Block Scope and Function Expressions

How a function scope is different than block scope.?

A variable having a function scope has a special property that it can be defined anywhere in the function and can be accessible anywhere inside the function

if we want to access that particular variable outside the function then it is not allowed we cannot access it.

but in block scope if we are defining/declaring something inside the function they are only accessible after they are declared we cannot access them before declaration and it is not accessible outside the block.

```
function fun(){
        console.log(x)
        var x = 10;
        console.log(x)
}
fun()
```

hence x is used before declaration, and this code works because x has a function scope due to var hence x is visible through the code.

```
function fun(){
        console.log(x) // cannot access before initialization
        let x = 10;
        console.log(x)
}
fun()
```

The above code shows the following error:

`ReferenceError: Cannot access 'x' before initialization`

This is because now x will behave as a block scope and we cannot access block scope variable before initialization, The same thing will happen with

Notes About Block Scope

Block scope variables are not accessible outside the block.
Block Scope variables are not accessible before they are declared.

Temporal Dead Zone:

This is the zone before the declaration of the block scoped variable.

The time taken from un-initialization to getting initialized that particular time taken is called Temporal Dead Zone. We cannot access anything during TDZ.

The area where we want to access block scope variable before its declaration is known as Temporal Dead Zone `matlab aisa zone jahan pe variable available nahi hai`

```
function getData(){
        console.log(x)
        var x = 10
        console.log(x)
}
```

In this above example I am trying to access the value of x before initialization, here I will not get any error infact I will get undefined

`kyun ki ek toh hame usko declare kiya hai using var and dusri baat jo hai usko hame ek function ke andar declare kiya hai so usko milega function scope and function scope matlab ham usko function ke andar kahin bhi access kar sakhte hai aggar` `declaration se pehle karenge toh undefined milega aur aggar baad mein karenge matlab declartion ke baad toh hame milega actual value of the variable`

```
function getData(){
        console.log(x)
        const x = 10
        console.log(x)
}
```

In this Example I will get an error

```
ReferenceError: Cannot access 'x' before initialization
```

`Yahan pe error esliye aa raha hai kyun ki ek toh woh block scope ban chuka hai hamne usko const se jo declare kiya hua hai and jo block scope hota hai aap usko access nahi kar sakhte uske declaration se pehle kyun ki ek point yeah bhi hai ki woh Temporal Dead Zone mein bhi hai and jo value hota hai Temporal Dead Zone mein aap usko access nahi kar sakhte`

More Facts on let:

let doesn't allow to redeclaration of variables once it is declare its done you cannot declare it again , But it will allow you to reinitialize

```
let a = 10
console.log(a)

let a = 30;
console.log(a )
```

`SyntaxError: Identifier 'x' has already been declared`

```
let userName ="deepak"
userName = "deepak_kumar"

// I have reinitialize value into it.
```

Hoisting:

Hoisting is the process of accessing something even before initialization. In most of the programming languages it will be a error we cannot access anything even before initializing
we cannot access any variables even before we have initialized any value into it.

Hosting works very differently in case of `var, let` and `const` in case of `var` we will not get any error infact we will get a special value called undefined.

Sanket Sir Hoisting:

Hosting is a consequences of the scoping mechanism of JS. Because of the fact that JS executed in two phases , a lot of variables are already known during the phase 1 and then they are accessed in Phase 2. So its looks like a lot of people think that JS knows about a few variables before their declaration. And indeed it is true during phase 1 all the formal declarations are read, So JS knows about all the variables before execution phase. And this mechanism of knowing variables before declaration is termed as Hoisting.

This word is not an actual word present in the official ECMAScript it is the JS Community who came up with a word called hoisting.

```

console.log(x)  #undefined
console.log(getName) #this will give us the whole function

var x = 7;
function getName(){
        console.log('name')
}
```

So in case of `x` we will get undefined but in case of `getName()` we will get the whole function

```
console.log(getName) #undefined
getName() #getName is not a function
var getName = () => {
        console.log('name')
}
```

Obviously I will get undefined because `getName` is being treated as a normal variable here in the above example I have tried with an arrow function so arrow function will be treated as a variable only because its actually same how we are creating a variable, Instead of `var` i can also use `let` and `const.`

and if I am using `let` and `const` then also I will get same result the way I used to get while declaring a variable with `let` and `const` and trying to access even before initialization.

Never say that `let` and `const` are not hois ted.?

Yes its is true because `let` and `const` are actually hoisted but the way of hoisting is actually different from `var`

if you have declare a variable using var in phase one during memory allocation you will see that the variable which you have declared using var will be assigned with a special value called undefined and you can find that one (var variable) inside the global scope in sort it is attatched to the global object.

But when you will declare something using const and let you can find those variable inside the script that means these are also allocated memory but they are stored in a different memory space they are not in the global they are stored in the separate memory space and we cannot access this let and const declarations before we have put in some value into it.

```
        console.log(userName) ReferenceError
        console.log(address) undefined

        const userName = "deepak_kumar"
        let gmail = "deepak@gmail.com",
        var address = "Odisha"
```

```
Uncaught ReferenceError: Cannot access 'userName' before initialization
```

If I try to access something inside the Temporal Dead Zone(TDZ) I will get an
Reference Error .
I can access the value I have declared with `let` and `const` after its initialization
The TDZ is only for `let` and `const` not for `var` .

```
        console.log(window.address) #Odisha
        console.log(window.userName) #undefined
```

Why I am getting undefined for `window.userName` because it is not available in the global it is available in another place inside the script and anything which is inside the script we cannot access those values.

`Let` and `Const` are actually strict then `var` .
`Var` allow us to redeclare and we can initialize a new value into it.

```
        var userName = "deepak"
        var userName = "deepak_kumar"
```

But `Let` and `Const` doesn't allow us to redeclare and reinitialize we can reinitialize a new value into `Let` but `Const` doesn't allow us to do both redeclare and reinitialize.

```
        let userName = "deepak"
        userName = "deepak_kumar" # this allowed

        let location = "Odisha"
        let location = "Cuttack" # not allowed we will get
         //Uncaught SyntaxError: identifier location has already been declared

        const userName; # not allowed we have to initalize some value into it.

        const userName = "deepak"
        const userName = "deepak_kumar"  #not allowed

        userName = "rahul" #not allowed
```

`TypeError: Assignment to a const variable`

`SyntaxError: Identifier "xyz" is already been declared`
we will get syntax error when working with let where redeclaration is not required look at the above example with `let` for more clarity

`ReferenceError:` When we are trying to access something inside the memory space but we cannot access it.

For example if I will try to access something which I have declared with let , I will try to access it even before initialization then I will get `Reference Error`

Important Points about Let and Const and Var, Block Scope and Function Scope

. Let and Const are known are Block Scope Variable.
. These are only declared within the block and are only accessible inside that Block.
. Where as we can access var outside the block as it is present inside the Global Scope
. If we try to access the Block Scope Variables outside the block then we will get an error as **Reference Error : xyz variable is not defined**
. In a block scope(let and const) the variables are declared within the block cannot access outside the block and even cannot access it

before declaration.
. If we declare something inside the function Scope that variable can be accessible anywhere inside the function
But in Block Scope the variables are accessible once they are declared we cannot access them before they are declared

Function Expression

The function expression and function declaration are same syntax but the main difference between two of them is in function expression the function name is omitted to create a Anonymous function (a function which doesn't have any name).

but in function declaration function keyword is used.

Function Declaration:

```
function displayName(){
        console.log('name')
}
```

Function Expression:

```
let displayName = function(){
        console.log(name)
}
```

The function expression in JavaScript is not hoisted unlike function declaration
even though the variable name (`notHoisted`) is hoisted but the definition isn't

```
console.log(notHoisted); // undefined
// Even though the variable name is hoisted,
// the definition isn't. so it's undefined.
```

```
notHoisted(); // TypeError: notHoisted is not a function

var notHoisted = function () {
  console.log("bar");
};
```

In JS functions are used as a first class citizens, why.?
. we can return a function from a function
. we can pass a function as an argument to another
. we can also store a function in a variable.

Let's focus more on storing a function in a variable:

```
const myfunc = function(){
        //....
}
```

here we are creating a function but the first valid token is not the function keyword, hence we call this type of instruction as a `function expression.`

More Ways to create a function Expression:

Example One

```
const func = function myfun(){ // named function expression
        //...
}
```

Example Two

```
const func = function(){ //anonymous function expression
//...
}
```

Example Three

```
const func = () => { // arrow function expression
//..
}
```

Detail Study About Function Expression:

Named Function Expression

A function expression with a name attached to it is called function expression.

```
const myFunc = function func(){
        //....
}
```

so here the name `func` is the name of the named function expression and this function is stored in the variable called `myFunc`

Anonymous Function Expression:

A function expression without a name.

```
const myFunc = function(){
//...
}
```

Here in this example the function expression has no name and it stored in a variable called `myFunc` so both are similar the only difference is in **Anonymous Function** there is no name is being used but in **Named Function Expression** name is used `func`

Should we use named function expression or anonymous function expression.?

. Anonymous function are hard to use in the recursion whereas a named function expression can be easily integrated in a recursive environment to understand this let's take an example of array.map.
. map is an inbuilt function for arrays in JS. It takes an argument which is expected to be a function. The function which we pass as an argument is automatically internally called by map function. The function that we pass is expected to have an parameter
and inside this parameter the map function automatically pass all the elements of an array one by one.

```
let arrayOne = [2,3,4,5,6]
const returnedValue = arrayOne.map(function(element){
        return element*2
})

console.log(returnedValue) // [2,6,8,10,16]
function expression is passed an an argument to map, map automatically internally called this function
```

!Map Function Important Note

Map mein jab ham return karenge it will give us an new Array, arrayOne mein koi changes nahi hoga woh waisa ka waisa hi rahega, and return mein hame jo milega woh ek new array milega

How to implement own map function.

```
function customMap(arrayOne, fn){
        const result = []
        for(let i=0; i<arrayOne.length; i++){
        result.push(fn(arrayOne[i]))
        }
        return result
}

const data = customMap(arrayOne, function fn(element){
        return element*2
        })
console.log(data)
```

Where we can used these function expressions.?

Function expression can be used to pass a function as an argument to another function

```
const myFunc = function(fn){
        fn()
}
myFunc(function DisplayName(){
        console.log('hello Deepak')
})
```

IIFE (Immediately invoked function expression)

A function which is immediately called as soon as it is defined is called IIFE.
An IIFE is written inside a parenthesis so to define in IIFE we have to wrap a function inside the parenthesis by putting a subsequent pair of parenthesis and passing arguments into it.

```
        (function square(x){
                return x*x
        })(10)

// (10) => susequent parenthesis with arguments
```

IIFE ko ek bar access karne ke baad dubara access nahi kar sakhta because its gets wiped out of the memory once the execution is done.

IIFE can be very useful to avoid naming conflict because they don't conflict with those variables who are in outer scope.

IIFE can be useful for some temporary logic as well.