Hoisting

Hoisting is a phenomenon in JS by which you can access variable & functions even before you have initialized it or put any value to it

```
1. getName();
2. console.log(x);
3. console.log(getName);
4. var x= 7;
5. function getName(){
6.     console.log("Something");
7. }
```

Something 6
undefined 2
f getName(){ 3
console.log("Something")
3

- In the phase 1 of the memory creation JavaScript will allocate memory to each and every variable and function.
- It places a special placeholder Undifined to the variables.
- In case of function the whole code is put in the memory. Even before we are trying to run the code.

Var →

```
console.log(c)
var c= 9
```

→ Script runs →

```
var c
console.log(c)
c= 9
```

that's how it's ide interpited
This is the reason it's undefined.

undefined

Any variable declared with var undergoes hoisting:
During the execution of script all variable declaration with var moved up to the top of the script

while in case of let & const Hoisting doesn't occurs.

Const and let are not binded to the window

let → value can be reintialized
Const → value can't be reintialized } Difference

For Arrow Function

```
~~var~~ ~~getName~~()
  function getName2() {
  }

var getName = () => {
    Console.log("Something")
  }
```

If we write an arrow function it will be undefined because getName above behaves like another variable.
Same way we do → var getName2 = function() {
} this will also be treated as variable getName2.

Window

Window is a global object which is created along the global execution context.

So whenever any JavaScript program is run a global object is created, a global execution context is created and along with that global execution content a this variable is created.

Global object created in case of Browsers is window

at this Global level or at the base level in the Global Execution Context this === window

```
this === window
→ true
```

Anything that is not inside a function is a global space.

```
Var a=10
function b(){
    var x=10
}
```

a & b will be attached to the window but not x.

```
Console.log(window.a) → 10
console.log(a)        → 10
Console.log(this.a)   → 10
```

Undefined vs Notdefined

When it allocates memory to all the variables and functions, to the variables it tries to put a placeholder. It is like a place holder which is placed in the memory that special keyword is undefined.

```
console.log(a) → undefined
var a=7

console.log(x) → notdefined.
```

The Scope Chain, Scope & Lexical Enviroment

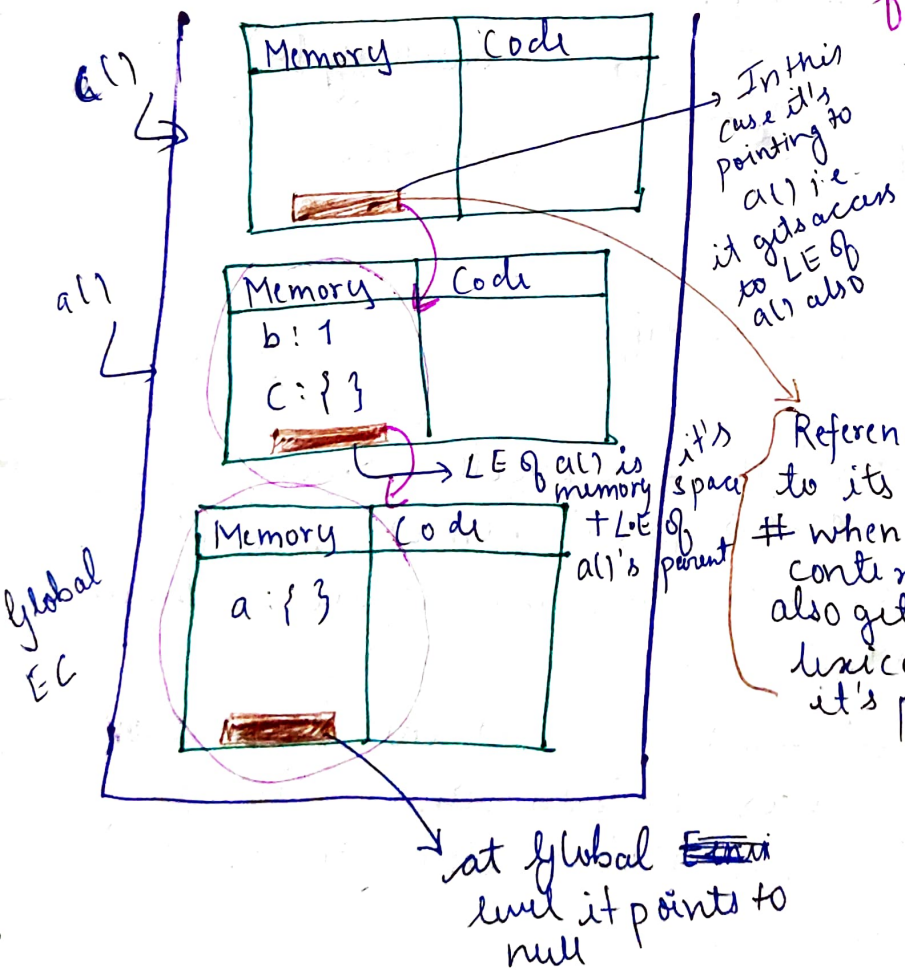
Scope → Scope is means where you can access a specific variable or function in a code

→ What is the scope of this variable — means where can the variable be accessed

→ Is variable inside scope of function - means can the variable be accessed inside the scope of the function.

Scope is directly dependent on the Lexical Environment

Lexical Environment




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

it's space

Reference to lexical environment to its parent

when ever the execution context is created you also get reference to the lexical environment to it's parient.

Scope Chain

Chain of all the lexical environment and the parent references is Scope Chain

Let & Const in JS Temporal Dead Zone

What is a Temporal Dead Zone?
Are let & const declarations hoisted?
Syntax Error Vs. ReferenceError Vs. TypeError?

→ let & const declarations are Hoisted. but they are declared very differently than var declarations.

→ these (let & Const) are in the temporal Dead Zone for time being.

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

```
console.log(a)
let a=10
var b=100
```
} Reference Error.

Global
- b: undefined

Script
- a: undefined

→ they are stored in a seperate memory space & you cannot access that memory space before you have put in some value in them so that is what is hoisting in let.

Temporal Dead Zone → is the time this let variable was hoisted and till it is intialised some value that time in b/w is that temporal dead zone.

→ When ever you try to access a variable inside a temporal dead zone it gives a reference error.
'Cannot access 'a' before initialization'

→ * We cannot do redeclaration of let. It will throw an error (this time Syntax error

→ * ~~Const~~ Const is very Strict

```
Const b
b= 1000
```

SyntaxError: Missing initializer in Const declaration.

Type Error → when you are reintializing a const variable to a var.

Syntax Error → Suppose when you are not initializing the value to const variable at the time of declaration.

Reference Error → When we try to access a variable in the temporal deadzone.

1. Const → whenever you can use const you should use it, whenever you want to put in some value which is not changed later use const.
2. Let → if not const try to use let wherever possible because let has a temporal deadzone & you will not run into ~~excepted~~ expected error.

Best way to avoid this Temporal dead zone is to always put your declarations & initialization at the top of the scope. so that as soon as your code start running it hits the initialization part first then you go to the logic.

You can also call it as we are ~~Shri~~ Shrinking the temporal Dead zone Window to zero while moving our intization at the top.

Block Scope & Shadowing

~~if (true)~~ {

~~var~~ a = 10;

{ } ← This is a block.

Block is also known as

We group multiple statement in block so that we can use it where javascript expects one statement.

```
if (true){
//
var a=10;
console.log(a)
}
```

We combine multiple statement in a block so and use it where Javascript expects a single statement.

```
{
  var a = 10;
  let b = 20;
const c = 30;
}
```

Block
b: undefined
c: undefined
a

Global
a: undefined.

Shadowing

```
var a = 100;
{
  var a = 10;
  let b = 20;
  const c = 30;
  console.log(a); ⟶ 10
}
console.log(a) ⟶ 10
```

Shadowing (var) If you have same named variable ouside the block then that variable Shadows the variable & the value will be changed or updated to that you give it in the block as in above example. This is called Shadowing. (This is because both variable are pointing to same reference

Shadowing (let)

```
let b = 100
{
  var a = 10;
  let a = 20;
  const c = 30;
  Console.log(a) ⟶ 10
  Console.log(b); ⟶ 20
  Console.log(c); ⟶ 30
} Console.log(b); ⟶ 100
```

this has the Script scope.
this as a Global scope.
this b has a blog scope
from Script scope.

~~Saho~~

Shadowing (Const)

```
Const C =100 ——→ Script scope
{
 var a= 10;
 let b= 20;
 const c = 30 ——→ block scope
 Console.log (c); ——→ 30
}
console.log (c) ——→ 100 (script scope.
```

Shadowing Concept works same for the function as well.

Illegal Shadowing

→
```
let a= 20;
{
  var a=20
}
```

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

```
let a=20
function x () {
  var a=20
}
```

Not illegal
reason being var is functional scope here in this case it is accesible only inside function.

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

reason being let is the block scope and it will not effect the outside scope
So it still not throw error