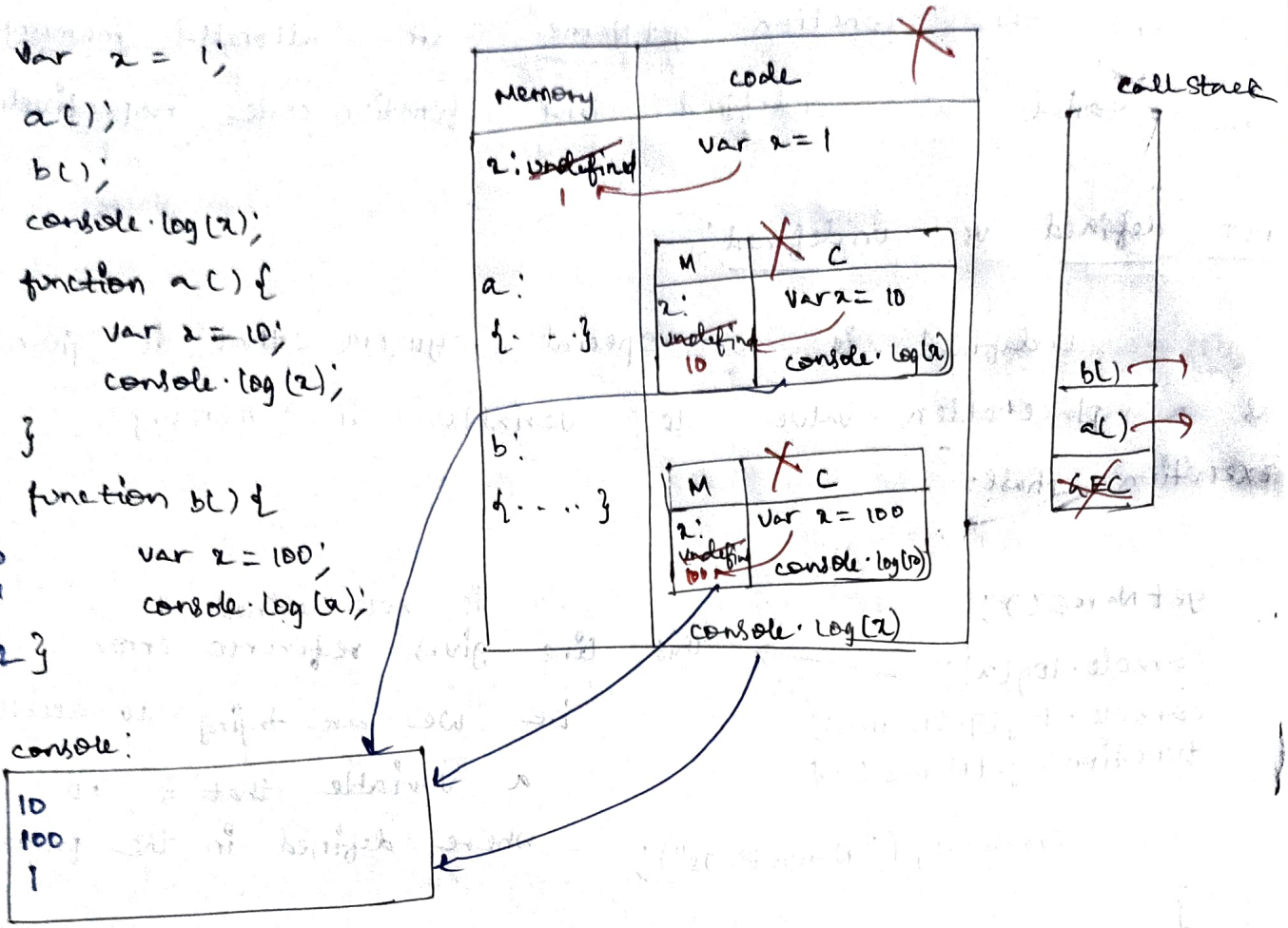
How functions work in JS:

```
1  var x = 1;
2  a();
3  b();
4  console.log(x);
5  function a() {
6      var x = 10;
7      console.log(x);
8  }
9  function b() {
10     var x = 100;
11     console.log(x);
12 }
```



→ When the program is run a GEC is created

→ At line 1, x is given as undefind before and now it changes to 1 after executing line 1.

→ Now at line 2, a() EC context is created and it is pushed onto stack.

→ In a()'s execution context, x is undefined initially and when control moves to line 6, x is changed to 100.

→ Then control goes to line 7, from the local memory of a's EC, x is taken as 10 and its logged to console on this line.

→ Now a's EC is popped off the stack.

→ The control goes back to line 2.

→ whenever JS goes to line 3, a new EC is created for b() and it goes into stack.

→ Here also x is given undefined and when line 10 is executed, x is changed to 100.

→ At line 11, x value is taken from local memory of b's EC and is given to console.

→ NOW b's EC is also popped off the stack.

→ The control goes to line 3, and nothing to do there.

→ So at line 4, x value is taken from local memory of GEC and it given to the console as 1.

→ NOW the GEC also popped off the stack.

Shortest JavaScript program:

The shortest JS program is an empty js file.

Eventhough the file is empty, a global EC is created whenever the file is run.

Along with this GEC, a global object "window" is created.

Also, a "this" keyword is also created.

Global object in case of browsers is known as "window"

At the global level

this === window

∴ this points to global object i.e window

Undefined vs not defined:

```
console.log(a);  ⟶ undefined

var a = 10;
console.log(x);  ⟶ x is not defined
console.log(a);
```

In case of undefined, a will be allocated memory when GEC is created.

But in case of x, memory will not be allocated as it not defined anywhere in the program.

JS is a loosly typed language: (Dynamically Typed)

```
var a;
console.log(a);  ⟶ undefined
a = 10;
console.log(a)   ⟶ 10
a = "Hello World";
console.log(a);  ⟶ Hello World
```

In JS, variables are not bound to a specific data type and their type can change during the execution of a program.

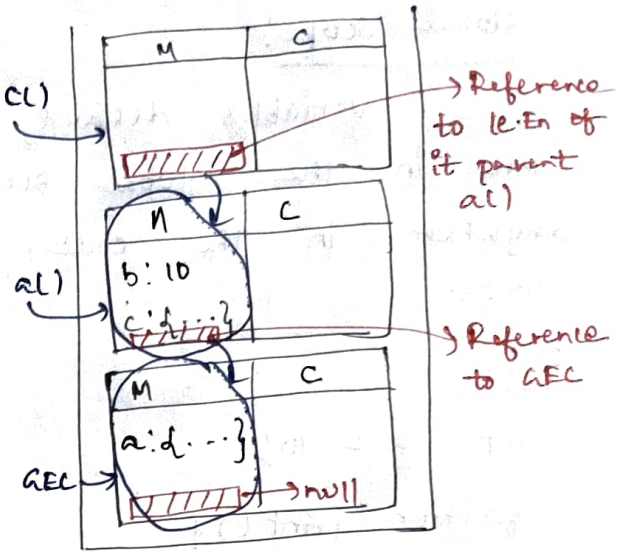
Lexical Environment:

A lexical environment is essentially a structure that holds variable bindings (variable names mapped to their values) and the reference to its parent environment, forming a chain known as scope chain.

Everytime a function is invoked, or a block of code is executed, a new lexical environment is created.

Ex:

```
1 function a() {
2     var b = 10;
3     c();
4     function c() {
5         console.log(b);
6     }
7 }
8 a();
9 console.log(b);
```



Lexical Env = local Memory + Lexical Env of the parent

↓

in a hierarchy / in a sequence

In the above example, we can say that c function is lexically sitting inside a function.

i.e c is physically present inside a.

Where the code is present

→ When JS encounters line 5, it tries to find b in local memory of c. As there is no b available inside c() JS engine goes to the reference of c which is its parent a().

→ b is found in a() and 10 is printed.

Scope:

Scope refers to a region or context where a variable or function is accessible.

Global scope:

variables declared outside of any function or block are in the global scope. These variables are accessible anywhere in the code, including inside functions and blocks.

Ex:

```
let a = 10;
function print() {
    console.log(a); → accessible here
}
print();
console.log(a);
```

Function scope:

variables declared inside a function are in function scope and are accessible within that function and not outside of it.

Ex:

```
function print() {
    let a= 10;
    console.log(a); → accessible here
}
print();
console.log(a); → error: a is not defined
```

Block Scope:

→ Introduced in ES6 with let & const

→ variables are confined to the block in which they are declared (like inside {} in loops, conditionals)

Ex:

```
if (true) {
    let a = 10;
    console.log(a); → accessible here
}
console.log(a); → error: not defined
```

Lexical Scope:

The scope of a variable is determined by its location in the code (i.e where it is written).

JS uses lexical scoping, meaning that a functions' scope is defined by its location within the nesting of function blocks at the time of writing, not during execution.

Ex:

```
function a(){
    let x = 5;
    function b(){
        console.log(x); → accessible due to lexical scoping
    }
    b();
}
a();
```

Scope chain:

When trying to access a variable, JS first looks in the local scope (function or block). If the value is not found there, it moves up to the next outer scope, continuing this process until it either finds the variable or reaches the global scope.

Let & Const in JS:

Let and const declarations are hoisted in javascript. But they are hoisted very differently than var declarations.

Infact let & const are in "temporal dead zone" for time being.

Ex:

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

Memory is allocated to a & b even before a single line of code is executed. ~~Even~~

So even before delaration and initialization, we can access b without any error. Infact we will get a special placeholder "undefined" for b. This is because of hoisting.

so in that case, a is also hoisted, we can access a in the same way. Right?

But if we run the below code

```
console.log(a); // uncaught Reference error: cannot access 'a' before initialization
let a = 10;
var b = 100;
```

Reason:

Even before a single line of code is executed, JS allocates memory to 'a'.

'a' & 'b' are in scope. (Check in chrome Dev tools)

Incase of var, b is in Global space, but incase of 'a' it is something else.

→ Memory is allocated to 'b', the var declaration and the variable b attached to the global object.

→ In case of let, 'a' is stored in a different memory space than Global. And we can't access 'a' before we put some value in it.

When JS

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

When JS executes line 2, JS engine gets value of 'a' as 10 and it is available now.

After completing line-3, b is also available as 100.

Temporal Dead Zone:

Time since when let variable is hoisted and until it is assigned some value.

The time in between is known as the temporal dead zone.

Whenever we try to access a variable in temporal dead zone, it gives reference error.

Reference Error:

If we try to access a variable, which is nowhere in the program

```
console.log(x)  // Reference Error: x is not defined
```

Ex:

```
console.log(a) // cannot access a before initialization
let a = 10;
```

Two different cases for Reference errors.

Relation of Global object & variables var, let & const:

var declarations are attached to window object

```
window.b // 100
```

let & const are not in global, so

```
window.a // undefined
```

a is treated as any other variable not in the program.

At global level, window = this

```
this.b // 100
```

Redeclaration:

```
let a = 10
let a = 100
```

```
let a = 10
var a = 100
```

Not possible

Syntax Error: Identifier a has already been declared.

```
let a = 10
c = 100;   → This is possible
```

```
let a;
a = 10;
```

Strict → const > let > var

const:

```
const b = 10;
```

↳ Declaration & Initialization on same line

Ex: const b; → Syntax Error:
b = 1000; Missing initializer in const declaration

Ex: const b = 10; → Type Error:
b = 100; Assignment to a const variable

Syntax Error:

Occurs when javascript code violates the rules of the language such as missing parentheses or using improper syntax.

```
let x = ;  // unexpected token;
```

Here javascript expects some value after the = operator

Type Error:

Occurs when an operation is performed on a value of wrong type. It means trying a method or property that doesn't exist on the type of value we are working with.

```
let x = 42;
x.toUpperCase(); // Type Error: x.toUpperCase() not a function
```

Reference Error:

Occurs when a variable or function is used before it has been declared or is out of scope.

Var, let, const:

Scope:

Var is function scoped, let & const are block scoped

Redeclaration:

can redeclare var variable multiple times in the same scope, but let & const do not.

Reassignment:

Both var and let allow re-assigning new values to the variable, while const does not allow re-assignment once value is set.

Hoisting:

All three are hoisted, but with let and const, you cannot access their variables before their declaration (temporal dead zone).

Initial value Required:

const requires an initial value when it is declared while var and let do not.

How to avoid temporal dead zone?

By pushing all the declarations, initializations to the top of the scope.

What is a block in Javascript?

A block is defined by {}.

```
{
    → perfectly valid JS code
}
```

Block is also known as compound statement. It is used to combine multiple statements into one group.

We need to group these statements together in a block, so that we can use multiple statements at a place where Javascript expects a single statement.

Ex:

```
if (true)
```

// Syntax Error: Unexpected end of input

if expects a statement here

That one statement can be anything like

```
if (true) true;
```

↳ perfectly valid one statement

But if there is a need to write multiple statements, we can only do that by grouping them together.

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

→ 'if' in itself does not have curly braces {}.

```
if (true) console.log("abc")
```

Block Scope:

Refers to the visibility and lifetime of variables defined within a specific block of code.

Variables declared within a block are not accessible outside that block. These variables exist only for the duration of block's execution.

Ex:

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

console.log(a); //10
console.log(b); // Reference Error: b is not defined
console.log(c);
```

Even before a single line of code is executed, 'b' & 'c' are allocated memory and assigned 'undefined' value in block scope (Mem other than global) because of Hoisting.

In the same way 'a' is also assigned with 'undefined' but in global Memory.

b & c are hoisted in ~~global~~ block scope while 'a' is hoisted in global scope.

∴ let & const are block scoped.

Once is finishes executing the block b and c are no longer accessible. But var can be accessed outside.

Shadowing:

Shadowing refers to a situation in programming where a variable declared in local scope (such as inside a function or block) has the same name as a variable in an outer scope (such as in parent function or global scope). The local variable "shadows" the outer variable, meaning the inner variable

this precedence, and the outer variable is not accessible within that local scope.

This shadowing behaves bit differently in case of var than with let or const.

In case of var:

```
var a = 100;
{
    var a = 10;
    let b = 20;
    const c = 30;
    console.log(a) //10
}
console.log(a) //10 ⟶ Here also a is modified to 10.
```

Here both memories of the variables 'a' are referring to the same memory space i.e global space.

This is not the case with let.

```
let b = 100;  ⟶ this 'b' will be in 'script'
{
    let b = 20;  ⟶ This 'b' will be in block scope
    console.log(b) //20
}
console.log(b) //100
        ↳ outside of block scope
```

Similar behavior is observed in 'const' case.

Illegal shadowing:

When we try to shadow a let variable using var, we will get an error.

```
let a = 10;
{
  var a = 20;
}
```
//Syntax Error: Identifier 'a' has already been declared

↘ Here var is crossing its boundaries as it is not block scoped.

```
var a = 10;
{
  let a = 20;
}
```
→ This is allowed

In case of a function:

```
let a=10;
function x(){
    var a=20;
}
```
→ This is allowed

There will be no error now as var is well inside its scope. var a is not interfering with let 'a'.

→ const:

```
const a=10;
{
  const a=20;
}
```

Lexical Block Scope:

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

Each and every block has a seperate memory. The blocks have its own lexical scope and follows a lexical scope chain pattern.

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

All the scope rules which work on functions are exactly same as in arrow functions.

Closures in JS:

Basic Example:

```
function x(){
      let a=10;
      function y(){
         console.log(a);
      }
      y(a)
   }
   x()
```

Closure in javascript is a feature where an inner function has access to variables from its outer (enclosing) function, even after the outer function has finished executing. This means the inner function "remembers" the environment in which it was created.

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

var z = x(); → x does not present after this line.

console.log(z); // prints function y; we have used y outside of its scope.

.
.
.

z() // → tries to find a, but a is no longer present.

But because of closure, it prints 7.

y function remembers where it came from.

When we return y, not only y but its closure also gets returned. and put inside z.

When we execute z somewhere in the program., it still remembers reference to a.

Uses:

- → Module Design pattern
- → Currying
- → Functions like once
- → memoize
- → maintaining state in async world
- → setTimeouts
- → Iterators

setTimeout + closures!

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

Namaste JavaScript
(after 1 sec)
1

call back function forms a closure, so it remembers reference to i, wherever it goes, it takes reference of i along with it.

→ setTimeOut takes the callback function and stores it somewhere and attaches a timer to it.

→ Meanwhile JS does not wait and proceeds to next line. and prints "Namaste Javascript".

→ Once the timer expires, setTimeout puts the function ~~agai~~ in the call stack and 1 is printed now.

First class functions!

Function Statement!

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

A named function that can be invoked later in the code.

Function Expression!

We can assign a function to a variable. Here function acts like a value.