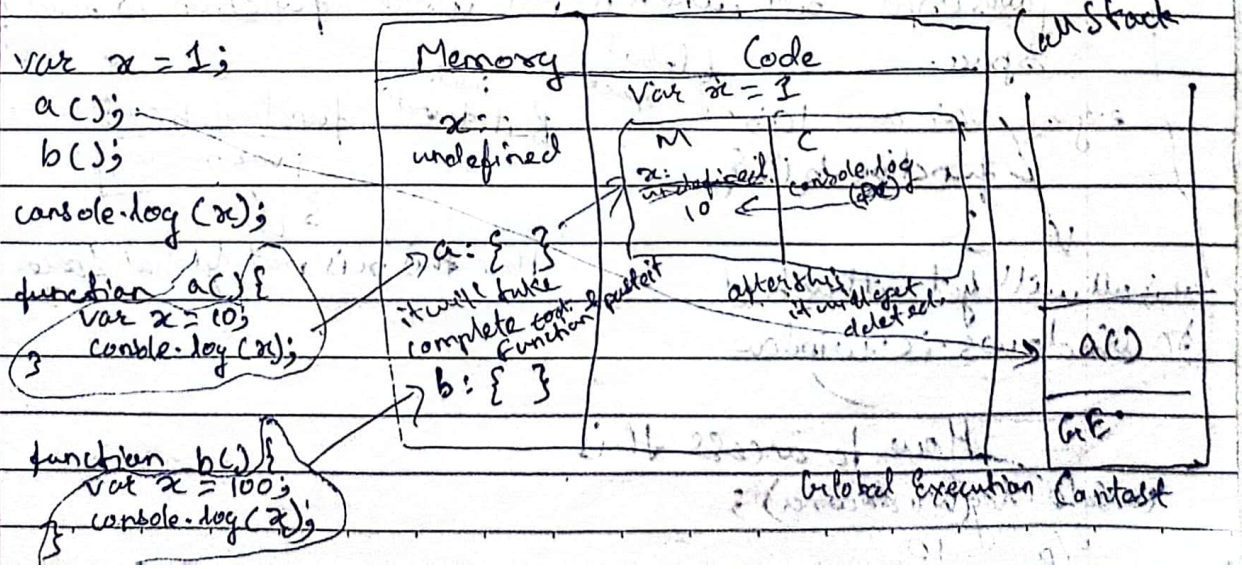
As we know

For function we can call function before we can initializing them (mentioning them) in hoisting.

i.e.

```
a();
b();
```
— ① just we called

```
function a(){
    var x = 10;
    console.log(x);
}
function b(){
    var x = 100;
    console.log(x);
}
```
② & then initialised.



Javascript -

Javascript not just run on browser

Javascript runs on servers : Javascript runs on lot of other devices also.

Wherever javascript running → There must be javascript engine there.

like eg- in chrome there is V8, Microedge has its own engine

So all this javascript engine has responsibility to create this global object in case of browser it is known as windows in case of node it is known something else.

Whereever you run javascript program is different

But there is always global object created.

even though file is empty javascript will create a this global object

window on console -

this === window in case of browser there is window.

true So we understood - different different browser has different different engine like, chrome

window { window: Window, self: Window.

- alert
- confirm
- fetch

} all this can be used with help of javascript.

javaScript engine → it itself creates global ie. alert, confirm

Global space → is nothing but any code you write in javascript but which is not inside a function is said global space.

e.g. - var a = 10; (global space)

function b() {

}

But if var is inside function b() {

var x = 10;

}

Here this x is not global space.

this all will get attached to windows i.e. window

How to access this

console.log(window.a);

o/p. 10

if we just give console.log(a) → it automatically assumes window.a & o/p - 10.

you are automatically assuming you are referring to global space

console.log(window.a)
console.log(a)
console.log(this.a)
} all are same. referring to same window/global space.

Auth :- user - pramodmakan@gmail.com
password :-

1) google - 2-step verification

At last - App passwords → click on App passwords - then select the app and device you want to generate the app password for -
select app select device

To select → Mail Windows Computer

password will be shown → copy it & paste it in password of auth.

As we know in javascript if we dont write any thing then also there are many thing going behind the screen. if we just type window in console & run it check it.

if var x = 10 given then just before running it we check in breakpoint already x is undefined. (debugger)

understand it deeply it has meaning

When it allocates memory to all the variables & functions → to the variable it keeps to put a place holder it is like a place holder which is placed in the memory that special key is undefined.

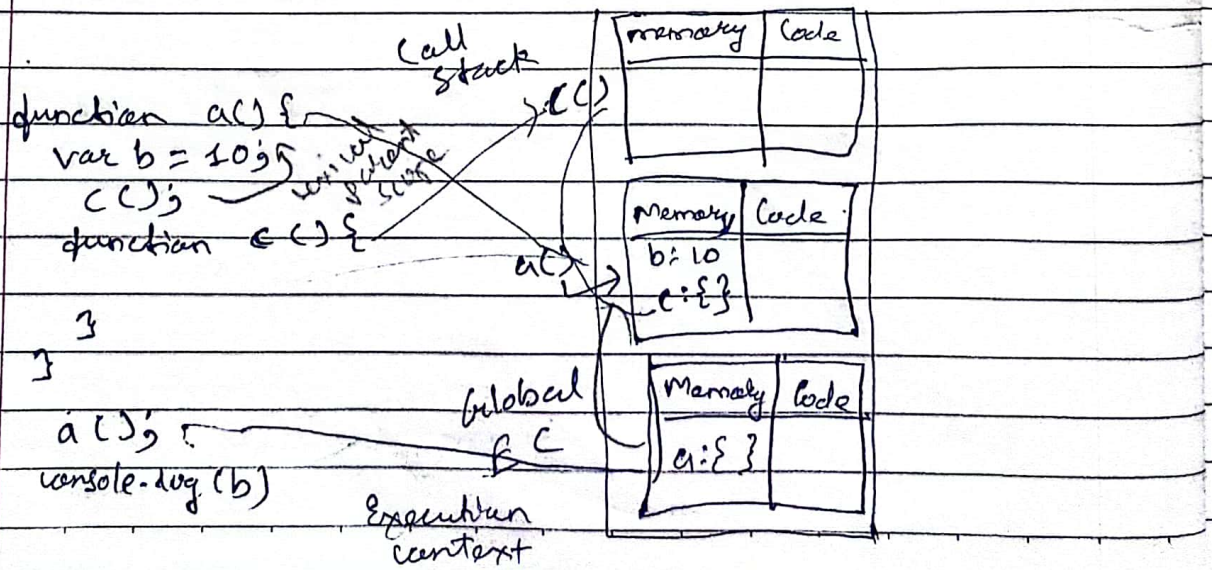
So if we run this in console - a undefined shows.

Javascript is loosely weakly typed language.

we can put it as per our wish.

```
var a;
console.log(a);
a=10;
console.log(a);
a="hello world";
console.log(a)
a=undefined → it is just a primitive.
console.log(a)
```

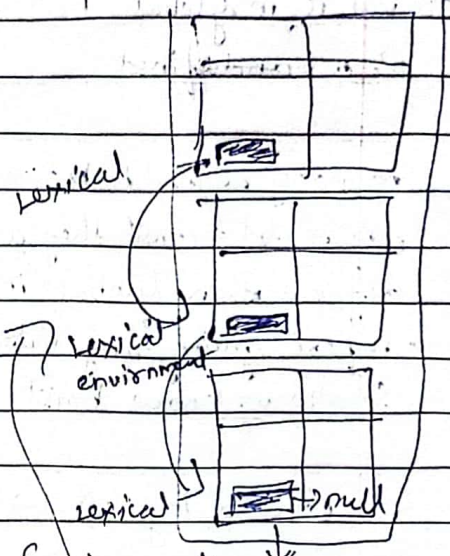
o/p
undefined.
10
hello world.
undefined



Lexical is hierarchy, in order, where the specific code is present physically inside the code.

here c is lexically inside a

& a is lexically inside Global EC



lexical environment chain

is guided Scope chain

Local memory + lexical environment in a()

Local memory + ~~parent~~ lexically environment of parent Environment

then program will see b is undefined.

if we check in

Sources → Call Stack.

```
function a(){
  var b = 10;
  c();
  function c(){
    console.log(b);
  }
}
a();
```

on Call Stack -

c

a

(anonymous)

It works like

Scope -

Local

Closure (a)

b: 10

Global.

Lexical environment chain → Scope chain.

As we learnt in Javascript → before declaring anything there also many things are running behind screen that check it type in windows.

if let a = 10, var b = 100 then before call it Javascript gives memory to it as a: undefined, b: undefined.

a & b are in Scope as undefined.

Error type.

```
let a = 10;
console.log(a);
var b = 100
```

Here var is in global But for let & const it is be shown in Script as a is undefined

memory assigned to b = 100 in variable was attached to globally object & in case of let & const they allocated memory but they are stored different than global it is in seperate memory space.
So before giving any value to them we cannot use them
So also we know hoist

1) console.log(a) → So here we call a & as it is let & const as we know it is block scope. & before giving value we call it ⇒ it is said hoist
2)
3) let a = 10;
4)
5) var b = 100

then it is in temperal dead zone.
it will show error as ① uncaught ReferenceError:

cannot access 'a' before initialization & it is in temperal dead zone
it means we cannot access them
they can only be accessed once some value initialized (given) to it
Before line 3 we call.

② Uncaught ReferenceError: x is not defined
i.e. means - console.log(x); Here x is not defined

② if in case of console.log(b) → then in b already memory is allocated
var b = 100 System will show "undefined"

in let a; first initiated
a = 10 then value given
this allowed here
But in const not allowed.
const b; } this will not allow. this one ⟹
b = 10 } error will come of

```
let a = 100;
console.log(a);
var b = 100;
```

console.log(window.b) => will it work -> Yes

as we know var is global & window is work global & ~~javascript~~.

But for let a = 10;

console.log(a) / window.a -> will it work -> No

o/p as undefined

~~as we know~~ System (window) will treat it as any other variables as a which was not present with us & give o/p as undefined.

Same for this.a / window.a } undefined. ~~But~~ let & const are reserved blockscope so it will not work on global. window object.

this.b / window.b / b } as we know var b so global & window help it global.

As we know got to know let & const is little strick as it block scope & re-declering is also not done

let a = 10; } uncaught SyntaxError: Identifier 'a' has
let a = 100; } already been declared.

as it duplicate of let. It is Syntax Error

Here system/javascript will not work any further or it not look to any other

if let a = 10; } Some error (3) Uncaught SyntaxError: Identifier 'a' has already been declared.
var a = 100;

~~Know~~ Know lets check for var.

var a = 10; } Here no error will come. As we know
var a = 100; } global/window/this

const b; Here const is very strict, we cannot use if a error comes
b = 1000; (4) uncaught SyntaxError: Missing initializer in const declaration

Acually it expects identifier must be initialized at

const b = 1000; otherwise it will not allow.

(5)

const b = 1000 → Uncaught TypeError: Assignment to constant variable.

b = 10000

You are trying to any other value to constant variable.

this b is type of const / constant

bez const b = 1000

It should be initialized, as well as declared together & if it is declared it should initialized right there only & you cannot assign any value later on. So it is Type error.

as const type.

How to avoid Temporal Dead zone:

Always put declaration & initialization always on top.

So that as code gets start, it will first hit initialization part first & then you go to logic or then you do something with this logic, variable. otherwise you will run into error in javascript.

let & const are Block Scoped.

Block means { // compound statement. }

Block is also known as compound Statement.
it is nothing just a { } curly bracket
Block is used to combine multiple javascript statement into one group.

i.e.
```
{ // compound statement
  var a = 10;
  console.log(a);
}
```
→ it is combining javascript statement into group.

Why do we need to group all this statements.
→ We need this together so that we can use multiple statement in a place where javascript expects only one statement.

if (true) ... Here it expects a statement i.e. just one line statement needed.
if (true) true. but if we want to write multiple statement then by grouping them together so we use this block { }

i.e.
```
if (true) {
  // compound Statement
  var a = 10
  console.log(a);
}
```
javascript need one statement remember.

that is reason we use block statement and use it with for loop, while loop, all other places.
Now we know what is Block.

What Block scope

What all variables, function we can access this block.

for eg -
```
{ var a = 10;
  inside Block. { let b = 20;
                  const c = 30;
                }
}
```

o/p - Sources → Call Stack
Scope
▽ Block
b: undefined
c: undefined
▽ Global
a: undefined

Block is hoisted seperate memory space which are reserved memory block.
i.e. As we know let & const are block i.e. seperate reserved memory

As we know.

```
{ 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)
```

As we know let & const are block so there it is outside Block so it will send an error

uncaught Reffrence error: b is not defined.

& Only var a = 10 as it is global so it is accessible here.

- What is shadowing in javascript.

```
var a = 100;
{
  var a = 10;
  console.log(a);
}
console.log(a)
```

o/p: 10, 10

global var means global.

But in let / const it will not work.

```
let b = 100
{
  let b = 20;
  console.log(b)
}
console.log(b)
```

o/p: 20, 100

Scope - b = 20 (script b = 100)

3 type of

```
const c = 100;
{
  const c = 30;
  console.log(c)
}
console.log(c)
```

o/p: 30, 100

c = 30 is shadowing c = 100

We have 3 type of Scope -

1) Global - , 2) Script 3) Block.

1) Global - where var → window / this / a is declared.

2) Script → Seperate memory space of i.e. primitive data type, individually declared. that is this seperate ~~seperate~~ seperate memory space -

3) Block → { } i.e. seperate memory space.

eg.

▽Block
b : 20
c : 30
▽ Script
b : 100
▽ Global
a : 10

Here let b = 20 b is shadowing let b = 100;

```
let b = 100
{
  let b = 20        this is shadowing to above let b
  console.log(b)
}
console.log(b);
```

Same thing happens to const also like -

Shadowing is same done for function also i.e :

```
const c = 100;
function x(){
  const c = 30;
  console.log(c);
}
x();
console.log(c);
```

shadow to this

o/p - 30, 100

Stack overflow.

illegal shadowing -

```
let a = 20
{
var a = 20;
}
```

shadow block; shadow

No this not been done, error will come like -

uncaught Syntax Error : Identifier 'a' has already been declared.

var a = 20 is shadowing let a = 20 No, will not be in block.

But for

```
var a = 20
{
let a = 20
}
```

shadow this will be allowed.

var should not cross the boundary.

Souhat is boundary.

```
let a = 20
{
    let a = 20
    console.log(a)
}
```

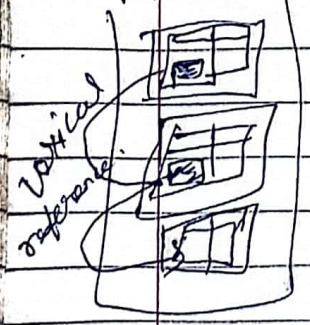
this is boundary

```
let a = 20
function xc() {
    var a = 20;
}
```

this is the boundaries.

Block scope also follows lexical scope.

As we know lexical scope -




```
const a = 20;
{
const a = 100;
{
const a = 200;
console.log(a)
}
}
```

Scope
- Block a = 200
- Block a = 100
- Script a = 20

One inside another.

it works same like lexical & Block Scope

d) For normal function & Arrow function Both have the same scope

Closures

```
function x(){
  var a=7;
  function y(){
    console.log(a);
  }
  y();
}
x();
```

As we know lexical. How it works -

when y execute it tries to find (a) inside local memory stored if it does not find it then it goes to lexical parent a=7

o/p -

source → call stack -

Scope →

Local

closure (x)

a=7

(buried)

o/p - 7.

This is what closure is

a closure is

means closure → A function binds together with its lexical environment.

A function along with its lexical scope forms a closure. is known as closure.

A closure is combination of function bundled together with references to its surrounding state (the lexical environment).

```
function x(){
  var a=7;
  function y(){
    console.log(a);
  }
  y();
}
x();
```

Function can be kept in variable

```
function x(){
  var a=function y(){
    console.log(a);
  }
  y();
}x();
```

Function can be used as we want.

```
function x(y){
  var a=7;
  y();
}
x(function y(){
  console.log(a);
});
```

We can use function as per our wise.

Similary we can return from functions also.

functions return function

function x(){ — Instead of calling it we can return it

var a=7;

function y(){

console.log(a);

}

return y;

}

var z=x(); — As we know it creates its own lexical env/P

console.log(z)

Here y means

function y(){

console.log(a);

}

return y

this y means

function y(){

console.log(a);

}

So when we invoke this x it returns y. & it returns y to the place where function was invoked.

function x(){ (2)

var a=7;

function y(){ (1)

console.log(a);

}

return y;

}

x();

So when x is called then in Scope closure (x)

a = 7

Here y function inside function x.

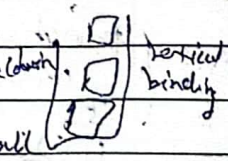
As we know javaScript is asynchronous.

Now here x is gone, no longer in call stack

it is completely gone, all variable gone

Right now what will happen for y:

As we also know lexical binding, parent, scope chain, all



But once function is returned outside:

return y

So now How will this behave in other lexical scope.

var z=x() → know z is having reference of x.

know we want outside the scope / outside of x.

there x is been once called & given its value to z i.e. var z=x(); then x is gone.

z(); → we call it

then function y it remembers from where it came i.e. function x.

```
function x(){
var a = 7;
function y(){
  console.log(a);
}
return y;
}
var z = x();
console.log(z);
//---
z();
```

not only code was returned, but the closure variable is returned.

when we return it, it means that closure is closed, function along with lexical scope was returned.

function & lexical scope that both was returned it was put in z.

So anywhere we execute in our program z() it still remembers the reference to a & it itself tries to find value of a i.e. 7 & it print it.

that is what closure is -

then z is called

What is Closure? anyone ask.

function & along with lexical scope bundle together forms a closure that is what closure is.

it is same if we give return in front of function i.e.,

```
function x(){
var a = 7;
return function y(){
  console.log(a);
}
}
var z = x();
console.log(z)
//---
z()
```

this is same as above.

it means we are just returning whole function.

this means reference to a memory location.

o/p is 100

```
function x(){
var a = 7
function y(){
  console.log(a);
}
a = 100;
return y;
}
var z = x()
console.log(z);
//---
z();
```

here a reference to memory location.

Here closure knows it will once value assigned & rest all will go but in fun y it will remember its parent lexical if it creates closure & a=100 is stored in closer as memory.

it is not garbage

Suppose one more level up in scope chain.

```
function z(){
  var b = 900;
  function x(){
    var a = 7;
    function y(){
      console.log(a, b)
    }
    y();
  }
  x();
}
z();
```

Now we will try to access its parent scope also & its parent-parent scope.
So Now lets check.

Here it retrieved things.

Source
o/p - scope.
Local
∇(closure (x))
a: 7
∇(closure (z))
b: 900

Here function remember things. So even when they are not in lexical scope. this is making more powerfull.

Uses of Closures:-

- Module Design Pattern
- Currying
- Functions like once
- memoize
- maintaining state in async world
- setTimeouts
- Iterators
- and many more...

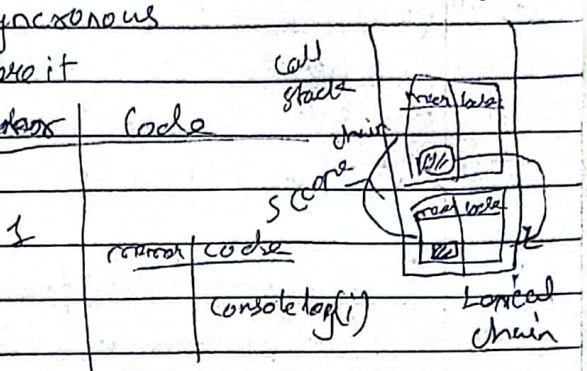
it's reference to 'i' & forms a closure

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

o/p: Namaste JavaScript
1
2
3
4
5

As we know JavaScript dont wait for anything it is like async asynchronous
Here it

Memory	Code
i = 1	



it take the reference of i

Here setTimeout it wait for 3 sec here.

So wherever this function goes it takes reference of (i).

What does setTimeOut do is setTimeout(function(){ console.log(i); }, 3000);
it takes the callback function & stores it at some place & attaches a timer to it of 3000 ms
Once timer expires it takes that function & puts it again to call stack & runs. it that is How setTimeout works.
We were given a problem to print 1, 2, 3, 4, 5 after each & every second
1 after 1 second, 2 after 2 seconds, 3 after 3 seconds.
Normally we will use for loop i.e. for(var i=0, i<5, i++)

o/p - Namaste JavaScript
6
6
6
6
6

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

reference
i value

it is working this way bec2 of closure. To understand closure.
A closure is like function & along with its lexical environment, even if it is taken out from original scope or it is executed in different scope still it remains its lexical environment. right.
When this setTimeout take this function & store it somewhere
So what function remember's to 'i', it is also told in closure video reference to i

i = "reference" & not value of i

it is just reference to "i" here all are pointing to i=value i.e. 6.
As we know javascript is asynchronous or does not wait for process to complete. it will go 1, 2, 3, 4, 5 & complete & the i=value will be 6. when timer expires, its too late bcz loop was running 1, 2, 3, 4, 5 continuously & complete to 6 & that time the settimeout completes & then function console(i) is searching i=value i.e. 6 then it calculate i*1000 6*1000 & displays.

6
6
6
6

Now How to fix it?

just use let as we know it is block scope. So every time loop run it creates new i value i.e.

i=1 { i=2 { i=3 { i=4 { i=5 {
(1 block scope.)
} } } } }

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

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

Look it in this view.

x() now

1	2	3	4	5

1 block scope
setTimeOut (function () {
console.log(1)
} 1*1000
it makes closure as closure (x)
1
	2 {
	setTimeout(function(){
	console.log(2)
	} 2*1000
	closure as closure (x)
	2
		3 {
		setTimeout(function(){
		console.log(3)
		} 3*1000
		closure (x)
		3
			4	5

As we know block create a new copy every time.

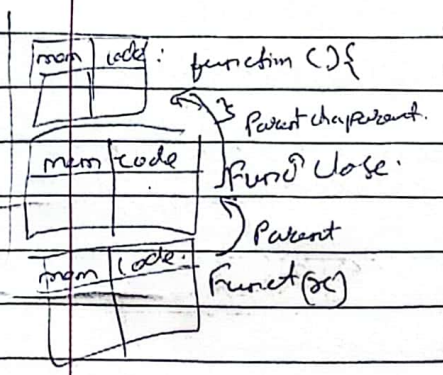
if you can't use let, you have to use var only then How it'll you do?.

→ Here only closure will help you Again.

Here we have to just make console.log(i) it refers to i value as we know, we have change it & we have to give new copy everytime i.e. 1, 2, 3, 4, 5.

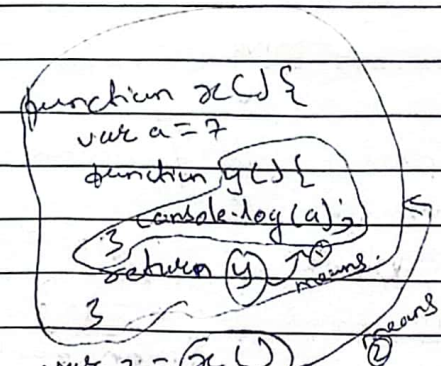
```
function x() {
for (var i=1; i<=5; i++){
  setTimeout(function(){
    console.log(i);
  }, i * 1000);
}
console.log("Namaste JavaScript")
}
x();
```

We have to do make closure.
we wrap this function into close() function & then
Here we have to supply the value of i
& we can just call the close function with (i) i.e. close(i)
we created new copy of set time funct. in function close(i)



e.g.

```
function x(){
  var a=7
  function y(){
    console.log(a);
  }
  return (y)
}
var z=(x())
console.log(z)
```



closure=7
function x()
closure 7.

same happened here function x() closure (1) i.e. close(i)

```
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();
```

```
function x()
  closure 1 { Block inside setTimeout function
}
functi- x()
  closure 2 { Block inside setTimeout function
}
function x()
  closure 3 {
}
4
5
```

o/p Namaste JavaScript
1
2
3
4
5

Parameter.

```
function outer(b) {          <- Parent lexical
    function inner() {       <- child lexical scope. parent lexical scope chain.
        console.log(a, b);
    }
    let a = 10;
    return inner;  calling inner function
}
```

closure outer: 10, Helloworld — system will know.

argument.

```
var close = outer("helloworld");
close();  <- calling function outer.
```

O/p 10 "helloworld"
Number String.

Here b is also part of outer function so it will be also executed as family member.

{ } means block.

let a = 10 means { a = 10 } — it power will be inside of block only not outside of block scope.
i.e. mean function outer(b){ let a = 10; function inner() } only inside this only. whatever be inside it it will consider.

outer function.
inner function

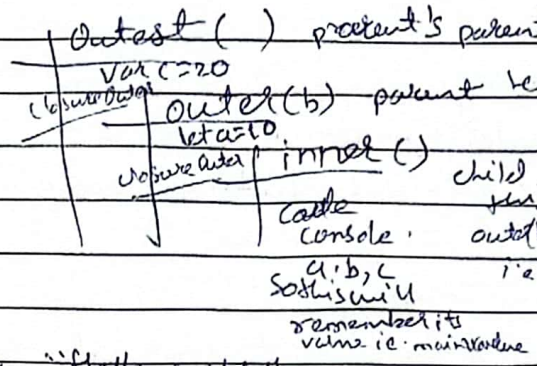
What if this all is put added one more function.

```
function outest() {
    var c = 20;
    function outer(b) {
        function inner() {
            console.log(a, b, c);
        }
        let a = 10;
        return inner;
    }
    return outer;
}
var close = (outer())("Helloworld");
close();
```

will go here.

this is this

o/p - 10 "Hello world" 20.



o/p - 10 "Hello world" 20

What if we give let a=100 outside the function outest() & what will happen.

```
function outest(){
    var c=20;
    function outer(b){
        function inner(){
        console.log(a,b,c);
        let a=10;
        return inner;
    }
    return outer;
}
let a=100;
var close = (outest())("helloworld");
close();
```

o/p 10 "helloworld" 20

Here let a=10 Both are completely different let a=100

(1) if we make //let a=10 commentout then it will go point a=100
How it works is here it will find in inner function, after that outer fun^ & after outest function & after finally global scope it will find.
more level deep into hierarchy.

(2) if comment out //let a=10 then if it don't get details then it will send an error as
uncaught ReferenceError: a is not defined.
it tries to check in scope chain, inner environment, parent environment, grand parent environment

Advantages of Closure.

- Closure is the beautiful part of javascript.

1) It is used in module pattern
2) It is used in function currying
3) Function Used in higher order function like memoize
4) One more important it helps in data hiding & encapsulations.

Data hiding & encapsulation.

Other pieces of code cannot have access to that particular data knows data privacying i.e. other part of program cannot access it.

e.g.

```
var counter = 0;                    this any body can change it or use it.
function incrementCounter() {
  counter++;
}                                   Lets wrap it inside closure.
```

→

```
function counter() {                now till this function call this inside
  var counter = 0;                  value nobody can use it(count)
  function incrementCounter() {
    counter = counter + 1;
    console.log(count);
  }
  console.log(count);
var counter 1 = counter()
counter1();                         calls here.
counter1();                         So how many times this counter1 is called it is
```

o/p. 1, 2 : incremented +1.

What if we call this function again in different place then?

```
function counter(){
    var counter = 0;
    return function increment(counter(){
        count++;
    }console.log(count);
}

var counter1 = counter()  - this is individual
counter1();                  1
counter1();                  2

var counter2 = counter();  - this is individual.
counter2();                  calls it & increase by 1
counter2();                  1
counter2();                  2
counter2()                   3
```

if it good in scalable. → When can make it constructor function.
& make separate function for increment & decrement.

```
function (counter() {        → named function.
    var count = 0;
    this.incrementCounter = function() {   ← anonymous function
        count.++;
        console.log(count);
    }

    this.decrementCounter = function (){
        count--;
        console.log(count);
    }
}

var counter1 = new (counter();

counter1.incrementCounter();     o/p -
counter1.increment(counter();    1
counter1.idecrement(counter();   2
                                 1
```

Disadvantages of Closure.

1) There could be over consumption of memory in closure because every time closure is formed. it consumes lot of memory.

2) Those closed over variables are not garbage collected. That means, it accumulating a lot of memory if we create lot of closures in our program. it expire.

Bcz those variable are not garbage collected till the program expires. if not handled properly then it can also lead to memory leaks. as we know memory accumulated over time every time. it can also freeze the browser if not handled properly.

As we know javascript engine.

There is grad garbage collector in javascript.

it takes out the memory or freeze out if it finds not needed or not in use

How are garbage & closures related to each other -

→ They are related.

```
function a(){
    var x=0;
    return function b(){
        console.log(x);
    }
}
var y=a();
//...
y();
```

As we know here it y() is called then y() is having closure of a & also having value x=0 it is been kept in memory or saved in memory. till the time function y() is called we cannot delete it or that garbage cannot delete it.

So like this we can create many function as many we wanted. Same will be happened x=0 memory it is saved so this it will make many space is consumed. not garbage collected.

V8 & chrome

What is smartly collects ~~garbage~~ garbage means? what smart?

```
function a(){
  var x=0; z=10;
  return function b(){
    console.log(x);
  }
}
var y = a();

y();
```

But now it system updated it will not show in system. for z = 10 direct garbage gone.

o/p - 0

console.log(x) -> 0

console.log(z) => Uncaught ReferenceError: z is not defined at eval (eval at b)

After this Read function.

After this call Back function.

Call Back Function.

Functions are first class citizen in javascript.
as we know First Class Function / First class citizen →
Ability to be used like values.
This means you can take a function & pass it into another function.
if we do that then the function which we passed in to other the function is known as call back functions.
This call Back functions are very powerful in javascript.

Function
function
function.

It gives access to the whole asynchronous world in a synchronous and single threaded language.

As we also know javascript is synchronous & single threaded language.
i.e. it can do one thing at a time & in specific order.
But due to call back we can do async thing inside javascript.

```
function x(y) {
}
x(function y() {
})
```

Here it is upto x function to call y - as we know everything is given by y to x. It can call whenever it want.

call back function to x

as we know this is one word. but returning big sentence { } means block

function y

function x
it can call any time to y when it want.

So this is know call back.

So lets see how it is used in asynchronous task in let take example of setTimeOut().
This is shortcut syntax of callback as arrow function.
setTimeOut take call back function i.e. (() => { })

```
setTimeout(function () {
    console.log("times");
}, 5000);
```

This means it will take call back function & store it in different place/seperate space & attach time of 5000 ms & store it

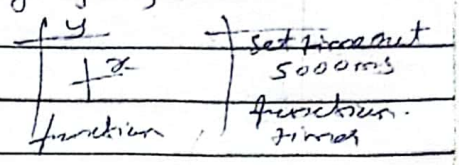
So we know javascript don't waste time to complete.

```
function x(y) {
    console.log("x");
    y() → to call here to y
}
x(function y() {
    console.log("y");
});
```

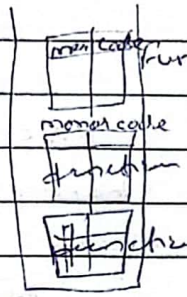
Parameter
argument

o/p
x
y
times ten 5000 ms.

that is way call back function gives us power of asynchronicity.
it does not wait till it finishes its time (5000ms)
it will process 100% but after seeing & javascript also don't wait.
check in sources > call stack & ⇒ console.log('y'); & run it



Everything is executed through call stack.
main thread.



So javaScript has just 1 one call stack. also call it as main thread.

So remember whatever is executed inside your page is execute only through your call stack (main thread) only.
for e.g. this, function x(y) x(function y), setTimeout(function() {})

So if any operation blocks this call stack i.e. that is known as blocking the main thread.

Suppose your function x is having heavy operation, suppose it take 20-30 secon to complete it arround complete it function. So by that time javascript as we know it's just 1 call stack thread. So it wont be able to execute any other code (function) i.e. many everything will be blocked on code. that is why it's called we never block it. So for this solution we have to use asynch operations for thing it take time. Same we done setTimeout (time) after 5 sec & this setTimeout will take it side and will execute after time it and till that it gets empty or free call stack.

So in Summary everything.
if javascript don't have this First class function & also we don't have this callback function, & we could not pass this function into another function we could not been able to do asynchronous operations - right. So using this web(api's - Set time out + callback function = we can achive this (asynchronous operation.)

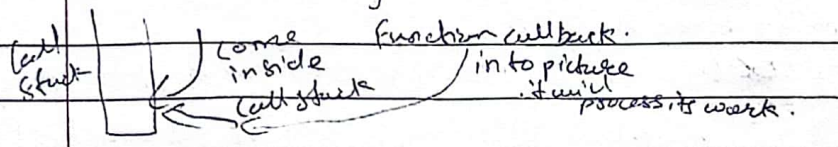
Click Me

Event.

```
<h1 id="heading">Namaste Javascript</h1>
<button id="clickMe">Click Me</button>
<script src="index.js"></script>
document.getElementById("clickMe").addEventListener("click",
function() {  callback function.
});
```

Incoming when javascript pick this line "clickMe" & it will add event listener i.e. event is clicked, so what will happens by calling event Listener it will call this call back function

So as we know about call back function, it will be stored somewhere & that will automatically comes into call stack.




```
document.getElementById("clickMe").addEventListener("click",
function xyz() { console.log("Button Clicked"); });
```

Sources → call Stack → Scope - & debugger on ▷ console.log Button Clicked

Console: Button Clicked

it will automatically come into call stack
call stack
⇒ xyz

Whenever you click a button, this call back function (xyz) will be pushed into call Stack & execute it i.e. Button clicked.

Lets see closure Demo along with closure Listeners

just added one more function.

about closure we know.

```
function -
  count = 0
  return function
    console.log(count)
```

Function xyz
count=0
return function
(console.log

lexical scope chain

closure function xyz
count 0

let count = 0 -> Now let means block scope ie. { } - a bracket

Before this -

```
let count = 0
document.getElementById("clickMe").addEventListener("click",
function xyz() { console.log("Button Clicked", ++count); });
```

o/p - Button Clicked 1
Button Clicked 2

Now to check how closure works,
So as we know to add one function.

```
function attachEventListeners() {
  let count = 0;
document.getElementById("clickMe").addEventListener("click",
  function xyz() {
    console.log("Button Clicked", ++count);
  });
}
attachEventListeners();
```

> know this callback function is forming closure to its outside scope

> we are calling this function.

What this will do? Javascript will do?
So this callback function will form a closure with count & it basically kind remember where this count is present.

o/p - Button clicked 1
Button clicked 2
Button Clicked 3

if we put debugger inside console.log line.
click on clickMe button.
Global call stack → xyz → Scope (closure (attachEventListeners))

closure (attachEventListeners)
count : 3.

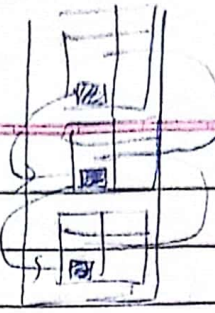
whenever we click on button it actually increases count : 3 in closure

Let's see inside dev tools → Elements → Event Listeners
Know click on Button ▽ click — what is the 'click doing here
button #clickMe.

Lexical ; if xyz() is callback function → this is known as
Scope → what is this → this is lexical scope chain

Elements -

Event Listeners



▽ click

▽ button #clickMe.

▽ handler : ϕ cxcy2()

▽ Scopes : Scopes(2)

1. Global : this is parent environment.

0. Closure : Parent environment.
Count 0

▷ Prototype

constructor.

Memory | Code

Count=0

Memory | Code

console.log(count)

Garbage Collection & removeEventListeners.

Why do we need to remove event Listeners?

Even in last event Listener. they remove event Listeners.

why do we do that.

Event Listeners are heavy i.e. means it takes memory.

As we know, we attach eventListeners it kind of form closure e.g. count=0.

function xcy2() { } count=0. even when the call stack is empty.

▽ callStack Empty. but still this program is not freeing this memory.

Not passed stored though it is empty but never no who will click page.

that is why event Listeners are heavy.

that is why we remove eventListeners when we don't use them.

Suppose if our page has thousands of buttons, thousands of event listeners attached on click, on mouse hover, on scroll, every all event listeners are placed. then our page can go a lot of slow because of this so many closures - closures like sitting into memory consuming a lot of memory of all these scope, all this callback functions will hold on those scope. So

So that is why a good practise is to free up / when we remove eventListeners. Suppose if we remove event Listeners then all this variables which are held by this closures will be garbage collected.

We also know How call stack works, Scope (lexical scope), (closures, lyGlobal, first class functions)