Miracle Pg.No: Date: / /20

Episode - 8

* let and const in JS.

→ let and const declarations are hoisted. but its memory is allocated at other place than window (global scope) which cannot be accessed before initialization.

→ Temporal Dead Zone (TDZ) is a block where a variable is inaccessible until the moment the computer initializes it with a value.

→ e.g.

```
let a = 10
const b = 100
var c = 1000
```

⇒ In console:
```
window. a ⇒ undefined
window. b ⇒ undefined
window. c ⇒ 1000
window. x ⇒ undefined
```

window. variable and this. variable are same

→ Thus, window. variable OR this. variable will return undefined if we use let and const.

→ We cannot redeclare the same variable with let and const. (even with using var the next time)

e.g.
```
let a = 10      ✗ SyntaxError: Identifier
let a = 100       a has already been
                  declared.

let a = 10      ✗
var a = 100
```

Same with const (in place of let)

```
const a = 10    ✗ TypeError: Assignment
a = 1000          to constant variable
```

→ const variable declaration and initialisation must be done on the same line.

→
```
const b = 10  ✓

const b   ✗ SyntaxError: Missing initializer
b = 10        in const declaration

let b   ✓
b = 10
```

→ Use const whenever possible followed by let, Use var as little as possible. (only if you have to) It helps avoiding error.

→ const ⇒ Most strict
let ⇒ Strict
var ⇒ least strict

→ Types of errors :

[1] Reference Error : When JS engine tries to find out a specific variable inside the memory space and you cannot access it.

e.g.
```
console.log(a)
let a=10
```
× Reference Error : Cannot access 'a' before initialization

```
console.log(a)
var a=10
```
⇒ undefined in output (console)

~~[2]~~
```
console.log(x)
let a=10
```
× Reference Error : x is not defined

[2] Type Error : When we change the value of a type that is not supposed to be changed i.e. const

e.g.
```
const a=10
a=100
```
× Type Error : Assignment to constant variable

[3] Syntax Error : When proper syntax (way of writing a statement) is not used.

e.g. Both examples in previous page

Pg No:
Date: / /20

→ Initializing variables at the top of the program is good, it helps to shrink TDZ (temporal dead zone) to zero.

Episode - 9

* Block Scope and Shadowing in JS

→ Code inside curly bracket is called block. {....}

→ Multiple statements are grouped inside a block so it can be written where JS expects single statements like in if, else, loop, function etc.

→ Block variables are stored inside separate memory than global. They are stored in block.

→ let and const are ~~blocked~~ blocked scoped.

→ So, a variable declared in a block with let is only available for use within that block (same for const)

→ e.g.

```
{
    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)
```

→
```
10
20
30
10
Reference Error: b is not defined (program
 stops here)
```

→ Shadowing in JS refers to a situation where a variable declared within a certain scope has the same name as a variable declared in an outer scope.

→ When this happens, the inner variable "shadows" or takes precedence over the outer one.

→ e.g.

Code:

```
let x=10 → This is in another scope (Script)

{                 This has block scope
   let x=5 ————→
   console.log(x)
}

console.log(x)
```

⟹ 5
10

→ Illegal Shadowing:

If you create a variable in a global scope with the let keyword and another variable with a var keyword in a block scope but with the exact same name, it will throw an error.

e.g
```
let a=20
{
   var a= 10
}
```

SyntaxError: Identifier 'a' has already been declared.

If I do, function x() { var a=10}, then it is fine. Because it is inside function scope (in its boundaries)

→ Block is also known as Compound Statement.

→ When the execution of block is completed, it is deleted from the scope.

→ Shadowing:

e.g.
```
var a = 10          → global scope
{
    var a = 100     → This shadowed the outer var a.
    console.log(a)  → global scope
}

console.log(a)
```

⟹ 100 Why 100? Because they both are
100 pointing to the same memory location in global scope (unlike let and const)

i.e. First 10 is assigned. Then 100. 100 is then permenantly stored in memory as a: 100.

→ Block scope follows lexical scope.

e.g.
```
const a = 10
{
    const a = 100
    {
        console.log(a)
      }
}
```

⟹ 100 (If not found inside its lexical env, it tries searching in parent and so on...)

e.g.

```
const a=10  ——→ In Script
{
    const a= 20 ——→ In Block  ⌉ different
    {                         ⌋ blocks
        const a=30 ——→ In Block
    }
}
console.log(a)
```

i.e.
→ Script
a: 10
→ Block
a: 20
→ Block
a: 30

⟹ 10