Scoping

Scope determines the accessibility (visibility) of variables.

JavaScript has 3 types of scope:

- Block scope
- Function scope
- Global scope

Block Scope

`let` and `const` provide **Block Scope** in JavaScript.Variables declared inside a { } block cannot be accessed from outside the block

```
{
  let x = 2;
}
// x can NOT be used here
```

Variables declared with the `var` keyword can NOT have block scope.

Variables declared inside a { } block can be accessed from outside the block.

```
{
  var x = 2;
}
// x CAN be used here
```

Local Scope

Variables declared within a JavaScript function, become **LOCAL** to the function.

```
// code here can NOT use carName

function myFunction() {
  let carName = "Volvo";
  // code here CAN use carName
}

// code here can NOT use carName
```

Since local variables are only recognized inside their functions, variables with the same name can be used in different functions.

Local variables are created when a function starts, and deleted when the function is completed.

Function Scope

JavaScript has function scope: Each function creates a new scope.

Variables defined inside a function are not accessible (visible) from outside the function.

Variables declared with `var`, `let` and `const` are quite similar when declared inside a function.

They all have **Function Scope**:

```
function myFunction() {
  var carName = "Volvo";   // Function Scope
  let carName1 = "Volvo";   // Function Scope
  const carName2 = "Volvo";   // Function Scope
}
```

Global JavaScript Variables

A variable declared outside a function, becomes **GLOBAL**.

```
let carName = "Volvo";
// code here can use carName

function myFunction() {
// code here can also use carName
}
```

Automatically Global

If you assign a value to a variable that has not been declared, it will automatically become a **GLOBAL** variable.

This code example will declare a global variable `carName`, even if the value is assigned inside a function.

```
myFunction();

// code here can use carName

function myFunction() {
```

```
  carName = "Volvo";
}
```

Global Variables in HTML

With JavaScript, the global scope is the JavaScript environment.

In HTML, the global scope is the window object.

Global variables defined with the `var` keyword belong to the window object:

```
var carName = "Volvo";
// code here can use window.carName
```

Global variables defined with the `let` keyword do not belong to the window object:

```
let carName = "Volvo";
// code here can not use window.carName
```

Variable Shadowing in JavaScript

when a variable is declared in a certain scope having the same name defined on its outer scope and when we call the variable from the inner scope, the value assigned to the variable in the inner scope is the value that will be stored in the variable in the memory space. This is known as **Shadowing or Variable Shadowing**. In JavaScript, the introduction of *let* and *const* in ECMAScript 6 along with block scoping allows variable shadowing.

```
function func() {
    let a = 'Geeks';

    if (true) {
        let a = 'GeeksforGeeks';  // New value assigned
        console.log(a);
    }

    console.log(a);
}
func();
```

Illegal Shadowing:

while shadowing a variable, it should not cross the boundary of the scope, i.e. 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 func() {
    var a = 'Geeks';
    let b = 'Geeks';

    if (true) {
        let a = 'GeeksforGeeks'; // Legal Shadowing
        var b = 'Geeks'; // Illegal Shadowing
        console.log(a); // It will print 'GeeksforGeeks'
        console.log(b); // It will print error
    }
}
func();

output: Identifier 'b' has already been declared
```

Declaration

var vs let vs const

```
var a;
var a;

it is completely fine and didn't give an error
```

we can’t redeclare a variable using let and const

```
let a;
let a;

Uncaught SystaxError: Identifier 'a' has already been declared

const a;
const a;

Uncaught SystaxError: Missing initializer in const declaration
```

But if we have something like this then this is completely fine which is discussed earlier

```
let a;
{
   let a;
}
```

Declaration without initialization

For var and let we can declare a variable without initialization. But for const, the variable has to initialize with value. otherwise, it will give an error

```
var a; //fine
let a; //fine
const a; //Uncaught SystaxError: Missing initializer in const declaration
const a = 5; //fine
```

Re-Initialization

```
var a = 5;
a = 6; //fine
let a = 5;
a = 6; //fine

const a = 5;
a = 6 ; //Uncaught TypeError: Assignment to constant variable
```

Hosting

During the creation phase javascript engine moves variable and function declarations to the top of the code and this is known as hosting

```
actual code:
console.log(count)
var count = 1;
output: undefined

how juavascript looks at this code:

var count;
console.log(count);
count = 1;
output: undefined
```

Variables defined with `let` and `const` are hoisted to the top of the block, but not *initialized*.

Meaning: The block of code is aware of the variable, but it cannot be used until it has been declared.

Using a `let` variable before it is declared will result in a `ReferenceError`.

The variable is in a "temporal dead zone" (it the term to describe the state where variable are in the scope but they are not yet declared) from the start of the block until it is declared:

```
function abc(){
 console.log(a,b,c);
 // Uncaught ReferenceError: can't access lexical declaration 'c' before initialization (in console)
 // Uncaught ReferenceError: Cannot access 'b' before initialization (in console)
 // for a : undefined (in console)
 // both b and c will show undefined in local scope
 const c = 30;
 let b = 20;
 var a = 10;

}
```

Note: JavaScript only hoists declarations, not initializations.