Page No.
Date / /

```
let x = -0
console.log(x < 0);          // false
```

In JavaScript, the value -0 is considered to be equivalent to 0 by the comparision operator, including the less than operator ("<").

when we compare -0 with 0, the comparision will return false. This is because -0 is considered to be equal to 0 by the JavaScript interpreter, so the comparision "-0 < 0" is equivalent to "0 < 0" which is false.

We can use the **Object.is()** method to check if -0 is different than 0

```
console.log(Object.is(x, -0));   // true
console.log(Object.is(x, 0));    // false
```

* **Math.sign(x)**
  - if x is Nan, the result is Nan
  - if x is -0, the result is -0
  - if x is +0, the result is +0
  - if x is negative and not -0, the result is -1
  - if x is positive and not +0, the result is +1

Boxing Concept

↳ process of converting primitive values to objects

* **Why 1.toString() is thrown an error.**

In JavaScript, the "." operator is used to access properties and methods of an object. When we write "1.toString(), the javascript interpreter is trying to access the toString() property or method of the object 1. However, the number 1 is not an object and it doesn't have any property or methods, so interpreter will throw a TypeError

In JavaScript, the number 1 is a primitive value, and the toString() method is a method of the Number object. So, we have to convert the primitive number to Number object to call the toString method

```
Console.log((1).toString())          // "1"
console.log(Number(1).toString())    // "1"
```

* **Scope:** → visibility → variables, functions

A scope refers to the area of a code in which a variable or function is accessible or visible.

variables & functions.

↳ Everything inside our code, will be used in one of the following 2 ways →

1. Either it will be getting some value assigned to it
2. Or some value would be retrieved from it.

- JavaScript is not purely interpreted or purely compiled language (not)

*→ JavaScrip code is executed in two phases.

1. Parsing Phase → scope resolution
   Execution code Phase.

Date / /

* Types of scopes
1) Global scope
2) Function scope
3) Block scope.

* Var :—
→ function scope
→ globally scope
→ supports 'hoisting'
{ no, block scope

* Let :-
→ let declaration declares a block-scoped variable.

* autoglobals :-
In JS, 'autoglobals' refers to variables that are automatically assigned to the global scope, even if they are declared inside a function or block of code.
Examples, window, document and console.

* Strict mode :-
Strict mode is a restricted mode in which we run javascript.
Strict mode helpful in some case such where we use autoglobals variable, because autoglobals are restricted in strict mode.

Date / /

5..toString() or (5).toString() → boxing

5.toString() → give syntax error

"Console.log()" is not a part of JavaScript. It is a part of web browsers

→ There are two types of scopes that dif languages supports,
1) lexical scoping → refers to compilers
2) Dynamics scoping

• lexical scoping :— It actually refers to compilers that the scoping of variable is determine during compile time

When the parser actually reads your whole code then the scope of every variable, function is determine before the execution phase altogether.

whereas in dynamic scoping, during runtime we actually determine the scope.

languages like bash script, dynamic scoping exist.

* Temporal Dead zone :—

The region before the declaration of a variable which is having a block scope made by let is actually called as Temporal Dead zone.

It means we cannot access the variable which is having a block scope before it has been declared.

```
{
    // wherever we have curly braces it
    // means it is a block scope
}
```

- **function scope :-**

function scope refers to the portion of the code where a function and its variables are accessible, usually within the function itself. Variable defined within a function scope are not accessible outside of that function.

- **Block scope :-**

It refers to the portion of the code where a variable defined within a block of code, such as a loop or an if statement, is accessible. In Javascript, block scope is introduced with the 'let' and 'const' keyword. These variable are not accessible outside the block they are defined in.

Variables defined with the 'var' keyword are function scoped, meaning they are accessible throughout the entire function, regardless of the block they are defined in.

var ⟶ function scope

let ⟶ block scope

* **Temporal Dead Zone :-**

It is a feature that refers to a specific area of code where a variable declared with 'let' or 'const' cannot be accessed until it has been fully initialized.

When a variable is declared with 'let' or 'const' in Javascript, the variable is assigned a value of 'undefined' before the initialization statement is executed. However, the variable cannot be accessed or used before it has been fully initialized.

This is the temporal Dead Zone. Attempting to access or use the variable before it has been initialized will result in a ReferenceError.

→ It is important to note that this behavior is specific to 'let' and 'const' variables, and does not apply to variables declared with 'var'. Variables declared with 'var' are hoisted to the top of the scope and are accessible before they are initialized, but they will have the value 'undefined' until they are initialized.

→ Var :- it allows us re declaration

E.g. // var x = 10;

// var x = 20; // re-declaration, no error

→ let :- it doesn't allows re-declaration

Eg. // let x = 10

let x = 20 // error (in browser)

* **Const** :—

In Javascript, 'const' is a keyword used to declare a constant variable. A constant variable is a variable whose value can not be reassigned. Once a variable is declared with 'const', its value can not be changed.

Example :—

```
const PI = 3.14;
console.log(PI);        // 3.14
PI = 3.15               // this will throws an error
```

The 'let' keyword is also used to declare variables in Javascript, but unlike 'const', variables declared with 'let' can be reassigned.

Example :—

```
let x = 5
console.log(x);   // 5
x = 6;
console.log(x);   // 6
```

* **Whether Temporal Dead zone is exist for const or not.**

Yes, Temporal dead zone, exist for variables declared with the 'const' keyword in Javascript.

When a variable is declared with 'const', the variable is assigned a value of 'undefined' before the initilization statement is executed. However, the variable can not be accessed or used before

it has fully initiliazed. This is the Temporal dead zone. Attempting to access or use the variable before it has initialized will result in a Reference Error.

Example :-

```
const PI ; // Reference Error : 'PI' is not initialized
console.log (PI);
```

```
const PI = 3.14 ;
console.log (PI);   // 3.14
```

* Function expression :-

In Javascript, a function expression is a way to create a function and assign it to a variable. A function expression can be anonymous (without a name) or named.

Example :-

```
// Anonymous function expression //

const add = function (a, b) {
    return a+b;
}
console.log (add (1, 2));   // 3
```

Example

Name function expression

```
const multiply = function multi(a, b) {
    return a * b;
};
console.log(multi(3, 4));    // 12
```

→ It's important to note that function expressions in javascript are not hoisted like function declarations, so we cannot call a function expression before it is defined.

→ Function expressions are different from function declarations in that function declarations are hoisted, that means that the interpreter will move function declarations to the top of the scope, which allows us to call the function before it is defined in the code.

* Concept:-

```
const f = function fun() {
    console.log("Hi, Are you doing fun");
};
f();
fun();
```

Function expression gets the scope of their corresponding variable in which they are assigned.

```
const f = function fun() {
    console.log("How much are you funny");
}

f();
fun();
```

This above function is only accessible by the 'f' variable. It is not accessible outside 'f'. So it is going to be so bound to the scope of 'f' only. It is technically having tight boundation with 'f'. Now we will only talk about the scope of 'f' rather than scope of function 'func()' because 'fun()' is just only accessible inside 'f'. It means 'fun()' is only accessible by a 'f' with 'f' fun() can't exist.