**BLOCK IN JS\***

Block - it is compound statement. It is used to combine the multiple JavaScript statements into a one group. We use block because JavaScript aspects place of multiple statements as single statements.

let x=10

if(x==10){          ////////

    sum=20+30;            //   if(){} =>statement count as one statements

    console.log(sum);     //

}    ////////

**SCOPE IN JS\***

Scope is a certain region of a program where a variable is defined and can be accessed but beyond that region, it cannot be recognized or accessed."

In other word scope is basically defined the boundary where we can access the variable And function definition

* Block scope
* Function scope
* Global scope

=> Let and const are block scoped because we can not access the variable out side the block.

=> Var is also a function scoped because we can not access the variable out side the function block.

function sum(){

    var x=30;

}

 console.log(x)// x is not define

=> if I declare a variable globally with let , const and var keyword we can access any where in the program

Note

Global variable can access inside the local scope but local variable can not access in global scope.

**SHADOWING IN JS**  
Variable shadowing in JavaScript occurs when a variable declared within a certain scope has the same name as a variable declared in an outer scope. This causes the inner variable to "shadow" the outer one

//Case 1 =////////////////////////////////////////////////////////////////////////////////

var age=20;// OUTER SCOPE

{

HERE INNER SCOPE VARIABLE SHADOW THE OUTER SCOPE VARIABLE AND IF WE TRY TO CONSOLE AGE VARIABLE OUT SIDE THE SCOPE IT RETURN THE VALUE OF INNER VARIABLE BECAUSE VAR IS A GLOBLE SCOPED .

 var age = 40; // INNER SCOPE

 console.log(age)  // 40

}

console.log(age)// 40

//Case 2= ///////////////////////////////////////////////////////////////////////////////////

let hight=5.6;// OUTER SCOPE

{

HERE ALSO INNER SCOPE VARIABLE SHADOW THE OUTER SCOPE VARIABLE AND IF  WE TRY TO CONSOLE HEIGHT VARIABLE OUT SIDE THE SCOPE IT RETURN THE VALUE OF OUTER VARIABLE BECAUSE LET IS A BLOCK SCOPED .

 let hight= 6.3; // INNER SCOPE

 console.log(hight)  // 6.3

}

console.log(hight)// 5.6

//Case 3= /////////////////////////////////////////////////////////////////////////////////

var roll=30;

function check(){

HERE ALSO INNER SCOPE VARIABLE SHADOW THE OUTER SCOPE VARIABLE AND IF  WE TRY TO CONSOLE ROLL VARIABLE OUT SIDE THE SCOPE IT RETURN THE VALUE OF OUTER VARIABLE BECAUSE VAR IS  ALSO A FUNCTION SCOPED .

 var roll = 55;

 console.log(roll)// 55

}

check()

console.log(roll)// 30

//Case 4 = ////////////////////////////////////////////////////////////////////////////////

let x = 10; // OUTER SCOPE

function example() {

HERE ALSO INNER SCOPE VARIABLE SHADOW THE OUTER SCOPE VARIABLE AND IF  WE TRY TO CONSOLE HEIGHT VARIABLE OUT SIDE THE SCOPE IT RETURN THE VALUE OF OUTER VARIABLE BECAUSE LET IS A BLOCK SCOPED .

  let x = 20; //   INNER SCOPE, shadows the outer 'x'

  console.log(x); // Outputs 20, accessing the inner 'x'

}

example();

console.log(x); // Outputs 10, accessing the outer 'x'

**LEGAL SHADOWING AND ILLEGAL SHADOWING IN JS**

we can shadow *var* variable by *let* variable but cannot do the opposite. So, if we try to shadow *let*variable by *var* variable, it is known as **Illegal Shadowing** and it will give the error as*“variable is already defined.”*

function fun() {

var a ='Hellow'; //WE CAN SHADOW VAR VARIBLE THROUGH THE LET VARIABLE I.e LEGAL SHADOWING

let b ='Good'; //WE CAN NOT SHADOW LET VARIBLE THROUGH THE VAR VARIABLE I.e ILLEGAL SHADOWING

    if (true) {

        let a = 'Good morning'; // Legal Shadowing

        var b = 'Bad'; // Illegal Shadowing

        console.log(a); // It will print 'Good morning'

        console.log(b); // It will through error

    }

}

fun();

**TDZ (Temporal dad zone )**

Basically TDZ is a state when we declare a variable but not initialized but after some time we initialized the value of that variable . between that time of deceleration to initialization is called TDZ.

 // Declare the variable

 var age;  //////////////////////////////////

 console.log(age)                          //

 function greate(){                        //

  console.log("good morning")              // Between the time of declaration to

 }                                         // initialization is called TDZ

 // initialize the value of that variable   //

 age=24; ////////////////////////////////////

 console.log(age)

**LEXICAL SCOPE IN JS**

Lexical scoping defines the scope of a variable by the position of that variable declared in the source code.

Imagine you have a parent function and a child function defined within it. The lexical scope allows the child function to access variables declared in the parent function, but not necessarily vice versa. This concept is often described as the child function being

"lexically bound" to the scope of the parent function.

// Boundary of funOne is a lexical environment for funTwo .When funTwo call first search myAge in self scope

//if it is exist then it print otherwise it serch in lexical environment.

let myAge=40;

function funOne(){

  let myAge=20;

  console.log(myAge);

        const funTwo=()=>{

          let myAge=10;

          console.log(myAge);

        }

        funTwo();

}

funOne();

// SCOPE CHANING

// Other example ()

let myName="Anurag";// Global Scope

    function funName1(){ // Function Scope 1

       console.log(myName);

        let funName2=()=>{  // Function Scope 2

          let myName= "Ram";

           console.log(myName);

                let funName3= function(){ // Function Scope 3

                    console.log(myName);

                }

                funName3();

        }

        funName2();

    }

    funName1();

CLOSURE IN JS

a closure gives you access to an outer function's scope from an inner function. In JavaScript, closures are created every time a function is created, at function creation time.

Closures in JavaScript allow inner functions to retain access to the variables of their outer functions, even after those outer functions have finished executing.

function makeFunc() { // This is outer function

    const name = "Mozilla"; // name is a local variable created by makeFunc()

    function displayName() { // This is inner function

    // displayName() is the inner function, that forms the closure

      console.log(name);

    }

    return displayName; //makeFunc returns the displayName function

  }

  const myFunc = makeFunc();

  myFunc();

//What's happening: The code defines a function called makeFunc, which creates another

function called displayName inside it. displayName uses a variable called name from the outer function.

//When makeFunc is called and assigned to myFunc, it essentially creates a closure. This closure includes both the displayName function and the environment in which it was defined, including the variable name.

//Even after makeFunc finishes executing, myFunc still retains access to the name variable because of the closure. So when myFunc is called later, it can still access and use the name variable.