1.Temporal dead zone

“Let” and “Const” have two broad differences from “Var”

1. They are block scoped.
2. Accessing a var before it is declared has the result undefined; accessing “let” or ”Const” before it is declared throws “Reference Error”.

For Example :

Console.log(aVar); // undefined

Console.log(aLet); // causes ReferenceError : Cannot access ‘aLet’ before initialization

Var aVar = 1;

Let aLet = 2;

Result:

Undefined

Error:{

“message”: “Uncaught ReferenceError: Cannot access ‘aLet’ before initialization ”,

“filename”: “https: //stacksnippets.net/js”

“lineno”: 13,

“colno”: 13

}

It appears from these examples that let declarations ( and const, which works the same way) may not be hoisted , since aLet doesn’t appear to exist before it’s assigned a value.

Exam That is not the case, however – let and const are hoisted (like var,class and function ), but there is a period between entering scope and being declared where they cannot be accessed. This period is the Temporal Dead Zone (TDZ).

2. Shadowing in Js

Block Scoping : To understand shadowing in JavaScript, we need to be clear with the scope first. In computer programming languages, Scope is a certain section/region of the program where a defined variable can have its existence and can be recognized, beyond that it can’t be accessed. In JavaScript, a Block is a compound statement that is defined by curly braces {} and used to combine multiple statements into one statement where JavaScript expects only one statement. And all the variables and functions that can be accessed inside a block are said to be inside that block scope, hence called Block scoped.

For example, *let* and *const*variables are stored in separate memory space, so it is called block-scoped but *var* variables can be accessed outside the block as it is stored in the Global object memory space, hence it is called Global scoped.

Shadowing: Now, when a variable is declared in a certain scope having the same name defined on its outer scope and when 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.

Example :

Function func() {

Let a =’Geeks’ ;

If (true) {

Let a = ‘GeeksforGeeks’; // New value assigned

}

Console.log(a);

Fun( );

Output:

GeeksforGeeks

Geeks

3.IIFY stands “ Immediate Invoked Function Expression “ & What is an IIFY ?

Immediately Invoked Function Expression (IIFE) is one of the most popular design patterns in JavaScript. It pronounces like iify. IIFE has been used since long by JavaScript community but it had misleading term "self-executing anonymous function"Ben Alemen” give it appropriate name Immediate Invoked Function Expression “.

As a know that a function in JavaScript creates the local scope. So,we can define variables and function inside a function which cannot be access outside of that function. However, sometime accidently pollute the global variables or functions by unknowingly giving same name to variables & functions as global variable & function names. For example, there are multiple .js files in our application written by multiple developers over a period of time. Single JavaScript file includes many functions and so these multiple .js files will result in large number of functions. There is a good chance of having same name of function exists in different .js files written by multiple developer and if these files included in a single web page then it will pollute the global scope by having two or more function or variables with the same name. Consider following example of two different JavaScript file included in single page.

Consider the following example of MyScript1.js and MyScript2.js with same variable & function name.

Ex:1

var userName = "Bill";

function display(name)

{

alert("MyScript1.js: " + name);

}

display(userName);

EX:2

var userName = "Steve";

function display(name)

{

alert("MyScript2.js: " + name);

}

display(userName);

Now, we included these JS files in our web page in the sense then it will,

Ex: Script tag in <head>

<!DOCTYPE html>

<html>

<head>

<meta name="viewport" content="width=device-width" />

<title>JavaScript Demo</title>

<script src="/MyScript1.js"></<script>

<script src="/MyScript2.js"></<script>

</head>

<body>

<h1> IIFE Demo</h1>

</body>

</html>

If we run above example, will find that every time it call display() function in MyScript2.js because MyScript2.js included after MyScript1.js in a web page. So JavaScript considers last definition of a function if two functions have the same name.

IEFE solves this problem by having its own scope and restricting functions and variables to become global. The functions and variables declare inside IIFE will not pollute global scope even they have same name as global variables & functions.

As name suggest, IIFE is a function expression that automatically invokes after completion of the definition. The parenthesis () plays important role in IIFE pattern. In JavaScript, parenthesis cannot contain statements; it can only contain an expression.

Example: Parenthesis ()

(var foo = 10 > 9); // syntax error

(var foo = "foo", bar = "bar"); // syntax error

(10 > 9); // valid

(alert("Hi")); // valid

First of all, define a function expression.

Example: IIFE

var myIIFE = function () {

// js code here

};

Now, wrap it with parenthesis. However, parenthesis does not allow declaration. So we are going to remove declaration part and we need to write anonymous function as below.

Example: IIFE

(function () {

// js code here

});

Now, use () operator to call this anonymous function immediately after completion of its definition.

Example: IIFE

(function () {

// js code here

})();

So, the above is called IIFE. We can write all the functions and variables inside IIFE without worrying about polluting the global scope or conflict with other's JavaScript code which have functions or variables with same name.

To solve the our above problem, wrap all the code in MyScript1.js & MyScript2.js file in IIFE as shown below.

Example: IIFE

(function () {

var userName = "Steve";

function display(name)

{

alert("MyScript2.js: " + name);

}

display(userName);

})();

So, even if MyScript1.js & MyScript2.js file includes functions and variables with the same name, they won't conflict with each other and pollute the global scope.

Also, we can pass arguments in IIFE as shown below.

Example: IIFE

var userName = "Bill";

(function (name) {

function display(name)

{

alert("MyScript2.js: " + name);

}

display(name);

})(userName);